Enhancing Path Efficiency: Innovations in the Artificial Feeding Birds Algorithm for Solving TSP and SOP Problems

Tim Cares
Fakultät IV
Hochschule Hannover
Hanover, Germany
tim.cares@stud.hs-hannover.de

Pit Simon Hüne
Fakultät IV
Hochschule Hannover
Hanover, Germany
pit-simon.huene@stud.hs-hannover.de

***Abstract*— The Traveling Salesman Problem (TSP) and the Sequential Ordering Problem (SOP) are well-known NP-hard problems in combinatorial optimization. In recent years, nature-inspired metaheuristics have become a popular method to tackle large TSP configurations, one of which is the Artificial Feeding Birds (AFB) algorithm. In this paper, we explore possible improvements of the AFB algorithm on TSP and SOP, and benchmark them on well-known problems from TSPLIB. Since our changes rely on newly introduced hyperparameters, we propose to use the AFB algorithm itself to optimize the hyperparameters when solving the TSP. Finally, we discuss possible explanations for the observed results and improvements.**

I. Introduction

In 2018, [1] introduced the Artificial Feeding Birds algorithm as a simple metaheuristic optimization scheme that can be easily adapted to new problems. In this paper, we explore possible algorithmic improvements when applying the AFB to the traveling salesman problem.

In Section II we define the Traveling Salesman Problem and in Section III we describe the basic Artificial Feeding Birds algorithm as defined in [1]. In Section IV we describe the methodology used to evaluate our adjustments of the algorithm and in Section V we iteratively improve the algorithm, describing the changes step by step and measuring their impact. In Section VI we apply the AFB algorithm to the Sequential Ordering Problem. In Section VII we analyze and interpret our results.

II. Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is defined as follows: Given a set of cities and distances between them, find the shortest route that visits each city exactly once and returns to the starting city. One may be fooled into thinking that this problem is simple, but it is one of the most intensely investigated problems in computational mathematics [2] and belongs to the class of NP-complete problems [3]. A brute-force approach would have a running time of $O(n!)$, which becomes infeasible even for small n (brute-forcing a TSP with 20 cities at 4 billion solutions per second would take about 20 years).

III. Basic AFB Algorithm

Artificial Feeding Birds (AFB) is a metaheuristic optimization algorithm. Unlike most nature-inspired algorithms, AFB is not inspired by rare behaviors exhibited only by a particular species, but by the feeding behavior common to all birds. The hope is that common behaviors that have been evolutionarily proven in a variety of natural environments will be equally efficient in solving complex optimization problems [1].

The AFB algorithm models a flock of birds, each bird representing a solution to the problem. In each iteration, every bird performs one of four moves:

Walk The bird performs a local search in the neighborhood of its current solution.

Random Fly The bird flies to a random location in the search space.

Memory The bird returns to the best solution it has found so far.

Join Other The bird joins another bird and adopts its solution.

Which move a bird performs is determined randomly [1].

Additionally, there is a difference between big and small birds. The difference being that small birds do not perform the join move because they are afraid of other birds. This prevents the flock from converging too quickly to a single local optimum [1].

IV. Methodology

We structure our experiments incrementally. Each improvement in Section V, even if not explicitly stated, builds on the adjustments made prior. We verify our improvements on a set of selected TSP problems from TSPLIB, namely eil101, pa561, pr1002, u2156, and pr2392, ranging from 101 to 2392 cities.

For each experiment, we run the problems 10 times. This accounts for the randomness of the initialization and reduces noise. We then record the *Mean Error* over the 50 trials, with respect to the optimal solutions, in order to compare the performance of different experiments. We also record the *Mean Runtime* in second, again, over the 50 trials.

All experiments are executed on a MacBook Pro (M1 Pro) 2021.

V. Improvements

A. Considerations about the number of birds

When we study the effects of how different numbers of birds change the solution of different tours, we learn that fewer birds seem to yield better solutions. This may seem counterintuitive at first, but there is an intuitive explanation:

The basic algorithm defines each evaluation of the length of a tour (represented by an agent) as an iteration. Although the number of iterations are not evenly distributed between the birds, as the algorithm only evaluates a tour upon the moves "fly" and "walk", and whether a bird executes those is determined randomly [1], on average, all birds will use an equal amount of iterations. Therefore, if we increase the number of birds while maintaining the iterations, we simultaneously decrease the number of iterations *per bird*. Since a decreased number of iterations per birds means each bird has less overall moves to find a good solution, there will be a less pronounced exploitation of the search space, which in turn leads to worse results.

One could avoid this by simply increasing the number of iterations, but this inevitably leads to longer running times. Instead, we focus on improving the birds behavior, so that each bird needs fewer steps to reach a good solution.

B. Swarm Behavior

Currently, if one bird decides to join another, it does so by picking the target randomly. This means joining any bird without considering how good the position of that bird might be. This contradicts the original idea of the authors that big birds tend to join others that have found a good food source (current solution seems promising) [1]. Therefore, we propose that a big bird will only be able to join the top b percent of birds that have the lowest current cost. If one chooses the right ratio, we expect that it will automatically nudge the swarm to exploit promising solutions.

We implement this by storing the indices i of the birds in an ordered integer array ord and introducing a new hyperparameter b, denoting which of the top-b percent can be joined. When we select the bird to join, we draw a random uniform number j between 1 and $b \cdot n$ (n denotes the number of birds), and get the index of the bird to join from the ordered array (ord[j]). The new hyperparameter b can be used to balance the algorithm between exploration (higher b means more variance in the target birds) and exploitation (lower b means tighter focus on the best birds).

The main disadvantage of this approach is that we have to continuously update ord, so that we only join the current top b percent at the moment of the move. We decide to update the list after each phase, where a phase is defined as a loop over all birds, in which each bird has moved once. We felt that updating the list after every single move was too costly, so this strikes a balance between more accurate results and less computation.

We tested numerous values for b and decided to build on joining only the best bird for future improvements, as this strategy gives the best results (Table 1). For intuitions on why such a low value performs so well, see Section VII.A.

b	1	0.25	0.20	0.15	0.05	0.01	**only best**
Error (in %)	497	323	279	246	150	79	**69**
Time (in s)	**19.7**	21.8	21.3	21.3	21.7	21.4	20.8

Table 1: Comparison of different parameters for our top-join. $b = 1$ means all birds are potential candidates for joining, which is the behavior of the original algorithm. It is not surprising that the baseline algorithms performs faster than our modified version, as we need additional time for sorting. Notice however that the computation time will not change for b-values in the interval $(0, 1)$, as we always need to sort all birds based on their performance.

The improvements reflect our intuition that good food sources (i.e. solutions) are more worthwhile to exploit than others. Interestingly, if during a phase all birds that want to join another are forced to join the same bird (the best one of the current phase, which does not change since we do not update the ranking within a phase), we get the lowest error out of all configurations. We analyze why this setting performs so well in Section VII.A.

C. 3-opt Walk

For the walk move, the original algorithm uses a modified version of the 2-opt heuristic as local search [1]. Instead of using 2-opt, we tested a 3-opt variant, as this often yields the best solutions considering the computational complexity [4]. At the same time, we decide to remove the estimation of local similarity from the algorithm, which was used together with 2-opt to perform a local search. We do this, because the authors did not provide any intuitions on why this might be beneficial [1].

With 3-opt, each bird creates 7 slight variations of its current tour, and chooses the tour with the lowest cost (see Figure 1). At which 3 points a tour is opened is determined by a random uniform draw of 3 integers representing the indices of the nodes in the tour.

Even though we now have to compute the lengths of 7 possible tours in just one move, we decided to still count this as just one iteration (the evaluation of each tour normally costs one iteration (compare Section V.A)). The results can be seen in Table 2.

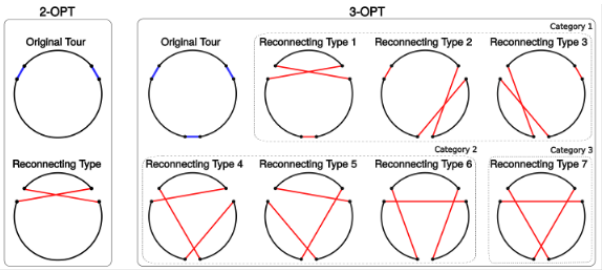

Figure 1: Comparison of 2-opt (left) and 3-opt (right) for TSP. Reconnections of category 1 are 2-opt variants [5].

Configuration	2-opt	**3-opt**
Error (in %)	69	**44**
Time (in s)	**21**	85

Table 2: Comparison of 2-opt and 3-opt. Note that the 2-opt configuration describes our previous configuration of the algorithm, which already includes the top-join of Section V.B. As noted in Section IV, each experiment builds on the most recent best performing one. Therefore, the configuration 3-opt denotes a top-join to the best bird of a phase, together with 3-opt for local search.

We acknowledge that this adjustment to the algorithm increases the runtime significantly, but because of the enormous improvement, we decide to commit to this change and focus on improving the runtime behavior in the following sections.

D. *Separating Responsibilities*

Apart from the obvious increase in complexity that results for the 3-opt algorithm itself, the main reason why the runtime quadruples is that every walk move has to evaluate seven possible new tours, from which the best is taken. To reduce the impact this has on the runtime, we decided to limit the responsibility of deep exploitation by 3-opt to a subset of birds.

We test the case where only big birds are able to perform a 3-opt walk, while small birds are only able to perform the usual 2-opt walk as specified in the paper. This should not only reduce the computational complexity, but also fits well with the assumption that big birds are "superior", since only they can join other birds and thus benefit from their discoveries. This effectively divides the flock into two groups: The small birds responsible for *exploration* and the big birds responsible for *exploitation*. See Section VII.B for details on this separation. For completeness, we also test the inverse approach: Only small birds can perform 3-opt.

Surprisingly, restricting 3-opt to big birds achieves the same performance as before, while cutting the runtime in half (Table 3). We also noticed an increased error rate for small birds, even though the ratio of small birds in our experiments is always greater than 50% ($r > 0.5$), meaning that the algorithm performs worse if we have more birds that perform 3-opt, given that those are small birds. This suggests that it is not only the number of birds performing 3-opt that is important, but also a clean separation of exploitation and exploration. Careless mixing of the two, resulting in extensive exploitation of suboptimal solutions, which will never lead to competitive results, can be detrimental to the performance of the algorithm.

Based on these results, we decided to use 3-opt only for big birds in our future experiments, as there seems to be no downside to this approach.

Configuration	2-opt	3-opt	**3-opt big birds**	3-opt small birds
Error (in %)	69	**44**	**44**	85
Time (in s)	**21**	85	40	69

Table 3: Because we usually configure more birds to be small than large ($r > 0.5$), we have a greater reduction in runtime when only big birds can perform 3-opt than when the reverse is true. Under these circumstances, it is surprising that we achieve better performance when big birds can perform 3-opt. Again, we compare this configuration to our best result so far (3-opt paired with top-join), and provide 2-opt paired with top-join as an additional reference.

E. *Nearest Neighbor Initialization*

So far, we have always initialized the birds with a random tour. While this allows for a very wide exploration, it also means that the algorithm needs many iterations to reach the first competitive solutions. To speed up convergence, we decided to use a simple heuristic to generate initial tours. Initially, we tried to implement the Christofides algorithm, which is the best-known polynomial-time heuristic for solving the TSP and provides a $\frac{3}{2}$-approximation. However, due to the complexity of the algorithm and the relatively long worst-case runtime of $O(n^3)$ [6], we decided to use the nearest neighbor heuristic instead. This heuristic is very simple, has a worst-case runtime of $O(n^2)$ [7], and still provides very good solutions.

Adding the heuristic initialization had a negligible impact on the runtime, which makes sense since the initialization is done only once. However, the results improved dramatically from an average error of 44% with random initialization to now 8%.

Initialization	Random	**Nearest Neighbor**
Error (in %)	44	**8**
Time (in s)	**40**	42

Table 4: We compare this approach to the best result from Table 3, which still uses the random initialization from the base algorithm. Adding nearest neighbor initialization reduces the error over all 50 tests by 36%. The increase in runtime is rather low, and therefore negligible, as the initialization is only done once.

F. Early Stopping

If we analyze the convergence behavior by plotting the cost of the best solution over the number of iterations (Figure 2), we notice that our improved version of the algorithm converges much faster than the original algorithm, even without a nearest neighbor initialization. Because it is difficult to estimate how many iterations are needed for a certain problem, and adapting the number of iterations to the problem at hand would be cumbersome, we decide to implement an early stopping mechanism. That way we do not waste computational resources on iterations that do not improve an existing solution, which will reduce the runtime even further, especially for smaller TSP configurations, while retaining a similar performance.

We implement this by introducing a new hyperparameter p, which denotes the number of phases without improvement of the best solution after which the algorithm stops. This requires us to store and continuously update the best solution for all birds. Fittingly, this is already implemented through the top-join (see Section V.B), as we already need to store the best solution over all birds in order to determine which bird to join. This is also the reason why we only check if the best solution has improved after each phase, and not after each iteration, as the ranking of the birds is not updated within a phase. A review during a phase therefore does not make sense.

EarlyStopping	No (default)	**Yes**
Error (in %)	**8**	10
Time (in s)	42	**9**

Table 5: Adding early stopping to our algorithm has the desired effect: We can reduce the runtime to less than half of the basic algorithm (compare Table 1), while increasing the error by a mere 2% compared to Section V.E. Overall, we are able to reduce the error for our benchmark problems (see Section IV) by **487%**.

This concludes our improvements of the algorithm. For visualizations of the optimization behavior, and a comparison between our incremental improvements, please refer to Figure 2 and Figure 3 in the Appendix.

G. Choosing Hyperparameters: Metabirds

Both the original algorithm and our extensions define various hyperparameters like the number of birds, the ratio of small birds, the top-b join percentage, the probabilities of the basic moves We have already discussed analytical approaches (for example Section V.A or Table 1) to choosing some of these values, but these tests were limited and did not capture the possible complex interplay between different hyperparameters.

To remedy this, we chose to apply an optimization algorithm to the hyperparameters of the TSP solver. The choice of optimization algorithm was simple: we used the same AFB algorithm that the TSP solver itself uses. We call this algorithm *Metabirds*, as every bird in the hyperparameter optimizer contains itself a swarm of birds solving the TSP.

To be more precise, every metabird represents a position in the hyperparameter space made up of values for the number of birds, the ratio of small birds, the top-b join percentage and the various move probabilities. For the walk of a metabird, we sample random deltas from a normal distribution and apply them to the hyperparameters. Flying to a new position is done by choosing random values for the hyperparameters.

The problem with this simple implementation is that they can produce invalid values. Specifically, the following conditions must be met:

- the sum of the move probabilities cannot exceed one
- the top-b join percentage must be large enough to include at least one bird
- the number of birds cannot be negative.

Since these meta-moves do not contribute a lot to the overall runtime, we decided to solve this by simply sampling new configurations until we find one that is valid.

To evaluate the cost of a metabird, we create a TSP solver with the bird's configuration, run it 10 times and average the results.

We ran the metabirds algorithm multiple times with different problems (eil101, d493, dsj1000, fnl4461) to optimize for different TSP sizes. To make these runs, which took multiple days, feasible, the algorithm was compiled to native code using GraalVM and executed on cloud resources.

VI. Sequential Ordering Problem

In addition to the TSP, we also applied the AFB algorithm to the Sequential Ordering Problem (SOP). The SOP is an extension of the TSP which puts additional contstraints on the order in which nodes are visited. Looking at the cost matrix C of a TSP instance, some edges C_{ij} are valued with -1, denoting that a valid tour must include node j before node i.

A. Naive Approach

The naive approach to extending the TSP solver to solve SOP instances is to simply repeat the existing fly and walk algorithms until the resulting path doesn't violate any of the constraints. However, as the number of constraints grows, this can lead to significantly longer processing times, especially for the fly move.

B. Generating Valid Paths

A more sophisticated approach is to modify the algorithms, so that they always return a valid path. Let's take a look at how to do this for the fly move.

Some nodes have dependencies, that is, a set of other nodes that must appear in the path before they do. To ensure that the generated random paths are always valid, we sequentially draw new random nodes, excluding those whose dependencies haven't yet been satisfied. To implement this, we maintain a pool of nodes that have not yet been drawn, but whose dependencies have been satisfied. To keep the pool up-to-date, we need to update it after each draw, removing the drawn node and adding the nodes that had the newly drawn node as their last unfulfilled dependency.

To find the nodes which have no more unmet dependencies, we store the number of unmet dependencies for every node in an array adjacent to the node array. After every draw, we loop through the relevant row of the cost matrix to find all nodes depending on the one drawn, and decrement their dependency count. If the count reaches zero, we add the node to the pool.

This approach limits the worst-case runtime of the algorithm to $O(n^2)$ (n draws and n possible dependents per draw), where n is the number of nodes.

VII. Analysis

A. Intuitions on our Improvements

Swarm algorithm usually include action of agents which can either be classified as exploitation or exploration. Exploitation means an agent uses its current result and tries to improve it, i.e. the agent continues to go into the direction he previously went in the search tree/space. For AFB, this is can be achieved using the walk move, so a local search.

Exploration means an agent tries to find a better solution not necessarily dependent on its current solution. Therefore, it can be seen as a global search. For AFB, this can be achieved using the fly move.

For a swarm algorithm to deliver a good approximation with respect to the global minimum it needs a good balance between exploitation and exploration: It needs to be able to improve a good solution, and search for different solutions if the current one does not seem promising.

If exploitation is too dominant, then the algorithm might get stuck in a local minimum, while other areas of the search space aren't explored at all, and vice versa.

As might have become apparent in the sections prior, the main focus of this paper was on exploitation (improving a good solution). This is mainly done by the introduction of 3-opt and that big birds can only join (the most) successful birds. Even though the latter cannot be seen directly as a local search, it does lead to more (big) birds performing exploitation of good solutions they adapted from other birds.

We explicitly do not modify the fly move, i.e. selecting a random tour, as this provides us with a rich selection of other possible solutions, which at the same time are completely independent of the current solution (of an agent).

The prior is crucial for the success of our algorithm, because the extreme join behavior modeled by the algorithm has a high risk of getting stuck in a local minimum: Starting with our improvements on the swarm behavior, a big bird will only be able to join the best bird. Because which bird is the best is not updated within a phase, it could be that during one phase a lot of big birds join the same bird, putting a lot (not all, as a join is a matter of probability) of agents in one place in the search space. The fly move enables birds to escape this single solution.

Furthermore, there is only a limited number of big birds, meaning small birds are able to explore the search space elsewhere, while the big birds are focusing on the best current solution.

Therefore, we get an algorithm that has an empirically tested balance between exploitation and exploration. It exploits good solutions in a harsh manner while also being able to switch to completely new solutions if they prove to be better. This process will be repeated until a solution is reached that will either be close to the optimum, or a local minimum. Either way, the algorithm converges.

B. Exploitators and Explorers

The base algorithm consists of two types of agents, small and big birds. With our improvements it may have been noticeable that we separated the roles of both agent types more and more from each other: While small birds can perform the usual 2-opt local search when they are walking, big birds can perform a more powerful 3-opt walk; big birds can join other birds. We do this in order to make the components contained in each swarm algorithm, exploration and exploitation, a more explicit part of the algorithm: We delegate small birds to the role of explorers, and big birds to the role of exploitators.

Small birds are able to access vastly different areas of the search space for possible better solutions than their current one. Using the join-move, big birds are able to profit from those that have found the best current solution by joining them and improving that solution using 3-opt (walk).

The circumstance that small birds can also perform exploitation, using their own version of the walk move (2-opt), is owed to the fact that they otherwise would only be able to

perform the fly move, i.e. jumping between random solutions. This wouldn't be a good foundation for the join behavior of big birds (see Table 6), which is essential for the performance of our algorithm. Also, since big birds can also join other big birds, and the solutions for small birds would be rather poor, the probability that big birds will exclusively join other big birds would be very high, making small birds essentially useless.

Exactly this can be verified by simply comparing how the algorithm performs when (1) small birds can only fly, (2) all small birds are removed from the algorithm, and only big birds are kept.

Surprisingly, the results show us that configuration (2) performs even better than variant (1), indicating that in (1) the big birds only join other big birds, and that small birds, whose only purpose is to perform the fly move (so not even returning to their best solution), provide no value to the algorithm. This is why we decided that small birds are also able to perform the walk move.

Configuration	**Regular**	Only fly	No small birds
Error (in %)	**8**	15	10

Table 6: If small birds are only able to fly, the algorithm performs worse than before. Notice however that it still achieves a reasonable performance. For our experiments we continuously used 200 birds, 150 of them being small birds. So by removing all small birds for experiment (2), we are left with 50 (big) birds.

It is important to note that to make this experiment fair, we also reduced the number of iterations by a factor of three quarters (same ratio with which birds have been reduced), so that the search depth, meaning the average number of steps each bird can perform in the search space, is the same as if we use the regular 200 birds (compare with Section V.A).

VIII. Conclusion

This paper focused on various improvements to the original AFB metaheuristic [1]. Our changes significantly reduced the error rate of resulting TSP tours and improved the convergence speed of the algorithm. We were careful to keep the algorithms simplicity so it remains easy to apply to new problems. We believe that our experiments show the AFB algorithm to be a competitive metaheuristic which can also be useful for educational purposes due to the ease of implementation.

References

[1] J.-B. Lamy, "Artificial Feeding Birds (AFB): a new metaheuristic inspired by the behavior of pigeons", *Advances in nature-inspired computing and applications*, pp. 43–60, 2019.

[2] W. J. Cook, D. L. Applegate, R. E. Bixby, and V. Chvatal, *The traveling salesman problem: a computational study*. Princeton university press, 2011.

[3] D. W. Hoffmann, *Theoretische Informatik*. Hanser Verlag, 2009.

[4] S. Lin, "Computer solutions of the traveling salesman problem", *Bell System Technical Journal*, no. 10, pp. 2245–2269, 1965.

[5] J. Sui, S. Ding, R. Liu, L. Xu, and D. Bu, "Learning 3-opt heuristics for traveling salesman problem via deep reinforcement learning", 2021, pp. 1301–1316.

[6] R. van Bevern and V. A. Slugina, "A historical note on the 3/2-approximation algorithm for the metric traveling salesman problem", *CoRR*, 2020.

[7] M. M. Flood, "The traveling-salesman problem", *Operations research*, no. 1, pp. 61–75, 1956.

Appendix

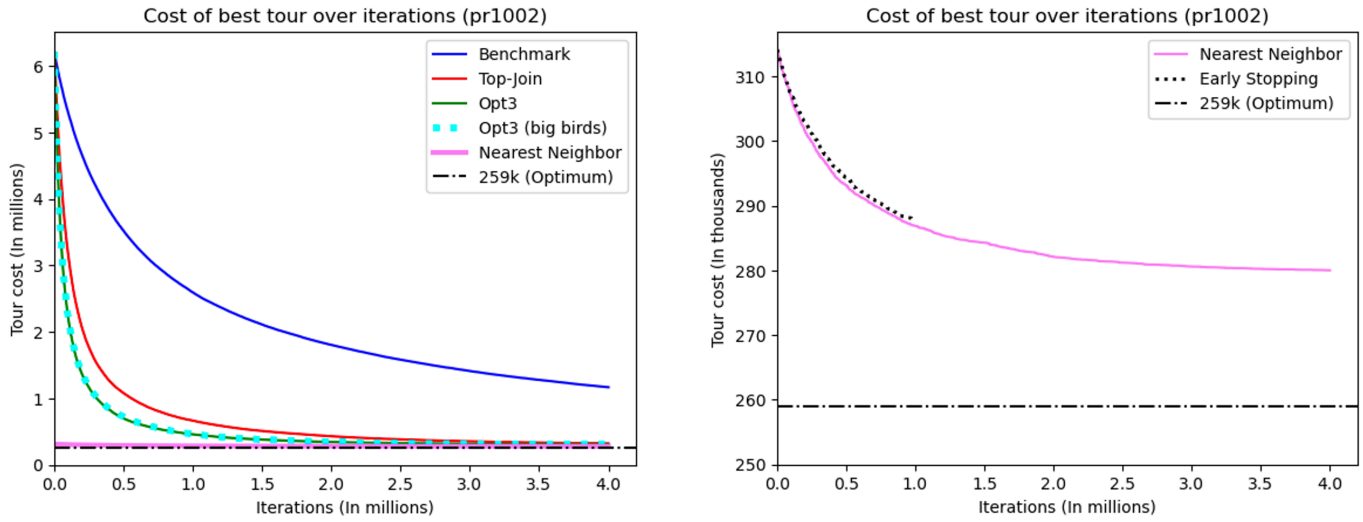

Figure 2: A visualization of the optimization behavior for different configuration of the AFB algorithm (TSPLIB problem pr1002). "Benchmark" denotes the normal AFB algorithm, without any adjustments. Each configuration, from top to bottom, includes the respective improvements made prior. For example, "Top-Join" consists of the basic algorithm with the top-join added, "Opt3" consists of the basic algorithm, together with the top-join and 3-opt, instead of 2-opt. Each addition to the algorithm results in a faster convergence of the algorithm, while consistenly converging to a lower cost, or shorter tour. Moreover, it is observable that the restriction of 3-opt to big birds does not appear to change anything in the performance and optimization behavior of the algorithm. With a nearest neighbor initialization the algorithm gets a significant head start. However, the optimization behavior can still be observed, even after this initialization.

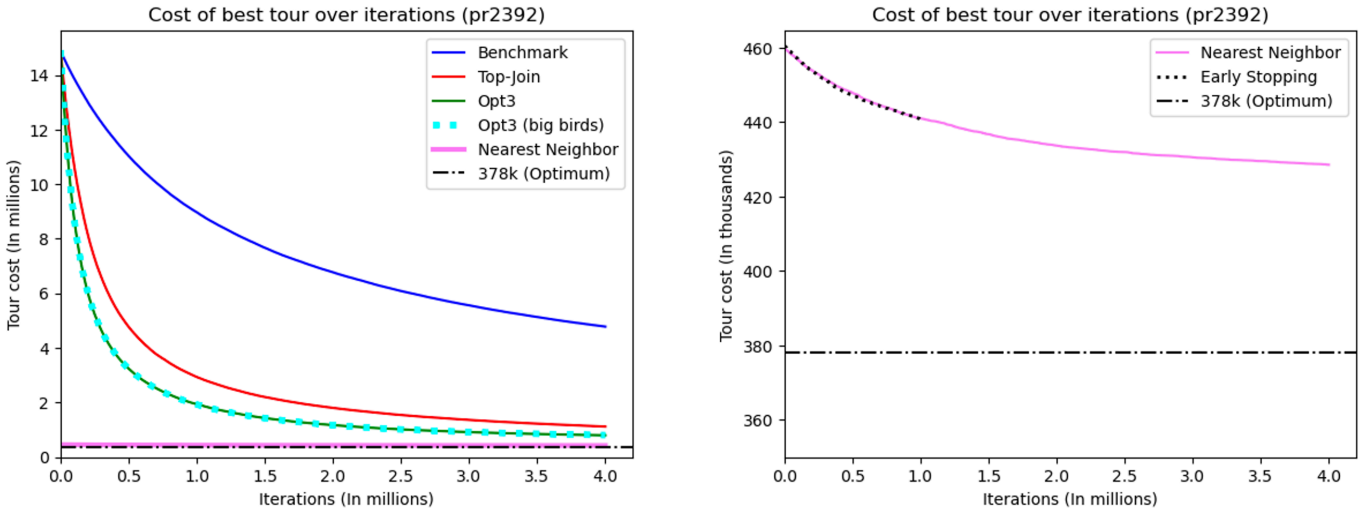

Figure 3: A visualization of the optimization behavior for different configuration of the AFB algorithm (TSPLIB problem pr2392). We observe the same behavior as in Figure 2, this time for a bigger problem with 2392 cities (instead of 1002). Here a nearest neighbor initialization again yields a significant head start. Nearest Neighbor, together with Early Stopping, is also visualized on the right hand side to better illustrate the optimization behavior when the initial solution is a lot closer the the optimum than a random initialization.