Evolutionary Computation Software Project Report

Ethan Beaird

April 24, 2023

Introduction

The minimum vertex cover problem is an optimization problem in graph theory. Given an undirected graph $G = (V, E)$, a vertex cover is a subset $V' \subseteq V$ such that every edge in E is incident to at least one vertex in V'. We wish to find a vertex cover of minimum size, but the minimum vertex cover problem is known to be NP-hard. In this report, we explore several techniques used to tackle this problem – Genetic Algorithms (GA), Simulated Annealing (SA), and Foolish Hill Climbing (FHC).

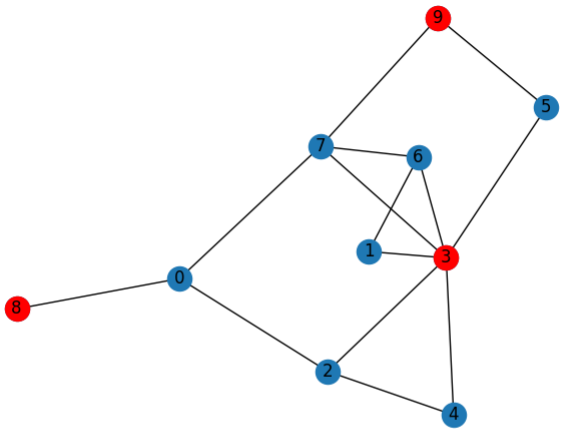
Genetic Algorithms are non-deterministic algorithms used to solve "hard" problems. In a simple genetic algorithm, we create a population of solutions, select candidates based on fitness, perform crossover between parents, and mutate newly produced children which become our new population of solutions. By repeating this process a number of times, we can reach near-optimal solutions for hard problems.

Simulated Annealing is another non-deterministic algorithm used to solve "hard" problems. In simulated annealing, we create an initial solution and perform perturbations on this solution, sometimes increasing or decreasing its fitness. If its fitness increases, we accept that new solution. Otherwise, we accept a worse solution according to a temperature parameter, allowing us to balance exploration and exploitation. By repeating this process a number of times, we can again reach near-optimal solutions for hard problems.

Foolish Hill Climbing is a lot like Simulated Annealing, except that we only accept new solutions that increase in fitness. This can often lead us to getting stuck in local optima. In general, Foolish Hill Climbing is expected to perform worse than both Genetic Algorithms and Simulated Annealing.

Chromosome and Fitness

To represent a vertex cover for an arbitrary graph $G = (V, E)$, we can use a binary encoding scheme, i.e. a bitstring of size ν. In this representation, a 1 in position i corresponds to the i-th vertex being included in the vertex cover. This representation is compact and allows us to easily manipulate an individual. The representation is demonstrated below.



Chromosome

0001000011

Infeasible chromosomes may occur, i.e. chromosomes that are not actually vertex covers. To avoid these infeasible chromosomes, our initial population generated is guaranteed to be feasible, and we do not allow modification to an individual if it would make that individual infeasible. Our initial population is generated by producing a random bitstring of size ν until it is feasible.

The fitness function is quite simple, and is defined as

$$fitness(chromosome) = -(size)$$

where size is the number of vertices in the vertex covering. The fitness function effectively rewards feasible vertex coverings with fewer vertices by assigning less negative utility to them.

Operators

GA - Selection Operators

Tournament:

1. Randomly select 2 individuals from the population.
2. Choose the individual with higher fitness 75% of the time. Otherwise, choose the individual with lower fitness.
3. Repeat this process 100 times to produce 100 parents in the parent pool.

Roulette:

1. Assign each individual in the population a probability proportional to its relative fitness, i.e. divide its fitness by the total fitness of the population. As such, individuals with higher fitness have a higher probability of being selected.
2. Since our probabilities sum to 1, we create a "pseudo-cumulative distribution function" that represents the cumulative probabilities of all individuals in the population. This involves dividing the range of probabilities (0 to 1) into subintervals that correspond to the probabilities of each individual.
3. Generate a random number between 0 and 1 and select the individual that corresponds to the subinterval that contains the random number.
4. Repeat this process 100 times to produce 100 parents in the parent pool.

GA - Crossover Operators

Uniform

1. Randomly select 2 parents from the parent pool.
2. For each position in the bitstring of the children, randomly select a bit from either parent with probability = 0.5 until the entire child's bitstring has been created.
3. Repeat this process with the selected parents twice to produce 2 children. If either of the children are infeasible, we cancel crossover for that child and produce a copy of one of the parents instead.
4. Repeat this process with new parents 50 times to produce 100 children in the child pool.

Single-point

1. Randomly select 2 parents from the parent pool.
2. Randomly select a crossover point between 0 and the parents' length - 1.
3. Produce the first child by copying parent 1's bits before the crossover point and parent 2's bits after the crossover point. If the first child is infeasible, we cancel crossover and produce a copy of the first parent instead.
4. Produce the second child by copying parent 2's bits before the crossover point and parent 1's bits after the crossover point. If the second child is infeasible, we cancel crossover and produce a copy of the second parent instead.
5. Repeat this process with new parents 50 times to produce 100 children in the child pool.

GA - Mutation Operators

Bitflip

1. Randomly select a bit from the child and flip it.
2. If the child is infeasible, we cancel mutation and flip the bit back.

Swap

1. Randomly select two bits from the child and swap their places.
2. If the child is infeasible, we cancel mutation and swap the bits back.

SA & FHC - Perturbation Operators

The operators used for perturbation in Simulated Annealing and Foolish Hillclimbing are exactly the mutation operators used in the Genetic Algorithm.

Parameters Used

Genetic Algorithm Parameters

- Population size = 100
- Crossover rate = 0.9
- Mutation rate = 0.05
- Elitism = enabled
- Generational GA = true
- Terminates after 5000 generations pass

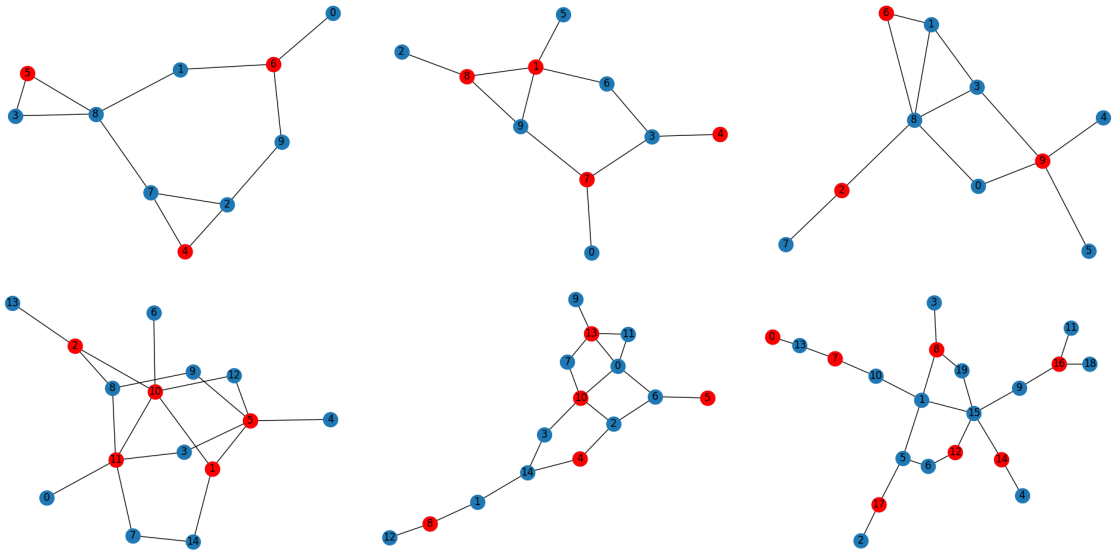
Simulated Annealing Parameters

- $T_0 = 100$
- $\alpha = 0.99$
- $\beta = 1$
- $I_0 = 10000$
- Cooling type = Geometric

Datasets

Toy Data Set

In order to verify that both my Genetic Algorithm and Simulated Annealing algorithms work, we construct a contrived dataset with known optimal solutions. We briefly examine 6 graphs with known minimum vertex covers, pictured below.



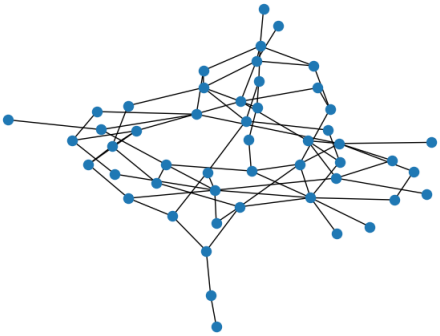
Since these toy problems are relatively small, one can verify by inspection that the vertex covers in red are indeed minimum (or they should be able to produce a minimum vertex cover of the same size). These vertex covers above were found by my genetic algorithm; the optimal solution was reached for each of my toy problems (using tournament selection, uniform crossover, and bitflip mutation).

Likewise, simulated annealing did not produce the exact same vertex covers pictured above, but they were still of the same size as the optimal solution (using bitflip perturbation). Overall, both my genetic algorithm and simulated annealing were able to produce optimal results for small toy problems, and it is reasonable to assume that they can be used for larger problems.

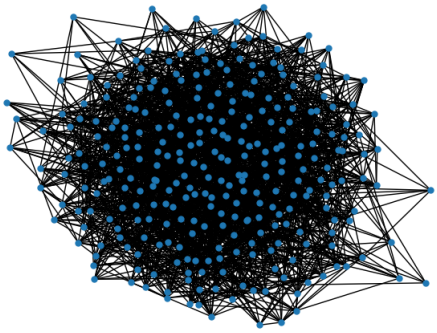
Other Data Sets

We will also test on medium and large data sets. The aforementioned small, or "toy" datasets all consisted of graphs with less than 16 vertices. We will also examine randomly-generated graphs with 50 vertices and 300 vertices, pictured below. These graphs, like the toy graphs, were produced using Python's NetworkX module.

The first graph has 50 vertices and probability of an edge = 0.05.



The second graph has 300 vertices and probability of an edge = 0.05.



One would likely conclude that finding a minimum vertex covering for these graphs by hand would prove to be incredibly difficult. Optimal solutions for these graphs are unknown (since they are randomly generated and far too large to verify by hand).

Results

Data

We tested different combinations of our selection, crossover, and mutation operators for our Genetic Algorithm. We tested different perturbation operators for our Simulated Annealing and Foolish Hill Climbing. These results are averaged over 5 simulations for each entry.

Below are the results for our Small dataset (in particular, the first graph of our Toy Data Set).

GA	Selection	Crossover	Mutation	Best Min. Cover	Average Min. Cover
	Tournament	Uniform	Bitflip	3	3
	Tournament	Uniform	Swap	3	3
	Tournament	Single-point	Bitflip	3	3
	Tournament	Single-point	Swap	3	3
	Roulette	Uniform	Bitflip	3	3
	Roulette	Uniform	Swap	3	3
	Roulette	Single-point	Bitflip	3	3
	Roulette	Single-point	Swap	3	3
SA	**α / β**	**T0 / I0**	**Perturbation**	**Best Min. Cover**	**Average Min. Cover**
	0.99 / 1	100 / 10000	Bitflip	3	3
	0.99 / 1	100 / 10000	Swap	5	6
FHC	**Perturbation**	**Best Min. Cover**	**Average Min. Cover**		
	Bitflip	5	5.4		
	Swap	5	6.2		

All combinations of selection-crossover-mutation operators for our GA as well as our bitflip SA performed equally well. This is expected, since the dataset is very small and contrived. Our swap SA performed poorly; in fact, it performed worse than bitflip FHC.

Below are the results for our Medium Data Set ($\nu = 50$).

GA	Selection	Crossover	Mutation	Best Min. Cover	Average Min. Cover
	Tournament	Uniform	Bitflip	14	14.6
	Tournament	Uniform	Swap	15	15.2
	Tournament	Single-point	Bitflip	14	15.2
	Tournament	Single-point	Swap	14	14.6
	Roulette	Uniform	Bitflip	14	14.4
	Roulette	Uniform	Swap	14	14.6
	Roulette	Single-point	Bitflip	14	15
	Roulette	Single-point	Swap	14	14.2
SA	**α / β**	**T0 / I0**	**Perturbation**	**Best Min. Cover**	**Average Min. Cover**
	0.99 / 1	100 / 10000	Bitflip	16	17.8
	0.99 / 1	100 / 10000	Swap	16	17.2
FHC	**Perturbation**	**Best Min. Cover**	**Average Min. Cover**		
	Bitflip	27	30		
	Swap	27	31.8		

Nearly all combinations of selection-crossover-mutation operators for our GA performed identically in regards to the best minimum cover. Still, roulette-single-point-swap achieved the best average minimum cover size. SA had little variation in performance between perturbation operators; FHC also behaved in a similar manner.

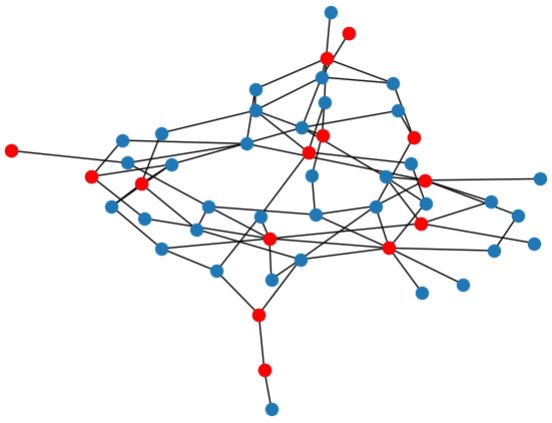
Below are the results for our Large Data Set ($\nu = 300$).

GA	Selection	Crossover	Mutation	Best Min. Cover	Average Min. Cover
	Tournament	Uniform	Bitflip	41	44.2
	Tournament	Uniform	Swap	41	42
	Tournament	Single-point	Bitflip	41	44
	Tournament	Single-point	Swap	43	45.6
	Roulette	Uniform	Bitflip	38	40
	Roulette	Uniform	Swap	42	42.8
	Roulette	Single-point	Bitflip	42	44
	Roulette	Single-point	Swap	45	47
SA	**α / β**	**T0 / I0**	**Perturbation**	**Best Min. Cover**	**Average Min. Cover**
	0.99 / 1	100 / 10000	Bitflip	114	137
	0.99 / 1	100 / 10000	Swap	161	161
FHC	**Perturbation**	**Best Min. Cover**	**Average Min. Cover**		
	Bitflip	143	165.2		
	Swap	161	161		

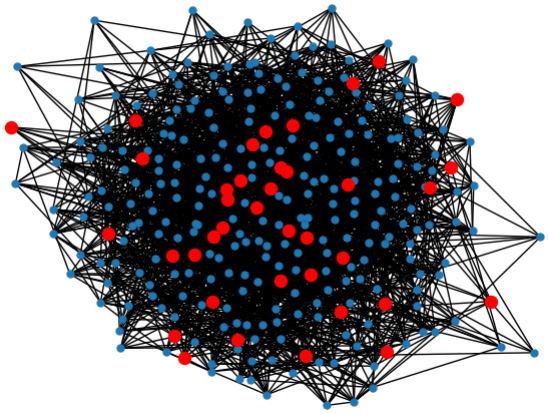
In general, trials with tournament selection performed marginally better than trials with roulette selection. Interestingly, the best results were produced by roulette-uniform-bitflip. Our bitflip SA performed monumentally better than our swap SA; the difference is not as vast in FHC but is still notable.

Minimum Vertex Covering Solutions

Below is the minimum vertex covering found for our medium dataset (found by roulette-single-point-swap GA).



Below is the minimum vertex covering found for our large dataset (found by roulette-uniform-bitflip GA).



Conclusion

In general, there was little difference between different combinations of selection-crossover-mutation operations for our Genetic Algorithm. It is perhaps feasible to state that trials with tournament selection, in general, performed marginally better (since both of the optimal solutions for our medium and large dataset were found in this manner). However, although the elapsed time for each trial was not recorded, trials with tournament selection were substantially slower than trials with tournament selection. It would perhaps be advisable for future expansions of this project to record this time data for reference.

Although our Simulated Annealing algorithm performed decently for smaller datasets, the difference in performance between the Genetic Algorithm and Simulated Annealing grew more apparent with larger problem sets. Still, the bitflip perturbation operator produced better results than the swap perturbation operator. Foolish Hill Climbing performed generally poorly for all tested datasets.

My population size, crossover rate, and mutation rate parameters for my Genetic Algorithm were well chosen and produced meaningful results. Were I to repeat the project, I would have definitely changed my means of termination to detect convergence in my solution. 5000 generations was painfully slow for tournament selection. My chosen parameters for Simulated Annealing produced no issues, and they allowed my algorithm to fully explore/exploit the state space.

One major issue I repeatedly ran into during implementation was unintended modification of chromosomes during crossover and mutation. My chromosomes preserved state as lists in Python for ease of use. When performing crossover or mutation, I made sure to check if that operation would make the chromosome infeasible; if it would, I cancelled the operation. Still, chromosomes that were not selected for crossover or mutation were changing their state erratically. I eventually solved this issue by making deep copies of each chromosome before interacting with them. I can't say that I entirely understand why those chromosomes were being modified, but I have decided it to be due to one of those weird intricacies of Python.

If I had to choose one selection-crossover-mutation operator to use forever in testing even larger problems, I would have to choose tournament-uniform-bitflip. Although roulette selection produced marginally more optimal solutions in several instances, I would prefer to reach a solution before the heatdeath of the universe, and I am not confident in roulette selection's ability to do so. Uniform crossover produced better average results than single-point crossover. Bitflip mutation seems to have a trend of producing better average results than swap mutation as ν increases.

I took a graduate course in Graph Theory at my previous university, and I drowned in proofs.

I appreciate those proofs, but it is very nice to finally dive into the programming aspects of Graph Theory. It is definitely a refreshing change of pace from the rigorous proofs. I think this project has both furthered my understanding of Graph Theory and Genetic Algorithms (since there is no better way to learn about Genetic Algorithms than actually creating one).