Performance Analysis of Using Mass Extinction Events in Genetic Algorithms

Nadika Bandara
jnbandara@stcloudstate.edu

Liew Jia Hui
hikerliew@gmail.com

ABSTRACT

Genetic algorithms are developed based on how biological evolution works. However genetic algorithms tend to oversimplify mechanisms of biological evolution. Mass extinction events play a major role in biological evolution to increase diversity in a population and change the demography drastically. Implementing mass extinction events in to genetic algorithms could improve the performance and increase the chance of finding global maxima.

Keywords

NP-hard, Combinatorial Optimization, Mass Extinction Events, 0-1 Knapsack problem

INTRODUCTION

Generally improving the performance of a genetic algorithm is a tedious process. This process is usually achieved by changing parameters and adding various mechanisms to the algorithm such as keeping elite chromosomes. Biological evolution itself can offer many interesting mechanisms to improve genetic algorithm performance. “The gaps in the fossil record gave rise to the hypothesis that evolution proceeded in long periods of stasis, which alternated with occasional, rapid changes that yielded evolutionary progress.” (T. Krink and R. Thomsen, 2001). In biology, a mass extinction can be described as an event where most of the organisms in a population vanish suddenly without considering the usual natural selection criteria. Mass extinction events can be experimented in different ways in a genetic algorithm with the goal of improving the genetic algorithm’s performance in finding an optimal solution.

1. Mass Extinction Events

1.1. Background

A mass extinction event is a phenomenon in the biological evolution which happens very rarely that can lead to a drastic change in the population pool of organisms. When a mass extinction event occurs, the population of organisms dilute regardless of their fitness. This can lead to a very small population size with organisms having a wide spectrum of diversity. Eventually the population will regrow. However, the new population might have a different set of organisms with the highest fitness than the population before the mass extinction event. This can be utilized in genetic algorithms to avoid local maxima.

1.2 Previous Work

Research on mass extinction events in genetic algorithms have been done before, however the available resources online are very limited. T. Krink and R. Thomsen’s Self-organized criticality and mass extinction in evolutionary algorithms article explores how mass extension can be used to improve diversity in chromosome populations. Multiple experiments are discussed in the paper which are based on multiple approaches to implement mass extinction events on a genetic algorithm.

2. Problem

0-1 Knapsack problem is chosen to conduct the performance analysis on the greedy algorithm, basic genetic algorithm and a modified genetic algorithm with mass extinction events. 0-1 Knapsack problem is a classical combinatorial optimization NP hard problem. This is a problem that often dealt in many real-world applications. “Many real-world problems involve simultaneous optimization of several incommensurable and often competing objectives” (Zitzler, Eckart, and Lothar Thiele). A simple example to explain this problem would be, suppose there is a bag with a set capacity and number of items with a value and a weight attached to it. The goal is to fit as much as items to the bag without exceeding the weight capacity. The mathematical formula for this problem is as follow

$$\text{Maximum} \quad \sum_{k=1}^{N} w_k x_k$$

2.1. Problem Instances

For this comparison experiment, 2 problem instances of the 0-1 Knapsack problem are used. One of them are from a published dataset and the other instance is randomly generated. The first problem instance which was published by artemisa.unicauca.edu.co has 20 items and the capacity of the container as 879. The weights and values are in the range of 0-100. The optimal solution for this problem should have a total value of 1025.

The second problem set is randomly generated and has 100 items and container capacity set to 20000. The items have value and weight ranging from 0-1000. This problem instance's optimal value was unknown.

3. Greedy Algorithm

A greedy algorithm is a non-evolutionary heuristic algorithm which can find a good enough solution for an optimization problem within a practical amount of time. Greedy algorithms build the solution step by step picking the best element at each step. Greedy algorithms find an approximation rather than an exact solution for the given problem (DeVore, Ronald A., and Vladimir N. Temlyakov).

For the chosen 0-1 Knapsack problem, the high-level design of the greedy algorithm is as follow. All items are sorted in decreasing order of ratio value/weight and insert into the container until the capacity reached. Following is a sample output of the implemented greedy algorithm.

[[46, 4, 4], [75, 14, 14], [61, 25, 9], [78, 32, 18], [40, 18, 12], [90, 43, 3], [77, 56, 11], [8, 6, 6], [55, 44, 5], [63, 58, 13], [91, 84, 1], [75, 70, 16], [72, 83, 2], [75, 92, 8], [29, 48, 15], [40, 68, 19], [35, 82, 7]]

Binary string representation: [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1]

result: [[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1], 1010, 827]

Total value: 1010 Total weight: 827

4. Basic Genetic Algorithm

A basic genetic algorithm is designed to solve the 0-1 Knapsack problem, so this can be eventually modified to an algorithm with mass extinction events. Genetic algorithms can be used to find optimal solutions for NP-hard combinatorial optimization problems effectively. Hence the 0-1 Knapsack problem is a NP-hard combinatorial optimization problem, a genetic algorithm can be applied to find an optimal solution. The algorithm described below was created in order to deal with a problem instances of 0-1 Knapsack problem, where a container with constant capacity C should be filled with n number of objects to have the maximum value V where total value W not exceeding the capacity of the container. Each of these objects has its own value vi, weight w_i, and an index i_i assigned to it. An object is represented as an array of 3 elements.

$$Object_i = [vi, w_i, i_i]$$
$$Object_1 = [54, 23, 0]$$

A candidate solution in the genetic algorithm should represent which objects are chosen and which aren't chosen to fill the container. So, a binary string with 1s and 0s is used to represent a candidate solution s_i. However, a chromosome requires more information such as the fitness of the candidate solution. This algorithm considers fitness as the total value of the selected objects. Altogether a chromosome looks like the following,

$$Chromosome_i = [[s_i], V, W]$$
$$Chromosome_1 = [[1, 1, 0, 0], 9, 4]$$

Evaluation of chromosomes is done to prevent having unstable chromosomes in the population pool. In the case of 0-1 Knapsack problem, solutions which exceeds the capacity of the container cannot exists. For an example, if a problem instance has the container capacity set to $C = 10$, each candidate solution is required to have the total weight equal or below 10.

$$if\ C == 10$$
$$W <= 10$$
$$\therefore\ W <= C$$

This evaluation is conducted by a for loop which compares each chromosome's total weight with the container capacity.

A very important component of a genetic algorithm would be its operations. Genetic algorithms use the Mutation and Crossover mechanisms from Biology as its operations. The mutation operation is supposed to make a small change to the offspring chromosomes to increase the diversity in the population, which will lead to find chromosomes with better fitness than in the previous generations. In this particular algorithm, the mutation operation changes one binary bit to its opposite in a random selection. This is achieved by using a random number generator to select a random index in the chromosome binary string which represents if objects are chosen or not and change the binary value to its opposite.

$$Before\ Mutation = [[1, 1, 0, 0], 9, 4]$$
$$After\ Mutation = [[1, 0, 0, 0], 5, 2]$$

As illustrated in the example above, when the mutation changes the selection of objects in the offspring chromosome, the total value/fitness and the total weight change accordingly.

The goal of the Crossover operation is also to increase the diversity in the population with parents who has better fitness. This operation brings hereditary traits from one generation to another which can increase the fitness of the overall population by time goes on. This particular algorithm gets a half of a parent chromosome and combine it with another half of another parent chromosome to conduct the crossover operation and generates an offspring chromosome as the result.

$$Parent_1 = [[1, 1, 0, 0], 4, 4]$$
$$Parent_2 = [[1, 0, 1, 0], 8, 4]$$
$$Offspring = [[1, 1, 1, 0], 10, 6]$$

Before crossover operation to happen, the prerequisite would be to have selected parents which can pass hereditary information. To achieve this, the algorithm sorts the population in each generation in descending order to have the most fit chromosome first. Then the algorithm selects parent pairs from the most fit and conduct the crossover operation on them until the required population size is reached.

Even though in biological evolution there is no halting factor which stops the process of evolution completely when the factor is met, its required to have a halting factory in a genetic algorithm it to be practical. This algorithm's halting factor is the Generation Amount constant. Once the algorithm is reached to the amount of generations initialized, the algorithm will halt and return the most fit chromosome in the population as the final solution. It's also important to initialize the generation amount a high value because few generations wouldn't be enough to find an optimal chromosome.

Overall functionality of the algorithm can be broken down in to few simple steps. First the parameters such as the Population Size and the required instance for the problem is initialized.

```
data = [[4, 2, 0], [5, 2, 1], [7, 11, 2],
       [7, 11, 3]]
container = []
capacity = 10
POP_SIZE = 5
GEN_AMOUNT = 1000
```

Then the initial population is generated which consists random chromosomes. Once the initial population is generated, it runs through a loop until the number of generations is met. Inside the loop there is another loop to select parent chromosomes, conduct crossover and mutation operations, generate an offspring and evaluate the offspring, and get the offspring to the new generation. The algorithm iterates thought this loop until the new generation meet the desired population size.

```
INITIAL_POPULATION
while (genCount != GEN_AMOUNT):
    while (popCount != POP_SIZE):
        Selection
        Crossover
        Mutation
        New Offspring
        Evaluation
        Append to New Generation
    Old Gen = New Gen
Return Best Chromosome in Population
```

After iterating through the loops, the algorithm returns the chromosome with the highest fitness in the resulting population.

5. Modified Genetic Algorithm with Mass Extinction Events

In biological evolution, mass extinction events happen rarely because of random distractive events. However, when mass extinction events are implemented into genetic algorithms, the frequency of the mass extinction occurrence should be decided on some variable or constant. The tested algorithm considers the amount of mass extinction events needed according to the population size of the chromosomes. Another impotent parameter to consider in the modified algorithm is the generation loop termination condition. This this particular algorithm terminates the generation loop after looping trough the given number of mass extinction events.

The mass extinction algorithm replaces the whole chromosome in the population with a new randomly generated chromosome population when the growth of fitness curves off after multiple generations. This is considered as a mass extinction event hence it drastically changes the whole population dynamics without the consideration of chromosome fitness. This usually ends up with a new population with higher diversity and low fitness. Right before each mass extinction event, the 10% of the fittest chromosomes are preserved. After iterating through multiple mass extinctions, the preserved chromosomes from each mass extensions are evaluated, and the best fit chromosome is selected.

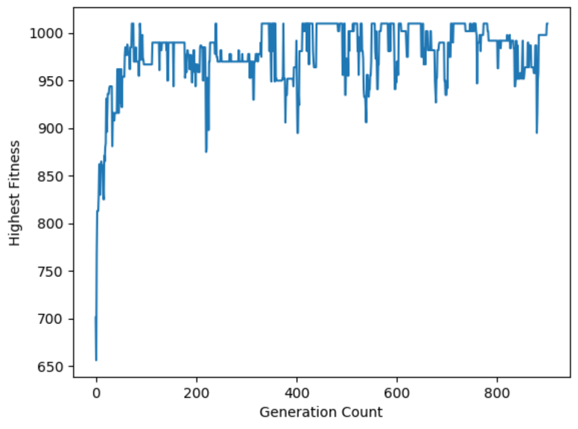


Figure 1: A sample output of the basic genetic algorithm. This graph displays the highest fitness of the population over the cause of generations.

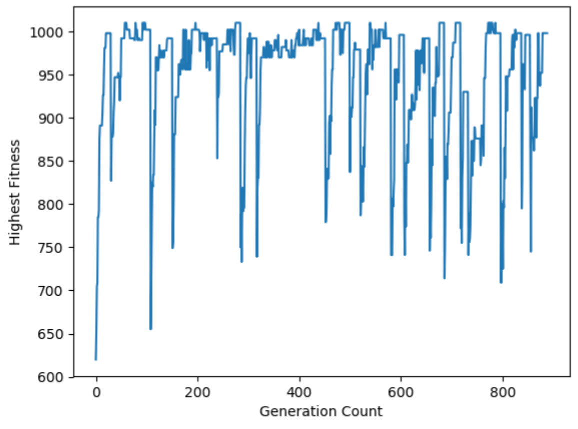


Figure 2: A sample output of the modified genetic algorithm with mass extinction events. This graph displays the highest fitness of the population over the cause of generations.

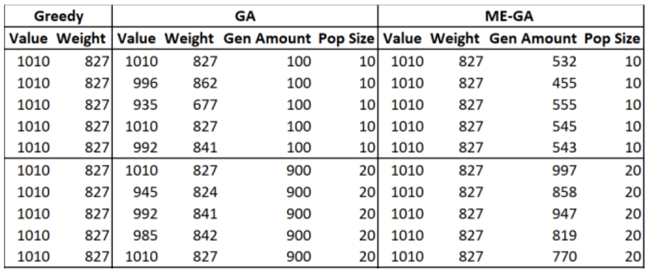
6. Result Comparison

6.1. Testing

Greedy algorithm, basic genetic algorithm and the modified genetic algorithm with mass extinction events were tested on the 2 0-1 Knapsack problem instances described. The testing procedure was to run all three algorithms on the two problem instances multiple times with changing parameters such as the generation amount and population size. The goal was the have optimized parameters for such that probabilities so that the genetic algorithms produced solutions close to the given optimal solutions. Each test case on the three algorithms were run 5 times and there were 2 test cases on each problem instance.

5.2. Assessment

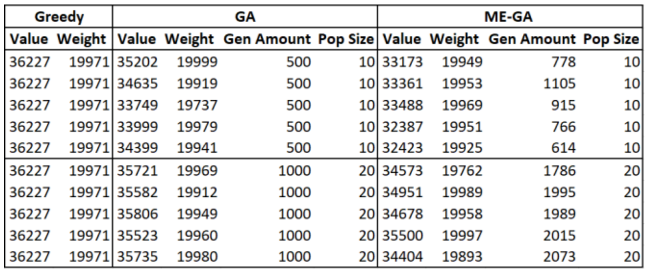
Two separate tables with result data was created in order to procced with the comparison. Following table was created for the problem instance 1 which had 20 items and a container with the capacity of 879



Greedy		GA				ME-GA			
Value	Weight	Value	Weight	Gen Amount	Pop Size	Value	Weight	Gen Amount	Pop Size
1010	827	1010	827	100	10	1010	827	532	10
1010	827	996	862	100	10	1010	827	455	10
1010	827	935	677	100	10	1010	827	555	10
1010	827	1010	827	100	10	1010	827	545	10
1010	827	992	841	100	10	1010	827	543	10
1010	827	1010	827	900	20	1010	827	997	20
1010	827	945	824	900	20	1010	827	858	20
1010	827	992	841	900	20	1010	827	947	20
1010	827	985	842	900	20	1010	827	819	20
1010	827	1010	827	900	20	1010	827	770	20

Figure 3: Result table for the problem instance number 1

When analyzing the able results, it is clear that the Greedy algorithm performed well in both test cases with giving consistent results for each run. The mass extinction genetic algorithm also performed well and had consistent results similar to the greedy algorithm. The basic genetic algorithm had comparable results in some test runs but overall the results were less effective than the other 2 algorithms and not consistent. One issue with the mass extinction algorithm would be using more generations for the first testcase. When the basic genetic algorithm used 100 generations to perform, the mass extinction algorithm used around 500 generations to preform for the first set of test cases. However, for the second set of test cases when the population size was higher, the mass extinction algorithm used lesser amount of generations when compared to the basic genetic algorithm and produced better results consistently.



Greedy		GA				ME-GA			
Value	Weight	Value	Weight	Gen Amount	Pop Size	Value	Weight	Gen Amount	Pop Size
36227	19971	35202	19999	500	10	33173	19949	778	10
36227	19971	34635	19919	500	10	33361	19953	1105	10
36227	19971	33749	19737	500	10	33488	19969	915	10
36227	19971	33999	19979	500	10	32387	19951	766	10
36227	19971	34399	19941	500	10	32423	19925	614	10
36227	19971	35721	19969	1000	20	34573	19762	1786	20
36227	19971	35582	19912	1000	20	34951	19989	1995	20
36227	19971	35806	19949	1000	20	34678	19958	1989	20
36227	19971	35523	19960	1000	20	35500	19997	2015	20
36227	19971	35735	19980	1000	20	34404	19893	2073	20

Figure 4: Result table for the problem instance number 2

Above table was created for the problem instance 2 which had 100 items and a container with the capacity of 20000. It is obvious that the greedy algorithm has performed the best and the mass extinction algorithm has performed the worst in this scenario in most cases. The basic genetic algorithm performed somewhat better than the mass extinction algorithm in this scenario but worse than the greedy algorithm.

CONCLUSION

Above results and observation can lead to an interesting conclusion about using mass extinction events in genetic algorithms. It seems when the dataset or the problem instance is smaller with less diverse data, a genetic algorithm with mass extinction events tend to perform way better than a basic genetic algorithm. However, when the dataset is larger, and the data has a wide range, a mass extinction genetic algorithm wouldn’t perform well.

REFERENCES

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

[2] T. Krink and R. Thomsen, "Self-organized criticality and mass extinction in evolutionary algorithms," Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546), Seoul, South Korea, 2001, pp. 1155-1161 vol. 2.

[3] Zitzler, Eckart, and Lothar Thiele. "Multiobjective optimization using evolutionary algorithms—a comparative case study." International conference on parallel problem solving from nature. Springer, Berlin, Heidelberg, 1998.