Alex White
Project Notes – Genetic Algorithm for Clique
CS780: Genetic Algorithms, Queens College
Prepared for Professor Goldberg
August 3, 2015

PROBLEM

The Karp problem I sought to solve using a genetic algorithm was clique. I would write a program that would take in an *nxn* adjacency matrix and would over successive generations seek to find a complete subgraph with k nodes, or its closest approximation. My (first) test case is outlined below.

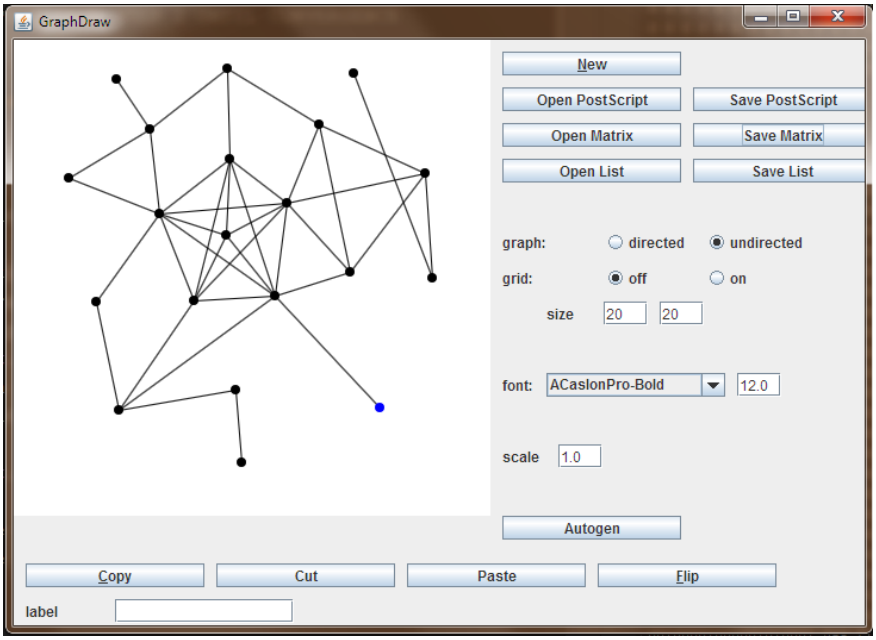

Fig. 1: GraphDraw

I found an excellent resource for creating and reading graph files using adjacency lists and matrices, developed by Keenan Crane:

http://www.cs.columbia.edu/~keenan/Projects/GraphDraw/

Using this I was able to quickly develop graphs with suitable features and output their representation in matrix form. As you can see, the above matrix features twenty nodes. It contains cliques of size three, four, five, and six. You can see here where the program would find the big ones, if it had eyes to see like a human:

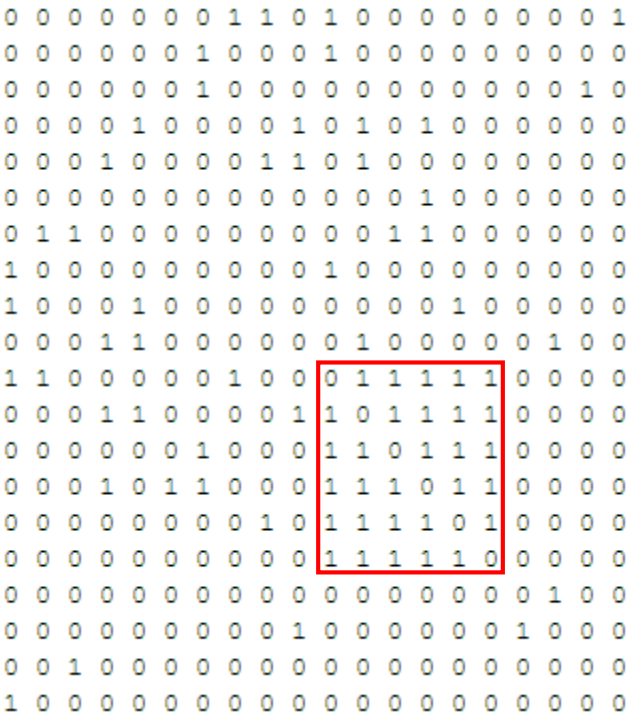

Fig. 2: First Matrix

OPERATORS AND DATA STRUCTURES

Approaching this problem, there were certain considerations. First, how to represent problem solutions. Second, how to measure their fitness. Third, how to store and retrieve those solutions. Fourth, how to mutate the chromosomes. Fifth, how to mate the chromosomes. Sixth, how to select them for mating.

I chose to represent solutions as bit strings. No surprise there. It was necessary to develop a method to populate the initial generation with bit strings of length n featuring exactly k '1's, as is necessary for the problem. The algorithm chooses random integers between zero and the length of the string, puts a '1' at that position, and then counts the '1's in the string. When it has assigned k of them, it stops. It does this until a generation is full.

Fitness is measured by a sets distance from being a complete subgraph, on a percentage scale from zero to one. If there are no edges in a set, it scores a zero. Otherwise it scores the percentage of existing edges as compared to possible edges, calculated as $\frac{n(n-1)}{2}$.

Each generation is stored in a Tree Set, so that I can maintain their chromosomes in sorted order. They are sorted first by fitness, then by the value encoded by their bitstring.

Mutating and crossover need to be respectful, because it's necessary that the number of true bits always be *k*. Therefore these algorithms proceed as follows:

Mutation doesn't flip bits, it instead swaps them with their neighbors according to some probability p. This maintains k '1's in the string. It iterates through the genotype and if a random number exceeds a certain threshold, it swaps the bit with its neighbor.

Crossover is a little more complex. The important feature to maintain when producing a child via crossover are stretches of matching zeros. That is, if both parents feature a zero in a certain position, that child *must* also feature a zero there. Otherwise, anything goes. The algorithm makes a list of positions where either parent features a '1', then randomly assigns *k* '1's to those positions on the child.

Selection is something like fitness proportionate. Since each generation's chromosomes are ordered by fitness, we can iterate over the list of them from highest to lowest fitness. Starting from the top, the selection algorithm chooses a random integer between 0 and 100, if successful removes the current chromosome and adds it to a collection of parents, then starts from the top of the list again. If it misses it keeps going down the list. So, fitter individuals have a higher chance of being chosen by virtue of their postions on the list, but those individuals at the bottom have some chance, too.

TESTING

Once these were in place I could try out the algorithm.

There were a few bugs, mainly one big one: an infinite loop. I didn't want any duplicates (TreeSets won't abide duplicates in any case), so I put in place certain restrictions to prevent the new Generation's constructor from allowing it. This seemed prudent (and was, it turned out). The problem ended up being my test case and the parameters I was using. I tested the above graph with a small (~20) number of generations, and low mutation and crossover rates. The loop kept happening. I tried again by increasing the mutation and crossover rates, to no avail.

Eventually I found the where the loop was happening: turns out that the parent generation was running out of members to select from before the child generation was full. Everything left was a duplicate, and this triggered an infinite search for non-duplicates. It happened when the crossover rate was high because that meant almost the entire generation were candidates, and it happened when the crossover rate was low, because that meant there weren't enough to choose from to build a unique set to the right size.

Turns out the main problem was the size of the generation. Increasing this to 1000, the algorith worked perfectly. The program would now solve the above graph in under twenty generations.

I now turned my attention to a harder problem. Fifty nodes, with a clique of size eleven. See Figure 3.

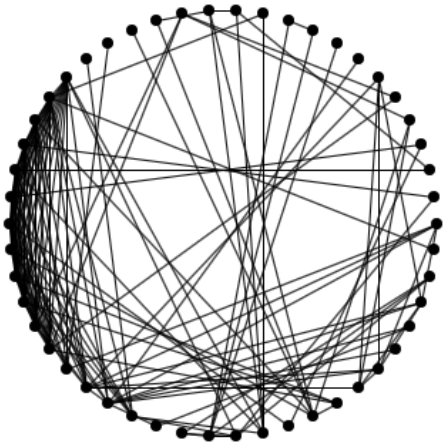

Fig. 3: Final Testing Set

The algorithm was not solving this problem. It converged to some fitness around 0.60 at generation 10. I wondered about this, and analyzing the bitstrings I concluded that there were areas that remained the same in the fittest individuals. This made me question how well my mutation operator was working, so I reevaluated it.

Now, instead of switching neighboring bits, I instructed it to switch random bits. This should now introduce '1's into areas they hadn't been in previously.

This also didn't work. However, it makes sense as a mutator, so I kept it.

Turns out the problem was in the adjacency matrix. The group that I had made complete was off by two columns, so that it wasn't really complete; two columns worth of nodes were connecting to nodes outside the group. Fixing this, the algorithm worked perfectly, in fact, converging after only four generations.

```
Generation 0 at Tue Aug 04 10:17:27 EDT 2015
Average Fitness: 0.1
Best Chromosome: 0100000010000000010101101011010100000000000000000 having fitness: 0.43
==================================
Generation 1 at Tue Aug 04 10:17:27 EDT 2015
Average Fitness: 0.34
Best Chromosome: 0000000000000000000111111101011100000000000000000 having fitness: 0.76
==================================
Generation 2 at Tue Aug 04 10:17:27 EDT 2015
Average Fitness: 0.46
Best Chromosome: 0000000000000000000011111111111000000000000000000 having fitness: 0.9
==================================
Generation 3 at Tue Aug 04 10:17:27 EDT 2015
Average Fitness: 0.57
Best Chromosome: 0000000000000000000011111111111000000000000000000 having fitness: 1
FINISHED
```

Fig. 4: Success Output

CONCLUSION

I need a harder problem to test this on. Unfortunately, all the large test sets available online are in list form, so I'll need to tweak my algorithm a bit.