INTERNATIONAL JOURNAL OF ACADEMIC RESEARCH IN BUSINESS & SOCIAL SCIENCES

Vol 4, Issue 5, (2014) E-ISSN: 2222-6990

Shared Crossover Method for Solving Knapsack Problems

Omar. I .Lasassmeh- - Muta'h University
Anas. A. Kasassbeh-- Muta'h University
Almotaz. M .Mobaedeen- Muta'h University
Emad. A. Kasasbeh - University Malaysia Perlis

DOI Link: http://dx.doi.org/10.6007/IJARBSS/v4-i5/851

Published Date: 01 May 2014

Abstract

A surprising number of everyday problems are difficult to solve by traditional algorithm. A problem may qualify as difficult for a number of different reasons; for example, the data may be too noisy or irregular, the problem may be difficult to model; or it may simply take too long to solve. It's easy to find examples: finding the shortest path connecting a set of cities, dividing a set of different tasks among a group of people to meet a deadline, or fitting a set of various sized boxes into the fewest trucks. In the past, programmers might have carefully hand crafted a special purpose program for each problem; now they can reduce their time significantly using a Genetic Algorithm (GAs). A Genetic Algorithm is key to solve knapsack problem, the goal of this paper is to show that successful Genetic Algorithm for solving and implementation knapsack problem, Genetic Algorithms are stochastic whose search methods model some natural phenomena. Genetic algorithms are relatively easy for finding the optimal solution, or approximately optimum value of NP-Complete problems, the coding scheme I've chosen for the knapsack uses a fixed-length, binary, position-dependent string, from the result, I find that crossover and mutation operation control exploration while the selection and fitness function control exploitation. Mutation increases the ability to explore new areas of the search space but it also disrupts the exploitation of the previous generation by changing them.

Keywords: Genetic Algorithms, Mutation, Crossover, String, Fitness Function, Coding Scheme.

1. Introduction

The basic principles of GAs were first laid down rigorously by Holland , GAs simulate processes in natural populations which are essential to evolution(Holland,1975). Exactly which biological processes are essential for evolution(davis,1987), and which processes have little or no role to play is still a matter for research; but the foundations are clear.

In natural, individuals in a population compete with each other for resources such as food, water and shelter. Also, members of the same species often compete to attract a mate. Those

individuals which are most successful in surviving and attracting mates will have relatively larger numbers of offspring. Poorly performing individuals will produce few of even no offspring at all. This means that the genes from the highly adapted, or "fit" individuals will spread to an increasing number of individuals in each successive generation. The combination of good characteristics from different ancestors can sometimes produce "superfit" offspring, whose fitness is greater than that of either parent. In this way, species evolve to become more and more well suited to their environment.

GAs is not the only algorithms based on an analogy with nature. *Neural networks* are based on the behavior of neurons in the brain. They can be used for a variety of classification tasks, such as pattern recognition, machine learning, image processing and expert system. Their area of application partly overlaps that of Gas(Hollands,1975).

In section 2 outline the Genetic Algorithm definition, section 3 describe the elements of GA, section 4 outline the three types of GAs operations, section 5 compare GAs with other search techniques, section 6 outline Algorithm Explored, section 7 describe the Knapsack Problem, section 8 developing a coding scheme, section 9 creating fitness functions, section 10 creates an initial population of strings, section 11, 12, and 13 describe parent selection, procreation, and mutation.

2. Genetic Algorithms Definition

It turns out that there is no rigorous definition of GAs accepted by all in the evolutionary-computation community that differentiates GAs from other evolutionary computation methods(Melanie,1990).

A GA is a search technique that uses the principles of evolution and natural selection theory in order to optimize business processes and best apportion limited resources (Goldberg,1989).

A GA is one of a relatively new class of *stochastic* search algorithms. Stochastic algorithms are those that use probability to help guide their search. John Holland developed GAs at the University of Michigan in the mid-1970s: As the name implies (Holland,1975), GAs behaves much like biological genetics. GAs encodes information into strings, just as living organisms encode characteristics into stands of DNA. (The choice of the term *string* is unfortunate. In the GA community, a string contains the potential solution and bears no relationship to a string in C or C++).

3. Elements of Genetic algorithm

The most methods called "GAs" have at least the following elements in common: population of chromosome, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring (Holland,1975).

The chromosomes in a GA population typically take the form of bit string. Each locus in the chromosome has two possible alleles: 0 and 1. Each chromosome can be thought of as a point in the search space of candidate solutions. The GA processes populations of chromosomes, successively replacing one such population with other. The GA most often requires a fitness function that assigns a score (fitness) to each chromosome in the current population. The fitness of chromosome depends on how well that chromosome solves the problem at hand.

3.1 Example of fitness function

One common application of GAs is function optimization, where the goal is to find a set of parameter values that maximize, say, a complex multi-parameter function. As a simple example, one might want to maximize the real-valued one dimensional function (Riolo,1992).

$$F(x) = y + \sin(32y) \qquad 0 \leq y \leq \pi \qquad (1)$$

Here the candidate solutions are values of *y*, which can be encoded as bit string representing real numbers. The fitness calculation translates a given bit string *x* into a real number *y* and then evaluates the function at that value. The fitness of a string is the function value at that point.

As a non-numerical example, consider the problem of finding a sequence of 50 amino acids that will fold to desired three-dimensional protein structure. A GA could be applied to this problem by searching a population of candidate solutions, each encoded as a 50-letter string such as:

IHCCVASASDMIKPVFTVASYLKNWTKAKGPNFEICISGRTPYWDNFPGI,

where each letter represents one of 20 possible amino acids. One way to define the fitness of a candidate sequence is as the negative of the potential energy of the sequence with respect to the desired structure. The potential energy is a measure of how much physical resistance the sequence would put up if forced to be folded into the desired structure- the lower the potential energy, the higher the fitness. Of course one would not want to physically force every sequence in the population into the desired structure and measure its resistance- this would be very difficult, if not impossible. Instead, given a sequence and a desired structure (and knowing some of the relevant biophysics), one can estimate the potential energy by calculating some of the forces acting on each amino acid, so the whole fitness calculation can be done computationally.

These example show two different contexts in which candidate solutions to a problem are encoded as abstract chromosome encoded as strings of symbols, with fitness functions defined on the resulting space of strings. A genetic algorithm is a method for searching such fitness landscapes for highly fit string.

4 Genetic Algorithm operations

The simplest form of genetic algorithm involves three type of operation: selection, crossover, and mutation. These operations are described below.

4.1 Selection

This operator selects chromosomes in the population for reproduction. The fitter the chromosome, the more times it is likely to be selected to reproduce.

4.2 Crossover

The operator randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two offspring. For example, the string 10000100 and 11111111 could be crossed over after the third locus in each to produce the two offspring 10011111 and 11100100. The crossover operator roughly mimics biological recombination between two single-chromosome organisms.

4.3 Mutation

This operation randomly flips some of the bits in a chromosome. for example, the string 00000100 might be mutated in its second position to yield 01000100. Mutation can occur at each bit position in a string with some probability, usually very small (e.g. 0.001).

5- Genetic Algorithms and Traditional Search Methods

There are three meanings of "search". These meaning are described below

5.1 Search for stored data

Here the problem is to efficiency retrieve information stored in computer memory. Suppose you have a large database of names and addresses stored in some ordered way. What is the best way to search for the record corresponding to a given last name? "Binary search" is one method for efficiently finding the desired record describes and analyzes many such search methods(Knuth,1973).

5.2 Search for path to goals

Here the problem is to efficiently find a set of actions that will move from a given initial state to a given goal. This form of search is central to many approaches in Artificial Intelligence (AI).

5.3 Search for solution

This is a more general class of search than "search for paths to goals". The idea is to efficiently find a solution to a problem in a large space of candidate solutions. These are the kinds of search problems for which genetic algorithms are used.

There is clearly a big difference between the first kind of search and second two. The first concerns problems in which one needs to find a piece of information (e.g., a telephone number) in a collection of explicitly stored information. In the second two, the information to be searched is not explicitly stored; rather, candidate solutions are created as the search process proceeds. The AI search methods for solving the problem do not begin with a complete search tree in which all the nodes are already stored in memory; for most problems of interest there are too many possible nodes in the tree to store them all. Rather, the search tree is elaborated step by step in away that depend on the particular algorithm, and the goal is to find an optimal or high- quality solution by examining only a small portion of the tree. Likewise, when searching a pace of candidate solutions with a Gas, not all possible candidate solutions are created first and then evaluated; rather, the GA is a method for finding optimal or good solution by examining only a small fraction of the possible candidates.

"Search for solutions" subsumes "search for paths to goals" since path through a search tree can be encoded as a candidate solution. The candidate solutions could be lists of moves from the initial state to some other state (correct only if the final state is the goal state). However, many "search for paths to goals" problems are better solved by the AI-tree-search techniques (in which partial solutions can be evaluated) than by GA (in which full candidate solutions must typically be generated before they can be evaluated).

However, the standard AI-tree-search (or, more generally, graph-search) methods do not always apply. Not all problems require finding a path from an initial state to a goal (Melanie,1990).

The GA is a general method for solving "search for solutions" problem. Hill climbing, simulated annealing, and tabu search are example of other methods. Some of these are similar to "search for paths to goals" methods such as branch-and-bound. For descriptions of these and other search methods (Davis,1987).

All the "search for solution " method (1) initially generate a set of candidate solutions (in the GA this is the initial population; (2) evaluate the candidate solutions according to some fitness criteria; (3) decide on the basis of this evaluation which candidates will be kept and which will be discarded; add (4) produce further variants by using some kind of operators on the surviving candidates.

The particular combination of elements in GAs parallel population-based search with stochastic selection of many individuals, stochastic crossover and mutation distinguishes them from other search methods.

GAs also has the quality of *robustness.* That is, while special-case algorithms may find more optimal solutions to specific problems. GAs performs very well over a large number of problem categories. This robustness results in part because genetic algorithms usually apply their search against a large set of points, rather than just a single point, as do calculus-based algorithms. Because of this, GAs are not caught by local minima or maxima. Another contribution to their robustness is that GAs use the strings fitness to direct the search; therefore they do not require any problem-specific knowledge of the search space, and they can operate well on search spaces that have gaps, jumps, or noise.

GAs also performs well on problems whose complexity increase exponentially with the number of input parameters. Such problems, called NP-complete, would take years to solve using traditional approaches. Furthermore, genetic algorithms can produce intermediate solutions; the program can stop at any time if a suboptimal solution is acceptable. Finally, GAs easily lend themselves to parallel processing; the can be implemented on any multiprocessor architecture.

6. The Algorithm Explored

Given a clearly defined problem to be solved and symbol string representation for candidate solutions, GA works as follows:

Step 1 Start with a randomly generated population of n *1-bit* chromosomes (candidate solutions to a problem).

Step 2 Calculate the fitness *f(x)* of each chromosome *x* in the population.

Step 3 Repeat the following *steps* until *n* offspring have been created:

Step 3.1 Select a pair of parent chromosomes from the current population, the probability of selection being an increasing function of fitness. Selection is done "with replacement", meaning that the same chromosome can be selected more than once to become a parent.

Step 3.2 With probability p_c (the "crossover probability" or "crossover rate"), cross over the pair at randomly chosen point (chosen with uniform probability) to form two offspring. In no crossover takes place, form two offspring that are exact copies of their respective parents.

Step 3.3 Mutate the two offspring at each locus with probability p_m (the mutation probability or mutation rate), and place the resulting chromosomes in the new population, if n is odd; one new population member can be discarded at random.

Step 4 Replace the current population with new population.

Step 5 Go to step 2.

Each iteration of this process is called *generation.* The entire set of generation is called a *run*. The basic GA is quite simple. While extensive research on GAs has produced on optimal implementation (many aspects of GAs are still debated), there are several algorithms that work quite well in most situations. This algorithm is one of this implementation, and it is simple and reliable. Even with an off-the-shelf GA, the programmer still faces two significant tasks: designing the coding scheme and creating the fitness function. The coding scheme defines how a string will represent a potential solution of the problem at hand. The fitness function uses the coding scheme to evaluate each string's fitness or worth. By combining these two parts the genetic algorithm can calculate how well any string solves the problem.

7. The Knapsack Problem

To understand how GAs work, consider a concrete example. The knapsack problem is a classic NP-complete problem(Sedgewik,1988) .Simply stated, given a pile of items that vary in weight and value, find that combination of items having the greatest total value but which does not exceed a maximum weight. In other words, the goal is to fill up hypothetical knapsack with the most expensive loot it can carry. While it's easy to describe, this goal can be difficult to accomplish. For example, a pile of just 50 items present 2^{50} different possible selections. Assuming a computer could test a million different combinations each second, it would still take 35 years to try them all. In this paper, I show how a GA solves such a problem, but for the sake of illustration I use a smaller number of items. The pile contains fourteen items, so it provides a little more than 16,000 possible combinations to try in the knapsack. There are five different kinds of items, ranging from 3 to 9 in weight and from 4 to 13 in value. The Knapsack can hold a maximum weight of 17, so it can carry one **A**, or two **B**s, two **Cs**, four **Ds** and five **Gs**. Table 1 list all the different items, their weight and values, and maximum number of each type that can fit into the knapsack

Table 1 *Description of the knapsack items*

Label	**A**	**B**	**C**	**D**	**E**
Weight	**9**	**8**	**7**	**4**	**3**
Value	**13**	**11**	**10**	**5**	**4**
Quantity	**1**	**2**	**2**	**4**	**5**

The program in listing 1 illustrates the use of a GA to solve the knapsack problem. Supporting functions and class definitions appear in listings 2 through 7.

8. Developing a Coding Scheme

The first step in writing a GA is to create a coding scheme. A coding scheme is a method for expressing a solution in a string. Many successful types have been discovered, but there is no mechanical technique for creating one. Like programming, creating a coding scheme is part science and part art, but also like programming, it gets easier with practice. Early

researchers used binary encoded starting exclusively, but higher order alphabets work without loss of efficiency and power. The type of coding scheme to used depends on the problem.

Listing 1 Demo program and global functions to solve knapsack problem

```
#include <stdlib.h>
#include <stdio.h>
#include "pop.h"
Static enum Bool {FALSE, TRUE};
// Maxweight defines the constraint of the knapsack example and ItemDes is simply
the //coding scheme.
static const int MAXWEIGHT = 17;
static cons struct
{
   int   Weight;
   int    Value;
} ItemDesc[] = {{3, 4}, {3, 4}, {3, 4}, {3, 4} {3, 4},{4, 5}, {4, 5}, {4, 5}, {4, 5},   {7, 10},
                  {7, 10}, {8, 11}, {8, 11}, {9, 13}};
void CalcWeightAndValu( CGAChromosome *Chromosome, int& Weight, int& Value);
Bool FoundSolution(CGAPopulation& Pop);
Void PrintPop(CGAPopulation& Pop);
//Main creates the population of chromosome and continues creating new generation
until
//a solution is found
int main(void)
{
   const float MaxMutationRate = 0.2;
   const int     PopulationSiz       = 30;
```

Listing 1 *continued*

```
.
  CGAPopulation Pop(PopulationSize, sizeof(ItemDesc) / sizeof(ItemDesc[0]),
                           MaxMutationRate);
While (!FoundSolution(Pop)) {
     Pop.CreateNextGeneration();
     PrintPop(Pop);
  }
    return EXIT_SUCCESS;
}
//print information and statistics about population
void PrintPop(CGAPopulatio& Pop)
{
   float TotalFitness = 0;
   printf("Idx  Fit  Val  Wt   Chromosome\n);
   for (size_t ChromIdx = 0; ChromIdx < Pop.GetPopulationSize(); ChromIdx++){
```

```
      int    Weight;
      int     Value;
      CGAChromosome *Chromosome = Pop.GetChromosome (ChromIdx);
      TotalFitness += Chromosome->GetFitness();
      CalcWeightAndAlue(Chromosome, Weight, Value);
      Printf("%3u  %4.0f  %3d  %3d  ", ChromIdx, Chromosome->GetFitness(),
              Value, Weight);
      for (size_t BitIdx= 0; BitIdx < Chromosome->GetLength(); BitIidx++){
      printf ("%1u", (*Chromosome)[BitIdx]);
   }
   printf("\n");
 }
 printf("Gen, Best, Avg, Worst: %4d, %6.2f, %6.2f, %6.2f\n", Pop.GetGeneration(),
           Pop.GetBestFitness(), TotalFitness / ChromIdx,
         Pop.GetChromosome(ChromIdx – 1)->GetFitness());
      getchar();
 }
//Check if a solution has been found. By definition it must have a value of ANSWER (24)
and //not exceed MAXWEIGHT (17). Since the fittest chromosome could violate the
weight //constraint FoundSolution must search  through the population of chromosome
Bool FoundSolution( CGAPopulation& Pop)
 {
   const int ANSWER = 24;
    int         Weight;
    int          Value;
    for (size_t ChromIdx = 0; ChromIdx < Pop.GetPopulationSize(); ChromIdx++){
        CalcWeightAndValue(Pop.GetChromosome(ChromIdx), Weight, Value);
        If (Weight <= MAXWEIGHT && value = = ANSWER){
            Return TRUE;
        }
     }
       return FALSE; }
//Calculate the fitness of each chromosome by adding its weight to its value then
subtracting
// A PENALITY for the excess weight.
 float CalcFitness(CGAChromosome *Chromosome)
{
```

Listing 1 *continued*

```
   const float   PENALITY  = 3.0;
    int            Weight;
    int             Value;
    CalcWeightAndValue(Chromosome, Weight, Value);
     if (Weight > MAXWEIGHT) {
        return Value – PENALITY * (weight – MAXWEIGHT);
     } else {
```

```
        return Value;
    }
}
//Calculate the weight and value of the chromosome by accumulating the weight and
value of //each item whose bit in the chromosome is set true
void CalcWeightAndValue(CGAChromosome *Chromosome, int& Weight, int& Value)
 {
    Weight  = 0;
    Value    = 0;
    for (size_t Idx = 0; Idx < Chromosome->GetLength(); Idx++ {
    If ((*Chromosome)[Idx]) {
      Weight += ItemDesc[Idx].Weight;
       Value +=  ItemDesc[Idx].Value;
    }
  }
 }
/* End of File */
```

Order defines the number of different characters in the alphabet. Do not confuse the GA term *character* with ASCII characters. A GA character is analogous to a gene in that it has a position and a value. A binary alphabet has an order of two, meaning that the characters can only have two values, 0 or 1. The coding scheme I've chosen for the knapsack uses a fixed-length, binary, position-dependent string. The pile in the example contains fourteen items so each string must have fourteen binary characters, one character for each item. The location of each character in the string represents a specific item and the value of the character indicates whether that item is in the knapsack or left in the pile. Fig 1 illustrates the coding of fourteen items into a GA string. The coding scheme's equivalent in C++ is the array of *struct*, *ItemDesc*, show in Listing 1. Each column of the table represents a character position in the string. The top three lines give the label, weight, and value of each character position. The bottom three lines show strings that define potential solutions to the knapsack problem. In this case a 1 means the item is in the knapsack and a 0 means the item is in the pile. The first string places six items into the knapsack; one A, B, C, and D, and two Es. For a total weight of 34 and total value of 47. The second string places five items in the Knapsack: two Ds, and three Es, for a weight of 17 and value of 22. The third string uses just two items: one A and one E for a weight of 12 and a value of 17.

9. Creating a Fitness Function

The next step is to create a function that will evaluate how well each string solves the problem- that is, calculate the string's fitness. The knapsack problem requires maximization of the loot's value in the knapsack. If this were the only requirement, a fitness function could simply rank a string by adding up the values of all the items put into the knapsack. The GA would then tell us that the best solution was to put all 14 items in the knapsack. However, a second requirement states that the weight of the
item cannot exceed a maximum (17, in this example).

Fig 1 the coding scheme for the knapsack example
A B B C C D D D D E E E E E Label

```
9  8  8  7  7 4  4 4  4  3  3 3  3 3   Weight
13 11 11 10 10 5  5 5  5  4  4  4 4 4   Value

Strings

1  0  1 1  0  0  0  0  1 1  1 0  0 0   34  47
0  0  0 0  0  1  1  0  0 1  1 0  1 0   17  22
1  0  0 0  0  0  0  0  0 0  0 0  1 0   12  17
```

So this fitness function fails miserably. In GA terminology, it results in a *constraint violation.* GA researchers have explored many approaches to constraint violation but none are perfect. Here are three possibilities:

Elimination

Elimination attempts to determine if a string violates the constraint before it is ever created. This approach has several problems. For starters, it may be too expensive to perform, or simply impossible. Second, preventing the creation of violators may cause GA to overlook perfectly valid solutions. That's because violators could produce legal (non-violating) offspring that would lead to a satisfactory solution more quickly.

High Penalty

This approach imposes a high penalty on violators. It reduces violators worth while allowing them to occasionally propagate offspring. A weakness of this approach becomes apparent when a population contains a large percentage of violators. In this case, legal strings will dominate the following generation and the violators will be left unexploited. This effect could lead to population stagnation.

Moderate Penalty

This approach imposes a moderate penalty on violators. It increases the probability that violators will procreate, thus reducing the chance of population stagnation. This approach exhibits its own problems, especially when violators rate higher than legal strings. In this case, if the violators do not create legal strings then violators will dominate the following generations. Furthermore, if violators rate-higher than legal strings then the criteria for ending the search must incorporate a mechanism for detecting violators. The knapsack example employs the third technique. Its fitness function (***CalcFitness*** in Listing 1) adds up the value of each item and subtracts a moderate penalty for violators. The penalty is three times the amount of excess weight. Table 2 shows the resulting fitness of the three example strings previously defined.

Listing 2 *Definition of class CGAChromosome and short member function*

```
 #include "random.h"
 #ifndef_INC_CHROM_
 #define_INC_CHROM_
 class CGAChromosome
{
  public:
    static void   InitializChromosomeClass(size_t Length);
    ~CGAChromosome();
    CGAChromosome(float Prob = 0.0);
    CGAChromosome* Compliment() const;
    void                    Mutate(float Prob);
    intline_size_t          GetLength() const;
    inline float            GetFitness() const;
   friend void  Crossover(CGAChromosome* Parent1, CGAChromosome* Parent2,
           CGAChromosome*& Child1,  CGAChromosome*& Child2, float Prob);
   friend float  CalcSimilarityRatio(CGAChromosome* Chrom1,
                                     CGAChromosome* Chrom2);
    inline GABool& operator[](size_t Idx);
   private:
     static void InitializeCrossoverMask(float Prob);
     static GABool*  m_CrossoverMask;
     static size_t       m_Count;
     static size_t       m_Length;
     GABool*         m_Data;
     float               m_fitness;
};
size_t CGAChromosome::GetLength() const
{
    return m_Length;
 }
float CGAChromosome::GetFitness() const
{
    return m_Fitness;
 }
GABool& CGAChromosome::operator[](size_t Idx)
{
    return m_Date[Idx];
 }
float CalcFitness(CGAChronosome*  Chromosome);
#endif
/* End of file */
```

Table 2 The results of the fitness function using a moderate penalty

Weight	34	17	12
Value	47	22	17
Fitness	- 4	22	17

Listing 3 More member functions of class CGAChromosome

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "chrom.h"
```

Listing 3 *continue*

```
size_t    CGAChromosome::m_Count   = 0;
size_t     CGAChromosome::m_Length = 0;
GABool* CGAChromosome::m_CrossoverMask   = 0;
//Default constructor and destructor. Create the chromosome with bits turned on at
the given //probability
 CGAChromosome:: CGAChromosome(float Prob)
{
  assert(m_Length);
   m_Fitness   = 0;
   m_Date      = new GABool[m_Length];
   assert(m_Date);
   m_Count++;
for (size_t Idx = 0; Idx < m_Length; Idx++)
   {
       m_Date[Idx] = Flip(Prob);
   }
   m_Fitness = CalcFitness(this);
}
CGAChromosome::~CGAChromosome(void)
}
 delete[] m_Data;
  if (--m_Count = = 0) {
      delete[ ] m_CrossoverMask;
       m_Length            = 0;
       m_CrossoverMask = 0;
    }
 }
//Mutate a single chromosome gene selected at random for a given probability.
void CGAChromosome::Mutate(float Prob)
 {
    for (size_t Idx = 0; Idx < m_Length; Idx++) {
            if (Flip(Prob)) {
            m_Data[Idx] = !m_Data[Idx];
            }
       }
}
//Create two new offspring using a uniform crossover operator created with the given
//probability
 void Crossover(CGAChromosome*    Parent1, CGAChromosome*    Parent2,
        CGAChromosome*& Child1, CGAChromosome*& Child2, float  Prob)
```

```
{
GAChromosome::InitializeCrossoverMask(0.5);
 Child1 = new CGAChromosome();
 Child2 = new CGAChromosome();
 assert(Child1 && Child2);
 for (size_t Idx = 0; Idx < CGAChromosome::m_Length; Idx++) {
  if (CGAChromosome::m_CrossoverMask[Idx]) {
    Child1->m_Date[Idx] = Parent1->m_Date[Idx];
    Child2->m_Date[Idx] = Parent2->m_Date[Idx];
    } else {
       Child1->m_Date[Idx] = Parent2->m_Date[Idx];
       Child2->m_Date[Idx] =Parent1->m_Date[Idx]; }
```

Listing 3 *continue*

```
  }
 Child1->Mutate(Prob);
 Child2->Mutate(Prob);
 Child1->m_Fitness = CalcFitness(Child1);
 Child2->m_Fitness = CalcFitness(Child2);
}
//Calculate the difference between two chromosome
 float CalcSimilarityRatio(CGAChromosome* Chrom1, CGAChromosome* Chrom2)
{
for (size_t Idx = 0, Idx = 0, MatchCount = 0; Idx < CGAChromosome::m_Length;
                Idx++) {
    If  (Chrom1->m_Data[Idx] = = Chrom2->M_Data[Idx]) {
           MatchCount++; }
   } return (float)MatchCount / (float)CGAChromosome::m_Length;
}
// Set the new chromosome to the opposite encoding of this chromosome.
CGAChromosome* CGAChromosome::complement(void) const
{
  CGAChromosome* Chromosome = new CGAChromosome;
  for (size_t Idx = 0; Idx < m_Length; Idx++) {
       Chromosome->m_Data[Idx] = !m_Data[Idx];
  }
  Chromosome->m_Fitness = CalcFitness(Chromosome);
   return Chromosome;
  }
//Setup the crossover mask for creating the next offspring
 void CGAChromosome::InitializeCrossoverMask(float Prob)
 {
  assert(m_Length && m_CrossoverMask);
  for (size_t Idx = 0; Idx < m_Length; Idx++)
  {
     m_CrossoverMask[Idx] = Flip(Prob);
   }
```

```
  }
// Setup the crossover mask for creating the next offspring.
void CGAChromosome::InitializeCrossoverMask(float Prob)
{
   assert(m_Length &&  m_CrossoverMask);
   for (size_t Idx = 0; Idx < m_Length; Idx++) {
        m_CrossoverMask[Idx] = Flip(Prob);
    }
}
//Setup the chromosome length and allocation memory for the crossover mask.
 void CGAChromosome::InitializeChromosomeClass(Size_t Length)
 {
    assert(Length);
    m_Length = Length;
    if (m_Count ! = 0) {
       delete [ ] m_CrossoverMask;
    }
    m_CrossoverMask  =  new  GABool[m_Length];
    assert(m_CrossoverMask);  }
/* End of File */
```

10. Initialization

After the coding scheme and fitness function are integrated into the GA it is ready to run. The GA's first task is to create an initial population of strings. The demo program stores this population in an object (*Pop*) of class *CGAPopulation*, (defined in listing 4). Each string in the population is an object of class *CGAChromosome* (Listing 2).

There are many ways to select an initial population; approaches range from randomly setting each character of every string to modifying the result of a search made previously by a human. The knapsack example uses a *modified weighted random* design. The initialization function (class ***CGAPopulation's*** constructor, Listing 5) creates strings with an increasing likelihood of setting each bit to 1.

Listing 4 Definition of class CGAPopulation and short member function

```
 #include "random.h"
 #include "chrom.h"
 #ifndef  _INC_POP_
 #define  _INC_POP_
 class CGAPopulation
  {
   public:
        ~ CGAPopulation();
        CGAPopulation(size_t Size, size_t Length, float MaxMutationRate);
        void          CreateNexGeneration(void);
        inline float  GetBestFitness(void);
        inline size_t Get Generation(void);
        inline size_t GetPopulationSize(void);
        inline         CGAChromosome*  GetChromosome(size_t Idx);
```

```
    private:
      CGAChromosome* GetParent();
      void ReplaceChromosome(CGAChromosome*  Chromosome);
  void Merge(CGAChromosome* NewChromosome);
  size_t                m_MaxSize;
  size_t                m_CurrentSize;
  CGAChromosome**  m_Date;
  size_t                  m_Generation;
  float                   m_MaxMutationRate;
};
  float CGAPopulation::GetBestFitness(void){
  return m_Date[0]->GetFitness();
}
 size_t  GAPopulation::GetGeneration(void){
     return m_Generation; }
 size_t  GAPopulation::GetPopulationSize(void) {
    return  m_CurrentSize;}
 CGAChromosome* GAPopulation::GetChromosome(size_t Idx) {
    return m_Date[Idx]; }
#endif
/* End of File*/
```

Fig 2 shows the result of creating ten strings. The probability of setting any bit to 1 for the first string, labeled U, is 10%. The probability increases incrementally for each new string created until all the strings are created and the probability reaches about 50%. After creating and initializing each string, the constructor creates a complement of that string, by calling member function *Complement* (Listing 3). The complement string has the opposite bit pattern of the original. Note that in the top half of the table the U string contains only one one-bit, whereas each successive string has an increasing number of one-bits, until the fifth string has about half ones and zeros. The bottom half of the figure shows the complement strings to the original five. The composition of the initial population can dramatically affect the performance of the genetic algorithm.

The more diverse the initial population the more opportunities the GA will have to exploit the search space. The above initialization scheme has the advantage of simplicity and diversity. It is simple in that it does not require any information about the problem. The scheme is diverse because the function creates strings ranging from mostly zeros to mostly ones and everything in-between. How large should the initial population be? The population should be large enough to create a diverse set of individuals for the GA to exploit but not so large that creating the initial population dominates computer time. The knapsack example sets the initial population to 30. I picked this rather small population to better illustrate how GAs work.

.

Listing 5 More member functions of class CGAPopulation

```
#include <stdlib.h>
#include <stdio.h>
#include <float.h>
#include <assert.h>
#include <string.h>
#include "Pop.h"
#include "random.h
//constructor creates a population of chromosomes where the probability that each bit is true //increase with each creation, then it creates a complete of each of the created chromosomes
   CGAPopulation::CGAPopulation(unsigned int Size, unsigned int Length,
                                    float MaxMutationRate)
{
    assert(Size && Length);
     m_MaxSize              = Size;
     m_CurrentSize          = 0;
     m_Generation           = 0;
     m_MaxMutationRate  = MaxMutationRate;
    m_Date                  = new CGAChromosome*[m_MaxSize];
    CGAChromosome::InitializeChromosomeClass(Length);
    for (unsigned int idx = 1; idx <= m_MaxSize / 2 ; idx++){
       CGAChromosome* Temp = new CGAChromosome(idx / (float) m_MaxSize);
       CGAChromosome* CompTemp = Temp->complement();
       Merge(Temp);
//Default destructor
CGAPopulation::~CGAPopulation()
 {
  for (unsigned int idx = 0; idx < m_MaxSize; idx++) {
        delete m_Data[idx]; }
```

Listing 5 *continued*

```
  delete m_Data;
}
//Merge sort the new chromosome, assume that there no holes in the array.
 void CGAPopulation::Merge(CGAChromosome* NewChromosome)
```

```
 {
   assert(m_CurrentSize < m_MaxSize);
   for (unsigned int idx = 0; idx <m_CurrentSize;  idx++) {
     if (NewChromosome->GetFitness() > m_Data[idx]->GetFitness()) {
         break;
    }
  }
  memmove(&m_Data[idx+1], &m_Data[idx], sizeof(m_Data) *
                                    (m_CurrentSize - idx));
  m_Data[idx] = NewChromosome;
  m_CurrentSize++;
}
// these function randomly select a chromosome from the rank array, where the
probability of //selection is related to the individual position in the probability for
replacement by Rank(x) / //Total. GetParent calculate an individual probability for
parenthood by (Total - Rank(x) /  //Total).
CGAChromosome* CGAPopulation::GetParent()
 {
    float Selection;
     do {
         Selection = Rand0UpTo1();
     } while (Flip(Selection));
     return m_Date[(int)(Selection * m_MaxSize)];
    }
//Replace a poor chromosome with the new one.
void CGAPopulation::ReplaceChromosome(CGAChromosome* NewChromosome)
{
  float  Selection;
do {
        Selection = Rand0UpTo1();
   } while (Flip(Selection));
     unsigned int idx = m_MaxSize –  (int)(Selection * m_MaxSize) - 1;
     delete m_Data[idx];
     memmove(&m_Data[idx], &m_Data[idx + 1], sizeof(m_Data) *
                                              (m_CurrentSize - idx - 1));
     m_CurrentSize--;
     Merge(NewChromosome);
}
 // create two offspring and replace two member of the chromosome array.
void CGAPopulation::CreateNexGeneration(void)
{
  CGAChromosome *Parent1, *Parent2, *Child1, *Child2;
  Parent1  = GetParent();
  Parent2  = GetParent();
  Crossover(Parent1, Parent2, Child1, Child2, CalcSimilarityRatio(Parent1, Parent2) *
                          m_MaxMutationRate) ;
```

Listing 5 *continued*

```
  ReplaceChromosome(Child1);
  ReplaceChromosome(Child2);
  m_Generation++;
}
/* End of File
```

Fig 2 Initializing a population of ten strings

Strings	1s Count	Prob
U 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1	10%
W 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0	2	20%
X 0 1 1 1 0 0 0 0 0 1 1 0 0 1	6	30%
Y 0 0 1 1 0 1 1 0 0 1 1 0 0 0	6	40%
X 1 1 0 1 0 1 0 1 1 0 1 1 0 0	8	50%

String's complement

```
~U  0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
~W 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1
~X  1 0 0 0 1 1 1 1 1 0 0 1 1 0
~Y  1 1 0 0 1 0 0 1 1 0 0 1 1 1
~X  0 0 1 0 1 0 1 0 0 1 0 0 1 1
```

11. Parent Selection and Procreation

After creating an initial population, the GA selects two parents for the purpose of procreation. Parent selection is based on string fitness. While creating the initial population, the fitness function calculates the worth of each string. This calculation occurs within each string's constructor. *CGAChromosome::CGAChromosome* (Listing 3).

The population constructor *CGAPopulation::CGAPopulation* then ranks the strings according to their fitness, by calling member function *Merge* (Listing 5). After the population constructor returns, the main program enters a *while* loop, and says there until a solution is found. It's unlikely that any strings in the initial generation contain a solution; if the *while* condition is satisfied (no solution found), the program calls *CGAPopulation::CreateNextGeneration* (Listing 5) to create a new generation of strings. The first step in creating a new generation is selection of two parents. The GA does not select strings directly by their rank in the population, so the best string is not guaranteed to be a parent. Instead, the string's worth, based on its rank in the population as a whole, biases the *probability* of that string being selected to parent the next generation. If a string ranks as the 25th best out of 100 strings then it has a 75% chance of becoming a parent.

Listing 6 random.h – header file for probability functions

```
#ifndef  _INC_GLOBAL_
#define  _INC_GLOBAL_
typedef  unsigned char GABool;
```

```
 float       Rand0UpTo1(void);
 float       Rand0To1(void);
 GABool  Flip(float Prob);
 #endif
/ * End of File */
```

Listing 7 Probability functions

```
 #include <stdlib.h>
 #include "random.h"
 // Return a pseudo-random number from 0.0 upto 1 but not
including 1.0.
 float Rand0UpTo1(void) {
    return rand() / (float)(RAND_MAX + 1.0)  }
// Return a pseudo-random number from 0.0 and 1.0 inclusive.
 float Rand0To1(void) {
    return rand() / (float)RAND_MAX;  }
// Return TRUE at the specified probability.
 GABool Flip(float Prob)
 {
   return Rand0To1()<= Prob;
 }
 /* End of File */
```

GA's method of selecting parent is very important: it can be penalty impact the efficiency of the search. Among the many types of selection functions, the two most widely used techniques are *proportional* selection and *linear rank* selection. The knapsack example uses linear rank selection, first because it is easy to implement (see *CGAPopulation::GetParent*, Listing 5). More important, I suspect that linear rank selection is inherently better behaved than proportional selection, because proportional selection has required many fixes to its original design over the years. Linear rank selection simply calculates the fitness of a string and then ranks it in entire population. This process involves two random operations. First, *GetParent* randomly selects a candidate string from the population:

```
.. .. .. .. .. .. .. ..
Selection =
        Rand0UpTo1();
… … .. .. .. .. .. ..
```

Next *GetParent* determines if the candidate string will parent an offspring, by performing a weighted probability (via function *Flip*, Listing 7) based on the string's rank. If the string does not quality as a parent, then *GetParent* repeats the cycle and randomly selects another string from the population.

12. Procreation

Once two parents have been selected, the GA combines them to create the next generation of strings. The GA creates two offspring by combining fragments from each of the two parents. The knapsack example uses *uniform crossover* to cut up fragments from the parents (see function *Crossover* , Listing 3). The positions where strings are cut into fragments are called the crossover points. *Crossover* chooses these points at random: uniform crossover means that every point has an equal chance of being a crossover point Crossover selection occurs via a crossover mask. Fig 3 illustrates the use of a crossover mask. The first child will receive a first parent's character if the bit is 1 and the second parent's character if the bit is 0. The second child works in reverse. Uniform crossover implies many crossover points with an even probability distribution across the entire length of each parent string. The fragments from the first parent combined with their complementary members from the second parent create two new strings.

13. Mutation

Sometimes the children undergo mutation. The knapsack example uses an interesting operator (see *CGAChromosome::Mutate,* Listing 3). Rather than a fixed mutation probability, *Mutate* uses a probability that changes based on the makeup of the population. *Mutate* compares the two parents of the child; greater similarity between parents increase the probability that a mutation will occur in the child. The reason for using a variable mutation probability is to reduce the chance of premature convergence. This condition occurs when the population rushes to a mediocre solution and then simply runs out of steam. There is little diversity left and the search becomes a random walk among the average. This is similar to a biological species becoming so inbred that it is no longer viable. To reduce premature convergence the mutation operate kicks in when the population shows Fitness diversity and adds new variety by introducing random mutation.

Fig 3 The uniform crossover operator

Parent1	1	1	0	0	0	1	0	0	0	1	1	1	0
Parent2	1	0	1	1	1	1	1	1	0	0	1	1	0
Mask		1	0	0	1	1	0	1	0	0	0	0	1	1
Child1	1	0	1	0	0	1	0	1	0	0	1	1	0
Child2	1	1	0	1	1	1	1	0	0	1	1	1	0

14. Conclusion

The two children now can replace two older strings from the population. This occurs in function *CGAPopulation::ReplaceChromosome* (Listing 5). As it does with parent selection, the GA chooses these older strings with a random bias. In this case, however, the worst string will have the greatest chance of being removed. After the insertion of new strings, the population is then ranked again. The program stops when any string solves the problem. This raises a question: if the best solution is unknown, how can the program determine if a it has found the best answer, or at least, one of the better answers? One approach would be to solve for a fixed solution that meets some predetermined minimally acceptable value. A second approach would be to run the program until its rate of finding better answers drop off or the rate of improvement of that answer flattens out.

The knapsack example ran until it found the known answer, which is 24. it took on average, about 160 generation to find the solution – about 350 out of 16,000 possibilities, or 2% of the search space. Indeed, this problem is small enough to solve with traditional methods, but by watching how the code operate in detail you can get a good idea of how GAs work. The example is quite small and expandable. You can try it on different problems simply by creating anew *ItemDesc* structure and the related *CalcFitness* function. All the I/O is confined to the *PrintPop* function (Listing 1), so you could easily drop the example into a large program.

Genetic algorithms balance exploitation with exploration. The crossover and mutation operators control exploration while the selection and fitness function control exploitation. Increasing exploitation decreases exploration. Mutation increases the ability to explore new areas of the search spaces but it also disrupts the exploitation of the previous generation by changing them. Genetic algorithms represent a new, innovative approach to search algorithms. Unlike most traditional algorithms, GAs are not deterministic rather they exploit the power of probabilistic operations. By definition and design they are adaptive. Survival of the fittest governs the progress of the search, and with the possibility of mutations, GAs may explore completely unexpected avenues

15. References

[1] Davis, L. (1987). Genetic algorithms and simulated annealing. London: Pitman ;.

[2] Davis, L. (1991). Handbook of genetic algorithms. New York: Van Nostrand Reinhold.

[3] Man, K. F., & Tang, K. S. (1999). Genetic algorithms: concepts and designs. London: Springer.

[4] Glover, F. (1989). Tabu Search—Part I.ORSA Journal on Computing , 1(3), 190-206.

[5] Glover, F. (1990). Tabu Search—Part II.ORSA Journal on Computing , 2(1), 4-32.

[6] Goldberg, D. E. (1989). Genetic algorithms in search, optimization, and machine learning. Reading, Mass.: Addison-Wesley Pub. Co..

[7] Holland, J. H. (1975). Adaptation in natural and artificial systems: an introd. analysis with applications to biology, control, and artificial intelligence.. Ann Arbor: Univ. of Michigan Pr..

[8] Knuth, D. E. (1973). The art of computer programming. Reading, Mass.: Addison-Wesley Pub. Co..

[9] Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization By Simulated Annealing. Science, 220(4598), 671-680.

[10] Mitchell, M. (1996). An introduction to genetic algorithms. Cambridge, Mass.: MIT Press.

[11] R.L, R. (1992). Survival of the fittest bit.. Scientific American, 267, 114-116.

[12] Sedgewick, R. (1988). Algorithms (2nd ed.). Reading, Mass.: Addison-Wesley.

[13] Winston, P. H. (1992). Artificial intelligence (2nd ed.). Reading, Mass.: Addison-Wesley.