Genetic Algorithm to find curve shapes

Menaka Lashitha Bandara
Monash University, Clayton,
Victoria, Australia, 3168

Friday October 1, 2004

Contents

1 Introduction

The Brachistochrone problem dates back to Galileo Galilei where it was mentioned in his *Dialogues*. The problem is concerned with the construction of a ramp which will allow a ball to move from coordinate $(0, 0)$ to some coordinate (x, y) with $x > 0, y \leq 0$ with a minimal time.

In this document, we discuss a *Genetic Algorithm* to evolve an optimal shape of the ramp. We make this algorithm general in the sense that it does not make assumptions about the shape of the curve and thus, can be used to evolve curves for a whole class of problems. We approximate the the curve between two points discretely by the use of a staircase function. The objective function for the Brachistochrone problem is then given by elementary Galilean mechanics.

We discuss the approach taken to to solve this problem, examine the design of our GA, and analyse the way in which changing its parameters affect the effectiveness of the algorithm.

2 The Genetic Algorithm

In this section, we describe the genetic algorithm implemented in order to solve the Brachistochrone problem. We note that this algorithm is not particular to this problem - it can solve any which requires for a curve to be evolved given an appropriate fitness function which determines that system. This GA is a variation on schemas found in the literature in that it has been designed in an attempt to converge rapidly towards a good solution for these particular class of problems.

2.1 Chromosome

The algorithm is abstracted away from the problem by simply viewing the chromosome as a vector (internally an array) of points, which represent values on the y axis, for points separated by Δx.

The following data structure is defined in `include/genotype.h`:

```
typedef struct _GenoType        GenoType;

struct _GenoType {
        /* Our heights */
        double          *y;
        unsigned int    size;
};
```

The implementation of the constructor and destructor can be found in `ga/genotype.c`.

Here, the variable y holds an array of length $size$. That is, our interval for which we are generating points on the curve are broken into $size - 2$, since we have chosen to fix the end points.

An `Individual` is represented then by the type defined in `include/individual.h`:

```
typedef struct _Individual      Individual;
struct _Individual {
        GenoType        *gene;
        double          fitness;
};
```

Here, an `Individual` consists of a chromosome $gene$, and a $fitness$. All members internally are stored as `Individuals` in the `Pool`. We use $fitness$ in order to rank individuals.

2.2 Selection Operator

We implement roulette wheel selection as our selection operator. This allows us to give the fitter individuals more preference in breeding. We associate a relative fitness (proportion) with each individual and select an individual with some uniform probability. Since a fitter individual will occupy a greater proportion of the interval, they will be selected with a higher probability.

The implementation can be found in `ga/ga.c` and `ga/pool.c`.

```
function roulette_wheel_selection (Pool generation):

        Individual      mother, father
```

```
        sort the individuals in terms of descending fitness

        for i = 0 to generation.population:
                total_fitness = generation.person [i].fitness + total_fitness
                i++
        endfor

        for i = 0 to generation.population:
                generation.relative_fitness [i] = generation.fitness [i]/total_fitness
                i++
        endfor

        probability = random (0, 1)

        mother = individual with relative_fitness <= probability, with
                their previous neighbour with relative_fitness > probability

        probability = random (0, 1)
        father = individual with relative_fitness <= probability, with
                their previous neighbour with relative_fitness > probability
                different to mother

Output: mother, father
```

2.3 Crossover Operators

The `breed` abstract data type provides an interface for the crossover and mutation operators. Definition of the interface is given in `include/breed.h`, and the implementation in `ga/breed.c`.

The three operators discussed below are parametrised by $crossovers$, as given in the `breed` structure.

2.3.1 Splicing Operator

It was anticipated that under certain circumstances, splicing was the choice of operator. This operator is in fact most useful at earlier stages of the algorithm when we want our operator to be crude by making large scale changes when crossing two chromosomes to produce an offspring. That is, it will introduce drastic changes to the shape of the curve.

We give the pseudo-code for the splicing operator.

```
breed_splice (GenoType father_mother [2]):

        List            L = [ crossover number of random points,
                                sorted ascending ]
        GenoType        child

        index = round ( random (0, 1) )

        for i = 0 to crossovers - 1:
                for j = L [i] to L [i + 1] :
                        child.y [j] = father_mother [index]
                        j++
                endfor

                i++
                index = 1 - index
        endfor
```

```
Output: child
```

Here, we compute the cross over points randomly and then copy intervals defined by the list of these indices while simultaneously alternating between the mother and father chromosomes.

2.3.2 Interleave Operator

The interleaving operator is a much more delicate operator when mixing two chromosomes. This makes much more subtle changes to the offspring chromosome structure than the splicing operator. It is therefore most useful in the middle stages of the GA. It works by retaining the shape of most of first parent chromosome, and then copying randomly selected elements from the second parent chromosome. The code below gives a more detailed description.

```
breed_interleave (GenoType father_mother [2]):

        List            L = [ crossover number of random points,
                              sorted ascending ]
        GenoType        child

        index = round ( random (0, 1) )

        child = father_mother [index]

        index = 1 - index

        for i to crossovers:
                child.y [i] = father_mother [index]
                i++
        endfor

Output: child
```

2.3.3 Means Operator

In contrast with the last two operators, this is the most delicate and suited for the final phase of the GA. Here, we copy a parent gene right into the child, and for some randomly selected points, we take the average of the two parents. Thus, it essentially serves the purpose of smoothing. This is particularly useful for curves which only have a countable number of non-differentiable points.

```
function breed_means (GenoType father_mother [2]):

        List            L = [ crossover number of random points,
                              sorted ascending ]
        GenoType        child

        index = round ( random (0, 1) )

        child = father_mother [index]

        index = 1 - index

        for i to crossovers:
                child.y [i] = 0.5 * (father_mother [index] + father_mother [1 - index])
                i++
        endfor

Output: child
```

2.4 Mutation Operators

The basis of our mutation operator is to either add or subtract some amount proportionate to the value at some index of our array. We perform this over $mutations$ number of random indices. It was discovered experimentally that such mutations were most effective when an interval is divided into approximately 20 segments. Thus, for an arbitrary number of intervals, we compute a radius and perform mutation for every index in that radius around a random point to the effect of operating on 20 segments.

We give detailed pseudo-code to our mutation operator. The implementation can be found in ga/breed.c.

```
function mutation (GenoType g,
        Integer number_of_intervals,
        Double height):

        diameter = (number_of_intervals + 1)/20

        for i = 0 to mutations:
                index = round( random (0, 1) )

                perbute_amount = random (0, 1)

                j_max = index + floor (diameter / 2.0) + 1
                j = index - floor (diameter / 2.0)

                if j < 0 then
                        j = 0
                endif

                if j_max > g.size then
                        j = g.size
                endif

                if ( round ( random (0, 1) ) ) then
                        for j to j_max:
                                g.y [j] += (height - g.y [j]) * perbute_amount
                                j++
                        endfor
                else
                        for j to j_max:
                                g.y [j] -= (height - g.y [j]) * perbute_amount
                                j++
                        endfor
                endif

                i++
        endfor
Output: g
```

2.5 Fast Convergence Operators

We discuss some operations that perform operations on offspring which in general leads to faster convergence. It was found that it is often the case that the shape of a curve is found quickly, but it takes much time for the curve to be scaled properly. These local operators are scaling operators, and were motivated by the fact that convergence could be assisted by scaling the solution.

These scaling operators are internal to the Genetic Algorithm and can be found in ga/ga.c.

2.5.1 Constant Scaling Operator

This is the simplest of the two operators. Here, we simply multiply each value by a constant factor. More precisely:

```
function locally_improve_constant (GenoType gene,
                        Double factor):

        for i = 0 to gene.size:
                gene.y [i] = gene.y [i] * factor
                i++
        endfor

Output: gene
```

2.5.2 Random Scaling Operator

In this operator, we scale the solution by a random amount, multiplied by some supplied factor.

```
function locally_improve (GenoType gene,
                        Double outfactor)

        factor = (random (0, 1) + 1) * outfactor

        for i = 0 to gene.size:
                gene.y [i] = gene.y [i] * factor
                i++
        endfor
Output gene
```

2.6 Replacement Scheme

The replacement scheme used is somewhat unique and cannot be found in any literature. It can be seen as a hybrid between a random and elitist strategy.

2.6.1 Creating a new population

We use the function `create_population` to create a whole new population from the old. In this function, we create 3 children from two parents via the splicing, interleaving, and means operators. We create 4 copies of the set of these children and apply the scaling operators described above.

Generally, the scaling operators become less important with time. In fact, it is obvious that if we take a near optimal solution and scale it, it will become a much worse solution. So, for that reason, we only allow the best of these 15 children to live in the new population.

This implementation can be found in ga/ga.c. Note that this operation is parametrised by $mute\_prob$.

```
function create_population (Pool old_generation):

        Pool            next_generation
        GenoType        children [15]

        Individual      mother, father, offspring
```

```
        for i = 0 to old_population.population:
                mother, father = roulette_wheel_selection (old_generation)

                create children [0, 1, 2] by splicing, interleaving, means

                duplicate children [0, 1, 2] for children [3...14]

                children [3, 4, 5] = locally_improve_constant (children [3, 4, 5], 1.5)
                children [6, 7, 8] = locally_improve_constant (children [6, 7, 8], 2.0)
                children [9, 10, 11] = locally_improve (children [9, 10, 11], 1.0)
                children [12, 13, 14] = locally_improve (children [12, 13, 14], 2.0)

                for i = 0 to 14:
                        if random (0, 1) < mute_prob
                                mutate (children [i])
                        i++
                endfor

                calculate the fitness of children [0..14]

                add the fittest child from children to next_generation

                kill off the rest of children
        endfor

Output: next_generation
```

2.6.2 Merging of the population

This operation is parametrised by a proportion $keep\_prop \leq 0.5$. The evolved population has $keep\_prop$ proportion of the old generation and of the new generation (from the previous operator). Note that this does not guarantee that we are going to get the best $keep\_prop$ proportion if the new generation and old were mixed together prior. The latter method was not implemented because the current method introduces a variability into the *better* solutions. Both methods allow us to keep the best solution of the two populations in the evolved population. The implementation can be found in ga/`pool.c`.

We fill the remainder of the evolved generation by randomly copying individuals from the old and new generations.

The following pseudo-code better describes this.

```
function merge (Pool old_generation, Pool new_generation):

        Pool            evolved_generation

        sort_descending (old_generation)
        sort_descending (new_generation)

        for i = 0 to keep_prop * old_generation.population:
                evolved_generation.person [i] =  old_generation.person [i]
                i++
        endfor

        for j = 0 to keep_prop * new_generation.population:
                evolved_generation.person [i] = new_generation.person [j]
                j++
                i++
        endfor
```

```
        for i to old_generation.size:
                randomly swap individuals from new_generation and old_generation
                        into evolved_generation
                i++
        endfor

Output: evolved_generation
```

2.7 Initial Population Techniques

The implementation of methods described here can be found in ga/ga.c. We omit giving pseudo code for these techniques as they are trivial.

2.7.1 Perbutations of a line

Here, we create solutions by firstly constructing a line, and then randomly perbuting the line. The coordinates of the line are $(0, y)$ and $(x, 0)$, where x, y are the two fundamental parameters in the GA.

2.7.2 Ordered random heights

In this method, we create our chromosome randomly and then order the values of the array descending.

2.7.3 Complete random heights

Here, we simply create the chromosome by simply inserting random values into the array.

2.8 Discussion of the Algorithm

2.8.1 Dumb evolution

Due to the trivial nature of this evolution technique, we omit describing it via pseudo code. Here, we simply create an initial population, from the three operators given above. Each individual has a chromosome of length $intervals$ as requested. This makes the search space of problems extremely large.

2.8.2 Smart evolution

This is a clever technique which is a significant improvement over dumb evolution. Here, we create an initial population with each individual having a chromosome of length 10. That is we solve the first $\frac{1}{4}$ of the total iterations by a discrete approximation of only 10 divisions of the interval. This allows us to look at the problem with a lower *resolution*. We increase this amount so that when algorithm terminates, we have exactly the number of intervals requested by the parameter $intervals$.

We also note that we introduce mutations to $mute\_prop$ proportion of the population with a probability of $mutation\_prob$. This allows us to introduce variations at the time of a child being created, and allow us to model external mutations.

We give the pseudo code.

```
function run_simulation_smart (Pool generation, Integer intervals,
        Double sub_interval, Integer iterations)

        current_intervals = 10
        current_sub_interval = (intervals * sub_interval)/10

        current_iterations = (iterations - (iterations / 4))/4
        increment_intervals = intervals/4

        create_initial_population (generation)
        for i = 0 to iterations / 4:
                new_generation = create_population (generation)
                generation = merge (new_population, generation)

                mutate_population (generation)

                i++
        endfor

        while i < iterations:
                if current_intervals + increment_intervals > intervals then
                        current_intervals = intervals
                else
                        current_intervals = current_intervals + increment_intervals
                endif

                current_sub_interval = (intervals * sub_interval) / current_intervals

                copy across solution of previous interval size to the
                        new interval size in generation

                for i to current_iterations:
                        new_generation = create_population (generation)
                        generation = merge (new_population, generation)

                        mutate_population (generation)

                        i++
                endfor
        endwhile

Output: generation
```

We note that the way we compute $current\_intervals$ and $increment\_intervals$, we will always be guaranteed that the algorithm will lastly run with a chromosome size of $intervals$.

3 Implementation of the Genetic Algorithm

3.1 Solving your own problem

In this section, we discuss the `ui_begin` function and the parameters to tune the GA The prototype of this function can be found in `include/ui.h`:

```
extern int ui_begin (char *progname, char *version,int argc, char **argv,
                double (*fitness) (double subint, double h, GenoType *g),
                double (*endpoint_function) (GA *ga, double xval));
```

This function is by far the easiest way to solve a problem with this GA implementation. The fitness function is a pointer to function which is evaluated at a point $subint$. An endpoint function is necessary because the end points of an interval is not stored in a `GenoType`.

A problem is always seen in the bounding rectangle given by the lower left coordinates $(0, -y)$ and upper right coordinates (x, y). Any problem outside of this domain must be *translated* to within this domain. (Note: this rule doesn't not strictly hold, i.e., the curve may evolve to be out of this rectangle.)

3.2 Command Line parameters

From the prototype of the `ui_begin` function, we can see that it takes $argc$ and $argv$ as parameters. These are respectively the argument count and argument vector. Thus, `ui_begin` provides a complete interface to tune all parameters of the GA. We describe the parameters below.

GA parameters

- `-n, --nintervals N`
  Takes the argument $N \in \mathbb{N}$, the number of intervals which we divide our function interval into.
- `-x, --xcoord X`
  Takes the argument $X \in \Re$ of the right hand bounding box x coordinate. Here, it makes no difference to give X or $-X$.
- `-y, --ycoord Y`
  Takes the argument $Y \in \Re$ of the right hand bounding box y coordinate. It makes no difference in using Y or $-Y$.
- `-i, --iterations I`
  Takes the argument $I \in \mathbb{N}$, the number of iterations for which the algorithm should run.
- `-p, --population P`
  Take the argument $P \in \mathbb{N}$, the number of individuals to use in the population.
- `-k, --keep K`
  Takes the real argument $K \in [0, 0.5]$, the proportion of the fittest individuals to keep from the old and new generations into the evolved generation.
- `-c, --crossovers C`
  Takes argument $C \in \mathbb{N}$, the number of crossovers to perform. If more crossovers are specified than intervals, then the number of crossovers are made small.
- `-m, --mutations M`
  Takes argument $M \in \mathbb{N}$, the number of positions to mutate of the chromosome array.

- `-u, --proportion U`
  Takes argument $U \in [0, 1]$, the proportion of individuals to mutate in the population when external mutations take place.

- `-b, --prob B`
  Takes argument $B \in [0, 1]$, the probability associated with introducing a a mutation (to an individual and the population).

- `-l, --linear L`
  Takes argument $L \in [0, 1]$ the proportion of linear slope perbutations of a slope from $(0, y)$ to $(x, 0)$ to include in initial population.

- `-r, --random R`
  Takes argument $L \in [0, 1]$, the proportion of of the initial population which are created randomly but with the heights ordered descending.

- `-s, --smart S`
  Takes argument $S \in \{true, false\}$, whether the algorithm should use smart or dumb evolution.

Other Parameters

- `-e, --seed E`
  Takes argument $E \in \mathbb{Z}$ to use as a seed for the random number generator.

- `-g, --graph G`
  Takes argument $G \in \mathbb{N}$, and produces a GNUPlot graph at each G iterations.

- `-d, --delete D`
  Takes argument $D \in \{true, false\}$, to specify whether to delete the old graph before re-plotting the new graph. It has no effect when -g is not used.

- `-o, --output Filename`
  Takes argument $Filename$ which to write the final graph. If this is omitted, then the graph is written to standard output.

- `-v, --version`
  Displays the version information about the program.

- `-h, --help`
  Displays the help screen which gives the information outlined here briefly.

Without argument, a program `function` would work identically as if it was called in the following way:

```
./function -n 100 -x 2.0 -y 2.0 -i 250 -p 200 -k 0.3 -c 33 -m 1 -u 0.3 -b -0.3 \
       -l 0.0 -r 0.0 -s true -g 1 -d false
```

4 Analysis of the Genetic Algorithm

We use an objective function which defines the Brachistochrone problem in the analysis of this Genetic Algorithm. It's implementation can be found in `prog/brach.c`.

We find the optimal parameters of solving the system are given by a population of 70 to 300 individuals, running for > 200 iterations (250 is sufficient enough for convergence), keeping 40% of best solutions with 3 or 4 (or approximately $\frac{1}{3}$ of the number of intervals) crossover points and 1 mutation, with a mutation probability of 0.3 whilst mutating 30% of the population. We have 0% of both linear based and random ordered individuals in the initial population.

We find that decreasing our population size to below 70 has the effect of faster convergence to a solution that is worse. That is, even though we let the algorithm run for longer, we find that the system does not necessarily converge to a better value (or rather that its very slow). As we increase the number of iterations to larger and larger numbers above 300, we find that the solutions become somewhat better, but not significantly. Furthermore, the algorithm becomes extremely slow.

The algorithm has been designed so that it only uses the smart method of convergence when we have 70 or more iterations. We find that increasing the number of iterations does produce better results. However, it does not make sense to use more than 500 iterations. On average, the system will converge to values fairly close to the optimum in about 200 to 250 iterations.

When decreasing the proportion of best solutions we keep, we find that the algorithm performs worse. It seems that 40% is a good value. This translates keeping 80% of good values, and 20% chosen at random. When we set this value to 0%, then we have 100% random selection of individuals to keep. This is a bad strategy. As this value approaches 50%, we find that the system converges slowly. This would be due to the fact that variation is lost.

Lower number of crossover points can be specified without considering the number of intervals which we divide our problem into. This is because the splicing operator will be dominant (at least initially). We can achieve better results by using crossover rates between $\frac{1}{3}$ to $\frac{2}{3}$ the number of intervals which we divide our problem into. This way, our splicing operator will become somewhat more like our interleaving operator. Also, we would find that no operation will necessarily dominate the other. Larger crossover points tend to work better. It should also be noted that the GA will reduce the number of crossovers if the value supplied it too high. So, by using larger cross over points, the GA will initially run approximately as if the GA was called with a lower number of crossover points.

We find that increasing the number of mutations makes the search more random. It seems that with this GA schema, between 1 to 3 mutations seems to work best. This is most likely due to the fact that 1 to 3 mutations work well on a interval size of 20, and because our mutation operator mimics this behaviour with larger intervals these values tend to work well.

In terms of mutation probabilities, increasing the mutation probability beyond 0.3 seems to introduce too much randomness. Furthermore, mutating more than 30% to 40% of the population seems to create too much randomness as well. In fact, solutions sometimes become worse from one generation to the next. This may seem contradictory when considering our elitist strategy in replacement, but it is important to bear the fact that the mutation of the population is external, and we can make a good solution (or best solution) worse.

Lastly we discuss the parameters for including linear based initial values and randomly ordered values. These were included since it seemed reasonable that including some solutions that are feasible for some problems may increase convergence to a better value. However it was found that this did not make a huge difference in convergence. In some cases, it made it more difficult for the GA to find the better solution because the best solution at each stage would be too close to the best solution of the last stage. Completely random points may be a good thing for the reason that solutions are spread across the search geometry better. If there is a good solution to begin with, then all solutions may converge towards that quickly killing off variation. It was found that in general, it was better to have 0% of the randomly ordered and linear sloped based solutions in the population.

There are some aspects of this GA which could be improved. In particular, much better convergence could be achieved if the smart evolution procedure was improved. For instance, it would be better to increase the number of intervals when the solutions have converged sufficiently. Also, placing additional scaling operators may improve the GA. It was observed that the lack of these scaling operators lead to much slower convergence. One operator that would be worthwhile is an operator that can scale across some axis of an angle $0 < \theta < \frac{\pi}{2}$ to the x axis.

It is important to emphasise that the analysis above is in terms of the Brachistochrone problem, and thus it might not hold for the GA in general. Furthermore, these parameters described may not be the best parameters that can be found. Time has not permit a thorough investigation of the GA further to better understand the emergent behaviour of this schema. It is also worth to mention that the GA was able to converge quite well to functions $c(cos(2x) + sin(x))$ and $xsin(x)$ when the objective function was defined as the difference of a genome and each of these functions evaluated at their respective x points, which is an indication the schema is general and does not make assumptions about the curve.

References

[1] **Russel S, Norvig P**. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Pearson Education Inc, 2003.

[2] **Bernd, M**. *CSE460 Lecture Notes*.

[3] *Intro to Genetic Algorithms*.
http://lancet.mit.edu/ mbwall/presentations/IntroToGAs

[4] *Genetic Algorithms Overview*.
http://geneticalgorithms.ai-depot.com/Tutorial/Overview.html