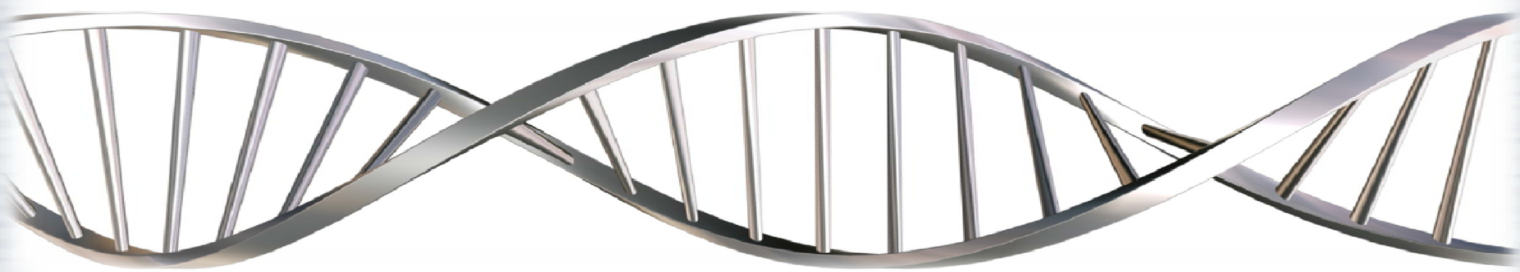
instantiations

Implementing
Genetic Algorithms efficiently
in Smalltalk



Bob Whitefield
ModelDesign Corporation

modeldesign

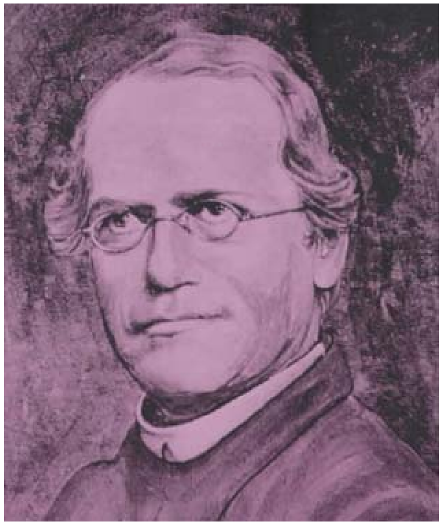
VASmalltalk 2009 SUMMIT

What are genetic algorithms?

- Problem-solving technique based on the principles of natural selection
- Part of the rapidly-growing field of *evolutionary computation*
- General-purpose method: does not require algorithms specific to the problem
- Creates a population of potential solutions, selecting, combining and mutating them to create successively better ones
- Capable of finding near-optimal solutions to otherwise intractable problems

A brief history of genetics

- The father of genetic theory was Gregor Mendel, an Austrian monk
- Mendel was the first to quantify how traits are inherited, through years of carefully breeding pea plants
- His 1865 monograph *Experiments with Plant Hybrids* showed traits do not blend, but pass intact to offspring

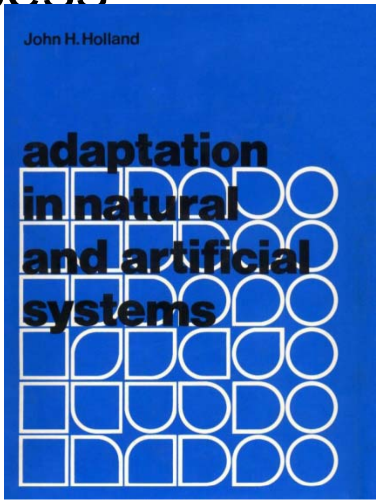


Gregor Mendel
1823-1884

- The significance of Mendel's work was not recognized in his lifetime, but the discrete nature of genetics later filled major gaps in Darwinian theory

A brief history of genetic algorithms

- Alan Turing proposed using evolutionary techniques for machine learning in his landmark 1950 paper *Computing Machinery and Intelligence*
- Beginning in the late 1950s, researchers attempted to simulate evolution using computers, with varying degrees of success
- John Holland's 1975 book *Adaptation in Natural and Artificial Systems* provided the first theoretical basis for how genetic algorithms work
- Today, GAs are a major segment of the evolutionary computation field



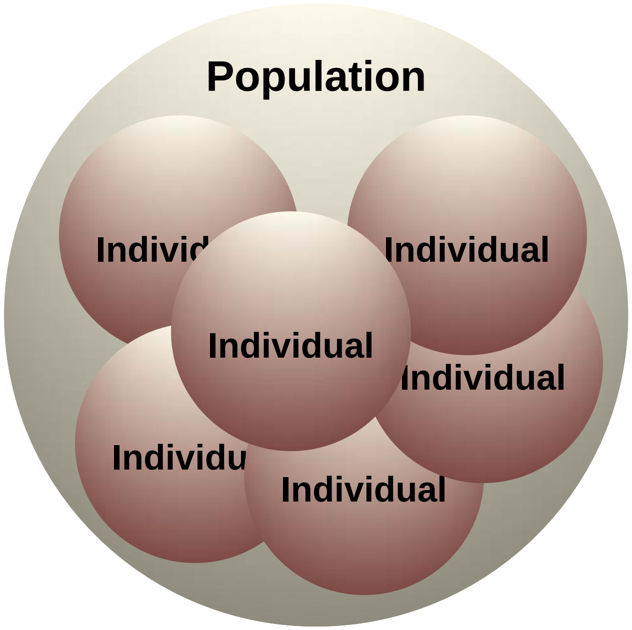
Genetic terminology

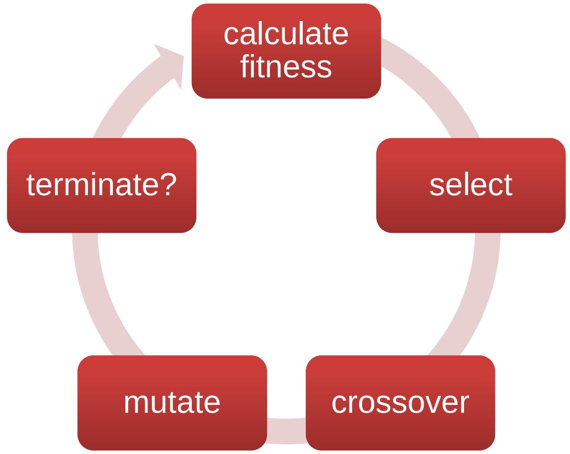
Structures	
individual	A potential solution to a problem
gene	A trait or characteristic of an individual
genome	All genes that collectively define an individual
population	A collection of individuals having the same genome
allele	The value of a particular gene
chromosome	All alleles for a specific individual
locus	Location of an allele in the chromosome

Operations	
fitness	A number measuring the quality of each individual
selection	Choosing individuals to mate (or die) based on their fitness
crossover	Combining chromosomes of two individuals to create offspring
mutation	Randomly changing an allele

How do genetic algorithms work?

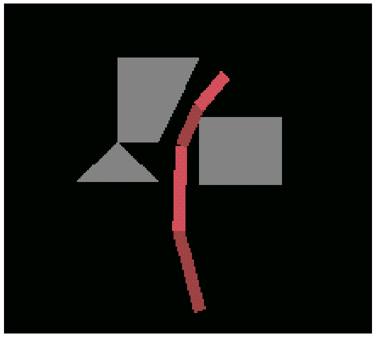
- A population of random individuals is created
- The *fitness* of each population member is calculated
- Individuals with higher fitness are *selected* to mate; less-fit ones are replaced
- *Crossover* combines alleles from two parents to create offspring
- *Mutation* is introduced to ensure genetic diversity
- The process continues until desired fitness is achieved, a time limit is reached, or improvement stops
- The most fit individual is the solution



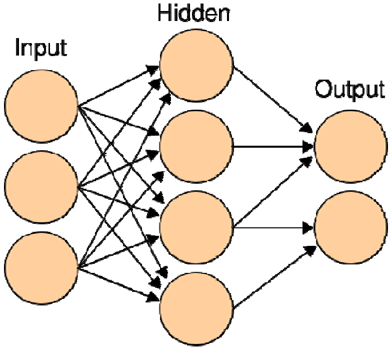


Applying genetic algorithms

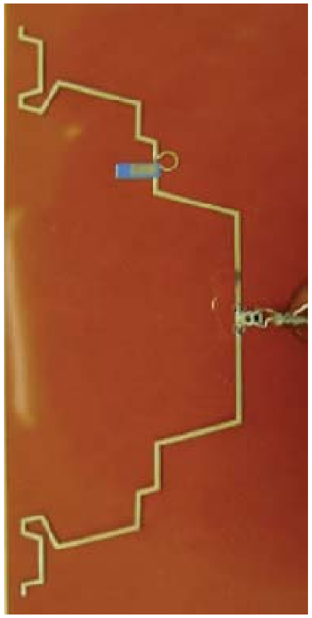
- Some examples of how GAs have been used
  - Optimization
    - Vehicle routing
    - Scheduling
    - Robot trajectory planning
  - Design
    - Aircraft wing design
    - Antenna design
    - Engine turbine blade design
  - Machine learning
    - Data mining
    - Neural network training
    - Stock trading and portfolio management



GA-generated robot trajectory



3-4-2 neural network



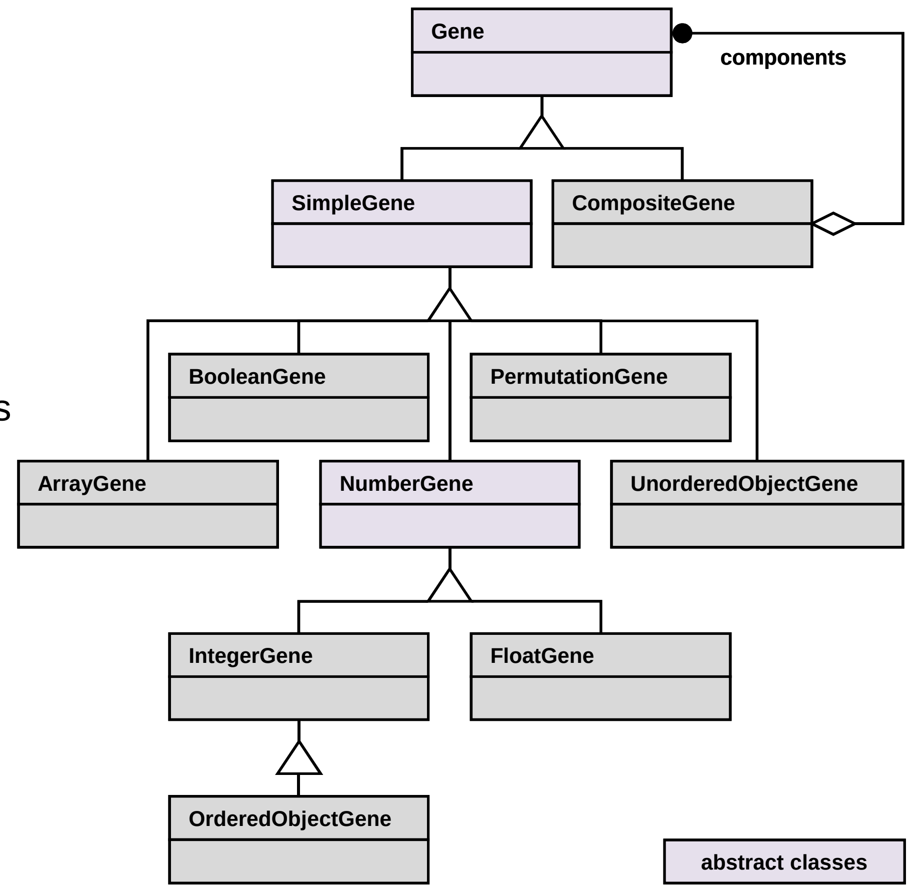
GA-designed GPS antenna

Goals of the Mendel framework

- Simple
  - Make applying genetic algorithms as easy as possible for the developer, requiring minimal code development and understanding of GAs
- Flexible
  - Provide a comprehensive set of gene types that can be easily reused to handle most problems
- Efficient
  - Provide a fast implementation competitive with other commercial products

Gene types

- Mendel provides a variety of gene types for encoding problems
  - Boolean, float, integer
  - Ordered and unordered object lists
  - Arrays and permutations
  - Grouping and grammar genes
- Optimized operators for each type
  - Float, integer, and ordered list genes use blending algorithm
  - Permutation genes use either *Enhanced Edge Recombination* or *Random Key* algorithms
- Hierarchical gene representation
  - Composite gene allows any number of nesting levels
  - Nested genes have private name scope



Mendel design features

- **Steady-state algorithm**
  - Continuous evolution helps avoid premature convergence
- **Elite management**
  - Ensures best solution is never lost
- **Tournament selection**
  - Chooses parents and deceased from small group of random individuals
- **Multiple crossover techniques**
  - N-point – cuts genome at specified number of points
  - Uniform – randomly swaps alleles on per-gene basis with bias parameter
  - Gene-specific
    - Radcliff's blending algorithm
    - Whitley's *Enhanced Edge Recombination* algorithm
    - *Random Key* encoding

How to solve a problem

1. Create a variableSubclass of *MendelIndividual*

```
MendelIndividual
   variableSubclass: #TravelingSalesman
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
```

2. Declare gene(s) in class method *genomeDefinition*

```
genomeDefinition
   ^super genomeDefinition
      at: #cities
      put: (MendelGene permutationOf: self cities)
```

3. Define instance method *fitnessFunction*

```
fitnessFunction
   "Iterate over the permuted city coordinates,
   summing the distance between each pair.
   Answer the result."
```

4. Set termination criteria and any other optional parameters

```
populationParameters
   ^super populationParameters
      memberCount: 2500;
      mutationProbability: 0.01;
      mutationMagnitude: 0.1;
      yourself
```

5. Run the algorithm and retrieve results

```
TravelingSalesman solve
```

Example: Traveling Salesman

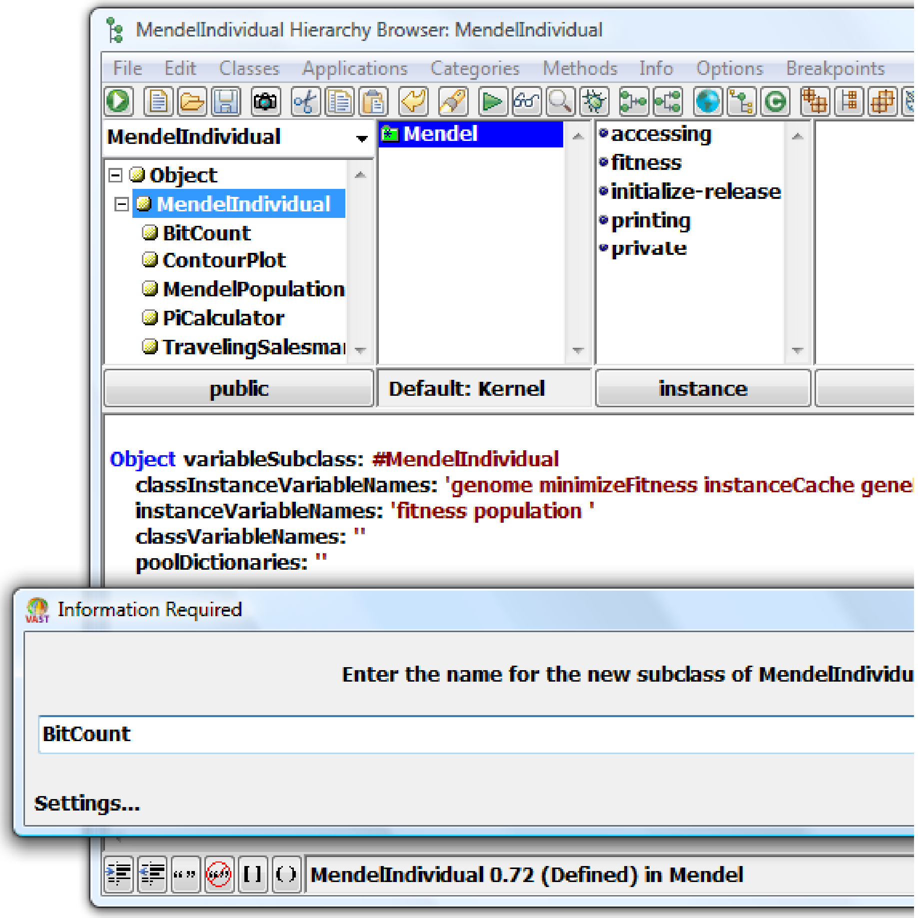
- Here's a simple search problem:
  - I have an array of 100 bits, zeroes and ones:

11010100001011011110101011001011001010001011011110101010010100100101101001011110010100111010011010

  - You must guess the value of every bit
  - The array is hidden from you, but if you give me a guess, I'll tell you how many are correct
- The following slides illustrate how to use Mendel to solve this problem
- To simplify the example, the hidden bit pattern is all ones—but you don't know that

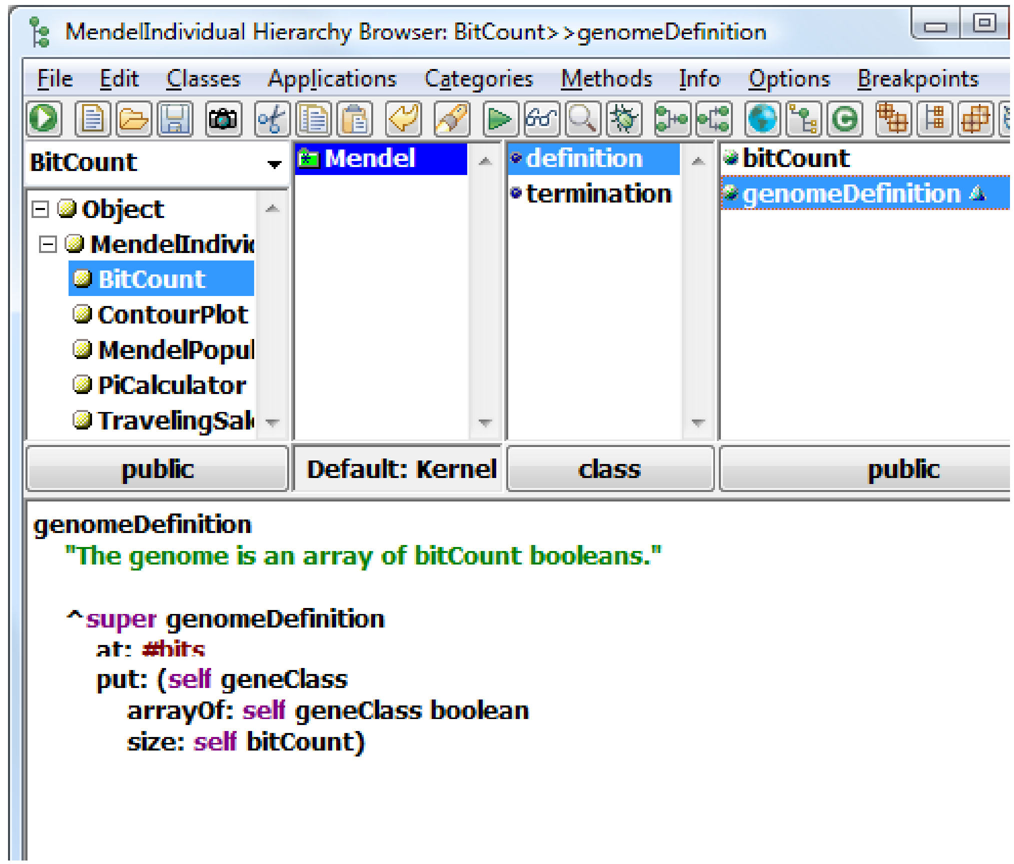
Step 1: Create problem class

- The first step in using Mendel is to create a Smalltalk class representing your problem
- It must be a variable subclass of *MendelIndividual*; the indexed slots hold the chromosome
- Gene structure, defined in the next step, determines chromosome size and layout
- Therefore, no need to define instance variables



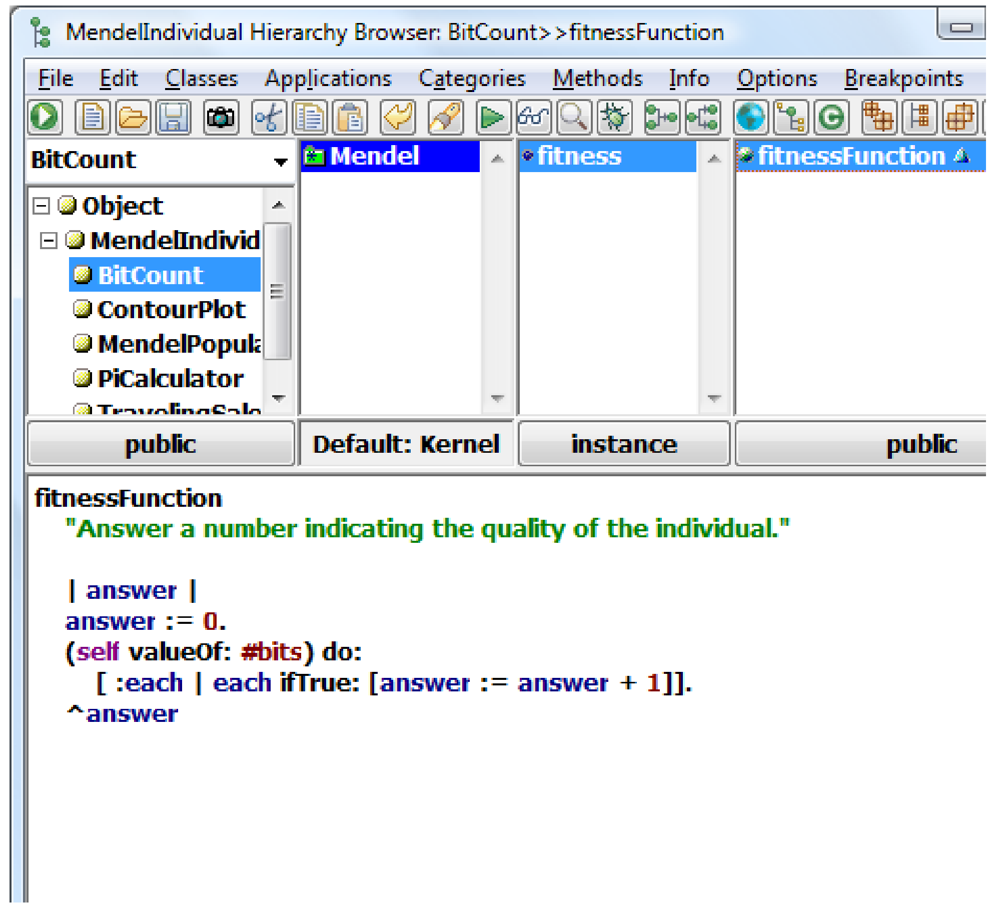
Step 2: Define genes

- In your new class, create a class method called *genomeDefinition*
- The method answers an instance of *MendelGene* describing the individual's genome
- Call the *super* version of *genomeDefinition* to get an empty composite gene
- Add named genes to the composite using *at:put:*
- Method *geneClass* answers the MendelGene class



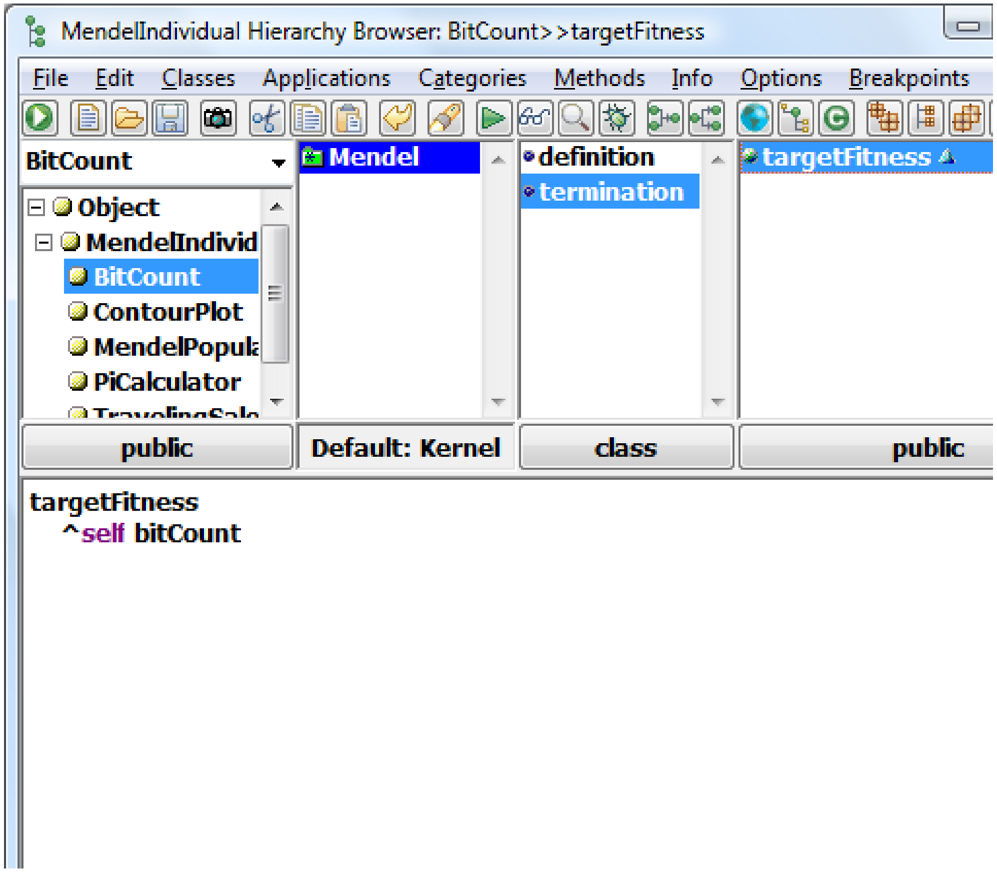
Step 3: Write fitness function

- Create an instance method called *fitnessFunction*
- It answers a number indicating the relative quality of the individual, calculated from its gene values (alleles)
- Use *valueOf:* or *alleleAt:* to access alleles by their symbolic name
- The fitness result can be any kind of *Number*
- In this example, it simply answers the number of true values in the *bits* array



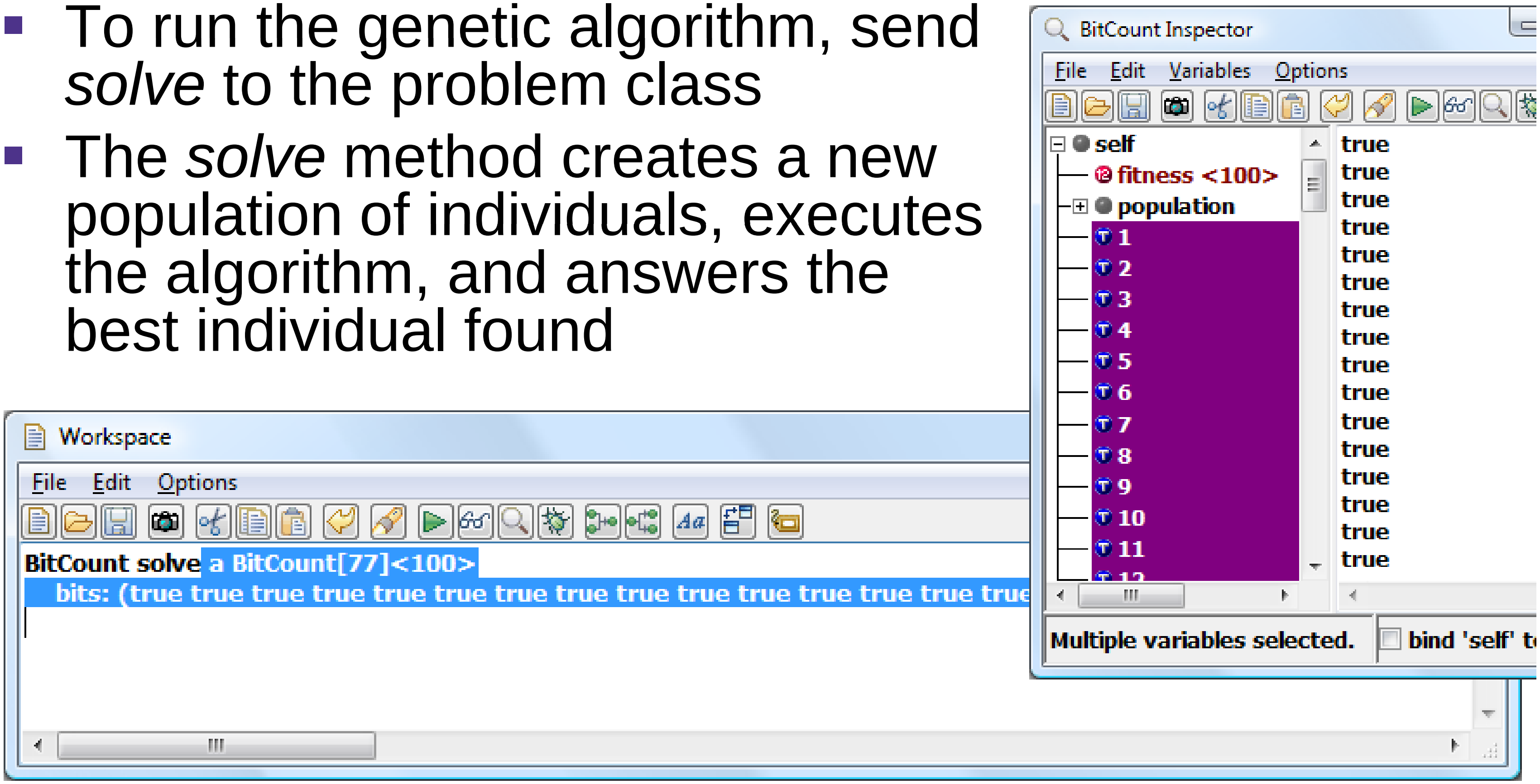
Step 4: Set termination criteria

- Mendel lets you define several kinds of termination criteria; by default, it terminates after five minutes without improvement
- But in the bit counting example, we know the exact target fitness: a perfect score is 100
- So override class method *targetFitness*
- The algorithm will now terminate when any individual achieves this fitness value



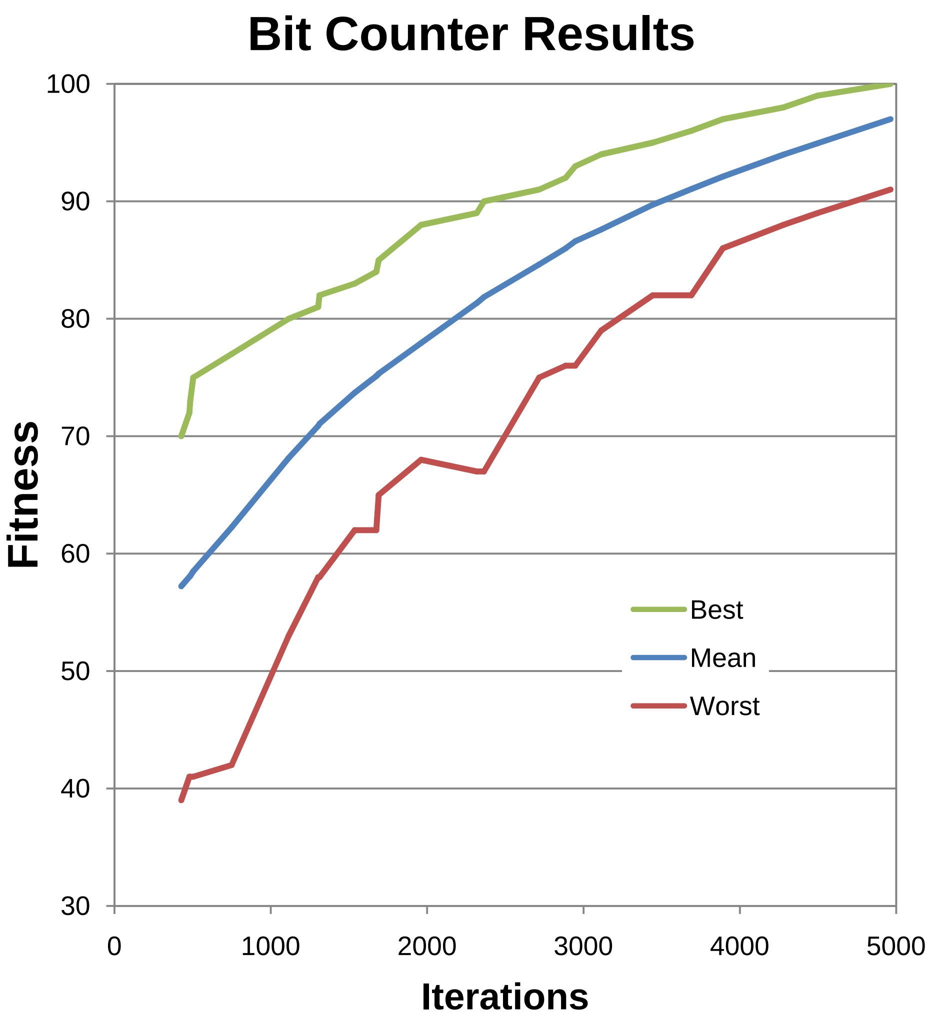
Step 5: Run it

- To run the genetic algorithm, send *solve* to the problem class
- The *solve* method creates a new population of individuals, executes the algorithm, and answers the best individual found



Bit counter results

- Correct result found after 5000 iterations and 10,000 evaluations
- Elapsed time: 2.7 seconds

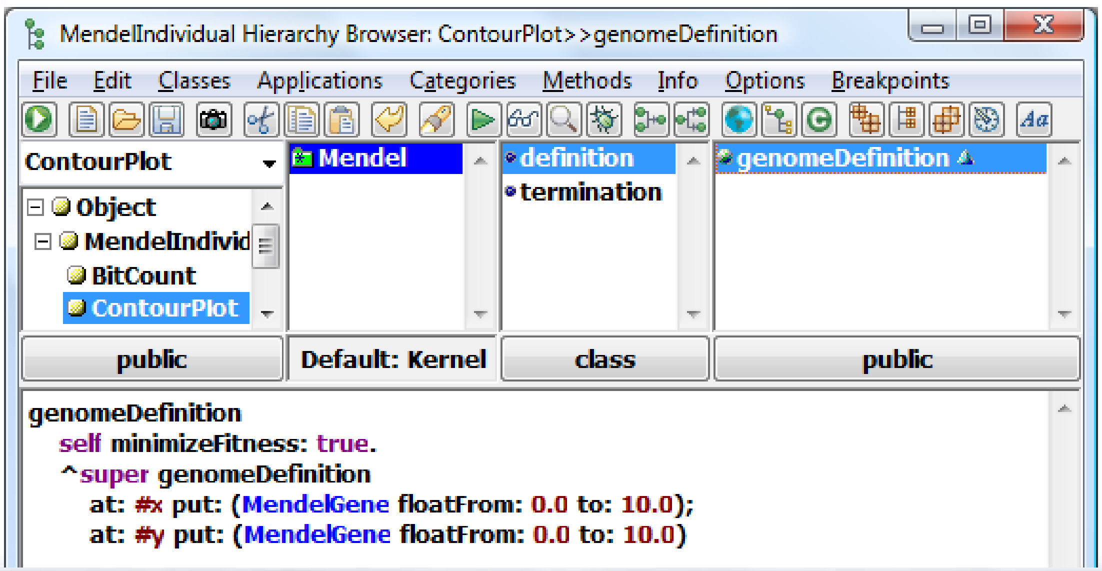


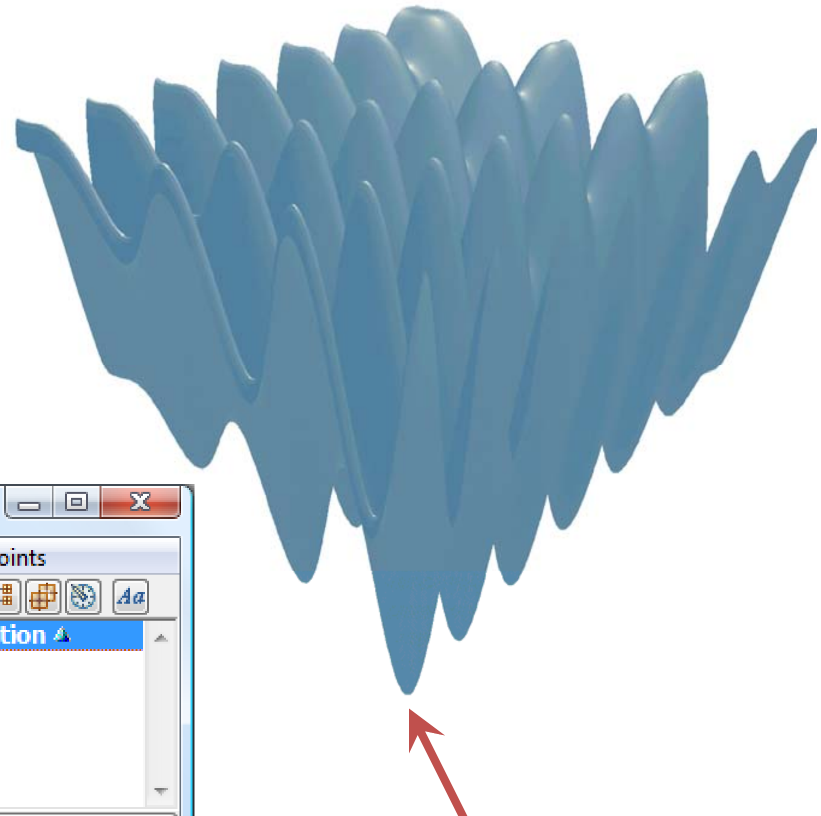
Example: Contour plot

- Find the minimum of this very bumpy function:

$$f(x, y) = x \sin(4x) + 1.1\ y \sin(2y)$$
$$0 \le x \le 10,\ 0 \le y \le 10$$

- Create a class called *ContourPlot* and define the genome:



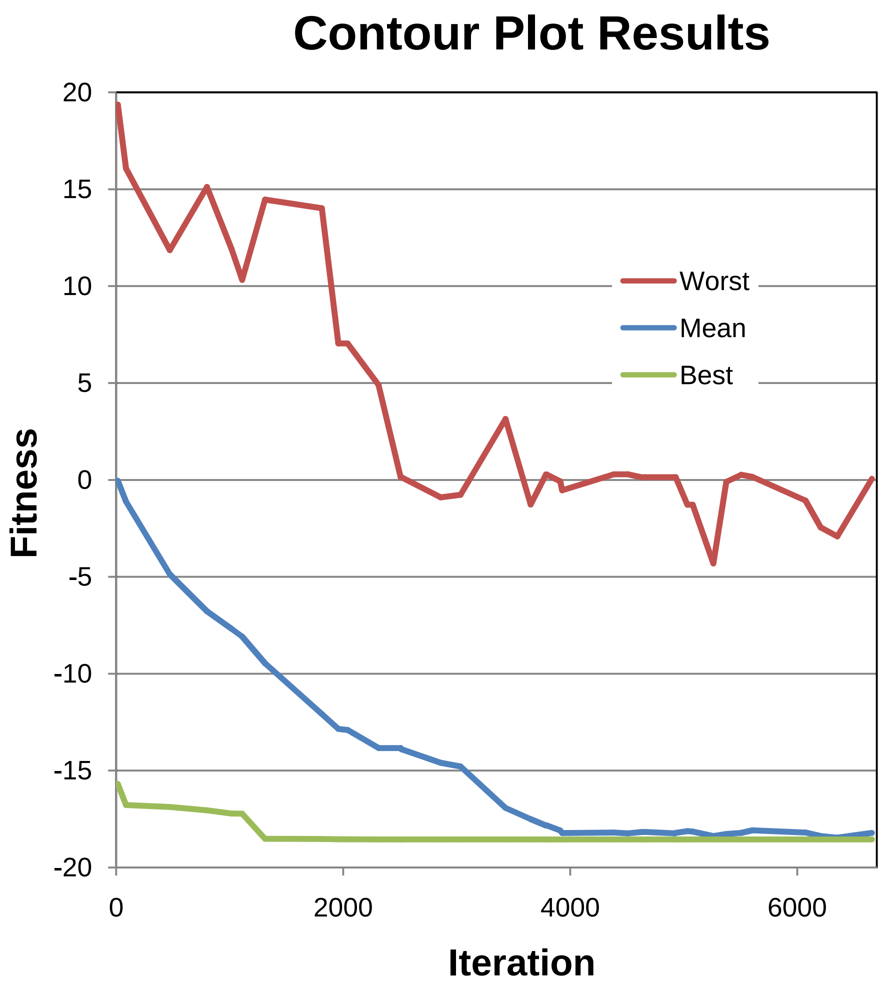


$x = 9.03899160$
$y = 8.66818896$

$f(x, y) = -18.5547210773827$

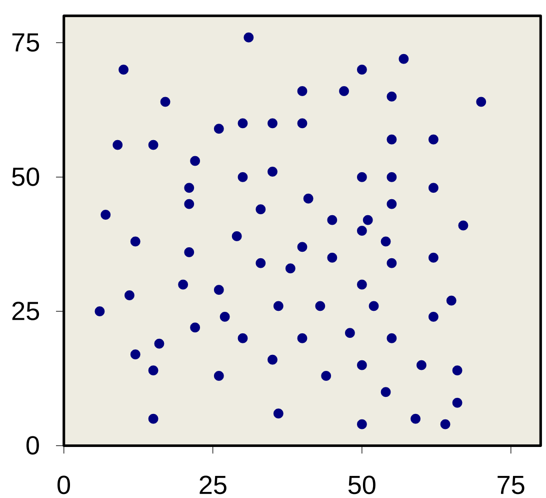
Contour plot results

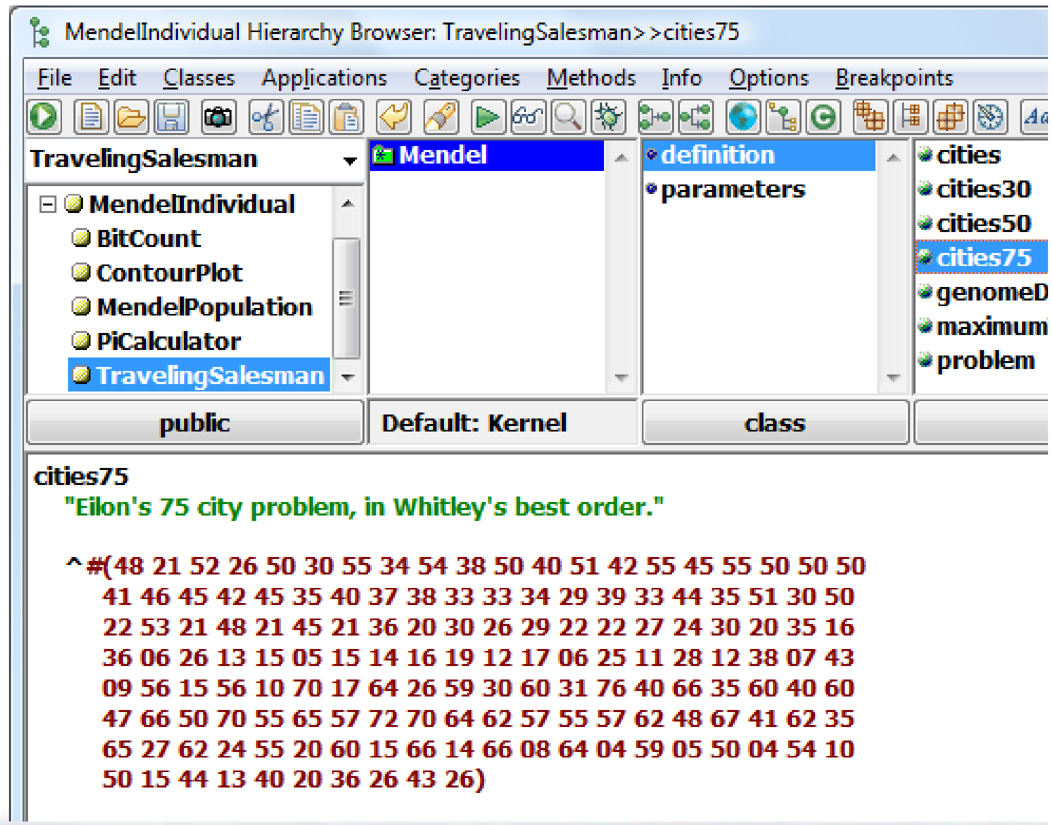
- Best member of the initial population had fitness of -15.7
- Required 6700 iterations, 14000 function evaluations, and 700 mutations to reach 15 significant figures of accuracy in *f(x,y)*
- Elapsed time: 270 ms



Example: Traveling salesman

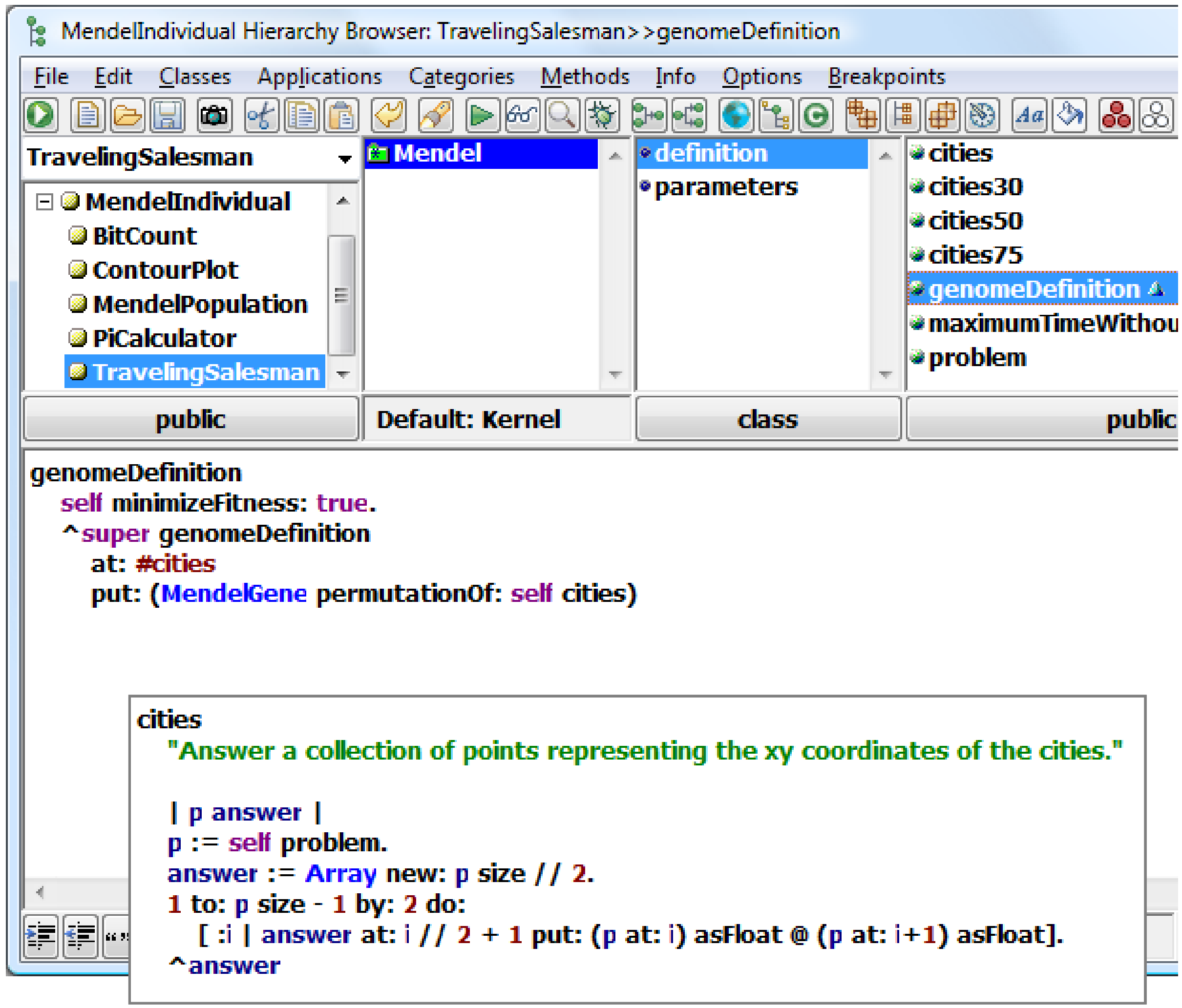
- Given a set of cities, find the shortest path that visits each city once and only once, and returns to the starting city
- A well-known benchmark is Eilon’s 75-city tour, consisting of 75 integer coordinate pairs





Traveling salesman genome

- The genome consists of a single permutation gene containing each of the city coordinate points
- Supporting methods add some flexibility for experimenting with different data sets
- Convert city coordinates to float now to prevent repeated conversions during the run

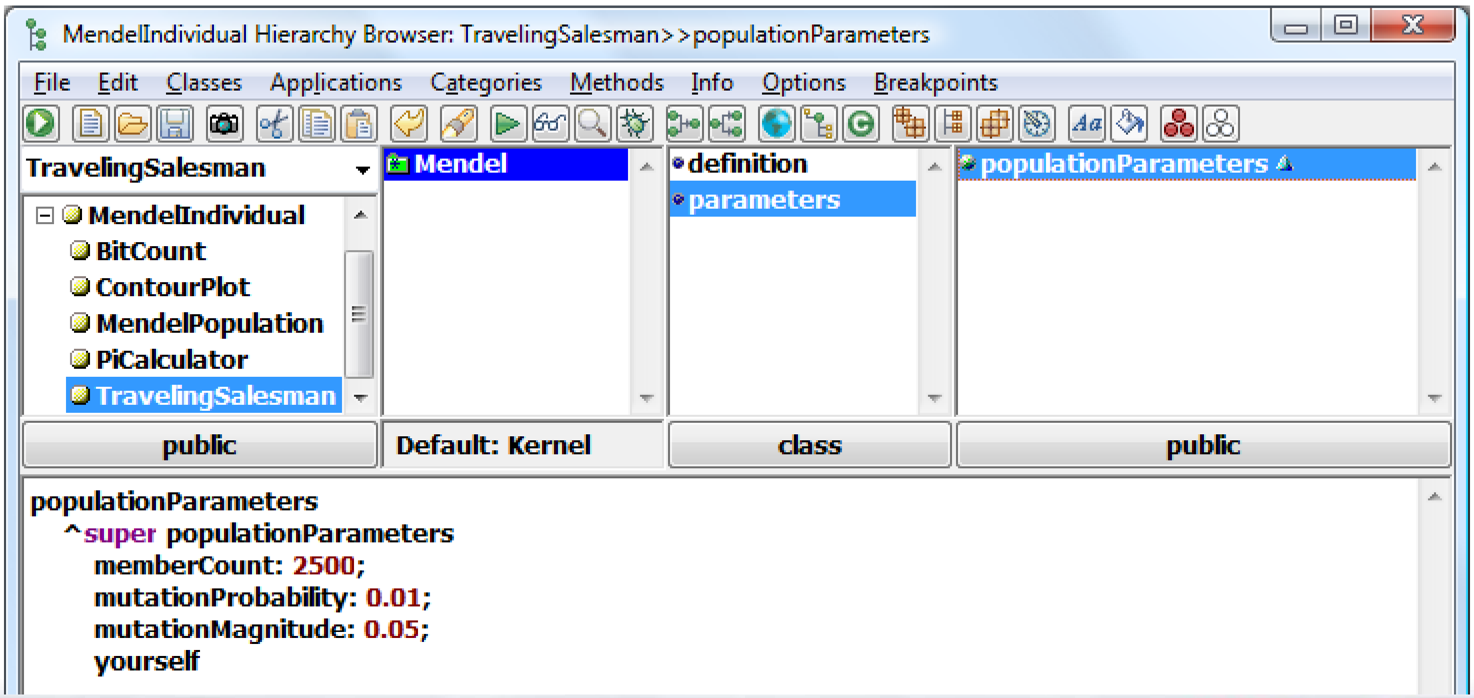


Mendel gene types

Type	MendelGene creation protocol	Description
Array	array: *size* of: *aGeneClass*	Arrays of a simple gene type, accessed by integer index
Boolean	boolean	True/false values
Composite	composite	Groups of related genes and/or other composites in their own name scope
Float	float floatFrom: *start* to: *end*	Any floating-point value Float within the specified range
Grouping	groupingOf: aCollection	Grouping of elements of aCollection
Integer	integer integerFrom: *start* to: *end*	Any integer value Integer within the specified range
Ordered Object	orderedList: *aCollection*	Object chosen from *aCollection*, whose elements have a natural ordering
Permutation	permutedList: *aCollection*	All elements of *aCollection*, permuted into a particular order
Unordered Object	unorderedList: *aCollection*	Object chosen from *aCollection*, whose elements have no natural ordering

Traveling salesman parameters

- Because this problem is more difficult and different than the previous examples, we need to tweak some parameters of the algorithm
- Override *populationParameters* class method to:
  - Increase population size from default of 500 to 2500
  - Make mutation rate and magnitude smaller than the default values of 0.1

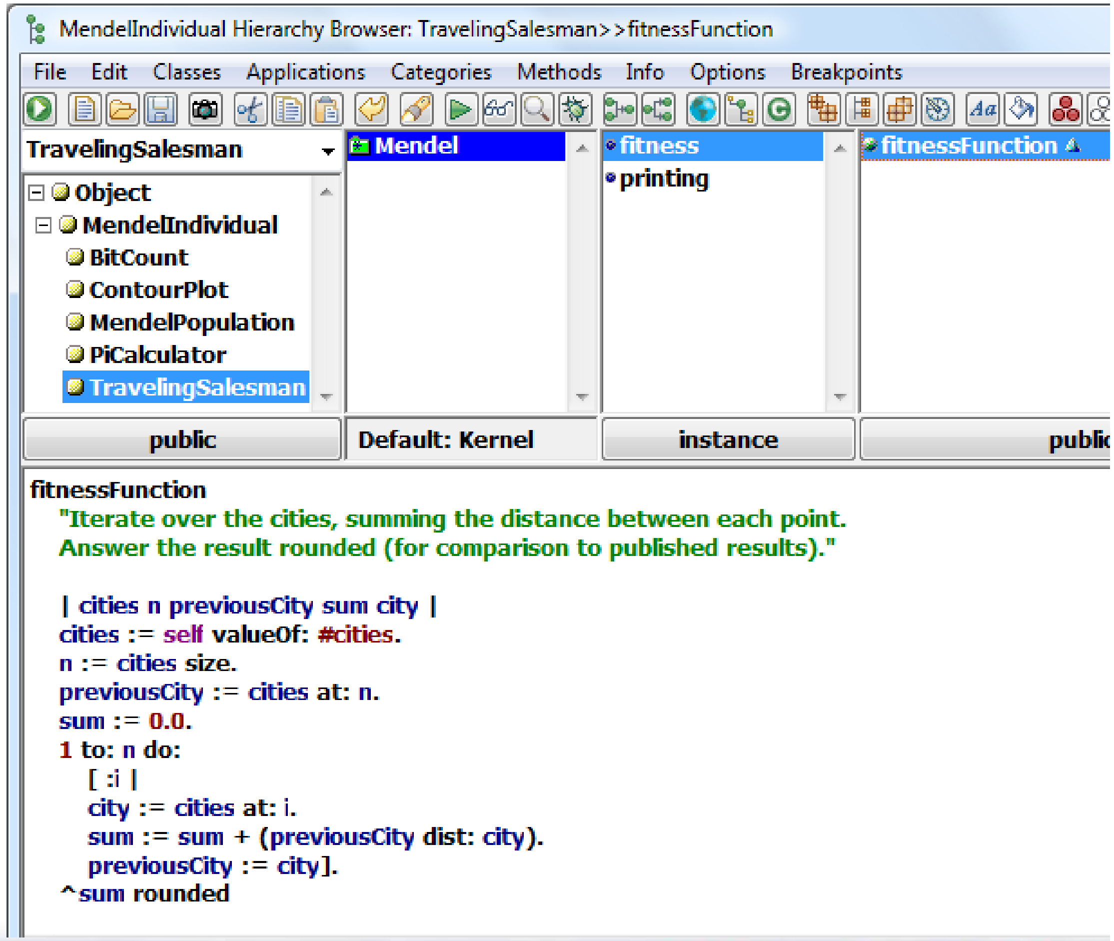


Population parameters

Parameter	Method name	Description	Default
Crossover Points	crossoverPointCount	Number of points at which parent chromosomes are cut during crossover, 0 = uniform crossover	2
Death Selection Pressure	deceased1Pressure deceased2Pressure	Number of individuals to compare against when choosing a deceased	4 4
Mutation Magnitude	mutationMagnitude	Maximum fraction of value range added or subtracted from a mutated allele	0.1
Mutation Probability	mutationProbability	Probability that a mutation will occur in a given iteration	0.1
Parent Selection Pressure	parent1Pressure parent2Pressure	Number of individuals to compare against when selecting parents to mate	4 0
Population Size	memberCount	Number of individuals in the population	500
Random Number Seeding	randomize: *aBoolean*	True if random number generator should be seeded from the clock	true

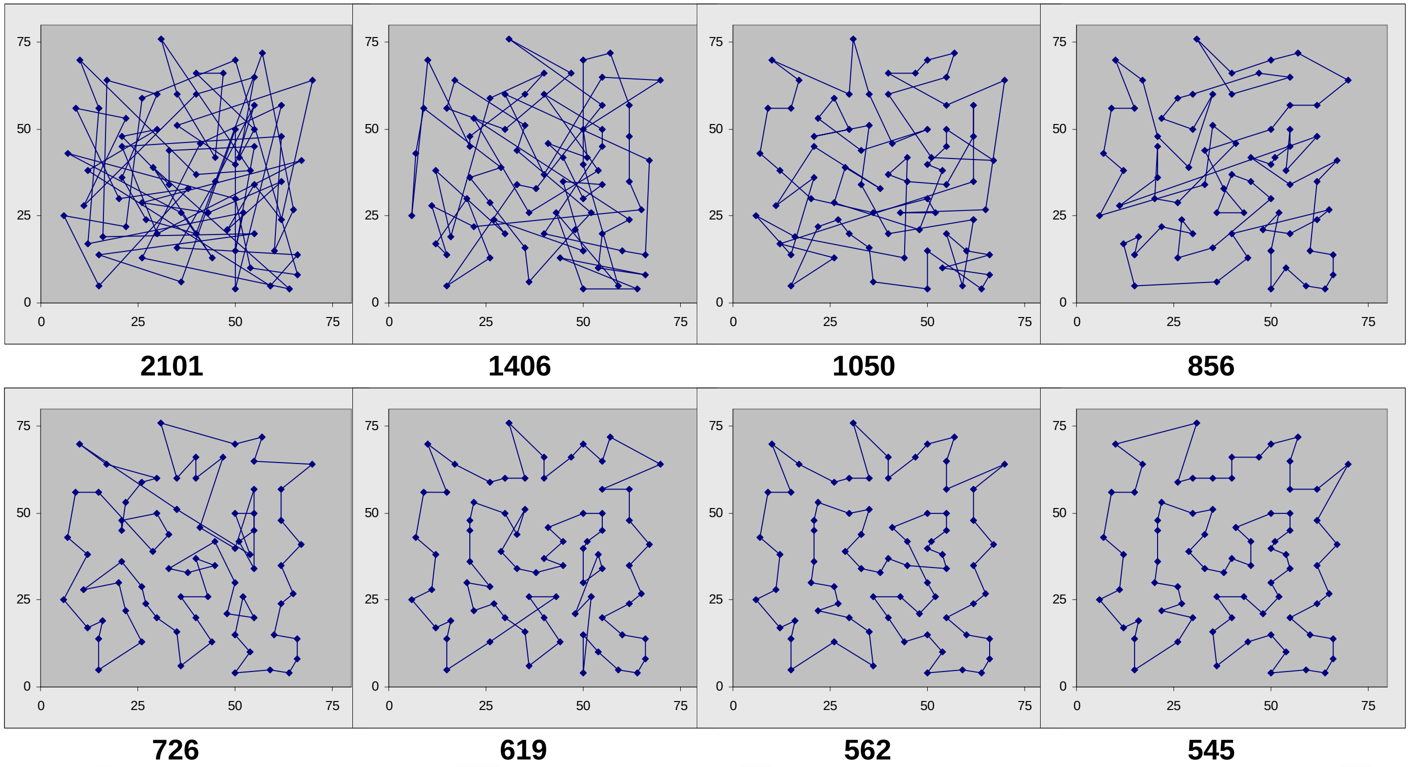
Traveling salesman fitness function

- The result of the fitness function is simply the total distance traveled
- Fetch the permuted cities, calculate the distance between each and sum them
- Answer the sum rounded to an integer



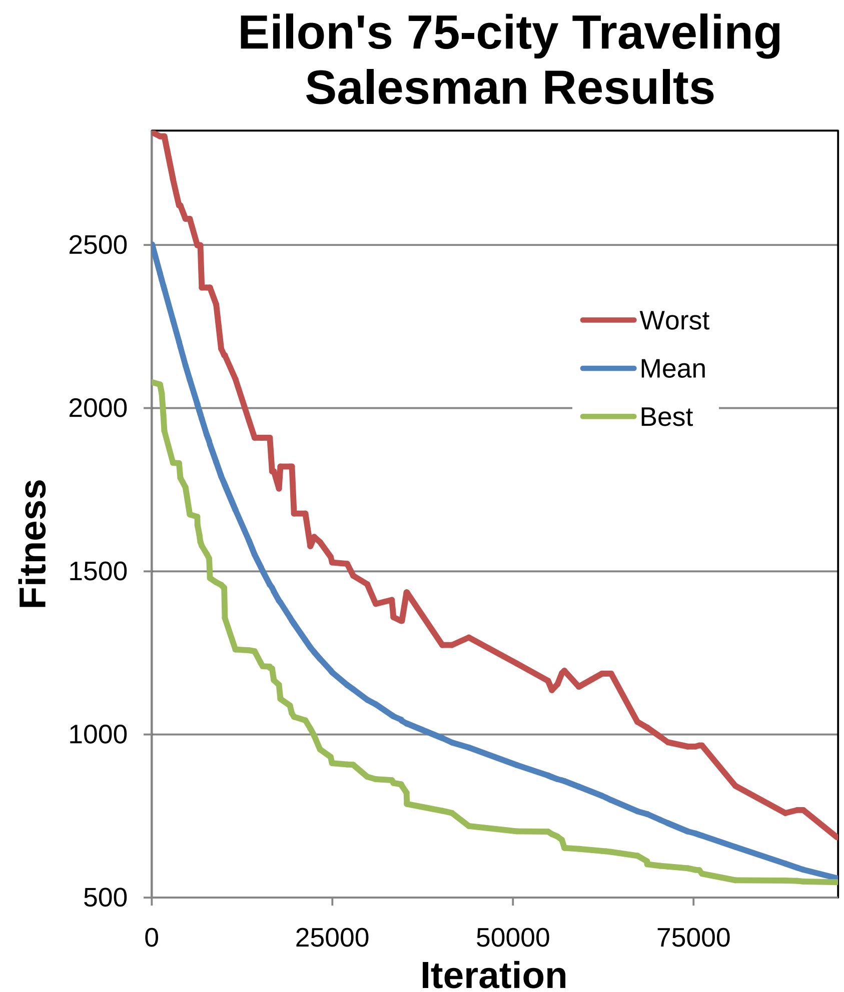
Traveling salesman progress

- Snapshots of Mendel's progress, showing the best individual and its fitness at various points in the run:



Traveling salesman results

- Best individual in the initial population had a fitness of 2101
- After 95,000 iterations, 190,000 function evaluations, and 935 mutations, the shortest known distance was found: 545
- Elapsed time: 41 seconds

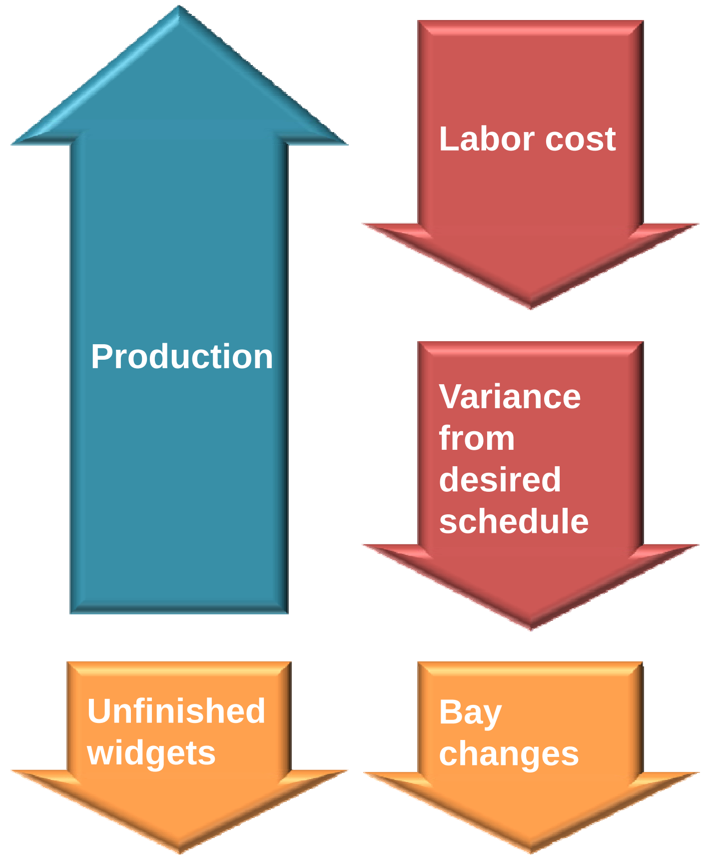


Example: Acme Widget production

- The Acme Widget Company has three product lines
- Their facility has 8 production bays, each dedicated to a product line
- Each product is constructed in three phases:
  - Setup
  - Assembly
  - Test and calibrate
- Each phase has specific skill requirements, provided by one or more employees
- Each employee is classified as to skill:
  - Apprentice
  - Journeyman
  - Master

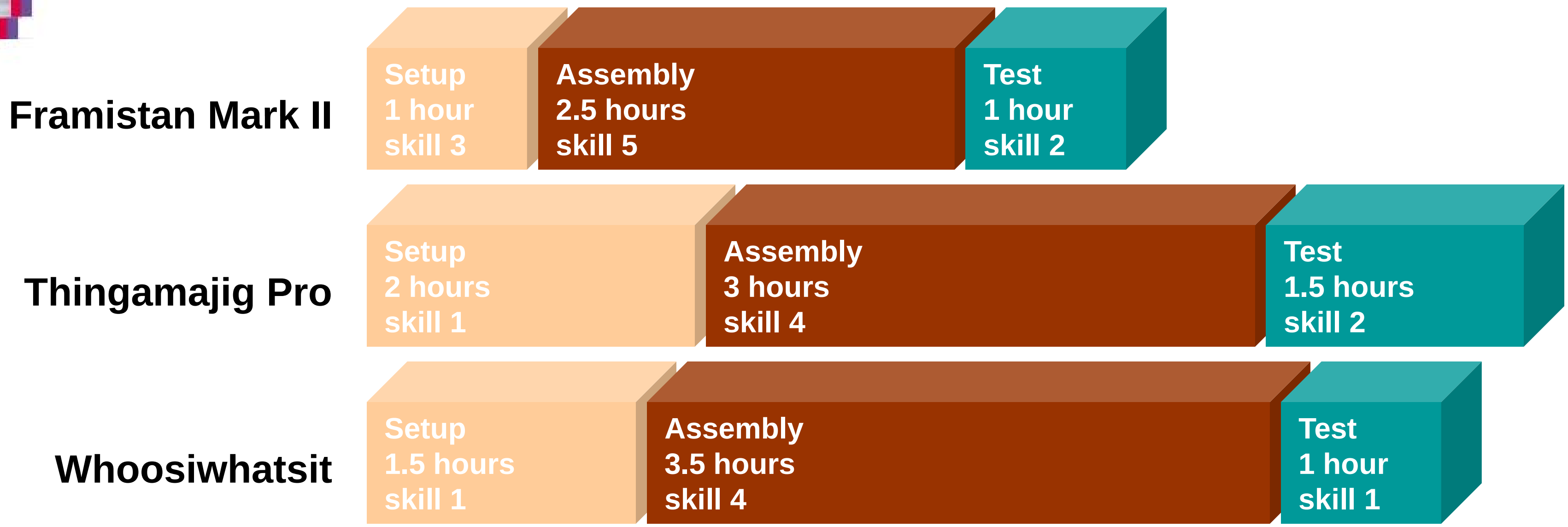
Acme's scheduling problem

- The task is to create an *employee production schedule* that maximizes the number of widgets produced while minimizing labor cost
- Another important goal is to let employees specify their desired work hours as much as possible
- Lesser goals include:
  - Minimize number of unfinished widgets at the end of the week
  - Minimize how often employees must change production bays



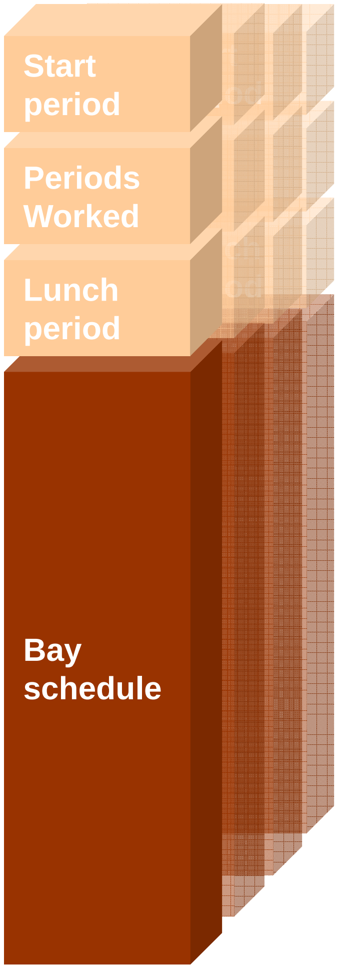
Acme production phases

- Each of the three production phases is allocated a specific amount of time for its completion
- The sum of skill levels for all employees assigned to a phase must meet a minimum skill total



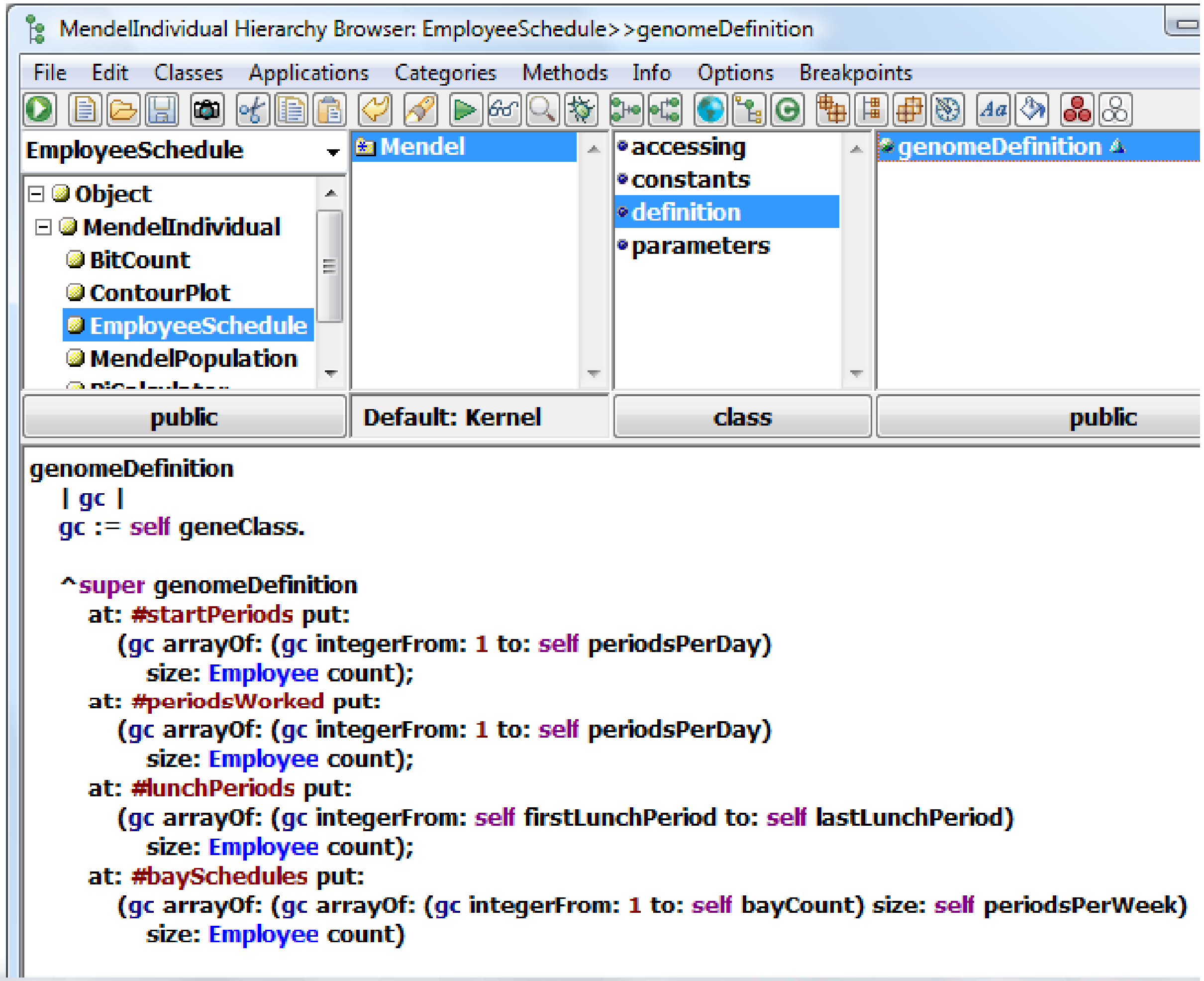
Encoding Acme schedules

- For each employee, we want to encode a one-week schedule specifying:
  - Punch in/out and lunch times for the week
  - An array of bay numbers to which the employee is assigned, one for each half-hour in the week
- The chromosome consists of the complete set of schedules, one for each employee
- The fitness function determines the number of production phases completed during the week using the set of schedules
- Fitness measure also factors in labor cost and other optimization goals



Acme schedule genome

- Acme is open 6am-6pm each weekday, or 24 half-hour periods
- *startPeriods* and *periodsWorked* contain integers from 1 to 24
- Employees are allowed a paid half-hour lunch period between 11am and 2pm
- *lunchPeriods* are integers from 11 to 16
- A work week contains 120 time periods
- *baySchedules* contains 120 bay numbers, all integers from 1 to 8



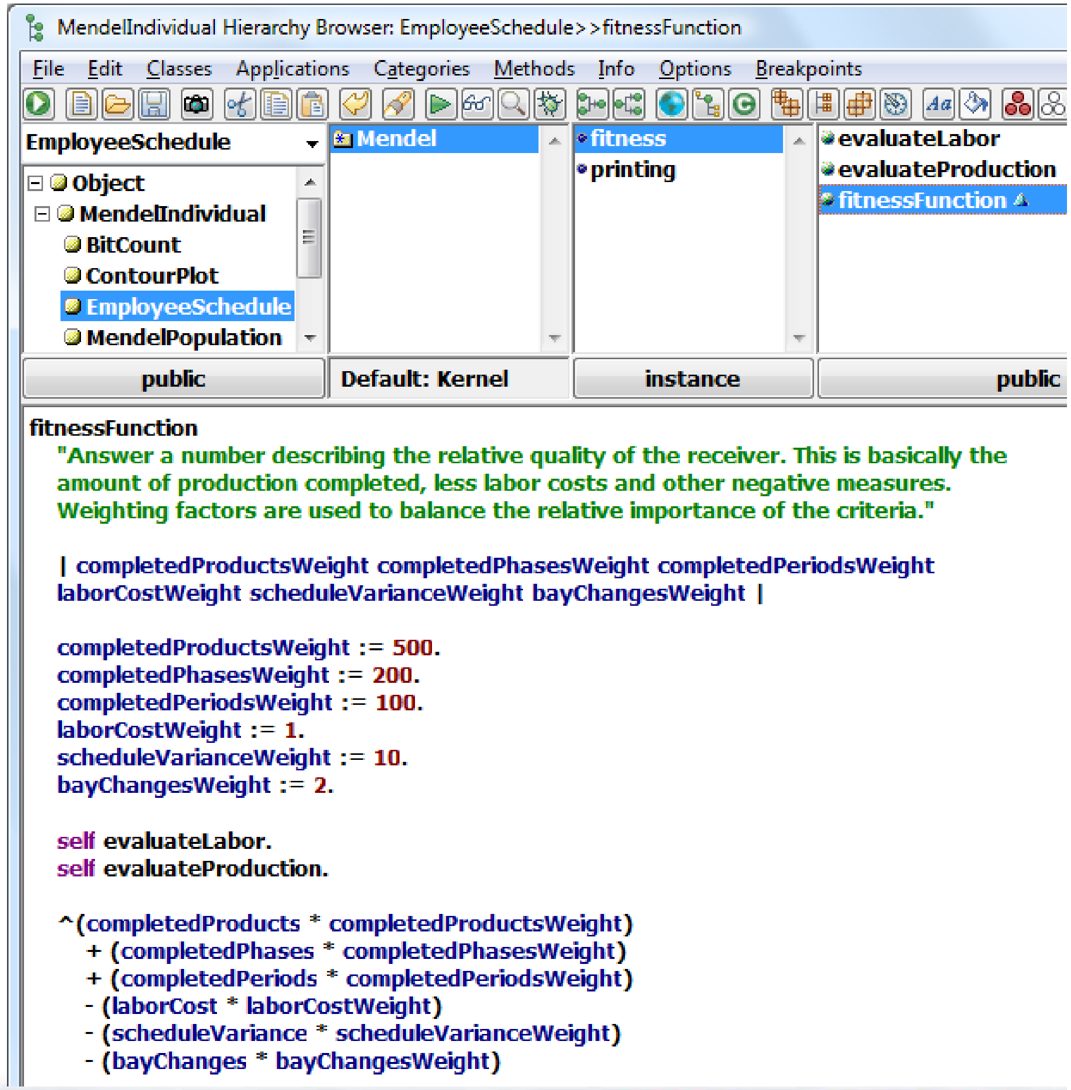
Genome design tips

- **Avoid redundant information**
  - Prevents contradictory states and need for synchronization or repair of separate genes
  - Example: production schedule is determined from employee schedules, not encoded separately
- **Use simple types with limited ranges wherever possible**
  - Appropriate granularity greatly improves efficiency
  - Example: encode times as integer period numbers, not hours and minutes
- **Instead of penalizing or repairing invalid states, find an encoding where all states are valid**
  - Example: Instead of *startPeriods* and *endPeriods*, use *startPeriods* and *periodsWorked*

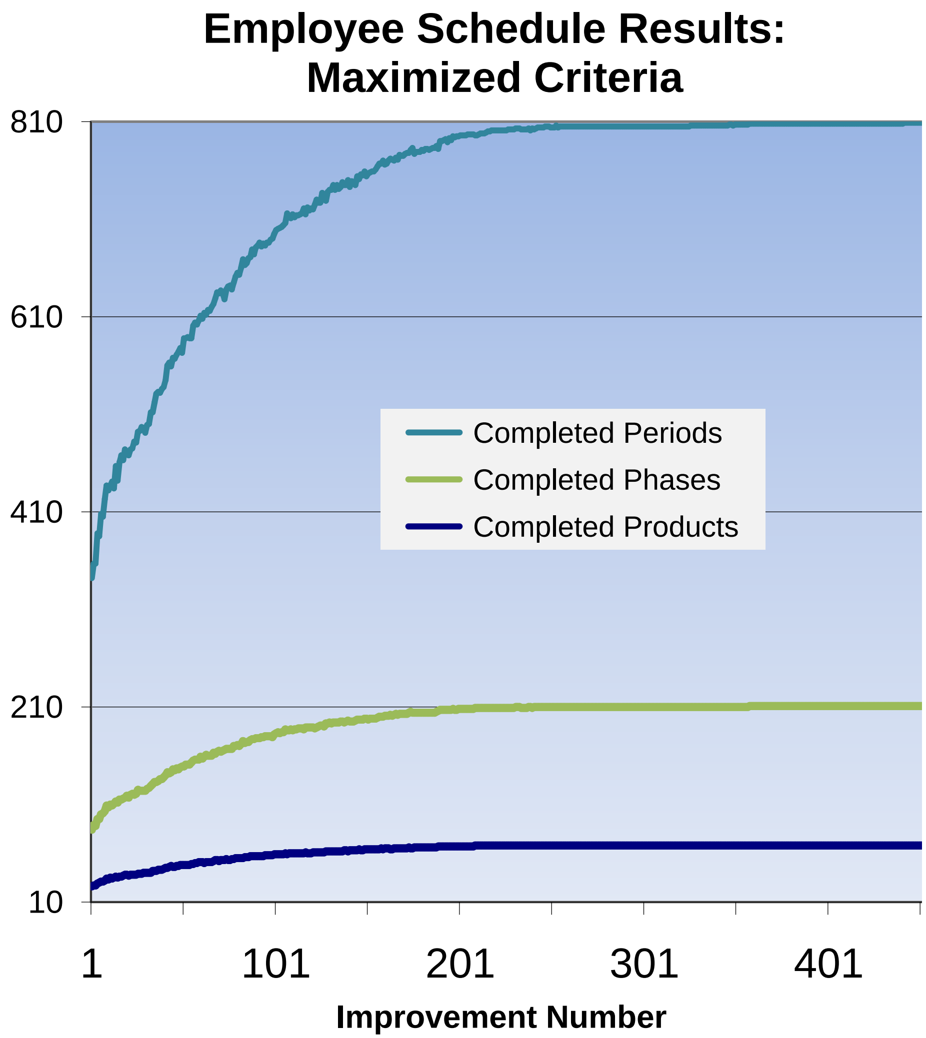
instantiations

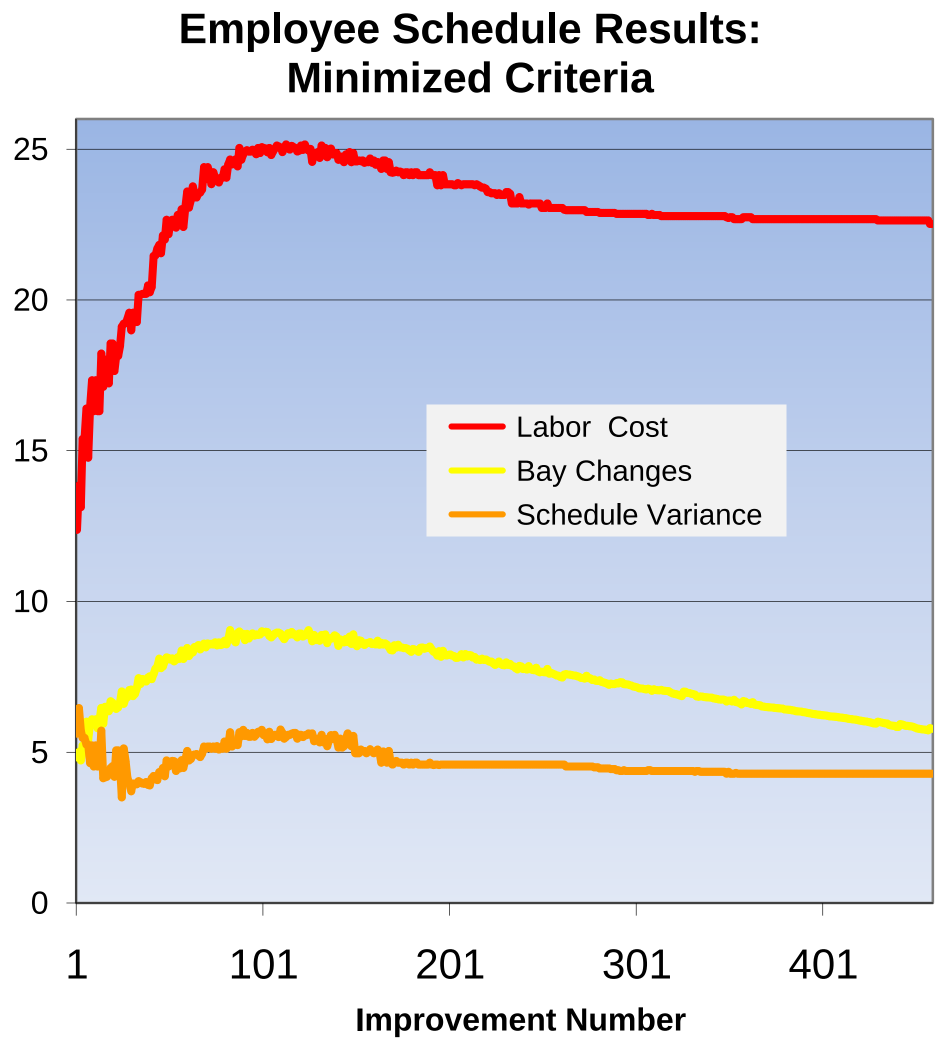
Acme schedule fitness function

- Six criteria go into the final fitness measure
- Each has an associated weight used to balance the goals
- Methods *evaluateLabor* and *evaluateProduction* calculate the criteria and store them into instance variables



Employee schedule results





OO design tradeoffs in Mendel

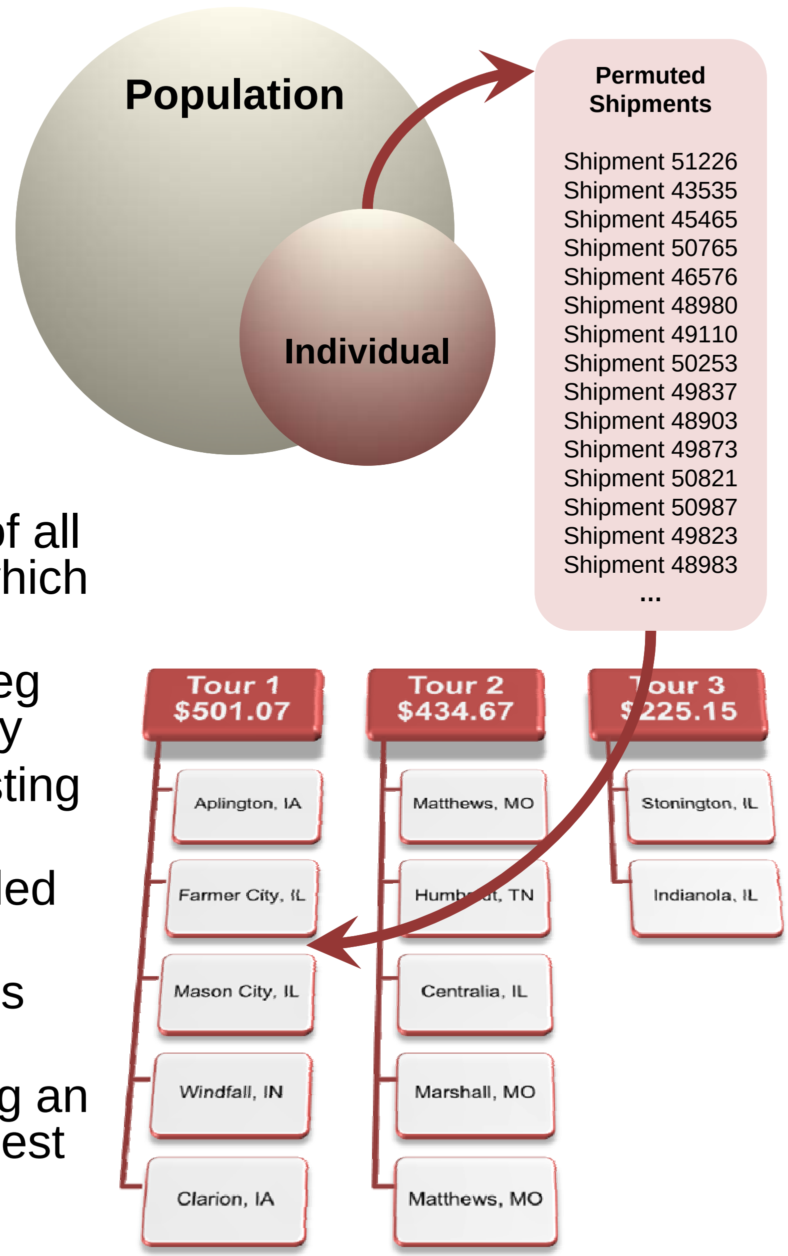
- Some compromises in good object-oriented design were made to achieve efficiency or flexibility goals:
  - Chromosomes are not first class objects; genes contain all the behavior, operating on the allele data (bad!)
    - **Cons:** Can't simply tell an allele to crossover or mutate, must pass individual and locus parameters to the gene instances
    - **Pros:** Greatly reduced memory usage for individuals, one copy of the genome is shared among all population members
  - Genes not implemented as instance variables
    - **Cons:** Less efficient allele access (requires dictionary lookup), less readable fitness function (*self valueOf: #name*)
    - **Pros:** Simplifies genetic operations, allows for gene reordering, allows problems to be defined without requiring a new class

Case study: Truck tour optimization

- A major agricultural supplier creates hundreds of shipments each week
- Significant cost savings are possible by creating *tours* that keep the same truck in service for up to five days
- Tour requirements:
  - Minimize empty distance
  - End tour as close as possible to origin
  - Satisfy time window constraints
  - Ensure all pickups and deliveries are during facility open hours
  - Provide adequate time for loading, unloading, and driver rest periods

Tour Builder approach

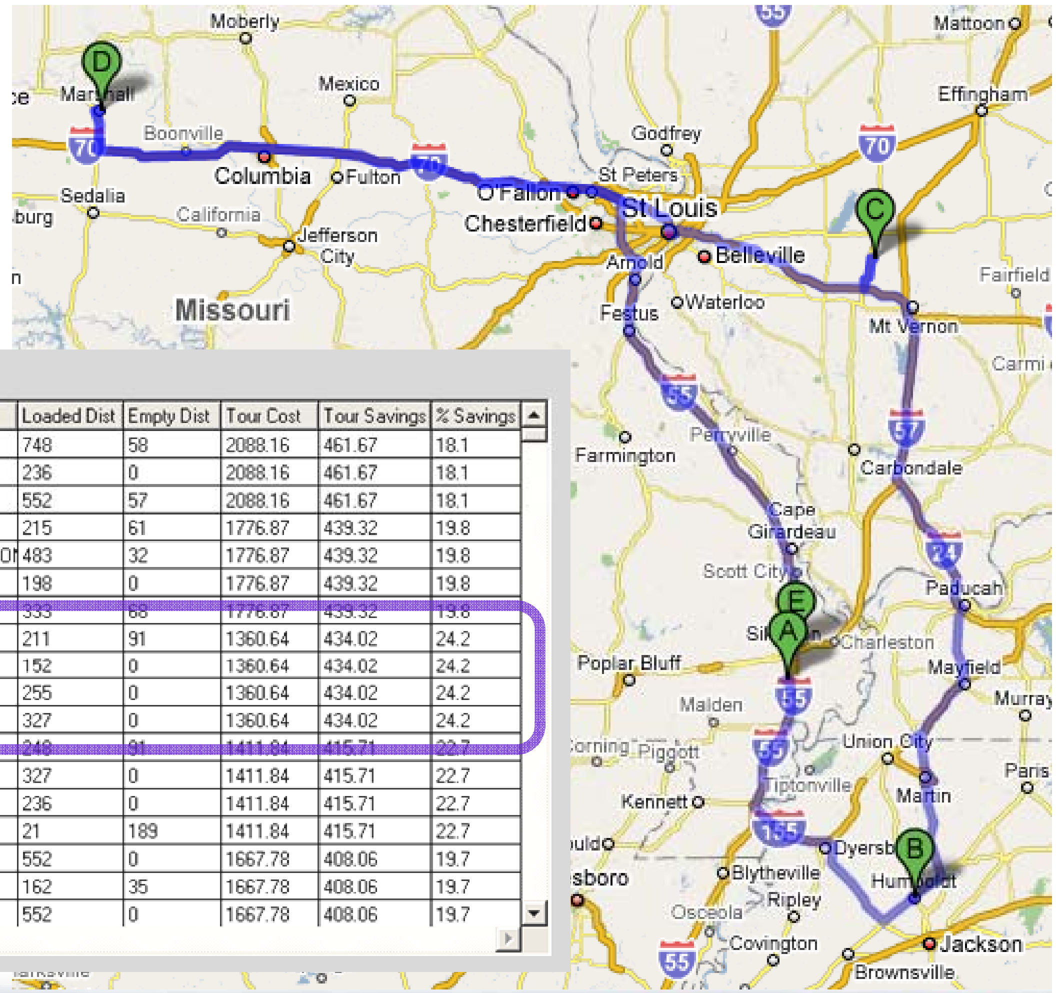
- A *hybrid GA* technique combines the Mendel framework with an *insertion algorithm*
- Each GA individual is a permutation of all shipment *legs*, defining the order in which they will be processed
- The insertion algorithm places each leg on the tour that saves the most money
- If no savings are possible on any existing tour, a new tour is created
- The resulting collection of tours is called a *shipping plan*
- The plan with the lowest overall cost is presented to the user
- The GA works at a high level, evolving an insertion order that creates the cheapest overall plan



.::instantiations

Tour builder results

- As the GA runs in the background, the result table is updated to show the current best tour plan
- This shows one tour with a savings of 24.2%

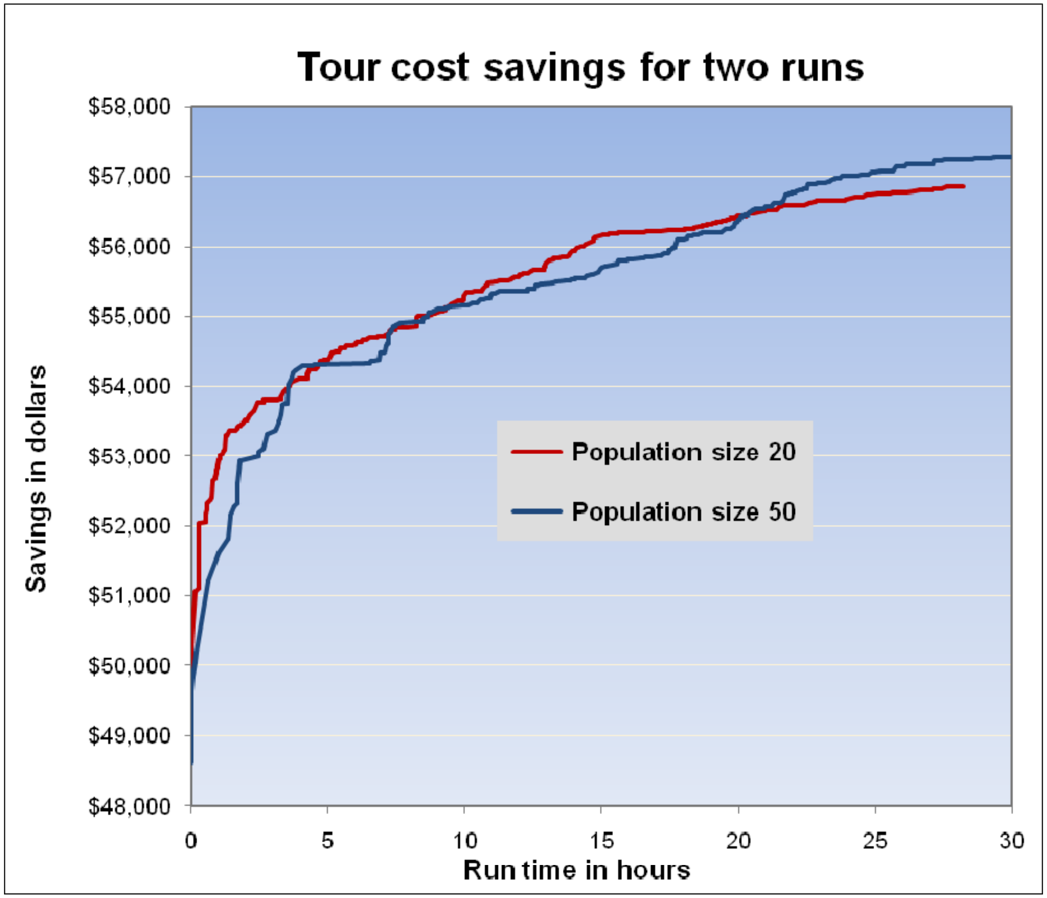


Tour Results

Tour	Leg	Load ID	Ship Date	Origin	Destination	Loaded Dist	Empty Dist	Tour Cost	Tour Savings	% Savings
1	1	51226	05/08/07	APLINGTON, IA	FARMER CITY, IL	748	58	2088.16	461.67	18.1
1	2	51202	05/08/07	MASON CITY, IL	WINDFALL, IN	236	0	2088.16	461.67	18.1
1	3	50987	05/08/07	WINDFALL, IN	CLARION, IA	552	57	2088.16	461.67	18.1
2	1	50409	05/07/07	STONINGTON, IL	HOMER, IL	215	61	1776.87	439.32	19.8
2	2	50244	05/07/07	DECATUR, IL	COLUMBUS JUNCTIO	483	32	1776.87	439.32	19.8
2	3	50654	05/07/07	MUSCATINE, IA	WEBSTER CITY, IA	198	0	1776.87	439.32	19.8
2	4	51112	05/08/07	WEBSTER CITY, IA	KILBOURNE, IL	333	68	1776.87	439.32	19.8
3	1	50302	05/07/07	MATTHEWS, MO	HUMBOLDT, TN	211	91	1360.64	434.02	24.2
3	2	50185	05/07/07	MATTHEWS, MO	CENTRALIA, IL	152	0	1360.64	434.02	24.2
3	3	51198	05/08/07	CENTRALIA, IL	MARSHALL, MO	255	0	1360.64	434.02	24.2
3	4	51120	05/08/07	MARSHALL, MO	MATTHEWS, MO	327	0	1360.64	434.02	24.2
4	1	51184	05/08/07	STONINGTON, IL	INDIANOLA, IL	248	91	1411.84	415.71	22.7
4	2	50968	05/08/07	STONINGTON, IL	MASON CITY, IL	327	0	1411.84	415.71	22.7
4	3	50858	05/08/07	MASON CITY, IL	WINDFALL, IN	236	0	1411.84	415.71	22.7
4	4	50957	05/08/07	WINDFALL, IN	KEMPTON, IN	21	189	1411.84	415.71	22.7
5	1	50627	05/07/07	CLARION, IA	WINDFALL, IN	552	0	1667.78	408.06	19.7
5	2	50753	05/08/07	WINDFALL, IN	PERU, IN	162	35	1667.78	408.06	19.7
5	3	50986	05/08/07	WINDFALL, IN	CLARION, IA	552	0	1667.78	408.06	19.7

Tour builder performance

- Chart shows cost savings for a high-volume week with 1463 total shipments
- Two runs were made, differing only in population size:
  - 20 individuals
    - 240 tours created
    - 7.5% savings after 28 hours
  - 50 individuals
    - 244 tours created
    - 7.6% savings after 30 hours



Smalltalk efficiency tips

- Minimize instance creation; it's comparatively slow (but much faster than in other OO languages). And eventually objects must be garbage collected
- Use *SmallIntegers* instead of *Floats* wherever possible
- Use *Arrays* and *OrderedCollections* instead of *Sets*, *Dictionaries*, or *SortedCollections* wherever possible
- Preallocate collections when you know their size (or a good guess); growing collections is a very time-consuming process
- Assignment is cheap, copying is expensive; don't copy objects unless you have a good reason
- Unless you're writing proxy objects, don't override *doesNotUnderstand:*, it's very slow
- Keep your design simple and modular: Complex, highly interdependent code can rarely be improved much
- Create a benchmark for profiling; record benchmark results before and after each efficiency improvement, and compare them

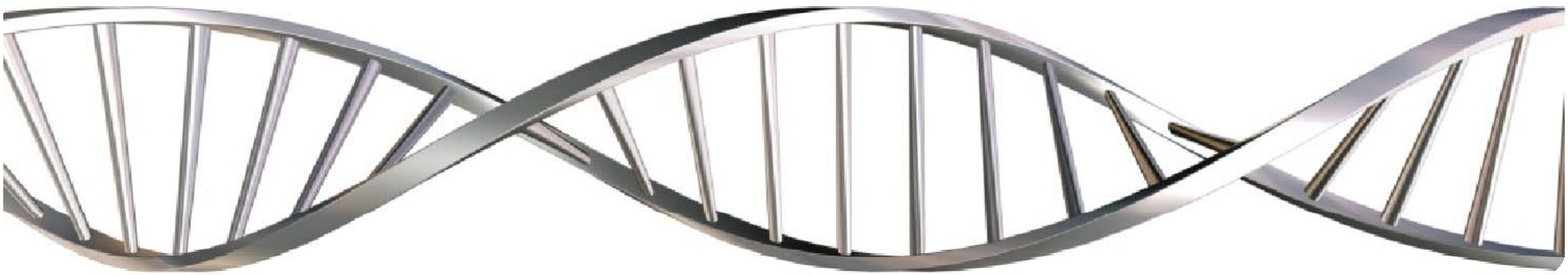
Smalltalk efficiency traps

- Don't optimize until and unless you know it will help, by profiling the code
- Don't try to wring microseconds from each method by being clever: *The simplest thing that could possibly work* is usually fast enough
- Don't be afraid to use accessors for instance variables; remove them only in the most critical code
- Don't be afraid to use lazy initialization; it's fast and sometimes actually results in *better* efficiency
- Always write code in the same consistent style; if you must make exceptions for efficiency reasons, document them well

Questions?

Bob Whitefield
ModelDesign Corporation

bob@modeldesign.com
(919) 418-0300



modeldesign