Introduction to Genetic Algorithms

Peter G. Anderson, Computer Science Department
Rochester Institute of Technology, Rochester, New York
anderson@cs.rit.edu http://www.cs.rit.edu/

August 24, 2002

Abstract

An introduction to emulating the problem solving according to nature’s method: via evolution, selective breeding, “survival of the fittest.”

We will present the fundamental algorithms and present several examples—especially some problems that are hard to solve otherwise.
Dealing with Hard (NP-Complete) Problems

Some Problems We Just Don’t Know How to Solve! . . . but we do know how to critique a “solution.”

Coloring graphs is hard, but counting colors and violations is easy (a violation is two adjacent vertices with the same color).

Finding the shortest salesman’s path is hard, but measuring a path is easy.

Scheduling examinations or assigning teachers to classes is hard, but counting the conflicts (ideally there are none) is easy.

Computer programs are hard to write, but counting bugs is easy.
GAs Emulate Selective Breeding

Designing tender chickens is hard; taste-testing them is easy.
Designing thick-skinned tomatoes is hard; dropping is easy.

So, the breeders iterate:

- **Selection**: Cull their population of the inferior members.
- **Crossover**: Let the better members breed.
- **Mutation**: X-ray them.
Applications I Have Known

Choosing among 1,500 features for OCR.

Scheduling the Chili, NY, annual soccer invitational.

Scheduling my wife’s golf league.

Designing LED lenses.

Programming a synchronizing cellular automaton.

$N$ Queens, Graphs, Salesmen, etc., etc.
Subsetting 1,500 OCR Features

The polynet OCR engine trains and executes rapidly.

Performance was competitive.

We wanted to embed it in hardware, but it used 1,500 features.

We could deal with 300 features.

So, we bred high-performance feature subsets.
Soccer Scheduling

Bill Gustafson’s MS Project, May, 1998

The Chili Soccer Association hosts an annual soccer tournament.

131 teams, 209 games, 14 fields, 17 game times.

a long weekend for a group of schedulers,
. . . . . and then some teams back out. . .
Soccer Scheduling Hard Constraints

A field can have one game at a time.

A team can only play one game at a time.

Teams must play on appropriate size fields.

Late games must be played on lighted fields.

A team must rest one game period (two is better) between games.

Teams can only play when they can be present (some cannot come Friday evening).
Soccer Scheduling Soft Constraints

A team’s games should be distributed evenly over the playing days.

Teams should play in at most two playing areas.

Each team should play at least once in the main playing area.

Teams should play in areas where they have a preference.

Games should finish as early as possible on Sunday.

Etc...
Scheduling My Wife’s Golf League

24 golfers in the Tuesday morning league.

20 Tuesdays in a season.

4 golfers make a foursome.

Every Tuesday, every golfer plays with 3 others.

Every pair of golfers should play together 2 or 3 times.
Golf Scheduling

Schedule: 20 (= # of Tues.) permutations of 24 of golfers.

In a permutation, foursomes are the 1st four, next four, etc.

There are $\binom{24}{2} = 276$ pairs of golfers.

Each schedule has $\frac{24}{4} \times \binom{4}{2} \times 20 = 720$ pairs of golfers.

Fitness: the number of pairs that meet 2 or 3 times.
Designing LED Lenses

A mostly rectangular LED is embedded in a plastic lens. Light source is not a point. Lens design is not “textbook.” Given a lens, determine the efficiency and uniformity. Need to trace thousands of rays per lens (1–30 minutes). This is a multi-objective problem.
Evolving LED Lenses

Uniformity & efficiency of first 400 & last 400 of 5,846 individuals.
Programming Computers To...

Synchronize like the fireflies.

Discover FSMs given the sentences.

Control robots.
Placing $N$ Non-Attacking Queens

Found by my genetic algorithm!
Placing $N$ Non-Attacking Queens

Queens attack on chess-board rows, columns, and diagonals.

Any permutation in $N$ rows avoids row & column attacks.

Exhaustive search works for $N \leq 10$, but $N!$ grows rapidly.

A GA can place 1,000 Queens in 1,344 fitness evaluations.

(This is not an NP-complete problem.)
100 Non-Attacking Queens in 130 Fitness Evaluations
Graph Coloring: Edge Ends Get Different Colors

![Graph Coloring Diagram]
Graph Coloring = Map Coloring
Graph Coloring = Map Coloring
Graph Coloring = Map Coloring
Graph Coloring

Coloring an arbitrary graph is hard: NP complete.

We constructed random subgraphs of $K_{n,n,n}$ with large $n$.

So, the graphs were 3-colorable.

The probability of an edge was $p = 0.1$.

The GA solved this in 2,378 evaluations for $n = 200$.

Clever, random coloring colors about 50% of the vertices.
Traveling Salesman Route Min.

Another classic, hard, NP-complete problem.

We tried cities on a $H \times W$ grid, so best distance is known.

Perfection is hard to achieve.

A clever algorithm costs $O(cities^2)$ to evaluate a fitness.

But, we get pretty good answers.
Outline of This GA Course

The genetic algorithm.

Representation of solutions via bit-strings, etc.

Lots of examples

The GA programs and their parameterization (params)

Permutation GAs, ordered greed, and Warnsdorff’s heuristic

Other methods: random search, systematic search, hill climbing, simulated annealing, Tabu search
Famous Problems & Concepts

N Queens
Traveling salesman
Knight’s tour
Bin packing
Scheduling
Function optimization
Graph coloring, Ramsey problems
Satisfiability
Computing Paradigms

Finite state machines
Cellular automata
Trees, LISP
NP complete problems
A Genetic Algorithm

1. Obtain several creatures! (Spontaneous generation??)

2. Evolve! Perform selective breeding:
   (a) Run a couple of tournaments
   (b) Let the winners breed
   (c) Mutate and test their children
   (d) Let the children live in the losers’ homes
Evolution Runs Until:

- A perfect individual appears (if you know what the goal is),
- Or: improvement appears to be stalled,
- Or: you give up (your computing budget is exhausted).
Uses of GAs

GAs (and SAs): the algorithms of despair. Use a GA when

- you have no idea how to reasonably solve a problem
- calculus doesn’t apply
- generation of all solutions is impractical
- but, you can evaluate posed solutions
Chromosomes represent problems’ solutions as genotypes

They should be amenable to:

- Creation (spontaneous generation)

- Evaluation (fitness) via development of phenotypes

- Modification (mutation)

- Crossover (recombination)
How GAs Represent Problems’ Solutions: Genotypes

- Bit strings — this is the most common method
- Strings on small alphabets (e.g., C, G, A, T)
- Permutations (Queens, Salesmen)
- Trees (Lisp programs).

Genotypes must allow for: creation, modification, and crossover.
Bit Strings

Bit strings, \((B_0, B_1, \cdots, B_{N-1})\), often represent solutions well and permit easy fitness evaluations:

Sample problems:
- Maximize the ones count = \(\sum_k B_k\)
- Optimize \(f(x)\), letting \(x = 0.B_0B_1B_2\cdots B_{N-1}\) in binary
- Map coloring (2 bits per country color)
- Music (3 or 4 bits per note)

Creation, modification, and crossover are easy.
The Simplest GA’s

• Individuals are bit strings we call the chromosomes

• The bit strings represent solutions to some problem.
Sample Problems

Search for the best bit sting pattern for some application, such as:

• “Baby Problem 1” —
  find the bit string with the largest number of 1’s.

Not very interesting,
but this simple problem can prove that the system works.

It’s a “stub.”
Pop Sizes 5, 10, 100, 300, 1000 For Problem 1

5: too much exploitation. 1000: too much exploration.
Mutation Rates 0.001, 0.005, 0.007, 0.01 For Problem 1

0.01: too much exploration.
Tournament Sizes 2 . . . 20 For Problem 1
Sample Problems

- “Problem 2a”
  find a bit string of size $N = 2 \cdot M$
  representing a $2 \times M$ matrix whose two rows are identical.

- “Problem 2b”
  find a bit string of size $N = 2 \cdot M$
  representing a $M \times 2$ matrix whose two columns are identical.

Two identical problems illustrate importance of representations.
Sample Problems

Find the maximum value of the function

\[ f(x) = \sin(2\pi x^3) \sin(25x) \quad \text{for } 0 \leq x < 1 \]

The bit string \((b_1, b_2, \cdots, b_N)\) represents \(x\) in binary:

\[ x = 0.b_1 b_2 \cdots b_n = \sum_{k=1}^{N} b_k 2^{-k} \]
Sample Problems

Find the maximum value of the function

\[ f(x, y) = \sin(2\pi x^3) \cos(\cos(42y)) \sin(25x) + y^2 \]

for \(0 \leq x < 1\) and \(0 \leq y < 1\).

The bit string represents both \(x\) and \(y\).
Sample Problems

The prisoner’s dilemma strategy, This is a game for two players. Each, secretly, chooses cooperate or betray.
— If both cooperate, both win 1.
— If both betray, both lose 2.
— If only one betrays, he wins 3.

Last $K$ turn pairs determine your next move.

The last $K$ turns can be represented by a bit string of length $2K$, so there are $2^{2K}$ possibilities for the history value.

A $2^{2K}$-bit string represents a strategy.

The GA can run actual tournaments compare strategies.
**MIS: “Maximum Independent Set”**

Given a graph $G = (V, E)$, find the largest subset of vertices $W \subset V$ such that no two vertices of $W$ form an edge in $E$

$$\forall x_1, x_2 \in W, \quad (x_1, x_2) \notin E$$

Example graph: a hexagon.

Example: the “N Queens graph”

(Generally, MIS is NP-complete.)
MIS Fitness Evaluation

We want $||W||$ to be large.

We want the edges spanned by $W$ to be small.

Suggest fitness:

$$F = ||W|| - 3 \cdot E_W$$

$E_W$ is the number of edges spanned by elements of $W$. 
Graphs to Try for MIS

$P_n$ - a path of $n$ vertices

$C_n$ - a cycle of $n$ vertices

The edges of a square grid

We know the MIS for these graphs.
N Queens is an MIS Problem

The “Queen’s graph” is \( G = (V, E) \)

\( V \) is the set of squares (64 squares on an \( 8 \times 8 \) board).

\((a, b) \in E\) in case a Queen on square \( a \) can move to square \( b \).
Rooks MIS & Job Assignment

A **rook** is the chess piece that looks like a castle.

The rook moves on rows and column.

“N rooks problem” solution is a **permutation**.

Give each board square a value: $v[i][j]$: How well person $i$ does job $j$.

Use a modified fitness:

$$F = \sum_{(i,j) \in W} v[i][j] - 3 \cdot E_W$$
Gleason’s Problem

Given a “random” matrix, $M$, with values $\pm 1$'s
Change all the signs in some rows, and in some columns.
Try to maximize the number of $+1$'s.
Solution Representation

The size of $M$ is $C \times R$

Use a bit string of length $C + R$: $B = (b_1, \ldots, b_C, b_{C+1}, \ldots, b_{C+R})$

Meaning of $B$:

for $1 \leq j \leq C$, $b_j = 1 \Rightarrow$ invert column $j$

for $C + 1 \leq j$, $b_j = 1 \Rightarrow$ invert row $j$

Apply the changes dictated by $B$ to $M$, to get $M'$

Then the fitness of $B$ is the sum of all the elements in $M'$
Hill Climbing Gets Local Optima

Locate rows or columns with negative sums and invert them. This gets caught in local optima!

Example: $5 \times 5$ matrix with 17 $+1$'s.

$$
\begin{bmatrix}
+1 & +1 & +1 & -1 & -1 \\
+1 & +1 & +1 & -1 & -1 \\
+1 & +1 & +1 & +1 & +1 \\
-1 & -1 & +1 & +1 & +1 \\
-1 & -1 & +1 & +1 & +1
\end{bmatrix}
$$

Every row, column has positive sum. No single inversion can improve it!
Four Moves Improves

Invert rows 1 and 2, to lose two +1's.

\[
\begin{bmatrix}
-1 & -1 & -1 & +1 & +1 \\
-1 & -1 & -1 & +1 & +1 \\
+1 & +1 & +1 & +1 & +1 \\
-1 & -1 & +1 & +1 & +1 \\
-1 & -1 & +1 & +1 & +1 \\
\end{bmatrix}
\]

Invert columns 1 and 2, to gain six +1's.

\[
\begin{bmatrix}
+1 & +1 & -1 & +1 & +1 \\
+1 & +1 & -1 & +1 & +1 \\
-1 & -1 & +1 & +1 & +1 \\
+1 & +1 & +1 & +1 & +1 \\
+1 & +1 & +1 & +1 & +1 \\
\end{bmatrix}
\]
The Structure of My GA Code

Start-up

Main program loop

Tournament parent and loser selection

 Initialization subroutine (somewhat problem dependent)

“fv” — the fitness evaluation code (is problem dependent)

Crossover

Mutation

Randomization
GA Main Program: Init

main( argc, argv ) int argc; char **argv; { int who;
    params( argc, argv ); init();

    for( who = 0; who < POP_SIZE; who++ )
        fitness[who] = fv(who);
    printf( "End of initial pop\n. . . Now evolve!\n" );

    . . . main loop goes here . . . }

}
Notes on GA Main Program: Init

1. `arg` gives the name of a `param` file

2. `params( argc, argv )` processes that

3. `POP_SIZE` is an example of a run-time param
for( trial = 0; trial < LOOPS; trial++ ) {
    if( hero >= MAX_HERO ) {
        printf( "Goal reached: %d\n", hero ); break;
    }
    tournament( &p1, &c1 ); tournament( &p2, &c2 );
    make_children( p1, p2, c1, c2 );
    if( MUT_RATE > 0.0 ) { mutate( c1 ); mutate( c2 ); }
    fitness[c1] = fv(c1);
    fitness[c2] = fv(c2);
}
printf( "%d evaluations", fitness_evals);
printf( "Hero = %d\n", hero );
Notes on GA Main Program: Loop

1. This is the **steady-state** algorithm. It puts the children right back into the population. There is no notion of “generation”

2. **LOOPS** and **MUT_RATE** are examples of run-time params
void tournament( winner, loser ) int *winner, *loser; {
    int size = tournament_size, i, winfit, losefit;
    for( i = 0; i < size; i++ ) {
        int j = random_int( POP_SIZE );
        if( j==0 || fitness[j] > winfit ) {
            winfit = fitness[j];
            *winner = j;
        }
        if( j==0 || fitness[j] < losefit ) {
            losefit = fitness[j];
            *loser = j;
        }
    }
}
Initialization Subroutine

```
init()
{
    int PS, i, j;
    srand48( seed );
    hero = 0;
    PS = POP_SIZE;
    p = unchar_matrix( 0, PS-1, 0, N-1 );
    fitness = ivector( 0, PS-1 );
    MAX_HERO = N; /* for problem #1 */
    for( i = 0; i < POP_SIZE; i++ ) {
        for( j = 0; j < N; j++ ) { p[i][j] = random_int( 2 ); } 
    }
}
```
**Initialization Subroutine**

We use matrix creation code from *Numerical Recipes in C*. Their subroutines perform the malloc for us.

1. `ivector( low, high )`

2. `imatrix( low1, high1, low2, high2 )`
```c
int fv( who ) int who;
{
    int i, the_fitness = 0;
    fitness_evals++;
    for( i = 0; i < N; i++ ) the_fitness += p[who][i];
    if( print_every_fitness ) printf( "%4d fitness: ... ");

    if( the_fitness > hero ) {
        hero = the_fitness;
        printf( "New hero %4d ... ");
    }
    return( the_fitness );
}
```
Uniform Crossover

make_children( p1, p2, c1, c2 ) int p1, p2, c1, c2;
{
    int i, j;

    for( i = 0; i < N; i++ ) {
        if( random_int(2) ) {
            p[c1][i] = p[p1][i];
            p[c2][i] = p[p2][i];
        } else {
            p[c1][i] = p[p2][i];
            p[c2][i] = p[p1][i];
        }
    }
}
Mutation

Randomize each bit with probability $\text{MUT\_RATE}$

```c
mutate( who ) int who;
{
    int j;

    for( j = 0; j < N; j++ ) {
        if( MUT\_RATE > drand48() ) { p[who][j] = random\_int(2); } 
    }
}
```
Randomization Function

/* returns integer in range 0 <= r < n */
int random_int( n ) int n;
{
    return( (int) ( n * drand48() ) );
}
This is experimental programming so, we need to be able to experiment!

There are many parameters for a GA: mutation rate, tournament size, population size, random seed, ...

I don’t want to write a specialized parameter-reading program, a GUI, lots of #defines, etc.

I want to perform and keep track of lots of experiments.
Params: Parameterizing Code

The following makefile and other files illustrate *params*.

It is very easy to add new parameters to a program.
**Params: Parameterizing Code**

params.d defines and initializes some variables, the “parameters.”

params is read in at run-time to redefine the params.

params is like a list of assignments: the last one succeeds; and comments can track your progress.

Shell scripts can build and modify params files and invoke ga.

params.d and params are the only files the programmer needs to think about.
Makefile for ga

CFLAGS = -O -L/usr1/lib -I/usr1/include
LIBS = -lm
INC = nr.h nrutil.h params.h headers.h
MYOBJJS = ga.o nrutil.o params.o
$(MYOBJJS): $(INC)

ga: $(MYOBJJS) $(INC)
     cc $(CFLAGS) -o $@ $(MYOBJJS) $(LIBS)

params.h params.c: params.d
     ./make-params

clean:
     rm -f $(MYOBJJS) ga params.[ch]
params.d: Names & Values

%int
print_the_params 1
POP_SIZE 100
tournament_size 2
print_every_fitness 1
print_every_hero 1
LOOPS 1000000
debug 0
N 10

%float
MUT_RATE 0.001
seed 0.0 # random seed
File params.c Produced by make-params

```c
int print_the_params = 1;
int POP_SIZE = 100;
int tournament_size = 2;
int print_every_fitness = 1;
int print_every_hero = 1;
int LOOPS = 1000000;
int debug = 0;
int N = 10;
float MUT_RATE = 0.001;
float seed = 0.0;
...
params( argc, argv ) int argc; char * argv[];
{ ... }
```
extern int print_the_params;
extern int POP_SIZE;
extern int tournament_size;
extern int print_every_fitness;
extern int print_every_hero;
extern int LOOPS;
extern int debug;
extern int N;
extern float MUT_RATE;
extern float seed;
params Overrides Default Values

print_the_params 1
POP_SIZE 100
print_every_fitness 0
print_every_hero 1
LOOPS 1000000
N 100
MUT_RATE 0.001
seed 0.0 # random seed
## everything worked up to here
MUT_RATE 0.000001 this is very small!!!!
## this failed:
MUT_RATE 0.000001 this is very small!!!!
## now it worked!
Setting the Parameters

How do we choose the “params”? There is a statistical discipline: experimental design.

For the present, we will treat this issue informally. Try different setting and get a problem to work. Then systematically try various settings.
 Params for Prob. #1: Test Seed

%int
   print_the_params   1
   POP_SIZE         20
   tournament_size  2
   print_every_hero 1
   uniform         1
   FITNESSES      20000
   N              200
   problem_number 1
%float
   MUT_RATE        0.001000
   seed           0.000000

We then ran this with many seed values.
bash Script to Test seed Settings

Run with the previous parameters, and

seed = 0.1000 (0.001) 0.2000

(( SEED = 1000 ))
while (( SEED < 2000 ))
do
  ( cat params; echo seed 0.$SEED ) > ,params11
  ./ga ,params11 | grep Stop | awk '{ print $3 }'
  (( SEED = SEED + 1 ))
done
Seed Test Results

We ran 1,000 tests with different seed values.

The number of fitness evaluations to solve the problem was:

<table>
<thead>
<tr>
<th>count</th>
<th>fitnesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>124</td>
<td>2620</td>
</tr>
<tr>
<td>126</td>
<td>2860</td>
</tr>
<tr>
<td>125</td>
<td>3064</td>
</tr>
<tr>
<td>124</td>
<td>3096</td>
</tr>
<tr>
<td>125</td>
<td>3236</td>
</tr>
<tr>
<td>124</td>
<td>3260</td>
</tr>
<tr>
<td>125</td>
<td>3262</td>
</tr>
<tr>
<td>127</td>
<td>4256</td>
</tr>
</tbody>
</table>
Seed Variation Results

We ran the system 1,000 times with different seed values
Params Settings for Problem #1

print_the_params 1
POP_SIZE 200
tournament_size 2
print_every_hero 1
uniform 1
FITNESSES 10000
N 400
problem_number 1
MUT_RATE 0.002000
seed 0.000000

These settings worked!
Result: Stopped after 8824 fitness evals. Hero = 400
Shell Script to Test POP_SIZE

Run with the previous parameters, and POP_SIZE = 5 (5) 495

```bash
(( P = 5 ))
while (( P < 500 ))
do
  ( cat params; echo POP_SIZE $P ) > ,params
  ./ga ,params | grep Stop | awk '{print pop, $3}' pop=$P -
(( P = P + 5 ))
done
```

2000/05/08 77
Population Test Results: Tiny & Huge Pops Failed

We only allowed 10,000 fitness evaluations.
Many Fitnesses. Use param print_every_fitness 1
Parent Selection

Selection is a primary tool of a GA. (The other main tool is crossover.)

Here is a common technique: let $F = \sum_{j=1}^{\text{popsize}} \text{fitness}_j$

Select individual $k$ to be a parent with probability $\text{fitness}_k / F$

There are some problems here:
- fitnesses shouldn’t be negative
- probabilities should be “right;” avoid skewing by super heros.
Parent Selection: Rank

Here is another technique.
Order the individuals by fitness rank
Worst individual has rank 1. Best individual has rank \( \text{POP\_SIZE} \)

Let \( F = 1 + 2 + 3 + \cdots + \text{POP\_SIZE} \)
Select individual \( k \) to be a parent with probability \( \frac{\text{rank}_k}{F} \)

Benefits of rank selection:
- the probabilities are all positive
- the probability distribution is “even”
- tournament selection implements this
Parent Selection: Rank Power

Yet another technique.

Order the individuals by fitness rank
Worst individual has rank 1. Best individual has rank POP_SIZE

Let $F = 1^s + 2^s + 3^s + \cdots + POP\_SIZE^s$
Select individual $k$ to be a parent with probability $rank_k^s/F$

benefits:
• the probabilities are all positive
• the probabilities can be skewed to use more “elitist” selection
• $(s+1)$-element tournament selection implements this
Crossover Methods

Crossover is a primary tool of a GA. (The other main tool is selection.)

Common techniques for bit string representations:

One-point crossover: Parents exchange a random prefix

Two-point crossover: Parents exchange a random substring

Uniform crossover: Each child bit comes arbitrarily from either parent

(We need more clever methods for permutations & trees.)
Another Clever Crossover

Select three individuals, A, B, and C.

Suppose A has the highest fitness and C the lowest.

Create a child like this.

```c
for(i = 0; i < length; i++) {
    else child[i] = 1 - C[i];
}
```

We just suppose C is a “bad example.”
Crossover Methods & Schemas

Crossovers try to combine good schemas in the good parents.

The schemas are the good genes, building blocks to gather.

The simplest schemas are substrings.

1-point & 2-point crossovers preserve short substring schemas.

Uniform crossover is uniformly hostile to all kinds of schemas.
Crossover for Permutations (A Tricky Issue)

Small-alphabet techniques fail. Some common methods are:

- **OX**: ordered crossover
- **PMX**: partially matched crossover
- **CX**: cycle crossover

We will address these and others later.
Crossover for Trees

These trees often represent computer programs.

Think Lisp

Interchange randomly chosen subtrees of parents.
The Breeding Pool

steady-state algorithm simply:

- selected two parents (via a tournament)
- selected two losers (same tournament)
- created two children
- entered the children in the population

A common alternative is to use a breeding pool.
The Breeding Pool Algorithm

Start with an empty breeding pool with capacity $POP_{SIZE}$.

Until the breeding pool is full, do
   Copy individuals from current pop into the breeding pool
   (use some decent selection mechanism)

Initialize the new population to empty.

Until the breeding pool is empty, do
   Remove two parents from the pool & create two children
   Add the children to the new population
Mutation: Preserve Genetic Diversity

Mutation is a minor GA tool.

Without mutation we may get premature convergence to a population of identical clones.

A Genetic Algorithm must combine:

- Exploration (e.g., random search)
- Exploitation (e.g., hill climbing)
Mutate Strings & Permutations

- Bit strings (or small alphabets)
  - Flip some bits
  - Reverse a substring (nature does this)

- Permutations
  - Transpose some pairs
  - Reverse a substring

- Trees . . .
One Dimensional Optimization

Locate $x$, $0 \leq x < 1$ where the function $F(x)$ is maximized.

\[
F(x) = a_1 e^{-((x-c_1)/s_1)^2} + a_2 e^{-((x-c_2)/s_2)^2}
\]

$a_1, a_2, c_1, c_2, s_1, s_2$ are parameters.

$s_2 \ll s_1$ and $a_2 > a_1$, so max is near $x = c_2$ and hard to find.
A Function to Optimize

\[ \exp(-((x-0.3)/0.4)^2) + \exp(-((x-0.9)/0.04)^2) \]
A Function to Optimize

\[ \exp(-((x-0.3)/0.4)^2) + \exp(-((x-0.9)/0.004)^2) \]
A Function to Optimize: See Next Visual

\[ \exp(-((x-0.3)/0.4)^2) + \exp(-((x-0.9)/0.001)^2) \]
A Function to Optimize: Zoomed in to See the Max

\[ \exp\left(-\frac{(x-0.3)^2}{0.4}\right) + \exp\left(-\frac{(x-0.9)^2}{0.001}\right) \]
Fitness Calculation for 1D Function Optimization

```c
#define sqr(X) (X)*(X)

double F(x) double x;
{
    return a1*exp(-sqr((x-c1)/s1)) + a2*exp(-sqr((x-c2)/s2))
}

double fv( who ) int who;
{
    int i; double x = 0.0;
    for( i = 0; i < N; i++ ) { x = (x + p[who][i])/2.0; }
    return( F(x) );
}
```
Strings with Small Alphabets

- **Times** for examinations to be held.
  - minimize the pain of conflicts

- **Colors** for regions on a map.
  - avoid same colors on touching regions

- **Colors** for edges of a graph ("Ramsey").
  - avoid certain monochromatic subgraphs
One Dimensional Cellular Automata

This is an example of a GA used to write a computer program.

It is a difficult, tricky problem in a strange domain.

The GA succeeded!
One Dimensional Cellular Automata

An array of identical finite state machines.

State at time $t + 1$ depends on: state of self & neighbors at time $t$
How to Program a CA?

Suppose the cells are 2-state machines.

We want any initial configuration to eventually synchronize.

At time $t$, all machines are in state 0.

At time $t+1$, all machines are in state 1.

At time $t+2$, all machines are in state 0.

How can we program the system? What are good rules?
How to Program a CA?

Use 300 cells arranged in a circle.

The right neighbor of cell 299 is cell 0.

Use 7-cell neighborhoods: left three, self, right three.

7 cells can be in $2^7 = 128$ states. A program is a 128-bit string.

Each 128-bit program has a fitness.
A GA for a CA

Compute the fitness of a program as follows.

Initialize the 300 cells randomly.

Run the program through $t = 600$ times steps, plus one more.

The cells at $t = 600$ have values $(c_0, c_1, \cdots, c_{299})$

The cells at $t = 601$ have values $(d_0, d_1, \cdots, d_{299})$

The fitness of the program is

$$\left| \sum_{k=0}^{299} c_k - \sum_{k=0}^{299} d_k \right|$$
A GA for a CA — Note on the Fitness

Every CA program (look-up table) tries to bring an initial, random CA to synchronization.

Every individual is given a new random (hence unique) problem to solve.

This is like life! It’s not fair! We all meet some luck!

But, this avoids training the CA to solve a fixed initial pattern. Better, more general solutions may appear.
The CA Movies

The CA world is one dimensional, so a movie is a two dimensional picture.

The top line corresponds all the cells at time=0.

Index the CA movie as $CA_{time,position}$

$CA_{0,k} = \text{random}$

For time > 0,

$$CA_{t+1,k} = LUT\left( \sum_{i=-3}^{+3} 2^{3+i} CA_{t,k+i} \right)$$
A Random Start: Time Moves Down
Progress Towards Synchrony
Synchronized Success (Cropped Images for Clarity)
Synchronized Success
Synchronized Success
Synchronized Success
Results of a GA Run

Our GA wrote a program! (even if it is just a look-up table)

Population size 200

Uniform crossover

Achieved “perfection” after 1,386 fitness evaluations
(a random initialization synchronized)

The best individual in initial population had fitness of 36 (12%)
Other CA Problems to Try

Majority problem: if over half on (or off) turn all on (or off).

Does synchronization need a seven neighborhood?

Program two-dimensional CAs.

Experiment with the parameters:
(population size = 70 ⇒ 794 evaluations)
(population size = 60 ⇒ 450 evaluations)
(population size = 50 ⇒ 4,178 evaluations)
(Also test: tournament sizes, mutation rates)
Another CA Problems to Try: Three Colors

Program a three-states CA that eventually synchronizes.

A program is a string on three symbols of length $3^N$ ($N$ is neighborhood size)
The Brachistochrone Problem

In 1696, Johann Bernoulli asked, “What curve gives the frictionless, sliding roller-coaster down the track fastest?”

Johann & Jacob Bernoulli, Leibniz, and Galileo solved it with a cycloid, using the calculus of variations.

JuHo Kim (2002) solved it using a GA to search for good end-point placement for a piecewise linear path.
Evolving Brachistochrone: Indiv. # 0, 5K, 50K, 319K
More Problems to Solve with Bit-string GAs

1. Multivariariable function optimization.

2. Multiple circuit-board component assignment.


4. SAT, 3SAT

5. See the operations research library (ORLib): http://www.ms.ic.ac.uk/info.html
My First GA: OCR Feature Set Reduction

At Kodak, we had a hand-printed character recognition system (aka “OCR engine”) that I liked a lot.

It trained rapidly, ran rapidly, and was very accurate.

But, it used 1,500 image features, and we wanted to put the algorithm on a chip.

The challenge: determine 300 good features to use.
A Feed-Forward Network Classifier

“$P$ feeds $F$ feeds $K$” network:

- $P$ pixels in an image: $\overline{u}$
- $F$ hand-crafted quadratic polynomial features: $\overline{x}$
- $K$ classifications (10 digits, 26 letters, ...): $\overline{y} = \text{Net}(\overline{u})$
A Feed-Forward Network Classifier

\( \vec{y} = \text{Net}(\vec{u}) \) means:

\[
\vec{x} = \sigma B \vec{u}
\]

\[
\vec{y} = A \vec{x}
\]

Classification for \( \vec{u} \) is determined by the largest of the \( K \) entries in \( \vec{y} \)

The two “weights” matrices:

- \( B \) is hand-crafted
- \( A \) is solved in the least-squares sense
Features for the Polynomial Method

Hand-crafting the transform: It’s not really \( \bar{x} = \sigma B\bar{u} \)

The features are quadratic polynomials of pixels.

*Quadratic* means products of two pixels: \( BLACK = 1 \quad WHITE = 0 \)

Example feature set:

- “Kings:” any two corners of a \(2 \times 2\) square.
- “Big knights:” diagonal corners of \(3 \times 2\) or \(2 \times 3\) rectangles.

Almost every pixel could be used in 16 features.
Features for the Buffalo Polynomial Method
Weights

The transform, \( \bar{y} = Ax \) is determined by a pseudo-inverse.

Use \( N \) labeled training exemplars: \( \bar{u}_i \) has class \( k(i) \), for \( i = 1, \ldots, N \).

Create two matrices

- \( X \) (size \( F \times N \)) is the \( N \) feature vectors, \( \bar{x}_i \)
- \( Y \) (size \( K \times N \)) is the \( N \) target vectors, \( \bar{y}_i = \bar{e}_{k(i)} \)

“Solve for \( A \)” in: \( AX = Y \), using the pseudo-inverse:
\[
A = YX^T(XX^T)^{-1} = YX^#
\]

This \( A \) has the least total error: \( \sum_{i=0}^{N} ||A\bar{x}_i - \bar{y}_i||^2 \)
Initial Bottom Line

Reported by Uma Srinivasan, Masters Student, SUNY at Buffalo:

1. Use hand-printed digits, binarized, normalized to 16×16 pixels.

2. 1200–1358 features.

3. 16,000 training exemplars.

4. 93% correct (without rejections)
How Can We Use & Improve The PolyNet?

1. Least squares error is the wrong thing to minimize.
2. Cluster centers do not imply cluster boundaries.
3. Wrong answer count is the right thing to minimize.
4. $\bar{e}_{k(i)}$ is an arbitrary target.
5. We need to over-represent cluster boundary exemplars.
Recall: \( A = YX^T(XX^T)^{-1} \)

We do not need to create and use \( X \) and \( Y \).

We need two matrices \( W \) and \( Z \) (sizes \( F \times F \) and \( K \times F \), resp.):

\[
W = XX^T = \sum_{i=0}^{N} x_ix_i^T \quad Z = YX^T = \sum_{i=0}^{N} y_ix_i^T \quad A = ZW^{-1}
\]

No memory storage penalty for large \( N \). (The larger \( N \) the better.)
Augmentations to the Basic Polynomial Method

1. Iteration: retraining poorly classified training exemplars.

2. Negative reinforcement for wrong answers.

3. Subsampling the training set in early epochs.

4. Subsampling the features in early epochs.

5. Artificially enlarging the training set.
Iteration

Improve classification accuracy near the boundaries:

Create a classifier—that is, create an $A$.

Save intermediate matrices $W$ and $Z$.

For each training exemplar, $\bar{x}_i$, if $A\bar{x}_i$ is wrong, then retrain $x_i$

1. Add $\bar{x}_i\bar{x}_i^T$ to $W$

2. Add $\bar{e}_{k(i)}\bar{x}_i^T$ to $Z$

Then get a new $A = ZW^{-1}$, and repeat.
Negative Reinforcement

For each training exemplar, $\bar{x}_i$, if $\bar{y} = A\bar{x}_i$ is wrong, then, for some $j \neq k(i)$, either

- either, $\bar{y}_j > \bar{y}_{k(i)}$ (wrong answer)
- or, $\bar{y}_{k(i)} - \bar{y}_j < \theta$ (low confidence)

In both these cases:

- Add $\bar{x}_i\bar{x}_i^T$ to $W$ (as above)
- Add $(\bar{e}_{k(i)} - \bar{e}_j)\bar{x}_i^T$ to $Z$ (inhibition)
Improved Negative Reinforcement

We learned by experimenting about learning rates:

- In epoch $m$, add $m(2\bar{e}_k(i) - \bar{e}_j)x_i^T$ to $Z$
Subsampling the Training Set

Plan on spending $E$ epochs.

In epoch $m = 1, \ldots, \frac{E}{2}$, use $\frac{2m}{E}$ of the training data.

In epoch $m = \frac{E}{2} + 1, \ldots, E$, use all of the training data.

Adjust $\theta$ (desired confidence) so that approximately 15% of the training exemplars are retrained.
Artificially Enlarging the Training Set

The larger $N$ the better.

We can *memorize* a small training set.

Slide around every given training exemplar
one pixel *North, North-East, East, . . .*

get nine training exemplars for the price of one.
What Features to Use?

We want to classify

- digits
- upper-case alphabetics
- lower-case (mixed, really) alphabetics
- alphanumeric, multiple-font machine print

We decided to use 30×20-pixel images.

Features are dilations of chess *king* and *knight* moves.
King and Knight Features

The 5-king and 5-knight features

- are products of pairs of pixels
- diagonally opposite
- on the periphery of a $5 \times 5$ square of pixels.
King and Knight Features
King and Knight Features

Similarly define

- 3-, 5-, 7-, and 9-king features
- 5- and 9-knight features

These features can be modified to *fuzzy features*:

- products of pairs of *sums* of three adjacent pixels
- diagonally opposite
- on the periphery of an $s \times s$ square of pixels.
What the Features Do

- King and knight features gives evidence for a line of a given slope at a given position.

- Fuzzy features are more lenient about the slopes.
Where Are the Features?

Using 5-kings and 5-knights, every pixel can be the center of 8 features. This means we can have nearly 4,800 features.

$W$ is an $F \times F$ matrix which must be inverted.

Inversion has time complexity $O(F^3)$.

Some good news: $W$ is symmetric with a dominant diagonal.
Experimenting with $F$

We needed to experiment with the training time vs. classifier accuracy for different values of $F$.

The $F/8$ feature centers need to be evenly spread out.

A simple rule for the $k$-th feature center pixel:

$$\overline{p}_k = (11k\%28 + 1, 11k\%19)$$

For reasonable values of $F$, the feature centers $\overline{p}_k$, for $k = 1, \ldots, F/8$, are reasonably smoothly spread over the image.
Subsampling the Features

Another “trick” leads to an improvement.

A prefix of length $f$ of $\overline{x}_i$ is a reasonable feature vector.

Build $W$ and $Z$, as usual, but in early epochs compute $A = Z_f W_f^{-1}$:

- $Z_f$ is the left $K \times f$ submatrix of $Z$.
- $W_f$ is the upper-left $f \times f$ submatrix of $W$.

Result: we can quickly sketch a classifier, and late epochs can avoid retraining many exemplars.
Subsampling the Features

Some costs we save:

- matrix inversion: $O(f^3)$
- retraining: $O(F^2)$
Some Results

Without any rejects:

- Digits (10 categories): 99%
- Upper-case alphabetics (26 categories): 95%
- Lower-case alphabetics (40 categories): 90%
- Machine print (36 categories): 99%

(The first three experiments use NIST competition data.)
The Augmented PolyNet Method Is:

- Competitive with backprop-trained nets in accuracy.
- Faster to train than backprop by a factor of 10–20.
- Faster to run—it uses only integer operations.
Reduce the Number of Features — Why?

- 1,500 features is time-consuming to gather.
- So many features can over-fit the training data.
- Generalization may be poor.
- A smaller classifier might fit in hardware.
Reduce the Number of Features — How?

- Good hand-crafted subsets are too hard to find.

- The right subset is probably problem dependent.

- A genetic algorithm can do this search.
Reduce the Number of Features

Results:

- A digits classifier that uses 300 features.
- Classification accuracy matches that of the 1,500-feature classifier.
- The GA can spend all the computing budget we saved.
Problem Statement

Try to get a good, small, digits classifier.

Fitness is accuracy on a given testing set of 20,000 NIST digits.
Genetic Algorithm Particulars

“Individual” ↔ subset of the 1,500-feature set.
(Use a bit string of length 1,500 with 300 ones.)

“Fitness:” accuracy of classifier using those features.
(Use 20,000 testing exemplars.)

2 “parents” ⇒ 2 “children”
randomly assigning bits to children
(Subject to: exactly 300 one-bits per individual.)
Genetic Algorithm Particulars

Maintain a single population of 100–300 individuals.

Choose individuals to be parents based on fitness rank:

- Fix some $k$ in the range $2 \ldots 5$, then randomly select $k$ individuals. The best of these is parent 1.

- Randomly select $k$ different individuals. The best of these is parent 2.

- Probability of being a parent is proportional to $(k - 1)$st power of rank.

Two new children displace the two least fit individuals in the current population. (Could displace the tournament losers.)
Ways to Use a Subset, S, Subset the Feature Set

1. “Brain Damage?”
   Delete least useful columns of $A$.

2. “Optimal Brain Damage?”
   Pare down the feature set and retrain.

3. “Optimal Brain Surgery?”
   Use sub-matrices of well-trained $W$ and $Z$ matrices.
Find a submatrix $A_S$ of our best classifier matrix $A$.

This is the fastest approach to create a trial classifier.

Simply delete the columns we don’t want to use.

But we never got a better classification rate than 95%.
“Optimal Brain Damage?”

For a set $S$ of 300 features,

- Build a one-shot polynomial classifier.
  
  Create $W_S$, $Z_S$, and $A_S = Z_S W^{-1}_S$ corresponding to the features subset.

- Evaluate $A_S$ on the testing set.

This got us up above 97%. Iterated training got up to 98%.
The Next Clever Idea

The $W_S$ and $Z_S$, for a features subset $S$, are sub-matrices of $W$ and $Z$ for the whole feature set.

Speed things up by building the two big $W$ and $Z$ first.

Obtain $W_S$ and $Z_S$ without scanning the training set.

This speeds things up a lot!
The Next Clever Idea, Continued

Instead of $W$ and $Z$ from one pass over the training set,

- use the $W$ and $Z$ after iterated training,

- which created a good classifier.
“Optimal Brain Surgery?”

“The next clever idea” netted a classifier as good as the 1,500-feature one. It could not be improved by iteration.
Crossover: How Parents Make Children

An individual is a subset of a 1,500-element set. Represent an individual as a bitstring, $B = (b_1, b_2, \cdots, b_{1500})$ with exactly 300 of the $b_k$’s non-zero.

If $B$ and $B'$ are two individuals chosen to be parents, they agree at $m$ bit positions and disagree at $1500 - m$.

Create two children, $C$ and $C'$, making their bits identical to the parents’ where the parents’ bits agree.

$1500 - m$ is an even number (why?). So, in the positions where $B$ and $B'$ disagree set the bits of $C$ half one and half zero. Set those $1500 - m$ bits of $C'$ opposite to those of $C$. 
Mutation

Perform mutation by swapping two randomly selected bits.

Without mutation, we will converge.

A little mutation does not hurt—it may help.

A lot of mutation messes up the works.
Fitness Hill-Climbing From 96%–97% to 98.8%

At one evaluation per minute on a SPARCstation2, this takes 1–2 weeks.

The next visual shows the best 300-feature subset we located
Summary

• GA can locate 300-feature classifier to match 1,500-feature one.
• GA can take up the computing resources Poly training does not.
• Thought is still required—must search in the right place.
• Precision requirements can also be reduced for hardware.
Permutations: Now it Gets Interesting!

Use permutations when bit strings fall short:

- N Queens
- Assigning people to jobs.
- Assigning professors to sections.
- Assigning tasks to computers.
- Map coloring (inc. Ramsey).
N Queens—an Illustration

Place $N$ mutually un-attacking Queens on an $N \times N$ chess board.

(Queens attack on rows, columns, and diagonals.)

((Noise on the Internet: $\mathcal{NP}$ complete, indeed!))
Permutation Is Placement

Let the GA creatures be permutations of \((0, 1, \cdots, N-1)\):

Individual = \((c_0, c_1, \cdots, c_{N-1})\).

Interpret this as a Queens placement:

The Queen in row \(k\) is in column \(c_k\).

Fitness: number of Queens unattacked by Queens on previous rows.
How to Make a Permutation

// first, make the trivial permutation
for( i = 0; i < N; i++ ) P[i] = i;

for( i = 0; i < N; i++ ) {
    k = random_int( N - i );
    interchange P[i] with P[i + k]
}

This uniformly generates permutations of \{0, \cdots, N - 1\}
Crossing Over: the Problem

Cross over the two permutations

\[
\begin{array}{ccccccc}
6 & 7 & 4 & 3 & 1 & 0 & 5 & 2 \\
\hline
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7
\end{array}
\]

by swapping the indicated substrings.

The results are not permutations:

\[
\begin{array}{ccccccc}
6 & 7 & 2 & 3 & 4 & 5 & 5 & 2 \\
\hline
0 & 1 & 4 & 3 & 1 & 0 & 6 & 7
\end{array}
\]
Repairing the Crossover Damage

- PMX: “partially matched crossover”
- OX: “ordered crossover”
- CX: “cycle crossover”
PMX: Partially Matched Crossover

Pick an arbitrary position in two parent permutations:

<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
<th>4</th>
<th>3</th>
<th>7</th>
<th>5</th>
<th>1</th>
<th>0</th>
<th>9</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>9</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

That choice means to interchange 5 with 8 in both parents.

<table>
<thead>
<tr>
<th>5</th>
<th>2</th>
<th>4</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>1</th>
<th>0</th>
<th>9</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

Perform this operation several times, creating children with characteristics of both parents.
OX: Ordered Crossover

Pick about half of the elements of the first parent, (here, we choose 2, 4, 5, 1, and 6) and copy them to the child, preserving the positions. Choose the remaining values (0, 3, 7, 8, and 9) from the second parent, and copy them to the child, preserving the order.

\[
\begin{array}{ccccccccccc}
8 & 2 & 4 & 3 & 7 & 5 & 1 & 0 & 9 & 6 \\
4 & 1 & 7 & 6 & 2 & 8 & 3 & 9 & 5 & 0 \\
7 & 2 & 4 & 8 & 3 & 5 & 1 & 9 & 0 & 6 \\
\end{array}
\]

This preserves the some orderings of elements in both parents and position of some in the first parent.
This crossover preserves the position and value of everything. Follow the reasoning: if the first position of $C_1$ is 4, then the first position of $C_2$ must be 3. Then the 3 in $C_1$ must agree with $P_1$, so the 6 in $C_2$ must agree with $P_2$. And so on.

The consequences of the 4 in the first position of $C_1$ is:

<table>
<thead>
<tr>
<th>4</th>
<th>1</th>
<th>7</th>
<th>6</th>
<th>2</th>
<th>8</th>
<th>3</th>
<th>9</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>
Both parents have 2 in the same position, so that is fixed.

The 7–3 pair can be interchanged, with consequences for 5.

```
4 1 7 6 2 8 3 9 5 0
3 9 0 1 2 4 6 8 7 5
```

The consequences of the 4 in the first position of $C_1$ is:

```
4 1 0 6 2 8 3 9 7 5
3 9 7 1 2 4 6 8 5 0
```

This looks a lot like uniform crossover—but only certain swaps are allowed.
An Efficiency Challenge for PMX, OX, and CX

How can these operations be performed without resorting to multiple scans of the strings?

Try for $O(N)$ not $O(N^2)$ steps!
Representing Permutations

A permutation’s *signature* is a list of places.

\[ \{p_0, p_1, p_2, \cdots, p_{N-1}\} \]

Satisfying: \(0 \leq p_k < N - k\)

Meaning: “Put \(k\) in position \(p_k\)”

(There are exactly \(N!\) lists.)
Creating a Signature

for( i = 0; i < N; i++ ) {
    S[i] = random_int( N - i );
}

This procedure uniformly generates permutation signatures.
Manipulating Signatures

Preserve the rule: $0 \leq s_k < N - k - 1$

Mutate:
- Increment or decrement $s_k \mod N - k$
- Replace $s_k$ with random value $\mod N - k$

Crossover, as usual:
- One point.
- Two point.
- Uniform.
A Simple Signature Decoding Method

for( i = 0; i < N; i++ ) L[i] = i;

for( i = 0; i < N; i++ ) {
    interchange L[i] with L[i+Sig[i]]
}

We require that $L[i+Sig[i]]$ does not precede $L[i]$
i.e., that $0 \leq Sig[i] < N-i$

This decoding only costs $O(N)$. 
Signature Decoding Issues

- What is the cost to decode?
  - $\mathcal{O}(N)$? $\mathcal{O}(N \log N)$? Or $\mathcal{O}(N^2)$?

- What schemata are preserved?

- Is signature $\rightarrow$ permutation invertible?
Ordered Greedy Queens

Permutations Order Placement

The GA creatures are permutation of $(0, 1, \cdots, N - 1)$:

$$\text{individual } = (c_0, c_1, \cdots, c_{N-1}).$$

Interpret this as a placement ordering:

For $j = 0$ to $M - 1$

place the Queen in row $c_j$
in the left-most safe column.

Fitness: the number of successfully placed Queens.
Ordered Greed in Action

Permutation individual: 3 4 1 5 7 0 6 2

Row 3, col 0
Row 4, col 2
Row 1, col 1
Row 5, col 4
Row 7, col 3
Row 0, col 5
Row 6, col 7
Row 2, col 6

Fitness = 8 = 100%
Greed Ordered by 3 4 1 5 7 0 6 2

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fitnesses of 20,000 Random $64^2$ Boards

Successfully placed Queens’ histograms.

Note the efficacy of Ordered Greed vs. random placement.
Fitnesses of Some Random $256^2$ Boards

Note the efficacy of Ordered Greed vs. random placement.
Warnsdorff’s Knight Heuristic


They refer to: H. C. Warnsdorff
*Des Rösselsprunges einfachste und allgemeinste Lösung*, Schamlkalden, 1823.

A knight can tour the chess-board by visiting hardest-to-visit locations first.

A Hamiltonian graph path attempt should try low-degree vertices first. (Degrees decrease as neighbors are visited.)
Warnsdorff’s N Queens Heuristic

Place the hardest-to-place Queen next.

In case of tie, use our permutation.
Histograms: W/ & W/O Heuristics

256 Queens Fitnesses
Histograms: The 2 Heuristics

256 Queens Fitnesses

- 'ordered_greed'
- 'warnsdorf'

Graph showing the fitness values for 256 queens with two different heuristics.
Warnsdorff Uses 2 0 7 6 4 5 1 3

Row 2 has 8 chances
  Row 2, col 0
Row 0 has 6 chances
  Row 0, col 7
Row 7 has 5 chances
Row 6 has 4 chances
  Row 6, col 6
Row 7 has 4 chances
Row 4 has 2 chances
  Row 4, col 5
Row 7 has 3 chances
Row 5 has 1 chances
  Row 5, col 1
Row 7 has 1 chances
  Row 7, col 4
Row 1 has 1 chances
  Row 1, col 3
Row 3 has 1 chances
  Row 3, col 2

New hero indiv. #1 fitness = 8 = 100.00%
**Warnsdorff Uses 2 0 7 6 4 5 1 3**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Warnsdorff Uses 1 2 4 5 6 7 0 3

Row 1 has 8 chances
  Row 1, col 0
Row 2 has 6 chances
  Row 2, col 7
Row 4 has 4 chances
  Row 4, col 6
Row 5 has 3 chances
Row 6 has 2 chances
  Row 6, col 1
Row 5 has 1 chances
  Row 5, col 3
Row 7 has 1 chances
  Row 7, col 4

Indiv. #5 fitness = 6
Warnsdorff Uses 1 2 4 5 6 7 0 3

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>(3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rows 0 and 3 received no Queen.
GA Results

We used the following parameters:

\begin{verbatim}
uniform 1
POP_SIZE 50
LOOPS 1000
tournament_size 2
MUT_RATE 0.001
N 256
\end{verbatim}

We ran the algorithms 100 times: seed = 1.1, 2.2, \ldots, 100.100.
**GA Results, N = 256, Using OG**

We only allowed 1,000 loops!
The OG GA succeeded in 57% of the trials

<table>
<thead>
<tr>
<th>count</th>
<th>tries</th>
<th>fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>277</td>
<td>256</td>
</tr>
<tr>
<td>10</td>
<td>331</td>
<td>256</td>
</tr>
<tr>
<td>10</td>
<td>593</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>637</td>
<td>256</td>
</tr>
<tr>
<td>14</td>
<td>948</td>
<td>256</td>
</tr>
<tr>
<td>43</td>
<td>2050</td>
<td>255</td>
</tr>
</tbody>
</table>

Allowing up to 2,050 trials, random search succeeded in 16% of 100 trials.
## GA Results, $N = 256$

Ordered Greed with Warnsdorff  
(all solutions in initial population)

<table>
<thead>
<tr>
<th>count</th>
<th>tries</th>
<th>fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>5</td>
<td>256</td>
</tr>
<tr>
<td>14</td>
<td>10</td>
<td>256</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>256</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>256</td>
</tr>
<tr>
<td>27</td>
<td>23</td>
<td>256</td>
</tr>
<tr>
<td>10</td>
<td>25</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>26</td>
<td>256</td>
</tr>
</tbody>
</table>
Results, $N = 256$ (without OG)

Permutations (total failure)

<table>
<thead>
<tr>
<th>count</th>
<th>tries</th>
<th>fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>2050</td>
<td>178</td>
</tr>
<tr>
<td>10</td>
<td>2050</td>
<td>180</td>
</tr>
<tr>
<td>36</td>
<td>2050</td>
<td>181</td>
</tr>
<tr>
<td>9</td>
<td>2050</td>
<td>182</td>
</tr>
<tr>
<td>16</td>
<td>2050</td>
<td>185</td>
</tr>
<tr>
<td>15</td>
<td>2050</td>
<td>187</td>
</tr>
</tbody>
</table>
GA Results, N = 8

Ordered Greed
(all solutions in initial population)

<table>
<thead>
<tr>
<th>count</th>
<th>tries</th>
<th>fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>14</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>15</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>
GA Results, $N = 8$

Ordered Greed with Warnsdorff
(all solutions in initial population)

<table>
<thead>
<tr>
<th>count</th>
<th>tries</th>
<th>fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>22</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>14</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>14</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
GA Results, N = 8 (without OG)

Permutations (the GA worked 65%)

<table>
<thead>
<tr>
<th>count</th>
<th>tries</th>
<th>fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>54</td>
<td>8</td>
</tr>
<tr>
<td>15</td>
<td>58</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>144</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>190</td>
<td>8</td>
</tr>
<tr>
<td>14</td>
<td>302</td>
<td>8</td>
</tr>
<tr>
<td>35</td>
<td>2050</td>
<td>7</td>
</tr>
</tbody>
</table>

This is marginally better than random search.
Graph Coloring Problems

- \( G = (V_G, E_G) \) is a set of vertices, \( V_G \), and a set of edges, \( E_G \).

- An edge, \((v, w)\), connects the vertices \( v \) and \( w \). Vertices \( v \) and \( w \), are adjacent in \( G \).

- A coloring of \( G \) is a function \( c : V_G \to K \) (\( K \) is the set of colors) such that \((v, w) \in E_G \Rightarrow c(v) \neq c(w)\). \( G \) is \( k\)-colorable if it has a coloring: \(|K| = k\).

- The smallest \( k \) is the chromatic number of \( G \).
Graph Coloring

Famous graph coloring problem: map coloring.

Vertices are countries. Edges connect countries touch.

Touching countries must be colored differently.

Graph coloring also models:
exam scheduling, process scheduling, memory allocation, ...

Graph coloring is NP-complete: there is no known efficient algorithm. Fast approximations are desirable.

We color random 3-colorable graphs with edge density $p = 0.1$. 
Kubale’s Coloring Algorithm

Given: a permutation of $V_G$: $v_0, v_1, v_2, \cdots, v_{N-1}$

for ( $k = 0; k < N; k ++$ ) {
    Color $v_k$:
    use the smallest color number
    not assigned to a vertex adjacent to $v_k$.
}

Different permutations of $V_G$ can give different color assignment. The best coloring is clearly achievable!
Heuristics?

Warnsdorff may improve the performance:

First color the hardest-color vertices.

A vertex is hard to color if its neighbors use many colors.

Break ties with the vertex permutation.
Fitness is the number of vertices colored with three colors.
100,000 Tries to Color $G_{3000}$ by Kubale
Building Random Graphs $G_{100}$ and $G_{3000}$

Build a graph with $V_G = \{0, 1, 2, 3, \cdots, N - 1\}$.

for $(v = 0; \ v < \ N - 1; \ v \ + \ +)$ {
    for $(w = v + 1; \ w < \ N; \ w \ + \ +)$ {
        if $v \not\equiv w \pmod{3}$
            then $(v, w) \in E_G$ with probability $p$.
    }
}

Edge density is $p = 0.1$ for all 3-coloring experiments.
Why Density $p = 0.1$?

We did 10,000 coloring attempts with graphs of 100 vertices & edge densities of 0.01–0.40.

Small density graphs have many small connected components, thus easy to color.

Large density graphs have many triangles. An easy coloring strategy: locate and color the vertices in a triangle, then color the vertices in triangles with edges in common with a colored triangle.
Why \( p = 0.1 \)? The Experiments: \( p = 0.01 \cdots 0.40 \)

Minimum, mode, & maximum fitness.
Kubale attempts to 3-color 100-vertex graph.
10,000 fitness evaluations for each density
The Experiments

Use 3-colorable $G_{100}$ (100 vertices, edge density $p = 0.1$)

Vary the population size.

Vary the tournament size.

Color the 3-colorable $G_{3000}$. 
Ordered Greed: Fitness Evals for Pop Sizes 10–999
Tournament Sizes 2–99 to Color $G_{100}$

Average = 2,674. Population size = 200. Doesn't tournament size matter?
An OG Applications Sampler

O. G. is natural & appropriate for a variety of problems.

A necessary condition: The optimal solution can be found by a greedy algorithm and the right permutation.

Many permutations will produce the same, or equivalent, answer.
An OG Applications Sampler

- N Queens
- Graph vertex & edge coloring
- Scheduling in general
- Exam scheduling
- Sports tournaments scheduling
- Multiprocessors scheduling
- Faculty teaching assignments
- Job assignments
- Matching
- Traveling salesman
- Bin packing, & 2D board cutting
- Pentominoes
- SAT
A Latin Square: A New Problem to Solve with GA’s

1. Many N-rooks solutions.

2. A graph coloring.
Latin Squares are Experimental Designs

\[
\begin{array}{ccccccc}
0 & 2 & 1 & 6 & 5 & 4 & 3 \\
2 & 1 & 0 & 4 & 3 & 6 & 5 \\
1 & 0 & 2 & 5 & 6 & 3 & 4 \\
6 & 4 & 5 & 3 & 1 & 2 & 0 \\
5 & 3 & 6 & 1 & 4 & 0 & 2 \\
4 & 6 & 3 & 2 & 0 & 5 & 1 \\
3 & 5 & 4 & 0 & 2 & 1 & 6 \\
\end{array}
\]

An experimental design: perform the experiment using parameters \( \{x = p, y = q, z = L_{pq}\} \).

Only \( N^2 \) experiments, but every possible pair of settings of \((x, y), (x, z), \) and \((y, z)\) occur.
A Greco-Latin Square: Two Orthogonal Latin Squares

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

1. Every digit pair is unique. Each digit forms a Latin square.

2. Greco-Latin squares exist for every \( N > 2 \) except \( N = 6 \). (Euler conjectured that they didn’t exist when \( N = 4M + 2 \), but he was wrong.)
A Greco-Latin Square: Two Orthogonal Latin Squares

<table>
<thead>
<tr>
<th></th>
<th>0, 0</th>
<th>3, 2</th>
<th>6, 1</th>
<th>5, 6</th>
<th>4, 5</th>
<th>1, 4</th>
<th>2, 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5, 2</td>
<td>0, 1</td>
<td>2, 0</td>
<td>6, 4</td>
<td>3, 3</td>
<td>4, 6</td>
<td>1, 5</td>
<td></td>
</tr>
<tr>
<td>4, 1</td>
<td>1, 0</td>
<td>0, 2</td>
<td>3, 5</td>
<td>2, 6</td>
<td>6, 3</td>
<td>5, 4</td>
<td></td>
</tr>
<tr>
<td>3, 6</td>
<td>4, 4</td>
<td>5, 5</td>
<td>0, 3</td>
<td>1, 1</td>
<td>2, 2</td>
<td>6, 0</td>
<td></td>
</tr>
<tr>
<td>6, 5</td>
<td>5, 3</td>
<td>1, 6</td>
<td>2, 1</td>
<td>0, 4</td>
<td>3, 0</td>
<td>4, 2</td>
<td></td>
</tr>
<tr>
<td>2, 4</td>
<td>6, 6</td>
<td>4, 3</td>
<td>1, 2</td>
<td>5, 0</td>
<td>0, 5</td>
<td>3, 1</td>
<td></td>
</tr>
<tr>
<td>1, 3</td>
<td>2, 5</td>
<td>3, 4</td>
<td>4, 0</td>
<td>6, 2</td>
<td>5, 1</td>
<td>0, 6</td>
<td></td>
</tr>
</tbody>
</table>

An experimental design: perform the experiment using parameters \( \{ x = p, y = q, z = L_{pq1}, w = L_{pq2} \} \).

Only \( N^2 \) experiments, but every possible pair of settings of \((x, y)\), \((x, z)\), and \((y, z)\) occur.
Satisfiability: “SAT”

SAT: Is a given boolean function, $f(x_1, x_2, \cdots, x_n)$ satisfiable?

Variables $x_i$ are TRUE or FALSE.

$f$ combines the variables with AND, OR, NOT, and parentheses.

SAT asks whether the truth table for $f$ has any T’s.
If so, where?

This is an NP-complete problem (a standard one).
Standard Formula Forms

Disjunctive normal form (DNF): \( f(x_1, x_2, \cdots, x_n) = \bigwedge_i \bigvee_j y_{ij} \)

Conjunctive normal form (CNF): \( f(x_1, x_2, \cdots, x_n) = \bigvee_i \bigwedge_j y_{ij} \)

In both cases, \( y_{ij} \) denotes \( x_k \) or \( \neg x_k \) for some \( k \).
3-SAT

3-SAT deals with DNF formulas whose clauses have at most three terms.

\[ f(x_1, x_2, \cdots, x_n) = \bigwedge \bigvee_{i=1}^{3} y_{i,j} \]

For instance:

\[ f(x_1, x_2, \cdots, x_n) = (x_5 \lor \neg x_2 \lor x_10) \land (x_20 \lor \neg x_33 \lor \neg x_{17}) \land \cdots \land (x_{15} \lor x_{12} \lor \neg x_{37}) \]
3SAT Representation

We write the formulas like this:

\[ f(x_1, x_2, \cdots, x_n) = (x_5, \neg x_2, x_{10}), (x_{20}, \neg x_{33}, \neg x_{17}), \cdots, (x_{15}, x_{12}, \neg x_{37}) \]

Or, inside the computer, like this:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>-2</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>-33</td>
<td>17</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>15</td>
<td>12</td>
<td>-37</td>
</tr>
</tbody>
</table>
n Variables ⇒ n Bits have a Fitness

For a bitstring $B = (b_1, b_2, \ldots, b_n)$, let $x_k = b_k$ ($0 \Rightarrow F, 1 \Rightarrow T$).

The fitness of $B$ can be the number of true clauses.

To test your GA, find problems at “ORLib” (“operations research library”) at [http://www.ms.ic.ac.uk/info.html](http://www.ms.ic.ac.uk/info.html).
A Permutation of \( \{1, 2, \cdots, n\} \) Can Have a Fitness

For a permutation, \( P = (p_1, p_2, \cdots, p_n) \), determine truth values for the \( x_k \)'s as follows:

For each \( p_k \), in order
1. Find the first clause that uses \( x_{p_k} \) (that is, for which the value of \( x_{p_k} \) can make a difference), and set \( x_{p_k} \) to a value, \( v \), to make that clause true.

2. Find all clauses that use \( x_{p_k} \) and replace the reference of \( x_{p_k} \) to \( v \).

3. Mark all satisfied clauses so they don't have to be satisfied in the future (i.e., \( x_{p_k} = v \) satisfies them).

4. Remove the reference of “F” from the clauses that used \( x_{p_k} \) but \( v \) didn’t satisfy.
A Permutation of \( \{1, 2, \cdots, n\} \) Can Have a Fitness

As with bit strings, count the number of satisfied clauses.

3SAT, therefore, is another application of ordered greed.

Student problem: prove that any satisfiable 3SAT instance is satisfiable using some permutation. (This is a requirement we want to satisfy when we apply ordered greed.)
Notes on Computational Complexity

1. Every NP complete problem can be expressed as a satisfiability problem in polynomial time.

   (Try this for graph coloring.)

2. For $K \geq 4$, every $K$-SAT problem can be converted to a $K - 1$-SAT problem in polynomial time.

   (Students should prove this).

3. 2SAT is not NP complete.

   (Can you devise a polynomial-time algorithm?)
Traveling Salesman

This is another type of experiment.

I use my favorite programming language: $\mathcal{J}$, a new dialect of APL.
Traveling Salesman

Given \( N \) cities, determine the shortest circuit.

\( \mathcal{NP} \)-complete problem!

The greedy heuristic:
- Given a permutation of cities
- Build a circuit with the first three cities
- Add each city to the circuit least expensively

This depends on an ordering of the cities
- What are the first three?
- What order are the cities entered?

Use ordered greed!
The Approach

Gain more experience with the $J$ language.

Maintain a population matrix: each row is a permutation.

Implement OX.

Try a novel generation-making strategy.
Generation Making

Sort the population by fitness.

Kill off the worst half .... “on the average.”

Duplicate and scramble the population.

Circularly adjacent individuals create a child.

Slightly mutate the new population.
Test Plan

$N$ randomly chosen cities.

$N \times M$ cities in a grid ($N \cdot M$ must be even).

48 state capitals.
J Lists

x =. i.10
x
0 1 2 3 4 5 6 7 8 9
y =. 10?10
y
8 2 4 3 7 5 1 0 9 6
x+y
8 3 6 6 11 10 7 7 17 15
x<y
1 1 1 0 1 0 0 0 1 0
x=y
0 0 0 1 0 1 0 0 0 0
+/x=y
2
### Tables

\[ z \] = \begin{array}{c} \$ \end{array} \begin{array}{c} 5 \end{array} \begin{array}{c} 5 \end{array} \begin{array}{c} 2 \end{array} \]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
### J Functions

```
plus =. +
5 plus 6
11

add_one =. +&1
add_one 17
18

5^2

25

square =. ^&2
square 6
36
```
Idiom: Tilde

+~ 5
10
−~ 5
0
*~ 5
25
,~ 1 2 3 4 5
1 2 3 4 5 1 2 3 4 5
~~ 5
3125
5 −~ 20
15
5 %~ 20
4
J Idiom: Tilde

5 ? 5 NB. ? is deal
4 0 1 3 2
?~ 5
4 1 0 2 3
] x =. ?~ 5
2 0 4 1 3
x {~ 2 NB. { is the subscript operator
4
J Idiom: Masking

] list =. ?~ 10
8 2 4 3 7 5 1 0 9 6
<./list
0
list = <./list
0 0 0 0 0 0 0 1 0 0
i. 10
0 1 2 3 4 5 6 7 8 9
(list = <./list) # i. 10
7
# list
10
(list = <./list) # i. # list
7
My \( \mathcal{J} \) OG GA Program for TSP

The \( \mathcal{J} \) program determines a collection of \( N \) cities’ coordinates and their pairwise Euclidean differences.

It tries to create a short circuit.

It uses the heuristic:
- Build a circuit with the first three cities.
- Add each city to the circuit least expensively.

This depends on an ordering of the cities
- what are the first three?
- what order are the cities entered?
Program Components

Define several utility functions.

Define some GA functions.

Create cities list and distance matrix.

Run the GA with specified population size and number of generations.
A Few Utility Functions In $J$

Convert internal binary to printable ASCII:

```j
fmt =. "::
```

Print to standard output.

```j
print =. 1!:2 & 2
```

We can print inside functions using:

```j
print someValues
```
A Few Utility Functions In J

Write an array to a named file:

\[
\text{\texttt{WRITE =. 4 : 0}}
\]

\[
\begin{align*}
\text{NB. } & \text{x. is an array to be written.} \\
\text{NB. } & \text{y. is the file name.} \\
\text{linefeed} & =. 10 \{ a. \\
(, x. ,. \text{linefeed}) 1!:2 < y.
\end{align*}
\]

Call this function using:

\[
(\text{fmt values_array}) \text{WRITE } '\text{my_file}'
\]
Some TSP Functions

Creation of cities (here are two ways to do it).

30 random cities using the coordinates 0..59:
   cities =. (N,2) \$ ?~ 2*N =. 30

$N^2$ cities on an $N \times N$ grid:
   cities =. ,/"0/~ i. 10
   N =. #cities

Length of a vector is the square-root of sum of squares:
   norm =. %: @: +/ @: *:

Create the distance matrix, D:
   D =. norm "1 -"1 1/~ cities
Functions to Support the Heuristic

How much does it cost to insert city \(a\) into a circuit between cities \(a\) and \(b\)?

\[
delta = 4 : 0
deltax = (0 \{ x.)
delty = (1 \{ x.)
deltaz = y.
(deltax \{ deltaz \{ D) + (delty \{ deltaz \{ D) - (deltax \{ delty \{ D)
\]

Determine the best position to insert a new city into the circuit:

\[
\text{best_pos} = 3 : \'0 \{ (y. = <./y.) \# i. \# y.\'
\]
The Fitness Function

Convert an ordered list of cities into a path.

evaluate =. 3 : 0
  path =. 3 {. y. NB. head
  L    =. 3 }. y. NB. behead

  while. 0 < #L do.
    c =. {. L
    best =. best_pos (path ,. _1|. path) delta "1 0 c
    path =. (best {. path), c, (best }. path)
    L =. }. L
  end.

  path
)
Here is a J session:

```
] path =. ?~ 10
3 9 0 1 2 4 6 8 7 5
_1 |. path
 5 3 9 0 1 2 4 6 8 7
```
path, .. _1 |.. path
3 5
9 3
0 9
1 0
2 1
4 2
6 4
8 6
7 8
5 7
Note: Another Idiom

(path ,. _1| . path) delta "1 0 c =. {. L

path is a \( N - k \times 2 \) array.

L is the list of \( k \) unprocessed cities

c is the first element of L

delta is a function that takes two arguments:
   list delta node

list is a short list of two nodes
delta gets the cost of inserting node in the middle of list.
array delta "1 0 c

determines the vector of $N - k$ costs. "1 0 denotes that delta applies to the 1-cells on the left and the 0-cells on the right.
The Length of a Path

length = 3 : 0
    +/ (0 "1 s) "0 1 (1 "1 s =. y. ,. 1 |. y.) { D}
Ordered Crossover: “OX”

Create a child permutation whose prefix (of a randomly chosen length) is taken from one parent, and the remainder according to the ordering in the other parent.

OX =. 4 : 0
   NB. first, choose a random cut point
   pos =. 1 + ?1 -~ N

   rest =. pos }. x.

   (pos {. x.), (((_1 ((y. i. rest) ]) y.) = _1) # y.)} )
The GA Program

evolve =. 4 : 0
    P =. x.           NB. population size
    pop =. ?~ "0 P#N NB. Create population

NB. Compute all fitnesses, and print best 10
fitness =. length "1 evaluate "1 pop
print 10 {. /:~ fitness
'path_0' WRITE~ ": (, (0&{)) cities {~ evaluate 0 { pop
i =. 0

continued on next page...
while. y. > 0 do.
    pop =. pop /: fitness NB. Sort by fitness
    i =. 1 + i

    f_name =. 'path_', ": i
    f_name WRITE~ ": (, (0&{)) cities {~ evaluate 0 { pop

    NB. Delete worst half; replicate; shuffle
    pop =. (?~P) /:/~ ,~ (=:P) { pop

    NB. Crossover
    pop =. pop OX "1 1 (1 |. pop)
NB. Compute all fitnesses, and print best 10
fitness =. length "1 evaluate "1 pop
print 8.2 " : 10 {. /:~ fitness
y. =. y. - 1

end.
)

NB. Call the evolve function.
60 evolve 60
Apply the permutation-making function to the 0-cells:
   NB. Create population
   pop =. ?~ "0 P#N

Sort the elements of pop:
   NB. Sort by fitness
   pop =. pop /: fitness

Note the function composition:
   NB. Shuffle
   pop =. (?~P) /:~ ,~ (=:P) {. pop
The Code

fmt =. "":            NB. Convert to printable ASCII
print =. 1!:2 & 2
WRITE =. 3 : 0
:
   NB. x. is an array to be written.
   NB. y. is the file name character string.
   linefeed =. 10 { a.
   (, x. ,. linefeed) 1!:2 < y.
)

best_pos =. 3 : '0 { (y. = <./y.) # i. # y.'

cities =. ,/"0/~i. 10

cities =. (N,2) $ ?~ 2*N =. 100
[ N =. #cities

'cities' WRITE~ "": cities

norm =. %: @: +/ @: *: NB. Length of a vector

D =. norm "1 -"1 1/~ cities NB. Distance matrix

delta =. 3 : 0
  :
    a =. (0 { x.) [ b =. (1 { x.) [ c =. y.
    (a { c { D) + (b { c { D) - (a { b { D)
The Code

eval =. 3 : 0
  path =. 3 {. y.
  L =. 3 }. y.
  while. 0 < #L do.

    c =. {. L
    best =. best_pos (path ,. _1|. path) delta "1 0 c
    path =. (best {. path), c, (best }. path)

    L =. }. L
  end.
  path
)
length =. 3 : 0
   '+/ (0 {"1 s) {"0 1 (1 {"1 s =. y., 1 |. y.) { D'

OX =. 3 : 0 NB. Ordered crossover
:
   pos =. 1 + ?1 -~ N NB. random cut point
   rest =. pos }. x.
   (pos {. x.), (((_1 ((y. i. rest }))y.) = _1) # y.)
evolve =. 3 : 0
  :
  P =. x. NB. population size
  pop =. ?~ "0 P#N NB. Create population
  
  print 10 {. /:\: fit =. length "1 eval "1 pop
  'path_0' WRITE~ "(: (, (0&{)) cities {~ eval 0 { pop
  x =. 0
  
  while. y. > 0 do.
  pop =. pop /: fit NB. Sort by fitness
  x =. 1 + x
  ('path_', "x) WRITE~ "(: (, (0&{)) cities {~ eval 0{pop
NB. Delete worst half; replicate; shuffle
pop =. (?~P) /:~ ,~ (=:P) {. pop

pop =. pop OX "1 1 (1 |. pop) NB. Crossover

print 8.2 ": 10 {. /:~ fit =. length "1 eval "1 pop
y. =. y. - 1
end.
)
60 evolve 60
30 Random Cities
100 Random Cities
100 Cities in a Grid
48 State Capitals
48 State Capitals
Searching for Good Bit-strings

Bit-strings represent **solutions** to problems.

Bit-strings have **fitnesses**

How can we evaluate a large search space?
Systematic search
Random search
Genetic algorithm
Hill climbing
Simulated annealing
Tabu search
Hill Climbing

Assume you have a trial solution 
B = some bit string with fitness F.

Examine the neighborhood of B for high fitness strings.

A neighbor of B differs from B by a very small change.

The neighborhood is a small search space.

Problem: local optima
Hill climbing stalls at solutions with all lower-fitness neighbors, but not global optima.
Hill Climbing Pseudocode

```
for (k = 0; k < N; k++) {
    B' = B with k-th bit inverted
    F' = fitness of B'
    if (F' > F) {
        B = B'
        break out of loop
    }
}
```

In case we fail to “break” out of the loop we have failed to improve over B. We may be stuck at a local optimum!
Hill Climbing Code (Main Loop)

```c
fitness = fv(); // evaluate the fitness of p
for( trial = 0; trial < LOOPS; trial++ ) {
    k = random_int( N );
    p[k] = 1 - p[k];
    g = fv();

    if( g > fitness ) {
        fitness = g;       // accept new p
    } else {
        p[k] = 1 - p[k];   // reset to previous
    }
}
printf( "Best fitness: %d\n", fitness );
```
Hill Climbing Variations

- Test all neighbors and move to the greatest improvement.
- Test all neighbors and move to the least improvement.
- Modify the notion of neighbor.
- Use another “mutation” to find a neighbor to test.
Simulated Annealing (SA)

SA sometimes accepts moves to lower fitnesses.

SA uses a temperature

If the system is very hot then SA accepts almost every move

If the system is warm then SA tends to accept improvements

If the system is cold then SA mimics hill-climbing.

This attempts to avoid local optima.

But: do not cool too rapidly
Simulated Annealing

Choose a random neighbor $B'$ of $B$

Compute its fitness, $F'$

Accept the move to solution $B'$ with probability:

$$\frac{1}{1 + e^{-\frac{F' - F}{T}}}$$

$F' - F$ is the fitness improvement.

$T \geq 0$ is the system's temperature
SA Probabilities

\[
\Pr[\text{accept } B'] = \frac{1}{1 + e^{-\frac{F' - F}{T}}}
\]

\(\Pr[\text{accept } B']\) is near 1.0 if \(F' - F\) is large.

\(\Pr[\text{accept } B']\) is near 0.0 if \(F' - F\) is small.

The probability varies with temperature.
SA Probability: \[ \frac{1}{1 + e^{-\frac{F'}{T}}} \]
SA Pseudocode

Initialize $T$ to hot
while($T > \epsilon$) // $\epsilon > 0$ is small
{
    $B' = \text{a random neighbor of } B$
    $F' = \text{fitness of } B'$
    Determine probability $P$
    $B = B'$ with probability $P$
    if($F' = \text{goal fitness}$)
        break out of loop
    Reduce $T \to 0$ // slowly
}
Temperature Reduction

“Reduce $T$” very slowly.

\[ T \leftarrow T \cdot 0.999 \]

\[ T \leftarrow T - 0.0001 \]
SA Code (Main Loop)

```c
fitness = fv(); // see "hill climbing" code, above
for( trial = 0; (hero < hero_goal) && (trial < LOOPS); trial++ ) {
    T = T * temp_decay;
    k = random_int( N );
    p[k] = 1 - p[k];
    g = fv();

    if( drand48() < 1.0 / ( 1.0 + exp( -(g - fitness)/ T ) ) ) {
        fitness = g;
    } else {
        p[k] = 1 - p[k];
    }
}
printf( "Results are: ....... ");
```
As in hill climbing and SA, work with one individual.

Iteratively, test every neighbor, and move to the best one, even if that means going “downhill.”

However, do not change a bit that was changed “recently.”

However, you may change a recently changed bit in case that moves you to a new champion.