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 nd a string with minimal length that contains every

shortest common superstring problem. Simulated annealing speci cally 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).

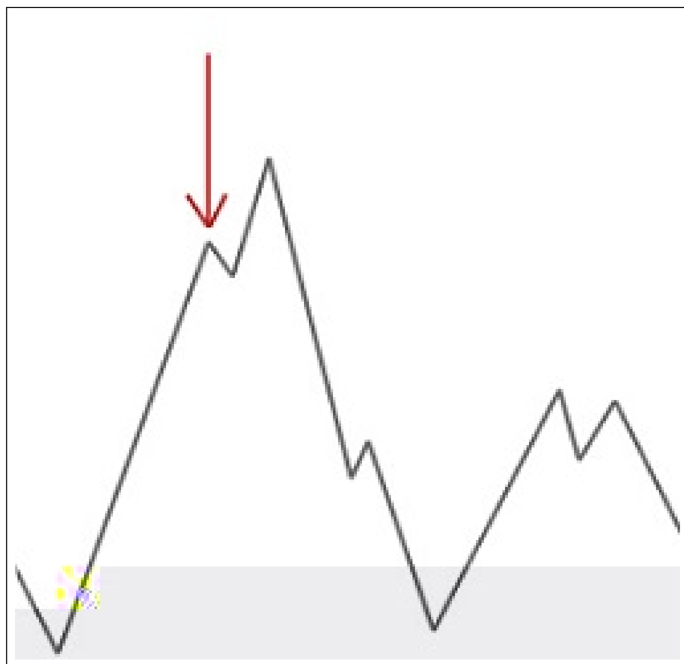

Fig. 1. The hill climbing algorithm gets stuck in local optima.

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 tness – the tter the individual, the more likely it will pass on its genetic material to the next generation. Over time, the tter 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 nds 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 re ect the standard greedy algorithm that appears in the literature. In short, the greedy algorithm operates by nding 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

```
1:  procedure GREEDY(strings)
2:      while length(strings) > 1 do            ▷ Stop when only one
    string is left
3:          for i    0, length(strings) do
4:              for j    0, length(strings) do
5:                  if strings[i] ≠ strings[j] then
6:                      if overlap(strings[i], strings[j]) < record
    then     ▷ Find the largest overlap between any two strings
7:                          record    overlap(strings[i], strings[j])
8:                          first    i
9:                          second    j
10:         strings.remove(first)
11:         strings.remove(second)
12:         strings.add(record)
13:     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 rst iteration begins by creating a random ordering of the input strings. The tness 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

```
1: procedure SA(strings, t_max, t_min, s)
2:     t    t_max
3:     l    rand(0, 1)                . random function non-inclusive
4:     saved_strings    rand_perm(strings)    . gets a random
   permutation of the string indices
5:     while t    t_min
```

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

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 speci cations). For both simulated annealing and the genetic algorithm, two different tests were run. The rst 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 rst 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 xed 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 nd the superstring, it is much preferable because it generates valid superstrings. Additionally, this version xed the crossover and mutation genetic operations. Both operations were modi ed 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 shuf e (see Materials and Methods, subsection D). This version improved substantially on the rst version, but it was still imperfect – it generated results comparable to the greedy algorithm, but rarely produced shorter strings.

The third and nal version xed 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 nal 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

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. Speci cally, the data representation scheme (using string orderings instead of directly generating random binary strings), and the genetic operations (using a localized shuf e 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 modi cations 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 modi cations 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.