Applications of Genetic Algorithms to Procedural Image Generation

Akhila Ananthram & Griffin Brodman
asa225 & gb282

December 16th

1 Introduction

Genetic Algorithms are a very interesting and unexplored area of Computer Science. We decided it would be interesting to apply genetic algorithms to different solved problems, and see if we could find improvements in performance.

The problem we decided to analyze was that of generating an image out of translucent polygons. We found a project (SOURCE) that given a source image, will modify a canvas by adding polygons, removing polygons, adding or subtracting vertexes to polygons, moving polygons, changing their color or their transparency. He claimed that he was using genetic algorithms, but on inspection, he had actually used hill stepping. He would generate a child through mutation, see if this was closer to the source image, if it was keep it, if else, throw it away. This isn't really an example of genetic algorithms as there is a guarantee that the next generation is better than the previous one. We were wondering if genetic algorithms could be applied to this problem, and even show some improvements in performance.

We theorized that looking at how genetic algorithms change the performance of this algorithm would give us a much better understanding of real world applications of genetic algorithms, and what their best use case is. From here, we could more accurately identify problems that would benefit from this approach to solving problems. We will vary our approach to genetic algorithms, and try a bunch of modifications to it, modifiers like elitism and multiple parents. We'll then compare the amount of iterations it takes, on average, for hill climbing vs genetic algorithms to get to a certain fitness threshold when compared to the source image.

WE FOUND

2 Problem Definition and Methods

2.1 Task Definition

We wanted to analyze whether hill climbing or genetic algorithms would have better performance in solving the problem of generating an image with procedurally generated polygons. We would see, in numbers of iterations, how long it would take our multiple methods to get to a certain fitness. We'd first run it under hill climbing, then under a variety of genetic algorithm modifications.

2.2 Algorithms and Methods

Our hill climbing algorithm is simple. We mutate an array of polygons, either adding or removing a polygon, adding or removing a vertex to a polygon, or changing the color or transparency of a polygon. If this new array has a better fitness than the old away we keep it, else we throw it away.

Our basic genetic algorithm is simple. First we created a population of random instances, where an instance is a list of polygons. Then we evolve the population by crossbreeding and applying mutations to the children. We do this until it converges.

For crossbreeding, we select two parents based on their fitness. We calculated the fitness of the entire population. Based on the fitness scores, we assigned a probability to each individual of how likely it was to be a parent. We then randomly selected two parents to crossbreed.

When selecting genes from a parent, we used reservoir sampling. First we determined the number of genes we would want from a parent. We simply picked the average length of the parents. Our implementation of reservoir sampling follows the wiki page.

The list of possible mutations is the same for both the hill stepping and genetic algorithm.

We added variations to this basic method to tune the genetic algorithm. This includes niche penalty, elitism, random individuals, and varying our parameters.

Individuals whose difference in fitness score was within a certain threshold are part of a niche. For these individuals, we applied a penalty to avoid having our algorithm converge to this local minimum.

Elitism is the idea that certain parents can live on to the next generation if they are the most fit. The idea behind this is that the most fit should be able to survive.

To avoid approaching a local minimum, we added a random person to the population every few generations.

Lastly, we also made our program capable of varying the parameters to better tune the algorithm.

We developed two different fitness functions, and sampled using both. Our basic fitness function was one that went pixel by pixel, and calculated the euclidean distance between the two images. Our second one relied on opencv feature matching. We calculated the descriptors for the source image and the current image. Then we used a brute force matcher to find the matches and then calculated the sum of the distances between the matches.

3 Experimental Evaluation

3.1 Methodology

3.2 Results

Because of our language choice, our program is extremely slow. Our first thought was to parallelize the program as much as possible. Our first attempt was using Python's ThreadPool. Unfortunately, Python uses a Global Interpreter Lock. Thus, adding multi-threading did not improve the performance. In fact, it actually slowed us down. We then looked into multi-processing. Because of hardare limitations, we decided to use 3 sub-processes. We were able to parallelize the fitness function and the creation of children. However, we reached an interesting roadblock on Windows with multi-processing when we were attempting to access global variables from a sub-process. On Unix based systems, Python's multiprocessing uses fork(), giving every child process a copy of its

parent's address space, including global variables. However, this is not the case for Windows. Any variable from a parent process that is accessed by a child must be explicitely passed along.

Another way we attempted to speed up our program was by adding the ability to apply the fitness function to just a sample of the image. As the polygons are never going to match the picture's pixels exactly, we can look at just a sample.

4 Related Work

5 Future Work

There were a few things that developed as we worked on the project. For one, working in Python severly limited our speed. We did get the results we needed, but we were constrained to smaller images. It would have been nice to support much larger images, which would have opened up different fitness functions to us, as it would have improved feature matching. The project we had used as inspiration had been written in c#, and though we didn't think the difference in languages would have such a significant effect, it seems to be huge.

6 Conclusion

I'd l