THE RED QUEEN ALGORITHM

AN APPROACH TO DYNAMIC TIME WARPING USING GENETIC PROGRAMMING

Kevin R. Lawrence

Table of Contents

I. INTRODUCTION

The name of the Algorithm is derived from a statement The Red Queen makes in Lewis Carol's "Through the Looking Glass."

"Now, here, you see, it takes all the running you can do, to keep in the same place.

To describe the method by which the selection of the fittest takes place in the genetic code.

II. DYNAMIC TIME WARPING (DTW)

In signal analysis, Dynamic Time Warping is a method of comparing two signals of different lengths.

a. Steps at a Glance
   i. Frame both signals into small enough sections to be processed.
   ii. Extract a vector of features for each signal, where each frame has the same number of features.
   iii. Take each frame of signal A, and find the difference between it and each frame of signal B. Store the results in a matrix that begins at the beginning of both signals and proceeds to the end of both signals.
   iv. The optimal solution is the path through the matrix that produces the minimal distance per frame of the path.

b. The DTW Problem Defined.
   The number of paths through a matrix of uneven dimensions is given by:

$$p_{MN} = \frac{\big((M+N)-2\big)!}{(N-1)! * (M-1)!}$$

It turns out we do not have a computer yet that can solve these kinds of problems (NP-Hard). Even though we can see clearly that there's a finite number of possible paths, the number of possibilities to consider quickly grows out of control. This describes the necessity for a solution using genetic code.

III. EUCLIDIAN-MATRIX

a. Theory and Notation:

For individual Frame, a constant number of features can be extracted. For simplicity, this example will use 26 Mel-Cepstrum Feature Coefficients (MFCC). However, the exact number and nature of coefficients per vector is arbitrary, just as long as the number of them is constant size per frame. Each frame m in signal A of size M is compared to each frame n in signal B of size N as the shortest distance between them. This is given by the Euclidian distance written in notation as:

$$d_{mn} = \sqrt{\sum_{i=0}^{n<26} (n_i - m_i)^2}$$

Which can be used to create a Matrix of distances between each frame of signal A to each Frame of signal B. The optimal solution can then be found by navigating from corner position (0,0) to opposite corner (M,N).

b. Euclidian Distance Function

```
double euclidian_distance(vector<double> a, vector<double> b){
    vector<double> euc_sum;

    for(size_t i = 0; i < a.size(); i++){
        euc_sum.push_back(pow((b[i] - a[i]), 2));
    }
    return sqrt(accumulate(euc_sum.begin(), euc_sum.end(), 0.0));
}
```

c. Matrix Formation Function.

```
vector<vector<double> > Form_Matrix(vector<vector<double> > A, vector<vector<double> > B){
    int M = A.size();
    int N = B.size();
    vector<vector<double> > euclidian_matrix;

    for(int i = 0; i < M; i++){
        vector<double> matrix_row;

        for(int j = 0; j < N; j++){
            matrix_row.push_back(euclidian_distance(A[i], B[j]));
        }
        euclidian_matrix.push_back(matrix_row);
    }
    return euclidian_matrix;
}
```

IV. PARTHENOGENESIS

From Greek word for "Virgin Birth." This portion of the genetic algorithm involves creating the first two parents for the genetic algorithm. This can be done easily via pseudo-random number generation (as it shows below) or by greedy algorithm variants such as the nearest neighbor search path. Doing so may decrease the total number of generations needed to find the optimal solution.

a. Parthenogenesis Function

```
bool parthenogenesis(vector<int> &parent, int N, int M){ //"Virgin Birth"
    int tracker_n = 0; //Trackers to keep track of position of matrix (N,M)
    int tracker_m = 0;

    if(N <= 0 || M <= 0){
        return false;  //exception handling ;)
    }
    for(;;){
        int g1 = rand() % 2;
        int g2 = rand() % 2; //Random 1 or 0.
        if(tracker_m >= M && tracker_n >= N){break;} //break (N,M) is reached

        tracker_m += g1; //Add more genes.
        tracker_n += g2;
        parent.push_back(g1);
        parent.push_back(g2);
    }
    return true;
}
```

Notice the solution (parent) is simply a vector of individual binary points, 1's and 0's. Each vector represents a path from *c(0,0) to c(M,N)* when parsed 2 integers at a time.

V. FITNESS

Each 2 digit iteration, or chromosome, of the solution vectors corresponds to a new group of Euclidian distances on the matrix (Euclidian_Matrix[m][n]). The combination of binary integers will advance according to the following pattern:

If c(x) is 0 and c(x+1) is 0, then continue.
If c(x) is 0 and c(x+1) is 1, then n = n++.
If c(x) is 1 and c(x+1) is 0 then, m++.
If c(x) is 1 and c(x+1) is 1 then n++, m++;

a. Fitness Function

```
double fitness(vector<vector<double> > euclidian_matrix, vector<int> c){
    int M = euclidian_matrix.size()-1;
    int N = euclidian_matrix[0].size()-1;
    int m_tracker = 0;
    int n_tracker = 0;
    vector<double> f;
    f.push_back(euclidian_matrix[0][0]);
    //Both signals start at [0],[0] so add back that position

    for(size_t i = 0; i<c.size()-2; i=i+2){ //iterate through chromosome, 2 by 2.
        if(c[i] == 0 && c[i+1] == 0){continue;} //if both are 0 c(0,0) continue.

        bool flag = true; //flag to shut down once c(M,N) is reached.
        if(n_tracker < N){ //Prevent from exceeding N range
            n_tracker += c[i+1];
            flag = false;
        }
        if(m_tracker < M){ //Prevent from exceeding M range
            m_tracker += c[i];
            flag = false;
        }
        if(flag == true){
            break; //Break if flag is thrown.
        }
         //Add euclidian distance to vector
        f.push_back(euclidian_matrix[m_tracker][n_tracker]);
   }
    return(accumulate(f.begin(), f.end(), 0.0)/f.size());
}
```

Notes:

(i) The denominator f.size() in the return value is used to normalize warping paths of different lengths.

VI. GENERATION

At this point, it may be of use to have a structure to pass through functions that will hold all the relevant information needed to perform the various transformations. The rest of this paper will reference this structure often.

a. Generation Structure

```
struct Generation
{
    int N; //Number of signal A frames
    int M; //Number of signal B frames
    vector<int> p1; //Parent 1's chromosomes
    vector<int> p2; //Parent 2's chromosomes
    double fitness_p1; //respective fitness scores.
    double fitness_p2;
};
```

VII. CROSSOVER

should occur somewhere in the mid 80% of the solution between 10% and 90%. For certain datasets, such as this one, it helps to crossover at two intervals. This keeps the children asymmetrical and helps hold off the repetitive outcome of inbreeding. The functions towards the end of the code will be described further on.

a. Crossover Function

```
void crossover(Generation gen, vector<int> &child1, vector<int> &child2){
    double pivot1 = rand() % 80 + 10; //Random double between 10 - 90
    double pivot2 = 0;
    do{
        pivot2 = rand() % 80 + 10; //Random double between 10 - 90
    }while(pivot1 == pivot2); //Exception handling for identical pivot points.
    pivot1 = pivot1/100.0; // transforming into percentage decimals.
    pivot2 = pivot2/100.0;
    int temp_m = pivot1 < pivot2 ? pivot2 : pivot1; //Arranging so that pivot 1 < Pivot 2
    if(temp_m == pivot1){pivot1 = pivot2; pivot2 = temp_m;}
    //Adding chromosomes from (0 - pivot 1) from parent 1 to child1.
    for(size_t i = 0; i < floor(gen.p1.size()*pivot1); i++){
        child1.push_back(gen.p1[i]);
    }
    //Adding bases from (0 - pivot 1) from parent 2 to child2.
    for(size_t i = 0; i < floor(gen.p2.size()*pivot1); i++){
        child2.push_back(gen.p2[i]);
    }
    //Adding bases from (pivot 1 - pivot 2) from parent 2 to child 1.
    for(size_t i = floor(gen.p1.size()*pivot1); i < floor(gen.p1.size()*pivot2); i++){
        child2.push_back(gen.p1[i]);
    }
    //Adding bases from (pivot 1 - pivot 2) from parent 1 to child 2
    for(size_t i = floor(gen.p2.size()*pivot1); i < floor(gen.p2.size()*pivot2); i++){
        child1.push_back(gen.p2[i]);
    }
    //Adding bases from (pivot 2 - M) from parent 1 to child 1
    for(size_t i = floor(gen.p1.size()*pivot2); i < gen.p1.size(); i++){
        child1.push_back(gen.p1[i]);
    }
    //Adding bases from (pivot 2 - N) from parent 2 to child 2.
    for(size_t i = floor(gen.p2.size()*pivot2); i < gen.p2.size(); i++){
        child2.push_back(gen.p2[i]);
    }

    perform_mutation(child1); //Mutate the children
    perform_mutation(child2);
    proofread(gen, child1); //Check for errors.
    proofread(gen, child2);
}
```

VIII. MUTATION

The Rate of mutation occurs at 0.03, which means that out of 100, 3 genes will be mutated. Mutation comes in 3 different flavors: insertions, deletions, and substitutions. Insertions push a new gene between two others, deletions remove a gene and substitutions invert a gene, i.e. 01 -> 10.

a. Mutation Function

```
void perform_mutation(vector<int> &c){
    for(int i = 0; i < c.size(); i = i + 2){
        int r = rand() % 300 + 1;//random integer between 1 and 300.
        if(r == 150){ //0.03 means that among 300 chromosomes, 1 will be mutated.
            int mut_opt = rand() % 3; //random int between 0 and 2.
            if(mut_opt == 0){ //insertion
                c.push_back(rand() % 2);
                c.push_back(rand() % 2);
            }
            else if(mut_opt == 1){ //deletion
                c.erase(c.begin()+i, c.begin()+(i+2));
            }
            else{ //substitiution
                c[i] = c[i] == 0 ? 1 : 0;
                c[i+1] = c[i+1] == 0 ? 1 : 0;
            }
        }
    }
}
```

IX. PROOFREADING

In DNA replication, proofreading is accomplished by proteins called polymerases, it can parse the replicated strand and makes sure the DNA will work. If it finds that the sequence of pairs will not work. The Red Queen Algorithm uses a proofreading function to parse through the children that have been crossed over and mutated, to ensure that the sequence will reach position (M,N). If the child sequence does not work, it launches a loop that corrects the genetic sequence so that fitness can be calculated without error.

a. Proofreading Function

```
 void proofread(Generation gen, vector<int> &c){
     int m_tracker = 0; //Trackers to keep track of position in matrix.
     int n_tracker = 0;
     if(c.size() % 2 != 0){c.push_back(rand() % 2);} //Adjust for uneven crossovers.

     for(size_t i = 0; i < c.size(); i = i+2){ //Iterate through child, 2 by 2.
         m_tracker += c[i];//track projected path.
         n_tracker += c[i+1];
     }
     /*
      * NOTE: it doesn't matter at this point if we are larger than N or M, the
      * fitness function will take care of that.
      * The following while loop will terminate when tracker_n and tracker_m
      * contain enough chromosomes to reach c(M,N).
      */
     while(m_tracker < (gen.M - 1) || n_tracker < (gen.N - 1)){
         int g1 = rand() % 2;
         int g2 = rand() % 2;
         c.push_back(g1);
         c.push_back(g2);
         n_tracker += g2;
         m_tracker += g1;
     }
     return;
 }
```

X. CAPACOCHA

Meaning "Solemn Sacrifice," capacocha was an Incan ritualistic sacrifice of children. Here the fitness values are compared to the current generation, and the strongest two of the four solutions will be chosen to father the next generation. With optimal solutions as parents, the children will never make it on to the next generation.

a. Capacocha Function

```
Generation capacocha(Generation g, vector<int> c, double f, int &gc){// "Sacrifice of Children"
    Generation next_gen; //Create return value.
    next_gen.M = g.M; //set M and N values for next Generation.
    next_gen.N = g.N;

    if(f < g.fitness_p2){ //if new fitness out-ranks parent 2
        if(f<g.fitness_p1){ //if new fitness out-ranks parent 1 as well.
            next_gen.fitness_p1 = f;
            next_gen.p1 = c; //move value for parent 1 to position parent 2
            next_gen.fitness_p2 = g.fitness_p1;
            next_gen.p2 = g.p1; //slaughter parent 2.
        }
        else{ //if new fitness only out-ranks parent 2.
            next_gen.fitness_p1 = g.fitness_p1;
            next_gen.p1 = g.p1;
            next_gen.fitness_p2 = f;
            next_gen.p2 = c; //slaughter parent 2.
        }
        gc++; //count generations.
        return next_gen; //return the new modified generation.
    }
    else{ //if new fitness does not out-rank either parent, return original generation.
        return g;
    }

}
```

XI. RED QUEEN ALGORITHM

a. Red Queen Function

```
void Red_Queen(vector<vector<double> > A, vector<vector<double> > B){
    const int MAX_ITERATIONS = 1000;
    int GEN_COUNT = 0;

    Generation fittest; //Generation to hold the fittest through iterative generations.
    fittest.M = A.size()-1; //Set size to adjusted vector sizes.
    fittest.N = B.size()-1;
    vector<int> Parent1; //Create vectors for the chromosomes of parents.
    vector<int> Parent2;
    double current_fit = 0;

    parthenogenesis(Parent1, fittest.N, fittest.M); //Virgin Birth of Parent 1
    parthenogenesis(Parent2, fittest.N, fittest.M); //Virgin Birth of Parent 2
    vector<vector<double> > euclidian_matrix = Form_Matrix(A, B); //Euclidian_matrix formation

    double p1_fitness = fitness(euclidian_matrix, Parent1); //Calculate fitness of Parent 1
    double p2_fitness = fitness(euclidian_matrix, Parent2); //Calculate fitness of Parent 2
    /*
      The following lines arange the components of the fittest generation so that the fittest
      parent is parent 1 and the lesser fit parent is parent 2. Keeping them in this order
      makes it easier later on.
    */
    double max_temp = p1_fitness > p2_fitness ? p1_fitness : p2_fitness;
    if(max_temp == p2_fitness){
        p1_fitness = p2_fitness;
        p2_fitness = max_temp;
        vector<int> temp_v = Parent2;
        Parent2 = Parent1;
        Parent1 = temp_v;
    }
    fittest.p1 = Parent1;
    fittest.p2 = Parent2;
    fittest.fitness_p1 = p1_fitness;
    fittest.fitness_p2 = p2_fitness;
    //Begin to iterate through generations.
    for(int i = 0; i < MAX_ITERATIONS; i++){
        vector<int> Child1; //each generation has 2 children.
        vector<int> Child2;

        crossover(fittest, Child1, Child2); //crossover occurs
        double c1_fitness = fitness(euclidian_matrix, Child1); //calculate fitness of children
        double c2_fitness = fitness(euclidian_matrix, Child2);
        fittest = capacocha(fittest, Child1, c1_fitness, GEN_COUNT); //slaughter the weakest
        fittest = capacocha(fittest, Child2, c2_fitness, GEN_COUNT);
        if(current_fit != fittest.fitness_p1){ //if new fittest, print value.
            cout<<"CURRENT GENERATION FITNESS: "<<fittest.fitness_p1<<endl;
            current_fit = fittest.fitness_p1;
        }
    }
    cout<<GEN_COUNT<<" GENERATIONS"<<endl;
}
```