Bio-Inspired Optimization Algorithms to Solve The Traveling Salesman Problem

Lukas Kelsey-Friedemann (kelse111)

14 May 2019

1 Motivation and Traveling Salesman Problem (TSP)

1.1 TSP

The Traveling Salesman Problem (TSP) is one of the most studied problems in computer science, as it poses a difficult enough problem that it takes a sophisticated algorithm to find an optimal solution in a reasonable amount of time. It was first studied in the 18th century by Sir William Rowam Hamilton of Ireland and Thomas Penyngton Kirkman of Britain. This level of difficult is considered to be NP-Hard. The problem is state as follows: you are given a list of cities, with each distance between each city, and you are to find the shortest route to visit all cities and end at the first city you visited(technically known as ATSP). This proves to be a particularly difficult problem when the search space becomes large (n! if given n number of cities). As a result, many optimization algorithms are tested by seeing how well they perform when solving TSP for various size search spaces. Additionally, TSP has many applications where fast and good solutions are necessary: school bus routing, drilling and printing circuit boards, overhauling gas turbine engines...etc [7]. Thus, TSP is not only a difficult problem to solve, but also an important and reoccurring problem throughout our world.

1.2 Need for Better Bio-Inspired Algorithms

Further motivation can be found in the practice and testing of bio-inspired algorithms to solve real engineering problems. In one instance, bio-inspired algorithms were tested to compute and optimize synthesis of offshore oil production risers. Computation time can be a serious problem and the algorithms need to reach certain benchmark it terms of the fitness of their solutions, in addition to having minimal run-time. Three bio-inspired algorithms were used: Genetic Algorithm, Particle Swarm Optimization, and Artificial Immune system. Particle Swarm Optimization and Artificial Immune System were able to reach the benchmarks and have limited run-time although were not stellar. Additionally, Genetic Algorithm, a commonly used optimization search algorithm, was not able to meet the fitness benchmarks [9]. Clearly there is room for improvement for bio-inspired algorithms to solve real world problems.

1.3 Standardized Data Sets and Common Implementations

This project references academic journal articles and papers for bio-inspired algorithms, each of which had very different implementations. This isn't to say that the algorithms should be similar, as this paper clearly outlines the differences in the algorithms, but rather in their adaptation to solve TSP. Later in this paper it is discussed how continuous optimization algorithms, which all of the bio-inspired algorithms within this paper are, cannot solve TSP (without modification). TSP is a discrete problem with permutations of cities, where a continuous optimization algorithm needs to be significantly modified in order to solve this problem. It is in this adaption and modification of the algorithms that a standardization is necessary. There are common solutions to this issues, which will later be discussed, and what this project explores is the standardization of adapting continuous optimization algorithms to solve TSP. Additionally, the algorithms outline in this paper have not all been tested on one standardized data set. Benchmark data sets exist where many algorithms are tested against these data sets to test performance. However, from the literature explored in this paper, these bio-inspired algorithms have not be thoroughly tested against each other on one common data set.

2 Bio-Inspired Algorithms: Ant Colony, Bat, FireFly Algorithms, Cat, and Bacterial Foraging

Optimization algorithms were in practice long before the field of computer science, where species around the planet have sophisticated means of conducting searches for food, water, mates...etc. Some computer scientists have turned to the natural world for inspiration into how to write optimization algorithms, trying to find clues for search optimization in the thousands of evolution manifesting in a species' search optimization techniques. This phenomena has resulted in some sophisticated optimization algorithms and here, five will be discussed: Ant Colony, Bat, FireFly, Cat, and Bacterial Foraging algorithms. These five algorithms are all population based algorithms, where there is an initial population that conducts a search over some space. All five algorithms are inspired by nature where a species is able to solve difficult search problems by independently navigating a system with other players. Each of these algorithms is well suited to solve TSP as they are optimization algorithms, however, some have been thoroughly tested in the field and others have not.

2.1 Ant Colony Search

Ant colonies have adapted to find short paths to food from their nests, and do so independently by leaving pheromones on their trails. When encountering obstacles on a path, say a rock, the ant will most likely choose the path around the obstacle with the most pheromone. Therefore, paths that are traveled more are more likely to be used and faster paths are more likely to be chosen(it can be traveled over more in a shorter period of time). The Ant Colony algorithm operates in a similar fashion where there is an initial population that finds paths to solutions and eventually an optimal solution arises from individual ”ants” taking paths with calculated probabilities (based on ”pheromones”)[2].

2.2 Bat Algorithm

Bats in the natural world use echolocation, by emitting ultrasonic pulses and then listening to their echoes, to navigate the dark with limited vision. In order to adapt this for the purposes of an optimization algorithm the following rules are assumed: bats have an unknown ability to distinguish between prey and objects, they fly with a certain velocity v, at position x, a fixed frequency f, varying wavelength (lambda) and loudness A, and lastly the loudness A is within a fixed ranged. These criteria are updated at every iteration of the algorithm to conduct a search. Each bat has the ability to conduct a complete search and thus as a collective population they can find an optimal solution [8].

2.3 FireFly Algorithm

The FireFly Algorithm was inspired by fireflies ability to find mates by flashing light. Similar to the Bat Algorithm there are three rules that structure the algorithm: all fireflies are unisex and ”attracted” to all other fireflies, a FireFly is attracted to brighter lights(fireflies), and brightness of a FireFly is determined by an objective function. Parameters of light intensity, initial attractiveness, and light absorption are defined by varying coefficients. These are updated with every iteration of the algorithm until a max iteration is reached[5].

2.4 Cat Algorithm

The Cat Algorithm works by following some key basic logic of cat behaviour: the cat-agent is either in searching mode (resting and thus moving slowly) or in tracing mode (hunting and thus moving quickly). The mode of the agent is determined by some mixing ratio. Furthermore, each agent is represented by its velocity, flag(searching or tracing mode), and its position (solution). Each agent has its velocity and flag determined by its own fitness (i.e. the quality of its solution to the problem at hand). In resting mode, the agent will carefully examine its surroundings to find a solution, whereas, in tracing mode the agent will ”trace” its own path proportional to its velocity[1].

2.5 Bacterial Foraging Algorithm

The Bacterial Foraging Algorithm represents the bacterial foraging(search technique) by four key concepts: Chemotaxis, Swarming, Reproduction, and Elimination and Dispersal. Chemotaxis is simply the bacteria having the ability to move, which is crucial in foraging for food because the bacteria can move to areas with higher nutrient density. Bacteria collectively release a combination of attractant and repellent so that they swarm nutrients in a concentric ring, so they ultimately work together to collect nutrients which is labelled as Swarming. Reproduction is represented as the basic concept that weak bacteria die and healthy bacteria will reproduce. Elimination and Dispersal is the concept that large environmental phenomena can effect the bacteria population at large, which results in a random dispersal of the bacteria. The four concepts are tied together to create an optimized search algorithm. [6]

3 Effectiveness of "Nature" Algorithms to Solve TSP

3.1 Ant Colony, Bat, and FireFly Algorithm Performance

The Ant Colony, Bat, and FireFly algorithms had specific implementations in each article referenced, and used different TSP's in order to test the performance of each of the algorithms. Ant Colony was invented decades before the other two algorithms which came about around the year 2010, and this algorithm is much more thoroughly tested and reliable. The three algorithms were all tested on one particular TSP, Oliver30, where the optimal solution is 420. Ant Colony completed with 423.7, Bat Algorithm with 420, and FireFly with 423.5. Unfortunately, between the three articles this was the only quantitative bench mark to compare the three algorithms. Although, all three were tested against a Genetic Algorithm, where both Ant Colony and Bat algorithms outperformed the GA. All three algorithms slightly different approaches to solving similar search problems, although from the analysis of the three articles Bat Algorithm seems to have a higher performance. This is due in part to improvements made to the basic algorithm, in order for it to perform well in an deterministic environment [8]. Additionally, this algorithm is quite newer and less tested than Ant Colony, with a more sophisticated process to determine a path to take. Therefore, Bat Algorithm will be used to further analyze and solve TSP.

3.2 Cat and Bacterial Foraging Algorithm Performance

Cat and Bacterial Foraging, from the resources explored, have not been tested on the same data sets as the other algorithms. The exception being the Cat Algorithm was tested on KroA100, a TSP data set that Ant Colony has also been tested on with a solution of 21285.44 [3]. Whereas, Cat Algorithm was able to find a solution that was optimal: 21282[1]. This goes to show that some newer bio-inspired optimization algorithms may be able to improve on older ones. Bacterial Foraging Algorithm was not tested on TSP, and thus this project can provide some key insight into its performance on TSP and also how it compares to other bio-inspired optimization algorithms.

4 Adapting Optimization Algorithms to Solve TSP

4.1 Continuous Optimization Algorithms Cannot Solve TSP Without Modification

The research referenced in this paper, which used the bio-inspired algorithms to solve TSP all had to modify their algorithms to solve TSP. There is an inherent issue when trying to solve TSP with continuous optimization algorithms. TSP is a discrete problem where a solution is represented by a permutation of cities, which have an associated cost to their order(determined by the permutation). Continuous optimization algorithms have solutions represented by a real number. The difference in these kinds of problems can be seen by the following example: if you have a continuous optimization algorithm you can modify a solution x, a real number, by simply multiplying it by any real number y and then testing its fitness on the function you are trying to optimize. Inherently, what the actual continuous function does is a bijective mapping from the real numbers to the real numbers. TSP on the other hand, has a domain of the discrete set of permutations (n! for n cities). So lets say a solution had permutation p and cost x, you cannot simply multiple p by a real number y or the cost x and then test it against some function to test for fitness. The fitness comes from the inherent structure of p, so a real number y must now mutate a permutation p in some way to improve its cost x. This process, of taking some real number y and meaningfully mutating p to lower the cost x, is where the real task of modifying continuous optimization algorithms to solve TSP lies. Moreover, the basic functions of determining the difference of two solutions, x-x', is difficult. If x and x' were real numbers this would be trivial, however, if x and x' are permutations then you need a way to compute the difference. To summarize, the two questions are how does one compute the difference between two points/solutions (the difference x-x') and how does one mutate a permutation (x+x' or x*x')?

4.2 Hamming Distance Vs. Swap Distance

There are two methods to determining the difference between two permutations: Hamming distance and swaps. The Hamming distance between two permutations is the number of elements that do not match between two permutations. For example: [1,2,3,4,5] and [2,1,3,4,5] have a Hamming distance of two. The alternative is the number of swaps required to turn one permutation, p1, into another permutation, p2. If p1 is [2,3,1,4,5] and p2 is [1,2,3,4,5] then the swap sequence is as follows: 0:[2,3,1,4,5], 1:[1,3,2,4,5], 2:[1,2,3,4,5]. The swaps follow the order that the first element of the permutation is swapped to be in the correct place, then the second element, then the third and so on. So two swaps are needed to turn p1 into p2. Notice that p1 and p2 would have had a Hamming distance of three.[4] Hamming distance is easy to compute, where for every index i of the permutations if the element at i of p1 is not equal the element of i at p2 then the Hamming distance is incremented by 1. The computation of swap distance is more involved, however, a sequence of swaps is simultaneously computed in the process. This promises to be more advantageous as it not only gives a value for the difference between two permutations, but also a sequence of actions(swaps) to mutate one permutation into another[5].

5 Modifying SwarmPackagePy to Solve TSP

5.1 Classes for TSP and Implementation of Swap Operator and Sequence

This project involved modifying SwarmPackagePy, a Python library with bio-inspired optimization algorithms, to solve TSP. Originally, each algorithm within the library took in some parameters, each specific to the algorithm being used, and a continuous function to optimized. This project aimed to modify each algorithm as little as possible, to keep the integrity of the algorithms, however, major modifications were necessary. Additionally, this project aimed to modify each algorithm in a similar way; being that basic operations of x-x', x+x', and x*x' could be replaced all by the swap method. First, classes for cities(nodes) were created. Where each city upon initialization would assign itself an x-coordinate and a y-coordinate, each of which was a random integer between zero and two-hundred. Each city also had a method to calculate the Euclidean distance to another city (when given the other city). Secondly a class to represent permutations of cities was created, called a country(this would later be used to also represent each agent). Each country held a permutation of cities in an array (this was passed as a parameter on initialization) and had three crucial methods. The first method was costOfRoute() which calculated the cost of associated with its permutation, swapOp(list-of-swaps,n-swaps) which would swap the first n-swaps of tuples in list-of-swaps (each tuple was contained two indices of the permutation array to be swapped), and numberOfSwapsTo(some-country) with returned the number of swaps and list-of-swaps necessary to mutate the country's mutate permutation to the permutation of some-country's. The second two methods are directly implementing the swap method mentioned earlier, critical to adapting continuous optimization algorithms to solve discrete TSP. Lastly, a tsp class was created, which aimed to ensure that each country(agent) would be working upon the same data set. So tsp would be passed a number of cities to be created, essentially the size of the TSP problem. A method for tsp was created called makeCountries(n) which would make n number of countries and where each would be passed the tsp problem (the cities) and each country would create a random permutation of the cities. Lastly, it contained a method called bestSolution() that would find the country(agent) with the best solution to the problem. These classes were designed so that each algorithm within SwarmPackagePy could be tested on a standardized data set. Additionally, minimal modifications were intended to be needed to change each algorithm, in terms of having the infrastructure for TSP and implementing the swap method.

5.2 Difficulty in Adaptation

Sharad N. Kumbharana and Gopal M. Pandey in their paper [5], explained the differences between adapting a continuous optimization algorithm to solve TSP using Hamming distance and swap distance, however, gave little detail into their exact implementation of these technique(s) into their FireFly algorithm. Furthermore, each paper referenced in this project gave a brief summary of their methods used to modify their algorithms but the details of their approaches were vague. Attempting to modify each algorithm with some standardization of utilizing the swap method proved to be extremely difficult. Where, for each algorithm it was difficult to determine which parts of the algorithm were crucial to maintaining the effectiveness of the algorithm, while simultaneously trying to adapt the algorithm to solve TSP. There are certainly places within each algorithm where individual pieces cannot be adapted for TSP and must be omitted. This laborious process, although within the scope of this project, could not be completed for every algorithm discussed in this paper. This is a much bigger project that anticipated where immense amount of time and research must be dedicated to every algorithm.

5.3 Modification to FireFly Algorithm

FireFly algorithm was adapted to solve TSP in several places, which resulted in the algorithm to successfully find a solution to various TSP's. The parameters specific to the mathematics and inherent structure of FireFly algorithm were at left as the default settings within SwarmPackagePy, as these variability of success by FireFly with regards to these parameters is not within the scope of this project. The method used to adapt FireFly algorithm was to replace any x-x', which compared to points(solutions), by the number of swaps needed to change one solution x into x'. Secondly any mutation of a point x*y where y was some real number derived from the difference of x from some other point z, was changed to x having some w swaps done. Where w was y proportionate swaps of the total number of swaps needed to change x to z. Lastly, the original FireFly algorithm heavily relied on the use of normal Gaussian distribution of points. This proved to be difficult since there isn't really a way to have a normal distribution of permutations. The original algorithm had upper and lower bound associated to the solution, which were passed as parameters to the algorithm. The Gaussian normal distribution was derived within these bounds. However, those boundaries don't exist for permutations (the points used to solve TSP), so a Gaussian normal distribution could not be applied to the points. So instead, the permutations of each agent were completely randomized.

6 Experimental Results

6.1 Experimental Procedure

FireFly was the only algorithm successfully modified, so to verify its solution Brute-Force algorithm was applied to the same data sets to determine the optimal solution. Brute-Force algorithm determined every permutation possible for each TSP and then calculated the cost for every permutation, thus determining the absolute best solution to each TSP. Run-time is excluded from the analysis of FireFly algorithm because the only metric to compare it against is Brute-Force, which has the worst run-time possible. So the only note to be made about FireFly's run-time is that it was always better than the worst possible run-time. Furthermore, the run-time of FireFly is really determined by the number of iterations ran within the algorithm. Where each iteration cycle causes mutations to be made to the points, following the logic of the algorithm's design. The use of Brute-Force to compare the solutions of FireFly inherently limited the problem of TSP the tests were able to use. Only small data sets were used because Brute-Force is a very time intensive procedure. The first round of computation used ten cities for TSP and FireFly used five agents, where only the number of iterations for FireFly algorithm varied (seen in table 1). The second round of computation used ten cities for TSP and FireFly used one-hundred iterations, and only the number of agents used by FireFly varied (seen in table 2).

6.2 Data Tables

Below in table 1 are the solutions for FireFly and Brute-Force on TSP with ten cities, where the iterations of FireFly vary for five agents.

FireFly	Brute-Force	Iterations
867.917	362.087	100
847.950	405.447	1000
829.912	415.912	10,000

Table 1: Solutions to TSP with 10 Cities and 5 Agents.

Below in table 2 are the solutions for FireFly and Brute-Force Algorithms to solve TSP with ten cities, where the number of agents for FireFly vary for one-hundred iterations.

6.3 Experimental Analysis

The modified FireFly algorithm did not come close to finding an optimal solution to TSP. For each computation of TSP FireFly had a significantly higher route cost than the optimal solution. By varying the number of iterations, coming at a cost of run-time, the solutions marginally improved. The difference in route

FireFly	Brute-Force	Agents
825.737	355.761	3
867.917	362.087	5
810.851	394.174	9

Table 2: Solutions to TSP with 10 cities and 100 iterations.

cost between FireFly's solution and Brute-Force's were 505.830, 442.503, and 414.000 for 100, 1000, and 10,000 iterations, respectively. So with an increase in iterations the solutions were able to improve. However, most notably is that the difference in FireFly's solution and the optimal solution, is greater than the optimal solution's actual value. Moreover, the same is true for the second round of computations. For the second round of computations the difference in route cost between FireFly's solution and Brute-Force's were 469.976, 505.83, and 416.677 for 3, 5, and 9 agents respectively. So changing the number of agents for the algorithm did not drastically improve or degrade the solution, and the solutions themselves were terrible. The extremely high route cost from FireFly's computation on TSP are most likely stemming from the adaptation of the algorithm in this circumstance. Sharad N. Kumbharana and Gopal M. Pandey in their experiments were able to have good solutions to TSP result from running their modified algorithm on the problem [5]. Whereas, in this circumstance the implementation of swap method must not be correct or too much of the original algorithm was changed and the integrity of the algorithms design was compromised. Additionally, these were not large data set and if Brute-Force was able to run in a reasonable amount of time, then going through the hassle of running FireFly would not be worth the result. As, implementing a Brute-Force solution is near trivial and guarantees an optimal solution. For larger data-sets, this implementation of FireFly found solutions, although the results do not have a metric for comparison so their are disregarded.

7 Conclusion

The experimental results of this project do not give key insight into the usefulness of bio-inspired algorithms, specifically FireFly algorithm. However, the research and programming put into this project can give some insight into the adaptation of continuous optimization algorithms to solve discrete problems such as TSP. The adapted FireFly algorithm of which this project created and tested against solving TSP most likely has defects within its implementation. That is to say, the poor solutions resulting from this algorithm's computation stem not from the algorithm's theoretical design but rather its implementation in this circumstance. What can be said about the work done in this project is that adapting continuous optimization algorithms to solve discrete problems such as TSP is not an easy procedure. The implementation of basic theoretical concepts such as Hamming distance and swap distance is non-trivial. This project aimed to create standardization for this process of adapting algorithms, however, in that regard it illustrated the difficult in achieving this very goal. With more resources this work could be expanded upon by firstly writing the bio-inspired algorithms from scratch, instead of attempting to adapt already existing software packages. Good search/optimization algorithms are in high demand and the work done by this project could be expanded upon to come to more conclusive methods to use these bio-inspired algorithms to solve real world issues.

References

[1] Abdelhamid Bouzidi and Mohammed Essaid Riffi. “Discrete Cat Swarm Optimization to Resolve the Traveling Salesman Problem”. In: *International Journal of Advanced Research in Computer Science and Software Engineering* 3 (2013), pp. 13–18.

[2] Marco Dorigo and Luca Maria Gambardella. “Ant colonies for the travelling salesman problem”. In: *biosystems* 43.2 (1997), pp. 73–81.

[3] Zhifeng Hao, Han Huang, and Ruichu Cai. “Bio-inspired Algorithms for TSP and Generalized TSP”. In: *Traveling Salesman Problem*. Ed. by Federico Greco. Rijeka: IntechOpen, 2008. Chap. 2. DOI: 10.5772/5583. URL: https://doi.org/10.5772/5583.

[4] Kang-Ping Wang et al. “Particle swarm optimization for traveling salesman problem”. In: *Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.03EX693)*. Vol. 3. Nov. 2003, 1583–1585 Vol.3. DOI: 10.1109/ICMLC.2003.1259748.

[5] Sharad N Kumbharana and Gopal M Pandey. “Solving travelling salesman problem using firefly algorithm”. In: *International Journal for Research in science & advanced Technologies* 2.2 (2013), pp. 53–57.

[6] George Lindfield and John Penny. “Chapter 6 - Bacterial Foraging Inspired Algorithm”. In: *Introduction to Nature-Inspired Optimization*. Ed. by George Lindfield and John Penny. Boston: Academic Press, 2017, pp. 101–117. ISBN: 978-0-12-803636-5. DOI: https://doi.org/10.1016/B978-0-12-803636-5.00006-2. URL: http://www.sciencedirect.com/science/article/pii/B9780128036365000062.

[7] Rajesh Matai, Surya Singh, and Murari Lal Mittal. “Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches”. In: *Traveling Salesman Problem*. Ed. by Donald Davendra. Rijeka: IntechOpen, 2010. Chap. 1. DOI: 10.5772/12909. URL: https://doi.org/10.5772/12909.

[8] Eneko Osaba et al. “An improved discrete bat algorithm for symmetric and asymmetric traveling salesman problems”. In: *Engineering Applications of Artificial Intelligence* 48 (2016), pp. 59–71.

[9] Ian Nascimento Vieira, Beatriz Souza Leite Pires de Lima, and Breno Pinheiro Jacob. “Bio-inspired algorithms for the optimization of offshore oil production systems”. In: *International Journal for Numerical Methods in Engineering* 91.10 (2012), pp. 1023–1044. DOI: 10.1002/nme.4301. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.4301. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.4301.