Genetic Algorithms

John Burkardt
Department of Scientific Computing
Florida State University
....
Mathematics Department
Ajou University
http://people.sc.fsu.edu/~jburkardt/presentations/...
genetic_2013_ajou.pdf

13 August 2013

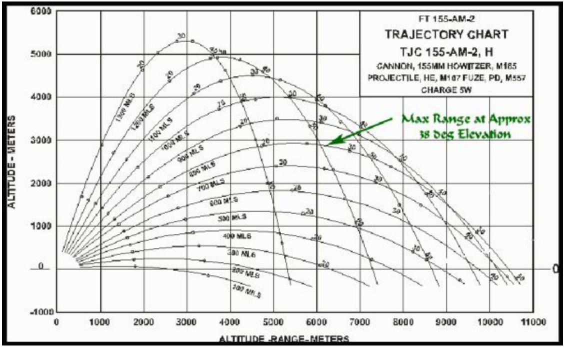
Reference

- Zbigniew Michalewicz,
  Genetic Algorithms + Data Structures = Evolution Programs,
  Third Edition,
  Springer, 1996,
  ISBN: 3-540-60676-9,
  LC: QA76.618.M53.
- Nick Berry,
  A "Practical" Use for Genetic Programming,
  http://www.datagenetics.com/blog.html
- John Burkardt,
  Approximate an Image Using a Genetic Algorithm,
  http://people.sc.fsu.edu/~jburkardt/m_src/image_match_genetic/image_match_genetic.html
- John Burkardt,
  A Simple Genetic Algorithm,
  http://people.sc.fsu.edu/~jburkardt/cpp_src/simple_ga/simple_ga.html
- Christopher Houck, Jeffery Joines, Michael Kay,
  A Genetic Algorithm for Function Optimization: A Matlab Implementation,
  NCSU-IE Technical Report 95-09, 1996.
- The Mathworks,
  Global Optimization Toolbox,
  http://www.mathworks.com/products/global-optimization/index.html

Introduction - Computers Implement “Recipes”

When computers arrived on the scene, scientists had had centuries to investigate important problems, develop a mathematical model that simulated the phenomenon, and algorithms to extract desired information.

One of the first uses of computers was to compute such artillery tables during World War II.



Introduction - Knock Down the Steeple

The flight of a cannon ball can be modeled as a mathematical parabola, with three degrees of freedom. The location and angle of the cannon determine two degrees and the amount of gun powder the third. By experiment, we can create tables that tell us whether we can hit the steeple, and if so, the minimum amount of gunpowder to use, and how to aim the cannon.

Introduction - Harder Problems

Thus, people tried to solve problems by understanding the model well enough to be able to see how a solution could be produced, and then implementing that procedure as an algorithm.

* Fourier developed a model of signals as sums of sines and cosines, and now we can solve heat conduction problems, and electrical engineers create electronic circuits with any desired property.

* Linear programming was able to solve many scheduling problems for airlines and allocation problems for factories.

But what happens when we find a problem that needs to be solved, but for which we simply cannot see an efficient way to determine a solution?

There are lots of problems like that!

Introduction - The Traveler

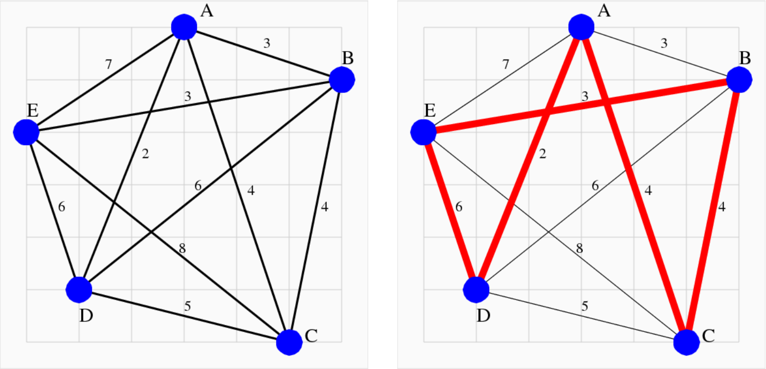
One simple problem involves a traveler who must visit plan a round trip that visits each city on a list once. The traveler is free to choose the order in which the cities are visited and wishes to find the route that minimizes the total mileage.

This problem is easy to state and understand, versions of it come up all the time in many situations, and yet there is
<u>no efficient algorithm for solving it!</u>

The only sure way to find the best route is to check every possible list. For n cities, that means $n!$ possible itineraries.

On the other hand, given two itineraries, it is certainly easy to check which one is better - just compare the mileages.

Introduction - The Traveler



Here's another simple problem: suppose you want to make a copy of a famous photograph or painting, but you are only able to use colored rectangles. You have rectangles in every size and color. The rectangles are transparent, and if two overlap, the overlapping region will have the average of the two colors.

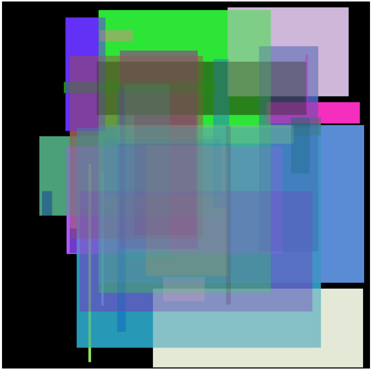
If that was the whole problem, we'd be done, because we make pixellated versions of images all the time.

But for this problem, we are only allowed to use 32 rectangles!

Suddenly, your mind goes blank. There's no obvious solution.

But given any two attempts to copy the picture, we can always determine which one is better, essentially by measuring the difference between the original and the copies.

Is it even possible to find an algorithm that can approximate the picture on the left using 32 colored rectangles? A random attempt is on the right.



Introduction - Optimization Problems

Our problem seems to be related to the standard **optimization problem**.

Let x represent data that defines a candidate to solve our problem.

Let $f(x)$ indicate a scalar function minimized by the best solution.

Our problem then is to minimize $f(x)$, and we have all the tools of calculus to help us...until we realize that our data x is constrained to be integers, and it is very likely that our function $f(x)$ is not differentiable, or continuous, even if we try to extend it to all real arguments.

Introduction - Discrete Optimization

To see why this problem becomes so hard, consider a simple example.

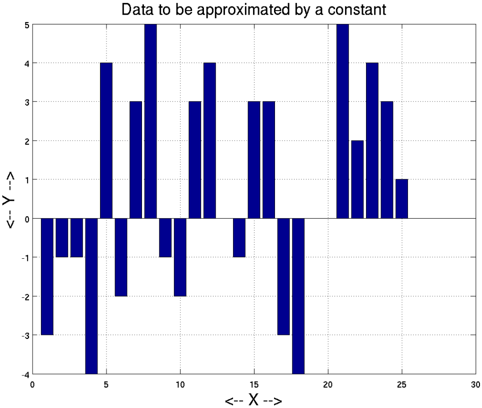
Suppose that at 25 equally spaced values x, we have values of a function $g(x)$.

Suppose we may choose a pulse function $p(x)$ with center c, half-width w, and height h, and we want to choose these (integer) values to minimize the function:

$$f(c, w, h; x) = |g(x) - p(c, w, h; x)|$$

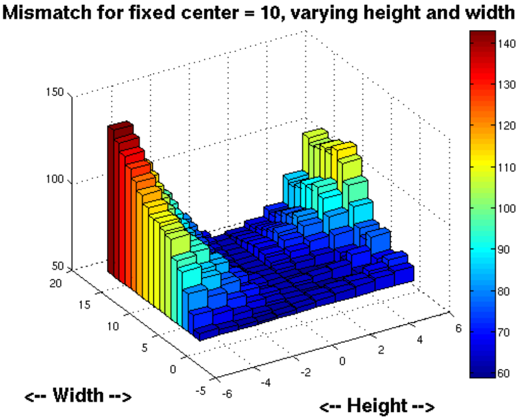
Because the parameters c, w and h are integers, if we look at plots of f, the surface is not smooth, but ragged, and contains many flat regions, and local minima. This suggests why the standard methods of calculus can be useless!

Here is an example of a function $g(x)$ to be approximated by a pulse:

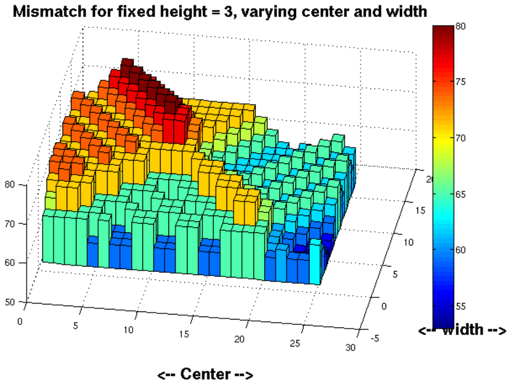


Introduction

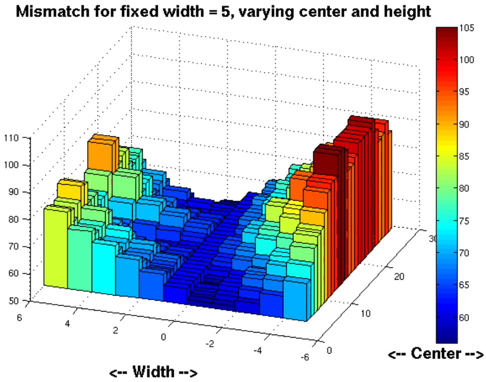
Plot $f(x)$ fixing the pulse center, varying height and width.



Plot $f(x)$ fixing the pulse height, varying center and width.



Plot $f(x)$ fixing the pulse width, varying center and height.



Our problem is a kind of **constrained search**, with two features:

- a finite (but big) list things to search;
- a function $f(x)$ that allows us to compare any two candidates (but **not** to say that one candidate is best!).

Solution methods might include:

- **brute force** - check every possibility;
- **Monte Carlo sampling** - sample many possibilities;
- **Hill climbing** - choose a candidate, then repeatedly search for small improvements;
- **Simulated annealing** - create a parameterized evaluation procedure; solve the easy version, then increase the difficulty a bit and solve that problem, and so on.

Introduction - Another Search Procedure

A drawback to sampling methods is that they have no memory; although they see many candidates, the sampling method doesn't look for patterns or structure *...when* x *is near* 2.5, *the value of* $f(x)$ *is pretty big....*

Meanwhile, a hill-climbing procedure picks a starting point at random, and focuses on improving that point. But it's easy to pick a starting point that's far from the best solution, or one that quickly reaches a local optimum, which can't be improved, but isn't the best.

Simulated annealing can get around the local optimum problem, but still has the problem that it concentrates on improving a single point.

- Introduction
- **Genetic Algorithms**
- A Simple Optimization
- The Patchwork Picture
- Conclusion

Genetic Algorithms - Biological Metaphor

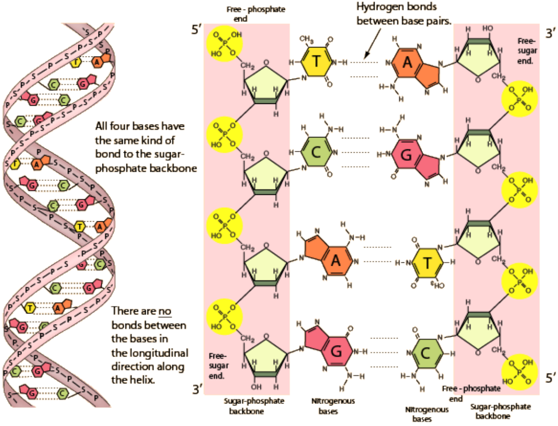
Genetic algorithms are based on a metaphor from biology, involving the ideas of genetic code, heredity, and evolution.

The suggestion is that life didn't know what it should look like, but kept trying new ideas. The crazy ones got squashed, and the better ones flourished.

We might assume life starts out "knowing" nothing about the world; however, after a while, the collection of living objects that are thriving embodies a great deal of implicit knowledge about the world - these are the body shapes and behaviors that work on this planet.

Let us recall some biology before we turn everything into numbers!

Genetic Algorithms - External Life Based on Internal Code



Flattened segment of the double helix

The sequence of A, C, G, and T pairs in DNA constitutes the genetic code.

Genetic Algorithms: Fitness, Survival, Modification

In our idealized model of biology, we notice that within a species, individuals exhibit a variety of traits.

We can explain these differences by looking at the genetic information carried by each individual. For a given species, we can think of the genetic information as being a list, of a fixed length, containing certain allowed "digits", perhaps **A**, **C**, **G**, **T**.

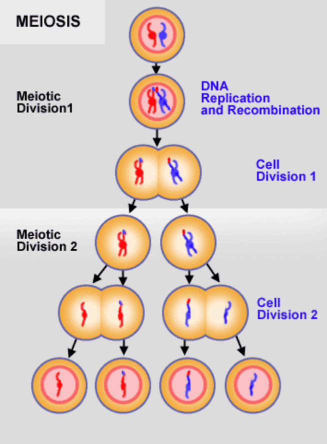
Two individuals of the same species, with different genetic information, will have different properties and abilities, and these differences will influence their relative fitness to survive.

Individuals that are more fit are more likely to survive and to breed.

Breeding requires two individuals, and involves the somewhat random selection of genetic information from each.

Changes in genetic information can also occur because of spontaneous mutation.

Genetic Algorithms - Meiosis



In meiosis, pairs of chromosomes containing genetic information are shuffled and split into halves, in preparation for combining with a half from the other parent.

Suppose we could measure the fitness of an individual named x for the given environment, producing a rating $f(x)$ which is higher for fitter individuals.

Then we could regard a population of individuals as an initially random sample of x values. In a "struggle" to eat, hold territory, and mate, the weaker individuals would be likely to die, their genetic information discarded.

Those individuals with the highest $f(x)$ values are preferentially allowed to reproduce - but reproduction involves shuffling the genetic information of two fit individuals to produce a new and different individual.

Over time, one might hope that the population of x values would evolve so that the values of $f(x)$ would increase, perhaps up to some limiting value.

Genetic Algorithms: The Genetic Algorithm Idea

A genetic algorithm is a kind of optimization procedure. From a given population X, it seeks the item $x \in X$ which has the greatest "fitness", that is, the maximum value of $f(x)$.

A genetic algorithm searches for the best value by creating a small pool of random candidates, selecting the best candidates, and allowing them to "breed", with minor variations, repeating this process over many generations.

These ideas are all inspired by the analogy with the evolution of living organisms.

Genetic Algorithms: Algorithmic Structure

A genetic algorithm typically includes:

1. a genetic representation of candidates;
2. a way to create an initial population of candidates;
3. a function measuring the "fitness" of each candidate;
4. a generation step, in which some candidates "die", some survive, others reproduce by breeding;
5. a mechanism that recombines genes from breeding pairs, and mutates others.

Genetic Algorithms: Choosing a Representation

The part of the genetic algorithm that varies the most from problem to problem will be the representation of the candidates.

Even for the traveling salesman problem, it's not too hard to see how to make this representation numeric - we can number the cities and then an itinerary simply lists those numbers in a particular order.

However, in order to make the breeding and mutation work, we will need to be able to convert the representation, whatever it is, back and forth to a binary representation, that is, a sequence of 0's and 1's.

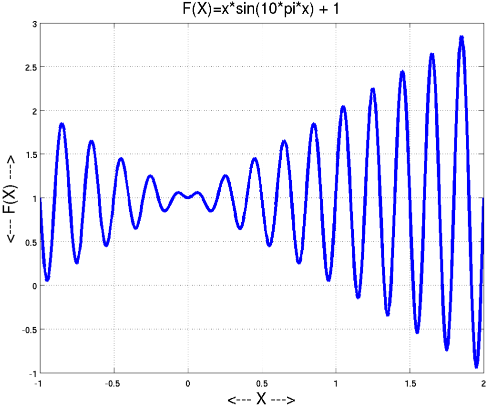
This has to be done in such a way that if we change one bit of the binary information, we are still describing a candidate.

OPTIMUM - Seek X that Maximizes F(X)

Let's begin by seeking to maximize a function:

$$f(x) = x \sin(10\pi x) + 1$$

over the interval $[-1, +2]$. The maximum occurs near $x = 1.8505$, where $f(x) = 2.85$.



OPTIMUM - Using Other Algorithms

To get six decimal places of accuracy, there are $3,000,000$ distinct values in $[-1,+2]$ we must consider.

A Monte Carlo scheme would have to sample perhaps 100,000 values before having a chance to find the right value.

A hill climbing scheme would have to start near the solution, or it will climb up one of the lesser mountains.

Our genetic algorithm will assume the solution can be coded using binary data, and will try to find that code using "evolution".

OPTIMUM - Represent the Possible Solutions

Six decimal digits of accuracy in $[-1, +2]$ corresponds to about 22 binary digits. A 22 digit binary string b can be converted to an integer k:

$$k = \sum_{i=1}^{22} b_i \cdot 2^{i-1}$$

The integer k becomes a real number u between 0 and 1 by:

$$u = k/(2^{22} - 1)$$

and u becomes a real number r between -1 and $+2$ by:

$$r = -1 + 3 \cdot u$$

so now we have a mapping between genetic information b and the objects r that we are interested in.

OPTIMUM - Initial Population

Each candidate is a string of 22 binary digits, which we might think of as an integer vector.

If we want a population of $n = 50$ candidates, then one way to do this would be to create a 2 dimensional array of size 50×22.

Now, to set up a random initial population, we simply need to randomly set the entries of this array to 0's and 1's.

One way to do this is to call the random number generator for each entry, and round each result.

```
b(i,j) = round ( random ( ) );
```

```
 i  ----------b-----------     ----x---
#1  1000101110110101000111  =  0.637197
#2  0000001110000000010000  = -0.958973
        ...
#50 1110000000111111000101  =  1.627888
```

OPTIMUM - Fitness Measurement

For this problem, it's easy to see that our fitness function is simply $f(x)$. We're looking for the candidate x in $[-1, 2]$ that makes this quantity the biggest. So we begin our iteration by measuring the fitness of each candidate:

```
 i   -----------b----------      ----x---      --f(x)--

#1   1000101110110101000111  =   0.637197  =>  1.586345
#2   0000001110000000010000  = -0.958973  =>  0.078878
           ...
#50 1110000000111111000101  =   1.627888  =>  2.250650
```

OPTIMUM - Death, Breeding, and Mutation

Based on our fitness results, we make the following decisions for the next generation:

- Out of our 50 candidates, let the 10 with lowest fitness "die";
- Let the 10 candidates with the highest fitness breed in pairs, creating 10 new candidates;
- Randomly select 2 of the nonbreeding, nondying candidates for mutation;

Our new population is the same size. It still contains the best candidates from the previous population, since it includes the 10 best, unchanged. The ten worst disappeared, replaced by offspring of the 10 best. For the intermediate 30, we kept 28 unchanged, and modified two by mutation.

Thus, the best fitness value in our population will never go down. Our goal, of course, is to make it go up, and quickly.

OPTIMUM - Implement Death, Breeding, Mutation

Death is easy - we just remove a candidate from the array.

Mutation is also easy. We pick a candidate i to mutate. We know the candidate has 22 bits of genetic information. We pick an index j between 1 and 22 and flip that bit:

```
b(i,j) = 1 - b(i,j)
```

For breeding, we have parents $i1$ and $i2$, creating children $i3$ and $i4$. We pick an index j between 1 and 21 and splice the parental information together:

```
b(i3,1:j) = b(i1,1:j) &  b(i3,j+1:22) = b(i2,j+1:22)
b(i4,1:j) = b(i2,1:j) &  b(i4,j+1:22) = b(i1,j+1:22)
```

OPTIMUM - Mutation Example

Recall that candidate #50 was:

```
 i   -----------b----------      ----x---      --f(x)--

#50 1110000000111111000101  =  1.627888  =>  2.250650
```

If we mutated the fifth gene in this candidate, the result is

```
    1110100000111111000101  =  1.721638  =>  -0.082257
```

and if we mutated the 10th gene, we would get:

```
    1110000001111111000101  =  2.343555  =>  2.343555
```

Thus, a mutation might decrease or improve the fitness value.

OPTIMUM - Breeding Example

Let us breed candidates #2 and #50:

```
 i   -----------b----------      ----x---       --f(x)--
#2   0000001110000000010000  = -0.958973  =>   0.078878
#50  1110000000111111000101  =  1.627888  =>   2.250650
```

If the crossover point is after $j = 5$, our two children will be

```
     -----------b----------      ----x---       --f(x)--
     0000000000111111000101  = -0.998113  =>   0.940865
     1110001110000000010000  =  1.666028  =>   2.459245
```

One child has a significant increase in the fitness function.

OPTIMUM - Running the Algorithm

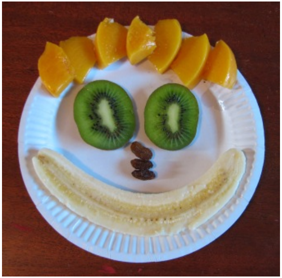
The algorithm took 145 generations to reach a good value:

```
-----------b----------      ----x---      --f(x)--
1111001101000100000101  =  1.850773  =>  2.850227
```

Gen	Best f(x)
1	1.441941
6	2.250003
8	2.250283
9	2.250284
10	2.250363
12	2.328077
39	2.344251
40	2.345087
51	2.738930
99	2.849246
137	2.850217
145	2.850227

PATCHWORK: Face = Sum of Fruits?

The Italian painter Arcimboldo enjoyed the puzzle of trying to approximate a human face using vegetables.



This is **not** the sort of problem you expect to give to a computer!

PATCHWORK: Face = Sum of Rectangles?

Nick Berry works for DataGenetics, and in his spare time, posts many short, interesting articles about computing.

He read an article about genetic programming, but didn't find the numerical example very interesting; after thinking for a while, he came up with a challenge that he wasn't sure genetic programming could handle, and so he put some time into the investigation.

Could he teach his computer how to paint a picture of him? Instead of using fruit, he would add together 32 rectangles of color.

This problem is much more realistic and useful than trying to optimize a function $f(x)$. We'll have to think about how to make it fit into the genetic algorithm framework, and we already know that there's no way that we can get a solution which is a perfect match for the original picture.

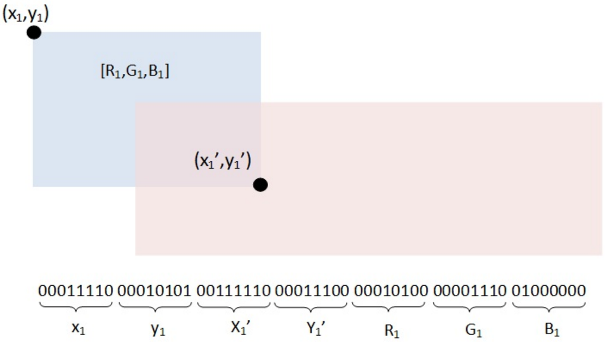
PATCHWORK - The Genetic Information

To make a genetic algorithm for this problem, we want to start by setting up the genetic information.

We assume the image is a **jpg** file containing 256×256 pixels. A candidate solution would be 32 colored rectangles. Each rectangle has a position and a color. Using the **jpg** format, the lower left corner is a pair (xl, yl) between 0 and 255, and (xr, yr) is similar. The color of the rectangle is defined by three integers, r, g, and b, also between 0 and 255. Thus, our numeric representation of a candidate solution is 32 sets of $(xl, yl, xr, yr, r, g, b)$, for a total of 224 integers.

A number between 0 and 255 requires 8 bits to specify, so our genetic information describing one candidate would require $224 * 8 = 1792$ bits.

PATCHWORK: One "Gene"



We have 10 candidates or "chromosomes". Each chromosome describes the color and position of 32 rectangles. The description of a single rectangle is termed a "gene".

PATCHWORK - The Fitness Function

Supposing we have specified a candidate solution x; then we must be prepared to evaluate its fitness $f(x)$.

To evaluate our candidate, we can simply create a 256×256 **jpg** file from the rectangles, and sum the (r, g, b) color differences pixel by pixel:

$$f(x) = \sum_{i=0}^{255} \sum_{j=0}^{255} |r1(i,j) - r2(i,j)| + |g1(i,j) - g2(i,j)| + |b1(i,j) - b2(i,j)|$$

where $r1$ and $r2$ are the reds for original and candidate, and so on.

Note that a perfect solution would have $f(x) = 0$, and that low values of $f(x)$ are better than high ones. We remember that for this problem, we are *minimizing* $f(x)$ instead of maximizing.

PATCHWORK - The Fitness Function

The rest of the procedure is pretty straightforward. We have to generate an initial random population of candidates, say 10 of them.

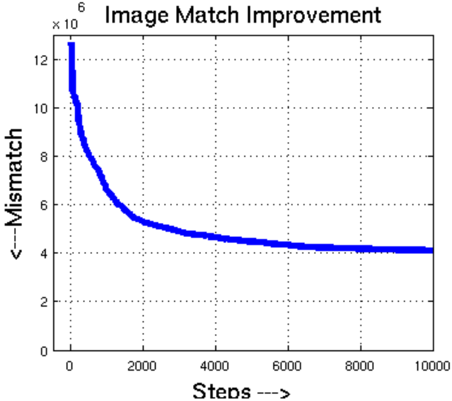
Our generation step will kill the 2 worst candidates, and replace them by the two children produced by breeding the two best candidates. Then we will also pick one candidate at random and hit it with a random mutation.

A typical run of the program might involve thousands or hundreds of thousands of generations. To monitor the progress of the program, we can look at how the fitness function evaluations are changing, or we can examine successive approximations to our picture.

PATCHWORK - Program Outline

```
read the original image "RGB_A";
generate 10 random candidates "B1", "B2", ..., "B10".

for 10,000 steps:

  convert each B to an image RGB_B;
  compare RGB_B to RGB_A to compute score;
  sort candidates, lowest to highest score.

  delete candidates B9 and B10;
  cross two remaining candidates, replacing B9 and B10;
  mutate one candidate of B2 through B8.

end
```

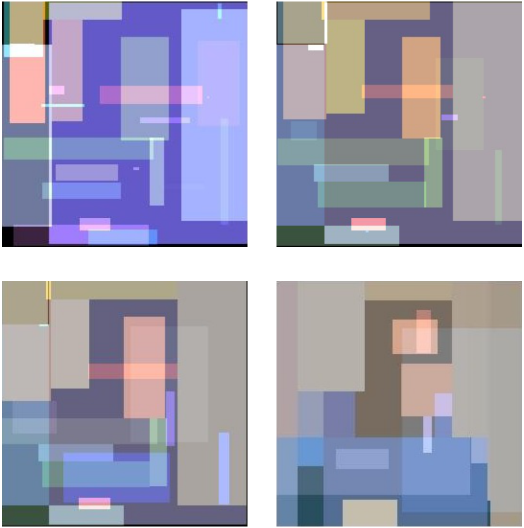
PATCHWORK - The Fitness Function



The graph of the fitness function values suggests that our rate of improvement is leveling off, and that it might be time to take a look at what we've computed!

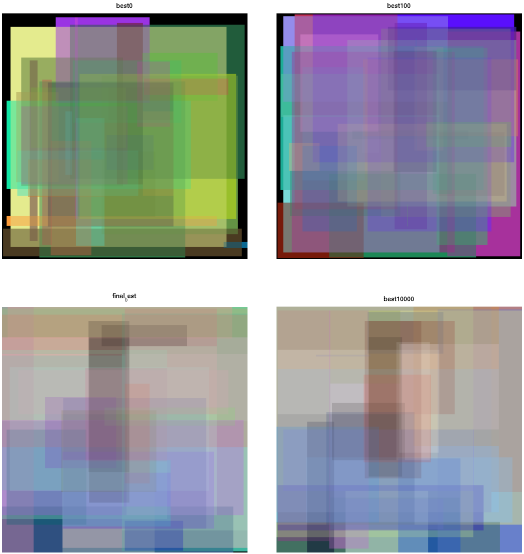
PATCHWORK - First, Middle, and Last Approximations

These were made with Nick Berry's program.



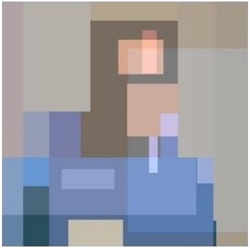
PATCHWORK - First, Middle, and Last Approximations

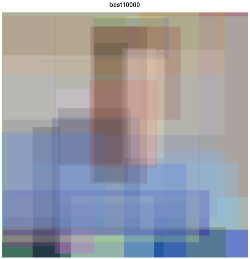
These were done with my own MATLAB program.



PATCHWORK - Comparison

Berry's approximation, the original, the MATLAB approximation





If you squint, or blur your vision, or stand back a distance, the approximations may start to look peculiarly good!

PATCHWORK - Color Combination

If you compare Nick Berry's results to mine, they seem to be by different "artists". In particular, it's easier to see the rectangle borders in Berry's pictures.

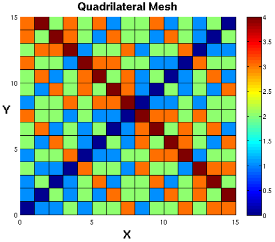
This is because, I believe, Berry and I differ in how we combined the colors when two (or more) rectangles overlapped. He takes the **maximum**, and I take the **average**. Thus, rectangles with RGB colors (50,100,150) and (250, 0, 100) would combine to

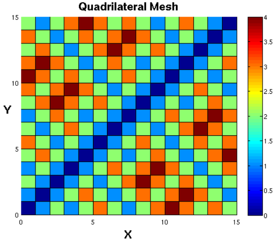
- (250, 100, 150) in Berry's method;
- (150, 50, 125) in my method.

so Berry's pictures are brighter and the boundaries between rectangles are sharper.

Another difference is that Berry used the **Gray code**, for which successive integers differ in just one digit, rather than the standard binary code.

For instance, the binary codes for 127 and 128 are 01111111 and 10000000, respectively. If the exact answer to our problem is 128, and we are very close, at 127, then in order for the genetic algorithm to move a distance of 1 in the “physical space”, it has to change all 8 digits in the “gene space”.





- Introduction
- Genetic Algorithms
- A Simple Optimization
- The Patchwork Picture
- **Conclusion**

Conclusion - Applications

Antennas	Turbine engines	Pharmaceuticals
Protein Folding	Factory scheduling	Stock trading
Commodity Pricing	VLSI partitioning	Circuit design
Chemical classification	Real estate appraisal	Healthcare

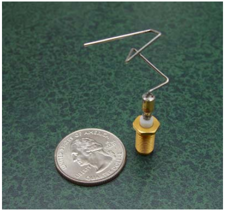


Figure: Left: Human-designed antenna, Right: Genetic Algorithm

Conclusion - Machine Learning

Genetic algorithms are an example of an exploding field in computational science, known as machine learning.

Instead of trying to come up with instructions for solving a problem, we can tell a computer what a solution looks like, let it look at lots of examples, and come up with its own solution methods. *(This is how spam filters work these days.)*

We used to think that if you can't understand how to solve a problem, then you can't solve the problem - but with genetic algorithms, we can see that's not always true.

One of the hardest problems we can imagine is how a life form can survive, outlive its rivals, and reproduce . . . and yet life has been stumbling on better and better solutions for a billion years.

Conclusion - THANKS

I thank Professor Hyung-Chun Lee for the invitation to speak!