Solving the Set Cover Problem with Serial and Parallel Ant Colony Optimisation

COMPX556 Project Report, 25/10/20

Hannah Trebes 1306378
Connor Fergusson 1299038

***Abstract*—This report outlines our attempts at solving the Set Cover Problem using Ant Colony Optimisation (ACO). We developed three versions of our code, one to run a serial implementation of the ACO, and two to run parallel versions of the ACO. The first parallel version of the code allows for the running of each ant in a generation to run simultaneously and the second parallel version can run the ants in a generation simultaneously as well as running multiple colonies at the same time. As expected, we found that in general the parallel implementations were able to run to completion faster than the serial implementation.**

I. Introduction

For this investigation into parallel implementations of metaheuristic algorithms, we decided to study Ant Colony Optimisation (ACO). This metaheuristic was first proposed in 1996 [1] and is modeled after the behaviour of Argentine Ant path following in nature [2]. We know from previous studies [1] [3] that ACO can be applied in serial and parallel. As the ACO was designed to solve combinatorial problems [1], we decided to apply it to a problem not mentioned in the initial paper on ACO [1]. After research, we settled on the Set Cover Problem (SCP), which has been previously solved serially and in parallel with the ACO [2] [3]. Our aim was to develop serial and parallel versions of the ACO and compare them on their ability to produce solutions to the SCP.

The Set Cover Problem is an NP-Hard combinatorial optimisation problem [4] [3]. This makes it a perfect candidate for solving by a metaheuristic algorithm like ACO. In the SCP a set S of sets of x is given. The universe U of x is defined as the union of all sets in S . A solution to this instance of the SCP is the minimal subset M of S whose union is equal to U [5] [2]. This problem has been solved with both serial and parallel implementations of ACO in the past [2] [3]

Ant Colony Optimisation is a metaheuristic developed by [1]. It was specifically designed to solve combinatorial optimisation problems, the classic example being the traveling salesperson problem (TSP) [1] [6]. ACO was designed to be robust in that small changes would allow it to solve combinatorial optimisation problems aside from the TSP, and to be amenable to parallel implementation due to its population-based approach [1]. The inspiration for the ACO was the way in which ants - which have poor eyesight - were able to follow the shortest (or near shortest) path between the nest and food. When an ant is traveling, it leaves behind it a trail of pheromones. Other ants are able to detect these pheromones and use their strength to decide which paths to take [7]. The strength of a pheromone trail accumulates as more ants walk the path, and degrades over time as the trails of earlier ants fade away [7] . Modeled after this idea, the ACO uses two factors to influence the decisions of 'ants' deciding which node in the problem to visit next. These factors are attractiveness of the node and the pheromone strength of the node. In the TSP the attractiveness depends on the length of the edge traveled to reach the next node, and the pheromone strength depends on the number of ants that have already traveled that edge [1]. For the SCP, we were interested in the nodes themselves, rather than the paths between them, this meant some alterations to the decision process. In our implementations, attractiveness depends on the number of new elements of the universe to be gained from visiting a node, and pheromone strength is associated with the node itself and the number of times it has been visited.

II. Method

We decided to use Java to implement our ACO as both team members are familiar with it. Also, parallel implementation was required and we knew that this is possible in Java. We defined classes SCP to run the ACO serially, SCPParallel and SCPColonyParallel to run the ACO in parallel . A separate class was defined to represent an Ant. The same Ant class is used by both the serial and parallel algorithms. As the parallel algorithms were developed from the serial, they have the same general structure (except where parallelisation is implemented). There are three main parts to the algorithm. These algorithm phases, and the parallel implementations are discussed in the following sections. It is expected that the serial implementation will run slower than either of the parallel programs and that the implementation with parallel colonies will run slower than that with only parallel generations.

A. The Problem

As discussed in the introduction, the SCP is in finding the smallest subset (the solution) of a set of sets, that is equivalent to the union of all the given sets (the universe). Our program accepts as arguments the name/path of the file containing the problem instance, the size of the Ant colony and the size of the generations to be used to generate a solution. Our program takes the universe and the set of sets as input in the form of a text file (.txt). This file is assumed to be formatted in the following way:

- Each row/line contains a set.
- Each set contains integer elements separated by commas.
- The first row/line is the universe set (union of all sets).

We used a simple 5 set test (taken from [8]) file to develop and test our code. Verbatim. this file contained the values following:

1,2,3,4,5
1,2,3
2,4
3,4
4,5

We chose this as a test file because the optimal solution must clearly be a combination of sets 0 and 3 containing the values 1,2,3 and 4,5 respectively. This known optimal solution allowed us to quickly ensure that over a small problem our program performed as expected. Once our program was

working as expected on this small test set we progressed to a secondary test file containing 10 sets. From here, in order to be able to test and compare our serial and parallel programs, we developed 3 larger test files. The sets in these files were generated using uniformly distributed restricted random number generation in Microsoft Excel. The sizes of these files was chosen with the assumption that every number would occur at least once and our testing proves this is the case. The first of these files (Test3) has a universe of 1-100 with 80 individual sets ranging in size from 2 to 26 elements. The second (Test4) has a universe 1-500 with 163 sets ranging from 5 to 115 elements. The largest file test5 has a universe 1-1000 with 500 sets ranging from 34 to 500 elements. Due to the size of these files and the way they were generated, the optimal solution is not known.

The program reads line by line through the given text file. Commas are used as delimiters to identify the elements within a set. The universe is separated into an ArrayList. For the other sets, the elements are added to an ArrayList, which when it holds the complete set, is in turn added to a 2D ArrayList of all sets (except the universe). The pheromone trail is initialised as a blank ArrayList with an element for each index in the sets. The amount by which the pheromone trail is to be degraded is also set here.

To represent a solution to the SCP we use an ArrayList which contains the indices of all sets (in the set of sets) that one ant has visited. This is dubbed the Ant's Path. Because of this, we discuss solutions in terms of path length i.e. the number of elements in an Ant's path.

B. Creating Colony and Generation

As mentioned previously, the program takes as arguments the sizes of the colony and the generations to be used to generate a solution. Basic input validation checks that the colony size is a multiple of the generation size before allowing the program to continue. The colony of all ants is represented as an ArrayList of Ants of the size specified by the input. Each Ant is responsible for maintaining its own path as an ArrayList, as well as an ArrayList containing all elements of the sets being searched that the ant has seen (seenNumbers) and a boolean flag for when it has seen all elements in the universe. Ants are initialised with empty paths and seenNumbers and the boolean flag set to false.

C. Running Ants

As the ACO describes, all the ants in a generation (of given size) are run through to find a valid solution before the next generation can begin. Each Ant is started on a uniformly randomly selected set/node chosen from the set of all sets. The Ant is responsible for choosing which node it goes to from the current node.

In order to choose this two factors [1] are taken into consideration; the attractiveness of next possible nodes, and the pheromone strength of possible nodes. The attractiveness is calculated on the number of new universe elements to be gained from the move. If the gain is higher, the move is more attractive. There is no penalty for visiting nodes with elements that the Ant has already seen. The pheromone strength depends on previous generations of ants. The pheromone strength of a node is increased each time an ant visits it. As with real ants, the pheromones decay over time, this is achieved by decreasing the pheromone strengths of each node after each generation by a set amount. Pheromone strength cannot go below 0.

In order to retain an element of chance of visiting unseen and/or unlikely nodes, once the attractiveness and pheromone strength of each node have been taken into account, there is an element of randomness inserted into the selection. This randomness is achieved by selecting a random number over a range of numbers. Nodes with stronger pheromones and that are more attractive have a larger portion of the numbers in the range being randomly selected from than those less attractive nodes. This ensures that more attractive nodes are chosen most of the time, but less attractive nodes can still be chosen. An Ant will keep visiting nodes until it has seen all elements in the universe or it has visited all nodes, whichever occurs first.

D. Parallel Generations

Both versions of parallel implementation are implemented using Java's Thread class [9]. The parallel generations program defines a class Generation that extends Thread. This class is used when a generation is run so that all the ants in the generation are run at the same time. The next generation does not start until the previous one has finished.

E. Parallel Colonies

the parallel colonies program takes an extra command line argument from the user, this is the number of colonies to run. This program runs multiple colonies at the same time. Each generation in each colony is also run in parallel. In this program, both Colony and GenerationColony are declared as classes that extend Thread. Once each colony has run through to completion the results are combined to generate the overall performance statistics.

III. Results

Once all of the ants have generated a solution to the SCP given, the programs report their results to the command line. The results that are reported are (in order):

- The Final Pheromone strength at each node.
- How much the pheromones are degrading each generation.
- The first instance of the best path.

- The Length of the best path.
- The number of times the best path (length) was achieved.
- The length of the worst path.
- The number of times the worst path (length) was seen.
- The average length of all paths taken.
- The time taken (wall-clock time) to run all ants, in seconds.

For our discussion of results here, we are interested mainly in the final 6 results presented as these allow comparison between the serial and parallel implementations. The full file of results can be seen in the Github repository linked in the further resources section. Here, we will focus mainly on the results around the best and average solutions, as well as run-time. There is some possibility for bias in these results as some program runs were completed on different machines as a result of availability.

A. Serial Program

Expecting that this would be the slowest of the three methods attempted, A time limit of 1 hour was placed on tests. It was initially planned to test each method with parameters up to 1 million ants in generations of up to 100000. This however was not possible in the case of the serial algorithm due to the imposed time limit.

1) 1-100 Test File: The general results of this experiment can be seen in table I. The worst path achieved had a length of 50 and the best a length of 9. Both the best and worst paths being less than the number of rows in the test file (80) indicates that the problem instance produced a valid, non-equal subset of sets that was equivalent to the universe. This also indicates that each set elements occurred enough times that no ant had to visit every possible node to generate a valid solution. This is encouraging as it means that our decision algorithm that chooses where an ant goes next was working well in favouring more appealing nodes.

An interesting observation from this results table I can be made when considering the run-times compared to the colony and generation size. For colony sizes that were run with multiple generation sizes (colonies of 10000, 100000 and 1 million) as the generation size increased, the run-time decreased.

We can also see that as run-time increases, generally the minimal path length found decreases. This indicates that solutions that take longer to generate may be closer to the (unknown) true global optimal path length.

2) 1-500 Test File: The increase in the size of the input text file to the serial program resulted in a severe decrease in the number of computations that could be completed within the 1 hour time limit. Only three combinations of colony and generation size could be run with this test file without breaching the time limit, the results from these runs can be seen in table II. Even with these limited results the same trends as were observed for the third test file can be seen. With 10000 colonies, the increase from 100 to 1000 ants in a generation can be seen to correlate to a drop in the average run-time by 42 seconds. It can also be seen that as the number of ants in a colony increases the minimal path length appears to get closer to a global optimum.

3) 1-1000 Test File: This test file impacted the run-time performance of the program even more than the second. Here the only combination that could be run without breaching the time limit was for a colony of 1000 and a generation size of 100. Unfortunately this means there cannot be anything inferred from this data on its own. This data may will however still be useful for comparison against the parallel models.

B. Parallel Generations

1) 1-100 Test File: Table IV shows the results from running the generation parallel implementation on the 1-100 test3. Interestingly, this data seems to contradict the observation from the serial tests that increasing the generation size decreases run-time for the same colony size. The best path observed had a length of 10. These results also show a tendency to produce better path lengths as the colony size increases.

2) 1-500 Test File: The results of running this test file can be seen in table V. As it has already been shown that the parallel test run faster than the serial, the number of tests shown in these results is reduced to contain enough to support this idea. Once again the general trend for more minimal paths as colony and generation size increase can be seen. The one example of the same colony size and differing generation sizes (10000 and 100, 1000 respectively)shows a slight decrease in average run-time of almost 5 seconds. It is likely that this difference is insignificant. Interestingly this method did not perform as well as the serial in terms of the solution produced.

3) 1-1000 Test File: For the final parallel generations test the results shown are only those used later in the comparison section. This is because it has already been established that the parallel implementations run faster than the serial. While there are no results to be drawn directly from this data, it supports this idea strongly.

C. Parallel Colonies

An interesting note about the parallel colonies is that the results are generated from multiple colonies. This means that if the program was run with two colonies and the run time was similar to that of the parallel generations program, it would mean parallel colonies is actually significantly faster at generating results as it generated twice the amount of data in the same amount of time.

1) 1-100 Test File: The results for the 1-100 Test file for Parallel Colonies can be seen in table VII. The best path achieved for this was a length of 10, although, on average across all the different colony sizes, generation sizes and number of colonies, the average minimal path length was around 11. The best results were produced when the colony

Number of Runs	Colony Size	Generation Size	Minimal Path Length	Average Path length	Average Run-time (s)
2	10	2	14	16	0
2	10	5	15	17	0
5	100	10	13	17	0
5	1000	100	11	17	1.00
3	10000	100	10	17	12.00
2	10000	1000	10	15	10.00
2	100000	100	10	18	147.00
3	100000	1000	9	14	104.67
2	100000	10000	9	14	95.50
1	1000000	100	10	19	3110.00
1	1000000	1000	9	16	1327.00
1	1000000	10000	9	13	936.00
1	1000000	100000	9	13	924.00

Table I: Serial Average Results Test File 3

Number of Runs	Colony Size	Generation Size	Minimal Path Length	Average Path length	Average Run-time (s)
5	1000	100	21	29	48.2
3	10000	100	20	30	509
2	10000	1000	19	29	467

Table II: Serial Average Results Test File 4

Number of Runs	Colony Size	Generation Size	Minimal Path Length	Average Path length	Average Run-time (s)
5	1000	100	11	16	845.2

Table III: Serial Average Results Test File 5

Number of Runs	Colony Size	Generation Size	Minimal Path Length	Average Path length	Average Run-time (s)
2	10	2	14	16	0
2	10	5	13	17	0
5	100	10	13	18	0
5	1000	100	11	18	1.00
3	10000	100	12	20	5.33
2	10000	1000	12	21	5.5
2	100000	100	11	21	52
3	100000	1000	10	21	53.66
2	100000	10000	11	21	56
2	1000000	100	11	21	620
2	1000000	1000	10	22	590.5
1	1000000	10000	10	22	680.5

Table IV: Parallel Generations Average Results Test File 3

size was larger and the generation size was in the middle. Another trend noticed is that as the number of colonies increases, the results of that test improve. This is because running more colonies produces more results, meaning there is a higher chance for more optimal paths to be discovered. This also works in the opposite direction.

Another trend that can be spotted is the increase in average path length as the colony size is increased. This is most likely due to the same reason that a larger colony produced a lower minimum. It will also produce a higher maximum, as there are more ants running paths, the chance an ant takes a worse route will also be higher, so the average will increase.

The way the run time increases with increasing number of colonies is also interesting. As more colonies are created, the average run time does not increase linearly but slightly less than linearly. A run with two colonies that takes 28 seconds, if working linearly, should take 42 seconds with three colonies. However from the results we can see that one such case actually took 38 seconds, shaving off 4 seconds from predicted time. This most likely means that the extra parallelisation improves run time on a non linear basis.

2) 1-500 Test File: The results of this test can be seen in table VIII. The smallest path length was achieved twice by a colony size of 10000, generation size of 10 with both the 3 and 2 number of colonies producing this result. This result is most likely reflects a working path selection algorithm in the code. As with each generation, more pheromones are deposited onto the better sets, meaning the tests with the most generations possible should produce the best results, which is shown here. This test file showed very similar trends as with the 1-100 test file, with larger colony sizes producing better results but also taking much longer. The trend of non linear run time increases continued as well. It was surprising however, that the longest run time in this test did not occur when the most generations were being run, instead it was with both a large generation size, and a large colony size.

3) 1-1000 Test File: This data can be seen in table IX. The minimal path achieved fro this test file is 11 for the colony

Number of Runs	Colony Size	Generation Size	Minimal Path Length	Average Path length	Average Run-time (s)
2	10	2	25	28	0
2	10	5	25	28	0
5	100	10	23	27	1.6
5	1000	100	22	32	17.6
3	10000	100	22	33	199.67
2	10000	1000	23	34	195

Table V: Parallel Generations Average Results Test File 4

Number of Runs	Colony Size	Generation Size	Minimal Path Length	Average Path length	Average Run-time (s)
5	1000	100	11	16	223.4

Table VI: Parallel Generations Average Results Test File 5

Number of Runs	Colony Size	Number of Colonies	Generation Size	Minimal Path Length	Average Path length	Average Runtime (s)
5	10	2	2	13	16	0
5	10	3	2	13	16	0
5	10	2	5	12	16	0
5	10	3	5	14	17	0
5	100	2	10	13	18	1
5	100	3	10	12	18	1
5	1000	2	10	12	19	3
5	1000	3	10	11	19	3
5	1000	2	100	12	20	2
5	1000	3	100	13	20	3
5	10000	2	10	10	18	22
5	10000	3	10	10	18	33
5	10000	2	100	12	21	21
5	10000	3	100	11	21	28
5	10000	2	1000	12	21	28
5	10000	3	1000	11	21	38

Table VII: Colony Results Test File 3

Number of Runs	Colony Size	Number of Colonies	Generation Size	Minimal Path Length	Average Path length	Average Runtime (s)
5	10	2	2	24	27	0
5	10	3	2	25	26	0
5	10	2	5	24	28	0
5	10	3	5	23	27	0
5	100	2	10	23	29	2
5	100	3	22	22	29	2
5	1000	2	10	22	31	16
5	1000	3	10	22	31	35
5	1000	2	100	22	32	16
5	1000	3	100	22	31	25
5	10000	2	10	20	30	158
5	10000	3	10	20	30	218
5	10000	2	100	21	34	152
5	10000	3	100	22	34	247
5	10000	2	1000	21	34	159
5	10000	3	1000	21	33	248

Table VIII: Colony Results Test File 4

parallelisation. One thing interesting to note is that this value was achieved by a large amount of the different test runs. This would suggest that the test file itself, while there is a large amount universe, has a many different sets that can produce an good result. Even the results that did not match best result came close, with results such as 12 and 13. The average path length for this test file is also very interesting to note, as it is the same accross all tests. This would further back up the suspicion that there is a wide range of sets that produce a good result. One thing to not about this data is that the colony size of 10000 was not included. This is purely because the time it would take to complete is far too long for testing purposes, and thus was left out. Both the serial and generation parallel programs did not include this colony size either, due to a long run time, so it would not be very useful for comparison anyway.

D. Comparison

For the sake of a fair comparison between the result sets, this section mainly considers the results from running 1000 Ants in 10 generations of 100 - a test that was able to be carried out on all programs. For a comparison of the raw run-time results of this series of tests see figure 1.

1) 1-100 Test File: When the colony and generation size is low, all three algorithms complete the process almost instantly, with a run time of zero. However on average, the colony parallelistation produces the best minimal result with an average of 13, while the other two are around 14. Colony

Number of Runs	Colony Size	Number of Colonies	Generation Size	Minimal Path Length	Average Path length	Average Runtime (s)
5	10	2	2	12	15	6
5	10	3	2	13	15	7
5	10	2	5	13	15	9
5	10	3	5	12	15	9
5	100	2	10	11	15	23
5	100	3	10	12	15	35
5	1000	2	10	11	15	194
5	1000	3	10	11	15	284
5	1000	2	100	11	15	170
5	1000	3	100	11	15	259

Table IX: Colony Results Test File 5

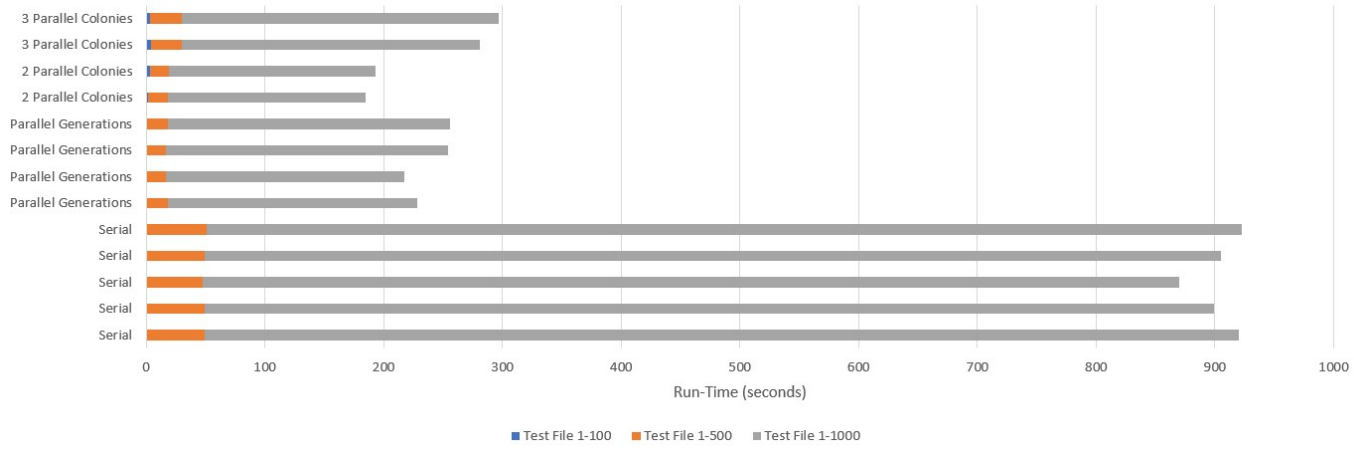

Figure 1: Run-Times of each implementation for the large test files. Colony size 1000, generation size 100

also produces the best average result with 16, whereas the others are both at 17.

Moving up to a larger colony size of 1000 and a generation size of 10, the run times are no longer zero. Colony runs at 3 seconds on average, for a number of colonies of both 2 and 3, while both the other two only have a run time of 1 second. Both the serial and parallel generations versions have the lowest minimal path length as well of 11, while colony parallel has a path length of 12. The same trend is shown in the average path length.

With an even bigger colony size of 10000 and a generation size of 100, we see that parallel generations is considerably fast than the other two programs, with a average run time of only 5 seconds, while serial and colony parallel, with 2 colonies, have run times of 12 and 22 seconds respectively. It must be noted that although the run time is longer for colony parallel, it produces twice the amount of data as there is two colonies being run, so essential it produces one set of data in 11 seconds. Serial produces both the lowest minimal path length and average path as well.

It is worth noting that this is a smaller test file, and therefore more simple programs are more likely to run faster, as there is little room for improvement in speed, and adding complex parallelisation will only slow the program down.

Overall for this test file it seems that parallel generations results in both a more accurate, or at least similarly accurate, result and a faster run time.

2) 1-500 Test File: On the second test file, the serial version of the program runs significantly slower than the other two, and the difference only increases as the colony size increases. With a colony size of 1000 and a generation size of 100, the serial version runs at an extremely slow pace of 48 seconds on average,while producing a minimal path of 21, and an average path of 29. Compare this to the 17.6 second run time of the parallel generations and the 16 second for 2colonies or 35 seconds for 3 colonies and it is far inferior. The path length however is one shorter than both the parallel version 22.

The difference in speed is clearly shown when we increase the colony size to 10000 and the generation size to 100. The average run time of the serial program is 467 seconds, compared to the parallel generations run time of 199.67 and the parallel colony average run-time of 152. We can clearly see that parallel colony is the fastest amount the three, but we must also consider that it is running two colonies at the same time, so we get twice the data in 50 seconds less time. Comparing the path length, we see that the serial version has both a lower minimal and average path length, although not by much. I believe the difference in average path length is caused by outliers that occur in the parallel colony program, it is running more instances of the program, and therefore has a higher chance to have ants take paths that are significantly

longer than other paths.

Overall for test files of this size, I would suggest using colony parallel program as although its average results are worse than the serial version, they aren't too far off, and it runs considerably quicker. The reason to not choose parallel generations is that its results are comparable to colony parallel, but its run times are slower.

3) 1-1000 Test File: This size of this test file led to the serial version of the program only being to run at a colony size of 1000 and generation size of 100. Even a colony size this small produced an average run time of 845.2 seconds, so it was deemed infeasible to run the program with a colony size larger than that, and thus have only tested the other two programs up to this colony value as well. Comparing the three programs at this colony size, all three produce a minimal path length of 11, however while both serial and parallel generation versions of this program have an average path length of 16, the parallel colony version has a consistent average path length of 15. The run times of both the parallel version are significantly faster, with generations having a run time of 223 seconds and colony, with 2 colonies, having a run time of 170. Parallel colonies with 3 colonies has a run time of 259 on average,which, while longer than parallel generations, produces more and better results.

Overall for this test file, the definitive result is that a parallel colony application provides both the best results and the fastest run time, with the serial version not even coming in close, which is to be expected.

IV. Discussion

A. Conclusion

We have found as we expected that in general, parallel implementations of Ant Colony Optimisation to solve the Set cover problem. However, less expected was the variation in the run-time of the parallel colonies approach, and how it was above and below that of the parallel generations. We had thought that as more colonies needed to be created and run the parallel colony approach would be slower than the parallel generation approach. Tested on the smaller files this was the case, but the larger of our test files (ranging from 1-1000) produced run-times faster than the parallel generation when run with 2 colonies, and slower when run with 3. It was no surprise that the serial implementation was slower, in particular over the larger input files. We therefore find ourselves in support of and agreement with the literature [1] and [3] that the Ant Colony Optimisation metaheuristic can be effectively implemented without too much difficulty in a parallel system, as well as serial.

B. Future Direction

There are a few functions that could be added to our programs that would make them more user-friendly. The first of these would be to integrate a maximum run-time as input. This would come in handy if (like in our case) the user wanted to impose a maximum time the program was allowed to run for, after which the current solutions would be treated as final and results would be calculated off of them. Another handy function would be a manual version of this so that the user can at any point in running the program, end it and take the current calculations as final. One final feature that could be added to make analysis of results easier, would be a feature to output the results to a file instead of the command line. This would make the data easier to process and would remove the most manual part of the process.

V. Further Resources

- **Video:** https://youtu.be/vYKH2COqeBc
- **Github repository:** https://github.com/trebesh/556Project.git

References

[1] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29–41, 1996.

[2] R. M. de A Silva and G. L. Ramalho, "Ant system for the set covering problem," in *2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat.No.01CH37236)*, vol. 5, 2001, pp. 3129–3133 vol.5.

[3] M. Rahoual, R. Hadji, and V. Bachelet, "Parallel ant system for the set covering problem," in *International Workshop on Ant Algorithms*. Springer, 2002, pp. 262–267.

[4] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[5] V. V. Vazirani, *Approximation algorithms*. Springer Science & Business Media, 2013.

[6] E. L. Lawler, "The traveling salesman problem: a guided tour of combinatorial optimization," *Wiley-Interscience Series in Discrete Mathematics*, 1985.

[7] E. David Morgan, "Trail pheromones of ants," *Physiological entomology*, vol. 34, no. 1, pp. 1–17, 2009.

[8] u. unknown, "Set cover problem," Sep 2020. [Online]. Available: https://en.wikipedia.org/wiki/Set_cover_problem

[9] u. Oracle, "Class thread," Jun 2020. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html