How to Solve the Independent Set Problem with Genetic Algorithm and Simulated Annealing

Shadow Pritchard

16 December 2019

1 Introduction

Finding the maximal independent set is an NP-hard problem that can be solved using non-deterministic algorithms. For this paper both a genetic algorithm and simulated annealing is used to solve the independent set problem. Different parameters and methods are used and compared to ascertain the relative difference in effectiveness of the respective methods. Testing shows that a genetic algorithm with tournament selection, single point crossover, and single bit flip mutation produces the best results. Foolish hill climbing is also tested, and the results are surprisingly good.

2 Chromosome and Fitness

2.1 Chromosome

The chosen chromosome format is a standard binary bit string. The length of the chromosome is N, the number of vertexes in the data set. Each bit indicates whether the associated vertex is included or not. '1' in the ith index means that the ith node is included in the independent set. '0' in the ith position means the ith node is not included in the independent set.

A chromosome is considered feasible if all nodes included have no connecting edges to any other node included in the chromosome. If a chromosome is categorized as infeasible it is fixed up by randomly selecting a '1' in the chromosome and flipping it to '0'. This is repeated until the chromosome is not infeasible.

2.2 Fitness

Fitness is the number of nodes included in a feasible chromosome. This is calculated by summing the chromosome since each included node is a 1 and each not included is a 0, so the sum is the number of included chromosomes. Infeasibles do not need a separate fitness function as all chromosomes are fixed up before fitness calculations.

3 Operators

3.1 Genetic Algorithm

The genetic algorithm uses two methods of crossover and two methods of mutation. Elitism is implemented to further explore high performers. We carry over the two best in the population to the next population.

3.1.1 Crossover

Binary strings have only a few known crossover methods that are effective, so two of the most popular ones were selected: uniform crossover and single point crossover.
Uniform crossover takes in two parents (P1, P2) and generates a random bit string of 1's and 0's (called u) then creates one child (C1) by taking the ith index from P1 if the ith index of u is 1, otherwise the bit is copied from P2. C2 is generated the same way but flipping where

to get the bit the parents. So for C2 if the ith bit is a 1 then the bit is copied from P2 and if the bit in u is a 0 then C2's ith bit is copied from P1.
Single point crossover also takes in two parents and outputs two children. A random number less than N is generated and at that index the parents are split and C1 copies all the bits up to the random number from P1 then the rest from P2. C2 uses the same split point but gets its bits from P2 for the first portion and P1 for the second.

3.1.2 Mutation

Two different mutation operators were used for the genetic algorithm: single bit flip and multi-flip. Single bit flip is rather self explanatory: a random index is selected and the bit is 'flipped'. If the bit is 1 it is changed to 0 and if it is 0 it is changed to 1.
Multi-flip is similar to single bit flip except it has a chance to change more than one bit. A random number from 0 to $\frac{N}{10}$ is generated and that many random indexes are selected and flipped. Since each index is random with replacement the same bit could be flipped twice in the same mutation operation. $\frac{N}{10}$ was selected as an upper bound as too large of a mutation typically hurt convergence.

3.2 Simulated Annealing

Simulated annealing [1] requires a perturbation function that will change the chromosome enough to find good results but not so much that it cannot converge. Binary strings do not have as many effective methods as permutation strings, so several methods were tested that turned out to be useless. Single bit flip (P1) was able to produce adequate results but multi-flip struggled to find anything better than sub par chromosomes. Thus, we used a modified multi-flip method (P2). Each bit has a low probability ($p = 0.01$) to be flipped, so with a very low probability all the bits in the string could be flipped. Most of the time only a few bits are flipped.

To avoid getting stuck in a local optima for too long, a 'kick' function was used like from [2]. After 500 outer iterations where there has not been a new best found, the string is 'kicked' to get it out. The kick function iterates through the string and with a probability of $p_k = .1$ a bit is changed selected. This bit then is set to 0 or 1 with equal chance. The existing value of the selected bit has no bearing on the new value. If the bit is not selected, then it remains the same. The kick operator is inspired by uniform crossover and attempting to create something that is stronger than a regular single bit flip operator.

The final operator is for foolish hill climbing. This is essentially the same algorithm as simulated annealing with the only difference the condition for when to accept new solution. In simulated annealing a new solution is accepted when it has a better fitness or with a decreasing random probability based on the difference in fitness and the temperature. Foolish hill climbing gets rid of this probability and only accepts a new solution when it has a better fitness. Thus, it is really good at finding a local optima but it can get stuck easily. The kick operator is left in for all of the tests.

4 Parameters

The parameters were mostly determined by trial and error and by referencing the source papers for simulated annealing [1], [2].

4.1 Genetic Algorithm

The pool size for the genetic algorithm was set to $0.6 * N$. This number seemed to work rather well for most data set sizes.

Variable	value
Data set size	N
pool size	$0.6 * N$
max generations	20,000
max no improvement	2500
crossover rate	100%
elites	2

Table 1: Genetic algorithm parameters

4.2 Simulated Annealing

Simulated annealing requires several initial values that have to do with temperature and the kick operator. I_0 is the initial value for the inner loop counter. This is increased once per outer loop iteration by multiplying the current number of inner iterations and β. The temperature is adjusted by multiplying at each iteration of the outer loop by α. Once the temperature reaches 1, it stops being reduced. Thus, at the end of each outer iteration $T = \max(1, T * \alpha)$.

Variable	value
data set size	N
I_0	100
T_0	50
β	1.01 *or* 1.001
α	.985
loops till kick	500

Table 2: Simulated annealing parameters

5 Dataset

The data were generated by a custom python program that can generate sets of various sizes as well as contrived sets with known optimals. A non-contrived data set is generated by creating an upper triangular matrix of 0's and 1's where a value of 1 at (i, j) represents an edge connecting node i to node j. The probability that a 1 is inserted at any given point is p, usually set to 0.5. The bottom half of the matrix is filled in with the appropriate values from the top half of the matrix and the primary diagonal is all 0's. This results is an adjacency matrix that with all the information we need. The reason to keep the bottom half of the matrix is ease of use. While only the top half is needed to hold all the information we need, it is easier to access if we can just check one row for a given node and know that contains all the edges we care about.

The contrived examples are generated by creating a matrix where every node where the node index mod 3 is not equal to 0. The nodes that are index mod 3 equal to zero, are not connected to any other node divisible by 3 but is fully connected to all other nodes. Thus, for $N = 5$ the matrix generated is:

$$\begin{matrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{matrix} \tag{1}$$

The 1st row (row 0) is connected to everything but the 4th row (row 3) and rows 1, 2, 4 are fully connected. The independent set is then clearly 0, 3. This generalizes to larger matrices and the optimal independent set size, S, can be calculated using

$$S = ceiling(\frac{N}{3}) \tag{2}$$

For this paper, we only use contrived examples in the results, as we need some metric to compare our results to.

6 Results

The results for each table was generated by running the respective algorithm with stated parameters five times. The times were recorded on an Ubuntu machine with a Ryzen 5 1600x CPU. The code is in python 3.6. Times will vary based on system running it.

Selection	Crossover	Mutation	Best Fit	Avg Gens	Avg Fit
roulette	unif	singleflip	17	4182.8	16.4
roulette	unif	multiflip	17	3285.0	16.0
roulette	single	singleflip	17	2713.6	16.4
roulette	single	mutliflip	17	4593.6	16.4
tournament	unif	singleflip	17	479.2	17.0
tournament	unif	multiflip	17	910.6	17.0
tournament	single	singleflip	17	417.8	17.0
tournament	single	multiflip	17	986.2	17.0

Table 3: Set size 50. Genetic Algorithm Results. Stopping condition: found optimal (17), best has not changed for 2500 gens, or exceeded 300 seconds

Selection	Crossover	Mutation	Best Fit	Avg Gens	Avg Fit
roulette	unif	singleflip	30	5037.2	27.8
roulette	unif	multiflip	28	5601.8	26.6
roulette	single	singleflip	31	5121.8	30.0
roulette	single	mutliflip	28	5058.2	26.4
tournament	unif	singleflip	34	519.4	34.0
tournament	unif	multiflip	34	2099.4	34.0
tournament	single	singleflip	34	603.2	34.0
tournament	single	multiflip	34	2205.4	34.0

Table 4: Set size 100. Genetic Algorithm Results. Stopping condition: found optimal(34), best has not changed for 2500 gens, or exceeded 300 seconds

Selection	Crossover	Mutation	Best Fit	Avg Gens	Avg Fit
roulette	unif	singleflip	38	5037	36.6
roulette	unif	multiflip	37	5529	34.0
roulette	single	singleflip	44	5606	42.4
roulette	single	mutliflip	40	6736	37.8
tournament	unif	singleflip	50	3110	50
tournament	unif	multiflip	50	4884.6	50
tournament	single	singleflip	50	3129	50
tournament	single	multiflip	50	3142	50

Table 5: Set size 150. Genetic Algorithm Results. Stopping condition: found optimal (50), best has not changed for 2500 gens, or exceeded 600 seconds

The stopping conditions for the simulated annealing was dependent on size of the set and if the algorithm reached the known optimal value. For a set size of 50, 60 seconds was the upper time limit, 100 had a time limit of 180 seconds, and 150 had a limit of 600 seconds. These bounds were chosen as most of the smaller test sets converged before the time limit was reached. The only methods to reach the limit are shown to be clearly worse, or the set size is quite large, making the run time to get the optimal too long for multiple tests.
α is set to 0.985 in all tests. The initial values for temperature and inner iterations are set to 50 and 100 respectively. Temperature has a minimum value of 1 that it can never go below.

Set Size	beta	Perturbation	avg time (s)	Best Fit	Avg Gens	Avg Fit
50	1.001	P1	3.1	17	44405	17.0
50	1.01	P1	62.0	17	704456	17
50	1.001	P2	43.8	17	457936	17
50	1.01	P2	64.9	17	737771	17
100	1.001	P1	270.8	34	581939	33.2
100	1.01	P1	251.5	34	591171	33.2
100	1.001	P2	300	32	838309	31.2
100	1.01	P2	300	32	885974	31.4
150	1.001	P1	600	49	561038	47.2
150	1.01	P1	602	49	619930	47.8
150	1.001	P2	600	45	764659	44.6
150	1.01	P2	603	45	827388	44.4

Table 6: Simulated Annealing Results. $I_0 = 100$, $T_0 = 50$, $\alpha = .985$

The next table is for foolish hill climbing. The contrived results are surprisingly good, but replicable. After running the standard 5 times I verified each result by running extra times and debugging to ensure results were not being carried over from other runs. The results will be discussed further in the following section.

Set Size	Perturbation	avg time (s)	Best Fit	Avg Gens	Avg Fit
50	P1	17	3.5	130200	17
50	P2	.9	17	7960	17
100	P1	10.9	34	270500	34
100	P2	300	33	692620	33
150	P1	14.7	50	240780	50
150	P2	51.8	50	51740	50

Table 7: foolish hill climbing Results. $\alpha = .985$

The final table is the results from a sparse matrix of generated data. Since this data set is not contrived, the optimal is not known. However, it contrasts the performance difference of a genetic algorithm and foolish hill climbing.

Set Size	Gen best	Gen Avg	Fool best	Fool Avg	FNK best	FNK avg
50	12	12	12	11.8	12	11.8
100	15	13.8	16	14.0	15	13.4
150	15	14.6	18	17.0	15	13.8

Table 8: foolish hill climbing vs genetic algorithm (tournament, single point crossover, single point mutation (mutation rate 0.25)) FNK is foolish hill climbing with no kick

7 Discussion

The results for the genetic algorithm show that tournament is considerably better than roulette. The cause of this is often the population pool all has the same fitness, so roulette is effectively random. An alternative would likely be rank, though there would still be many ties. Tournament helps by picking the better of the two randomly chosen chromosomes without taking into account the magnitude of the difference of the fitness', much like rank selection does.

Uniform crossover is slightly worse in most cases, but occasionally converges to the optimal slightly faster. presumably this is because the change is quite substantial between generations so the population changes quickly, but is volatile so it is difficult to approach the optimal.

The single flip mutation is slightly better than multi-flip for likely the same reason single point crossover is better: the change is too much between generations. While random change is needed, a balance should be struck so the algorithm still can search in the space near strong genes.

Simulated annealing gives slightly worse results than genetic algorithms, but for many cases the optimal is still found. Simulated annealing does take longer, so in the large data sets

that do not quite converge, it is likely they would given more time.
Oddly enough, foolish hill climbing has very, very good results. However, if we look at the state space of the contrived problems the only local optimal is the global optimal. This is the one situation where foolish hill climbing succeeds, because it only will go up hill and it is quite good at that. So the high performance on the contrived examples is surprising, but can be explained. The results of Table 8, however, are harder to justify. One can assume that the same issue has occurred, and the foolish hill climber is swiftly moving to a local optima which happens to be a global optima or near a global optima. Then, if it gets stuck, the kick operator solves the issue and pulls it out. When the kick operator is removed, the performance is still good, but it is not inexplicably good. Doing foolish hill climbing with a kick has potential to be a fast and effective way to solve the independent set problem, but it is not fully tested.

8 Conclusion

The independent set problem is easily solvable by genetic algorithm and simulated annealing. The performance varies greatly across algorithms, operators, and parameters, but these can be discovered through some work with contrived data sets. Genetic algorithm with tournament selection with single point crossover and single bit flip mutation is the best method. It converges as fast or faster than most of the other methods and reaches the optimal in every tested case. Foolish hill climbing is extremely good at finding local maxima, and the contrived examples only have one maximum, so foolish hill climbing performs very well for this our specific tests but it may not generalize well. The non-contrived cases also worked well if the kick operator was used, but further testing would need to be done with cases that have more local maxima to verify the results. In the future contrived examples with attractive local optima should be used on the foolish hill climber with kick operator. The result would likely be similar to a random-restart hill climber but with softer restarts.

9 References

[1] Nahar, S. Sahni, S. and Shragowitz E. "Simulated Annealing and Combinatorial Optimization" 1989.
[2] Enochs, B. Wainwright R. ""Forging" Optimal Solutions to the Edge-Coloring Problem" 2001.