Evolutionary Computation: Genetic Algorithms and Simulated Annealing Implementations on NP-Complete Problem Spaces

Jayson C. Garrison

The University of Tulsa
Tandy School of Computer Science
CS-4623-01 (22/SP)
April 25, 2022

`https://github.com/jayson-garrison/EC-Sum-of-Subsets`

1 Introduction

1.1 Evolutionary Computation and Annealing

Evolutionary Computation is a sub-field of Artificial Intelligence that uses principles of biology to determine optimal solutions to known computationally hard problems. More specifically, Genetic Algorithms (GAs) often produce near-optimal results for many hard optimization problems. Annealing methods, deriving their principles from metal annealing techniques, also offer approaches to solving hard problems.

1.2 Genetic Algorithms

Genetic Algorithms effectively use fundamental principles from biology, such as survival of the fittest, crossover, and mutation to search a problem space for a good solution. By implementing these biological techniques along with stochastic methods, GAs provide highly robust algorithms with the option to alter different operations in the process to produce good solutions. Specifically, different parent selection, crossover, and mutation routines can be implemented and compared based on the specific problem domain. Combined with the option to differ operators, GAs explore a large area of the problem domain when implemented efficiently. For this reason, GAs are a reliable method in determining near-optimal solutions for computationally hard problems.

1.3 Simulated Annealing

In addition to Genetic Algorithms, Simulated Annealing is often used to solve hard optimization problems. While fundamentally different from Evolutionary Computing techniques,

Simulated Annealing offers a simple approach to modifying an initial solution using perturbation functions to yield near-optimal solutions. Simulated Annealing handles exploration of large problem spaces by sometimes settling for a worse solution for the potential to gain a better solution later by perturbing the worse option.

2 Problem Domain and Data Sets

In order to analyze the effectiveness of the Genetic Algorithm and Simulated Annealing implementations, a well-known computationally hard problem was used for both approaches. The Sum of Subsets (SoS) problem allows for the application of the two techniques on trivial and non-trivial data sets.

2.1 The Sum of Subsets Problem

Given a set $\mathcal{S}$ of size $|\mathcal{S}|$ and a constant k find the subset S where w is defined by

$$w = \sum_{i=1}^{m} S_i$$

with the restriction

$$w \leq k$$

such that w is maximized.

This problem describes a variation of the Sum of Subsets problem and is known to be a computational complexity of $\mathcal{O}(n!)$ due to the combinatorial aspect of the optimization. Obviously, an iterative approach to solving this problem is computationally explosive in large domains and therefore requires a more efficient algorithm to determine near-optimal solutions.

2.2 Data Sets

In order to determine the effectiveness of the two approaches to solving the SoS problem, two different types of data sets were used. First, a trivial, or contrived, data set was constructed to verify that the GA and SA could yield the optimal or near-optimal solution in a small known domain. With the verification of the two implementations, three larger domains were used to test the performance of these approaches. In the latter data sets, the optimal solution is not known.

2.2.1 Contrived Data Set

In the contrived data set, $\mathcal{S}$ was initialized with 95 randomly generated numbers, $r \in [20, 2000]$. Then, $\{1, 4, 7, 3, 2\}$ was inserted into $\mathcal{S}$ such that $|\mathcal{S}| = 100$. Therefore $k = 17$ where the optimal solution of $\{1, 4, 7, 3, 2\}$ is known. Even though this is a contrived data set, the problem space is $|P(\mathcal{S})| \approx 1.268 \times 10^{30}$ which is massive.

2.2.2 Experimental Data Sets

Three different data sets of different sizes were used to test the accuracy of the two approaches. Each of them are described by

$$\mathcal{S}_1 : |\mathcal{S}_1| = 100, \text{each value drawn randomly from } [1, 250], k = 2\,500$$
$$\mathcal{S}_2 : |\mathcal{S}_2| = 1\,000, \text{each value drawn randomly from } [1, 10\,000], k = 2\,500\,000$$
$$\mathcal{S}_3 : |\mathcal{S}_3| = 10\,000, \text{each value drawn randomly from } [1, 100\,000], k = 25\,000\,000$$

The initial populations for the GA consisted of 250, 500, and 1 000 randomly generated subsets of random size according to each data set respectively. The SA algorithm instead used one initial randomly generated solution.

3 Genetic Algorithm Explanation and Implementation

In order to solve the SoS problem, a generational Genetic Algorithm with an randomly generated initial population was used with different selection, crossover, and mutation operators. Different implementations of the operators allowed for several near-optimal solutions to be found. In addition to the three basic operations in this GA, elitism was used to directly pass on strong genetic data to the next generation.

3.1 Chromosomal Representation and Fitness Calculation

Solutions to the SoS problem were stored in chromosomal form. To do this, solutions were represented in binary bit strings. To do this, the set $\mathcal{S}$ is stored as a key for decoding a given subset solution S. Therefore, a 1 in the chromosomal representation denotes the presence of that positional value in $\mathcal{S}$ in the solution. Similarly, a 0 represents the absence of that positional value in $\mathcal{S}$ in the solution.

$$\begin{aligned} \{1, 3, 13, 9, 7\} &\rightarrow \text{The set } \mathcal{S} \text{ as a key} \\ \{9, 7, 1\} &\rightarrow \text{A possible subset as a solution} \\ [1\ 0\ 0\ 1\ 1] &\rightarrow \text{The chromosomal representation} \end{aligned}$$

Figure 1: Example of a chromosomal representation using a key

Since the w associated with any solution set S can be greater than the desired value of k, infeasible solutions often occur. Instead of ignoring the infeasible chromosomes, which could

contain genetic material worthwhile for crossover, a fitness function was defined to reduce the fitness of infeasibles:

$$f(S) = |w - k| + \frac{\beta \cdot k}{10}$$

The GA becomes a minimization problem with this definition of the fitness function. If $w > k$ then $\beta = 1$ applying the punishing term to the fitness calculation. Otherwise, the punishing term is not applied.

3.2 Operators

As stated prior, the effectiveness of a GA derives from the implementation of the different operators used to find a near-optimal solution. Varying certain parameters of these functions allow for faster convergence, a larger search of the problem space, and different solutions to be generated. The three main functions used in one generation of the GA are the selection, crossover, and mutation operations performed on the current population pool.

3.2.1 Reproduction Selection

Based on the fitness function defined for this GA, two parents are selected to produce one child. Two methods of selecting parents for crossover were used based on the fitness. For both implementations, in general, the more fit solutions (which in this case have a lower value produced by the defined fitness function) are chosen more often to reproduce.

The first method for parent selection is the commonly used roulette wheel selection implementation. This method simply creates a wheel where the probability of the selection of any given solution in the current population pool is reflected by

$$p(S_i) = \frac{\bar{f}}{f(S_i) \cdot \sum_{j=1}^{n} \frac{\bar{f}}{f(S_j)}}$$

Where $\bar{f}$ denotes the aggregate total fitness of all the solutions in a given pool.

The second method ranks the solutions based on their fitness where the probability of selecting any given solution in the current population pool is shown by

$$p(S_i) = \frac{\text{rank}(S_i)}{\sum_{i=1}^{n} \text{rank}(S_j)}$$

For the rank implementation, solutions with a lower fitness defined by the function f are more fit and therefore receive a higher rank allowing them to be selected at a higher probability.

In both of these implementations each solution obtains a weight for selection and is simply drawn with its assigned weight using a random number.

After the selection period in the GA has ended, a pool, with the same length as the current population minus the number of elites, is created with pairs of parents for crossover.

3.2.2 Crossover Techniques and Elitism

Now with parents selected for reproduction, two different crossover techniques were used to produce one child from each parent pair. In both crossover operations, the crossover rate was 100%.

The first implementation for crossover was a common method known as uniform crossover. To perform this crossover, a random bit string is generated and each allele of the chromosome is chosen from the corresponding parent represented by the bit. The result is a child with random genetic material from its two parents.

The second crossover method was another popular crossover method known as n-point crossover. Given an integer n as a parameter, n number of split points are created overlaying both of the parents. The result is a child with alternating sections of genetic material from both parents.

After all pairs of parents have produced children according to either of the two crossover methods, a number of elites, which are simply the most fit solutions in the population, from the current population are directly inserted into the new generation with the new collection of children. The new population replaces the old population where the size of the population never changes. For each of the three non-trivial data sets 5, 10, and 100 elites were used respectively.

3.2.3 Mutation Methods

After the new population with the elites and children has been generated, mutation takes place to promote exploration in the problem space. Every solution in the new pool has a random probability to mutate. Similar to selection and crossover, two methods were also used to mutate a chromosome. Both mutation operations have a 5% chance to occur across each data set.

Bit flip mutation was used as the first method. If mutation is successful, a number of bits in the chromosome representation of the solution are activated or deactivated. The number of bits to flip combined with the mutation rate promote exploration of the problem space which is crucial to a problem with a large pool of possible solutions such as the SoS problem.

The second mutation operator was called Force Reduction-Increase (FRedInc) developed by myself. Using the fundamental goal of bit flipping, FRedInc, upon successful mutation, selects a number of bits to exclusively reduce to 0 or increase to 1 with equal probability. The intuition of this mutation operator derives from the problem of reducing or increasing solutions when the desired k is significantly smaller or larger than the average solution in the population. Therefore, it accelerates exploration of the problem space by mutating solutions directly closer to the desired value of k.

3.3 Convergence Criteria

In order to obtain results, the GA must stop. The algorithm stops whenever one of three criteria are met first. The first convergence condition is after a certain number of generations, the GA on $\mathcal{S}_1$ stops after 50 generations and the algorithm on data sets $\mathcal{S}_2$ and $\mathcal{S}_3$ stop after 100 generations. The algorithm can converge before the given number of generations pass based on if $\gamma \leq \epsilon \leq 1.0$, where $\gamma \in \{0.99, 0.999, 0.9999\}$ and $\epsilon = \frac{k-\bar{f}}{k}$. Finally, the algorithm can converge is the optimal solution is reached. That is, if any solution is actually equal to k.

4 Simulated Annealing Explanation and Implementation

A different approach to solving the SoS problem was to use Simulated Annealing. The SA approach significantly reduced the computational complexity time of finding a solution but with only one solution to consider at a time. Similar to the Genetic Algorithm, the initial solution was simply a randomly generated subset. Two different perturbation functions were used to modify the single solution, the solution was replaced if the perturbed solution had a better fitness or a random number, $r \in (0, 1)$, was below a value ϕ which was defined as

$$\phi = \exp\left(\frac{-(f(S) - f(S_p))}{T}\right)$$

Where S is the current solution, S_p is the perturbed solution, and T is the temperature.

To test the affect of choosing a worse solution compared to the current solution, the sigmoid function was also used and tested in $\mathcal{S}_2$.

4.1 Hyper-parameters

Several hyper-parameters are available to modify in the Simulated Annealing process. The first is temperature which directly affects the functions responsible for choosing a worse solution. The number of the inner loop are also variable. Parameters α and β scale the temperature and iterations respectively where $\alpha < 1.0$ and $\beta > 1.0$. In this implementation, $\alpha = 0.95$ and $\beta = 1.01$ with the number of iterations initialized to 1000. The temperature varied between data sets.

4.2 Perturbation Functions

The two mutation operators from the Genetic Algorithm used on the SoS problem were adapted for the SA algorithm. Therefore, bit flip and FRedInc were used as perturbation functions and their operation was the exact same as in the GA. However, they are always applied when the perturb function is called on S and not with a 5% chance as in the case of GA mutation.

4.3 "Foolish" Hill Climbing

The removal of the secondary condition for replacing S, which is if $r < \phi$, defines the "Foolish" Hill Climbing modification of the SA algorithm. It is called foolish because a worse solution given by S_p is never chosen over the current S meaning that the algorithm often converges prematurely to a local optima. The "Foolish" Hill Climbing algorithm was used to test if SA and GA methods could do better than a naïve search of the problem space.

5 Results and Analysis

5.1 Genetic Algorithm

For the Genetic Algorithm, all combinations of the three implementations of selection, crossover, and mutation were tested to determine the best configuration for finding a near-optimal solution to the SoS problem. To compare solutions generated from the GA, a metric was defined as the ratio between the estimated solution and the value of k. The toy data set is not presented as the GA was able to always find the optimal solution and little difference was present among operators.

Selection-Crossover-Mutation	# of Gen.	Best Soln.	Metric	Avg. Fitness
rank-npt-flip	15	2495	0.9980	23.15
rank-npt-FRedInc	15	2495	0.9980	21.00
rank-uniform-flip	50	2392	0.9568	183.06
rank-uniform-FRedInc	50	2432	0.9728	224.35
wheel-npt-flip	17	2497	0.9988	13.62
wheel-npt-FRedInc	5	2500	1.0000	193.12
wheel-uniform-flip	19	2500	1.0000	50.60
wheel-uniform-FRedInc	6	2498	0.9992	19.00

Figure 2: Performance of a Genetic Algorithm with Different Operators on $\mathcal{S}_1$, $k = 2\,500$, $0.99 \leq \epsilon \leq 1.00$, and npt = 10.

For the first two data sets the number of generations often did not exceed the maximum which meant that the GA often found an optimal solution within the bounds for ϵ. In $\mathcal{S}_1$, $k = 2\,500$ was not known to be possible as an exact solution however two of the configurations found it.

Across all three data sets the Genetic Algorithm results were near-optimal solutions. Most surprisingly, the GA on $\mathcal{S}_3$, which has a problem space of $2^{10\,000}$, found solutions with metric scores greater than 0.9999. Though solutions were found in $\mathcal{S}_3$, the GA never converged early implying that more generations were required. The GA not only found good results, but found extremely competitive results as seen in $\mathcal{S}_2$ where the solution found differed from k by only 6.

Selection-Crossover-Mutation	# of Gen.	Best Soln.	Metric	Avg. Fitness
rank-npt-flip	100	2 494 902	0.9980	7 460
rank-npt-FRedInc	35	2 499 820	0.999928	2 396
rank-uniform-flip	100	2 465 942	0.9864	92 844
rank-uniform-FRedInc	100	2 498 396	0.9994	89 649
wheel-npt-flip	26	2 499 950	0.99998	2 318
wheel-npt-FRedInc	12	2 499 924	0.9999696	1 741
wheel-uniform-flip	100	2 999 994	0.9999976	4 831
wheel-uniform-FRedInc	7	2 499 989	0.9999956	1 408

Figure 3: Performance of a Genetic Algorithm with Different Operators on $\mathcal{S}_2$, $k = 2\,500\,000$, $0.999 \leq \epsilon \leq 1.00$, and npt $= 50$.

Selection-Crossover-Mutation	# of Gen.	Best Soln. $(k - estimation)$	Metric	Avg. Fitness
rank-npt-flip	100	623	0.99997508	331 755
rank-npt-FRedInc	100	892	0.99996432	26 742
rank-uniform-flip	100	infeasible	1.06	1 165 096
rank-uniform-FRedInc	100	460 017	0.982	1 111 094
wheel-npt-flip	100	292	0.99998832	129 667
wheel-npt-FRedInc	100	319	0.99998724	106 773
wheel-uniform-flip	100	46	0.99999816	329 120
wheel-uniform-FRedInc	100	61	0.99999756	32 965

Figure 4: Performance of a Genetic Algorithm with Different Operators on $\mathcal{S}_3$, $k = 25\,000\,000$, $0.9999 \leq \epsilon \leq 1.00$, and npt $= 100$.

The largest noticeable difference in performance was between the mutation operators. Force Reduction-Increase usually converged much faster than the flip mutation operator. This suggests that FRedInc explores the problem space much earlier than the flip implementation and therefore has a larger average fitness in the final generation. The downside to this is that the GA might converge to a slightly less optimal value than if it were given more time. However, if time is a restriction, then FRedInc yields near-optimal solutions much faster than the flip operator.

Regarding selection techniques, there was not a significant difference in parent selection across all three data sets. However, when paired with crossover, the selection methods vary in performance. The best of these combinations was with roulette wheel and uniform crossover. This combination with both mutation methods found competitive solutions where FRedInc converged faster and earlier than the flip operator. On the other hand, the worst combination involved rank selection with uniform crossover. At worst, no solution was found in $\mathcal{S}_3$ and in the other two data sets significantly worse solutions were found.

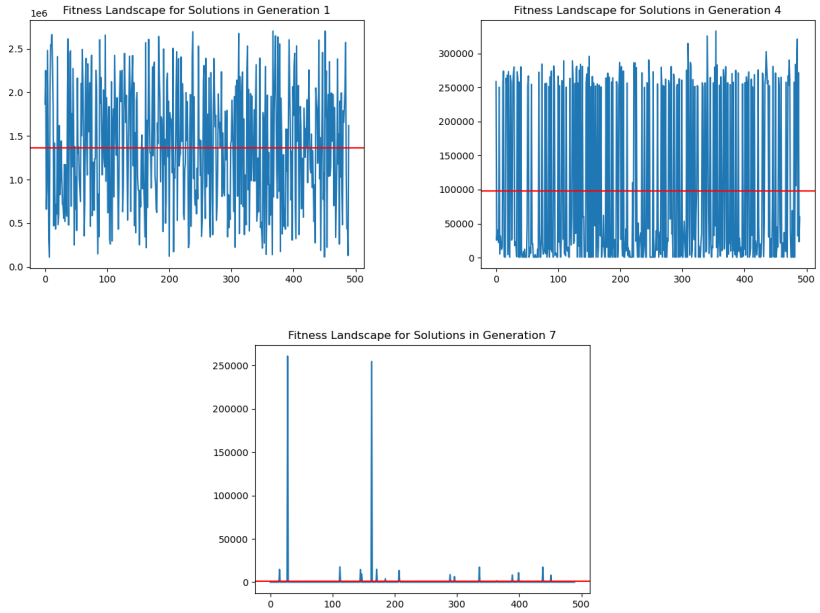

Figure 5: Example of rapid reduction using Force Reduction-Increase to obtain a near optimal-solution. Used in a GA with wheel-uniform-FRedInc in $\mathcal{S}_2$. The obtained solution only differed from the value of k by 11.

5.2 Simulated Annealing

The Simulated Annealing implementation generated results from all data sets except $\mathcal{S}_3$ due to the large amount of time to copy solution chromosomes for the perturbation function. Despite this, the SA performance is clearly highlighted in the other two data sets. In both data sets where the SA was tested, the same α, β and number of iterations were used.

Algorithm-Perturbation	# Iterations	Best Soln.	Metric
SA-flip	52 815	2500	1.0
SA-FRedInc	57 140	2500	1.0
Fool-flip	1005	2500	1.0
Fool-FRedInc	2068	2500	1.0

Figure 6: Performance of Simulated Annealing on $\mathcal{S}_1$ with $T = 1\,000, \alpha = 0.95, \beta = 1.01$, $iterations = 1\,000$, and $0.999 \leq \epsilon \leq 1.00$.

The Simulated Annealing approach, similar to the GA, resulted in optimal or near-optimal solutions in almost all cases. The most important difference in performance is between the two perturbation functions. The flip operator often did better than FRedInc. At worst, the FRedInc operator did not produce a result in the second data set. To overcome this, the sigmoid function was used to determine when to select a worse solution instead of ϕ. This resulted in the SA algorithm with FRedInc to find a near-optimal solution, but after far more iterations than the flip counterpart.

Function-Algorithm-Perturbation	# Iterations	Best Soln.	Metric
Phi-SA-flip	1 560	2 499 760	0.999904
Phi-SA-FRedInc	N/A	N/A	N/A
Sigmoid-SA-flip	671	2 499 793	0.9999172
Sigmoid-SA-FRedInc	24 317	2 499 876	0.9999504
N/A-Fool-flip	558	2 499 954	0.9999816
N/A-Fool-FRedInc	1 216	2 499 914	0.9999656

Figure 7: Performance of Simulated Annealing on $\mathcal{S}_2$ with $T = 10\,000, \alpha = 0.95, \beta = 1.01$, $iterations = 1\,000$, and $0.9999 \leq \epsilon \leq 1.00$.

Surprisingly, the "Foolish" Hill Climbing implementation arrived a near-optimal solution rather than a significantly worse local optimal solution. This suggests that the perturbation function combined with the initial solution, which was randomly generated, alone combine to yield strong results. However, the Foolish approach with FRedInc resulted in a worse local optimal solution. This observation suggests that the FRedInc function directly relies on selecting worse solutions to explore the problem space to find a near-optimal result.

5.3 Comparison

The Genetic Algorithm, in general, produced better solutions than the Simulated Annealing algorithm. However, the SA approach took far less time and computational effort to arrive at worse near-optimal solutions. This makes SA implementations better than GAs within other algorithms as the GA takes far more computational power and storage. Despite a slower performance, GAs explore more of the problem space by directly crossing over solutions and mutating them over several generations which results in better solutions over time.

6 Conclusion

Evolutionary Computation techniques such as Genetic Algorithms and other methods like Simulated Annealing result in competitive solutions to known hard problems. Both implementations offer several functions and operators to vary to find better solutions. GAs and SAs prove their efficiency by extracting near-optimal solutions to the Sum of Subsets problem. In all three non-trivial data sets, with massive problem spaces, both algorithms found competitive solutions. In addition to finding near-optimal solutions, both algorithms were shown to perform better with different configurations.