Universal Turing Machine Search

EECS 600: Ruby on Rails

Steven Dee
Justin Gray
Joshua Lee
Neil Sandberg

{sxd98,jxg274,jcl26,nls13}@cwru.edu

March 2, 2009

1 Abstract

We propose to investigate Darwin's Theory of Natural Selection, in abstract, by analyzing the results of a Genetic Algorithm optimization on a population of Turing Machines. Using the Turing machines as analogs for simple organisms, the goal of the research is to perform an optimization where the result is a population which contains some Universal Turing Machines. We take Universal Turing Machines to represent more complex organisms on an evolutionary path. We will include a web based visualization environment, written in Ruby on Rails, to help visualize the results of the optimizations.

2 Introduction

In 1861 Charles Darwin published his theory of natural selection as a means by which living creatures evolved to best meet the challenges presented to them in their environment. (Darwin 1861) According to Darwin, species undergo changes because certain members of a population posses traits which allow them to reproduce more successfully than others. Those organisms offspring retain the advantageous traits and hence continue to reproduce more successfully themselves. It is an interesting thought experiment to consider Darwin's theory in the extreme, where life exists as very simple organisms and progressively evolves into more complex ones eventually resulting in mammals and then primates and then humans. When examined in the extreme, Darwin's theory seems almost impossible.

We have developed a method to examine this extreme view of Darwin's theory, in an abstract manner, by applying a global optimization process to Turing Machines (TM). TMs, initially conceived as a thought experiment by Alan Turing in 1937, "are simple abstract computational devices intended to help investigate the extent and limitations of what can be computed". (Barker-Plummer 2004) These simple machines can be constructed so that they are capable of computations that range in complexity from essentially useless to practically unlimited. In their most complex form, TMs are given a special designation: Universal Turing Machines (UTM). A UTM can reproduce the output of any other TM (even other UTM's), given the same input. We employ special capability of a UTM as an analogue to the complexity that exists in modern living organisms. By using TMs as the subjects of a global optimization, and carefully constructing the function to evaluate the fitness of each TM, an evaluation of the possibility that "complex" TMs can evolve from effectively random "simple" TMs is performed.

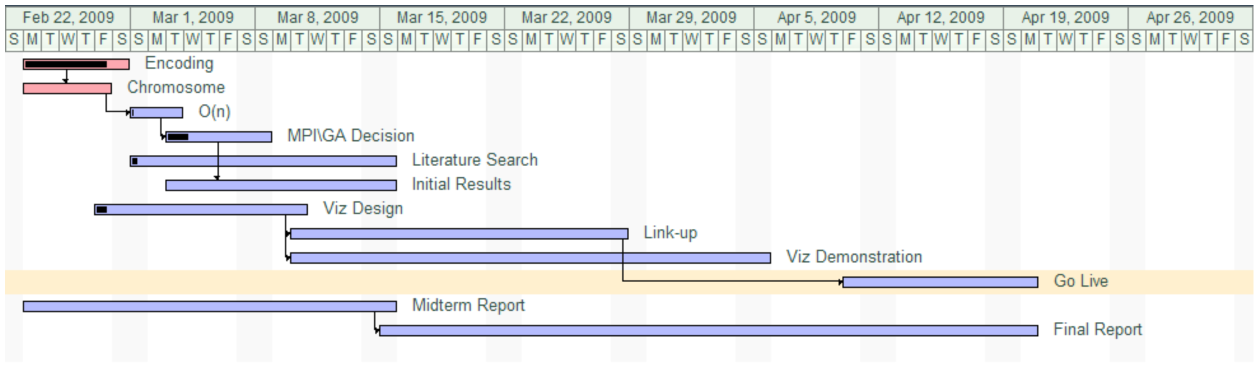

Figure 1: Project timeline of major milestones

3 Project Timeline

Figure 1 shows a detailed timeline for the completion of the project. The major milestones for all aspects of the project present, including necessary work for the creation of an optimization scheme to search for UTMs as well as the creation of a web based visualization tool to facilitate the examination of the project results. The key to the task names used in Figure 1 is presented below:

- **Encoding**: Definition of the encoding method for Turing Machines completed
- **Chromosome**: Implementation of the chromosome class for the AI4R Genetic Algorithm package completed.
- **O(n)**: Computational cost for fitness evaluation of a Turing Machine is determined.
- **MPI/GA Decision**: Decision made regarding necessity of use for MPI along with AI4R Genetic Algorithm to reduce computation time.
- **Literature Search**: Literature search to generate a list of known UTMs
- **Initial Results**: Initial Turing Machine Optimization run completed. Data analyzed to ensure the viability of optimization scheme
- **Viz Design**: Visualization web-app design complete
- **Link-up**: Link between genetic algorithm generation data and visualization database completed.
- **Viz Demonstration**: Ruby on Rails visualization application implementation completed.
- **Go Live**: Ruby on Rails visualization application goes live with link to genetic algorithm running perpetually.
- **Midterm Report**: Midterm Report Delivery
- **Final Report**: Final Report Delivery
- **Presentation**: Final Presentation

4 Organizational Structure

4.1 Project division

The project was devided into two main parts: The genetic algorithm optimization and the results visualization. The work load was split evenly between team members, but each section was managed by lead. Josh Lee led on the genetic algorithm implementation and Neil Sandberg led the web visualization implementation. The genetic algorithm implementation required the larger development efford, and Josh Lee and Steven Dee focused their efforts on this development, with algorithm design support provided by Justin Gray. Neil developed the initial web visualation tool and Justin Gray contributed to the final data representation design and implementation.

Each of the two parts was developed in parallel, using a database as the data bridge between them. By using a simple database design that was fixed early on, the task of integration of the two parts was greatly simplified.

All work was performed collaboratively using a github source repository. The project is stored in an open source repository at http://github.com/mrdomino/utm/. The repository was used to store all files regarding the implementation of the genetic algorithm and the web visualization tool.

5 Genetic Algorithm Optimization

Genetic algorithm optimization is designed to mimic the evolutionary process described by Darwin's Theory. In this genetic algorithm, potential solutions are encoded as strings, the fittest of which survive to produce more solutions in later generations. The key to a successful optimization using the Genetic Algorithm is the selection of the fitness function with which to evaluate the subject of the optimization. However, determining the fitness test was not the only step required to execute the optimization. A method of encoding a TM into a "genome" was required along with some method of reproduction between parent TMs.

5.1 Turing Machine Encoding

The encoding translates a binary string into an actual TM. Binary string encoding was chosen for its simplicity and well documented use in genetic algorithms. To define the encoding scheme, first it was necessary to define a set of constants for the TM population:

- NUM_STATES: Total number of states represented in each state table
- GENE_LENGTH: The number of bits being used to represent a single entry in the state table

The size of GENE_LENGTH is required to be at least large enough to represent NUM_STATES while reserving two bits at the end for the **write** bit and **movement** bit(left = 0, right = 1). However, it is allowed for there to be many more bits than necessary to accommodate future growth in NUM_STATES without changing the alignment in all previous TMs. To handle this issue, the modulus operator was employed to ensure that all possible state values fall within a valid range.

current state	**read bit**	**next state**	**write bit**	**movement**
1	0	11	1	1
1	1	10	1	0
2	0	10	0	1
2	1	10	0	1
3	0	01	0	1
3	1	01	1	1

Table 1: State transition table for notional 3-state TM where the next state has been represented in an appropriate length binary string

Take the example encoding: '111110101001100101000110', with NUM_STATES=3 and GENE_LENGTH=4. This is broken up into segments which are GENE_LENGTH bits long: 1111,1010,1001,1001,0100,0110. Each gene segment then represents a single entry in a state transition table. The resulting state transition table can be seen in Table 1.

It should be noted that all TM using this encoding assume that the 0 state is the halt state. It is not required that any TM utilize the halt state at all. The example TM above did not use the halt state.

5.2 Fitness Evaluation

The function of the fitness evaluation in a Genetic Algorithm is to provide the means of scoring one population candidate against another. The particular formulation that is chosen for a fitness evaluation will effect the entire outcome of any optimization. For this research, the purpose is to investigate the likelihood that an evolutionary process can produce very ”complex” organisms from ”simple” ones. Taken TMs as analogs for organisms, the best possible result we can hope for is a population of UTM's. So the fitness function employed must grade TMs that are most like a UTM higher than ones that are less like a UTM.

There is not simple test for that can be applied to a TM to determine if it is in fact a UTM. If there were, you would simply apply that test to every member of a population and grade them accordingly. Lacking such a test, it is necessary to establish certain measurable qualities of a UTM that could be tested for on a candidate. The fundamental problem with this approach is that when a specific candidate has been shown to share a lot of the qualities that you have associated with a UTM, there is no single logical path that can extend that information to show that the candidate is actually a UTM (that's why there is no test!).

Instead of testing directly for qualities of candidate TMs that are similar to those of a UTM, a more simple fitness evaluation was used, which follows more directly with the stated hypothesis. TMs are scored based on the complexity of their output. This complexity is measured by applying a compression algorithm to the output string of the turing machine, and then using the resulting size of the compressed data as the fitness of the TM. The DEFLATE compression algorithm was selected because it is a lossless compression algorithm and is part of the ruby standard library.

Since the output of a TM is dependent on the input given to it, the outcome of this fitness evaluation is highly dependent on the input given to each candidate TM as well. Three interesting input types were evaluated:

1. Fixed input string for all generations
2. Randomly generated input string for each generation
3. Input string with some mutation between generations

The first option tests the outcome of an optimization where each generation is being given effectively the same exact test as all the others. TMs which are good at making complex output for this specific input will thrive. The second option basically generated a random new test for each generation, and only TMs that can produce complex output for arbitrary input will thrive. These TMs could be considered very adaptable. The third option presents a hybrid approach to the previous two; here each successive generation is operating on an input string which is similar to the last (how similar depends on the mutation rate) but not identical. Over a large number of generations however the input string is effectively appears random. For this research, the when using the third input string option, a mutation rate of 50% was used.

5.3 Genome Mutation

The mutation operator in a GA serves to introduce new genes to the population. Without that injection of new genes the only possible results would have to consist of some combination of the genes that were created at random as part of the initial population. The mutation algorithm applied here used a bit swapping method where a certain percentage of bits in the encoding are swapped with their neighboring bits. Using this method both the probability that a genome will be mutated and the frequency of bit swaps along the encoding can be varied to adjust the performance of the genetic algorithm optimization. For this work, the probability of mutation was set to XX% and the frequency of bit swaps was XX%.

Keeping these parameters set to higher values ensures a constant and high rate of introduction of new genes into the population. This much churning of the gene pool also helps to prevent too much homogeneity from developing in the gene pool.

5.4 Parent Selection and Breeding

Between each generation a GA selects a subset of the current population for breeding to produce the next generation of candidates. This research utilizes the competition selection method. The competition selection method was chosen for the flexibility it provides in tuning GA to preserve population diversity. The competition method selects parents for breeding by creating a competition with COMPETITION_SIZE contestants. The contestants are selected at random from the population and the one with the highest fitness wins and becomes a parent. By making COMPETITION_SIZE smaller the probability that a weaker contestant will win increases, thus preserving diversity by allowing weaker genomes to survive to the next generation. For this research COMPETITION_SIZE was set to 2, ensuring maximum diversity.

6 Web Visualization Tool

The results visualization system provides a graphical summary of the progression of an optimziation. It was written using Ruby On Rails and takes advantage of two graphical plotting tools: gcharts and graphviz. Gcharts is used to generate the graphcial plots of relevant optimization metrics from different optimizations. The graphviz package is used to render images of the

6.1 Optimization Metrics

The progress of the optimziation is charachertized on a per generation bassis. Relevant metrics include:

- Average fitness
- Maximum fitness

For a good optimization one would expect both the maximum fitness and the average fitness to increase from one generation to the next. However, the data represnetation provides valuable information regarding the relative values of the maximum and the average fitness. If the average fitness begins to approach a value close to the maximum fitness, this is an indication that the population is becoming too homogeneous and it is unreasonable to expect further optimization to result in large fitness gains. Since the optimization was

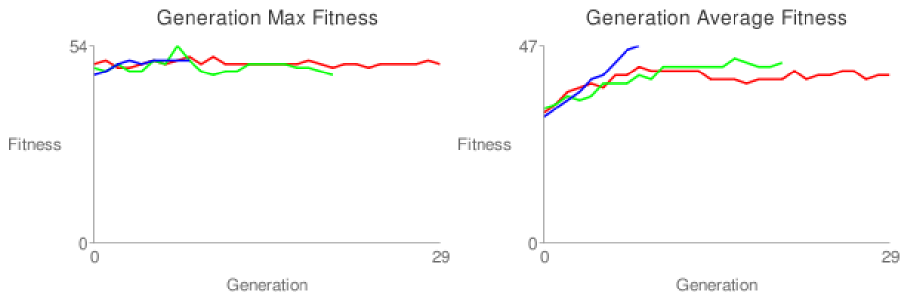

Figure 2: Example metric graphs for maximum and average fitness.

sub-devided into three separate optimization runs to test the three different fitness testing methods, the visualization tool also provides graphical representation of each of the three sub-data sets simultaneously. This representation allows a quick comparison between the effectiveness of the three different fitness evaluation methods.

6.2 TM Representation

The vizualization of individual TMs provides a valuable tool for their inspection. There are litterally thousands of TMs created over the course of an optimziation, and graphviz package allows or the creation of a state transition diagram for quick visual inspection of particular TMs. Figure 3 is the state transition diagram for the example TM in Table 1. This visual representation is much easier to interperate than the state transition table.

7 Optimization Results

8 Conclusion

Shit don't work!

References

Barker-Plummer, D. (2004), 'Turing machines'. Stanford Encyclopedia of Philosophy.

Darwin, C. R. (1861), *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life*, 3 edn.

Fierense, S. & Ferret, J. (2009), *Genetics Algorithms in Ruby :: ai4r*.

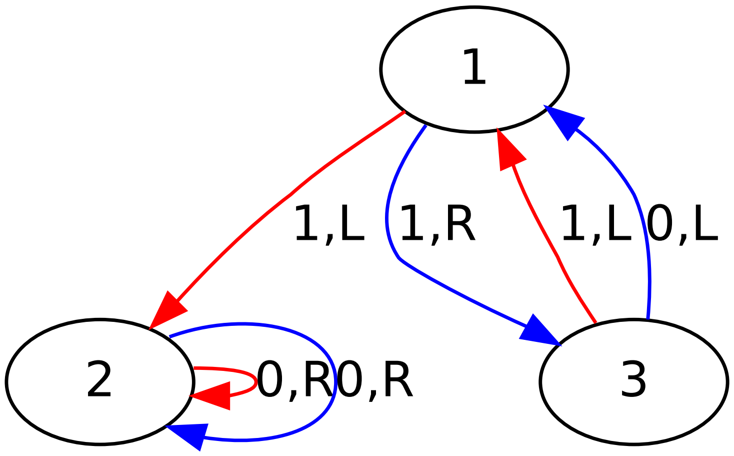

Figure 3: State transition diagram for notional 3-state TM. Edge color indicates the bit being read from the tape (red=0,blue=1). Edge label indicates the write bit and movement direction for a given transition.