A Brief Survey on Genetic Algorithms

Video Link:https://youtu.be/rZWkN9aLrQE

Sheel Taskar, Abdul Samadh Azath

***Abstract*—This literature review provides an overview of genetic algorithm research and applications, a powerful optimization technique based on the principles of evolution and natural selection. The survey covers a wide range of topics, including the history of genetic algorithms, their theoretical underpinnings, and the various types of genetic operators used to generate new solutions. It also investigates the various variations of genetic algorithms that are compared and contrasted based on methodologies suggested by various eminent authors in the field of genetic algorithms. The survey concludes with a discussion of current challenges and future directions in genetic algorithm research, emphasizing the opportunity for additional innovation and advancement in this rapidly evolving field.**

I. Introduction

John Holland and his teams theorized and developed genetic algorithms in the 1960s. Holland laid the foundation of Genetic Algorithms in [Outline for a Logical Theory of Adaptive Systems] in 1962, which suggested treating the entire set of solutions for a problem as a population and proposed the creation of a method by which we can replace the keep better individuals (solutions) and discard other individuals paralleling nature's way of evolution[Darwin reference] and the 'Survival of the fittest' theory. Holland built upon this idea in [1] in 1992 where he detailed how organisms consolidate desirable traits by rearranging their genetic material to survive in dynamically changing and adverse environments. Holland demonstrates a generalized mathematical model which allows us to simulate natural selection, crossover, and mutation to optimize the search for solutions to complex problems.

Since then, GAs have become increasingly popular in computer science and engineering fields, and they are used to solve a wide range of optimization problems, including machine learning, robotics, finance, logistics, and many others. GAs have also played a significant role in the development of evolutionary computation, which includes other computational techniques inspired by natural selection, such as genetic programming and evolutionary strategies.

II. Connection with Biology

In today's world, many problems involve searching an incredibly large space to find an optimal solution to a problem. These problems are typically in the NP-hard class of problems. This set of problems is considered to be computationally intractable, which means it is very unlikely that there is an efficient algorithm that can solve them in polynomial time. The reason for the difficulty in solving NP-hard problems stems from the fact that the solution space for these problems grows exponentially with their size and it is not always computationally feasible to go through all possible solutions to get the optimum one.

There is also a different class of problems for which solutions are too complex to be written by hand. Early AI practitioners believed in using a static set of rules such as the ones used in decision trees to simulate artificial intelligence, but they quickly realized that it will not be possible to model all the rules underlying the intelligence of a model by hand. Instead, they came up with a disruptive paradigm called deep learning where the model developers only write simple rules and the algorithm developed the complex ones by itself when training on data.

GAs aim to use evolution in nature as an inspiration to address these two problems. In general, evolution can be theorized as a culmination of years of development of an organism, generation by generation, which leads to the accumulation of desired traits and rejection of undesired traits resulting in a 'better' individual after every generation. At each generation, there is an enormous set of directions that an organism might take to develop but in the end, through natural selection, only the fittest survive. GAs aim to take a random population of solutions from the solution set and 'evolve' them into more optimal solutions. At each generation, the best solutions are kept and the worse ones are discarded. GAs work on the premise that choosing and crossing over the best solutions in each generation will yield an equally or more optimal solution than the previous generation.

III. Definition

Genetic algorithms are a class of optimization algorithms that are inspired by natural evolution. The basic idea behind GAs is to iteratively generate a population of candidate solutions, evaluate their fitness based on some objective function, and use selection, crossover, and mutation operators to create new candidate solutions for the next generation.

IV. Skeleton of Genetic Algorithms

In this section, we will take a deeper dive into the general framework of GAs. Although there is no thumb rule for the implementation of GAs the basic methodology remains the same through most of the implementations.

A. Genetic Representation

Genetic representation refers to encoding a solution in a way such that it can be used by our GA. Many of the encodings are of fixed length but recently varied length encodings are also

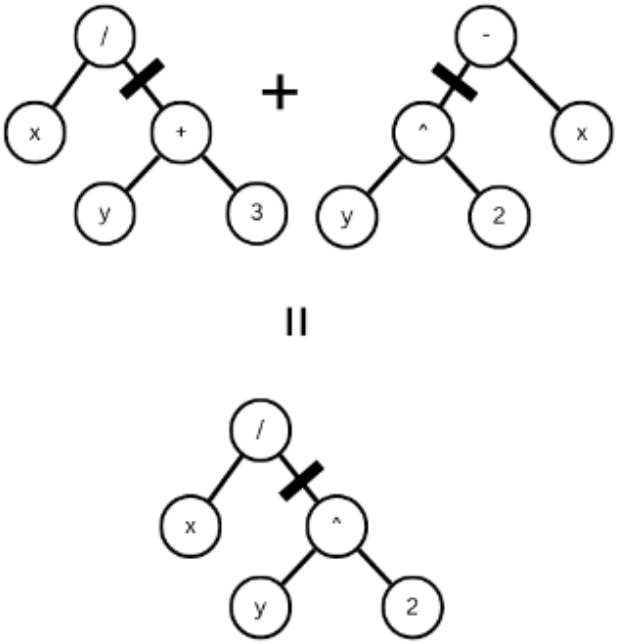

Fig. 1. Tree Encoding for Genetic Representation

being studied. In this subsection, we will go through fixed-length encoding techniques that are commonly used.

1) Binary Encodings:

Binary encodings are fixed-order representations of a solution where every value is either 0 or 1. It is easy to implement but it is not possible for many problem solutions to have such representations. There are also many variations of binary encoding like gray coding as described in [2] and Hillis's diploid binary encoding scheme which is discussed in [3] and [4].

2) Many-Character and Real-Valued Encodings:

Instead of a 0/1 representation of a solution, this encoding uses alphabets and real numbers to represent a solution. According to Holland's schema-counting argument (1975), multi-character encodings should perform worse than binary encodings. However, several studies such as [5] and [6] question them and show better performance for multi-character encodings.

3) Tree Encoding:

Tree encodings are used to encode computer programs and have multiple advantages. John Koza's (1992) representation allows the size of the tree to increase via mutations or crossovers. This provides a robust representation of a solution but can also lead to uncontrolled growth by preventing the formation of hierarchical solutions. There are very few attempts made to extend this encoding format which are detailed in [7] [8].

B. Fitness

A fitness function is used in GA to define how fit a solution in our population is. There is no specific type of fitness function

```
1.Compute the total expected value T of all
individuals in the population.
2. Repeat the following N times:
    a. Select a random value R between 0 and T.
    b. Iterate through the individuals in the
    population and sum their expected values
    until the sum is greater than or equal to R.
    c. Select the individual whose sum of
    expected values is greater than or equal to R.
```

Fig. 2. Pseudocode for Roulette Wheel Sampling

```
/* Returns random number
uniformly distributed in [0,1] */
ptr = Rand();
for (sum = i = 0; i < N; i++)
    for (sum += ExpVal(i,t); sum > ptr; ptr++)
        Select(i);
```

Fig. 3. Stochastic Universal Sampling

and it is usually handcrafted according to the requirement of the problem to be solved. This fitness function is extensively used by the selection process in every epoch to decide which solution is going to make it to the next generation.

C. Selection

Selection is a process of selecting the best individuals from the population to create offspring. We perform selection to discard the weaker individuals and select the fitter individuals so their offspring will also create even fitter individuals in the next generation. In this subsection, we will review multiple selection strategies.

1) Roulette Wheel Sampling:

Here, an individual is allotted a Roulette wheel of size proportional to the its fitness. On each step, a pool of parents is selected by spinning the wheel. This is done for N times, N being the total individuals. The steps for implemention is given in Figure 2 this algorithm. After multiple spins, this method makes it very unlikely that every child will be allocated to the worst parents in the population.

2) Stochastic Universal Sampling:

Introduced by James Baker in [9], instead of spinning the wheel N times, spin it once but with N pointers that are uniformly placed on the wheel for selecting N parents. Figure 3 shows the code in C provided by Baker (1987). Where i is the loop index for populations, ExpVal(i,t) represents the value expected at time t from an individual. This method ensures that the number of times an individual does not exceed ExpVal(i,j).

3) Sigma scaling:

Sigma Scaling attempts to match the fitness function with the expected value. One of the added benefits of doing this is that the in each iteration the number of children allowed to pass to the next generation is relatively constant. In the example of Sigma Scaling detailed in [10], ExpVal(i,t) is a function of the mean and standard deviation of the population as well as the fitness of an individual i. The equation in Fig. 4 shows

$$\text{ExpVal}(i,t) = \begin{cases} 1 + \dfrac{f(i) - \bar{f}(t)}{2\sigma(t)} & \text{if } \sigma(t) \neq 0 \\ 1.0 & \text{if } \sigma(t) = 0, \end{cases}$$

Fig. 4. Sigma Scaling

$$\text{ExpVal}(i,t) = \frac{e^{f(i)/T}}{\langle e^{f(i)/T} \rangle_t},$$

Fig. 5. Boltzman Selection

sigma scaling. Here, f(i) is the fitness of individual i at time t and the latter is the mean fitness of the entire population. The term in denomination is the standard deviation of the entire population.

4) Boltzman Selection:

We have seen earlier that Sigma scaling tries to keep the selection rate constant but in practice, we might need to have some variance of selection. For instance, in the earlier run, we should allow more individuals to pass but in later stages, we should be strict regarding the selection of individuals. Boltzman Selection introduces a term called Temperature which initially is high and gradually decreased. Examples of this approach are given by [11], [12], and [13]. Fig. 5 is an example of Boltzman Selection. T denotes temperature and the (denominator) t denotes the average population at time t.

5) Rank Selection:

Rank selection was first introduced in [14] to prevent solutions from converging very quickly. Individuals are ranked in ascending order of fitness starting from 1 to N. The user selects ExpVal, the maximum of the individual with rank N. The ExpVal is represented in Fig. 6

Min is the expected value of the individual with rank 1 and Max is the expected value of rank N.

6) Tournament Selection:

Tournament Selection is better in terms of computational efficiency as it allows parallel implementation. Just like how a knockout tournament is organized, in this selection, a match is set between two individuals that are randomly chosen from the population, also a for every match a random r is generated between 0 and 1, and the fitter of the two is chosen as the parent if r ¡ k (k is a parameter, usually k is greater than 0.75) else less fit individual is selected if r ¿ k. The analysis of this method is given in [15]. The algorithm is given in Fig. 7. This is repeated for every generation. This is one match in a tournament in a generation, multiple matches can be held parallelly for efficient implementation.

D. Crossover

Crossover is combining the solutions from the process of selection. Crossover functions are of multiple types.

$$\text{ExpVal}(i,t) = Min + (Max - Min)\frac{\text{rank}(i,t) - 1}{N - 1},$$

Fig. 6. Rank Selection

```
Initialize k        # threshold for selection
Parent1 = RandomlySelect(PopulationList)
Parent2 = RandomlySelect(PopulationList)
r = randint(0,1)
If fitness(Parent1) > fitness(Parent2)
    fitParent = Parent1
    unfitParent = Parent2
Else
    fitParent = Parent2
    unfitParent = Parent1
If r<k
    return fitParent
else
    return uniFitParent
```

Fig. 7. Tournament Selection

1) Single-point crossover:

One position is chosen in a solution randomly, and parts of parents before and after the solution are exchanged between parents.

2) Two-point crossover:

This is similar to a one-point crossover but it selects two points randomly and exchanges the segments from the two parents between the two positions.

3) Uniform crossover:

We define a probability and assign a value from either of the parents based on probability to every position of the offspring.

Some practitioners, such as in [16] strongly believe in the supremacy of ”parameterized uniform crossover,” wherein a data exchange occurs with probability p at each bit position (typically 0.5 to 0.8).

E. Mutation

Crossover is exploitative, it takes the best solution from every run but the mutation is more about exploration. It lets us explore new permutations of solutions. Hence, many times a mutation function is added to slightly alter the solution from crossover to try out different solutions. According to Melanie Mitchell, It is not a choice between crossover or mutation but rather the balance among crossover, mutation, and selection that is all important.

V. Review of 0/1 Knapsack Problem using Genetic Algorithms

In this section, we will be solving the 0/1 Knapsack problem using GA. The problem can be solved with Dynamic Programming with the worst-case time complexity of O(N*W). With large values of N and W, the time taken by the algorithm

Fig. 8. Genetic Encoding for 0/1 Knapsack Problem

to execute would be significant. We will review each step involved in solving the Knapsack Problem using the GA approach.

A. Encoding

We begin our implementation by defining an encoding scheme that converts our problem to a numeric representation. We will be using Binary Encoding. We represent selected as represented as 1s, and omitted as 0s. For example, we have eight items, ni, where 1¡= i ¡= 8, and n1, n3, n7, n8 are selected, the binary representation of this solution will be as shown in Fig. 8.

B. Population Creation

We need to define a scheme to "Initialization of population". For optimization, while initializing the population, we can keep checking if our generated solution satisfies the weight constraint, sum(weight of set item) ¡= W. Otherwise, the solution would be invalid. One approach is to start from the left and set a bit as 0 or 1 based on certain probability and at the same time keep a check on weight constraint. But, doing so will skew our solution set to the left. To fix this, we can randomly select and set or unset them according to the probability threshold. The algorithm shown in Fig. 10 describes the initializing process. The following is the pseudocode for population creation.

```
def create():
items_copy = list(items)
random.shuffle(items_copy)
weight = 0
chromosome = 0

for i in items_copy:
    if weight+i["weight"]<=capacity:
        weight += i["weight"]
        chromosome+=2**items.index(i)

return chromosome
```

C. Fitness

A fitness function will score our solution. This function is important as we need to discard solutions that are not "fit" and proceed with the solutions that pass the fitness test. In this problem, we have two goals, one is to maximize the profit and the other is to maximize the weight until the weight limit. The higher the profit, the higher the score should be. The algorithm defined in Fig. 10 defines our fitness function for the Knapsack problem. Fig. 11 plots profits vs score for the fitness function chosen for the knapsack problem. The following is the pseudocode for calculating fitness scores.

```
def score(chromosome):
# assess return weight and value
# of chromosome
weight, value = assess(chromosome)

if weight > capacity:
    return 0.0

return value
```

D. Selection

The selection method we will be using is a Tournament selection. We will take a sample from our generation. Then we will pick a few solutions from the sample and select the solution with the highest score. This type of selection method might need a few more epochs to converge but the number of fitness function invocations would be less. Hence, this should be used when our fitness function is expensive.

```
Step 1: Define an array of size N.
Step 2: Define a probability function
        to output 0 or 1
Step 3: Randomly select any
        uninitialized index in the array.
Step 4: Set the value of the index bit as
        0 or 1, using the probability function.
Step 5: If the bit is set as 1, subtract
        the weight of the item from W.
Step 6: Goto Step 3 if any index remains
        initialized or W > 0.
```

Fig. 9. Algorithm for population creation


```
Step 1: Define an array of size N.
Step 2: Define a probability function
        to output 0 or 1
Step 3: Randomly select any
        uninitialized index in the array.
Step 4: Set the value of the index bit as
        0 or 1, using the probability function.
Step 5: If the bit is set as 1, subtract
        the weight of the item from W.
Step 6: Goto Step 3 if any index remains
        initialized or W > 0.
```


Fig. 10. Algorithm for population creation

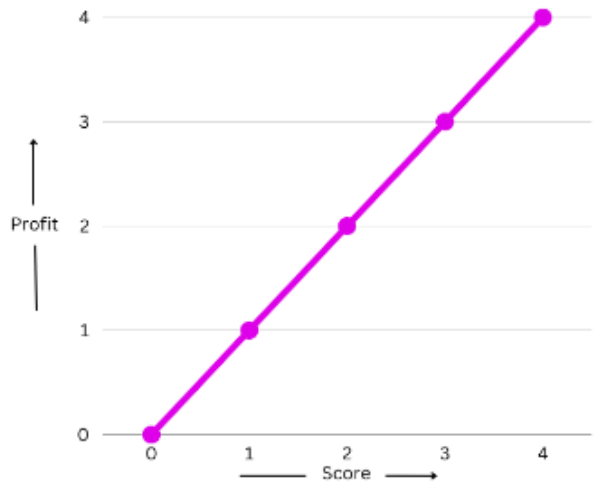

Fig. 11. Profit vs Score for fitness function to solve 0/1 Knapsack Problem

```
1. Initialize a new ChildArray
2. For i = 0 to N
    ChildArray[i] = RandomlySelect(Parent1[i],Parent2[i])
```

Fig. 12. Algorithm for crossover

E. Crossover

We use uniform crossover. For each bit integer at a particular index, we will assign it the value of either of the parent bit integers at that index with 50% probability. This is useful for fast convergence for easier problems. The algorithm for crossover is given in Fig. 12. RandomlySelect returns any of the passed values with a 50% probability. Fig 13 shows the pseudocode for crossover.

F. Mutation

Mutation would be used to change a bit of a child randomly. This kind of mutation is called point mutation and the type of mutation is inversion. The chances of mutation also depend on the probability set in the function. The introduction of Mutation also helps in converging to the solution quickly. Fig 14 shows the pseudocode for mutation.

G. Solution

In each generation, we select the fittest solutions, cross them over, add mutations, and proceed to the next generation. We do this iteratively until our fittest solution starts converging.

VI. Comparison between Genetic Algorithms vs Dynamic Programming to solve 0/1 KnapSack Problem

One way to solve the 0/1 Knapsack problem would be to evaluate all the 2^N combinations of N items that can be included in the knapsack. However, the evaluation of 2^N solutions for large values of N is computationally intractable. But the DP approach to this problem can solve it in O(N*W) where N is the number of available objects and W is the target weight of the Knapsack. However, this is still a pseudo-polynomial time algorithm since the time complexity depends on W.

```
def crossover(parent1, parent2):
    length = chromosome_length
    child = 0

    for i in range(length):
        if random.random() < crossover_prob:
            mask = 2 ** i
            if parent1 & mask:
                child |= mask
            else:
                child &= ~mask

            parent1, parent2 = parent2, parent1

    return child
```

Fig. 13. Psuedocode for fitness score

```
def mutate(chromosome):
    for i in range(chromosome_length):
        if random.random() < mutation_prob:
            mask = 2 ** i
            if chromosome & mask:
                chromosome &= ~mask
            else:
                chromosome |= mask

    return chromosome
```

Fig. 14. Psuedocode for mutation

The experiment was performed with N being 1000 items. Each item had a random profit value between (10,100) and a random weight value between (100,10000). The GA solution had a population size of 10 and each individual of the population is N bit long and a maximum of 100 generations was allowed. The graph in Fig. 15 shows the time taken by both the algorithms with increasing values of W (capacity of knapsack) from 10 to 10000.

As expected the dynamic programming solution time increases linearly with time since it depends on W but the GA solution time stays constant as it only depends on the individual size (N bits), population size (10), and maximum generations (100).

GAs are non-deterministic that may or may not yield an optimal solution, but a defined GA will often converge to the global optimum after a number of generations of evolution. The graph in Fig. 16 compares the optimality of the GA when compared to the dynamic programming solution which would give the most optimal answer every time.

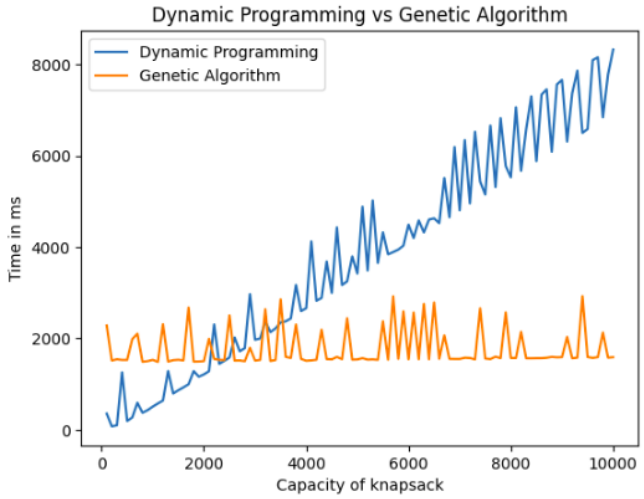

Fig. 15. Time complexity comparison between dynamic programming and GAs to solve 0/1 Knapsack problem

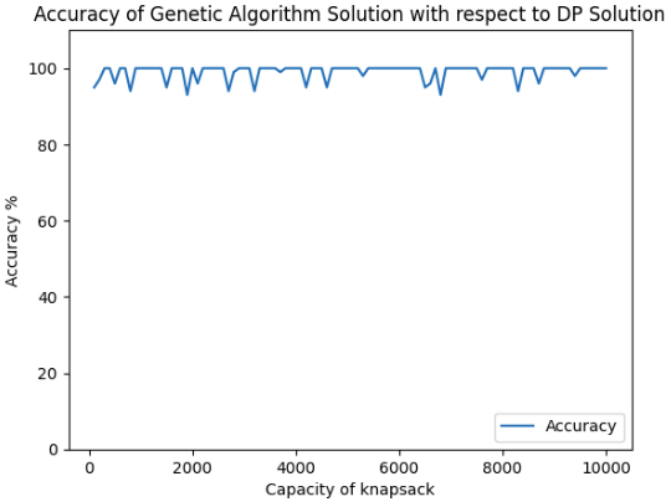

Fig. 16. Comparison of Accuracy of GA solution to Dynamic Programming Solution

REFERENCES

[1] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992.

[2] R. A. Caruana and J. D. Schaffer, *Representation and Hidden Bias: Gray vs. Binary Coding for Genetic Algorithms*, J. Laird, Ed. San Francisco (CA): Morgan Kaufmann, 1988. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780934613644500219

[3] J. H. Holland, *Genetic Algorithms and Adaptation*, O. G. Selfridge, E. L. Rissland, and M. A. Arbib, Eds. Boston, MA: Springer US, 1984. [Online]. Available: https://doi.org/10.1007/978-1-4684-8941-5_21

[4] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," ser. Foundations of Genetic Algorithms, G. J. RAWLINS, Ed. Elsevier, 1991, vol. 1, pp. 69–93. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780080506845500082

[5] J. Antonisse, "A new interpretation of schema notation that overturns the binary encoding constraint," in *Proceedings of the Third International Conference on Genetic Algorithms*, George Mason University, Fairfax, VA, 1989.

[6] C. Z. Janikow and Z. Michalewicz, "An experimental comparison of binary and floating point representations in genetic algorithms," in *International Conference on Genetic Algorithms*, 1991.

[7] W. A. Tackett, "Recombination, selection, and the genetic construction of computer programs," Ph.D. dissertation, USA, 1994, not available from Univ. Microfilms Int.

[8] F. Oppacher and U.-M. O'Reilly, "An analysis of genetic programming," 1995.

[9] J. E. Baker, "Reducing bias and inefficienry in the selection algorithm," in *International Conference on Genetic Algorithms*, 1987.

[10] S. Forrest and R. Tanese, *Documentation for prisoners dilemma and norms programs that use the genetic algorithm*, 1986.

[11] D. E. Goldberg, "A note on boltzmann tournament selection for genetic algorithms and population-oriented simulated annealing," *Complex Systems*, vol. 4, pp. 445–460, 1990.

[12] M. DE LA MAZA and B. TIDOR, "Increased flexibility in genetic algorithms: The use of variable boltzmann selective pressure to control propagation," in *Computer Science and Operations Research*, O. BALCI, R. SHARDA, and S. A. ZENIOS, Eds. Amsterdam: Pergamon, 1992, pp. 425–440. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780080408064500332

[13] A. Prügel-Bennett and J. L. Shapiro, "Analysis of genetic algorithms using statistical mechanics," *Physical Review Letters*, vol. 72, no. 9, p. 1305, 1994.

[14] J. E. Baker, "Adaptive selection methods for genetic algorithms," in *Proceedings of the first international conference on genetic algorithms and their applications*. Psychology Press, 2014, pp. 101–106.

[15] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," in *Foundations of Genetic Algorithms*, 1990.

[16] W. M. Spears and K. A. De Jong, "An analysis of multi-point crossover," in *Foundations of genetic algorithms*. Elsevier, 1991, vol. 1, pp. 301–315.