**Christopher Goes**

**Assignment 2 – Genetic Algorithms – Solving a simple Substitution Cipher**

| Parameter | Final value used |
|---|---|
| Genetic Algorithm | Steady State |
| Fitness Function | Bhattacharyya |
| Crossover Function | PMX |
| Crossover Probability | 0.8 |
| Mutation Function | Single swap |
| Mutation Probability | 1.0 |
| Population Size | 256 |
| Fitness Evaluations | 100,000 |
| Tournament Size | 3 |
| Elitism | Yes |
| Local Search | No |

Parameters I experimented with: crossover probability, fitness function, mutation probability, mutation function, number of evolutions, population size, mutation function used, punishment used.

**Fitness Function**

Varying the fitness function used to Euclidian did not significantly change the deciphering accuracy. In light of this, I decided to stick with Bhattacharyya. This was due to its ease of reading the fitness values, and a slightly faster running (~100-200ms as measured by Linux's "time" tool).

**Crossover**

Initially, I was going to use an Order One crossover, due to its high speed and relative simplicity. However, I ultimately went with PMX, for two reasons. First, for the reason you mentioned in class: it works well for most things. Second, the code was provided, which saved valuable time needed to code everything else.

I experimented with varying the probability from 0.1 to 1.0. Going below 0.7 began to adversely affect performance, and below 0.3 yielded horrible performance for all codes. 0.9 and 1.0 worked well for the test and p1209 codes, which had frequencies close to English. For the others, 0.7 and 0.8 were the sweet spot, with 0.8 ultimately yielding the best performance overall.

**Mutation**

After getting selection working, I threw in a simple swap mutation that would simple swap two random characters of the key, even if they were the same position. Later, I implemented double-swap and shift-swap mutations. The double-swap did not seem to have much if any effect, and my shift-swap

implementation did not work. Therefore, I stuck with single-swap, since it worked “good enough”, and was faster than the other two.

Mutation probabilities from 0.0 to 1.0 were experimented with. No mutation (0.0) was very bad, and anything below 0.5 also yielded poor performance for all codes. Interestingly, performance variance seemed to correlate with population size for values above 0.5, with large populations experiencing more variance. After Wednesday’s class, testing 1.0 further with several population sizes (200, 256, 400) yielded good enough performance to settle on it.

**Population Size**

For most of my testing, this did not make a significant difference. However, once I sorted out a performance issue (increasing my speed by 1000%) and increased the number of evolutions, this DID make a significant difference, with a population larger than 200 but smaller than 400 being the sweet spot. 256 worked good enough, so I stuck with it.

**Evolutions (Fitness evaluations)**

For my program, this was the number of times a child was created, and their fitness evaluated. This value did not seem to make a significant difference for above 5,000 – 20,000, usually improving performance ~5% per 5,000. After fixing a performance issue, I increased it to 100,000, significantly improving performance as-expected, which I stuck with. However, it also had a side-effect of making population tweaks much more noticeable, likely due to mitigating the effect of elites dominating my populations (due to Tournament selection) with a high number of mutations.

**Punishment**

If the lookup for the key in the ciphertext table gave a non-zero result, and the English table frequency was zero, I would punish. This was done by multiplying the fitness by a value, and assigning it back to the fitness. That value would be either large or small depending on fitness function, and was intended to only slightly modify the fitness. However, the performance that resulted was terrible. On average, the fitness values were 40-60% worse than without the punishment for the b115 code, and ~25-30% worse for the test.txt code.

**Issues I ran into**

Until Monday, everything I submitted gave essentially random keys on the test machine. That was the result of my assumption that “freq.txt” was on the test machine, and consequently not including it in my tarball.

**Testing Methods**

Ran against every code one or ten times, while timing using “time”. The former was for rapid iterative development. The latter was useful for averaging the performance for each code, providing a more accurate picture of how that version of the code performed.

Compiled using GCC 4.4 on Wormulon and 4.8 on Bash for Windows during testing, with Bash for Windows on three independent systems. Performance did not vary between platforms, only speed of execution.