Genetic Algorithm Optimizations and Applications

Mr. Kevin Shurtz

Abilene Christian University  
624 Honeysuckle Hollow  
(214) 966-8439

kms14d@acu.edu

**ABSTRACT**

This paper explores the viability of genetic algorithmic techniques for solving various NP-hard problems, particularly focusing on theoretical and practical combinatorial problems. Initial experimentation investigated the utility of applying a genetic algorithm system to solving the knapsack problem, a well-known NP-hard combinatorial problem popular in academic thought. Multiple genetic techniques were employed; namely, the genitor approach – popular GA design that produces a single child each generation – is contrasted with a less selective alternative. Research concluded that genetic solutions to the knapsack problem were highly efficient, and with proper parameters, were accurate with remarkable frequency. Furthermore, it was discovered that various factors such as mutation rate, the number of convergences required before acceptance, and the number of parents in each generation all had radical effects on the efficacy of the algorithm. Furthermore, I found that the genitor approach was often unacceptably slow, and alternative systems driven by large quantities of offspring appeared more efficient. Then, the paper addresses a real-world application to which the same algorithm could be extended and generalized – namely, the task of determining an optimal student schedule over the course of several semesters. The underlying system driving both problems was identical, and with only minor extension, the GA system employed to the knapsack problem proved equally effective at providing efficient schedules for hypothetical students. Finally, I investigated the impact of varying the mutation rate for populations of different sizes. I found that, while a mutation rate of zero typically prevents the discovery of any solution, nearly all populations favor extremely low mutation rates. On a gene-by-gene basis, mutation rates below 3% were often optimal per speed considerations. Generally, it was also observed that high mutation rates correlated to higher degrees of accuracy, however, only small input sizes could sustain higher mutation rates and still reach a result in a reasonable timeframe.

**CCS Concepts**

• **Computing methodologies➝Artifical Intelligence➝Search methodologies➝Discrete space search** • **Computing methodologies → Artificial intelligence → Planning and scheduling → Planning for deterministic actions**

**Keywords**

Artificial intelligence; depth-first; heuristic; discrete space; search; optimization; genetic algorithm; genitor; scheduling; knapsack; Java

# INTRODUCTION

Genetic algorithms are emerging subset of the artificial intelligence domain that have already begun to impact a variety of industries. GA systems have been employed in airline booking systems, CAD software, and a multitude of other environments. Consequently, understanding the properties and features of effective GA systems is an engaging subject of increasing academic and professional significance.

To investigate this domain of artificial intelligence and the properties of its effective use, I begin by exploring the feasibility of using a GA system to solve the knapsack problem, a well-known, NP-hard combinatorial problem. I then extend the same system for use in a real-world domain, testing the feasibility of the GA system for generating optimal schedule for hypothetical classes and students. Finally, I run vary the mutation rate across several tests, evaluating the impact of different mutation rates for differently sized problem sets. These final experiments were conducted in the context of the knapsack problem.

All tests were conducted in Java 1.8. Statistical analysis and graphing was conducted with the R statistical programming language, using version 3.3.1 “Bug in Your Hair”. Much of the test automation was conducted via various shell scripts, mostly written for Bash. Together, these tools help illustrate the viability of GA systems in both academic and practical contexts.

# BACKGROUND

Before addressing the implications of the generated reports, we must first discuss the challenge and parameters surrounding the knapsack problem, the basis for my initial suite of tests.

The knapsack problem is a well-known NP-hard combinatorial problem. It is conceived as such: suppose there is a container of with limited capacity. Additionally, suppose there is some limited number of items of varying costs and values. The knapsack problem requires that some set of items is returned such that the sum of item costs does not exceed the capacity of the “knapsack” container, while the sum of the items’ values is maximized [2].

As noted by John J. Bartholdi, III from the Georgia Institute of Technology, the knapsack problem has been found to have various applications in economics, engineering, and business. Given the frequency of phenomena with comparable constraints to those of the problem, it was decided that a course-scheduling system would be a valuable for exploration later in this research project.

There has been a wealth of research on this subject, as it represents an interesting problem in the mathematical subfield of combinatorics and has a variety of potential applications.

Genetic algorithms represent are a particular variant of the stochastic beam search algorithm. Stochastic beam search algorithms are a class of algorithms comparable to stochastic hill climbing algorithms. With the intent to incrementally progress toward a local maximum, several successor nodes must be identified and subsequently explored. Stochastic beam searches generate several potential successor nodes based upon the current best state, and, assuming a better state is produced, select the new preferred state. While stochastic beam searches often generate successors by modifying the current optimal state. Genetic Algorithms generate successor nodes by identifying optimal states already generated in the explored state space, and combining their characteristics to produce preferable offspring.

Unfit or unnecessary offspring are removed from the gene pool for future generations, thereby ensuring that each generation does no worse than its predecessor. Should the population ever become stagnant, mutations can be used to artificially increase the diversity of the population between generations and improve the likelihood that it settles upon a correct answer. While there are many potential mechanisms for deciding upon a solution, the GA system built for the purposes of this project decides upon a solution when all members of a population converge upon a common “genome”. Some computer scientists are quite insistent that the components comprising a solution to a problem domain be represented solely as bit-strings, a representation which is quite sensible for combinatorial problems such as knapsack. For many other domains, however, this representation is unintuitive and even counterproductive. To satisfy all parties, all experiments for the purposes of this paper are combinatorial in nature, and therefore represent genomes with bit-strings.

The optimality of any “genome”, or bit-string, is determined with some “fitness function” specific to the domain. For the knapsack problem, the fitness function merely sums the value all included “genes”, or items, unless the cost of all items exceeds the maximum capacity of the knapsack. In this case, a penalty or negative value may be incurred. How such cases are handled is discretionary. When applying a GA system to a different domain, however, a new fitness function must be identified to evaluate the new “genome”, or set of all “genes”.

# METHODOLOGY

## Implementation

The details of the implementation for the project are rather extensive, and the various components of the entire system are built with several different languages and tools. The GA system was built with Java 1.8, and possess a modular, object-oriented architecture that allows for rapid extensibility.

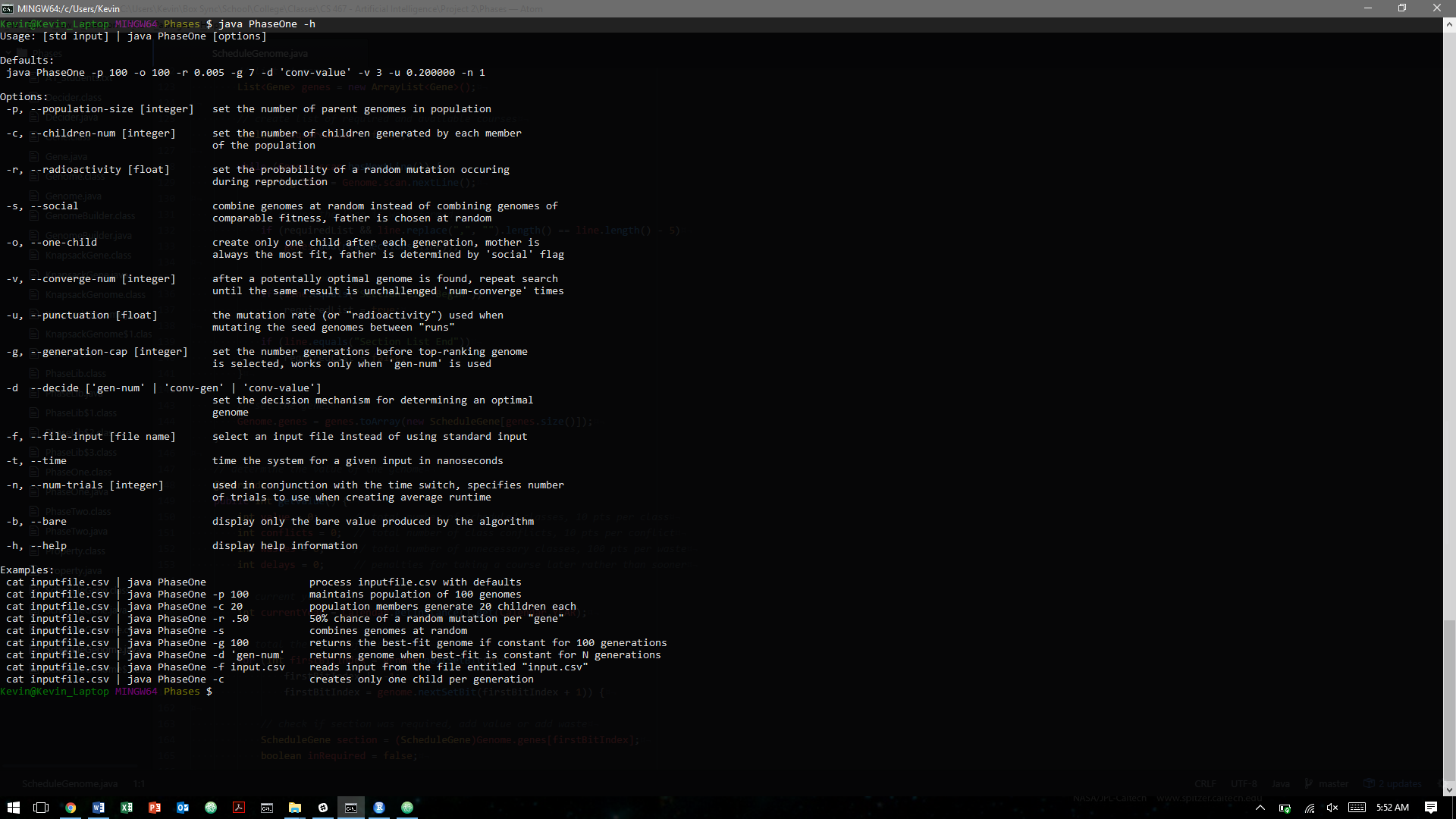
All Java source files are contained in the “Phases” directory, which contains all Java source files and class files. Because the project was divided into various Phases, all common functionality for the system is contained in “PhaseLib.java”, a library of static GA functions and members that allow the various client modules to generate preferred results for arbitrary problem domains. Every domain is represented as a set of “genes”, the properties of which are specified in a concrete child class of the abstract Gene object, which is defined in “Gene.java”. Examples of such child classes include “KnapsackGene.java”, used in Phase 1, and “ScheduleGene.java”, used in Phase 2. Similarly, the Genome class can be extended to provide a fitness function appropriate to the problem domain, as well as custom functions for reading input data into associated Gene objects.

Some additional features were added between the time of code submission and the conclusion of this paper. No core functionality of the codebase was changed. Several timing and formatting options were added to better facilitate the collection of runtime information outside of the system. The additional command-line arguments added are “—time, -t”, “—num-trials, -n”, and “—bare, -b”.

A variety command-line options were made available, making possible a broad range of experiments. In particular, the “-o” flag specifies that a single child is to be generated upon each generation. When specified, in conjunction with the other defaults, the GA system applies the genitor approach. Without this option, every two parents generates a child. The pool of children is then mutated, mixed with the parent population, sorted, and then bred. All genomes that fall outside the dedicated population size are then replaced by the next generation.

Only children are irradiated, so as to avoid corrupting potentially valuable parents. Most genitor implementations likely mutate the general population periodically, rather than just one child, so it is possible that the effectiveness of this implementation is artificially limited.

Figure I. Screenshot Depicting the Volume of Command-Line Options Available



## Sampling Techniques

For the final phase of the project, runtimes were sampled in nanoseconds from within the GA system. A Bash script handled the data collection and formatting, constructing CSVs for specified files with the mutation rate and associated runtime in nanoseconds. This data was then plotted and graphed using RStudio, a free statistical computing environment for use with the R statistical programming language. For this phase, each tested file was run with all mutation rates between 0 and 0.5, inclusive, at increments of 0.01. If previous tests rendered timely results, and subsequent tests exceed 30 seconds three times in a row, all remaining tests are preemptively halted. The code handling this procedure can be found in “phase.sh”, a bash script that generates the CSVs used for the graphs.

Each test is handled from within the GA system, and when run by the phase.sh script, uses the average of five runs for each data point graphed. This can be modified by changing the arguments passed to the system from within the Bash script, however.

# RESULTS

## Pruned vs Unpruned Search

The transition from an unpruned exhaustive search to a pruned search improved search time by several orders of magnitude. While the general result was not surprising, the size of the improvement was a bit unexpected.

The scale of this improvement is best demonstrated in the original test files, of which there are five. They are entitled “c01.csv”, “k05.csv”, “k10.csv”, “k24.csv”, and “k30.csv”. The first of these files I created myself, primarily to test the functionality of my system. The others, however, test larger and more interesting datasets; this is reflected in the naming conventions. There are five items in the second file, ten in the third, twenty-four in the fourth, and thirty in the fifth.

In the smallest of these files, the improvement is negligible. As seen in Figure I, for the datasets of length 5, the runtime improvement of the pruned search is nearly imperceptible. In other tests conducted while refining the output, it was actually observed that the unpruned search often ran a millisecond faster that its pruned counterpart. This is likely due to the extra branching logic necessitated by the pruning function. For the larger datasets, however, it become immediately evident that the pruned search is significantly faster.

For the list of length ten, the unpruned search runs for eight seconds while the pruned search runs for merely five. In some earlier tests, the pruned search ran for three seconds, suggesting that the recorded figure may be high. By the time the list reaches length twenty-four, the pruned search becomes many times quicker than the alternative. In this instance, the pruned search is 17.513 times faster than its unpruned counterpart. Depending on the ordering of the inputs and the quality of the greedy estimates, the improvement can be even starker. For “k30.csv”, the unpruned search is particularly slow, taking in 515,555 milliseconds in the selected test, and only 33 milliseconds in the pruned test. This represents 1,562,200% improvement for this test. This is – needless to say – remarkable.

Given the fact that this was a larger dataset, and yet the pruned time for the dataset of twenty-four items is actually fifteen times slower than that of the larger dataset, I am led to believe that the ordering of the data and the quality of the greedy estimates that prune the exhaustive search are dominant in shaping the outcome of enhanced search.

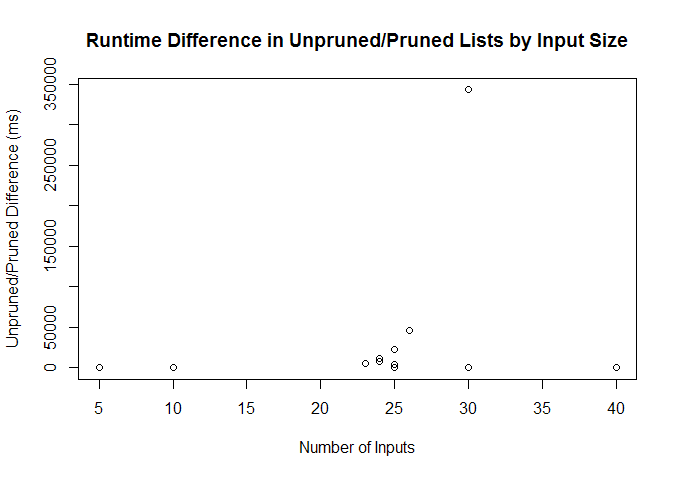
Figure I. Table Depicting Improvement Between Pruned and Unpruned Searches for Original Input Set

|  |  |  |  |
| --- | --- | --- | --- |
| **Length** | **Capacity** | **Unpruned Time (ms)** | **Pruned Time (ms)** |
| 5 | 10 | 3 | 3 |
| 5 | 13 | 2 | 2 |
| 10 | 30 | 8 | 5 |
| 24 | 60 | 8,704 | 497 |
| 30 | 96 | 515,555 | 33 |

This same procedure was applied systematically to both the “original” and “easy” datasets and converted into a CSV table for further processing, the results of which are seen in Figure II. Excepting two outlying data values, mentioned above, it appears that the difference in pruned and unpruned search times is likely exponential, hooking upward sharply between list lengths twenty-three and twenty-six.

Graphics for this data were generating using R, and open-source statistical programming language.

Figure II. Scatterplot Depicting Improvement Between Pruned and Unpruned Searches



## Elimination of Stack Overhead

The improvements garnered from the pruned search are relatively unsurprising. Consequently, I spent significant time implementing and analyzing additional optimizations for the pruned search.

The first of these optimizations was a small optimization in the stack structure used to manage the search tree. Clearly, when expanding a depth-first search, after each iteration, the deepest node in the frontier is expanded before any neighboring nodes are reached [1]. This is typically managed with a FIFO queue, or “stack”, which manages all nodes along the frontier. The first node added to the frontier is the first to be removed during frontier expansion, meaning that the memory burden of a search is a factor of the search space’s depth.

When expanding the search space, there are many ways to handle the stack. In my original implementation, once reaching a state stored on the stack, I expand all branching nodes. After the children are explored, the parent is removed from the stack. While this is functional, is unnecessarily inefficient. Once a parent stack has been reached, assuming that all children are expanded along the frontier, there is no utility in preserving that state in memory. This also necessitates more logic for determining if a state has been previously visited, leading to more branching logic in the body of the function.

In an effort to improve the performance of the function, I simplify my frontier maintenance, popping parent nodes off of the frontier stack immediately reaching them. This led to notable and consistent gains in performance, although these gains appear more poignant when the capacity for a particular input is lower. There also does not appear to be an exponential improvement with that accompanies the length of the input list, as with the previously discussed optimization.

Figure III. Table Depicting Improvements for Stack Optimization in Frontier Maintenance for A and B input files

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File** | **Length** | **Capacity** | **Control (ns)** | **Optimized (ns)** |
| A1.csv | 23 | 149 | 680,793 | 295,886 |
| B1.csv | 23 | 300 | 2,764,291 | 2,566,182 |
| A2.csv | 24 | 107 | 547,969 | 308,376 |
| B2.csv | 24 | 200 | 5,483,275 | 4,170,788 |
| A3.csv | 25 | 1116 | 171,727 | 126,618 |
| B3.csv | 25 | 1,637,333 | 713,829,142 | 617,150,726 |
| A4.csv | 26 | 900 | 3,133,731 | 2,609,037 |

Files with smaller capacities, such as “A1.csv” and “A2.csv” run 230.09% and 131.47% faster, respectively, when optimized. Inputs with larger capacities, such as “B3.csv”, however, do not demonstrate such an improvement. There are outliers to this generalization, however. The driving phenomenon behind these discrepancies could be a subject of later research.

# FUTURE WORK

Virtual compilers are enormously complex, and understanding the subtle optimizations conducted by the Java 1.8 compiler could be instrumental to understanding the performance of future tests. It is probable that the Java compiler makes optimization decisions that obfuscate the source of performance gains, thereby limiting the utility of tests such as these. This, in turn, could lead to more conclusive results.

# ACKNOWLEDGMENTS

I would like to thank Stuart Russell and Peter Norvig for their 2010 work, “Artificial Intelligence: A Modern Approach Third Edition”, which was invaluable in examining this subject. REFERENCES

1. Norvig, Peter, and Russel, Stuart 2010. Artificial Intelligence A Modern Approach Third Edition.

[2] <Https://Www.Utdallas.Edu/~Scniu/OPRE->

6201/Documents/DP3-Knapsack.Pdf. 1st ed. 2017. Print.