Multi-objective optimization of the Multiple Traveling Salesmen Problem Using a Non-dominated Sorting Genetic Algorithm (NSGA-II)

Rahul Balamurugan

*Abstract*— **The multi-objective multiple traveling salesmen problem (MOMTSP) is a rather popular multi-objective combinatorial optimization problem (MOCOP) with many real-world application. The non-dominated sorting genetic algorithm NSGA-II is a fast, elitist genetic algorithm, and has been used to solve the problem and other similar MOCOPs in literature. This project report details the implementation of the NSGA-II algorithm to solve two variants of the MOMTSP— with a common objective to minimize the total traveled distance, one with objective to minimize the difference between the longest and shortest individual tour lengths (MinMax SD-MTSP), and the other, to minimize the sum of the difference between the average tour time and time taken for each individual tour. Four crossover operators, a small range of mutation probabilities, and three population sizes were tested to find the best combination of values among them for both cases. Additionally, the implementation showed better or similar performance when compared against the benchmark and results in literature. The obtained experimental results are detailed in this report.**

1. INTRODUCTION

Solutions to multi-objective optimization problems (MOOPs) often find applications in the real world [], as many real world problems often have multiple conflicting objectives. A few of the domains that benefit from research into such solutions are resource management [], manufacturing [], network coverage [], and various other domains []. A popular example of such problems is the multi-objective multiple traveling salesmen problem (MOMTSP), first studied by Frederickson, Hecht, and Kim [] in 1978.

The MOMTSP, which is NP-Hard, can be briefly formulated as minimizing the total distance travelled and balancing the workload for *m* salesmen visiting *n* cities exactly once, with each salesman originally departing and finally returning to a common origin city or ‘depot’. A particular modern application of the problem that comes to mind would be in the deployment of multiple drones to achieve certain goals (e.g. delivery of packages) at certain locations and return. Due to limitations on battery life, there is a need to ensure all deployed drones are active for similar times so the coverage of locations can be maximized and all drones allowed to return before the end of their battery life. In addition, many other multi-objective combinatorial optimization problems (MOCOPs) like the vehicle routing problem [] and job shop scheduling problem [] can be transformed into the MOMTSP and solved, which allows the MOMTSP to act as an adequate test problem for MOCOP solvers.

Evolutionary algorithms have been at the forefront of heuristic algorithms for solving MOCOPs and MOOPs for over a decade [], with the non-dominated sorting genetic algorithm NSGA-II, proposed by Deb [] in 2002, being the most cited and used algorithm in literature for solving problems in this class []. Some salient points which make this algorithm so popular are that it incorporates an explicit diversity preservation mechanism, as well as implicit elitism [] to retain the best solutions across generations. Due to the mentioned advantages, proven effectiveness [] as well as the availability of the NSGA-II, the algorithm has been used as a sort of gold standard for comparing other (novel) algorithms [], especially since similarly (or more) efficient tools used in industry are inaccessible.

However, the NSGA-II when used with conventional crossover operators devised for permutation-based problems like the Partially Mapped Crossover (PMX) has been noted to converge prematurely for combinatorial problems []. Although there is a lack of studies as to why this is the case, it may be intuitively attributed to two reasons: first, a lack of evolvability when using the mentioned operators, as the ‘good’ subsequences or arrangement of genes are not preserved during inheritance; and secondly, because in later generations, the population of solutions may lack in diversity []. Hence, as [] notes, most applications of the NSGA-II for MOCOPs involves some level of modifications.

On the other hand, [], a comprehensive review, noted that small modifications to the NSGA-II, in terms of specialized genetic operators, replacing or supplementing the diversity preservation mechanism, or hybridizing with other algorithms [] can allow the resultant algorithm to be just as good if not better than the benchmark algorithms for the considered problem domain.

In this project, the main objective is to implement the NSGA-II algorithm to solve the MOMTSP and test a number of crossover operators, and effects of varying population size and mutation probabilities on instances from TSPLIB, comparing the results with results from literature and benchmarks (where available).

Running with the drone application mentioned earlier, two different objective functions were considered separately in addition to minimizing the total distance traveled— *first*, minimizing the difference between the longest and shortest individual tours, which would be representative of the ideal case where all salesmen (or drones) deployed travel at the same speed and without any delays between cities; and the *second*, minimizing the difference of the total travel times of each salesman with the average travel time, assuming that each traversal between two cities *i* and *j* is done at a random speed between a given range. The former MTSP problem is known in literature as the MinMax SD-MTSP and has benchmarks [] for certain instances from TSPLIB in literature obtained using the brute force CPLEX method, while the latter has not been studied much [] and does not have a benchmark. Either ways, the effectiveness of the implementation can be observed by comparing the final range of total traveled distances obtained with the benchmarks as well as from [] and [].

Chart, line chart

Description automatically generated

Fig. 1: Plot showing two convex pareto fronts that could be obtained while optimizing an arbitrary bi-objective problem with objective functions obj fn 1 and obj fn 2. The blue front dominates the red front. Other pareto fronts could be concave or irregularly shaped (neither concave nor convex).

The main deliverable for the project is the software written for the implementation in Python, submitted alongside the report and also made available in a public access GitHub repository. The repository contains functions to implement the NSGA-II as well as the various crossover operators mentioned, four mutation operators, and a selection operator; and also the main code to solve the MOMTSP (both variations) for both random instances and select instances (eil51, berlin52, eil76, and rat99) from TSPLIB, which are also included as text files.

The project report beyond this introduction is structured as follows: first, the relevant background information needed for the project, such as terms related to multi-objective optimization and the NSGA-II algorithms are introduced along with the mathematical formulation(s) of the MOMTSP; the next section briefly mentions related work on NSGA-II as well as its (and other GA) application to variations of the MTSP with specially mentioned proposed solutions to the above discussed issues with EAs; thirdly, the implementation of the algorithm and genetic operators is detailed along with the design of the experiments to test the implementation; and lastly, the obtained results are discussed, with some critical analysis connecting the results to the broader issues in EAs and their usage in solving MOCOPs.

2. BACKGROUND

***2.1. Pareto-Optimality and Dominance***

In multi-objective problems, it is required to find an optima for every one of the objectives, and thus, for conflicting objectives, it becomes impossible to find a singular optimal solution. Instead, the goal of solving the problem is to determine an optimal pareto front, which is the set of optimal solutions to the problem (Pareto-Optimal Solutions or POS), from which appropriate trade-offs can be considered to choose a suboptimal solution.

The pareto-optimal front is composed of solutions that “dominate” all other solutions but not each other. Here, a solution, say X, is said to dominate another, say Y, if X evaluates to better than Y for at least one objective function, and equal to Y for the rest of the objective functions. Graphically, this can be seen in Fig. 1, where the round solutions dominate the square ones for two objectives (obj1 and obj 2). Thus, the round solutions constitute the optimal pareto front.

More generally, a pareto front (alternatively pareto frontier) consists of solutions that do not dominate each other. This is the concept of non-dominance which is exploited in the NSGA and NSGA-II.

***2.2. Multiple Traveling Salesmen Problem***

As described earlier, the MTSP involves *m* salesmen visiting *n* cities exactly once. While there are multiple variations that arise from this point, such as multiple origin cities, different origin and final destination cities, and so on, the problem considered is of the conventional closed circuit with all salesmen originating and finally returning to the same depot, called the SD-MTSP.

Considering the problem in graph form, it can be represented by the undirected graph with *V* being the set of vertices representing cities and *A* being the set of arcs between the cities. There is an associated distance matrix C of dimensions , with each element being the distance between *i* and *j*. Each salesman , travels from city *i* to city *j* and is an indicator of whether *k* travelled the arc between *i* and *j* (if yes,=1; else,=0).The set of constraints on the tours can then be mathematically described as:

|  |  |
| --- | --- |
|  | (1) |
|  | (2) |
|  | (3) |
|  | (4) |
|  | (5) |

where ‘0’ is taken to be the depot city.

Constraints (1) and (2) limit the traversal of each arc to just one time across all salesmen; constraints (3) and (4) ensure all salesmen start and end their tours at the depot; and lastly, constraint (5) is to ensure that no sub-tours excluding the depot are created.

The first combination of objectives is to minimize the total distance travelled by all salesmen and the difference between the longest and shortest tours, which can be represented by the equations (6) and (7) respectively:

|  |  |
| --- | --- |
|  | (6) |

and,

|  |  |
| --- | --- |
|  | (7) |

where,

The second combination of objectives is (6) and also minimizing the differences of the travel times of each salesman with the average, represented by (8).

|  |  |
| --- | --- |
|  | (8) |

where,

and,

The first combination, i.e., is a well-known SD-MTSP called the MinMax SD-MTSP, while the second is a much lesser known combination without much depth of study in literature.

***2.3. NSGA-II***

The Non-dominated Sorting Genetic Algorithm was proposed by Deb *et al.* [] in 2002 as an improvement to the older NSGA, with the addition of elitism. The basic principle behind the titular fast sorting mechanism is the concept of ‘non-dominance,’ or simply that the solutions in a population be ranked based on their level of non-dominance.

In the algorithm, the solutions which are not dominated by any other solutions are ranked first, with solutions that are dominated by a single other solution being ranked second. Naturally, solutions having the same rank constitute a pareto frontier, and thus, each solution only needs to be compared once with all other solutions in the population, which is the mechanism exploited by the algorithm to speedily sort the solution population and determine elite solutions.

There are two main methods in the NSGA-II: fast non-dominated sorting, and crowding distance assignment, of which the former is described above. The crowding distance of a solution, as described by the authors, is the sum of the differences between each objective function evaluation of the nearest neighbors of the solution. The nearest neighbors are solutions such that they ‘box’ in the solution. Thus for a bi-objective problem, the crowding distance of a solution would be the perimeter of the cuboid formed if the function values of its nearest neighbors were plotted on diagonally opposite vertices.

According to the algorithm, the less crowded solutions, i.e., ones that are farther away from any other would be relatively unique and thus worth preserving more than more crowded solutions in the same non-dominance level. Thus the algorithm discriminates between the solutions first on the basis of non-dominance level, and second (for solutions of same rank), on the basis of crowding distance. This is the diversity preservation and selection mechanism in the algorithm.

While the crowding distance operator acts as a comparison function for the binary tournament selection of parents, it also acts as a way to sort the solutions of the same non-dominance level. In the algorithm, the offspring population is combined with the parent population after recombination and mutation, and the next generation (same size as parent population) is chosen from the best of this combined population, first sorting by non-dominance level, and if there is a front that will have leftover solutions after cutting off the rest, enough solutions in that front are chosen starting from the least crowded.

The steps of the algorithm are: *1)* Initial population formation (size=*N*); *2)* Selection of parents for next generation by binary crowded tournament; *3)* Recombination and mutation to form an offspring population of size *N*; *4)* Parents and offspring are clumped together in a population of size *2N* from which the top *N* solutions are selected via non-dominated sorting and comparison of crowding distance (elitism); and, *5)* The steps *2*-*5* are repeated until a satisfactory number of iterations or until there is no to little change in the pareto-front (convergence).

3. RELATED WORK

Verma, Pant & Snasel [] reviewed the usage of NSGA-II in solving MOCOPs rather thoroughly in their article, selecting papers published up to 2020 from journals with high impact factor. The authors categorized the papers based on the level of modification done to the NSGA-II and on the problem solved (Table 4 in []), from which it could be seen that most papers on the MTSP used hybrid NSGA-II, with others using modified NSGA-II. None used the conventional algorithm as is. In the review, the authors compared the results of using modified/hybrid NSGA-II for the MTSP against other evolutionary and non-evolutionary algorithms, namely Ant Colony Optimization (ACO) and variants, Strength Pareto Evolutionary Algorithm (SPEA-II), Decomposition-based Multi-Objective Evolutionary Algorithms (MOEA/D), and the benchmark setting method CPLEX. The authors concluded from their review that the use of NSGA-II has been increasing over the years, proving the versatility and adaptability of the algorithm, or alternatively, its ease of modification. They also concluded that very few papers included case studies, parametric analysis, and further analysis after obtaining pareto-optimal fronts; and that the vast majority of papers used the NSGA-II for solving bi-objective problems.

Bolaños *et al.* [] use the NSGA-II to solve the version of the MTSP, with a novel stacking selection scheme that was different from the regular algorithm, and use different neighborhood structures as local search strategies, which the authors integrated with the conventional NSGA-II. They tested the novel algorithm on a real-world transportation problem case and the pr75 instance and found that the combination of strategies to be effective.

Shuai, Yunfeng & Kai [] proposed the use of the NSGA-II algorithm with a hierarchical crossover operator, the combined-HGA crossover to solve the MinMax SD-MTSP. The authors tested the algorithm on a variety of instances from TSPLIB and compared with results from [] and the benchmark. The HX operator implemented in this project was adapted from this paper. Where the implementation differs from [] is the mutation operators and the additional focus of the project, which is the comparison of crossover operators, and testing the effects of varying the parameters. It can be seen from the results in Section 5 that despite the similarities between the implementation in this project and [], significant differences exist for the same instances, population size and number of iterations.

Sofge [] described a range of evolutionary computational approaches to solving the MTSP and proposed the use of a neighborhood attractor schema. Alves & Lopes [] investigated the use of GAs in solving mono-objective and multi-objective formulations of the MTSP. [] proposed a partheno-genetic algorithm for the MTSP. Yousefikhoshbakht *et al*. [] modified the ACO for solving the MTSP. [] proposed a Particle Swarm Optimization (PSO) algorithm for the MTSP with added time and capacity constraints. Necula *et al.* [] also use an Ant Colony System to solve the MinMax SD-MTSP, and they also provide the benchmark results for a few instances of the problem from TSPLIB using the CPLEX method in []. These results are the ones used to compare the implementation in this project.

4. METHODOLOGY

***4.1. Design***

*4.1.1. Chromosome Representation*

The encoding of the information such that the crossover operations lead to new solutions in the search space effectively is a major consideration in the design of GAs. Two chromosome representations were considered in the project: a two-string chromosome as shown in Fig. [], and a single-string chromosome as shown in Fig. []. The former had a real-number string consisting of a permutation of the city IDs, and another string denoting which salesman is assigned to the city. The two strings encoded the information about the tours jointly, with each city ID and its counterpart salesman ID in the second string forming a single gene. The second method involved appending a shorter string containing breakpoints to a permutation string (of the city IDs). The breakpoints demarcated each salesman’s individual tours.

While the two-string method worked for all cases, it had the distinct disadvantage of creating a perpetual link between each city and a salesman, which was carried over during the crossover operations, limiting the effectiveness of finding new solutions in the search space. The second was far better at the task, and performed better for all cases, with the optimal pareto fronts obtained always dominating the ones obtained using the first method. Hence, the single-string with breakpoints representation was adopted for the rest of the project.

*4.1.2. Crossover Operators*

As mentioned in previous sections, the crossover operators considered in the project are Partially Mapped Crossover (PMX), Cyclic Crossover (CX), Order Crossover (OX) and Hierarchical Crossover (HX).

The PMX algorithm involves selecting two random breakpoints on a parent chromosome, and exchanging the respective sections contained within the breakpoints between each of the two parents. After the exchange, the resultant offspring are minimally adjusted such that each city appears once in the string, as is required for permutation-based operators. A distinct disadvantage of the operator is that sometimes the children turn out to be exactly the same as the parents, which decreases the effectiveness of the algorithm. In addition, this operation leads to breaking up of at least two, but usually more linkages (arrangement of cities).

The CX operator is based on the principle of finding subsets of cities that are present in both parents by cycling through the parents. In the method, first a starting point (here, the first gene) *a* is chosen on the first parent and pushed into the first child (treating the child as a stack). The gene *b* in the corresponding position on the second parent is located on the first parent. Now, the gene in the corresponding position on the second parent in pushed into the first child. The process is repeated until a cycle is complete, i.e., the gene found on the second parent is the same as *a*. The rest of the genes in the first child are filled with the genes of the first parent which are not already present in the child. The second child is formed similarly but starting from the second parent. Like PMX, this crossover operator also results in some of the children being copies of the parents.

The OX algorithm, like PMX, first involves choosing two random cut-points on the first parent, with the portion in between to be copied onto the first child (in the same positions). Then, the remaining positions are filled by the genes in the second parent that are not yet present in the first child, by iterating from the second cut-point to the end and back from the starting to the first cut-point. The process is similarly repeated for the second child, starting from the second parent with the same cut-points.

The HX method was adopted from [], where the authors called it the ‘combined HGA’ crossover. The general concept, shown in Fig. [], is to first choose a city *k* and add it to an empty list *A*. The city adjacent to *k* (to the right of, unless *k* is at the last position, in which case it is to the left of) in the two parents are compared to see which of them is closer to *k* (as in Euclidean distance). *k* is then deleted from each parent and the closer city is chosen as the next *k*, and appended to *A*. The above steps are repeated until the parents are empty, with the list *A* containing the permutation part of the first child. One of the breakpoint parts of the two parents is copied to the child, thus fully forming the first child. For the second child, the permutation parts of the parents are decoded, adding 0 at the beginning of each individual sub-tour as shown in Fig. []. The steps of forming the first child are followed to get the list *A*, from which any zeroes are removed. To the resultant permutation of cities, randomized breakpoints (*m-1*) are added to form the second child.

*4.1.3. Mutation Operators*

Four mutation operators were considered in the project— Insert mutation, Swap mutation, Invert mutation, and Scramble mutation. All four were used, with each having equal probability of being chosen.

Insert mutation involves selecting a random gene in the chromosome and a location to insert. The selected gene is then removed from its current location and placed at selected location. Swap mutation simply swaps the genes at two random locations in the chromosome. Invert mutation involves selection of a range (two breakpoints) of genes and inverting the order of their placement. Lastly, Scramble mutation involves, similar to Inversion, selection of a range and scrambling the genes’ positions. Each of the mutation operators involves breaking apart linkages in the chromosome, which is effective in producing mutations for permutation-based problems, and especially problems like TSP, where the order is doubly important.

*4.1.4. Overall Algorithm*

The overall algorithm is given in Listing 1 and is mostly self-explanatory. Lines 7 to 11 describe the initial population creation and the first offspring formation. Lines 12 to 27 describe the full evolutionary loop which runs for the set number of iterations. Inside the loop, the combined population of parents and children is sorted (non-dominated sort) and the best added to the next generation in lines 15 to 23. The process repeats after that until the end of the evolutionary process. The optimal pareto fronts obtained from running the program multiple times can be combined, sorted and the best (size of population) of the lot used as the result for that instance. This is a sort of parallel island GA [] strategy to exploit the non-deterministic behavior of the GA. Thus, although the number of times the program is to run is set to 1 (default), it can be set to any number of times. The results were obtained by running the program 10 times for each instance.

***4.2. Implementation***

LISTING 1:

OVERALL ALGORITHM FOR THE PROGRAM

|  |  |
| --- | --- |
| 1: | START PROGRAM [input: *pop\_size*,*n\_iter*,cx,mut\_prob] |
| 2: | Instance←getUserInput() |
| 3: | If Instance == random: |
| 4: | generateRandomInstance() |
| 5: | Else: |
| 6: | readInstanceFromFile() |
| 7: | Population = createInitialPopulation(*pop\_size*=*N*) |
| 8: | assignCrowdingDistance(Population) |
| 9: | Parents = tournamentSelection(Population) |
| 10: | Children = crossover(pairs of Parents) |
| 11: | mutate(Children, mutation\_probability) |
| 12: | For *n\_iter* iterations, do: |
| 13: | Combined\_population = Parents U Children |
| 14: | Fronts = nondominatedSort(Combined\_population) |
| 15: | Next\_gen\_parents = None, i = 1 |
| 16: | While len(Next\_gen\_parents)+len(Fronts[i])<N, do: |
| 17: | assignCrowdingDistance(Fronts[i]) |
| 18: | Next\_gen\_parents = Next\_gen\_parents U Fronts[i] |
| 19: | i = i +1 |
| 20: | End While. |
| 21: | crowdedSort(Fronts[i]) |
| 22: | Remnant = Fronts[i][1:*N*-len(Next\_gen\_parents)] |
| 23: | Next\_gen\_parents = Next\_gen\_parents U Remnant |
| 24: | Parents = tournamentSelection(Next\_gen\_parents) |
| 25: | Children = crossover(pairs of Parents) |
| 26: | mutate(Children, mutation\_probability) |
| 27: | End For. |
| 28: | END PROGRAM [output: optimal pareto front] |

The program to run the algorithm and solve the MOMTSP was written in Python 3 and run on an Intel Core i7-7700HQ CPU (Windows 10 OS). The code was written from scratch, with the rare references to existing code on GitHub, marked clearly in the appended code. The only external libraries used are NumPy, Matplotlib, Pickle, and Tkinter.

The functions written were grouped in a few modules for ease of use, each group based on their overarching purpose. All functions to read user input, configuration file, save and plot data, and other utility functions went into the ‘utils’ module. Also contained in the module was a function to generate random instances for the MTSP., which returned a distance matrix when input with the number of cities, map size, and depot city location (at (0,0) or center of map).

The genetic operators (crossover, mutation, selection) went into ‘genops’, and the functions for offspring generation and overall evolution that called the operators went into ‘evolution’. Similarly, the NSGA-II functions (e.g. non-dominated sorting) and MTSP objective functions each got their own modules. The chromosome(s) were implemented as class objects in a separate module.

Regarding the chromosome, a point to note is that while it is a single string, it was implemented as a class object with the permutation and breakpoint parts as separate attributes. Its other attributes included rank (non-domination level), a list of dominated solutions, and crowding distance.

The other chromosome type was also implemented and can be used by setting the relevant option. This was because, to compare the effect of using each chromosome type, all functions that manipulated the object had to be adapted to both types, and it would be a waste to simply discard half of each function.

The overall control of the program is handled by a ‘main.py’ script, which returns an optimal pareto front and a scatter-plot showing the distribution of the cities when run. If the save option is selected by a user when prompted, the finally obtained fronts are saved in .pkl (pickle) format, so any particular solution can be investigated later if needed. A note regarding the use of the saved fronts— the chromosome class objects from ‘chromosome.py’ in the ‘src’ folder should be included in any scripts that make use of the saved data.

***4.3.Experiment Design***

There were three questions involved in the design of the solver that were interesting enough to design the experiments around: Which crossover operator(s) and mutation probability allow the NSGA-II to converge properly when solving the problems (comparing performance)? How important is the population size to the performance of the NSGA-II, and does this importance change for different crossover operators? How does the implementation, considering the best parameters, compare against solutions in literature and the benchmark, if such exists? To answer the questions, three separate experiments were designed:

The first, vary the crossover operators {PMX, CX, OX, HX} and mutation probability values {0.01, 0.05, 0.08} to select best combination, keeping population size at 100. This experiment was performed using random instances, multiple times to ensure inferences were not incidental. The random instances were generated with depot city at (0,0) and all other cities randomly located in square region with diagonal vertices (0,0) and (500,500).

Second, to gauge the importance of the population size, check the obtained optimal pareto front after 1000 iterations of the algorithm, for each crossover operator, keeping the mutation probability at 0.05. Similar to the first experiment, this was done for multiple random instances, generated similarly. NOTE: Both the first and second experiments were performed for both problem variations.

Lastly, with the parameters decided from the previous two experiments, solve the eil51, berlin52, eil76, and rat99 instances from TSPLIB for MinMax SD-MTSP, i.e., ; and the pr75 instance for . For the eil51 and berlin52 instances, the number of iterations was set to 1500, while for the other instance, it was set to 2500. The former set of results (for the was compared with the benchmark values presented in [] as well as the implementations in []; and the latter compared with the lone implementation in []. Note that the results in [] can actually be treated as a benchmark for most other EAs for the chosen problem instances, as it is significantly better than other algorithms.

5. RESULTS

The results from running the experiments detailed in the previous section are shown and inferenced from in this section, which as such is split into two parts— the first detailing the results of the parametric experiments, and the second presenting how well the implementation performed against the benchmark and/or results from literature.

***5.1. Parametric tests***

The parameters considered for this half of the experiments are the type of crossover operator, mutation probability, and population size. Note that as the obtained pareto fronts are irregular and very different from one another, it is not meaningful to have a metric quantifying how much better one parametric combination is over another. Rather, the results are discussed more subjectively and in the relative terms “slightly”, “moderately”, and “significantly”. Another point of interest besides the level of difference is the observation of which objective function axis the pareto front(s) is oriented towards, and the degree to which that is the case. Such observations provide insight into which objective was emphasized by the combination, which could then be used to hybridize or tune a GA for future problems. NOTE: Since the results for this section of the experiments came out similarly for both MTSP variations, the plots in the report are only for the MinMax SD-MTSP cases.

*5.1.1. Crossover and mutation operators*

The result of running the program with .

*5.1.2. Population size*

From experiments on the population size shown in Fig. [], it is apparent that it has only slight influence on the performance with HX operator, with marginal differences between the extremes of the pareto fronts and towards the center. However, it was noticed that it did have a big role on the performance of other operators (results shown in appendix), which corroborated one of the results in [].

The biggest impact of increasing the population size was not on the convergence of the solutions, but in the time taken per run. The time taken for a single iteration of non-dominated sorting and next generation formation increased by 2.7 times (from 0.22s/it to 0.6 s/it) when the population size was increased from 100 to 200, and less drastically by around twice when further increased to 300. Although, it was expected as the NSGA-II has computational complexity O(MN2), with N being the population size, such long time durations limited the scope of the testing.

***5.2. Benchmark and literature comparison***

Due to space constraints, not all the results are shown in the report, with the remainder available in the appendix. The results shown and discussed here are those that are representative of the rest or deviating from the norm.

*5.2.1. min (F1,F2)*

.

*5.2.2. min (F1,F3)*

.

***5.3. Further discussion***

While the implementation was successful at getting solutions in the neighborhood of the benchmark most of the time, there is a need for a more focused and in-depth parametric study of NSGA-II (with HX) being used to solve the MTSP. This may allow some hitherto unknown relationships between the parameters to come to light, besides being able to see if selecting better parameters could vastly affect the performance of the solver. From the limited experiments done in this project, such a thing could not be identified beyond the fact that changes in the parameters had little to no effect on the program with HX operator and there was some perceivable effect with the other operators. However, [] shows that such studies are not meaningless and could result in improvements.

In the NSGA-II, the crowding distance is sometimes believed to be an insufficient diversity preservation method []. By replacing the mechanism with other diversity preservation methods such as a neighborhood attractor schema [], it may be possible to find out if that is the reason why the program, in all runs with PMX, CX, and to a lesser extent OX, converged prematurely. If that is disproven, then it can be finally concluded that the three crossover operators were after all, unsuitable for the problem, and not due to the failings of the NSGA-II. This is one of the things that could not be done due to lack of time in this project.

On the other hand, a commentary on GAs [] remarks that checking if the envisioned crossover operator is suitable for the problem at hand is probably the more important question to ask rather than if a GA is suitable for the problem. The author states that the crossover operator is the only mechanism in the GA that “by chance bring(s) together good ingredients of a solution to produce something better because they co-occur.” Thus, regardless of the algorithm, the blame for poor performance can be placed on the crossover operators. Indeed, when comparing the HX to the other operators used in the project, it is clear that HX manages to overcome all (if any) other factors to have a performance comparable to the benchmark.

As an aside, while discussing why a particular GA worked or failed, it has to be acknowledged that although GAs are widespread in their use, their effectiveness at somehow identifying characteristics and exploring the search space to find a somewhat optimal solution is mysterious at best. In addition, even when GAs are shown to work for some problem, having a number of runs and combining the results from them to get at the possibly most optimal pareto front is necessary to mitigate the randomness of GAs. This was rather acutely observed in the project, and as such the results shown in this report and appended are some of the best obtained, and not repeatable. It is possible to use a seed and an update mechanism to make the whole program pseudo-random as it were, but that would mean achieving a set of god results would be dependent on what seed is chosen, which seems untenable. It may be better to leave things to chance, which is the basis of GAs.

While the implementation proved that the MinMax SD-MTSP can be optimized satisfactorily using the NSGA-II, the problem itself is a bit simple to be of direct use in the real world. Again considering the drone application, there are at least two possible additional constraints— maximum flying time of each drone, certain locations to be visited compulsorily as recharging spots. Adding such realistic constraints to the implementation or changing the problem would necessitate a reworking of quite a few of the functions, and a re-tuning of the entire algorithm. It may be wiser to look into methods such as restating the problem as a bunch of TSPs with each salesman’s tour considered one TSP (i.e., go from NP-Hard to NP-Complete), transform the problem into the Boolean SAT problem and solve using modern SAT/SMT solvers (adding/modifying constraints is significantly easier in SAT solvers with no need for modifying the program). Although such transformations may incur a significant computational cost, the process could potentially be simpler than modifying an entire program to include the constraints (like in the case of GAs).

6. CONCLUSION

.

REFERENCES

[1].

[2].

[3].

[4].

[5].

[6].

[7].

[8].

[9].

[10].

[11].

[12].