Chapter 1

Graph Based Genetic Algorithms

Contributed by Kyle M.D. Sweeney

1.1 Introduction

Genetic Algorithms are fundamentally a searching algorithm for finding "good" solutions when the solution space is excessively large. Many problem spaces, such as NP-Hard or NP-Complete problems, are difficult because the possible solution space grows exponentially as the input size of the problem increases. Consequently, there are no guaranteed easy solutions that can be found in reasonable time. Searching algorithms, such as Simulated Annealing and Genetic Algorithms look to nature to find methods of finding reasonably good solutions. In the case of Genetic Algorithms, solutions are found by simulating evolution, pursuing a survival of the fittest approach.

Let's take the classic travelling salesman problem [4] where a saleswoman would like to travel from city to city, visiting each city only once, and taking the shortest route. The problem is classically known to be NP-Hard. There are $N!$ possible combinations of routes to search through. To solve the problem as a genetic algorithm, we can imagine the solution, an ordered list of cities, to be like "DNA", and each city is a gene. When organisms breed, they swap genes, and thus produce new, unique children which may or may not be fitter. Genetic Algorithms require a fitness function in order to sort out which solutions are moving towards a "good" solution, and are better than other solutions. In this case, the fitness function is the cost of the trip, given the ordered list of cities. In each generation, we produce a certain number of children from the solutions in the specimen pool, add them to the pool, and then only keep a certain number which are most fit, according to the fitness function. Eventually, we choose to stop, and the most fit function is our "good" solution.

Of course, while exploring the natural extrema of the solution space, it's possible for our solutions to get stuck around a local extrema. What this means is that our solution specimens have become too homogenized, and there's not enough unique variations to choose from. In genetic terms, there's not enough genetic variation. One possible solution to solve this is via mutations. By introducing mutations during the breeding stage, solutions can jump from one area of the solution curve to another, ideally pulling the rest of the gene pool away from a local extrema, and back on the path towards a better, more optimal solution. But this genetic variation has to be carefully controlled. Too much mutations, and the pool can never stabilize and never travel along the curve. Too little, and mutations don't introduce enough variability.

1.2 The Problem as a Graph

The problem of shrinking genetic variation can be partially solved by mutations, but can also be solved by the introduction of graphs into the problem space. In nature, the same species can be found in multiple places around the world, yet are still breed-able with one another. These groups are genetically similar to one another, and distinct from their cousins in different pockets in different environments. For the purposes of solving a problem like the traveling salesman, we can employ graphs to take advantage of community isolation while permitting limited genetic-crossover. Ideally, this means that each sub-group will develop a unique solution and by crossing over, they can help push the other groups towards more optimal solutions. The effectiveness of this approach in speeding up/improving solutions comes from a combination of the right kind of graph for the problem being solved.

1.3 Some Realistic Data Sets

To demonstrate integrating graphs as helpers in genetic algorithms, the rest of this chapter will focus on the application of Genetic Algorithms in finding ideal complementary codon-sequences to generate protiens in non-human cells at human-rates.

Every protein is comprised of Amino Acids, built inside of cells according to DNA [3]. Inside of a DNA strand, three nucleotides are strung together to form a codon, which then codes for either a specific amino acid, is a stop marker, or is a start marker. Each codon is used with a certain amount of frequency, and these frequencies are species specific. Work done by Clark et al. [2] discuss the implications of these frequencies, and work done by Rodriguez et al. [1] demonstrates an algorithm for harmonizing DNA sequences between humans and a targeted species. These papers discuss a method, where given a DNA sequence, and the frequencies for different species, a series of scores for each codon, based off of those frequencies. These scores can be seen as a function over the codon positions. While the same protein is constructed from each sequence, the "human" function and "bacteria" function could be very different. Harmonizing the DNA, in this case, means altering which codons are chosen in the bacteria so that the resulting "bacteria" function will be as similar to the "human" function as possible.

Solving this harmonization problem via genetic algorithms can be done by imagining the solution space as the chosen sequence of codons which still produce the same protein. The fitness function would then be the difference in the area between the two functions when plotted out. By minimizing the distance between the two functions, a harmonization can be accomplished.

1.4 Graph Based Genetic Algorithms-A Key Graph Kernel

When we apply graphs to isolate the breeding pairs of each potential solution, we perform a sort of graph "kernel" by traversing the graph to each of the neighbors from every node. The rough psudeo-code looks something like 1. For each vertex in the graph, breed it with every neighbor that it has. Of all these children, the most fit one will replace it after breeding has finished. Thus, in every round, only the most fit specimens remain.

The evaluation of this would be to test this algorithm on different graphs, measuring for speed and best solution score.

Algorithm 1 Graph Based Breeding:
G, V, E

```
1: procedure BREED(G, V, E)
2:     R = {}
3:     for v in V do
4:         N = Neighbors(v)
5:         for n in N do
6:             C = children(n, v)
7:             for c in C do
8:                 if fitness(c) < fitness(v) then
9:                     R+ = (c, v)
10:            end for
11:        end for
12:    end for
13:    for r in R do
14:        replace(G, r[0], r[1])
15:    end for
```

1.5 Prior and Related Work

Much of this chapter will be applying the work done by Ashlock et al. in their 1999 paper "Graph Based Genetic Algorithms", where they took different kinds of graphs and applied them to three genetic algorithms problems. They explored the time it took to solve the problems given many different kinds of graphs.

1.6 A Sequential Algorithm

The Kernel proposed above, if taken to be sequential, would have a rather complex execution time, dependent on the execution time of each of the underlying functions, specifically *children* and $fitness$. In the worst case scenario, a fully connected graph, the execution time is $O(V^2S^2C)$ where V is the number of vertices in the graph, S is the size of a given solution, and C is the number of children produced by *children*. The pseudo-polynomial nature of the solution is the power of the genetic algorithm, as a good chunk of the solution space is evaluated, but done in an algorithmic manner.

1.7 A Reference Sequential Implementation

Evaluation of this approach was done in Python3, ran using PyPy3 to speed up execution.

1.8 Sequential Scaling Results

Discuss here results from your sequential implementation. Include software and hardware configuration, where the input graph data sets came from, and how input data set characteristics were varied. Did the performance as a function of size vary as you predicted?

1.9 A Parallel Algorithm

1.10 A Reference Parallel Implementation

Discuss here an implementation of the basic parallel code. Include what language/paradigm you used for the code.

1.11 Parallel Scaling Results

Discuss here results from parallel algorithm. Include software and hardware configuration, where the input graph data sets came from, and how input data set characteristics were varied. Ideally plots of performance vs BOTH problem size changes AND hardware resources are desired. Did the performance as a function of size vary as you predicted?

1.12 Conclusion

Summarize your paper. Discuss possible future work and/or other options that may make sense.

1.13 Response to Reviews

This will be included only in the second and third iterations, and will be a summary of what you learned from the reviews you received from the prior pass, and how you modified the paper accordingly.

Bibliography

[1] Scott Emrich Patricia L. Clark Anabel Rodriguez, Gabriel Wright. %minmax: A versatile tool for calculating and comparing synonymous codon usage and its impact on protein folding. *Protein Science*, 1(27):356–362, 2018.

[2] Thomas F. Clarke, IV and Patricia L. Clark. Rare codons cluster. *PLOS ONE*, 3(10):1–5, 10 2008.

[3] Wikipedia contributors. Genetic code — Wikipedia, the free encyclopedia, 2018. [Online; accessed 14-September-2018].

[4] Wikipedia contributors. Travelling salesman problem — Wikipedia, the free encyclopedia, 2018. [Online; accessed 14-September-2018].