# Novel Applications of Stochastic Global Optimization Algorithms to the Shortest Common Superstring Problem

TYLER GIALLANZA

*Compiled February 17, 2016*

**The shortest common superstring problem aims to find a string with minimal length that contains every string in a given set. This problem has various applications in the fields of DNA sequencing and data compression. Because it is a max SNP-hard problem, with no polynomial time solution, in practice a greedy algorithm is used to approximate a solution. Since the greedy algorithm is only guaranteed to approximate a solution that is 248% optimal, any algorithm that can approximate a solution shorter than the greedy algorithm can is desirable. This research focused on implementing two stochastic optimization algorithms with the goal of outperforming the greedy algorithm by achieving shorter solutions: first, a random-sampling Metropolis-Hastings variant known as simulated annealing, and second, a genetic algorithm. The results of testing both algorithms against the greedy algorithm on a set of several randomly generated sample data showed, on average, that the genetic algorithm performed about 5% better than the greedy algorithm. An inverse relationship between size of test data and relative performance of the genetic algorithm was also witnessed, demonstrating the superiority of the genetic algorithm over the greedy algorithm for small data sets.**

## 1. INTRODUCTION

The shortest common superstring problem is a mathematical problem that has applications in the fields of data compression [1] and DNA sequencing [2] [3]. The formal definition of the shortest common superstring (also known as the shortest common supersequence) can be broken into two parts. First, a string $s_k$ is a superstring of $s_l$ if and only if $s_l$ appears entirely and uninterrupted at some location within $s_k$. Second, the shortest common superstring of a set $S = \{s_1, ..., s_n\}$ over the finite alphabet $\Sigma$ is the shortest string that is a superstring of every $s_i \in S$. Since finding the shortest common supersequence of any set of strings where $\Sigma \geq 2$ has been proven to be NP complete (meaning there is no deterministic polynomial time solution possible), a brute force algorithm entails unreasonably long runtimes, and thus is in practice impossible [4]. The preferred algorithm, instead, is a greedy algorithm: this algorithm has the advantage of a substantially faster runtime than brute force, but is merely an approximation algorithm that is not guaranteed to converge on the optimal solution.

The greedy algorithm is commonly used because of its deterministic nature, low runtime, and relative ease to program [5][3]. The algorithm selects two strings $s_i$ and $s_j$ from the list $S = \{s_1, ..., s_n\}$ such that the overlap of $s_i$ and $s_j$ is greater than the overlap of $s_k, s_l \in S$ (for any $k \neq i$ and $l \neq j$). $s_i$ and $s_j$ thus have the greatest overlap of any of the strings in $S$. The algorithm then replaces $s_i$ and $s_j$ with the newly formed overlapping string. This is repeated until only one string is left; this string is returned as the superstring.

The greedy algorithm has a much faster runtime than the brute force algorithm, but it approximates to such a degree that it is only guaranteed to produce a result that is within 248% of the length of the optimal solution, meaning the result returned by the greedy algorithm has the potential to be twice as long as the actual shortest common superstring [6].

The primary goal of this research is to produce an algorithm that can outperform the greedy algorithm, either on the basis of physical runtime or on the basis of the length of the resulting superstring. Any solution that either ran faster than the greedy algorithm while producing comparable results or generated a shorter superstring than the greedy algorithm was thus considered a successful solution. The greedy algorithm was used as a benchmark because of its centrality in the literature; the greedy algorithm is not only often used in practice, but also is the focus of the most research in the field. Researched less often, however, are the applications of non-deterministic algorithms to the shortest common superstring problem.

This research focused on two non-deterministic algorithms, the first of which is known as simulated annealing. Simulated annealing is a specific type of non-deterministic algorithm known as a stochastic global optimization algorithm. This means that the algorithm uses some form of randomness in an attempt to find a globally optimal solution to a problem, in this case the shortest common superstring problem. Simulated annealing specifically goes about this in a manner similar to that of the hill climbing algorithm. First, the algorithm picks a random starting point. Next, it looks at a neighbor of that point. If the neighbor results in a shorter superstring than the current point, the neighbor will become the current point, and the algorithm will continue neighbor to neighbor, improving as it goes along. The issue with the hill climbing algorithm is its tendency to get stuck in local optima (Figure 1).

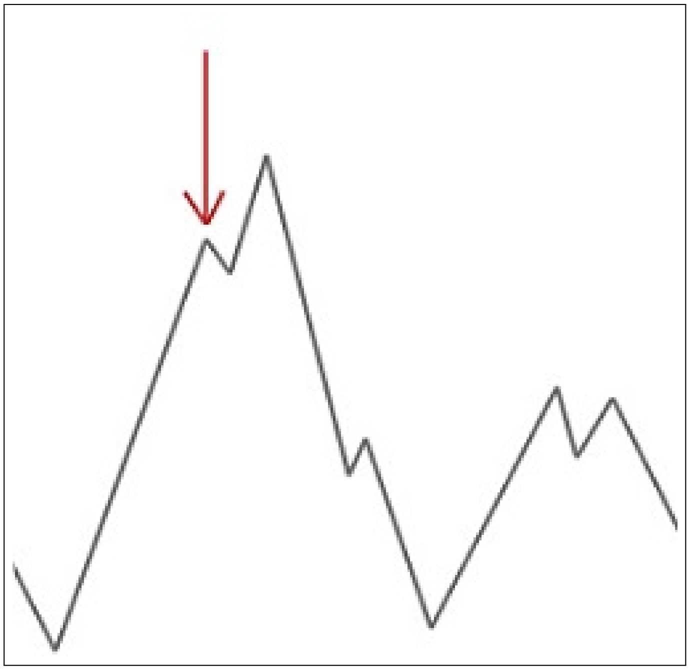

**Fig. 1.** The hill climbing algorithm gets stuck in local optima.

Because the algorithm only moves to a neighboring point if it is better than the current point, it has no way to escape a locally optimal solution. Simulated annealing improves upon this model by adding a probability of accepting a worse solution. This allows the simulated annealing algorithm to escape from local optima. At the start, there is a high probability of accepting a bad solution, meaning the algorithm can easily escape locally optimal solutions. As time goes on, however, the probability of accepting a worse solution decreases, allowing the algorithm to "lock in" on the globally optimal solution.

The second algorithm tested is a genetic algorithm. Genetic algorithms, like simulated annealing, are stochastic global optimization algorithms. Genetic algorithms work by mimicking the process of evolution, gradually selecting for the best solution. The genetic algorithm begins with a random population of individuals. From here, the probability that each individual will pass on their genetic material is determined based on fitness – the fitter the individual, the more likely it will pass on its genetic material to the next generation. Over time, the fitter individuals begin exchanging genetic material, resulting in better and better results as time goes on. Genetic variance is also introduced through random mutations, which allows genetic algorithms to escape from local optima as well.

Due to the non-deterministic nature of the two algorithms tested, running each algorithm multiple times yields different results. Therefore, parallel versions of both algorithms were also tested, with multiple versions of the algorithm running simultaneously. This provides a large advantage to the genetic algorithm especially because the different instances can exchange information: if one instance finds a particularly promising solution, it can pass that information along so the other instances can use it as a starting point.

Stochastic global optimization algorithms were chosen for their inherent parallelization and for the lack of research on their applications to the shortest common superstring problem. Because both algorithms have outperformed greedy algorithms in other problems [7] [8], it was of particular interest whether or not they would be able to do the same for the shortest common superstring problem.

## 2. MATERIALS AND METHODS

### A. Materials

The entirety of the research was conducted on an HP envy Touchsmart 15 laptop with an Intel i7-4700mq processor and 8 GB of DDR3 RAM. The program was coded for in the Python 2.7 programming language, and was compiled by the 64 bit PyPy compiler for Python 2.7 on Windows 10. Code was written using the Vim text editor running on the Linux Mint 15 operating system. The code is attached in appendix A; the entirety of the code is present, and every line was written solely by the author of the paper.

### B. Greedy Algorithm

For the purposes of running as a benchmark to test the other algorithms against, a greedy algorithm was implemented. The greedy algorithm was implemented based on that presented in [9]; the implementation was designed to best reflect the standard greedy algorithm that appears in the literature. In short, the greedy algorithm operates by finding the two strings in the list with the largest overlap. The algorithm then merges those strings into one string, and repeats the process until only one string remains (see Introduction for a more formal description). A pseudocode outline is found in algorithm 1.

**Algorithm 1.** Greedy Algorithm

```
procedure GREEDY($strings$)
    while $length(strings) > 1$ do        ▷ Stop when only one string is left
        for $i \leftarrow 0, length(strings)$ do
            for $j \leftarrow 0, length(strings)$ do
                if $strings[i] \neq strings[j]$ then
                    if $overlap(strings[i], strings[j]) < record$ then        ▷ Find the largest overlap between any two strings
                        $record \leftarrow overlap(strings[i], strings[j])$
                        $first \leftarrow i$
                        $second \leftarrow j$
        $strings.remove(first)$
        $strings.remove(second)$
        $strings.add(record)$
    return $strings[0]$
```

### C. Simulated Annealing

The simulated annealing algorithm operates by taking a directed random walk through all the possible solutions to the input. The first iteration begins by creating a random ordering of the input strings. The fitness of each random ordering, a measure of how desirable that particular random solution is, is then determined by calculating the length of the superstring created by overlapping all adjacent strings. From this fitness measure, it can be determined which random ordering is preferable; for example, consider the following strings:

| 0 | 1 | 2 |
|---|---|---|
| abba | baaaa | bbbabba |

Two different strings orderings, such as $\{0, 1, 2\}$ and $\{2, 0, 1\}$, yield different length strings:

| $\{0, 1, 2\}$ | $\{2, 0, 1\}$ |
|---|---|
| abba+baaaa+bbbabba | bbbabb+abba+baaaa |
| abbaaaabbbabba | bbbabbabaaaa |
| length: 14 | length: 12 |

If the fitness of the current iteration is better than the fitness of the previous iteration, the current ordering is a better solution and it is thus saved as the value to beat (herein referred to as the "saved value"). If the fitness of the current iteration is worse than the fitness of the previous iteration, there is a probability that it will still be chosen as the saved value. This probability is based on two factors.

First, the difference in the fitnesses – this makes it much less likely that a large setback is incurred. Second, the amount of time that the algorithm has been running – this means bigger risks will be taken when the algorithm first starts, but near the end the algorithm will "zero in" on the best solution instead of jumping around randomly to worse solutions. The rationale behind sometimes choosing a worse solution is the genius of the simulated annealing algorithm – by choosing a worse value in the short term, the long term benefit is the ability to escape local optima (see introduction). Finally, a neighbor of the saved value is selected randomly, and the same process is repeated. A neighbor is determined by randomly swapping two adjacent strings in the saved value: by making such a relatively small change, good solutions are conserved because merely swapping two adjacent strings is a minute change that is unlikely to negatively impact the solution too much. The process of selecting a neighbor of the saved value, changing the saved value if a better result is found, and repeating continues until a certain number of iterations is reached, at which point the algorithm returns the saved value as its solution. The pseudocode of the implementation is outlined in algorithm 2.

**Algorithm 2.** Simulated Annealing

```
procedure SA($strings, t_{max}, t_{min}, \sigma$)
    $t \leftarrow t_{max}$
    $\lambda \leftarrow rand(0, 1)$    ▷ random function non-inclusive
    $saved\_strings \leftarrow rand\_perm(strings)$    ▷ gets a random permutation of the string indices
    while $t \geq t_{min}$ do
        $new\_strings \leftarrow rand\_swap(saved\_strings)$    ▷ gets a random shuffle of the saved strings
        $\Delta \leftarrow fitness(new\_strings) - fitness(saved\_strings)$
        if $\Delta \leq 0$ or ($\Delta > 0$ and $rand(0, 1) \leq e^{\frac{-\Delta}{t}}$) then
            $saved\_strings \leftarrow new\_strings$
        $t \leftarrow te^{\frac{-\lambda t}{\sigma}}$
    return $saved\_strings$
```

As previously mentioned, the probability of choosing a worse solution decreases over time. This probability is represented as a "temperature," or t, which is reduced by a factor of $e^{\frac{-\lambda t}{\sigma}}$ every iteration. The use of this reduction factor is known as Boltzman Annealing [10], and it was chosen because it is a conservative solution that slowly reduces temperature; there are other reduction factors that are used, but they lower temperature more quickly [11]. In the implementation, the value of $\sigma$ was set to 0.09. Additionally, the value of $t_{max}$ was set to 100, and the value of $t_{min}$ was set to 0.0005.

In its final implementation, the simulated annealing algorithm was parallelized. This was accomplished simply by running multiple instances of the algorithm simultaneously and selecting the result that was the best across all instances; because simulated annealing conducts a random walk through the possible solutions, running multiple instances simultaneously simply increases the likelihood that an optimal solution will be found by covering more possible solutions.

### D. Genetic Algorithm

A sequential version and a parallel version of a genetic algorithm were implemented as well. The genetic algorithm follows the basic outline patterned in the literature [12]: first, an initial population is randomly generated. Next, the viability of each solution is determined by a fitness function. Finally, various genetic operations are applied to the initial population in order to generate a new population of potentially fitter individuals. The genetic operations are applied in proportion to the fitness of each individual of the population; the idea is for fitter individuals to be more likely to pass their genetic material to the next generation than non-fit individuals, mimicking the natural process of evolution.

The fitness function implemented is modeled off of the rank space method [13]. The rank space method is unique in that it operates based on a relative ordering of individuals rather than an absolute ranking; instead of using the actual fitness value of each individual to determine the probability of reproduction, all of the individuals are ranked in order of fitness and the rank is used to determine the probability of reproduction. This subtle difference yields large results; the relative ranking system does not place as much value on fitness as does the absolute ranking system, encouraging diversity of solutions and allowing the algorithm to escape local optima.

In short, the fitness function ranks all of the individuals based on fitness; the higher an individual is on this list, the more likely it is to be picked for genetic operation, thus passing itself on to future generations. In detail, a value $P_c$ represents the probability that the best individual is chosen. The chance that each subsequent individual is chosen is reduced by a factor of $P_c$ such that the probability of the $n$th best solution being chosen is

$$(1 - P_c)^{n-1}$$

In the implementation, $P_c$ was set to 0.05 [12]. This means that there was a 5% chance of the best individual being chosen, a 4.75% chance of the second best individual being chosen (if the first was not chosen), a 4.5125% chance of the third best individual being chosen (if the first and second were not chosen), and so on.

After the fitness function decides which individuals will survive into the next generation, the genetic operations are applied. The genetic algorithm implemented utilizes three different genetic operations: selection, crossover, and mutation. Of these, selection is the simplest. Selection simply means the current individual is replicated in its entirety. The advantage of selection is that very good solutions get conserved. If a solution is particularly good, it makes since that it should be copied into the next generation. However, the drawback of selection is that in and of itself it does not allow for diversity. The selection operation doesn't change anything; nothing is allowed to improve, which undermines the purpose of evolution.

Crossover attempts to add diversity to the next generation. In genetic crossover, information from two different individuals is exchanged, creating two new children with some information from each parent. The type of crossover implemented in this algorithm is two point crossover. In Two point crossover, two parents are chosen. Then, part of each parent is exchanged with the other parent at a random location to form two children (Figure 2).

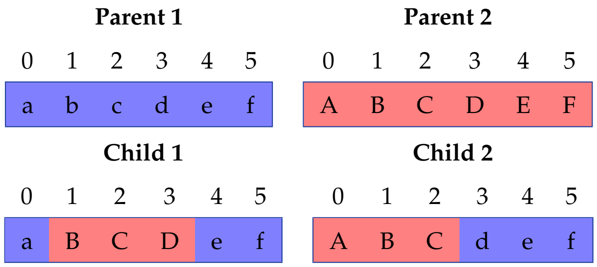

**Fig. 2.** Two point genetic crossover with a length of 3. Child 1 is formed by crossing parent 2 into parent 1 at the indices 1, 2, and 3. Child 2 is formed by crossing parent 1 into parent 2 at the indices 0, 1, and 2. The crossover is "two point" because the starting location is different in each parent.

Crossover has an advantage over selection because it increases the diversity of the next generation. Each child is different from both parents, making it more probable that a new solution will be found. The issue with crossover is the potential of finding a worse solution – it is possible that the parts taken from each parent combine poorly. However, just like genetic crossover in the biological sense, it is far more likely that a good solution will be found. Therefore, crossover is the “middle of the road” genetic operator – it creates more diversity than selection, but is less error-prone than the final operator: mutation.

Mutation is the most extreme of the genetic operators. As the name implies, mutation randomly changes an individual in the pursuit of diversity. The mutation implemented relies on a localized shuffle of the original individual. Two random numbers are generated: a start index, and an end index. Then, every value between the start and end indices are randomly shuffled. This was the chosen method because it has a more moderate effect – only a small area of the individual is effected rather than the entirety, causing small, incremental changes rather than large, sporadic ones.

In combination, all three genetic operations work together to ensure diversity as well as conservation of the best solutions. As implemented, selection was used 10% of the time, crossover was used 45% of the time, and mutation was used 45% of the time [14]. The process of fitness selection and genetic operation to generate the next generation continues for 800 generations, where each generation has a population size of 50 individuals [12], and then the algorithm terminates, returning the best individual of the most recent generation as its solution. The pseudocode is as follows in Algorithm 3.

**Algorithm 3.** Sequential Genetic Algorithm

```
procedure GA($strings, gens, pop, P_{selection}, P_{mutation}, P_{crossover}$)
    initialize $current\_gen$ with first generation
    for $i \leftarrow 0, gens$ do
        $current\_gen\_fitness \leftarrow fitness(current\_gen)$
        $next\_gen \leftarrow sort\_by\_fitness(current\_gen, current\_gen\_fitness)$
        $current\_gen.clear()$
        while $j < pop$ do
            if $j < \lfloor P_{selection} * pop \rfloor$ then
                $current\_gen.add(selection(current\_gen))$
                $j \leftarrow j + 1$
            else if $j < \lfloor P_{mutaiton} * pop \rfloor$ then
                $current\_gen.add(mutation(current\_gen))$
                $j \leftarrow j + 1$
            else
                $current\_gen.add(crossover(current\_gen))$
                $j \leftarrow j + 2$
    return $next\_gen[0]$
```

As with simulated annealing, a parallel version of the genetic algorithm was also implemented. This amounted to multiple instances of the algorithm running, seeded with different starting points. The added advantage of parallelizing a genetic algorithm is the introduction of a new genetic operator: migration. Migration works across all running instances of the genetic algorithm. Beforehand, a random generation is chosen to be the start of the migration period. Once that generation is reached, every ten generations two of the running instances exchange their top performing individual (Figure 3).

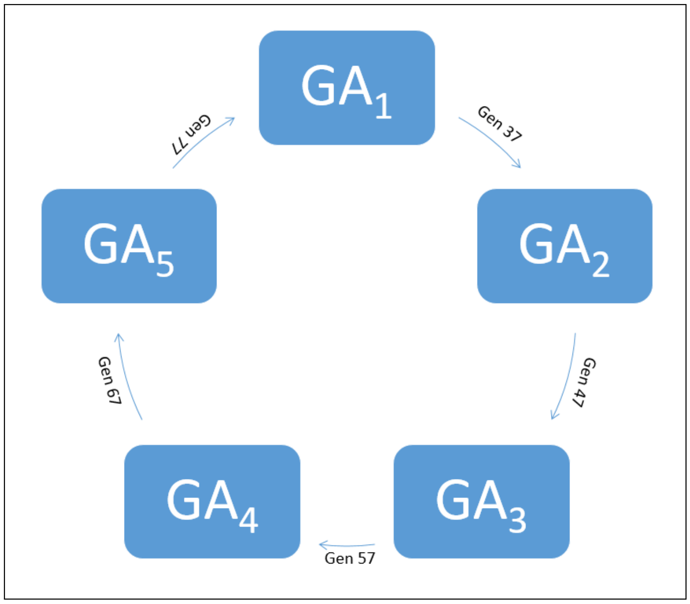

**Fig. 3.** An example of migration starting at generation 37.

This allows all instances of the genetic algorithm to converge on optimal solutions; if one of the instances has found a globally optimal solution, it can share its results with the rest of the instances. When all of the running instances have terminated, the overall best solution is chosen and returned as the solution.

## 3. RESULTS

The data were collected by compiling the attached code with the PyPy 64-bit compiler for Python 2.7. Tests were run on the Windows 10 operating system (see materials for machine specifications). For both simulated annealing and the genetic algorithm, two different tests were run. The first test used a sample size of 10 randomly generated strings, each of a random length between 10 and 20, and the second used a sample size of 20 randomly generated strings, also of a random length between 10 and 20 [15]. Each sample size was tested over 5 trials, and the results of those trials were averaged.

Results are reported in terms of relative performance to the greedy algorithm benchmark. For example, a score of 5% means the tested algorithm produced a result 5% shorter than the greedy algorithm.

**Table 1. Genetic Algorithm Length Data (Average)**

| Strings | Greedy | Genetic | Genetic Improvement |
|---|---|---|---|
| 10 | 107 | 101 | 5.698% |
| 20 | 207 | 198 | 4.4% |

The data for the genetic algorithm demonstrate a 4.782% improvement over the greedy algorithm, with the genetic algorithm generating a shorter, or better, superstring in every trial. This can be broken down into the results for 10 strings and 20 strings: when run on 20 strings, the improvement was 4.4%, and when run on 10 strings the improvement was 5.698% (Table 1).

**Table 2. Simulated Annealing Length Data (Average)**

| Strings | Greedy | SA | SA Improvement |
|---|---|---|---|
| 10 | 110 | 113 | -11.82% |
| 20 | 212 | 236 | -2.95% |

The data for the simulated annealing algorithm demonstrate a -7.38% improvement over the greedy algorithm, with the simulated annealing algorithm generating a longer, or worse, superstring in every trial. As with the genetic algorithm, this can be broken into 10 string and 20 string results: the 20 string data set showed a -11.82% improvement, and the 10 string data set demonstrated a -2.95% improvement over the greedy algorithm (Table 2).

**Table 3. Timing Data (Average)**

| Strings | Greedy(ms) | Genetic(ms) | SA(ms) |
|---|---|---|---|
| 10 | 597 | 10439 | 3715 |
| 20 | 1133 | 23995 | 23768 |

Both the genetic algorithm and the simulated annealing algorithm took far longer to run than the greedy algorithm. On average, the genetic algorithm took 28.2 times as long to run, and the simulated annealing algorithm took 24.7 times as long to run (Table 3).

## 4. DISCUSSION

Much of the work conducted was aimed at improving the results of the genetic algorithm. As a result, the algorithm underwent a variety of iterations, but the majority of the improvements can be separated into three distinct versions.

In the first version, the initial population was randomly generated from a solution space that includes all strings of a viable length, even strings that are not valid superstrings. Since the actual input strings tested were in binary, each individual was simply a random binary number. This solution was based off of the common method outlined in the literature [16]. However, the obvious issue with this representation scheme is that only a small minority of all strings are valid superstrings for a given data set; by including all strings, the solution space increased substantially. Additionally, the crossover and mutation genetic operations introduced the possibility of yielding an invalid superstring.

The second version fixed the representation issue by generating a random ordering of the input strings rather than an entirely random string. The rationale behind this decision is that every superstring can be represented as some ordering of the input strings that are overlapped. This is the same representation scheme used in the simulated annealing algorithm (see Materials and Methods, subsection C). Although this new version takes longer to run because it has to overlap all of the input strings in order to find the superstring, it is much preferable because it generates valid superstrings. Additionally, this version fixed the crossover and mutation genetic operations. Both operations were modified to ensure that each new string ordering has one and only one copy of each string: crossover gets re-run if any duplicates are found, and mutation is a localized shuffle (see Materials and Methods, subsection D). This version improved substantially on the first version, but it was still imperfect – it generated results comparable to the greedy algorithm, but rarely produced shorter strings.

The third and final version fixed this issue by seeding the genetic algorithm with the greedy algorithm. This means that instead of starting with a random ordering of strings, the genetic algorithm starts with the ordering of strings generated by the greedy algorithm. In other words, the result from the greedy algorithm is “fed in” to the genetic algorithm for further improvement. The rationale behind this is that there is no reason for the genetic algorithm to start from scratch if it can instead start with the results of the greedy algorithm. The genetic algorithm, then, functions as a heuristic optimization to the greedy algorithm – the advantage of this is that if in the future someone comes up with a major improvement to the greedy algorithm, the genetic algorithm will improve as well.

The final version of the genetic algorithm, ran in parallel, ended up, on average, outperforming the greedy algorithm, generating superstrings 4.782% shorter than the greedy algorithm. The simulated annealing algorithm, on the other hand, generated superstrings 7% longer than the greedy algorithm. As such, the genetic algorithm is considered a viable alternative to the greedy algorithm, but the simulated annealing algorithm is not.

Additionally, the genetic algorithm incurred better performance with smaller data sets. The average improvement for the data set with 20 strings was 4.4%, and the average improvement for the data set with 10 strings was 5.698%; this suggests that the genetic algorithm is especially viable for smaller data sets.

Despite generating shorter superstrings than the greedy algorithm, the genetic algorithm will most likely not replace the greedy algorithm in implementation. This is primarily due to the long time required to run the genetic algorithm; it took, on average, 28.2 times as long to run as did the greedy algorithm. Additionally, even though the genetic algorithm outperformed the greedy algorithm in every case tested, the stochastic nature of the algorithm means that it is possible, if not likely, that the genetic algorithm could generate a solution worse than the greedy algorithm. The deterministic nature of the greedy algorithm makes it predictable – that kind of stability is often desired in real-world applications like data compression.

Even though the genetic algorithm will not likely replace the greedy algorithm in practice, this research has generated many improvements that are of interest to future research. First of all, the design of the genetic algorithm offers a few unique approaches not found elsewhere in the literature. Specifically, the data representation scheme (using string orderings instead of directly generating random binary strings), and the genetic operations (using a localized shuffle method for mutation to preserve the superstring property of the data and using a "safe" two-point crossover that also preserves the superstring property) can be implemented in future genetic and evolutionary solutions to the shortest common superstring problem and other related problems. Additionally, parallelism can help to overcome the large time requirement of the genetic algorithm. This research was conducted on a processor with 8 logical cores; by using more parallelized hardware, for example graphics cards, the genetic algorithm can be reduced down to similar times as the greedy algorithm.

The goal of this research was to develop an algorithm that could either produce a shorter superstring than the greedy algorithm or produce the same length superstring as the greedy algorithm in less time. In the end, the most important aspect of the research was the genetic algorithm. Various modifications were made that make the genetic algorithm feasible for solving the shortest common superstring problem. The genetic algorithm did generate a shorter superstring than the greedy algorithm on average, making it a success. If the time can be reduced, it is possible that the modifications and improvements made to the genetic algorithm will allow it to replace the greedy algorithm as the de-facto standard solution to the shortest common superstring problem.

## REFERENCES

1. E. Schreiber and R. Korf, "Using partitions and superstrings for lossless compression of pattern databases," in "Twenty-Fifth AAAI Conference on Artificial Intelligence," , vol. 25 (2011), vol. 25.
2. P. Pevzner, *Computational Molecular Biology: An Algorithmic Approach* (MIT Press, 2000).
3. M. Tammi, "The principles of shotgun sequencing and automated fragment assembly," Center for Genomics and Bioinformatics, Karolinska Institutet (2003).
4. K. Raiha and E. Ukkonen, "The shortest common supersequence problem over binary alphabet is np-complete." Theoretical Computer Science **16**, 187–98 (1981).
5. B. Ma, "Why greed works for shortest common superstring problem," Theoretical Computer Science **410**, 5374–5381 (2009).
6. A. Golovnev, A. Kulikov, and I. Mihajlin, "Approximating shortest superstring problem using de bruijn graphs," in "Combinatorial Pattern Matching," (Springer, 2013), pp. 120–129.
7. M. Dorigo and L. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," Evolutionary Computation **1**, 53–66 (2002).
8. D. Zhao, W. Xiong, and Z. Shu, "Simulated annealing with a hybrid local search for solving the traveling salesman problem," Journal of Computational and Theoretical Nanoscience **12**, 1165–1169 (2015).
9. A. Frieze and W. Szpankowski, "Greedy algorithms for the shortest common superstring that are asymptotically optimal," Algorithmica **21**, 21–36 (1998).
10. D. Johnson, C. Aragon, L. McGeoch, and V. Schevon, "Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning," Association for Computing Machinery: Operations Research **37**, 865–892 (1989).
11. L. Ingber, "Very fast simulated re-annealing," Mathematical and Computer Modelling **12**, 967–973 (1989).
12. L. Ingber and B. Rosen, "Genetic algorithms and very fast simulated reannealing: A comparison," Mathematical and Computer Modelling **16**, 87–100 (1992).
13. A. Eiben, *Theoretical Aspects of Evolutionary Computing* (Springer Science and Business Media, 2013), chap. Evolutionary Algorithms and Constraint Satisfaction: Definitions, Survey, Methodology, and Research Directions.
14. P. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach* (Prentice Hall, 2009), 2nd ed.
15. X. Liu, "Sequential and parallel algorithms for the shortest common superstring problem," Parallel Numerics pp. 97–107 (2005).
16. R. Parsons, S. Forrest, and C. Burks, "Genetic algorithms for dna sequence assembly," in "International Conference on Intelligent Systems for Molecular Biology," , vol. 1 (1993), vol. 1, pp. 310–318.