Stochastic Optimization using the Genetic Algorithm

Simon Austin Lee[*]
Jack Baskin School of Engineering
University of California, Santa Cruz
1156 High St., Santa Cruz 95064
siaulee@ucsc.edu

Abstract

Stochastic optimization methods refer to the use and generation of random variables. This is prevelant in the class of evolutionary algorithms, in which the focus of this paper is specifically that of the *genetic algorithm*, which is used in biology, physics, computer science, and engineering to compute solutions for complex combinatorial optimization problems for which we have no other way to calculate for solutions. In computer science, there remains a major unsolved hypothesis called the *"P versus NP problem"*, which discloses whether every problem whose solutions can be quickly verified, can also be quickly solved for. *P* stands for *polynomial time*, which roughly means a set of relatively easy problems to be solved for. *NP* stands for *non-deterministic polynomial time* which is a set of difficult problems to be solved for. So, if *P* = *NP*, this would imply that the difficult set of problems have relatively easily computed solutions. This theoretical question is the foundation of the *genetic algorithm*, which aims to find the best solution, by the evolutionary process of *natural selection*, at exponentially fast time complexities which would take years to compute if brute-forced. Therefore, we apply this algorithm to our research where we attempt to generate a solution for a *NP-hard* problem in bioinformatics: *The Phylogenetic Tree Construction Problem*.

1 Introduction

Stochastic Dynamical Systems have wide applications in biology, physics, computer science and engineering especially in that of *combinatronics*, which is the area of mathematical studies that focuses on counting to obtain the results and specific properties of finite structures. What makes a dynamical system *stochastic* is its subject to being effected by noise which will be discussed in Section 2.4. So in simpler terms, finding the best combination of *things*, which will be a term used loosely to describe processes from a birds-eye view, can lead to optimized solutions for problems which simply cannot be solved for in any other way.

A clear cut example of such optimization could be seen through the *Knapsack problem*, which given a set of items, their corresponding weights, and a value, the goal is to determine the number of items to include in a collection so that the total weight remains less than or equal to the given limit while still trying to take a value as large as possible. The issues that arises within the *Knapsack Problem*, lies when there is some large *n* integer of items to be chosen from, in which trying to compute all the combinations by hand loses efficiency. This whole idea of efficiency leads to one of the largest unsolved theoretical hypotheses in computer science called the *P equals NP problem* [1], where it asks whether every problem whose solution can be quickly checked for correctness can also be computed quickly. An answer to the *P versus NP* question would open up whether problems that can be verified in polynomial time (*denoted as P*) can also be solved in polynomial time. If it turns out that P $\neq$ NP, which is the more popular belief, it would mean that there are problems in non-deterministic polynomial time (*denoted as NP*) that are harder to solve than to verify: they could not be solved in polynomial time, but the answer could be verified in polynomial time.

Therefore, a solution to the *Knapsack problem* and other time-consuming problems, is the focus of this paper called the *genetic algorithm*, which is widely used in stochastic optimization because it is great at taking large,

[*]Supported in part by the Directed Reading Program of the UCSC Math Department as well as Professor Daniele Venturi of the Baskin School of Engineering, Applied Mathematics Department and Darrell Long of the Baskin School of Engineering, Computer Science Department

potentially massive search spaces and navigating through them, finding the solutions, to problems you might not find otherwise in a lifetime. It takes the possibility of approaching these *NP-hard* problems, and finds a very accurate approximations for them.

2 The Genetic Algorithm

The Genetic Algorithm (GA), is a metaheuristic algorithm inspired by *Charles Darwin's* theory of *Natural Selection*, used for solving optimization and search problems in machine learning. This algorithm's strict purpose is to solve for problems that would take a long time to solve, often producing one of the best approximated solutions. The ability to begin solving *NP* problems is just scratching the surface and with this algorithm there is an expectation for more complex systems and hypotheses to be approached in this manner.

2.1 The Setup

The way the genetic algorithm works is through bits represented in binary showing membership to a set, where 0 is denoted as not being a member of a particular set, and 1 being a member of a particular set. The use of binary is in efforts to reach a goal to emulate a genetic representation of a solutions domain. Before populating a solutions domain, three factors must be determined: how long each bit sequence or *genome* is to represent the n amount of individual *things* or *phenotypes* that are to be selected from, how many possible solutions we want to be randomly produced, and a limit that if succeeded will disqualify it as a valid solution. For example, the singular binary sequence *1001010111* which signifies a single possible solution, shows a bit length of 10 resembling 10 possible phenotypes within a genome, while we see that 6 phenotypes are members of this solution, simply by counting the number of 1's that are present in the sequence. It is important to note that the genome length never increases or decreases because there are a fixed amount of phenotypes present, and that any number of bits can be flipped to a 1 as long as it remains less than or equal to the limit set. Once a limit and integer n are chosen, n amount of unique solutions will be produced of some selected arbitrary length l. This population of random possible solutions are what we refer to as a *generation* and this first generation is called generation 0 (generations can be tracked by *i-1 generations*, where i marks the number of iterations). Furthermore, because all the possible solutions were randomly generated, this evolutionary process begins in complete *chaos*.

Next begins an iterative procedure that will continue to reproduce as long as there are more than one non-zero solutions from the pervious generation. A solution is changed to zero if and only if the total sum of the collection of items succeeds the limit. Before beginning to reproduce even more different solutions, the *Natural Selection* process is introduced, where *fitness, single-point crossovers, and mutations*, all concepts from *Darwin's* Theory of Evolution are assessed.

2.2 Fitness

```python
def fitness(genome: Genome, things: List[Thing], weight_limit: int) -> int:
    '''
    This evaluates a solution and sees its fitness level and whether it succeeds the
        limit that was established
    '''
    if len(genome) != len(things):
        raise ValueError("genome and things must be of same length")

    weight, value = 0

    for i, thing in enumerate(things):
        if genome[i] == 1:
            weight += thing.weight
            value += thing.value

            if weight > weight_limit:
                return 0
    return value
```

In the Python code above, we revisit the *Knapsack Problem* to demonstate each particular component of the *Natural Selection*. To begin, the fitness function's primary purpose is to assess a fitness value (the closer it is to the limit, the better), which essentially tells the person how good of an approximation a given solution actually is. It begins by first checking that the string length of the genome, is equivalent to that of the number of items (phenotypes) present. Next we initializes a weight, and value to 0 so the sum can always be calculated correctly. Finally as it iterates through the for loop, the function will check the bits that have been flipped to 1, and add those weights and value of items up, checking whether it succeeds the limit. The return value of this function is the list of items as long as it complies with the weight limit, where if it succeeds it simply just returns 0.

2.3 Single-point Crossovers

```python
def single_point_crossover(a: Genome, b: Genome) -> Tuple[Genome,Genome]:
    '''
    performs the exchange of n length bits to be swapped to produce new solutions
    '''
    if len(a) != len(b):
        raise ValueError("genome and things must be of same length")

    length = len(a)
    if length < 2:
        return a, b

    p = randint(1, length -1)
    return a[0:p] + b[p:], b[0:p] + a[p:]
```

After evaluating a fitness score, the program will look to perform the single point crossover. In evolutionary biology, gene conversion can be *allelic*, meaning that one allele of the same gene replaces another allele from another gene. Similarly in this Python function, we perform a swap of a random number of bits in the range $1 \leq x \leq$ (length of genome - 1). This will then swap a portion of the two randomly selected parent genomes with one another usually resulting with a better approximation that is closer to the limit. These two newly generated solutions begin what would be the next generation (Generation 1 in our case). We repeat this process as long as there are non-zero solutions from the previous generation to complete this new generation.

Additionally, whats so fascinating about this algorithm is that in both nature and computers, it simulates and generates better offspring/solutions from generation to generation which models the *Natural Selection* behavior almost perfectly. However, this algorithm unfortunately is not perfect because our single-point crossover and selection functions are governed by randomness. The reason randomness is an issue is because there is no way to guarantee that we won't destroy our best solutions produced from each preceding generation. To resolve this issue, we must introduce a process called *elitism* [2]. Elitism can be described as, the most fit (having a fitness score close to the limit) handful of solutions are guaranteed a place in the next generation - without undergoing mutation. So, in order to preserve the most fit solutions, we must select another self selected number of n-top solutions, whose genomes will be copied into is successive generation.

2.4 Mutation

```python
def mutation(genome: Genome, num: int = 1, probability: float = 0.5) -> Genome:
    '''
    randomly flips a bit within a genome to behave like a mutation
    '''
    for _ in range(num):
        index = randrange(len(Genome))
        genome[index] = genome[index] if random() > probability else abs(genome[index] -
            1)
    return genome
```

At last, we have *mutations*. Mutations in evolutionary biology are simply a change in a sequence of an organism's DNA. However, since we are not working with nucleotide bases but rather with bits, we must also familiarize ourselves with *noise*, a concept that dates back to an interests of Einstein (1905). In the communication domain, noise (unwanted random disturbances) make it difficult to have a trivial signal. These fluctuations occur when there is a suspected origin that implicates the action of a very large number of "degrees of freedom" or variables. The coupling of noise to nonlinear dynamical systems can lead to non-trivial effects like, unstable equilibria and shift bifuracations [3]. Therefore it is widely believed that noise drastically modifies the deterministic dynamics of this system adopting its stochastic qualities. So in our case, noise behaves similarly to mutations where a singular random bit might get flipped. The way we simulate this behavior on a computer is by random probability. In the Python function above, we run a single iteration of the for loop, with a probability of your choice, in our case 0.5 to determine whether a random bit gets flipped. Though it may be possible to destroy one of our best solutions, mutations are essential to evolution. The new genetic variant (allele) spreads via differential reproduction and is a defining aspect of evolution. So although there is some random probability to destroy our best fit genome, there is also an equal chance that we construct an even better solution.

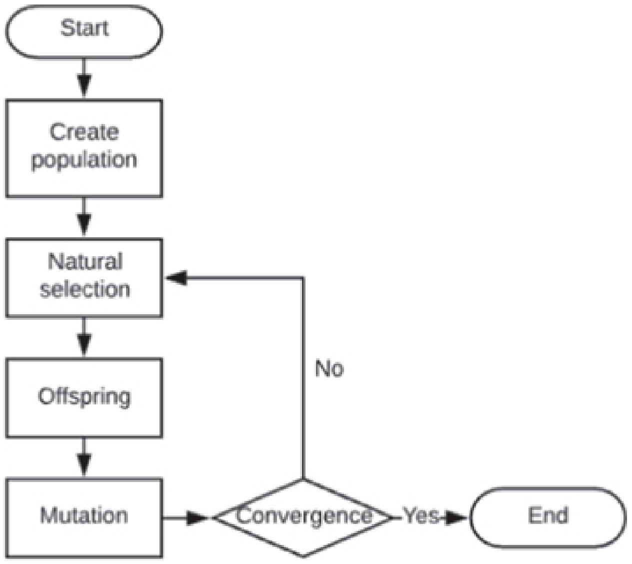

Figure 1: A flowchart of the Genetic Algorithm [4]

After the completion of this whole process as shown in Figure 1, this whole algorithm iterates from the top with the new generation until there are no satisfying solutions or if we have reached a extremely close approximation after a maximum number of generations. We will now begin to look at how this algorithm can be applied to our research, on the *Phylogenetic Tree Construction*.

3 The Phylogenetic Tree

The *Phylogenetic Tree* are important representations constructed in many areas of biological research, ranging from systematic studies to the methods used for genome annotation [6]. This method has been used by bioinformaticians, to identify disease causing microbes, and determine its origin, how it might have spread, and routes of transmissions. In fact, the 2020 *Severe Acute Respiratory Syndrome Coronavirus 2* (SARS-CoV-2), and its variants can be well tracked using the tools from phylogenetics, where each viral nucleic sequence can be analyzed to determine how it mutated based off of both the parent sequences and in a broader scale the *SARS-CoV-1*. By identifying the sequences of these variants, they can then be classified to be more dangerous by its characterstics/behaviors and whether they can forgo the *mRNA* vaccines used by *Pfizer BioNTech* and *Moderna*.

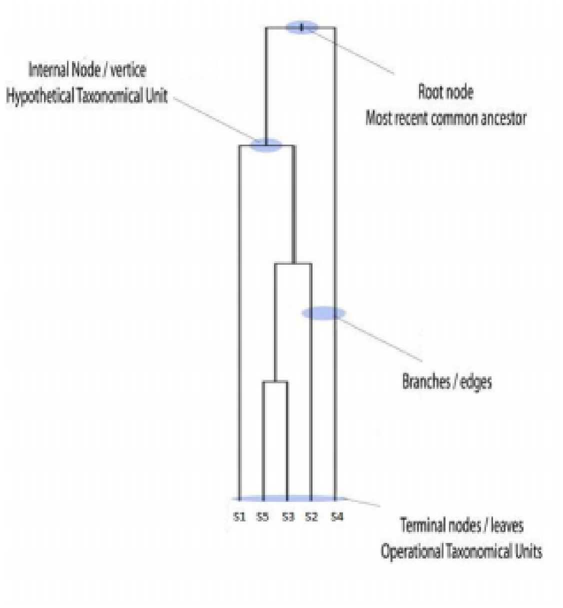
Internal Node / vertice
Hypothetical Taxonomical Unit
Root node
Most recent common ancestor
Branches / edges
S1 S5 S3 S2 S4
Terminal nodes / leaves
Operational Taxonomical Units

Figure 2: A visual representation of the *Phylogenetic Tree and its components* [5]

The basic components of a phylogenetic tree as represented by Figure 2, contains branches, also called edges that are connected to and terminate nodes or vertices. Branches can be classified in one of two ways: internal or external (terminal). These terminal nodes that are found at the tips of trees represent *operational taxonomic units* also reffered to as OTU's. The essence of an OTU is to correspond their molecular sequences or species (taxa) from which the tree is deduced. On the other side, internal nodes represent the last common ancestor to the nodes that arise from the that point. The type of tree, we intend to work with is composed of a multi-gene family called a gene tree which looks at one species and all its different variations.

3.1 The NP-hard Problem

Finding the most fit tree (solution) under any criterion, is considered to be a NP-hard problem within phylogenetic trees. When trying to construct a tree to find optimized solutions, it is best to use a heuristic that can both be classified as a search and optimization algorithm. Research has previously been done on the *subtree-pruning-and-regrafting* (SPR) and *nearest-neighbor-interchange* (NNI) methods [6], in which it was proven to show that the SPR was able to reproduce very close optimized solutions that gave trees that were not too significantly off to that of the best tree. In our case, the GA is a metaheuristic which both is a tool for search and optimization problems and would be a perfect tool to be used to test its capabilities within a specific **species** dataset.

3.2 The Maximum-Likelihood Approach

One of the slower (still fast for an NP-hard problem) but more frequent methods in tree reconstruction is that of the Maximum-Likelihood (ML) approach. This method is a statistical model of evolution where for each nucleotide position in a sequence, it approximates the probability of that position being a particular nucleotide, based on whether the internal node possess that specific nucleotide. The probabilities are then calculated for the branches of the bifurcating tree. ML is based on the idea that each nucleotide grows independently, enabling the phylogenetic relationship to be analayzed at each site.

Mathematically we can visualize this method with the Cavendar-Farris model [8]

$$\tilde{L}(\chi;T,\mathbf{p}) \cong -ln2\mathbb{P}[\chi|T,\mathbf{p}] = -ln(\sum_{\hat{\chi}\in H(\chi)} \prod_{e=(u,v)\in E(T)} p_e^{1\{\hat{\chi}(u)\neq\hat{\chi}(v)\}}(1-p_e)^{1\{\hat{\chi}(u)=\hat{\chi}(v)\}}) \tag{1}$$

Suppose we are given some arbitrary tree T on n leaves and the probabilities of transition on edges is, $\mathbf{p} = \{p_e\}_{e\in E(T)} \in [0,1/2]^{E_T}$, where E(T) is the set of edges for the tree and $E(T) \cong |E(T)|$ is the cardinality of E(T) [8]. An understanding for the model can be obtained from the following steps: choose any vertex as the root; pick a state {0,1} for the root uniformly at random; When moving farther away from the root, each edge e flips the state of its ancestor with the given probability p_e. And lastly we have to also assign a state for the leaves using the character χ. An extension of χ occurs when the the vertices of T are assigned a state of {0,1} for those equal to χ on the leaves. This set of all extensions χ can be reffered to $H(\chi)$.

Now that the fundamental componenets are understood, let us also break down from Equation 1, the Cavendar-Farris model. In this model if the set A is 1, it is known that A occurred, and would be 0 otherwise. In the maximum likelihood problem, we wish to compute $(T*, \mathbf{b}*)$, which is minimizing $\tilde{L}(\chi;T,p)$

3.3 The Distance-Matrix Approach

References

[1] Fortnow, L., *The Status of the P versus NP problem*, September 2009, Communications of the ACM 52(9):78-86

[2] Ahn, C.W., Ramakrishna, R.S., *Elitism-based compact genetic algorithms*, IEEE Transactions on Evolutionary Computation, Volume: 7, Issue: 4, Aug. 2003, Pages 367-385.

[3] Gammaitoni L., Hänggi P., Jung P., and Marchesoni F., *Stochastic Resonance* Rev. Mod. Phys. 70, 223–288 (1998)

[4] Gutierrez-Navaro, D., Lopez-Aguayo, S., *Solving ordinary differential equations using genetic algorithms and the Taylor series matrix method*, 2018 J. Phys. Commun. 2 115010

[5] Gupta, M., Singh S,. *A Novel Genetic Algorithm based Approach for Optimization of Distance Matrix for Phylogenetic Tree Construction*, International Journal of Computer Applications (0975 – 8887)

[6] Money, D., Whelan, S., *Characterizing the Phylogenetic Tree-Search Problem*, Systematic Biology, Volume 61, Issue 2, March 2012, Page 228

Volume 52– No.9, August 2012

[7] Eiben A.E., Raué P.E., Ruttkay Z. (1994) *Genetic algorithms with multi-parent recombination In*: Davidor Y., Schwefel HP., Männer R. (eds) Parallel Problem Solving from Nature — PPSN III. PPSN 1994. Lecture Notes in Computer Science, vol 866. Springer, Berlin, Heidelberg.

[8] Roch, S., *A Short Proof that Phylogenetic Tree Reconstruction by Maximum Likelihood Is Hard*, IEEE/ACM Transactions on Computational Biology and Bioinformatics, Volume 3 Issue 1, January 2006