

# Evolution of Industrial Melanism:

A Spatial, Predator-Prey Genetic Algorithm

*Project for Artificial Life*

Mikel Maron

Evolutionary and Adaptive Systems

University of Sussex

January 11, 2004

## **Abstract**

Industrial Melanism in England's Peppered Moths is the classic textbook example of microevolution. This simple and deep evolutionary example is an ideal testing ground for investigating many areas, through modeling, including predator-prey interactions, morphogenesis and pattern formation, vision, ecosystem response to pollution, and spatial dynamics. This paper describes the development of a spatial, predator-prey genetic algorithm, simulating bird predation on peppered moths, represented by cellular automata, and analyses the dynamics of the evolutionary response to a changing environment.

# 1 Introduction

Industrial Melanism in the Peppered Moth is the classic example of microevolution, an elegant and deep example connecting many interesting areas. The actions of civilization can have unintended and dramatic consequences on natural systems, and it is vitally important to understand and manage these activities for least harm to the environment. There are still many details to investigate in how the process of evolution responds to changes in the environment, to interaction with other species, and to spatial distribution of individuals.

This investigation focused on the construction of a model of Industrial Melanism. A stochastic, spatial Lotka-Volterra model was developed, with predation and reproductive success dependent on the fitness of the prey. Each moth's genes consisted of a cellular automata rule and state, which through a "developmental process", resulted in a wing pattern. Fitness was determined by visibility of that pattern against a random background, with density dependent on the level of "pollution". The clear, and somewhat unexpected, results of the experiment are detailed below.

Despite being such a popular example of Evolution, there have been surprisingly few serious Artificial Life style investigations. One exception, is a Peppered Moths model developed for education, in StarLogo [11]. However, this model differs in some important respects. There is no "development" in this model, rather a direct mapping from genotype to phenotype and the spatial distribution of predators is not considered at all.

## 2 Industrial Melanism in Peppered Moths

The story of the Peppered Moth, *Biston betularia*, is one of the best known examples of evolution and natural selection. This moth appears in a few different forms, differing in wing pattern; though until the mid 19th century only the *typica* variety, with speckled wings, was predominant. In the second half of the 19th century, the *carbonaria* variety, with dark wings, came to completely dominate industrialized areas of England. With the passage of clean air laws in the 20th century, forest pollution decreased and the speckled variety again became predominant. Naturalists came to the explanation that darker wings provided a selective advantage, since that form was more difficult to spot on the soot covered trees of the Industrial Revolution and therefore, less susceptible to bird predation.

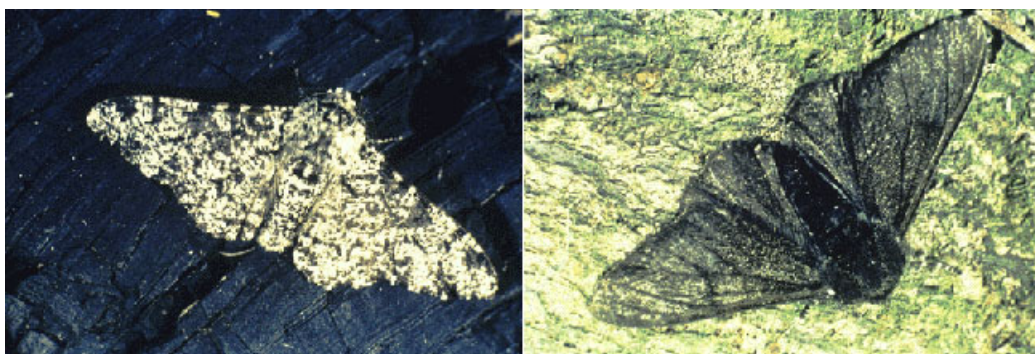


Figure 1: Peppered moths, in *typica* and *carbonaria* varieties

H. B. Kettlewell was the first to examine Industrial Melanism scientifically, in the 1950s, and went on

to widely publicize this example of evolution in action [3]. His core experiments consisted of releasing equal numbers of the two varieties in polluted and non-polluted forests, then recording the number of recaptured moths in subsequent days. In the polluted forests he recaptured eight times as many dark varieties as light varieties, and vice versa in clean forests. This seemed to clearly demonstrate the principles of micro-evolution.

Recently, Kettlewell's experiments, and the entire Peppered Moth story, have come under criticism [5]. Kettlewell released moths during the day, during the time they would normally be resting, and also placed the moths directly on trees, in places they may not normally seek refuge. He made no account of immigration from surrounding areas into his traps, and had little understanding of bird vision. On reexamination, there do seem to be many experimental flaws. This criticism of the prime example of evolution has been seized upon by creationists, and launched into a wider controversy and debate.

However, despite the renewed discussion, the basics of the story are still solid, a fact acknowledged by the chief critic of Kettlewell's procedures [5]. Camouflage in insects is widely noted. Butterflies, also in the family Lepidoptera, commonly have a similarly speckled dorsal wing pattern, in contrast to their conspicuous ventral wing patterns [9]. Butterflies rest with their wings folded up, exposing the dorsal pattern, while moths rest with wings open, exposing the ventral pattern. The similarity of this pattern, and behavior, suggests similar function. Additional experiments and observations have always confirmed the conclusion of industrial melanism. In Michigan, a subspecies of the peppered moth underwent a similar evolutionary trajectory, from speckled variety to dark and back, under changing pollution conditions [1].

### 3 Basis of the Model

This model incorporates ideas from several other experiments, elaborated here before discussing the implementation in depth.

#### 3.1 Spatial Predator-Prey Population Model

Lotka-Volterra is the well known set of differential equations that describes population fluctuations of species in a predator-prey relationship. Renshaw [10] follows a major theme of comparing deterministic and stochastic population models. Unlike deterministic models, stochastic models incorporate the variation of complex environments, allow for the possibility of extinction, and always maintain a whole, rather than real, numbered population size. In the stochastic model, the variables are used as a probability of some event taking place, rather than as a parameter to a differential equation.

However, the stochastic Lotka-Volterra model exhibits fundamentally different behavior. Both predator and prey populations quickly go extinct. To reconcile this difference, other assumptions in the model are questioned. Lotka-Volterra assumes populations are distributed homogeneously throughout the landscape, though most often populations are clumped into distinct subpopulations or colonies. When space is incorporated into the model, Lotka-Volterra type equilibrium reemerges [10], [6].

#### 3.2 Spatial Predator-Prey Genetic Algorithm

Incorporating space into genetic algorithms has been shown to enhance results [4]. Predators and prey are placed on a lattice. Similarly to Hillis [2], predators represent an optimization problem to solve and prey are potential solutions to that problem. In difference to Hillis, predators do not evolve as well. During each generation, prey move and attempt to reproduce with high fitness individuals in their

neighborhood. Predators also move and prey upon low fitness individuals in their neighborhood, which are completely removed from the population. The spatial distribution of prey ensures sustained genetic variability in the population.

Spatial predator-prey genetic algorithms performed as well or better than standard GAs on benchmark optimization problems. There were no experiments to model natural populations, though interestingly, they suggest that adding additional biological plausability, such as energy level and lifespan, could enhance the effectiveness of the algorithm on optimization problems.

### 3.3 Evolving Cellular Automata

Mitchell [8] has shown the genetic algorithms are effective at evolving cellular automata for optimization problems. The gene of individuals represents the rules to a cellular automata. Fitness is determined by running the cellular automata from an initial state, representing the problem, to a final state, representing the solution.

## 4 Implementation

### 4.1 Landscape and Run

The simulation begins by creating a population of random moths on a discrete landscape, with wrap-around edges. Each moth occupies a unique position. Predators are also placed randomly on the landscape, though any number of predators can occupy a single location. Predators have a randomly assigned energy level. During the simulation, that energy level determines when predators can reproduce, and when they die. Various parameters are set, including initial population sizes, mutation rate, landscape size, probabilities for reproduction, death, and movement of predator and prey, and probability of predation.

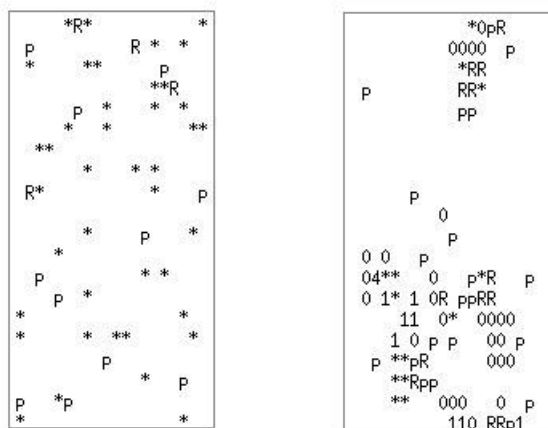


Figure 2: Two landscape snapshots. On the left, an initial random state, and on the right, later in a sample run. \* represents prey and 0 – 9 high fitness prey (0 is highest fit). p represents a predator and R is a predator and prey in a single location.

A run consists of iteratively executing events, depending on probability.

- Each prey has a probability of reproduction. In reproduction, an offspring is constructed by crossover (and mutation) with the highest fit individual in its Moore neighborhood, with the addition of the reproducing moth itself. If the moth has no neighbors, its offspring is simply a copy of itself, with mutation.
- Each prey has a probability of dying. A dead individual is completely removed from the simulation. Death is not tied to the fitness of the individual, instead representing a natural death rate for the population. The probability is dependent on the size of the entire prey population. In the absence of predators, the prey population would display logistic growth.
- Each prey has a probability of moving to an adjacent space on the lattice. The direction is chosen randomly.
- Each predator has a probability of birth. Reproduction simply copies a new identical predator to a neighboring site.
- Each predator has a probability of death. The energy level of the predator is decreased, and if below a threshold, that predator dies.
- Each predator has a probability of moving.
- Each predator has a probability of feeding. The least fit prey in the predators neighborhood is selected. If that prey has high fitness, then there's a further possibility that the prey will escape. If predation is successful, that prey is removed, and the energy level of the predator is increased. If the predator's energy level is above a certain threshold, it reproduces and its energy is halved. If predation fails, then the predator moves to an adjacent location.

Each event takes some random amount of time, dependent on the total amount of probabilities. It's calculated as  $-\log(\text{random})/R$ , where *random* is a random number between 0 and 1, and *R* is the summation of probabilities of all events. This time is added to the generation count.

## 4.2 Moth Wing Pattern and Vision

Each moth's wing pattern was represented by an 18 digit bitstring. The first 8 bits encode a one-dimensional, 2-neighbor, 2-state cellular automata look up table, in similar fashion to Mitchell [8]. The next 10 bits are the initial state for a cellular automata run. The wing pattern emerged as a "process of development", that is a run of the cellular automata for ten steps.

This process results in a wide variety of patterns, and a highly structured fitness landscape. Some small mutations in the bitstring are neutral, such as a change in a rule in the lookup table that is never consulted. Some are smooth, like a change in the initial state which shifts the overall pattern one step right. Other changes are dramatic, such as a change in the lookup table that results in an entirely different cellular automata class. Nijhout [9] finds examples of each of these evolutionary changes in the development of *Lepidoptera*.

The fitness of a moth is determined as follows. The cellular automata is run to build a wing pattern. Then, a background "tree" is constructed; essentially, it's a 50x50 random matrix of 1s and 0s, depending on a density parameter. The moth is placed at a random position on the tree, by replacing a section of the matrix with the cellular automata results.

Very basic image processing is used to determine how visible the moth is. The matrix is subdivided into 5x5 blocks, and the block with the density farthest from average is determined. If the moth is present

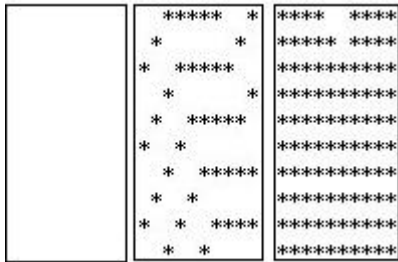


Figure 3: Three examples of wing patterns

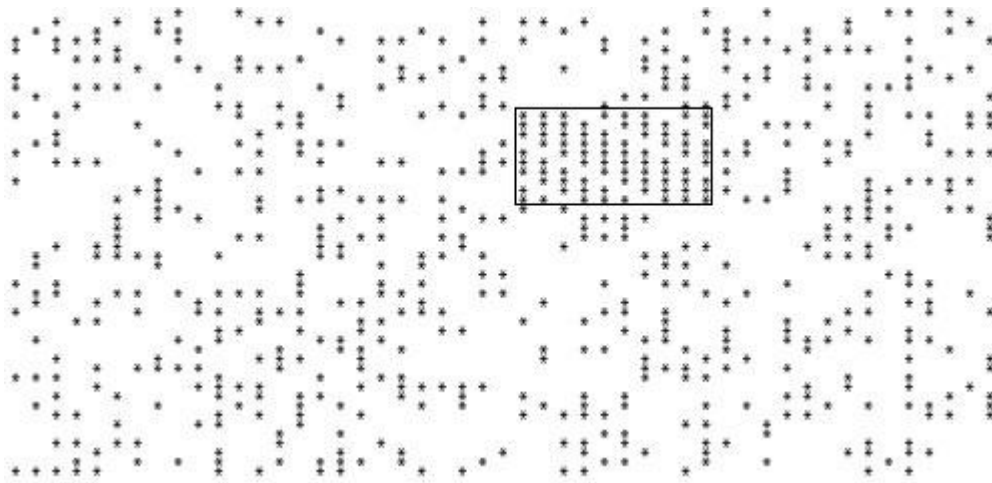


Figure 4: A sample moth highlighted on a background with .25 density

in that chosen block, then its fitness is calculated as minus the square of the difference from average density. If the moth is not present in the chosen block, then its fitness is 0. Fitness can vary, due to the randomness of the background and placement, so it is recalculated from time to time.

### 4.3 Code

This project was coded in C++, for execution on Unix based systems. The code was designed to be maximally modular, flexible, and reusable.

The core classes are `SpatialPopulation<I>` and `moth`. `SpatialPopulation<I>` is a subclass of `Population<I>`, and requires specification of the type of the individuals. `Population<I>` is a highly configurable class for running many styles of genetic algorithm, however the spatial predator-prey algorithm was complicated enough to warrant its own class.

`moth` is a subclass of `bitstring`, a type of individual that implements all the methods required by a `Population`. `moth` implements its own fitness calculation, using an internal member of type `CA`. `CA` is a class implementing a one-dimensional, two-state Cellular Automata, using bitstrings.

The project is tied together by `run.cpp`, which initializes the `SpatialPopulation`, runs a simulation of the changing environment, and outputs results.

## 5 Results and Analysis

The simulation was run with the initial prey population at 50, mutation rate .06, and Single Point Crossover. The Predator population was started at 15, in a  $20 \times 20$  arena. For prey, the growth probability was .6, the death probability .1 with a logistic limit of 100, and the movement probability .2. For predators, the growth probability was 0 (all growth was due to predation), a death probability of .6, a predation probability of 2, and a movement probability of .02. Prey were considered highly fit with fitness  $> -10$ , and predators were considered well fed with energy level 4.

The simulation was run for 500 generations. For the first 100 generations, the background density was held at zero, to simulate the pre-industrial non-polluted forest. For generations 100-200, the background density increased steadily, and from generations 200-300, the background density was at maximum, to simulate the height of industrialization. For generations 300-400, the density was gradually decreased, then from 400-500, held at minimum, to simulate the post-industrial environment. The simulation was run several times, with similar results each time.

The first question to ask is whether the run faithfully simulated the effect of Industrial Melanism. Fortunately, the answer is yes!

Initially none of the moths are adapted against the zero density background, though quickly the percentage of highly fit moths jumps and approaches 100. These moths are completely blank, to match the empty background. When the background density begins to rise, the number of highly fit moths drops, and then drops dramatically at about 30% density. It seems that many moths adapted to the zero background are able to evade detection with slight increases in background density, but are susceptible at a certain threshold. High fitness continues to decline until the background density reaches 100.

After a short time of stability, the moth population again approaches 100% highly fit individuals. At

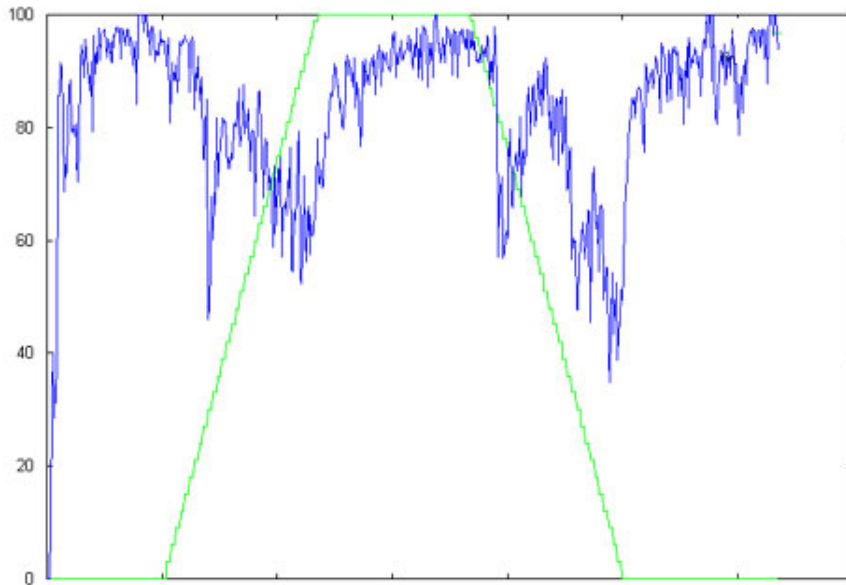


Figure 5: Percentage of Highly Fit Moths as Background Density Varies.

this point the background would be completely black, as are most of the moths. Environmental stability allows the population to become highly adapted. Similar behavior is noted in the “post-industrial” stability phase. At about 70% background density, there’s a sharp drop in fitness, as pollution adapted moths are no longer camouflaged. There’s some recovery, then a steady decline as the background density heads to zero. As the environment restabilizes, the population again becomes highly adapted.

Also, just to note, there is nothing particular about the moth development process that makes them better adapted to the extreme densities. The population will, in time, become highly adapted to any stable background density.

To further explain the dramatic drops in percentage of highly fit individuals during the transition periods, its useful to look at the genetic diversity of the population, measured as the percentage of unique genotypes in the population. During environmental stability, diversity drops to around 70%, as the population settles into successful strategies. Within the transition, diversity jumps, as the changing conditions don’t favor any particular strategy for long.

It’s also worth noting that Lotka-Volterra type dynamics are observed, with the addition of fitness dependent predation. Spikes in the population of moths are followed by population increase in birds, which leads to a drop in the moth population and subsequently a drop in the birds. At that point the cycle starts again, though with quite a lot of noise in this complicated arrangement.

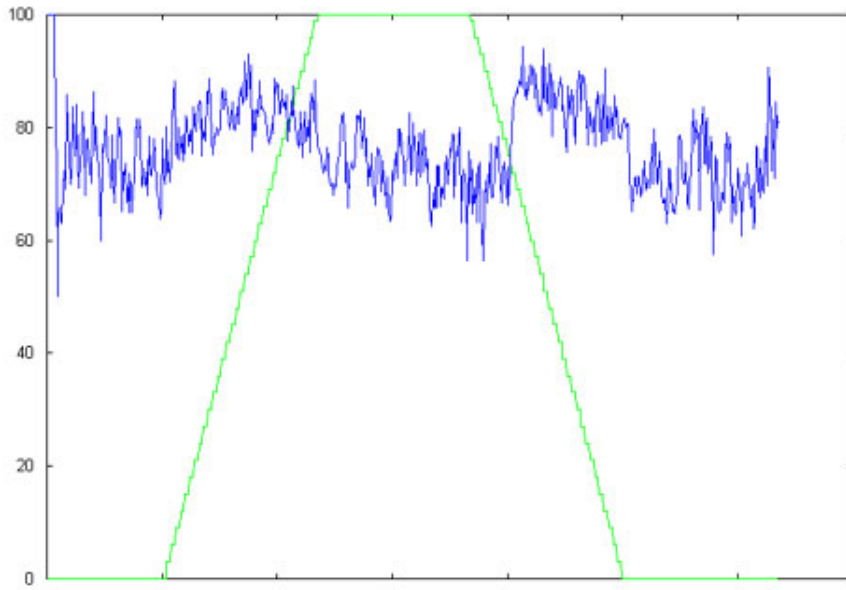


Figure 6: Genetic Diversity of Moths, as Background Density Varies.

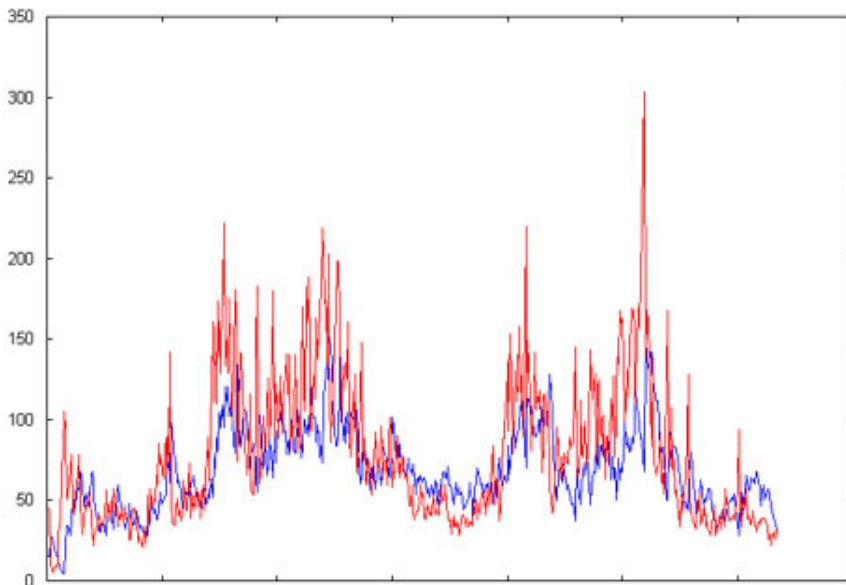


Figure 7: Population variation of Moths and Birds

## 5.1 Nonintuitive relationship between fitness and population size

The biggest surprise came in examining the variation of moth population size. Initially, the population dropped to below ten, made a strong recovery under evolution, then settled down into cycles around 30-50 individuals. During the transition period, the population size skyrockets and becomes quite chaotic. During the high density stability period, the population drops back down to cycles around 30-50 individuals. The same behavior is seen in the next transition period and the following stability period.

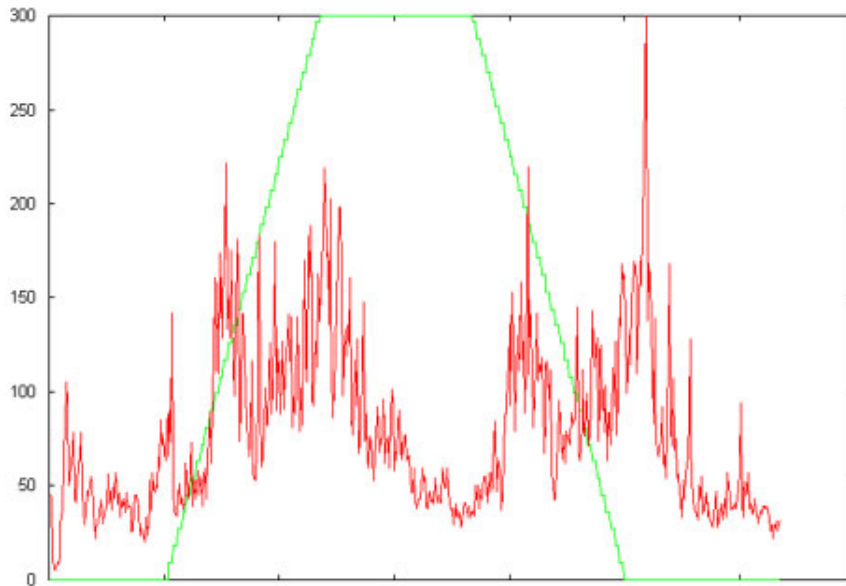


Figure 8: Population variation of **Moths** as **Background Density** Varies.

At first this seems counterintuitive. A highly adapted population would logically have a larger population size than a poorly adapted population. Yet, it's very strategic. Rapidly increasing the population size when the population is at low fitness is an effective method of searching the genetic fitness landscape. But this could not be the explanation; the fitness of the entire population is of no consequence to an individual moth. In fact, it turns out that the spatial dynamic plays the crucial part in the emergence of this counterintuitive population behavior.

When a predator attempts to feed and fails, it moves, in search of prey. In a high fitness moth population, predation will often fail, so the predators are highly mobile. In that situation, any newer low fitness moths are quickly found and high fitness moths are eventually found out. The high fitness of the moth population dampens their population growth, through the behavior of the predator.

When the moth population is low fitness, predators are relatively stationary. Its easy to find prey in the local neighborhood, and no movement is triggered. This leaves large areas of the landscape completely absent of birds, giving the opportunity of undisturbed, local moth population growth. Eventually, as the predator population increases, they are forced into movement, though quickly become stationary upon finding the new rich source of prey.

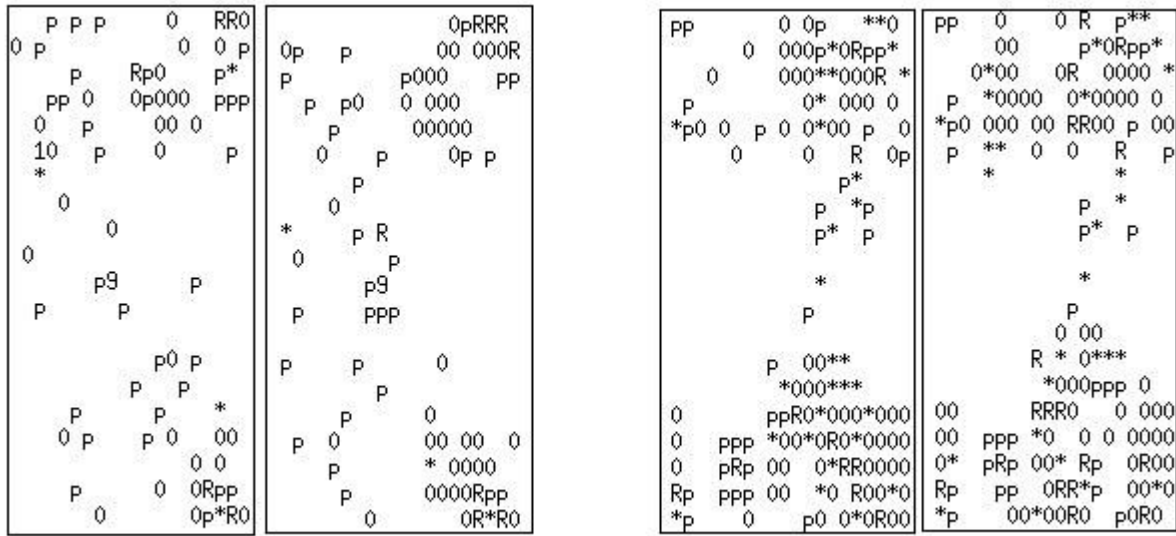


Figure 9: Sets of two consecutive states of the system, with one half generation time elapsed between. On the left, with highly adapted moths, many predators are mobile. On the right, with many low fitness moths, most all predators are relatively stationary.

## 6 Further Questions and Criticism

Overall, the simulation successfully demonstrated the objective, and provided some surprising results. Still, there are many problems to address, and future lines of inquiry are possible.

The biggest risk in this sort of biological modelling is developing toy simulations, with little biological relevance and parameters tweaked to just resemble the process being modelled [7]. Many of the parameter choices in this project, like the birth/death/move probabilities, are quite arbitrary or chosen to exhibit “good” behavior. Parameters like these should have some basis in the extensive research on this example, or should emerge from properties of the system.

Just as additional “reality” was added to this model from previous efforts, so was “unreality”. For example, reproduction of predators was completely asexual, and prey were sometimes asexual. This could possibly affect the dynamics of the simulation fundamentally. It would be worthwhile to simplify the model, and carefully add more complex features, to ascertain the particular dynamic changes of each portion of this model.

To complicate the model, the visual analysis has great room to improve. The bird’s visual system could even possibly co-evolve with the moth wing patterns. The pattern formation algorithm also has a great deal of possible extensions, into reaction-diffusion systems and the like.

The spatial dynamics offer many interesting possible experiments. Adding more constraint to the bird movement, so that every agent in the simulation occupied a unique area, could be interesting; it’s possible to envision high fitness moths forming barriers for the rest of the population. Another possibility is setting

up point sources of pollution, so that the degree of pollution would vary over the landscape, possibly producing spatial gradients of adaptations.

In analysis of spatial dynamics, it's worth pursuing more detail into what sort of population configurations result in system behaviors. There is a good deal of theoretical and practical research in this area in population biology. Another area for deeper analysis is measuring genetic diversity, moving from simply counting unique genotypes to general "coverage" of the genotypic landscape.

## References

- [1] Grant, Bruce S. *Fine Tuning the Peppered Moth Paradigm* Evolution 53(3):980-4, 1999
- [2] Hillis, W. Daniel *Coevolving Parasites Improve Simulated Evolution as an Optimization Procedure*. Artificial Life II, Addison-Wesley, 1991
- [3] Kettlewell, Bernard *The evolution of melanism :the study of a recurring necessity, with special reference to industrial melanism in the Lepidoptera*. Oxford University Press, 1973
- [4] Li, X. and Sutherland, S. *A Cellular Genetic Algorithm Simulating Predator-Prey Interactions*. Proceedings of the 4th Asia-Pacific Conference on Simulated Evolution and Learning, 2002
- [5] Majerus, M. E. N. *Melanism: Evolution in Action*. Oxford University Press, 1998
- [6] Maron, Mikel *Modelling Populations: From Malthus to the Threshold of Artificial Life*. <http://brainoff.com/easy/report.pdf>, 2003
- [7] Miller, Geoffrey E. *Artificial life as theoretical biology*. COGS Research Paper 378
- [8] Mitchell, M., Crutchfield, J., and Das R. *Evolving Cellular Automata with Genetic Algorithms*. Proceedings of the First International Conference on Evolutionary Computation and Its Applications, 1996
- [9] Nijhout, H. Frederik *The Development and Evolution of Butterfly Wing Patterns*. Smithsonian Institution Press, 1991
- [10] Renshaw, Eric *Modelling Biological Populations in Space and Time*. Cambridge University Press, 1991
- [11] Wilensky, Uri *Connected Mathematics: Peppered Moths*. <http://www.ccl.sesp.northwestern.edu/cm/models/moths/>, 2002

## A Code

### A.1 population.h

```
/*
 * Population Template Class
 *
 * Runs GA on population of any type of Individual
 * Configurable: Selection Type, Mutation Rate, Elitism, Crossover Type,
 * Population Size, Generations
 *
 */

#ifndef GA_POPULATION_READONCE
#define GA_POPULATION_READONCE 1

#include <cstdlib>
#include <list>
#include "gaconstants.h"

using std::list;

template <class I>
class Population {

protected:

    list<I> Indi;
    typename list<I>::iterator IndiIt;

    float genCount; //float, in case of steady-state GA
    int totalIndi; //total # of individuals created during run

    int maxGen;
    int popSize;
    float elitism;
    int replaceCount;
    int selectType;
    float mutationRate;
    int crossType;

    void TournamentSelect();
    void SteadyStateSelect();
    void ChooseTwoParents(I *p1, I *p2);
    void SortPopulation();

public:
```

```

Population () {};
Population (int ps, int st, float e, float mr, int ct, int mg);
~Population () { };

virtual void SelectAndMutate();
float Finished();
void RecalcAllFitness();

void Print(char *buf) { Indi.back().Print(buf); };
float GetGenCount() { return genCount; };
int GetTotalIndividuals() { return totalIndi; };
int IsMaxFit() { return Indi.back().IsMaxFitness(); };

int GetHighFitnessCount();
int GetGeneticDiversity();
};

#include "population.cpp"

#endif //GA_POPULATION_READONCE

```

## A.2 population.cpp

```

#include <map>

/*
 * Public Methods
 */

template <class I>
Population<I>::Population(int ps, int st, float e, float mr, int ct, int mg) {

    if (ps < 2) { ps = 2; }; popSize = ps;
    selectType = st;
    elitism = e;
    replaceCount = (int) ((1 - elitism) * popSize); if (replaceCount == 0) { replaceCount = 1; }
    mutationRate = mr;
    crossType = ct;
    maxGen = mg;

    totalIndi = 0;

    I *iP;
    for (int i = 0; i < popSize; i++) {
        iP = new I();
        iP->Create();
        Indi.push_back(*iP);
        totalIndi++;
    }
}

```

```

    genCount = 1;
    SortPopulation();

}

template <class I>
void Population<I>::SelectAndMutate() {
    if (selectType == GA_TOURNAMENT_SELECT) {
        TournamentSelect();
    } else if (selectType == GA_STEADYSTATE_SELECT) {
        SteadyStateSelect();
    }
}

template <class I>
float Population<I>::Finished() {
    if (Indi.back().IsMaxFitness()) { return genCount; }
    else if (genCount < maxGen) { return 0; }
    else { return genCount; }
}

template <class I>
void Population<I>::RecalcAllFitness() {
    for (IndiIt = Indi.begin(); IndiIt != Indi.end(); IndiIt++) {
        IndiIt->CalcFitness();
    }
    SortPopulation();
}

/*
 * Private Methods
 */

template <class I>
void Population<I>::TournamentSelect() {

    for (int i = 0; i < replaceCount; i++) {
        Indi.pop_front();
        //delete??
    }

    I *p1;
    I *p2;
    I *iP;

    for (int i = 0; i < replaceCount ; i++) {
        ChooseTwoParents(p1, p2);
        iP = new I;
        iP->Crossover( *p1, *p2, crossType);
        iP->Mutate(mutationRate);
    }
}

```

```

        Indi.push_back(*iP);

        totalIndi++;
    }

    SortPopulation(); //Always keep sorted, by fitness
    genCount++;
}

template <class I>
void Population<I>::SteadyStateSelect() {
    int r;
    I *iP;
    I *p1;
    I *p2;

    ChooseTwoParents(p1, p2);
    iP = new I;
    iP->Crossover( *p1, *p2, crossType);
    iP->Mutate( mutationRate );
    totalIndi++;

    //Elitist replace; May want fitness proportional replace later
    IndiIt = Indi.begin();
    int i = 0; int t = rand() % replaceCount;
    while (i < t) { IndiIt++; i++; }

    Indi.erase(IndiIt);
    Indi.push_back(*iP);

    SortPopulation();

    float inc = (float) 1 / replaceCount;
    genCount = genCount + inc;
}

template <class I>
void Population<I>::ChooseTwoParents(I *p1, I *p2) {
    typename list<I>::iterator a;
    typename list<I>::iterator b;

    int j = 0;

    while (j < 2) {
        a = Indi.begin(); b = Indi.begin();
        int t = rand() % Indi.size(); int i = 0;
        while (i < t) { a++;i++;}
        t = rand() % Indi.size(); i = 0;
        while (i < t) { b++;i++;}
    }
}

```

```

    //if (a == b) { continue; }

    if ( *a < *b ) {
        if (j == 0) p1 = &(*b);
        else p2 = &(*b);
    } else {
        if (j == 0) p1 = &(*a);
        else p2 = &(*a);
    }
    j++;
}
}

template <class I>
void Population<I>::SortPopulation() {
    Indi.sort();
}

template <class I>
int Population<I>::GetHighFitnessCount() {
    int count = 0;
    for (IndiIt = Indi.begin(); IndiIt != Indi.end(); IndiIt++) {
        if (IndiIt->IsHighFitness()) {
            count++;
        }
    }
    return count;
}

template <class I>
int Population<I>::GetGeneticDiversity() {
    std::map<long, int> genes;

    for (IndiIt = Indi.begin(); IndiIt != Indi.end(); IndiIt++) {
        genes[ IndiIt->GetBits() ] = 1;
    }

    return genes.size();
}

```

### A.3 bitstring.h

```

/*
 * Bitstring Individual Class
 *
 * Implements Crossover(), Mutate(), IsMaxFitness(), Compare(), CalcFitness(), Create(), Print()
 * as required by Population Template Class
 * for bitstring individuals
 *
 * This is a Pseudo-Abstract class. Must be subclassed,

```

```

*   but not strictly, by requirements of Compare()
*   subclass must implement CalcFitness() and Print()
*
*/

#ifndef GA_BITSTRING_READONCE
#define GA_BITSTRING_READONCE 1

#include <cstdlib>
#include "gaconstants.h"

class bitstring {
private:
    int CrossHere (int crossType, int i, int r1, int r2);

protected:
    long bits;
    long fitness;
    int size;
    long maxFit;
    int highFit;

    int arenax, arenay; //for spatial GA

    void RandomBits();

public:
    bitstring ();
    bitstring operator=(bitstring a);

    void Create();

    virtual void CalcFitness() {};
    virtual void Print(char *buf) {};

    void Crossover (bitstring a, bitstring b, int crossType);
    int Mutate (float mutationRate);

    int GetFitness() { return fitness; }
    int IsMaxFitness() { return fitness >= maxFit; }
    int IsHighFitness() { return fitness >= highFit; }
    long GetBits() { return bits; }

    int X(int i=-1) { if (i == -1) { return arenax; } else { arenax = i; } };
    int Y(int i=-1) { if (i == -1) { return arenay; } else { arenay = i; } };

    bool operator<(bitstring b) {
        return GetFitness() < b.GetFitness();
    }
};

```

```
#endif // GA_BITSTRING_READONCE
```

## A.4 bitstring.cpp

```
#include "bitstring.h"
```

```
bitstring::bitstring () {  
    size = 16;  
    Create();  
}
```

```
void bitstring::RandomBits () {  
    bits = rand() % ( (1 << (size - 1) ) - 1);  
}
```

```
bitstring bitstring::operator= (bitstring a) {  
    bits = a.bits;  
    fitness = a.fitness;  
    size = a.size;  
    maxFit = a.maxFit;  
}
```

```
void bitstring::Create() {  
    RandomBits();  
    CalcFitness();  
}
```

```
void bitstring::Crossover (bitstring a, bitstring b, int crossType) {  
    bits = 0;  
    int mask = 1;  
    int r1 = rand() % size; int r2 = rand() % size;  
  
    for (int i = 0; i < size; i++, mask = mask << 1) {  
        if ( CrossHere(crossType, i, r1, r2) ) {  
            bits = bits + ( a.bits & mask );  
        } else {  
            bits = bits + ( b.bits & mask );  
        }  
    }  
    CalcFitness();  
}
```

```
int bitstring::CrossHere(int crossType, int i, int r1, int r2) {  
    if (crossType == GA_UNIFORM_CROSSOVER) {  
        return (rand() % 2 == 0);  
    } else if (crossType == GA_SINGLE_CROSSOVER) {  
        return ( i < r1 ? 1 : 0 );  
    } else if (crossType == GA_DOUBLE_CROSSOVER) {
```

```

        return ( i < r1 || i > r2 ? 1 : 0 );
    } else if (crossType == GA_NO_CROSSOVER) {
        return true;
    }
}

int bitstring::Mutate (float mutationRate) {
    int mask = 1;
    int mutant = 0;
    for (int i = 0; i < size; i++, mask = mask << 1) {
        if ( (rand() / (RAND_MAX + 1.0)) < mutationRate) {
            if (bits & mask) {
                bits = bits - mask;
            } else {
                bits = bits + mask;
            }
            mutant++;
        }
    }
    if (mutant > 0) {
        CalcFitness();
    }
    return mutant;
}

```

## A.5 moth.h

```

/*
 * Moth Individual Class
 *
 * This subclass of bitstring, calculates fitness based "visibility" of Cellular Automata
 * against a random background of given density.
 *
 */

#ifndef MOTH_READONCE
#define MOTH_READONCE

#include "CA.h"
#include "bitstring.h"

class moth: public bitstring {
    CA *ca;

public:
    moth() { size = 18; maxFit = 0; highFit = -10; fitness = -1023; };

    void CalcFitness();
    void Print(char *buf);
}

```

```

//
// bd is static, so only need to be set once for an entire population
//
static int BackgroundDensity(int inc=0) {
    static int bd = 0;
    bd += inc;
    return bd;
};

};

#endif

```

## A.6 moth.cpp

```

#include "moth.h"

void moth::CalcFitness() {
    //
    //Build Cell Automata
    //
    int r = bits & ((1 << 8) - 1);
    int s = (bits >> 8) & ((1 << 10) - 1);
    int mothsize = 10;
    int mothDensity = 0;
    ca = new CA(3,mothsize,mothsize,r,s);
    ca->iterate();

    //
    //Construct the "Tree"
    //
    int w = 50; int h = 50;
    int tree[w][h];
    int density = BackgroundDensity();
    for (int i = 0; i < w; i++) {
        for (int j = 0; j < h; j++) {
            if ((rand() % 100) < density) {
                tree[i][j] = 1;
            } else {
                tree[i][j] = 0;
            }
        }
    }
}

//
//Moth flies onto Tree
//
int x = rand() % (w - mothsize);
int y = rand() % (h - mothsize);

```

```

for (int i = 0; i < mothsize; i++) {
    for (int j = 0; j < mothsize; j++) {
        tree[x+i][y+j] = ca->onOff(i,j);
        mothDensity += ca->onOff(i,j);
    }
}

//
//"Bird Vision"
// Divide up surface into 5x5 squares
//
int sight[w/5][h/5];
int count;
for (int i = 0; i < w/5; i++) {
    for (int j = 0; j < h/5; j++) {

        sight[i][j] = 0;

        for (int k = 0; k < 5; k++) {
            for (int l = 0; l < 5; l++) {
                sight[i][j] += tree[i*5 + k][j*5 + l];
            }
        }

        count += sight[i][j];
    }
}

//
//"Vision Processing"
// determine how visible moth is.
// simple difference from average density (not necessarily the same as BackgroundDensity
//
int avg = (int) count / 100;
int worst = 0; int worstx; int worsty;

for (int i = 0; i < w/5; i++) {
    for (int j = 0; j < h/5; j++) {
        if (((sight[i][j] - avg) * (sight[i][j] - avg)) > worst) {
            worst = (sight[i][j] - avg) * (sight[i][j] - avg);
            worstx = i;
            worsty = j;
        }
    }
}

//
// If spotted, fitness is its visibility. Otherwise it's "0", or invisible
//
if ( ((worstx * 5) >= x - 5) && ((worstx * 5) < x + 10) && ((worsty * 5) >= y - 5)

```

```

        && ((worsty * 5) < y + 10) ) {
    fitness = -worst;
} else {
    fitness = 0;
}
}

void moth::Print(char *buf) {
    sprintf(buf, ca->print().c_str());
}

```

## A.7 CA.h

```

/*
 * This class implements one-dimensional, two-state cellular automata
 * The Rule lookup table and States are stored as bit strings
 *
 */

#ifndef CELLAUTOMATA_READONCE
#define CELLAUTOMATA_READONCE

#include <string>

using namespace std;

class CA {
private:
    int neighborSize;
    int size;
    int numIterations;

    //rule and startState are bitstrings
    int rule;
    int startState;

    int *sequence;

public:
    /*
     n is neighborhood size
     s is world size
     i is number of iterations
     r is rule (as a bitstring)
     t is startstate (as a bitstring)
     */
    CA(int n, int s, int i, int r, int t);

    void iterate();

```

```

string print();

//used in fitness function for moth
int onOff(int i, int j);
int addup();
};

#endif

```

## A.8 CA.cpp

```

#include "CA.h"

#include <string>

using namespace std;

CA::CA(int n, int s, int i, int r, int t) {
    neighborSize = n;
    size = s;
    numIterations = i;
    rule = r;
    startState = t;

    sequence = new int[numIterations+1];
    sequence[0] = startState;
    for (int j = 1; j <= numIterations; j++) {
        sequence[j] = 0;
    }
}

void CA::iterate() {
    int h,i,j,k,l;
    int neighborhood,nextState;

    for (h = 0; h < numIterations; h++) {

        for (i = 0; i < size; i++) {
            neighborhood = 0;

            for (j = 0; j < neighborSize; j++) {
                k = j - ((neighborSize - 1) / 2);
                l = (i + k + size) % size;

                neighborhood += ((sequence[h] & (1 << l)) != 0) << j;
            }

            nextState = (rule & (1 << neighborhood)) != 0;
            sequence[h + 1] += nextState << i;
        }
    }
}

```

```

    }

}

}

string CA::print () {
    string buf;

    for (int i = 0; i <= numIterations; i++) {
        for (int j = size - 1; j >=0; j--) {
            if (((sequence[i] & (1 << j)) >> j) == 1) {
                buf += "*";
            } else {
                buf += " ";
            }
        }
        buf += "\n";
    }

    return buf;
}

int CA::addup() {
    return sequence[numIterations];
}

int CA::onOff(int i, int j) {
    return (sequence[i+1] >> j) & 1;
}

```

## A.9 spatialpopulation.h

```

/*
 * This subclass of Population<I> implements the Spatial Predator Prey Selection
 *
 */

#ifndef GA_SPATIALPOPULATION_READONCE
#define GA_SPATIALPOPULATION_READONCE

#include "population.h"
#include <list>

using std::list;

template <class I>
class SpatialPopulation: public Population<I> {
private:

```

```

I *** arena; //2D array of pointers
int arenaSize;
//Predator structure
struct coord {
    int x;
    int y;
    int fed;
};
list<coord> pred;
typename list<coord>::iterator predIt;

void SpatialSelect();
void preyBirth();
void preyDeath();
void preyMove();
void predBirth();
void predDeath();
void predEat();
void predMove();

float yG; //prey growth
float yD; //prey death
float yM; //prey move
int yC; //prey carrying capacity
float dG; //predator growth
float dD; //predator death
float dE; //predator eat
float dM; //predator move
int dF; //predator well fed -> reproduce
int hF; //high Fitness cutoff

public:
    SpatialPopulation(int ps, int st, float e, float mr, int ct, int mg):
        Population<I>::Population(ps,st,e,mr,ct,mg) { };

    /* must recalc fitness of individuals in the arena structure */
    void RecalcAllFitness();
    void SelectAndMutate();
    /* sets up additional parameters, and initializes additional structures */
    void SetSpatial(int ar, int numPred, float yg, float yd, float ym, int yc,
        float dg, float dd, float de, float dm, int df, int hf);

    void PrintArena();
    void PrintPopulation();
};

#include "spatialpopulation.cpp"

#endif

```

## A.10 spatialpopulation.cpp

```
#include <math.h>
#include <iostream>

using namespace std;

template <class I>
void SpatialPopulation<I>::SetSpatial(int ar, int numPred, float yg,
                                     float yd, float ym, int yc,
                                     float dg, float dd, float de,
                                     float dm, int df, int hf ) {

    arenaSize = ar;
    yG = yg;
    yD = yd;
    yM = ym;
    yC = yc;
    dG = dg;
    dD = dd;
    dE = de;
    dM = dm;
    dF = df;
    hF = hf;

    arena = new (I **)[arenaSize];
    for (int i = 0; i < arenaSize; i++) {
        arena[i] = new (I *)[arenaSize];
    }

    // pred = new coord[numPred];
    coord* coordp;
    for (int i = 0; i < numPred; i++) {
        coordp = new coord;
        coordp->x = rand() % arenaSize;
        coordp->y = rand() % arenaSize;
        coordp->fed = rand() % dF;
        pred.push_back(*coordp);
    }

    for (IndiIt = Indi.begin(); IndiIt != Indi.end(); IndiIt++) {
        //find a space and create
        int x = rand() % arenaSize;
        int y = rand() % arenaSize;

        if (arena[x][y] == NULL) {
            IndiIt->X(x);
            IndiIt->Y(y);
            arena[x][y] = &(*IndiIt);
        } else {
            typename list<I>::iterator tmp = IndiIt;
```

```

        tmp--;
        Indi.erase(IndiIt);
        IndiIt = tmp;
    }
}

template <class I>
void SpatialPopulation<I>::SelectAndMutate() {
    SpatialSelect();
}

template <class I>
void SpatialPopulation<I>::SpatialSelect() {
    float R = 0.0;

    //
    //Add up Probabilities for Prey Birth, Death, Movement
    //
    for (IndiIt = Indi.begin(); IndiIt != Indi.end(); IndiIt++) {
        R = R + (float) yG;
        R = R + (float) (yD / yC * Indi.size());
        R = R + (float) yM;
    }

    //
    //Add up Probabilities for Predator Birth, Death, Eat, Movement
    //
    for (predIt = pred.begin(); predIt != pred.end(); predIt++) {
        R = R + dG;
        R = R + dD;
        R = R + (dE * Indi.size());
        R = R + dM;
    }

    float event = R * (rand()/(RAND_MAX + 1.0));
    float S = 0.0;

    for (IndiIt = Indi.begin(); IndiIt != Indi.end() && event != -1; IndiIt++) {
        S += yG; if (event < S) { preyBirth(); event = -1; break;}
        S += (yD / yC * Indi.size()); if (event < S) { preyDeath(); event = -1; break; }
        S += yM; if (event < S) { preyMove(); event = -1; break; }
    }

    for (predIt = pred.begin(); predIt != pred.end() && event != -1; predIt++) {
        S += dG; if (event < S) { predBirth(); event = -1; break; }
        S += dD ; if (event < S) { predDeath(); event = -1; break; }
        S += (dE * Indi.size()); if (event < S) { predEat(); event = -1; break; }
        S += dM; if (event < S) { predMove(); event = -1; break; }
    }
}

```

```

SortPopulation();
genCount += - log( rand() / ( RAND_MAX + 1.0 ) ) / R;
}

template <class I>
void SpatialPopulation<I>::preyBirth() {
    I *mostFit = NULL;

    //Pick mate
    for (int i = -1; i <= 1; i++) {
        for (int j = -1; j <= 1; j++) {
            int x = (IndiIt->X() + i + arenaSize) % arenaSize;
            int y = (IndiIt->Y() + j + arenaSize) % arenaSize;

            if (arena[x][y] != NULL &&
                (mostFit == NULL || arena[x][y]->GetFitness() > mostFit->GetFitness())) {
                mostFit = arena[x][y];
            }
        }
    }

    //Place new individual
    for (int i = -1; i <= 1; i++) {
        for (int j = -1; j <= 1; j++) {
            int x = (IndiIt->X() + i + arenaSize) % arenaSize;
            int y = (IndiIt->Y() + j + arenaSize) % arenaSize;
            if (arena[x][y] == NULL) {
                I *newIndi = new I;
                newIndi->Crossover( (*IndiIt), *mostFit, crossType);
                newIndi->Mutate( mutationRate );
                newIndi->X(x);
                newIndi->Y(y);

                arena[x][y] = newIndi;
                Indi.push_back(*newIndi);
                totalIndi++;
                break;
            }
        }
    }
}

template <class I>
void SpatialPopulation<I>::preyDeath() {
    arena[ IndiIt->X() ][ IndiIt->Y() ] = NULL;
    Indi.erase( IndiIt );
}

```

```

template <class I>
void SpatialPopulation<I>::preyMove() {
    int newx = (IndiIt->X() + ((rand() % 3) - 1) + arenaSize) % arenaSize;
    int newy = (IndiIt->Y() + ((rand() % 3) - 1) + arenaSize) % arenaSize;
    if (arena[newx][newy] == NULL) {
        arena[newx][newy] = arena[ IndiIt->X() ][ IndiIt->Y() ];
        arena[ IndiIt->X() ][ IndiIt->Y() ] = NULL;
        IndiIt->X(newx);
        IndiIt->Y(newy);
    }
}

template <class I>
void SpatialPopulation<I>::predBirth() {
    coord* coordp = new coord;
    coordp->x = (predIt->x + ((rand() % 3) - 1) + arenaSize) % arenaSize;
    coordp->y = (predIt->y + ((rand() % 3) - 1) + arenaSize) % arenaSize;
    coordp->fed = rand() % dF;
    pred.push_back(*coordp);
}

//
// Predator Dies if it isn't "well fed"
//
template <class I>
void SpatialPopulation<I>::predDeath() {
    predIt->fed--;
    if (predIt->fed <= 0 || rand() % predIt->fed == 0) {
        pred.erase( predIt );
    }
}

template <class I>
void SpatialPopulation<I>::predEat() {
    I* leastFit = NULL;
    I* curIndi = NULL;

    //find leastFit individual
    for (int j = -1; j <=1; j++) {
        for (int k = -1; k <=1; k++) {
            curIndi = arena[ (predIt->x + j + arenaSize)
                % arenaSize ][ (predIt->y + k + arenaSize) % arenaSize ];
            if (curIndi != NULL && (leastFit == NULL || curIndi->GetFitness() < leastFit->GetFitness() ))
                leastFit = curIndi;
        }
    }
}

//
// High Fitness Individuals have proporionate chance of surviving

```

```

// Slightly app specific code here
//
if (leastFit != NULL) {
    if (leastFit->GetFitness() > hF) {
        int c = (-hF + leastFit->GetFitness()) / 2;
        if ( c <= 0 ) c = 1;
        if (rand() % c != 0) {
            leastFit = NULL;
        }
    }
}

//
//This is very inefficient
//
if (leastFit != NULL) {
    for (IndiIt = Indi.begin(); IndiIt != Indi.end(); IndiIt++) {
        if (IndiIt->X() == leastFit->X() && IndiIt->Y() == leastFit->Y()) {
            arena[ leastFit->X() ][ leastFit->Y() ] = NULL;
            Indi.erase( IndiIt );

            predIt->fed++;
            if (predIt->fed >= dF) {
                predBirth();
                predIt->fed = dF/2;
            }
            break;
        }
    }
} else {
    predMove();
}

template <class I>
void SpatialPopulation<I>::predMove() {
    predIt->x = (predIt->x + ((rand() % 3) - 1) + arenaSize) % arenaSize;
    predIt->y = (predIt->y + ((rand() % 3) - 1) + arenaSize) % arenaSize;
}

template <class I>
void SpatialPopulation<I>::RecalcAllFitness() {
    for (int i = 0; i < arenaSize; i++) {
        for (int j = 0; j < arenaSize; j++) {
            if (arena[i][j] != NULL) {
                arena[i][j]->CalcFitness();
            }
        }
    }
}

```

```

    Population<I>::RecalcAllFitness();
}

template <class I>
void SpatialPopulation<I>::PrintPopulation() {
    printf("Predator: %d; Prey: %d; Time: %f\n", pred.size(), Indi.size(), genCount);
}

template <class I>
void SpatialPopulation<I>::PrintArena() {
    for (int i = 0; i < arenaSize; i++) {
        for (int j = 0; j < arenaSize; j++) {
            int predHere = 0;
            for (predIt = pred.begin(); predIt != pred.end(); predIt++) {
                if (predIt->x == i && predIt->y == j) {
                    predHere = 1;
                }
            }

            if (arena[i][j] != NULL) {
                //Somewhat specialized code here
                char buf[1];
                if (arena[i][j]->GetFitness() > hF) {
                    sprintf(buf, "%d", - arena[i][j]->GetFitness());
                } else {
                    sprintf(buf, "*");
                }
                //

                if (predHere) {
                    cout << "R";
                } else {
                    cout << buf;
                }

            } else {
                if (predHere) {
                    cout << "p";
                } else {
                    cout << " ";
                }
            }
        }
        cout << "\n";
    }
}
}

```

## A.11 gaconstants.h

```
/*
 * Constants
 *
 */

#define GA_TOURNAMENT_SELECT 0
#define GA_STEADYSTATE_SELECT 1
#define GA_SPATIAL_SELECT 2

#define GA_SINGLE_CROSSOVER 0
#define GA_DOUBLE_CROSSOVER 1
#define GA_UNIFORM_CROSSOVER 2
#define GA_NO_CROSSOVER 3
```

## A.12 run.cpp

```
/*
 * Run the peppered moth simulation
 *
 */

#include <iostream>
#include <cstdlib>
#include <string>

#include "spatialpopulation.h"
#include "moth.h"

using namespace std;

typedef SpatialPopulation<moth> MothPopulation;

int main () {
    char solution[1023];
    srand(static_cast<unsigned>(time(0)));

    MothPopulation *P;

    P = new MothPopulation(50, GA_SPATIAL_SELECT, 0, .06, GA_SINGLE_CROSSOVER, 400);
    P->SetSpatial( 20, 15, .6, .1, .2, 100, 0, .6, 2, .02 , 4, -10);
    int i = 1; int inc = 3;

    moth::BackgroundDensity(0);

    while (1) {
        //Increase, then decrease, the background density
        if ((P->GetGenCount() * 2) > i) {
```

```

if (i % 4 == 0 && i > 100) {

    if (inc > 0) {
        if (moth::BackgroundDensity() >= 150) {
            inc = 0 - inc;
        }

    }
    moth::BackgroundDensity(inc);
    P->RecalcAllFitness();
    P->SelectAndMutate();
}

i++;

//Print High Fit Moth and Arena
P->Print(solution);
cout << solution;
cout << "\n\n";
P->PrintArena();

// Print Stats
int hf = P->GetHighFitnessCount();
int gd = P->GetGeneticDiversity();
printf("High Fitness: %d; GeneticDiversity: %d; ", hf, gd);
printf("Background Density: %d; ", moth::BackgroundDensity());
P->PrintPopulation();
}

P->SelectAndMutate();
}

}

```

### A.13 Makefile

```

moth: CA.o moth.o bitstring.o run.cpp
    g++ -L/usr/local/lib -I/usr/local/include run.cpp moth.o CA.o bitstring.o
    -ggdb -o moth

CA.o: CA.h CA.cpp
    g++ -c CA.cpp -ggdb -o CA.o

moth.o: moth.h moth.cpp
    g++ -c moth.cpp -ggdb -o moth.o

bitstring.o: bitstring.h bitstring.cpp
    g++ -c bitstring.cpp -ggdb -o bitstring.o

```