

# Using Genetic Programming to Create Smart Ants

Evolutionary Computing

John Hall

## Abstract

*This research implements a genetic program to solve the Santa Fe Trail problem. In order to understand the effectiveness of various parameters, the population size, the mutation rate, and the crossover type were varied. In general, it appears that larger populations sizes and higher mutation rates were effective. The effects of crossover were less clear. In general the genetic program solved the problem well. In fact, some of the algorithms created by the genetic program were better than the author was able to create by hand.*

## Introduction

Evolutionary computing solves problems by applying evolutionary concepts. Simple genetic algorithms do this by creating a random population of individuals, selecting parents based on the fitness of the individuals, and applying crossover and mutation operators to the parents to create offspring. Eventually the population evolves to represent better solutions.

Genetic programming works by evolving a program to solve a problem instead of solving the problem directly. Genetic programming works by creating a population of initial random programs. In order to create syntactically correct programs, the individuals are usually represented as expression trees. The genetic program models evolution by selecting parents based on how good a program an individual represents. Just as in a genetic algorithm, the evolutionary operations of crossover and mutation are applied to the programs. However, these operators behave slightly differently on trees. This process causes the population of programs to evolve to better solve the problem. Genetic programming actually designs an algorithm based on some criteria. Genetic programming is the only known technique to automatically generate an algorithm.

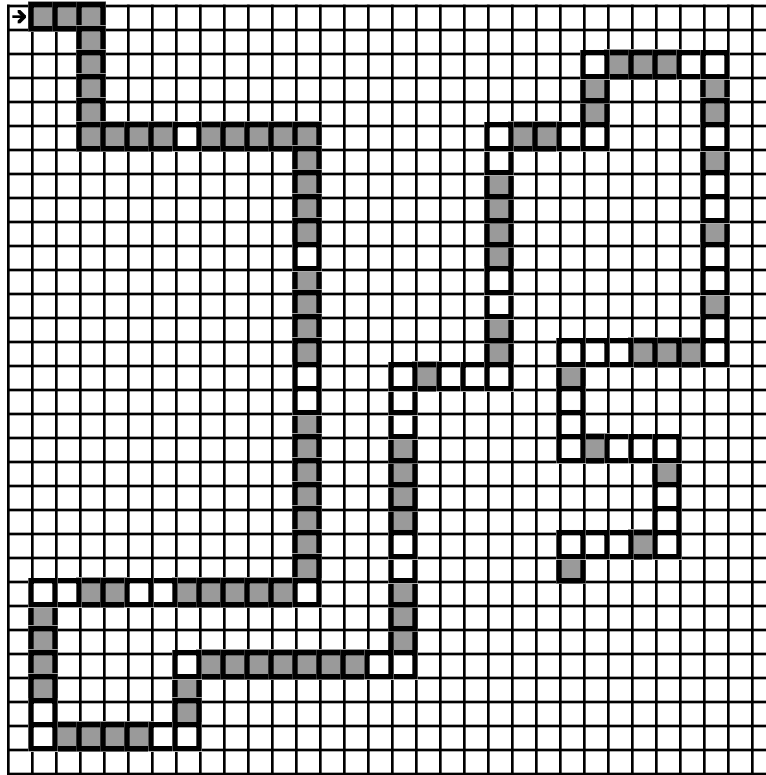


Figure 1, The Santa Fe Trail

This research investigated applying genetic programming to a problem known as the Santa Fe Trail. The programs built represent ants that explore a theoretical 32x32 grid looking for food. A successful program is one that locates as much food as possible. The food is located along a trail. The programs do not need to follow the trail, but they the ant is limited to 600 operations. Not all spaces on the trail include food. Figure 1 shows this theoretical landscape. The dark squares represent spaces with food. The squares with dark edges represent spaces that are on the trail, but do not contain food. The ant starts in the upper left corner facing right. The ant eats a piece of food by occupying the same grid location. Once a piece of food is eaten, the ant will not gain anything by revisiting that location. The grid wraps around in all directions in a torrid. If the ant goes off the grid in one direction, it will end up on the other side of the grid.

The trail is designed so that it becomes more difficult to find food the farther along the trail the ant travels. In order to successfully find the next piece of food on the trail, the programs will need to evolve some type of searching technique. In order to solve this problem, the genetic program must invent simple searching techniques and combine them to make better search techniques. More information on both the Santa Fe Trail problem and Genetic Programming in general can be found at [3].

## Experiment

In order gain a better understanding of genetic programming, this research implemented a genetic

program to investigate the Santa Fe Trail problem discussed above. The code developed as well as the output data is available at: <http://www.johnmhall.net/classes/evolution/ant.tar.gz>. Excerpts of interesting code are shown in Appendix A.

Each program was represented by an expression tree. The terminal nodes on this tree represented actions that a simulated ant could perform. There were three terminals, “Forward”, “Left”, and “Right”. As implied by their names, these actions would move or rotate the simulated ant. The non-terminals in the tree were the structure of a program. There were three non-terminals, “If Food Ahead”, “Prog2”, and “Prog3”. “If Food Ahead” will execute the commands on the left branch if there is food immediately in front of the simulated ant, and will execute the commands on the right branch if there is not food immediately in front of the simulated ant. “Prog2” and “Prog3” allow compound statements. “Prog2” will execute the commands on the left branch followed by the commands on the right branch. “Prog3” will execute the commands on the left branch, then the commands on the center branch, and then finally, the commands on the right branch. All non-terminals have three branches. “If Food Ahead” and “Prog2” ignore the commands on the center branch. All valid solutions are constrained in that they must contain 500 or fewer nodes. Any solution that is

The fitness of each solution is calculated by actually running the program for the simulated ant. The ant is limited to 600 actions including “Forward”, “Left”, and “Right”. The simulated ant will immediately stop moving once this limit is reached. The fitness of each program is represented by an integer. In a simplified sense, the fitness is the number of food squares visited. However, the actual value is a little more complicated. In order to favor smaller programs, a process known as applying parsimony pressure, the number of food locations is actually multiplied by 10,000 and the number of nodes in the program is subtracted from 1000 and added to this result. Since a tree can contain a maximum of 500 nodes, the number of nodes in a program will only be used as a tiebreaker in the event that two solutions visit the same number of food locations.

The initial population of solutions is created randomly using a process of ramped half and half. Using the method, half of the trees are full and half are randomly grown. The maximum size of the tree cycles between 4, 5, and 6. The grown trees all have a non-terminal for the root node. At all other positions in the tree, there is an 80% chance of a non-terminal, and a 20% chance of a terminal. In all trees, both terminals and non-terminals are chosen uniformly from the appropriate node types.

The genetic program uses a generational model to update the population. This means, that in each generation, a new population is created from the current population. The new population then replaces the current population. In order to guarantee that the best solution is never lost, a technique known as elitism is used. Two copies of the best solution in a generation are added to the next generation. Offspring from the current generation are added until the generation is full. The offspring are created by selecting two parents. This selection is done using a process known as tournament selection. In this system, tournament selection chooses a parent by choosing the

most fit of three randomly chosen individuals. Crossover and mutation are done to copies of the two selected parents. If either of these individuals contains more nodes than the maximum discussed above, it is discarded. The surviving offspring are added to the next population.

In order to test the effectiveness of various parameters on the genetic program, several population sizes, multiple versions of crossover, and several mutation rates were used. Population sizes of 50, 100, 200, 500, and 1000 were tried. In order to compare these accurately, the number of generations varied to produce a total population count of 100,000 individuals. In order to minimize the effect of a lucky or unlikely random success, each experiment was tried over 20 different trials with different random number seeds. Two types of crossover were used. The first was random crossover where the crossover branches were chosen completely randomly. The second type selected the branch closest to the top of the tree from two or four branches chosen at random. The intent of this selection method was to increase the chance that selection would occur on nodes higher in the tree. The mutation rates tried were 1, 2, and 3 per program. Both crossover and mutation are always applied. Varying these three parameters resulted in 45 different test cases.

Mutation will always change a terminal to a terminal, and a non-terminal to a non-terminal. Since all non-terminals have three branches, this process may cause operative branches to become inoperative and vice-versa. Unfortunately, since there is a 1/3 chance that the mutation will operate on an unused branch, and there is a 1/3 chance that the mutation will change an operator to the same type, there is only a 4/9 probability of each mutation actually modifying a program.

In previous experiments, a simple linear convergence random number generator was shown ineffective. As a result, the Mersenne Twister random number generator [2] was used to generate all random numbers.

## **Results**

In general the algorithms ran much as an evolutionary algorithm may be expected to. Figure 2 shows the improvement over time of our simulated ant problems. This data is from using a population of size 100 for 1000 generations with one mutation per tree and random crossover point selection. As shown, the solutions improve greatly at first, and then begin to improve more slowly the more generations that are run. The average solution behaves similarly. In general, the average follows a similar curve with smaller values, but with much more variation. It is interesting how the average fitness curve is only about 60% as large as the best solution curve. This implies that there are a large number of bad programs produced by crossover and mutation. In a genetic program, these operators seem to be damaging to a program's fitness.

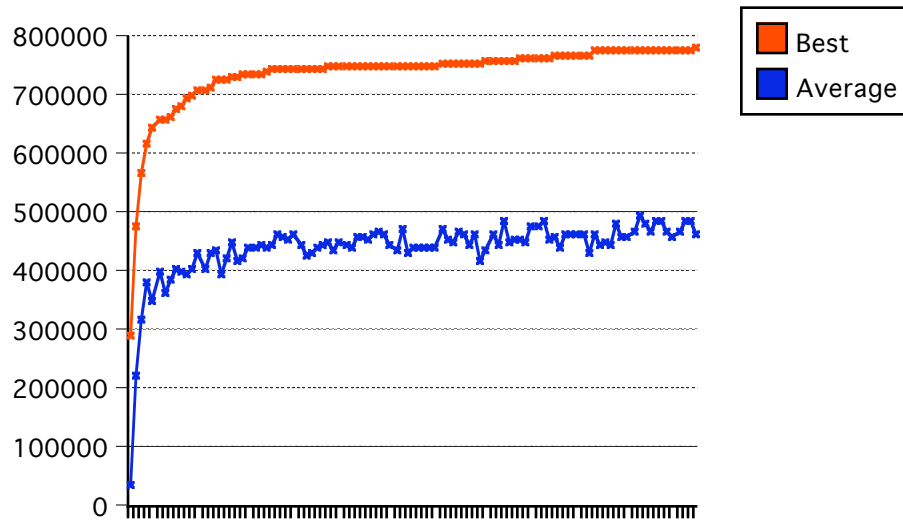


Figure 2, Improvement in the Best and Average Fitness of the Population

In order to determine the effectiveness of the population size on finding the best solution, many different population sizes were tried. Figure 3 shows the results of different population sizes. The mutation rate was one per tree and the crossover points were chosen randomly. In order to make the results somewhat comparable, all experiments were run until 100,000 individuals were added to the population.

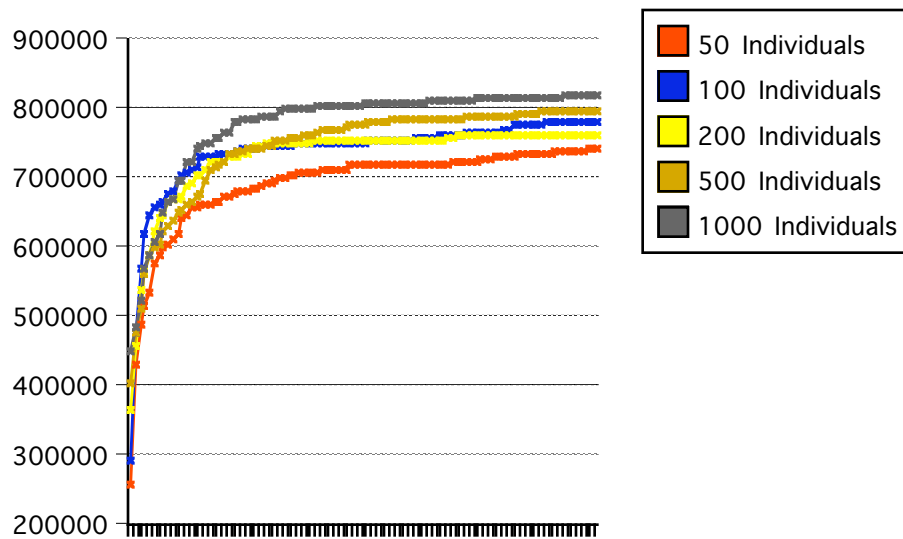


Figure 3, Improvement of Best Individual Based on Size.

Increasing the mutation rate also improved the best solutions. Figure 4 shows the effect of different mutation rates. The population size was 1000 and the crossover points were chosen

randomly.

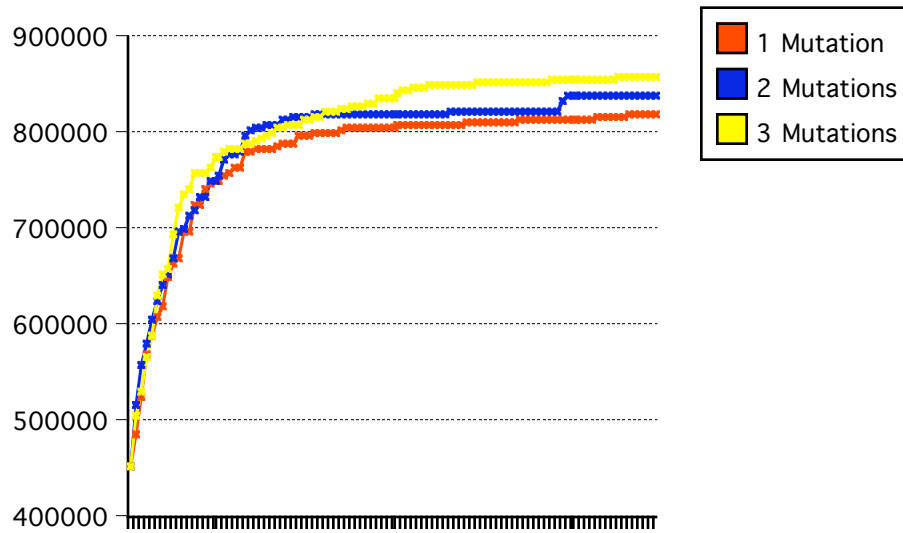


Figure 4, Improvement of Best Individual Based on Mutation Rate

Figure 5 shows the effect of using different crossover operations. The population size was 1000 and the mutation rate was one per tree. The mechanics of these variations are discussed in the introduction section above.

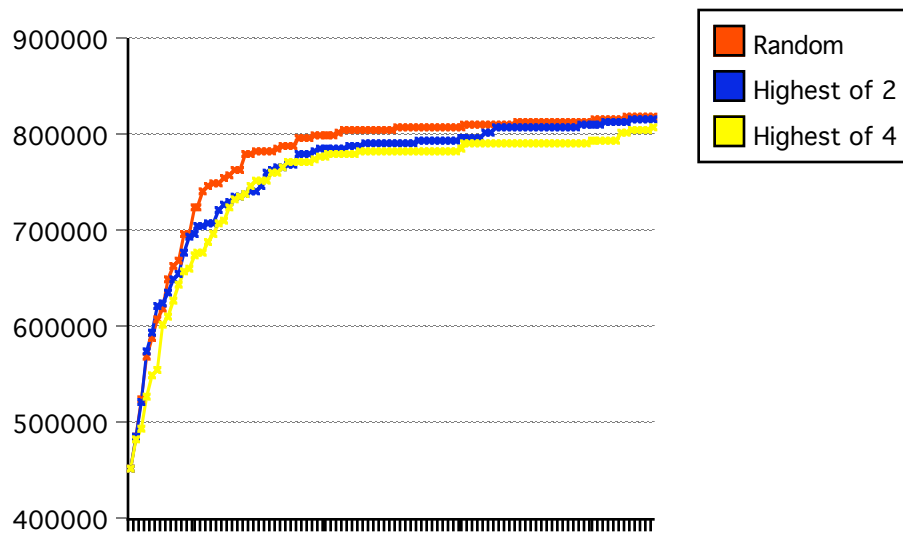


Figure 5, Improvement of Best Individual Based on Crossover Type

All 45 test cases were evaluated 20 times. The genetic program ran for approximately 12 hours on a fast (~2 GHz) Linux server. The original 90 MHz server used in the previous project would

have taken at least 18 days to run these tests. Of these 900 total runs, 399 found an optimal solution that visited each food location. Of the remaining 501 runs, all but 8 are shown in Figure 6. The remaining 8 solutions all found 40-49 of the food items.

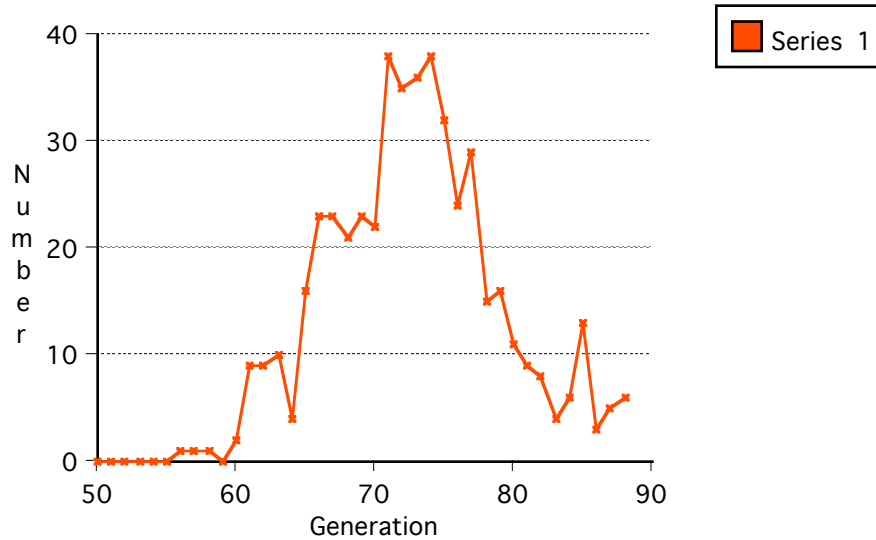


Figure 6, Distribution of Test Case Results

### Conclusions

This experiment varied the population size, the mutation rate, and the crossover method. In general larger population sizes resulted in better improvements. Presumably this is because the larger populations allow for more diversity. It would be interesting to have tried even larger population sizes and seen when the effectiveness was diminished.

The mutation rate also helped improve the best solutions. Mutation allows individuals in the population to make small changes to improve their solution. From these results it appears that mutation is an extremely beneficial algorithm in a genetic program.

The results from trying different crossover methods are not as clear. Choosing random branches gave slightly better results than the other two methods. Since the other two methods were designed to pick nodes higher in the tree, it would appear as though crossover is too destructive. These results may have been influenced by the presence of parsimony pressure. Since the solutions are being driven to eliminate introns, crossover is especially damaging. However, the differences between the randomized branch selection and the highest of 2 branch selection are not significantly different. It would be interesting to try highest of 3 and highest of 5 selection methods to see if these results are indeed correct.

Growth was effectively controlled using two methods, a maximum node count and parsimony

pressure. The genetic program used both of these methods to effectively limit the size of solutions generated. An early version of the genetic program had a bug that caused the size of a tree to be calculated incorrectly. In this version the algorithm quickly slowed down and used excessive memory despite parsimony pressure. Since parsimony pressure was only imposed as a tiebreaker, the algorithms were still allowed to improve.

The best program encountered out of all 4.5 million generated individuals found all 89 food locations and consisted of 13 nodes. The program is shown in simplified form, all program statements were flattened in Figure 7. Figure 8 shows the same program with all tree elements shown. This program shows how impressive a genetic program can be. This example exhibits advanced searching techniques in an extremely simplified program. This program does have a slight overfit to the problem being solved. For example, the move forward at the beginning is unconditional. It will also only search forward from the last food location visited. Since the Santa Fe Trail food distribution always puts a piece of food along this path, the ant follows the entire path.

```
Fitness: 890987
Move Forward
Turn Right
if (food_ahead) {
  Move Forward
} else {
  Turn Right
  Turn Right
  if (food_ahead) {
    Move Forward
  } else {
    Turn Right
  }
}
```

Figure 7, Best Program Discovered (Simplified) (89 Food, 13 Nodes)

```

Fitness: 890987
Prog 3
  Move Forward
  Turn Right
  if (food_ahead) {
    Move Forward
  } else {
    Prog 3
    Turn Right
    Turn Right
    if (food_ahead) {
      Move Forward
    } else {
      Turn Right
    }
  }
}

```

Figure 8, Best Program Discovered (Complete) (89 Food, 13 Nodes)

Several other configurations were played with while doing this research. One of the most interesting was changing the location of food spaces on the trail. The first method simply put 100 food spaces in random locations. The second variation added 100 food spaces to the standard trail. In the first case, the ants tended to use complex programs to find as much food as possible. It seems as though this indicated overfitting. In the second method, the programs followed most of the sections of the trail, but they tended to take other paths that allowed them to pick up more of the randomly placed food along the way.

This genetic program worked well. Genetic programming is a powerful technique that almost mimics creativity. It would be interesting to try genetic programming on a larger problem.

## **Bibliography**

- [1] Mitchell, Melanie, An Introduction to Genetic Algorithms, The MIT Press, Cambridge Massachusetts, 1996.
- [2] Narasimhan, B., The Mersenne Twister in Java, <http://citeseer.nj.nec.com/313273.html>, February 2003.
- [3] Soule, Terry, CS472/572 GA Project. <http://www.cs.uidaho.edu/~tsoule/cs472/GPproj.html>, April 2003.

## Appendix A, Headers and Other Selected Code Sections

From CLandscape.h:

```
4 /*
5  CLandscape
6  Defines a landscape used in the Santa Fe Trail problem. This landscape is covered with
7  food items. Currently the landscape is hardcoded to a 32 by 32 toroidal grid. Each piece
8  of food on the landscape will be numbered. The landscape does not change as an ant is
9  executing. Instead, another class will be responsible for determining if a given piece of
10 food has already been consumed.
11
12 Author: John M. Hall
13 */
14
15 #include "foundation.h"
16
17 class CLandscape
18 {
19 public:
20  CLandscape(); // Hard Coded Landscape.
21  CLandscape(CRandom&); // Random Landscape.
22
23  bool  HasFood(SInt32,SInt32) const;
24  UInt32 GetFoodIndex(SInt32,SInt32) const;
25
26  UInt32 GetFoodCount() const;
27
28  void  ClearFood();
29  void  AddFood(UInt32,UInt32);
30  void  AddFood(UInt32,UInt32,UInt32,UInt32);
31
32  enum Direction
33  {
34    kDirectionNorth,
35    kDirectionEast,
36    kDirectionSouth,
37    kDirectionWest,
38    kDirectionCount
39  };
40  typedef CTriple<SInt32,SInt32,Direction> Position;
41  typedef CArray<Position> Path;
42
43  void DisplayMap(CTextWriter&,const Path&) const;
```

```

44
45 private:
46 void DumpMap(CTextWriter&,const CMatrix<char>) const;
47
48 UInt32      mWidth;
49 UInt32      mHeight;
50 UInt32      mFoodCount;
51 CMatrix<UInt32>  mData;
52 };

```

From AntState.h:

```

4 /*
5  CAntState
6  The state of an ant going through a landscape. This includes the ant's location,
7  direction, and which food items have been eaten.
8
9  Author: John M. Hall
10 */
11
12 #include "CLandscape.h"
13
14 class CAntState
15 {
16 public:
17     typedef CLandscape::Path Path;
18
19     CAntState(const CLandscape&,UInt32,Path* =NULL);
20
21     bool  AntTooTiredToContinue();
22
23     bool  IsFoodAhead() const;
24     UInt32  CountFoodEaten() const;
25
26     void  MoveForward();
27     void  TurnRight();
28     void  TurnLeft();
29
30 private:
31     static const SInt32 kDeltaX[CLandscape::kDirectionCount];
32     static const SInt32 kDeltaY[CLandscape::kDirectionCount];
33
34     const CLandscape&      mLandscape;
35     SInt32                  mXPosition;

```

```

36  SInt32          mYPosition;
37  CLandscape::Direction  mDirection;
38  UInt32         mStepsLeft;
39  CBinaryString  mFoodEaten;
40  Path*          mSteps;
41 };

```

From CStrategy.h:

```

4 /*
5  CStrategy
6  This is a program used by a simulated ant in the Santa Fe Trail problem.
7
8  Author: John M. Hall
9 */
10
11 #include "CNode.h"
12 #include "CLandscape.h"
13
14 class CStrategy
15 {
16 public:
17  CStrategy(const CLandscape&,CRandom&,UInt32); // Full Tree
18  CStrategy(const CLandscape&,CRandom&,float,UInt32); // Grown Tree (Root is
nonterminal)
19  CStrategy(const CStrategy&);
20  CStrategy();
21  ~CStrategy();
22
23  typedef CLandscape::Path Path;
24  UInt32 CalculateFitness(Path* =NULL) const;
25
26  bool IsTooBig() const;
27
28  UInt32 GetFitness() const;
29
30  void Mutate(CRandom&);
31  void Crossover(CStrategy&,CRandom&,UInt32);
32
33  void PrintPath(CTextWriter&) const;
34
35  CStrategy& operator=(const CStrategy&);
36
37  bool operator==(const CStrategy& s) const { return (GetFitness()==s.GetFitness()); }

```

```

38  bool operator<(const CStrategy& s) const  { return (GetFitness()<s.GetFitness()); }
39 private:
40  static UInt32 GetRandomBranch(CNode*&,CRandom&,UInt32);
41
42  static const UInt32 kMaxSteps;
43
44  const CLandscape*  mLandscape;
45  CNode*             mRoot;
46
47  mutable UInt32     mFitness;
48
49  friend CTextWriter& operator<<(CTextWriter&,const CStrategy&);
50 };
51
52 CTextWriter& operator<<(CTextWriter&,const CStrategy&);

```

From CPopulation.h:

```

4 /*
5  CPopulation
6  A population of programs for the Santa Fe Ant problem. It handles a generational model.
7
8  Author: John M. Hall
9 */
10
11 #include "CStrategy.h"
12
13 class CPopulation
14 {
15 public:
16  CPopulation(const CLandscape&,UInt32,CRandom&);
17
18  CStrategy const& SelectBest() const;
19  CStrategy const& Select(CRandom&) const;
20
21  UInt32 GetAvgFitness() const;
22
23  CPopulation& RunGeneration(CRandom&,UInt32,UInt32);
24
25 private:
26  UInt32          mSize;
27  CVector<CStrategy> mPrimaryData;
28  CVector<CStrategy> mSecondaryData;
29  CVector<CStrategy>* mPrimary;

```

```

30  CVector<CStrategy>* mSecondary;
31
32  friend CTextWriter& operator<<(CTextWriter&,const CPopulation&);
33  };
34
35  CTextWriter& operator<<(CTextWriter&,const CPopulation&);

```

From CNonterminal.h:

```

13  class CNonterminal : public CNode
14  {
15  public:
16  CNonterminal(CRandom&,UInt32); // Random Full Tree
17  CNonterminal(CRandom&,float,UInt32); // Random Grown Tree
18  CNonterminal(const CNonterminal&);
19  virtual ~CNonterminal();
20
21  virtual CNonterminal* Clone() const;
22
23  virtual void Evaluate(CAntState&) const;
24
25  virtual UInt32 CountNodes() const;
26
27  virtual void Mutate(CRandom&);
28  virtual CNode& GetNodeIndex(UInt32);
29  virtual CNode*& GetBranchIndex(UInt32,CNode*&);
30  virtual UInt32 GetNodeDepth(UInt32);
31
32  virtual void Output(CTextWriter&,UInt32) const;
33
34  static VoidPtr operator new(UInt32);
35  static void operator delete(VoidPtr);
36
37 private:
38  UInt32 GetNodeCount() const;
39
40  enum Operation
41  {
42  kIfFoodAhead,
43  kProgTwo,
44  kProgThree,
45  kOperationCount
46  };
47

```

```
48 Operation    mOperation;
49 CNode*       mLeft;
50 CNode*       mCenter;
51 CNode*       mRight;
52 mutable UInt32 mNodeCount;
53 };
```

From CTerminal.h:

```
13 class CTerminal : public CNode
14 {
15 public:
16   CTerminal(CRandom&);
17   CTerminal(const CTerminal&);
18   virtual ~CTerminal();
19
20   virtual CTerminal* Clone() const;
21
22   virtual void Evaluate(CAntState&) const;
23
24   virtual UInt32 CountNodes() const;
25
26   virtual void Mutate(CRandom&);
27   virtual CNode& GetNodeIndex(UInt32);
28   virtual CNode*& GetBranchIndex(UInt32,CNode*&);
29   virtual UInt32 GetNodeDepth(UInt32);
30
31   virtual void Output(CTextWriter&,UInt32) const;
32
33   static VoidPtr operator new(UInt32);
34   static void operator delete(VoidPtr);
35
36 private:
37   enum Action
38   {
39     kMoveForward,
40     kTurnRight,
41     kTurnLeft,
42     kActionCount
43   };
44
45   Action mAction;
46 };
```

From CPopulation.h:

```
13 class CPopulation
14 {
15 public:
16   CPopulation(const CLandscape&,UInt32,CRandom&);
17
18   CStrategy const& SelectBest() const;
19   CStrategy const& Select(CRandom&) const;
20
21   UInt32 GetAvgFitness() const;
22
23   CPopulation& RunGeneration(CRandom&,UInt32,UInt32);
24
25 private:
26   UInt32      mSize;
27   CVector<CStrategy> mPrimaryData;
28   CVector<CStrategy> mSecondaryData;
29   CVector<CStrategy>* mPrimary;
30   CVector<CStrategy>* mSecondary;
31
32   friend CTextWriter& operator<<(CTextWriter&,const CPopulation&);
33 };
34
35 CTextWriter& operator<<(CTextWriter&,const CPopulation&);
```

From CAntState.cpp:

```
23 bool CAntState::IsFoodAhead() const
24 {
25   SInt32 x=mXPosition+kDeltaX[mDirection];
26   SInt32 y=mYPosition+kDeltaY[mDirection];
27   if (mLandscape.HasFood(x,y))
28   {
29     return !mFoodEaten[mLandscape.GetFoodIndex(x,y)];
30   }
31   return false;
32 }
33
34 UInt32 CAntState::CountFoodEaten() const
35 {
36   UInt32 total=0;
37   for (UInt32 i=0;i<mLandscape.GetFoodCount();i++)
38     if (mFoodEaten[i])
39       total++;
```

```

40 return total;
41 }
42
43 void CAntState::MoveForward()
44 {
45     if (!AntTooTiredToContinue())
46     {
47         mXPosition+=kDeltaX[mDirection];
48         mYPosition+=kDeltaY[mDirection];
49         if (mLandscape.HasFood(mXPosition,mYPosition))
50         {
51             mFoodEaten[mLandscape.GetFoodIndex(mXPosition,mYPosition)]=true;
52         }
53         mStepsLeft--;
54         if (mSteps!=NULL)
55             mSteps->Append(CLandscape::Position(mXPosition,mYPosition,mDirection));
56     }
57 }
58
59 void CAntState::TurnRight()
60 {
61     if (!AntTooTiredToContinue())
62     {
63         mDirection=(CLandscape::Direction)(((UInt32)mDirection+1)%CLandscape::kDirectionCount)
;
64         mStepsLeft--;
65         if (mSteps!=NULL)
66             mSteps->Append(CLandscape::Position(mXPosition,mYPosition,mDirection));
67     }
68 }
69

```

From CNonterminal.cpp

```

65 void CNonterminal::Evaluate(CAntState& state) const
66 {
67     if (mOperation==kIfFoodAhead)
68     {
69         if (state.IsFoodAhead())
70             mLeft->Evaluate(state);
71         else
72             mRight->Evaluate(state);
73     }

```

```

74  else if (mOperation==kProgTwo)
75  {
76      mLeft->Evaluate(state);
77      mRight->Evaluate(state);
78  }
79  else if (mOperation==kProgThree)
80  {
81      mLeft->Evaluate(state);
82      mCenter->Evaluate(state);
83      mRight->Evaluate(state);
84  }
85  else
86  {
87      ASSERT(false);
88  }
89 }

96 UInt32 CNonterminal::GetNodeCount() const
97 {
98     if (mNodeCount==kUnknownDepth)
99         mNodeCount=(1+mLeft->CountNodes()+mCenter->CountNodes()+mRight-
>CountNodes());
100     return mNodeCount;
101 }
102
103 void CNonterminal::Mutate(CRandom& random)
104 {
105     mOperation=(Operation)(random.GetUInt32((UInt32)kOperationCount));
106 }
107
108 CNode& CNonterminal::GetNodeIndex(UInt32 x)
109 {
110     // order: left center right this
111     UInt32 leftCount=mLeft->CountNodes();
112     if (x<leftCount)
113         return mLeft->GetNodeIndex(x);
114     x-=leftCount;
115     UInt32 centerCount=mCenter->CountNodes();
116     if (x<centerCount)
117         return mCenter->GetNodeIndex(x);
118     x-=centerCount;
119     UInt32 rightCount=mRight->CountNodes();
120     if (x<rightCount)

```

```

121     return mRight->GetNodeIndex(x);
122     x-=rightCount;
123     ASSERT(x==0);
124     return *this;
125 }

```

From CTerminal.cpp:

```

24 void CTerminal::Evaluate(CAntState& state) const
25 {
26     if (mAction==kMoveForward)
27         state.MoveForward();
28     else if (mAction==kTurnRight)
29         state.TurnRight();
30     else if (mAction==kTurnLeft)
31         state.TurnLeft();
32     else
33         ASSERT(false);
34 }

```

From CPopulation.cpp:

```

8 CPopulation::CPopulation(const CLandscape& landscape,UInt32 size,CRandom&
random)
9 :mSize(size)
10 ,mPrimaryData(mSize)
11 ,mSecondaryData(mSize)
12 ,mPrimary(&mPrimaryData)
13 ,mSecondary(&mSecondaryData)
14 {
15     UInt32 depth=kMinDepth;
16     bool fullTree=true;
17     for (UInt32 i=0;i<mSize;i++)
18     {
19         if (fullTree)
20             (*mPrimary)[i]=CStrategy(landscape,random,depth);
21         else
22             (*mPrimary)[i]=CStrategy(landscape,random,kNonterminalProb,depth);
23         if (depth==kMaxDepth)
24         {
25             depth=kMinDepth;
26             fullTree=!fullTree;
27         }
28     }
29 }

```

```

64 CPopulation& CPopulation::RunGeneration(CRandom& random,UInt32
mutationCount,UInt32 crossCount
)
65 {
66     UInt32 i=0;
67     // Elitism
68     CStrategy best=SelectBest();
69     (*mSecondary)[i++]=best;
70     (*mSecondary)[i++]=best;
71     // Evolution
72     while (i<mSize)
73     {
74         CStrategy first=Select(random);
75         CStrategy second=Select(random);
76         first.Crossover(second,random,crossCount);
77         for (UInt32 j=0;j<mutationCount;j++)
78         {
79             first.Mutate(random);
80             second.Mutate(random);
81         }
82         if (!first.IsTooBig())
83         {
84             (*mSecondary)[i++]=first;
85         }
86         if ((!second.IsTooBig())&&(i<mSize))
87         {
88             (*mSecondary)[i++]=first;
89         }
90     }
91     // Use this new pool.
92     Swap(mPrimary,mSecondary);
93     return *this;
94 }

```

From CStrategy.cpp:

```

54 UInt32 CStrategy::CalculateFitness(Path* steps) const
55 {
56     CAntState state(*mLandscape,kMaxSteps,steps);
57     while (!state.AntTooTiredToContinue())
58     {
59         mRoot->Evaluate(state);
60     }

```

```

61  return (10000*state.CountFoodEaten())+(1000-mRoot->CountNodes());
62 }
63
64 void CStrategy::Mutate(CRandom& random)
65 {
66     mFitness=kUnknownFitness;
67     // Pick a random node.
68     UInt32 index=random.GetUInt32(mRoot->CountNodes());
69     mRoot->GetNodeIndex(index).Mutate(random);
70 }
71
72 void CStrategy::Crossover(CStrategy& second,CRandom& random,UInt32 count)
73 {
74     mFitness=kUnknownFitness;
75     // Do not perform crossover on the same tree... This could be bad.
76     if (this!=&second)
77     {
78         // Pick a random branch in each tree.
79         UInt32 branchPos1=GetRandomBranch(mRoot,random,count);
80         UInt32 branchPos2=GetRandomBranch(second.mRoot,random,count);
81         CNode*& branch1=mRoot->GetBranchIndex(branchPos1,mRoot);
82         CNode*& branch2=second.mRoot->GetBranchIndex(branchPos2,second.mRoot);
83         // Swap the subtrees.
84         Swap(branch1,branch2);
85     }
86 }
87
88 UInt32 CStrategy::GetRandomBranch(CNode*& root,CRandom& random,UInt32 tries)
89 {
90     UInt32 count=root->CountNodes();
91     UInt32 bestIndex=random.GetUInt32(count);
92     UInt32 bestPos=root->GetNodeDepth(bestIndex);
93     // Try a total of 4 nodes this gives us about a 20% chance of picking a leaf node.
94     // It favors nodes up the tree in this manor.
95     // If tries is 1, then we will just randomly select a node.
96     for (UInt32 i=1;i<tries;i++)
97     {
98         UInt32 challengerIndex=random.GetUInt32(count);
99         UInt32 challengerPos=root->GetNodeDepth(challengerIndex);
100        if (challengerPos<bestPos)
101        {
102            bestIndex=challengerIndex;
103            bestPos=challengerPos;

```

```
104     }
105 }
106 return bestIndex;
107 }
108
109 UInt32 CStrategy::GetFitness() const
110 {
111     if (mFitness==kUnknownFitness)
112         mFitness=CalculateFitness();
113     return mFitness;
114 }
115
116 bool CStrategy::IsTooBig() const
117 {
118     UInt32 count=mRoot->CountNodes();
119     return (count>kMaxNodeCount);
120 }
```