CHAPTER 5

Building a Better Genetic Algorithm

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots.

So far, the Universe is winning.

Richard Cook

By now, I hope you are starting to get a feel for the mechanism of genetic algorithms. If not, review the last couple of chapters and play around with the code some more. I cannot stress enough how important it is to play around with code. It's a bit like when you learned how to program. Remember those days? Remember how much you learned through a hands-on approach, rather than just sitting down with a heavy programming book? Well genetic algorithms (and neural networks) are very much like that. You learn much faster by writing your own code and experimenting with all the different parameters because you develop a feel for what works and what doesn't. This "feel" is very important because, to date, there are few hard and fast rules about genetic algorithms—they are as much an art as they are science. It's only with time and experimentation that you will learn what the right population size should be for a problem, just how high the mutation rate should be set, and so on.

This chapter is all about experimentation. First, I want to get you used to looking at operators and thinking about what changes you can make to improve them, and then I'll discuss some additional techniques that may improve the performance of your genetic algorithm, such as various fitness scaling techniques. I say *may* improve the performance of your genetic algorithm because every problem is different and a technique that helps one problem may actually *hinder* another. After you've tackled a few problems of your own, though, you'll get to know pretty quickly which techniques are appropriate for which problems. It's all about that "feel" I mentioned earlier.

Just to see how much the techniques covered here are going to help solve the TSP from the last chapter, check out the executable for this chapter on the CD (Figure 5.1 shows a screenshot). Big improvement, huh? Anyway, now I've given you a taste for what's to come; let's get on with the theory.

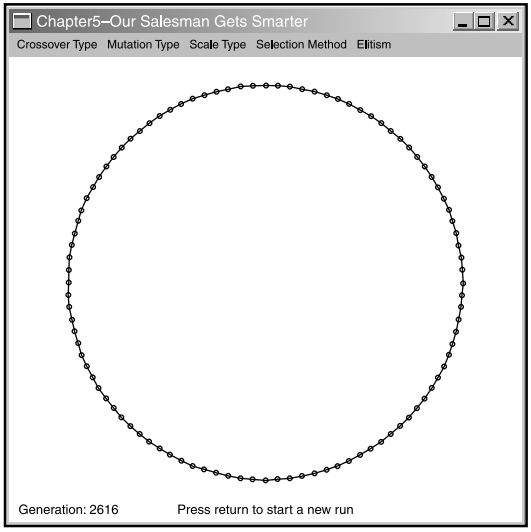

Figure 5.1

The salesman gets smart.

Alternative Operators for the TSP

The first topic I'm going to cover will be a discussion of those alternative mutation and crossover operators for the traveling salesman problem. Although none of them will improve the algorithm by a staggering amount, I feel I should spend a little time going over the more common ones because it's interesting to see how many different ways there are of approaching the problem of retaining valid permutations. Also, some of them give very interesting and thought provoking results when you watch the TSP algorithm in progress. More importantly, though, it will teach you that for every problem, there can be a multitude of ways to code the operators. Again—and I know I keep saying this—please make sure you play around with different operators to see how they perform. You will learn a lot. Hell, even go one step further and try to invent your own operators! That might prove a little tricky for the crossover operator for the TSP, but I bet you could think of a novel mutation operator, at the very least.

Alternative Permutation Mutation Operators

There have been many alternative mutation operators dreamed up by enthusiastic genetic algorithm researchers for the TSP. Here are descriptions of a few of the best followed by their code implementations.

Scramble Mutation (SM)

Choose two random points and "scramble" the cities located between them.

0 . 1 . 2 . **3** . **4** . **5** . **6** . 7

becomes

0 . 1 . 2 . **5** . **6** . **3** . **4** . 7

Here's what the code looks like.

```
void CgaTSP::MutateSM(vector<int> &chromo)
{
   //return dependent upon mutation rate
   if (RandFloat() > m_dMutationRate) return;

   //first we choose a section of the chromosome
   const int MinSpanSize = 3;

   //these will hold the beginning and end points of the span
   int beg, end;

   ChooseSection(beg, end, chromo.size()-1, MinSpanSize);
```

`ChooseSection` is a small function which determines a random start and end point to a span given a minimum span size and maximum span size. Please see the source code on the CD if further clarification is required.

```
   int span = end - beg;

   //now we just swap randomly chosen genes with the beg/end
   //range a few times to scramble them
   int NumberOfSwapsRqd = span;

   while(--NumberOfSwapsRqd)
   {
       vector<int>::iterator gene1 = chromo.begin();
       vector<int>::iterator gene2 = chromo.begin();

       //choose two loci within the range
       advance(gene1, beg + RandInt(0, span));
```

```
        advance(gene2, beg + RandInt(0, span));

        //exchange them
        swap(*gene1, *gene2);

    }//repeat
}
```

STL Note

erase()

`erase()` is a method for some STL containers that enables you to remove elements from a container. You can either just pass `erase()` a single element position (as an iterator)

```
//create an iterator pointing to the first element
vector<elements>::iterator beg = vecElements.begin();

//erase the first element
vecElements.erase(beg);
```

or you can pass `erase()` a range to remove. The range is defined by start and end iterators. So, to remove the first to the third element of an `std::vector`, you would do this:

```
vector<elements>::iterator beg = vecElements.begin();
vector<elements>::iterator end = beg + 3;
vecElements.erase(beg, end);
```

insert()

`insert()` is a method that enables you to insert elements into a container. As with `erase()`, you can choose to insert a single element at a position pointed to by an iterator or you can insert a range of elements. Here is a simple example, which inserts the first four elements in `vecInt1` at position five in `vecInt2`.

```
vector<int> vecInt1, vecInt2;

for (int i=0; i<10; ++i)
{
  vecInt1.push_back(i);
  vecInt2.push_back(i);
}

vector<int>::iterator RangeStart = vecInt1.begin();
```

```
vector<int>::iterator InsertPos  = vecInt2.begin()+5;

vecInt2.insert(InsertPos, RangeStart, RangeStart+4);
```

assign()

`assign()` is a method that enables you to assign a range of elements in one container to another container. For example, if you had the `std::vector` of `ints`, `vecInts`, which contained all the integers from 0 to 9 and you wanted to create a new `std::vector` containing the range of integers from positions 3 to 6, you could do it like this:

```
vector<int>::iterator RangeStart = vecInt.begin() + 3;
vector<int>::iterator RangeEnd = vecInt.begin() + 6;

vector<int> newVec;
newVec.assign(RangeStart, RangeEnd);
```

You can also use it to add an element `n` number of times to an `std::vector`. The following example adds six copies of the integer 999 to the `std::vector`, `vecInts`.

```
vector<int> vecInt;
vecInt.assign(6, 999);
```

advance()

`advance()` is a handy method that enables you to advance an iterator by a required number of positions. To use it, you just pass `advance()` an iterator and the number of element positions it's to be advanced.

```
vector<int>::iterator RangeStart = vecInt.begin();
advance(RangeStart, 3);
```

sort()

To sort all the elements in a container, you can use `sort`, which will sort all the elements in a given range, like so:

```
sort(vecGenomes.begin(), vecGenomes.end());
```

This will only work provided some sorting criteria has been defined for the elements to be sorted. In the TSP program, I have provided a `<` overload for the `SGenome` struct, so `SGenomes` will be sorted by the member variable `dFitness`. It looks like this:

```
friend bool operator<(const SGenome& lhs, const SGenome& rhs)
{
   return (lhs.dFitness < rhs.dFitness);
}
```

Displacement Mutation (DM)

Select two random points, grab the chunk of chromosome between them, and then reinsert at a random position displaced from the original.

0 . 1 . 2 . **3** . **4** . **5** . 6 . 7

becomes

0 . **3** . **4** . **5** . 1 . 2 . 6 . 7

This is particularly interesting to watch because it helps the genetic algorithm converge to a short path very quickly, but then takes a while to actually go that few steps further to get to the solution.

```
void CgaTSP::MutateDM(vector<int> &chromo)
{
   //return dependent upon mutation rate
   if (RandFloat() > m_dMutationRate) return;

   //declare a minimum span size
   const int MinSpanSize = 3;

   //these will hold the beginning and end points of the span
   int beg, end;

  //choose a section of the chromosome.
   ChooseSection(beg, end, chromo.size()-1, MinSpanSize);

   //setup iterators for the beg/end points
   vector<int>::iterator SectionStart = chromo.begin() + beg;
   vector<int>::iterator SectionEnd   = chromo.begin() + end;

   //hold on to the section we are moving
   vector<int> TheSection;
   TheSection.assign(SectionStart, SectionEnd);

   //erase from current position
   chromo.erase(SectionStart, SectionEnd);

   //move an iterator to a random insertion location
```

```
    vector<int>::iterator curPos;
    curPos = chromo.begin() + RandInt(0, chromo.size()-1);

    //re-insert the section
    chromo.insert(curPos, TheSection.begin(), TheSection.end());
}
```

Insertion Mutation (IM)

This is a very effective mutation and is almost the same as the DM operator, except here only one gene is selected to be displaced and inserted back into the chromosome. In tests, this mutation operator has been shown to be consistently better than any of the alternatives mentioned here.

0 . 1 . **2** . 3 . 4 . 5 . 6 . 7

becomes

0 . 1 . 3 . 4 . 5 . **2** . 6 . 7

I use insertion mutation as the default mutation operator in the code project for this chapter.

```
void CgaTSP::MutateIM(vector<int> &chromo)
{
    //return dependent upon mutation rate
    if (RandFloat() > m_dMutationRate) return;

    //create an iterator for us to work with
    vector<int>::iterator curPos;

    //choose a gene to move
    curPos = chromo.begin() + RandInt(0, chromo.size()-1);

    //keep a note of the genes value
    int CityNumber = *curPos;

    //remove from the chromosome
    chromo.erase(curPos);

    //move the iterator to the insertion location
```

```
    curPos = chromo.begin() + RandInt(0, chromo.size()-1);

    chromo.insert(curPos, CityNumber);
}
```

Inversion Mutation (IVM)

This is a very simple mutation operator. Select two random points and reverse the cities between them.

0 . **1** . **2** . **3** . **4** . 5 . 6 . 7

becomes

0 . **4** . **3** . **2** . **1** . 5 . 6 . 7

Displaced Inversion Mutation (DIVM)

Select two random points, reverse the city order between the two points, and then displace them somewhere along the length of the original chromosome. This is similar to performing IVM and then DM using the same start and end points.

0 . 1 . 2 . 3 . **4** . **5** . **6** . 7

becomes

0 . **6** . **5** . **4** . 1 . 2 . 3 . 7

I'll leave the implementation of these last two mutation operators as an exercise for you to code. (That's my crafty way of getting you to play around with the source!)

Alternative Permutation Crossover Operators

As with mutation operators, inventing crossover operators that spawn valid permutations has been a popular sport amongst genetic algorithm enthusiasts. Here are the descriptions and code for a couple of the better ones.

Order-Based Crossover (OBX)

To perform order-based crossover, several cities are chosen at random from one parent and then the *order* of those cities is imposed on the respective cities in the other parent. Let's take the example…

Parent1: 2 . **5** . **0** . 3 . 6 . **1** . 4 . 7
Parent2: 3 . 4 . 0 . 7 . 2 . 5 . 1 . 6

The cities in bold are the cities which have been chosen at random. Now, impose the order—5, 0, then 1—on the same cities in Parent2 to give Offspring1 like so:

Offspring1: 3 . 4 . **5** . 7 . 2 . **0** . **1** . 6

City one stayed in the same place because it was already positioned in the correct order. Now the same sequence of actions is performed on the other parent. Using the same positions as the first,

Parent1: 2 . 5 . 0 . 3 . 6 . 1 . 4 . 7
Parent2: 3 . **4** . **0** . 7 . 2 . **5** . 1 . 6

Parent1 becomes:

Offspring2: 2 . **4** . **0** . 3 . 6 . 1 . **5** . 7

Here is order-based crossover implemented in code:

```
void CgaTSP::CrossoverOBX(const vector<int>    &mum,
                          const vector<int>    &dad,
                          vector<int>          &baby1,
                          vector<int>          &baby2)
{
  baby1 = mum;
```

```
baby2 = dad;

//just return dependent on the crossover rate or if the
//chromosomes are the same.
if ( (RandFloat() > m_dCrossoverRate) || (mum == dad))
{
  return;
}

//holds the chosen cities
vector<int> tempCities;

//holds the positions of the chosen cities
vector<int> positions;

//first chosen city position
int Pos = RandInt(0, mum.size()-2);

//keep adding random cities until we can add no more
//record the positions as we go
while (Pos < mum.size())
{
  positions.push_back(Pos);

  tempCities.push_back(mum[Pos]);

  //next city
  Pos += RandInt(1, mum.size()-Pos);
}

//so now we have n amount of cities from mum in the tempCities
//vector we can impose their order in dad.
int cPos = 0;

for (int cit=0; cit<baby2.size(); ++cit)
{
  for (int i=0; i<tempCities.size(); ++i)
  {
    if (baby2[cit]==tempCities[i])
```

```
        {
          baby2[cit] = tempCities[cPos];

          ++cPos;

          break;
        }
      }
    }

    //now vice versa. Choose the same positioned cities from dad and impose
    //their order in mum
    tempCities.clear();
    cPos = 0;

    //first grab the cities from the same positions in dad
    for(int i=0; i<positions.size(); ++i)
    {
      tempCities.push_back(dad[positions[i]]);
    }

    //and impose their order in mum
    for (cit=0; cit<baby1.size(); ++cit)
    {
      for (int i=0; i<tempCities.size(); ++i)
      {
        if (baby1[cit]==tempCities[i])
        {
          baby1[cit] = tempCities[cPos];

          ++cPos;

          break;
        }
      }
    }
  }
```

TIP

Often, when you want to find the optimum route for a unit in a game, it's not desirable to just take into account the distances involved. For example, say your game is based around a 3D terrain engine. You will probably want to consider such factors as the gradients encountered during the route (because units moving uphill usually travel slower and use more fuel) and also the surfaces the unit will be traveling over. (Moving through mud is a lot slower than moving across asphalt.)

To find the optimal route, a fitness function, which takes into account all these factors, must be defined. This way, you get the best trade off between distance covered and the surfaces and gradients traveled over. For example, create a sliding scale of penalties for all the different surfaces in your game. The slower the surface is to travel over, the higher the score your unit gets when you calculate distances between waypoints (remember, high values when calculating the distances convert to poor fitness scores). And similarly with the gradients, penalize for ascents and reward for descents. It may take a little fiddling to get the balance right, but eventually you will end up with a genetic algorithm that finds the *optimum* path for each different kind of unit, rather than just the *shortest* path.

Position-Based Crossover (PBX)

This is similar to Order-Based Crossover, but instead of imposing the order of the cities, this operator imposes the *position*. So, using the same example parents and random positions, here's how to do it.

Parent1: 2 . **5** . **0** . 3 . 6 . **1** . 4 . 7
Parent2: 3 . **4** . **0** . 7 . 2 . **5** . 1 . 6

First, move over the selected cities from Parent1 to Offspring1, keeping them in the same position.

OffSpring1: * . **5** . **0** . * . * . **1** . * . *

Now, iterate through Parent2's cities and fill in the blanks if that city number has not already appeared. In this example, filling in the blanks results in:

Offspring1: 3 . **5** . **0** . 4 . 7 . **1** . 2 . 6

Get it? Let's run through the derivation of Offspring2, just to be sure. First, copy over the selected cities into the same positions.

Offspring2: * . **4** . **0** . * . * . **5** . * . *

Now, fill in the blanks.

Offspring2: 2 . **4** . **0** . 3 . 6 . **5** . 1 . 7

And here's how it looks in code:

```
void CgaTSP::CrossoverPBX(const vector<int>    &mum,
                          const vector<int>    &dad,
                          vector<int>          &baby1,
                          vector<int>          &baby2)
{
  //Return dependent on the crossover rate or if the
  //chromosomes are the same.
  if ( (RandFloat() > m_dCrossoverRate) || (mum == dad))
  {
    //make sure baby1 and baby2 are assigned some cities first!
    baby1 = mum;
```

```
  baby2 = dad;

  return;
}

//initialize the babies with minus values so we can tell which positions
//have been filled later in the algorithm
baby1.assign(mum.size(), -1);
baby2.assign(mum.size(), -1);

int l = baby2.size();

//holds the positions of the chosen cities
vector<int> positions;

//first city position
int Pos = RandInt(0, mum.size()-2);

//keep adding random cities until we can add no more
//record the positions as we go
while (Pos < mum.size())
{
  positions.push_back(Pos);

  //next city
  Pos += RandInt(1, mum.size()-Pos);
}

//now we have chosen some cities it's time to copy the selected cities
//over into the offspring in the same position.
for (int pos=0; pos<positions.size(); ++pos)
{
  //baby1 receives from mum
  baby1[positions[pos]] = mum[positions[pos]];

  //baby2 receives from dad
  baby2[positions[pos]] = dad[positions[pos]];
}

//fill in the blanks. First create two position markers so we know
```

```
    //whereabouts we are in baby1 and baby2
    int c1 = 0, c2 = 0;

    for (pos=0; pos<mum.size(); ++pos)
    {
      //advance position marker until we reach a free position
      //in baby2
      while( (baby2[c2] > -1) && (c2 < mum.size()))
      {
        ++c2;
      }

      //baby2 gets the next city from mum which is not already
      //present
      if ( (!TestNumber(baby2, mum[pos])) )
      {
        baby2[c2] = mum[pos];
      }

      //now do the same for baby1
      while((baby1[c1] > -1) && (c1 < mum.size()))
      {
        ++c1;
      }

      //baby1 gets the next city from dad which is not already
      //present
      if ( (!TestNumber(baby1, dad[pos])) )
      {
        baby1[c1] = dad[pos];
      }
    }
}
```

Now that you've seen how others have tackled the crossover operator, can you dream up one of your own? This is not an easy task, so congratulations if you can actually invent one!

I hope running through a few of the alternative operators for the traveling salesman problem has given you an indication of the scope you can have with genetic algo-

rithm operators. For the remainder of this chapter, though, I'm going to talk about various tools and techniques you can apply to just about any kind of genetic algorithm to improve its performance.

The Tools of the Trade

Envision the complete set of possible solutions to a problem as a kind of landscape that dips and rises as you travel across it. The lower the ground, the more "fit" is the solution represented at that point. Conversely, the high ground represents extremely poor solutions. Imagine the genetic algorithm as a ball that rolls around on that landscape until it falls into a trough. As I've explained, this would represent a solution, but, it may not be the *best* solution. Only now, the ball is stuck and cannot roll any further. This is what is known as a *local minima*. Figure 5.2 shows you what I mean.

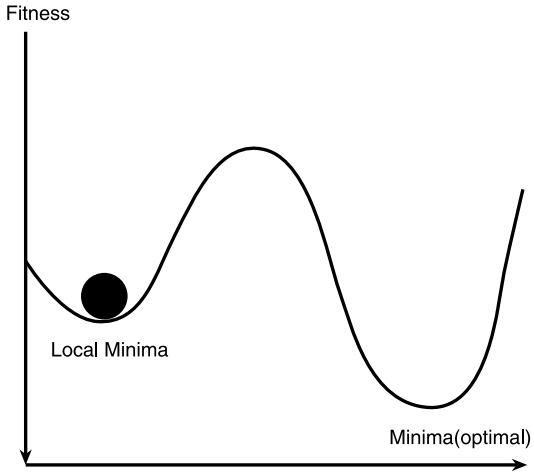

Figure 5.2

A GA stuck in a local minima.

Ideally, you want the ball to roll over as much landscape as possible, until it finds the deepest trough to fall into. This represents the best solution. Or, failing that, at least provide some way of "kicking" the ball out of the shallow troughs so it can continue its journey across the landscape.

NOTE

In some texts, you will find the fitness landscape inverted and, therefore, the author may refer to an algorithm getting stuck at a local *maxima*. Either way, the concept is the same.

Figure 5.2 shows the fitness landscape for a problem with just one parameter that needs solving. A two-parameter fitness landscape would be in 3D, as shown in Figure 5.3.

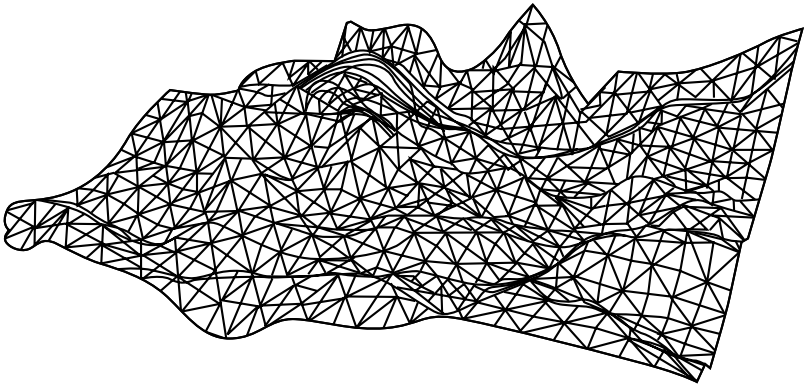

Figure 5.3

A two-parameter fitness landscape.

The x- and z-axis represent the parameters, and the y-axis represents the fitness. Now, there are all sorts of hills, troughs, ridges, and other features for the genetic algorithm to negotiate before it can settle at the optimum. And that's just with two parameters! When you go above two parameters, you have to let your imagination go wild because you've entered the incredible domain of mathematical hyperspaces. However, the concept is just the same and there will be hills and troughs in the landscape, just the same. (If Dali had still been alive, I would have asked Premier Press to hire him to draw you a diagram of hyperspace, but unfortunately, you'll have to make do with your imagination.)

To keep the ball rolling, you need to equip yourself with tools you can use to cajole your genetic algorithms into doing what you want. In this section, I'm going to spend some time discussing various techniques and additional operators you can use to help your genetic algorithms converge on a solution more efficiently.

Selection Techniques

In junior high, I was useless at most sports, particularly team sports, and for ages, I had to suffer the daily playground humiliation of being the last to be chosen for a game of soccer. Remember how all the kids would stand in a line and the two most athletic boys in the school would, as captains, take turns selecting their team? Well, I was the boy who was always chosen last and put safely out of the way in the goalie position. Imagine my relief when after a couple of years of this, an even geekier kid moved into town. Oh what joy!

Selection is how you choose individuals from the population to provide a gene base from which the next generation of individuals is created. This might mean individuals are selected and placed into the new generation without modification ala elitism, as we discussed in the last chapter, but usually it means the chosen genomes are selected to be parents of offspring which are created through the processes of mutation and recombination. How you go about choosing the parents can play a very important role in how efficient your genetic algorithm is. Unlike choosing a soccer team, if you choose the fittest individuals all the time, the population may converge too rapidly at a local minima and get stuck there. But, if you select individuals at random, then your genetic algorithm will probably take a while to converge (if it ever does at all). So, the art of selection is choosing a strategy which gives you the best of both worlds—something that converges fairly quickly yet enables the population to retain its diversity.

Elitism

As previously discussed, elitism is a way of guaranteeing that the fittest members of a population are retained for the next generation. In the last chapter, the code example used a little bit of elitism to select two copies of the best individual to go through to the next generation. To expand on this, it can be better to select `n` copies of the top `m` individuals of the population to be retained. I often find that retaining about 2-5% of the population size gives me good results. The function name I give for this expanded version of elitism is called `GrabNBest`. Its prototype looks like this:

```
void    GrabNBest(int              NBest,
                  const int        NumCopies,
                  vector<SGenome> &vecNewPop);
```

So, to retain two copies each of the fittest three members of the population, you would call

```
GrabNBest(3, 2, vecNewPop);
```

When you play around with the example program, you will discover that using elitism works well with just about every other technique described in this chapter (except stochastic universal sampling, which will be discussed soon).

Steady State Selection

Steady state selection works a little like elitism, except that instead of choosing a small amount of the best individuals to go through to the new generation, steady

state selection retains all but a few of the worst performers from the current population. The remainder are then selected using mutation and crossover in the usual way. Steady state selection can prove useful when tackling some problems, but most of the time it's inadvisable to use it.

Fitness Proportionate Selection

Selection techniques of this type choose offspring using methods which give individuals a better chance of being selected the better their fitness score. Another way of describing it is that each individual has an expected number of times it will be chosen to reproduce. This expected value equates to the individual's fitness divided by the average fitness of the entire population. So, if you have an individual with a fitness of 6 and the average fitness of the overall population is 4, then the expected number of times the individual should be chosen is 1.5.

Roulette Wheel Selection

A common way of implementing fitness proportionate selection is roulette wheel selection, as I have already discussed. This technique does have its drawbacks, however. Because roulette wheel selection is based on using random numbers and because the population sizes of genetic algorithms are typically small (sizes between 50 and 200 are common), the number of children allocated to each individual can be far from its expected value. Even worse, it's probable that roulette wheel selection could miss the best individuals altogether! This is one of the reasons elitism is a good idea when utilizing roulette wheel selection—it ensures you never lose the best individuals to chance.

Stochastic Universal Sampling

Stochastic Universal Sampling (SUS for short) is an attempt to minimize the problems of using fitness proportionate selection on small populations. Basically, instead of having one wheel which is spun several times to obtain the new population, SUS uses `n` evenly spaced hands, which are only spun *once* as shown in Figure 5.4. The amount of pointers is equal to the amount of offspring required.

Here is the code which implements this type of sampling:

```
void CgaTSP::SUSSelection(vector<SGenome> &NewPop)
{
  //this algorithm relies on all the fitness scores to be positive so
  //these few lines check and adjust accordingly (in this example
```

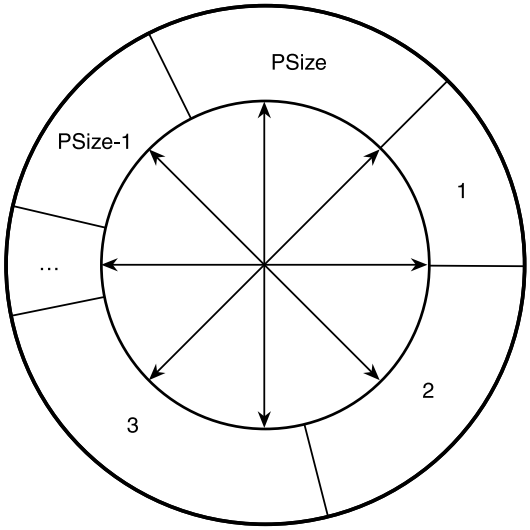

Figure 5.4

The SUS wheel of probability.

```
//Sigma scaling can give negative fitness scores
if (m_dWorstFitness < 0)
{
  //recalculate
  for (int gen=0; gen<m_vecPopulation.size(); ++gen)
  {
    m_vecPopulation[gen].dFitness += fabs(m_dWorstFitness);
  }

  CalculateBestWorstAvTot();
}
```

Some of the scaling techniques discussed in this chapter can result in negative fitness scores for some of the population. The preceding lines of code check for this possibility and readjust the scores accordingly. If you know for sure your fitness scores will never be negative, you can omit this.

```
int curGen = 0;
double sum = 0;

//NumToAdd is the amount of individuals we need to select using SUS.
//Remember, some may have already been selected through elitism
int NumToAdd = m_iPopSize - NewPop.size();

//calculate the hand spacing
```

```
  double PointerGap = m_dTotalFitness/(double)NumToAdd;

  //choose a random start point for the wheel
  float ptr = RandFloat() * PointerGap;

  while (NewPop.size() < NumToAdd)
  {
    for(sum+=m_vecPopulation[curGen].dFitness; sum > ptr; ptr+=PointerGap)
    {
      NewPop.push_back(m_vecPopulation[curGen]);

      if( NewPop.size() == NumToAdd)
      {
        return;
      }
    }

    ++curGen;
  }
}
```

If you use SUS in your own genetic algorithms, it is inadvisable to use elitism with it because this tends to mess up the algorithm. You will clearly see the effect that switching elitism on or off has on SUS selection when you run this chapter's executable.

Tournament Selection

To use tournament selection, `n` individuals are selected at random from the population, and then the fittest of these genomes is chosen to add to the new population. This process is repeated as many times as is required to create a new population of genomes. Any individuals selected are *not* removed from the population and therefore can be chosen any number of times. Here's what this algorithm looks like in code.

```
SGenome& CgaTSP::TournamentSelection(int N)
{
  double BestFitnessSoFar = 0;

  int ChosenOne = 0;

  //Select N members from the population at random testing against
```

```
  //the best found so far
  for (int i=0; i<N; ++i)
  {
    int ThisTry = RandInt(0, m_iPopSize-1);

    if (m_vecPopulation[ThisTry].dFitness > BestFitnessSoFar)
    {
      ChosenOne = ThisTry;

      BestFitnessSoFar = m_vecPopulation[ThisTry].dFitness;
    }
  }
  //return the champion
  return m_vecPopulation[ChosenOne];
}
```

This technique is very efficient to implement because it doesn't require any of the preprocessing or fitness scaling sometimes required for roulette wheel selection and other fitness proportionate techniques (discussed later in the chapter). Because of this, and because it's a darn good technique anyway, you should always try this method of selection with your own genetic algorithms. The only drawback I've found is that tournament selection can lead to too quick convergence with some types of problems.

I've also seen an alternative description of this technique, which goes like this: A random number is generated between 0 and 1. If the random number is less than a pre-determined constant, for example cT (a typical value would be 0.75), then the fittest individual is chosen to be a parent. If the random number is greater than cT, then the weaker individual is chosen. As before, this is repeated until a new population of the correct size has been spawned.

> **Interesting Fact**
>
> NASA has used genetic algorithms to successfully calculate low altitude satellite orbits and more recently to calculate the positioning of the Hubble space telescope.

Scaling Techniques

Although using selection on the raw (unprocessed) fitness scores can give you a genetic algorithm that works (it solves the task you've designed it for), often your

genetic algorithm can be made to perform better if the fitness scores are *scaled* in some way before any selection takes place. There are various ways of doing this and I'm going to spend the next few pages describing the best of the bunch.

Rank Scaling

Rank scaling can be a great way to prevent too quick convergence, particularly at the start of a run when it's common to see a very small percentage of individuals outperforming all the rest.

The individuals in the population are simply ranked according to fitness, and then a new fitness score is assigned based on their rank. So, for example, if you had a population of five individuals with the fitness scores shown in Table 5.1, all you do is sort them and assign a new fitness based on their rank within the sorted population. See Table 5.2

Once the new ranked fitness scores have been applied, you select individuals for the next generation using roulette wheel selection or a similar fitness proportionate selection method. (Please note you would never, in practice, have a population of just five, I'm just using five to demonstrate the principle).

This technique avoids the possibility that a large percentage of each new generation is being produced from a very small number of highly fit individuals, which can quickly lead to premature convergence. In effect, rank scaling ensures your population remains diverse. The other side of the coin is that the population may take a lot longer to converge, but often you will find that the greater diversity provided by this technique leads to a more successful result for your genetic algorithm.

Table 5.1 Fitness Scores Before Ranking

Individual	Score
1	3.4
2	6.1
3	1.2
4	26.8
5	0.7

Table 5.2 Fitness Scores After Ranking

Individual	Old Fitness	New Fitness
4	26.8	5
2	6.1	4
1	3.4	3
3	1.2	2
5	0.7	1

Sigma Scaling

If you use raw fitness scores as a basis for selection, the population may converge too quickly, and if they are scaled as in rank selection, the population may converge too slowly. Sigma scaling is an attempt to keep the selection pressure constant over many generations. At the beginning of the genetic algorithm, when fitness scores can vary wildly, the fitter individuals will be allocated less expected offspring. Toward the end of the algorithm, when the fitness scores are becoming similar, the fitter individuals will be allocated more expected offspring.

The formula for calculating each new fitness score using sigma scaling is:

if $\sigma = 0$ then the fitness = 1

else

$$\text{NewFitness} = \frac{\text{OldFitness} - \text{AverageFitness}}{2\sigma}$$

where the Greek letter sigma, σ, represents the *standard deviation* of the population.

A few of you will probably be wondering what the standard deviation is and how it's calculated. Well, the standard deviation is the square root of the population's *variance*. The variance is a measure of spread within the fitness scores. Figure 5.5 shows an example of a population with a low variance.

The hump in the middle of the graph represents the mean (the average) fitness score. Most of the population's scores are clustered around this hump. The spread, or variance, is the width of the hump at the base. Figure 5.6 shows a population with a high variance, and as you can see, the hump is lower and more spread out.

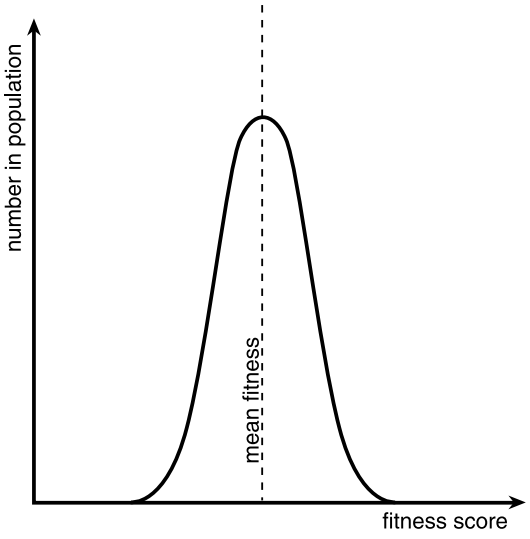

Figure 5.5

Population with a low spread.

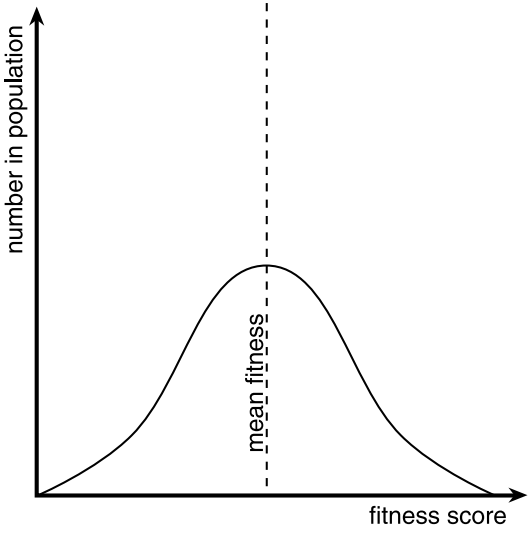

Figure 5.6

Population with a high spread.

Now that you know what variance is, let me show you how to calculate it. Imagine we are only dealing with a population of three, and the fitness scores are 1, 2, and 3. To calculate the variance, first calculate the mean of all the fitness scores.

$$\text{mean} = \frac{1 + 2 + 3}{3} = 2$$

Then the variance is calculated like this

$$\text{variance} = \frac{(1-2)^2 + (2-2)^2 + (3-2)^2}{3} = 0.667$$

A more mathematical way of writing this is

$$\text{variance} = \frac{\sum (f - mf)^2}{N}$$

Where `f` is the fitness of the current individual, `fm` is the average fitness of the population, and `N` is the population size. The weird Greek symbol Σ is also called sigma but it's the capital of σ, just like A is the capital of a. The Σ symbol is a summation symbol, and in this example it indicates that all the values of

$$(f - mf)^2$$

should be summed before being divided by `N`.

Once the variance has been calculated, it's a trivial matter to compute the square root to give the standard deviation:

$$\sigma = \sqrt{\text{variance}}$$

The code for applying sigma scaling to the traveling salesman problem looks like this:

```
void CgaTSP::FitnessScaleSigma(vector<SGenome> &pop)
{
  double RunningTotal = 0;

  //first iterate through the population to calculate the standard
  //deviation
  for (int gen=0; gen<pop.size(); ++gen)
  {
    RunningTotal += (pop[gen].dFitness - m_dAverageFitness) *
                    (pop[gen].dFitness - m_dAverageFitness);
  }

  double variance = RunningTotal/(double)m_iPopSize;

  //standard deviation is the square root of the variance
  m_dSigma = sqrt(variance);

  //now iterate through the population to reassign the fitness scores
```

```
  for (gen=0; gen<pop.size(); ++gen)
  {
    double OldFitness = pop[gen].dFitness;

    pop[gen].dFitness = (OldFitness - m_dAverageFitness) /
                        (2 * m_dSigma);
  }

  //recalculate values used in selection
  CalculateBestWorstAvTot();
}
```

The last call to `CalculateBestWorstAvTot` is there to recalculate all the best, worst, and average values for the entire population, which some of the selection types use. `m_dSigma` is a member variable because it can be used to stop the run if the variance becomes zero (all the fitness scores are therefore identical and so there is not much point continuing). There are a few areas in which this function could be speeded up, but I've written it like this so that it follows the equations more literally.

Sigma scaling is interesting to watch in action because the population converges very quickly in the first few generations, but then takes a long time to finally reach a solution.

Interesting Fact

The bots created for the game Quake3 were developed using genetic algorithms. A genetic algorithm was used to optimize the fuzzy logic controllers for each bot. Briefly put, fuzzy logic is logic extended to encompass partial truths. So, instead of something having to be black or white, as in conventional logic, when using fuzzy logic, something can be shades of gray too. The Quake3 bot uses fuzzy logic to indicate *how much* it wants to do something. It doesn't just indicate that it wants to pick a certain item; using fuzzy logic it can determine that it is in 78% favor of picking up the railgun and 56% in favor of picking up the armor.

Boltzmann Scaling

You've learned how to keep the selection pressure constant over a run of your genetic algorithm by using sigma scaling, but sometimes you may want the selection pressure to vary. A common scenario is one in which you require the selection pressure to be low at the beginning so that diversity is retained, but as the genetic algorithm converges closer toward a solution, you want mainly the fitter individuals to produce offspring.

One way of achieving this is by using *Boltzmann scaling*. This method of scaling uses a continuously varying *temperature* to control the rate of selection. The formula is

$$\text{NewFitness} = \frac{\text{OldFitness} / \text{Temperature}}{\text{AverageFitness} / \text{Temperature}}$$

Each generation, the temperature is decreased by a small value, which has the effect of increasing the selection pressure toward the fitter individuals. This is the code implementation of Boltzmann scaling from the TSP project.

```
void CgaTSP::FitnessScaleBoltzmann(vector<SGenome> &pop)
{

  //reduce the temp a little each generation
  m_dBoltzmannTemp -= BOLTZMANN_DT;

  //make sure it doesn't fall below minimum value
  if (m_dBoltzmannTemp< BOLTZMANN_MIN_TEMP)
  {
    m_dBoltzmannTemp = BOLTZMANN_MIN_TEMP;
  }

  //first calculate the average fitness/Temp
  double divider = m_dAverageFitness/m_dBoltzmannTemp;

  //now iterate through the population and calculate the new expected
  //values
  for (int gen=0; gen<pop.size(); ++gen)
  {
    double OldFitness = pop[gen].dFitness;

    pop[gen].dFitness = (OldFitness/m_dBoltzmannTemp)/divider;
  }

  //recalculate values used in selection
  CalculateBestWorstAvTot();
}
```

In the TSP solver, the temperature is initially set to twice the number of cities. `BOLTZMANN_DT` is `#defined` as 0.05 and `BOLTZMANN_MIN_TEMP` is `#defined` as 1.

Alternative Crossover Operators

Utilizing different crossover and mutation operators for your genetic algorithms can often be a good idea. How you implement these operators depends very much on how your problem is encoded. As you've already seen, using a crossover operator that works well for one type of problem may have disastrous results when applied to another. The same goes for mutation operators. Although, for most genome encodings you are limited in what you can do with a mutation operator—it's typically such a simple operation—there are usually a few different ways of performing crossover. Here are explanations of the most popular types.

Single-Point Crossover

This is the first crossover operator I introduced you to in Chapter 3, "An Introduction To Genetic Algorithms." It simply cuts the genome at some random point and then switches the ends between parents. It is very easy and quick to implement and is generally effective to some degree with most types of problems.

Two-Point Crossover

Instead of cutting the genome at just one point, two-point crossover (you guessed it) cuts the genome at two random points and then swaps the block of genes between those two points. So, if you had two binary encoded parents like this,

Parent1: 1010001010
Parent2: 1101110101

and the chosen crossover points were after the third and seventh genes, two-point crossover would go like this.

Parent1: 101 **0001** 010
Parent2: 110 **1110** 101

Swap the "belly" block of genes giving offspring.

Child1: 101 **1110** 010
Child2: 110 **0001** 101

See Figure 5.7 for an illustration of this process.

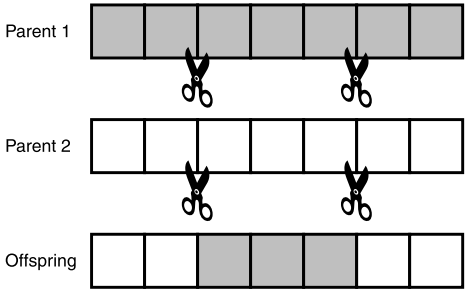

Figure 5.7
Two-point crossover.

Two-point crossover is sometimes beneficial because it can create combinations of genes that single-point crossover simply cannot provide. With single point, the end genes are *always* swapped over and this may not be favorable for the problem at hand. Two-point crossover eliminates this problem.

Multi-Point Crossover

Why stop at just two crossover points? There's no need to limit the amount of crossover points you can have. Indeed, for some types of encoding, your genetic algorithm may perform better if you use multiple crossover points. The easiest way of achieving this is to move down the length of the parents, and for each position in the chromosome, randomly swap the genes based on your crossover rate, as in Figure 5.8.

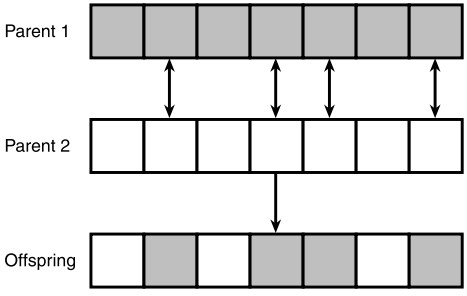

Figure 5.8
Multi-point crossover.

Here is what the code implementation of multi-point crossover looks like.

```
void CGenAlg::CrossoverMultiPoint(const vector<gene_type> &mum,
                                  const vector<gene_type> &dad,
                                  vector<gene_type>       &baby1,
                                  vector<gene_type>       &baby2)
{

  //iterate down the length of the genomes swapping genes
  //depending on the crossover rate
  for (int gen=0; gen<mum.size(); ++gen)
  {
    if (RandFloat() < CrossoverRate))
    {
      //swap the genes
      baby2.push_back(mum[gen]);
      baby1.push_back(dad[gen]);
    }

    else
    {
      //don't swap the genes
      baby1.push_back(mum[gen]);
      baby2.push_back(dad[gen]);
    }
  }
}
```

Sometimes you will see this type of crossover described as *parameterized uniform crossover*. I tend to favor the name multi-point crossover because it says exactly what it does.

For some types of problems, multi-point crossover works very well, but on others it can jumble up the genes too much and act more like an over enthusiastic mutation operator. Common values for the crossover rate using this type of crossover operator are between 0.5 and 0.8.

Niching Techniques

Niching is a way of keeping the population diverse by grouping similar individuals together. One of the most popular niching techniques is called *explicit fitness sharing*.

This is a method in which the individuals in the population are grouped together according to how similar their genomes are, and then the fitness score of each individual is adjusted by "sharing" it amongst that group's members. This ensures similar individuals in the population are punished, thereby retaining diversity. To clarify, let's take the example of a population of binary encoded genomes. You can measure the difference between two genomes by counting all the bits in the genome that match. For example, the genomes

Genome1: 1**0100**0101**0**
Genome2: 0**0100**1010**1**

match at five places shown in bold. You can say their compatibility score is 5. To group the population into niches of similar genomes, you just test each genome against a sample genome to obtain a compatibility score for each. Genomes with similar compatibility scores are grouped together and then their fitness score is adjusted by dividing the raw fitness by the size of that genome's niche. Table 5.3 should help make this clearer for you.

Table 5.3 Fitness Sharing In Action

Genome ID	Niche ID	Raw Fitness	Adj Fitness
1	1	16	16/3 = 5.34
5	1	6	6/3 = 2
6	1	10	10/3 = 3.34
3	2	6	6/1 = 6
2	3	9	9/3 = 3
4	3	10	10/3 = 3.34
8	3	20	20/3 = 6.67
9	4	3	3/2 = 1.5
10	4	6	6/2 = 3

As you can see, fitness sharing is very effective at penalizing similarly constructed genomes and can be a terrific way of making sure your population remains diverse. I'll be discussing niching techniques in more detail later on in the book.

Summing Up

By the time you've experimented with a few of the techniques described in this chapter, you will have developed a pretty good feel for what genetic algorithms are all about.

In the next few chapters, I'll often be using the simplest combination of genetic algorithm techniques possible to achieve the desired result. This will give you the opportunity to try out, first hand, the techniques you've looked at so far to see how they may aid or hinder the evolution of different types of problems.

Stuff to Try

1. Go back and apply what you have learned in this chapter to the Pathfinder problem discussed in Chapter 3, "An Introduction to Genetic Algorithms."
2. Can you create a genetic algorithm for solving the 8-puzzle? (The 8-puzzle is that puzzle in which you have to slide numbered tiles around in a tray until all the numbers appear in order. See Figure 5.9)
3. Create a genetic algorithm to calculate the combination of letters that will give the highest score possible on a Boggle board.

2	4	5
8	1	
6	3	7

Figure 5.9

An unsolved 8-puzzle board.