*i*Journals: International Journal of Software & Hardware Research in Engineering (IJSHRE)
ISSN-2347-4890
Volume 10 Issue 7 July 2022

Comparing Heuristic Efficiencies in Solving the Set Cover Problem

Author: Eric Kim
Affiliation: Seoul International School
E-mail: erickim2281@gmail.com

DOI: 10.26821/IJSHRE.10.7.2022.100509

ABSTRACT

Most combinatorial optimization problems can be solved using various heuristic approaches, but it is often a challenge to determine which is the best option out of the vast choices of heuristics. One such heuristic is the Particle Swarm Optimization (PSO), which is compared to the greedy algorithm in order to determine which is more efficient for solving the Set Cover problem. In order to compare the efficiency of the two algorithms, the runtime and solution of each of the algorithms will be used to find the runtime to solution ratio. Unlike the greedy algorithm, which has no adjustable variables, the PSO algorithm consists of three adjustable variables--maximum generation, maximum fitness, and particle numbers-- that are to be increased in various magnitudes while keeping the ratio between the three variables constant. The code is then run ten times for each magnitude, and the resulting runtime and solution are recorded and averaged. From the recorded data, the time to solution ratio of the two algorithms suggest that the PSO algorithm is more efficient than the greedy algorithm. Although the solution for the PSO was inconsistent for lower magnitudes of variables, it was optimized as the magnitude increased. The efficiency of the PSO algorithm indicates potential for the algorithm to be applied in other combinatorial optimization problems.

Keywords: Combinatorial Optimization Problem, Optimization, Particle Swarm Optimization, Algorithm, Combinatorics, Set Cover Problem

1. INTRODUCTION

The power of computer processors has shown continuous improvement over the years, and according to Moore's Law, it will continue to grow at an exponential rate. However, computers today still face limitations when faced with problems that require complex calculations. Some examples are determining the optimal path for delivery (traveling salesman problem), assigning classes for students (nurse scheduling problem), or decrypting RSA [1]. Even with powerful supercomputers, it would take a great amount of time to solve these problems, as the time complexity is too great for brute-force algorithms. Therefore, the best method is to search for an answer that is not the most optimal, but the most efficient answer within the given time limit. Combinatorial optimization uses heuristics in order to find a solution within polynomial-time [2].

Although optimization problems data back to ancient Greece, combinatorial optimization problems (COP) were introduced quite recently. Leonid Kantorovich first introduced linear programming in 1937, which was soon followed by techniques from George Dantzig and John Von Neumann. Early work from these scientists and mathematicians set a strong foundation for combinatorial optimization, developing different algorithms, such as the genetic algorithm and heap's algorithm [3]. Later, combinatorial optimization problems emerged, which are mainly divided into scheduling, resource allocation, route, and graph problems. COP problems require the use of discrete mathematics, computer science, and probability theory, which has encouraged scientists from a variety of disciplines to learn more about the field.

On May 24, 2000, the Clay Mathematics Institute introduced seven problems, with an award of 1 million dollars for anyone with the correct solutions [4]. The most well-known and arguably the most difficult problem among these seven is the P vs. NP problem, and it is yet to be solved. The problem questions whether a computer that can

check if a problem is easy to solve can also solve the problem itself. Problems that can be solved within polynomial time are considered a group of "P", while problems in which the time to solve grows exponentially is considered "NP" [5]. If N!= P, researchers can focus on finding a partial solution, not the best solution. However, if P = NP, all the NP problems that couldn't be solved will eventually be a part of P, making them possible to solve with computers. Some examples are curing cancer or even breaking the encryption system [6].

Within the P vs. NP problem, there are different classes known as the NP-hard and NP-complete. NP-hard problems represent problems that can be reduced to polynomial time. NP-hard problems are at least as hard as the NP problem because if there is an algorithm for the NP-Hard problem, it can be applied to solve any of the NP problems. If all the NP problems can be reduced to one specific NP problem within polynomial time, the problem is NP-Complete. Approximation and heuristic methods are commonly used to solve NP-Complete problems [7].

One such NP-Hard COP is the set cover problem, in which the objective is to choose the minimum number of sets that contain all the elements from a given input set, while minimizing the costs of the set. For the set cover problem, greedy algorithms, genetic algorithms, and particle swarm optimization are all options for solving the problem [8]. The PSO, which is derived from the flocking of birds, is similar to the GA, which is favored by many, due to its simplicity and versatility. For the set cover problem, both the GA and PSO will be used to get practical solutions and will be compared to see which is more efficient for the task at hand [9].

2. SET COVER PROBLEM (SCP)

The Set Cover Problem (SCP) is a well-known problem in combinatorial mathematics and has a reputation for being hard in literature [10]. The decision version was proven to be NP-complete by Karp et al. in 1972 [11]. It has several applications in real life, one of which is the crew scheduling problem [12]. The main objective of the crew scheduling problem is to find the minimum cost pairings among members of a crew, where the pairings are trips that can be covered by the same crew.

2.1 Definition of the Set Cover Problem

In the SCP, there are a number of sets that belong to a population, and within each individual set, there is a collection of arbitrary positive integers. The objective is to select the minimum number of sets so that the elements inside the chosen sets will cover the entire population. For example, if we have the following sets A, B, C, and D:

A = {1, 5, 3}
B = {2, 1, 3}
C = {4}
D = {1, 2, 3, 5}

The population will be P = {1, 2, 3, 4, 5}, which contains all the unique positive integers from the four sets. Extra places in the sets will be placed with zeros in order to simplify the matrix rows with equal columns.

A possible solution would be to select sets A, B, and C, in order to cover the entire population. However, this is not the minimum number of sets, as selecting sets C and D will still cover the population elements. The difficulty of the problem comes from the fact that there is always a possibility of a better solution.

The formal definition of SCP would be: For a universe U containing n elements, S is a collection of subsets U, $S = \{S_1, S_2, ..., S_k\}$, find a sub-collection s from S that completely covers U [13].

2.2 Modeling the SCP

Let's assume that we have a set $R = \{r_1, r_2, ..., r_m\}$ and set $S = \{S_1, S_2, S_3, ..., S_n\}$ of subsets $S_j \subset R$, where S_j is assigned with positive weight $c_j > 0$. Any subset $S' = \{S_{j1}, S_{j2}, ... S_{jk}\}$ from S is said to be covering set R if $\cup S_{ji} = R$ [10]. Here, minimal covering is when i=1 cost c_{ji} is minimal. Assume a matrix $A = (a_{ij})_{mxn}$ to model SCP as:

$$a_{ij} = \begin{cases} 1, if & r_i \in S_j \\ 0, if & r_i \notin S_j \end{cases}$$

It is assumed that r_i is in one of S_j. x_j is another binary variable where $j = 1,2,3,...,n$:

$$x_j = \begin{cases} 1, \text{ if } S_j \text{ is in covering} \\ 0, \text{ if } S_j \text{ isn't in covering} \end{cases}$$

Therefore, In the SCP, the objective is to cover an m-row, n-column matrix composed of zero or one by a subset, while minimizing the cost. X_j will be 1 if column j (cost $c_j > 0$) is in the

solution and 0 if column j is not in the solution . The SCP can be modeled as:

$$\text{Minimize } \sum_{j=1}^{n} c_j x_j \qquad i = 1,\ldots, m$$

$$\text{Subject to } \sum_{j=1}^{n} a_{ij} x_j \geq 1,\ j = 1,\ldots, n$$

$$x_j \in \{0, 1\} \text{ [10]}$$

2.3 Computational Complexity of SCP

Since the SCP is an NP-hard problem, there are no solutions that can yield the most optimal answer within polynomial time. However, the greedy method can be implemented to find an approximation of the solution within polynomial time. Using the brute-force method, combinations of 1, 2, 3, ..., n can be made, but requires a lot of computational power. As the number of available subsets increases, the time complexity will start to grow exponentially, making it difficult to yield an efficient solution. Therefore, the best approach towards the SCP is to utilize the greedy method or a genetic algorithm, as they can be applied towards minimization problems.

3. GREEDY ALGORITHM FOR SOLVING SET COVER PROBLEM

In essence, the greedy algorithm is only concerned about the (best option/immediate benefits) at any point in time, without consideration of the effects in the long run [13]. For example, when using the greedy approach in a vertex cover or traveling salesman problem, it will purely select the path with the lowest cost, disregarding the future consequences. Even if selecting the lowest cost path leads to a worse outcome, the greedy algorithm fails to see the final outcome. Greedy algorithms have shown great success for a number of problems, but suffer from larger problems with a wider range of data and constraints.

3.1 Application of Greedy Algorithm in SCP

The greedy algorithm fits naturally with the SCP, as the set with the maximum number of uncovered elements can be selected and the elements from the set can be removed from the universe. The process of selecting the next set with the most uncovered elements can be repeated until the universe has been filled with the elements.

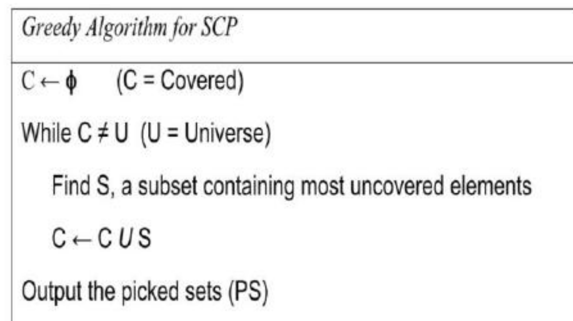


Figure 1. Outline for the greedy algorithm in the Set Cover Problem.

3.2 Time Complexity

Even though the greedy solution for the SCP is within polynomial time, the greedy algorithm is only an approximation algorithm. Therefore, the time complexity is analyzed as an approximation, meaning it won't find the exact time complexity. For the SCP, the greedy algorithm is said to be an H_n factor approximation algorithm, where

$H_n = 1 + 1/2 + \ldots + 1/n$ [14]
Since the total cost of each selected set is distributed among all the uncovered sets, the total cost is equal to
$H_n = 1 + 1/2 + \ldots + 1/n$.

3.3 Comparison with Artificial Neural Network

Multiple variations of the greedy algorithm have been applied to the SCP, as well as methods such as the Artificial Neural Network (ANN). Results have shown that the greedy algorithm outperformed ANN when solving combinatorial optimization problems, while ANN performed better when used for random problems [15].

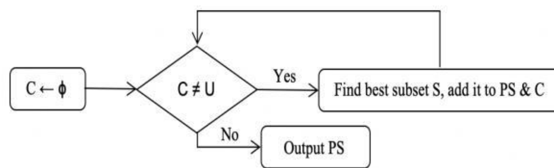


Figure 2. Flowchart for the Artificial Neural Network in the Set Cover Problem.

4. GENETIC ALGORITHMS SOLVING SET COVER PROBLEM

The Genetic Algorithm mimics Charles Darwin's idea of natural selection, since the generations or iterations evolve over time [16]. First, a generation of random agents is initialized in the search space. In order to select the fittest solution for the next generation, the fitness of each of the solutions are calculated. The most fit solutions and other random solutions are crossed over to create the next generation. Just like how certain organisms with an unfavored trait die out, the least-fit solutions are discarded from each generation. After crossing over and discarding some solutions, the next generation consists of the crossover solutions, as well as some of the original solutions from the previous generation. The process is repeated until a stopping criterion is met, which can be the number of generations or the fitness level.

4.1 Swarm Intelligence Algorithms

Swarm intelligence algorithms, just like genetic algorithms, imitate the behavior of organisms to mathematically model and solve complex optimization problems. However, in swarm intelligence algorithms, the positions are measured instead of measuring the fitness of the solutions. A swarm of solutions/agents are first randomly spread across a search space, while their positions are constantly updated based on the flocking behavior of birds. The solutions will go towards the best solution, as the positions of the agents are updated based on the position of the other agents and the best solution so far. Ultimately after a set number of generations, the sufficient solution can be found.

It is important to distinguish the difference between exploration and exploitation. All swarm intelligence algorithms have a good balance between the two. At the start of the optimization process, the algorithm is in an exploration phase, where the agents try to search the entire space as much as possible for promising areas. After the exploration phase, the algorithm shifts towards the exploitation phase. During this phase, the solutions utilize the areas discovered in the exploration phase in order to accurately converge towards the best solution, known as the global best.

4.2 Introduction to Particle Swarm Optimization

Particle Swarm Optimization (PSO) was originally published in 1995 to solve optimization problems and has been constantly used ever since. The PSO is inspired by the swarming behavior of birds and animals, while its evolutionary methods make it relatable to genetic algorithms [17]. Initially, swarms of particles are randomly initialized within the search space of the optimization problem. In each iteration, the location of the particles are updated based on the two major contributors, the Personal Best and the Global Best. The PSO is commonly used for its ease of implementation, along with its small number of parameters to tune and high convergence speed [18].

Since its original publication in 1995, there have been numerous attempts to improve the particle swarm optimization and its performance in specific optimization problems. However, according to the ‘No Free Lunch Theorem for Optimization’ [19], one algorithm cannot have the best performance for all optimization problems, which leads to improvements, hybrids, and variants.

4.3 Components of the PSO

In the particle swarm optimization, a potential solution of an optimization problem is considered as a particle in an n dimensional space. The dimensions of a particle are according to the dimensions of the optimization problem, making it a point in n dimensional space. Within the search space, N number of particles are randomly generated and passed through certain iterations to finally converge at the optimum. In each iteration, the positions of the particles are updated based on

the velocity of each particle and the following factors:

1. Personal Best (pBest)
2. Global Best (gBest)

The Personal Best refers to the best position found by a single particle out of all the iterations. The Global Best, however, refers to the best position found by any of the particles in the swarm from all the iterations so far. Each particle has a pBest, which refers to the position of that particle where its fitness was highest throughout the iterations. The pBest is updated only if there is a new position that is better than the previous best position. Likewise, the gBest refers to the position that has the highest fitness value out of all the particles in the swarm. Here, the best position is defined as the position with the lowest cost or the highest fitness value. The cost can be applicable to minimization problems, while the fitness value can be applicable towards maximization problems. Assigning these nomenclature helps visualize the nature of the problem.

The positions of the PSO particles are based on the following equations:

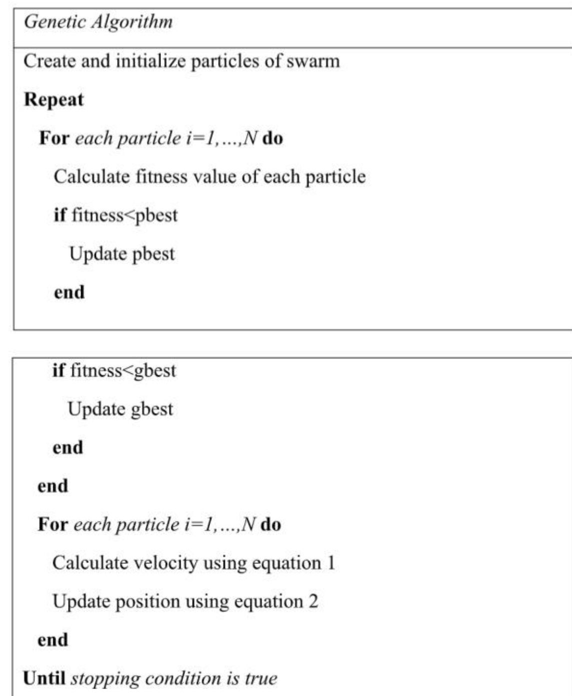
$$v_i^{k+1} = uv_i^k + c_1 r_1 (p_i^k - x_i^k) + c_2 r_2 (p_g^k - x_i^k) \quad (1)$$

$$x_i^{k+1} = x_i^k + v_i^{k+1} \quad (2)$$

Where r_1 and r_2 are random numbers between [0, 1] and u is the inertia weight [20]. c_1 and c_2 are positive constants, which represent the acceleration components [21]. The acceleration components control the learning rate of the algorithm and can also be adjusted to adopt the algorithm according to the applications. The inertia weight controls the velocity of particles, which helps it balance between exploration and exploitation. Without an inertia weight, there will be no control over the particles' velocities, leading to poor optimization performance. v_i^{k+1} is the velocity of the ith particle at time $k + 1$. v_i^k is the previous velocity of the ith particle. p_i^k is the personal best position of particle i, whereas, p_g^k is the global best position of the swarm at time k.

Position is updated in equation 2 using the updated velocity by equation 1. Where x_i^{k+1} is the updated position of particle i and x_i^k is the previous position of particle i.

The factor uv_i^k in equation 1 takes the previous velocity of a particle into account. Factor $c_1 r_1 (p_i^k - p_i^k)$ in equation 1 accommodates the effect of the personal best of a particle in relation to the current position of that particle. Factor $c_2 r_2 (p_g^k - x_i^k)$ in equation 1 adjusts the effect of the global best in relation to the current position of a particle. While calculating the new velocity, the factors above are taken into effect. Equation 1, in contract, simply updates the position of a particle using the updated velocity calculated in equation 1. [18]



Genetic Algorithm

Create and initialize particles of swarm
Repeat
  For *each particle i=1,…,N* **do**
    Calculate fitness value of each particle
    if fitness<pbest
      Update pbest
    end
    if fitness<gbest
      Update gbest
    end
  end
  For *each particle i=1,…,N* **do**
    Calculate velocity using equation 1
    Update position using equation 2
  end
Until *stopping condition is true*

Figure 3. Outline for the genetic algorithm in the Set Cover Problem.

4.4 Application of PSO in Set Cover Problem

Since the PSO is only applicable for optimization problems, we first need to make the set cover problem into an optimization problem. Focusing on the set covering optimization problem, the cost of covering all the elements of the universe must be minimized. A solution of PSO will include a set of zeros and ones, where ones indicate the inclusion of a subset and zeros indicate the exclusion of a subset. Ultimately, the cost is calculated based on the number of subsets that are included to cover the elements of the universe.

Secondly, in order to apply the PSO in the set cover problem, the problem must be made discrete, since the PSO is only applicable for continuous number problems. When transitioning from one generation to the next, we need to apply an estimating mechanism to make the problem discrete.

Working of PSO for SCP

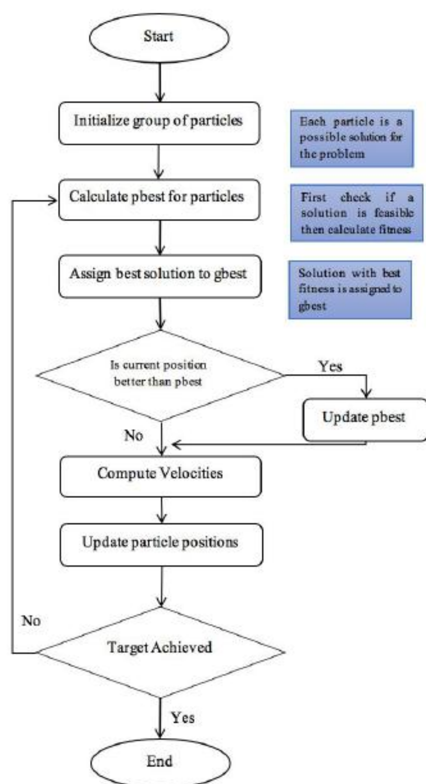


Figure 4. Flowchart for the Particle Swarm Optimization algorithm in Set Cover Problem.

5. Efficiency Analysis

The greedy and PSO algorithms each have their own merits and weaknesses due to their different approach towards solving the problem and the resulting time complexities. It is hypothesized that the greedy algorithm will be more efficient--giving a better time to accuracy ratio--but the PSO algorithm will have a greater accuracy given a longer amount of time to calculate. In order to test which is more effective in terms of accuracy and speed, one small and two large datasets from the *Frequent Itemset Mining Dataset Repository* will be used [22]. The resulting solutions will then be compared with the solutions presented in *Set Cover Algorithms For Very Large Datasets* [23]. The Camera Problem will be the small dataset, which consists of 12 sets and 25 elements in the universe, which is drastically small compared to the other three large datasets, which have sets ranging from 3000 to 50,000 and elements from 100 to 8000.

For the data collection, the computer used was a MacBook Pro (13-inch, 2020) with a 2 GHz Quad-Core Intel Core i5 processor, 16 GB 3733 MHz LPDDR4X memory, and Intel Iris Plus Graphics 1536 MB graphics.

5.1 Camera Problem

"CBS has received the TV contract for this year's Super Bowl festivities. Producers have identified 12 potential camera locations within the stadium. They have also identified 25 stadium areas that may require camera coverage during the pre-game, game, and post-game activities. The camera locations and the stadium areas the camera can cover are given below:

Camera Location	Stadium Areas
1	1, 3, 4, 6, 7
2	4, 7, 8, 12
3	2, 5, 9, 11, 13
4	1, 2, 18, 19, 21
5	3, 6, 10, 12, 14
6	8, 14, 15, 16, 17
7	18, 21, 24, 25
8	2, 10, 16, 23
9	1, 6, 11
10	20, 22, 24, 25
11	2, 4, 6, 8
12	1, 6, 12, 17

Figure 5. Data table for the Camera Problem.

CBS executives are concerned about costs for the production. Consequently, they set an objective of minimizing the number of camera locations used. In seeking this objective, they want at least one camera to be available to cover each stadium area. Camera location 7 is the "blimp," and executives have decided that the blimp will be used because of viewer expectation and fascination with the shots from this location. Stadium areas 1 and 2 are locker

room locations. The viewer interest in football personalities has led the executives to request that at least two camera locations be available to cover each of these areas. Formulate a model to determine the minimum number of cameras needed for coverage" [24].

After running ten trials of the greedy algorithm, the result consistently gave an answer of 8. All ten trials returned a value of [1 0 1 1 1 1 1 1 0 1 0 0], with "1" meaning the set is being used, and "0" meaning the set is not. This was not the most optimal solution, as the total elements in the universe could be covered with only seven sets. Although there were no variables altered throughout the course of the ten trials, there were different runtimes for each trial, which averaged to 0.0017647 seconds.

However, the PSO algorithm had different variables that were altered; namely, the maximum generation, maximum fitness, and particle number. The default variable orientation was [6, 100, 6], and the magnitude of the three variables were increased while keeping the ratio of 3:50:3 constant. There were a total of fifteen different variable orientations with magnitudes ranging from 1 to 50000.

The table in *Figure 6* only shows the variable orientation with magnitudes of 5, 50, 500, 5000, and 50000, with each variable orientation having ten trials. Unlike the greedy algorithm, it can be seen that there were several variants in the results of the PSO algorithm. With a total of 15 different variable orientations, the graph in *Figure 7* shows that the PSO algorithm seems to give more accurate results as the magnitude of the variables increases. Here, the accuracy was measured by checking whether or not the sets chosen could indeed cover all the elements of the universe. While a magnitude of 1 yielded an average accuracy of 30%, that number quickly rose to 100% with a magnitude of 50. The increase in runtime seems to have an upward concavity as the magnitude of the variables increase. However, the data collected is not sufficient enough to prove if the time complexity increases exponentially or linearly.

Comparing the greedy and PSO algorithms, it is evident that the PSO algorithm gives more accurate results given a longer runtime, as it consistently had an accuracy of 100% with a magnitude of 50 or higher, as shown in Figure 2. Instead, by comparing the PSO algorithm that had a similar runtime as the greedy algorithm, it is clear that the greedy algorithm outperformed the PSO algorithm. The PSO algorithm with a variable orientation of 2.5 had an accuracy of 40%, while the greedy algorithm always had an accuracy of 100%. Since the runtime for the PSO algorithm is "unreasonable," meaning the runtime increases exponentially, it is crucial to find a variable orientation that would be within a reasonable timeframe. A reasonable variable orientation for the algorithm could be the one with a magnitude of 50, which has an average of 0.15105425 seconds. In such problems with smaller sizes and numbers of sets, the accuracy of the results is much more important than the runtime; therefore, it is important to keep the accuracy at 100%. Moreover, having a variable orientation greater than 50 is unnecessary, since the accuracy always remains at 100%.

Table 1. Data results for the Camera Problem with limited variable magnitudes.

Variable Magnitude	x1 (Default)	x10	x100	x1000	x10000
Trial	Solution	Solution	Solution	Solution	Solution
1	110011100101	111101010100	101101010101	101101010101	101111010100
2	100001111000	111101010100	111101010100	101111010100	101111010100
3	010111101101	101111010100	101111010100	101101010101	101101010101
4	000011101101	011111010100	101101010101	101101010101	111101010100
5	010000001101	101111010100	101111010100	011111010100	111101010100
6	101111010110	111101010100	101101010101	111101010100	101111010100
7	011111010110	011111010100	011111010100	101111010100	101101010101
8	101111010100	111101010100	101101010101	101111010100	101101010101
9	011111010110	101101010101	101101010101	111101010100	011111010100
10	101111010100	101111010100	101101010101	101111010100	101111010100

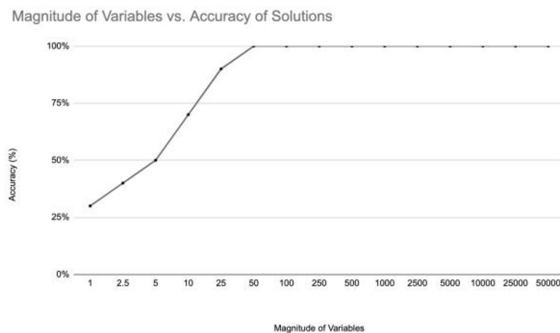


Figure 6. Graph for the relationship between magnitude of variables and accuracy of solutions for Camera Problem.

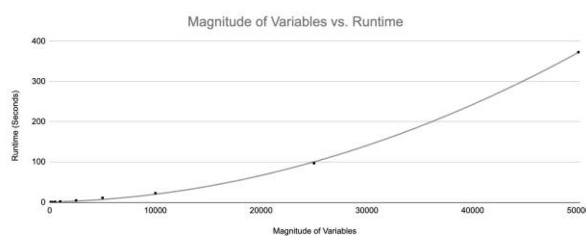


Figure 7. Graph for the relationship between magnitude of variables and the runtime (seconds) for Camera Problem.

5.2 Chess.dat

Chess.dat contained a total of 3196 sets, each with 37 elements. The elements ranged between 1 and 75. The file size was 334 Kb.

After ten trials of the greedy algorithm, the result always yielded a solution of 38 with an average runtime of 0.0803431 seconds. This result varied slightly from that of Cormode, as he got a solution of 26 and a runtime of 0.05 seconds using the naive greedy approach. Nonetheless, there was still a major difference when compared to the most optimal solution of 8.

Similar to the Camera problem, data for the PSO algorithm was collected by running ten trials for each magnitude. The default variable orientation was (50, 50, 4) with the numbers representing the maximum generation, maximum fitness, and the particle numbers respectively. The magnitude of the variables ranged between 0.5 to 10000.

As shown by the graph, the average solution was initially 8.4 with a magnitude of 0.5. Although this was already significantly better than the solution obtained from the greedy algorithm, it meant that the program wasn't yielding an answer of 8 consistently. However, after doubling the magnitude, it consistently gave a solution of 8, which was the most optimal solution for this dataset. As shown by the graph, it is clear that the runtime increased linearly based on the magnitudes.

In order to determine which algorithm was more effective for Chess.dat, the ratio of the solution and runtime from the greedy algorithm will be compared to that of the PSO algorithm. Since the PSO algorithm had different runtimes based on the magnitude of the variables, the runtime from the lowest magnitude that yielded the most optimal solution will be used. In this case, it is the default magnitude of 1. The solution to time ratio for the greedy algorithm will then be 38/0.08, and the ratio for the PSO algorithm will be 8/0.17. Comparing the solution to time ratio of the two algorithms, it concludes that the PSO algorithm was 10.1 times more efficient than the greedy algorithm.

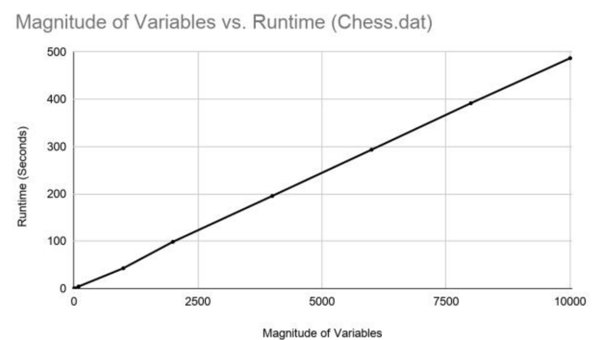


Figure 8. Relationship between magnitude of variables and runtime (seconds) for Chess.dat.

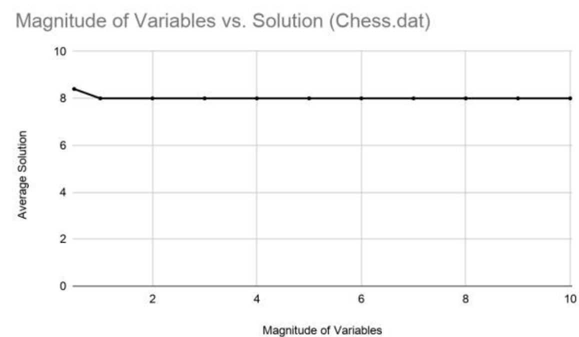


Figure 9. Relationship between magnitude of variables and average solution for Chess.dat.

5.3 Mushroom.dat

Mushroom.dat contained 8124 sets that each contained 23 elements. The elements ranged anywhere from 1 to 115. The file size was 557 Kb. Similar to the previous datasets, the greedy algorithm consistently yielded the same answer of 46 and an average runtime of 0.11 seconds. This result was very similar to the results presented by Cormode, since his result for the naive greedy algorithm was 44 and the average runtime was 0.09 seconds. However, there was still a major difference when compared to Cormode's most optimal answer of 22.

For the PSO algorithm, the default variable orientation was (50, 50, 4), with each number representing the maximum generation, maximum fitness, and particle numbers. The magnitude ranged between 0.5 and 10000 and each of the variables were adjusted accordingly. Data was collected by running ten trials for each of the magnitudes and the solutions and runtime were averaged. Once again, the runtime increased linearly based on the magnitude of the variables as seen in the graph.

As shown in Figure 12, the PSO algorithm initially gave higher solutions for lower magnitudes of the variables, but the solution centered towards 19 as the magnitude increased. In fact, the algorithm consistently gave a solution of 19 after increasing the magnitude by a factor of 4. Comparing the results, 19 was an even more optimal result than the results from Cormode, who used various versions of the greedy algorithm.

Similar to how the efficiency of the PSO and greedy algorithms were compared in Chess.dat, the runtime to solution ratio of each algorithm will be compared. For the PSO algorithm, the lowest magnitude that resulted in the most optimal answer was 4 and the respective runtime was 0.68 seconds. Dividing 19 by 0.68 yields a ratio of 27.94. Likewise, diving 46 by 0.11 results in a ratio of 418.18 for the greedy algorithm. After comparing the ratio of the two algorithms, the PSO algorithm was 14.96 times more efficient than the greedy algorithm.

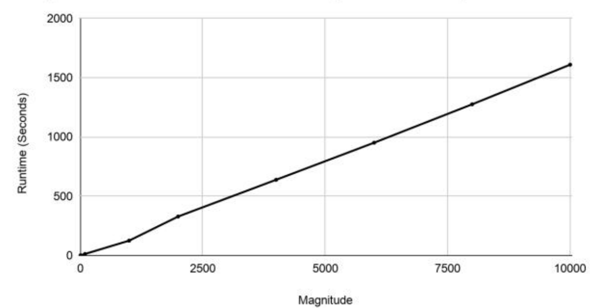


Figure 10. Relationship between magnitude of variables and runtime (seconds) for Mushroom.dat.

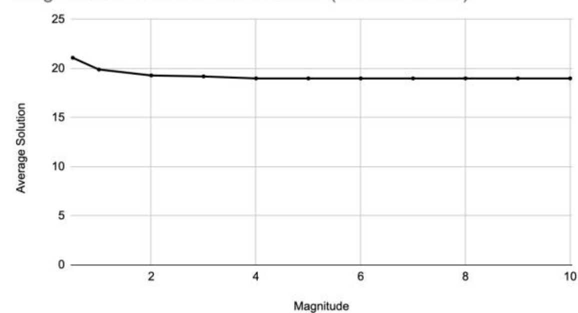


Figure 11. Relationship between magnitude of variables and average solution for Mushroom.dat.

6. CONCLUSIONS

Following the three datasets, the results suggest that the hypothesis was not supported. Although the greedy algorithm outperformed the PSO algorithm in the smaller dataset, the PSO algorithm proved to be more efficient in both Chess.dat and Mushroom.dat. For the camera problem, efficiency was compared by checking the accuracy to time ratio, while the efficiency for the larger datasets were compared by finding the runtime to solution ratio. It should be noted that Increasing the magnitude of the variables past a certain point proved to be unnecessary, as the PSO algorithm yielded the most optimal solution only after a small increase in the magnitude of the solutions.

Overall, the graphs of the different problems indicate a linear relationship between the magnitude of the variables and the runtime for the PSO algorithm. There was an eccentricity for the runtime of the camera problem, as the runtime seemed to increase with some concavity. Since the

camera problem was drastically smaller compared to the other datasets, it was expected that the runtime would increase at a constant rate. Moreover, it was anticipated that the magnitude of the variables wouldn't affect the linearity of the runtime; only the size of the dataset would have an effect on the runtime. There weren't enough suitable datasets for the code to prove this, as the code only performed with datasets of equal sizes.

However, these conclusions are not final, as there were several potential flaws with the experiment. Firstly, the variable orientation of the PSO algorithms have been fiddled with in order to find the default configuration that best suits the PSO algorithm. This means that the PSO algorithm was adjusted to yield the most optimal solutions and any other variable orientations would most likely fall short in terms of efficiency. Secondly, there was no way to confirm whether the results from running the two larger datasets were possible solutions, whether they were the most optimal solution or not. Unlike the camera problem where the numbers were small enough to check manually, the number range in the other two datasets were too large to check in a timely manner. Therefore, it was assumed that the results covered all the numbers in the universe, and the solutions were compared to the results of Cormode.

Provided with more time, the experiment could be improved by running the different datasets with even larger magnitudes of variables to confirm that the runtime does indeed increase linearly. Moreover, a brute-force code could be created to check whether the solutions from running the larger datasets covered all the elements in the universe and was a possible solution. This would help sort out some of the results that would not work, which would make the comparison between the efficiency of the greedy and PSO algorithm more accurate.

With the basic principles of the Particle Swarm Optimization, the PSO algorithm can be applied to other combinatorial optimization problems like the travelling salesman, knapsack, and the minimum spanning tree problem. By applying it to other problems, it is possible to determine what type of heuristic approach is the best in terms of efficiency. Although the PSO algorithm is used far less compared to the greedy algorithm in combinatorial optimization problems, there is potential for the PSO algorithm with proper adjustments of the variables.

7. REFERENCES

[1] Gao, X., Du, H., & Han, M. (2017). Approximation Algorithms for the Generalized Stacker Crane Problem. In *Combinatorial Optimization and Applications 11th International Conference, COCOA 2017, Shanghai, China, December 16-18, 2017, Proceedings, Part I* (pp. 111–112). essay, Cham: Springer International Publishing.

[2] Uppman, H. (2015). On Some Combinatorial Optimization Problems. *Algorithms and Complexity*. https://www.diva-portal.org/smash/get/diva2:806491/FULLTEXT01.pdf. Accessed 10 August 2020

[3] Schrijver, A, Henderson, S. G., & Nelson, B. L. (2005). On the History of Combinatorial Optimization (Till 1960). In *Handbooks in Operations Research and Management Science* (pp. 1–68). essay, Amsterdam: Elsevier.

[4] Cook, S. (2000). The P Versus NP Problem. *Clay Mathematics Institute*. https://www.claymath.org/. Accessed 19 August 2020

[5] Landman, N., Moore, K., & Williams, C. (2016). P versus NP. *Brilliant Math & Science Wiki*. https://brilliant.org/wiki/p-versus-np/. Accessed 19 August 2020

[6] Pavlus, J. (2020, April 2). What Does 'P vs. NP' Mean for the Rest of Us? *MIT Technology Review*. MIT Technology Review. https://www.technologyreview.com/2010/08/19/262224/what-does-p-vs-np-mean-for-the-rest-of-us/. Accessed 25 August 2020

[7] Hardesty, L. (2009, October 29). Explained: P vs. NP. *MIT News | Massachusetts Institute of Technology*. https://news.mit.edu/2009/explainer-pnp. Accessed 25 August 2020

[8] Stern, T. (2006). *What is the Set Cover Problem* (Chapter 2.1, 12). Massachusetts Institute of Technology. https://math.mit.edu/~goemans/18434S06/setcover-tamara.pdf. Accessed 10 August 2020

[9] Shabir, S., & Singla, R. (2016). *A Comparative Study of Genetic Algorithm and the Particle Swarm Optimization*. International Research

Publication House. https://www.ripublication.com/irph/ijee16/ijeev9n2_06.pdf. Accessed 19 August 2020

[10] Chvatal, Vasek. "A greedy heuristic for the set-covering problem." Mathematics of operations research 4.3 (1979): 233-235.

[11] Karp, Richard M. "Reducibility among combinatorial problems." Complexity of computer computations. Springer, Boston, MA, 1972. 85-103.

[12] Caprara, Alberto, Matteo Fischetti, and Paolo Toth. "A heuristic method for the set covering problem." Operations research 47.5 (1999): 730-743.

[13] DeVore, Ronald A., and Vladimir N. Temlyakov. "Some remarks on greedy algorithms." Advances in computational Mathematics 5.1 (1996): 173-187.

[14] Vazirani, Vijay V. Approximation algorithms. Springer Science & Business Media, 2013.

[15] Grossman, Tal, and Avishai Wool. "Computational experience with approximation algorithms for the set covering problem." European Journal of Operational Research 101.1 (1997): 81-92.

[16] Holland, John H. "Genetic algorithms." Scientific american 267.1 (1992): 66-73.

[17] Kennedy, J. and Eberhart, R., 1995, November. Particle swarm optimization. In Proceedings of ICNN'95-International Conference on Neural Networks (Vol. 4, pp. 1942-1948). IEEE.

[18] Ren, J. and Yang, S., 2011, October. A particle swarm optimization algorithm with momentum factor. In 2011 Fourth International Symposium on Computational Intelligence and Design (Vol. 1, pp. 19-21). IEEE.

[19] Wolpert, D.H., Macready, W.G. (1997), "No Free Lunch Theorems for Optimization", IEEE Transactions on Evolutionary Computation 1, 67.

[20] Eberhart R,Shi Y.Comparing inertia weights and constriction factors in particle swarm optimization[C].IEEE Congress on Evolutionary Computation,2000:84-88

[21] JI Zhen, LIAO Hui-lian and WU Qing-hua. Particle Swarm Optimization and Application [M]. BeiJing: The Science Publishing Company, 2009

[22] B. Goethals. Frequent itemset mining dataset repository. http://fimi.cs.helsinki.fi/data/. Accessed 13 November 2020

[23] Cormode, G., Karloff, H., & Wirth, A. (2010, October). Set Cover Algorithms For Very Large Datasets. http://www.dimacs.rutgers.edu/~graham/pubs/papers/ckw.pdf. Accessed 2 January 2021

[24] Bowling Green University. Set Covering Model - Class Examples. https://www.coursehero.com/file/31649465/Set-Covering-handout-1docx/. Accessed 10 November 2020

8. APPENDIX

main.m

```
clear all
clc

Max_Gen= 500;
Max_FES= 500;
Particle_Number= 20;

tic;

%each row indicates a set
% mm = [1 5 3 0; 2 1 3 0; 4 0 0 0;1 2 3 5];
mm=importdata('chess.dat'); %import data from chess.dat or mushroom.dat

[gbest,gbestval,cg_curve,sol_best]=
PSO_func3(mm,Max_Gen,Max_FES,Particle_Number);
sol_best             % best sequence of 1/0
gbestval=sum(sol_best)
           % best result after summation
toc
```

evaluate.m

```
function [fitness,sol] = evaluate3(pos,lenofsol,mm,universe)
    NoE=lenofsol; % total no. of elemnets in sequence
    per=pos; %per  is for percentage
    NoO=round(NoE*per/100); % no. of ones calculated from precentage
    sig=[ones(1,NoO),zeros(1,NoE-NoO)]; % creating a sequence of 1s & 0s
                                        %permutate position of
                                        %above 1s/0s

    for i1=1:1 % loop is optional to try more number of permutations and
keep the best
               % for the timing only one iteration means one permutation
    sol=sig(randperm(length(sig))); %  sol is permutated 1s/0s
    result=zeros(1,length(mm));
    for i=1:length(mm)
        a=0;
        universe1=universe;
```

```
        if sol(i)==1
        a=[a,mm(i,:)];
        a(a==0)=[];
        end
        for j=1:length(a)
            for k=1:length(universe1)
                if(a(j)==universe1(k))
                    universe1(k)=0;
                    result(i)=1;
                    break;
                end
            end
        end
        if(sum(universe1)==0)
            break
        end
    end
    fitness1(i1)=sum(result);
    end
    fitness=min(fitness1);% If first for loop checks multiple premuations
then keep the minimum 1
                           % Not recomended
end
```

PSO_func.m

```
function [gbest,gbestval,cg_curve,sol_best]=
PSO_func3(mm,Max_Gen,Max_FES,Particle_Number)
sol_best=zeros(1,size(mm,1));
universe=unique(mm);
lenofsol=length(mm(:,1));
VRmin=0.23501; % Minimum percentage of 1s
VRmax=2; % Maximum percentage of 1s
rand('state',sum(100*clock));
me=Max_Gen; %Number of generations
ps=Particle_Number; %Number of Particles
D=1;    %Dimensions
        % For any dataset dimension is 1 as we hv taken percentage of 1s
        % in sequence
cc=[2 2];   %acceleration constants
iwt=0.5.*ones(1,me);    %Inertia Weight

%Boundary Check

if length(VRmin)==1
    VRmin=repmat(VRmin,1,D);
    VRmax=repmat(VRmax,1,D);
end
mv=0.5*(VRmax-VRmin);
VRmin=repmat(VRmin,ps,1);
VRmax=repmat(VRmax,ps,1);
Vmin=repmat(-mv,ps,1);
Vmax=-Vmin;
pos=VRmin+(VRmax-VRmin).*rand(ps,1);

for i=1:ps
[e(i,1),sol]=evaluate3(pos(i,:),lenofsol,mm,universe);  % I hv returned sol
as to keep a copy of sequence
end

fitcount=ps;
vel=Vmin+2.*Vmax.*rand(ps,D);%initialize the velocity of the particles
pbest=pos;
pbestval=e; %initialize the pbest and the pbest's fitness value
[gbestval,gbestid]=min(pbestval);
gbest=pbest(gbestid,:);%initialize the gbest and the gbest's fitness value
gbestrep=repmat(gbest,ps,1);
g_res(1)=gbestval;

tmp1=abs(repmat(gbest,ps,1)-pos)+abs(pbest-pos);
temp1=ones(ps,1);
temp2=ones(ps,1);

for kkk=1:D
    temp1=temp1.*tmp1(:,kkk);
    temp2=temp2.*(max(pbest(:,kkk))-min(pbest(:,kkk)));
end

i=1;
while i<me && fitcount<=Max_FES
i=i+1;
    for k=1:ps
    %Updating the Velocities of particles

aa(k,:)=cc(1).*rand(1,D).*(pbest(k,:)-pos(k,:))+cc(2).*rand(1,D).*(gbest-po
s(k,:));
    vel(k,:)=iwt(i).*vel(k,:)+aa(k,:);
    vel(k,:)=(vel(k,:)>mv).*mv+(vel(k,:)<=mv).*vel(k,:);
    vel(k,:)=(vel(k,:)<(-mv)).*(-mv)+(vel(k,:)>=(-mv)).*vel(k,:);
    %Updating Positions of particles
    pos(k,:)=pos(k,:)+vel(k,:);
    pos(k,:)=((pos(k,:)>=VRmin(1,:))&(pos(k,:)<=VRmax(1,:))).*pos(k,:)...

+(pos(k,:)<VRmin(1,:)).*(VRmin(1,:)+0.25.*(VRmax(1,:)-VRmin(1,:)).*rand(1,D
))+(pos(k,:)>VRmax(1,:)).*(VRmax(1,:)-0.25.*(VRmax(1,:)-VRmin(1,:)).*rand(1
,D));
    %Function Evaluation

    [e(k,1),sol]=evaluate3(pos(k,:),lenofsol,mm,universe);
    tmp=(pbestval(k)>e(k));
    temp=repmat(tmp,1,D);
    if pbestval(k)>e(k)
        pbest(k,:)=pos(k,:);
        pbestval(k)=e(k);
    end
%     pbest(k,:)=temp.*pbest(k,:)+(1-temp).*pos(k,:);
%     pbestval(k)=tmp.*pbestval(k)+(1-tmp).*e(k); %updating the personal
best
    if pbestval(k)<gbestval
        gbest=pbest(k,:);
        gbestval=pbestval(k);
        gbestrep=repmat(gbest,ps,1);    %updating the global best
        sol_best=sol;
    end
    cg_curve(fitcount-ps+1)=gbestval;    %Storing current gbest for
convergence curve
    fitcount=fitcount+1;
    if fitcount>Max_FES
            break;
    end
    end

if fitcount>Max_FES
    break;
end

if (i==me)&&(fitcount<=Max_FES)
    i=i-1;
end
end
```