ABHATH AL-YARMOUK: "*Basic Sci. & Eng.*" Vol. 17, No.1B, 2008, pp. 191-202

An Efficient Genetic Algorithm for Solving Knapsack Problem.

Awni Mansour Hammouri*

Received on Feb. 26, 2007 **Accepted for publication on Oct. 16, 2007**

Abstract

The goal of this study is to provide an enhanced solution to the knapsack problem using Genetic Algorithm. The solution is based on using a new fitness function which reduces the number of iterations needed, as a result reducing the computation time. Genetic Algorithms are relatively easy to use for finding the optimal solution, or the approximate optimal value of NP-Complete problem. For the sake of illustration, fourteen items of five different kinds have been used in this problem. Therefore, it provides 16,384 possible combinations to try in the knapsack. As a result of using the enhanced fitness function, it has been found that finding the optimal answer took, on average, about 29 iterations compared to 126 iterations using traditional fitness function. This gives an improvement of 77% in computation time.

Keywords: Genetic Algorithms, Knapsack problem, NP-Complete problems, Crossover, Mutation.

Introduction:

A number of problems are difficult to solve by traditional algorithms. 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: (traveling salesman problem) finding the shortest path connecting a set of cities, circuit design, job shop scheduling, dividing a set of different tasks among a group of people to meet a deadline, fitting a set of various sized boxes into the fewest trucks, or video & sound quality optimization. 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 algorithms.

Genetic Algorithms (GAs), originally conceived by Holland [7], represent a fairly abstract model of Darwinian evolution and biological genetics. They evolve a population of competing individual using fitness-biased selection, randomly mating, and a gene-level representation of individual together with simple genetic operators (crossover and mutation) for modeling inheritance of traits. These GAs have been successfully applied to a wide variety of problems including multimode function optimization, machine

* Department of Information Technology, Mutah University, Jordan

learning, and the evolution of complex structures such as neural networks and Lisp programs.

Genetic Algorithms were initially developed by Bremermann [1] in 1958 but popularized by Holland who applied GA to formally study adaptation in nature for the purpose of applying the mechanisms into computer science [5]. This work lead to the development of the Schema starting in 1968 [6]. The Schema was explained in detail in his 1975 book Adaptation in Natural and Artificial Systems [7]. The Schema Theorem represented Holland's attempt to place Genetic Algorithms on firm theoretical framework. The first advancement on the Schema theory was by Goldberg who made the popular supposition known as Building Block Hypothesis that crossover is the major source of Genetic Algorithm performance [4]. This is in contrast to the Schema theory, which is focused mainly on the destructive behavior of the crossover and mutation operators.

In the 1990s, criticisms of the Schema theorem have appeared. Grefenstette argued [3] that the Schema theorem formulates that the GA will converge schemas that are winners of actual competition rather than on schemas with the best-observed fitness. Fogel and Ghozeil [2] criticized the Schema theorem for not being able to estimate the proportion of Schema in a population when fitness proportionate selection is used in the presence of noise or other stochastic effects. In Holland's defense, Poli argued [9] that Fogel and Ghozeil's criticisms were not based upon Holland's original theorem and that the original theorem is very good at modeling Schema in the presence of noise. Radcliff also defended [10] the Schema theorem by explaining that many of the criticisms were not with Holland's theorem itself but with it over-interpretation. To augment the Schema theory, more "exact" mathematical models have also been developed to make predictions about the population composition, the speed of population convergence and the distribution of fitness's in the population over time which the Schema theory does not directly address. In 1991, Vose and Liepins [11] solved a simple genetic algorithm with an exact model that provided a geometric picture of the GA's behaviour and since then various other authors have provided "exact' models. Other less exact methods such as applying Markov Chain analysis to modeling GA by Nix Vose [8] have been attempted; however, the formulations were difficult to solve due to high dimensions and nonlinearities.

One difficulty in applying GAs to the knapsack problem is that the bit-string representation of the canonical GA chromosome does not provide a direct mapping of the problem on to the GA chromosome. In [12], Ku and Lee proposed a new chromosome representation of the GA, they call "set-oriented GA". A chromosome in the set-oriented GA is a set, while in the canonical GA it is a bit-string. Crossover and mutation operators are described using the combinations of set operations, such as union, intersection and complement. A performance comparison of the canonical GA and the set-oriented GA on the knapsack problem were presented. The set-oriented GA turns out to be not only effective in representing the problem but also efficient in finding the solution. Li *et. al*. [13] proposed a genetic algorithm solution for the unbounded knapsack problem. The proposed algorithm is based on two techniques. One is a heuristic operator, which utilizes problem-specific knowledge, and the other is a preprocessing technique. Their results showed that the proposed algorithm is capable of obtaining high-quality solutions for problems of standard randomly generated knapsack instances, while requiring only a modest amount of computational effort.

2. Genetic Algorithm Operators

The simplest form of genetic algorithm involves three types of operators: selection, crossover, and mutation. **Selection** operator selects chromosomes in the population for reproduction. The fitter the chromosome, the more times is likely to be selected to reproduce. **Crossover** operator randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two offspring. For example, the strings 10000100 and 11111111 could be crossed over after the third locus in each to produce the two offspring's 10011111 and 11100100. The crossover operates roughly mimics biological recombination between two single-chromosome organisms. **Mutation** operator 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).

Pseudo-Code for Genetic Algorithms

The following is a pseudo-code for general algorithm approach:

```
1. Define a genetic representation of the system.
2. Generate random population of n chromosomes.
3. Evaluate the fitness of each chromosome in the population.
4. Create a new population by repeating the following steps until new population is
   complete.
    4.1. Select two parent chromosomes from a population according to their fitness
    4.2  With a crossover probability mutate new offspring at each locus.
    4.3  Place new offspring in a new population.
5. Use new generated population for a further run of algorithm
6. If the end condition is satisfied, stop, and return the best solution in current
   population
7. Go to step 3.
```

It can be seen that, each iteration of this process is called a generation. A GA is typically iterated anywhere from 50 to 500 generation or more. The entire set of generations is called a run. At the end of a run there is often one or more highly fit chromosome in the population. Since randomness plays a large role in each run, two runs with different random-numbers seeds will generally produce different detailed behaviors. GA researchers often report statistics (such as the best fitness found in a run and the generation at which the individual with that best fitness was discovered) averaged over many different runs of the GA on the same problem. The simple procedure just described is the basis for most applications of GAs. There are a number of details to fill in, such as the size of the population and the probabilities of crossover and mutation, and the success of the algorithm often depends greatly on these details. As a more detailed example of a simple GA, suppose that l (string length) is 8, $f(x)$ is the number of ones in bit string x, and n (the population size) is 4, $p_c = 0.7$, and $p_m = 0.001$. The initial (randomly generated) population might look like:

Chromosome label	Chromosome string	Fitness
A	00000110	2
B	11101110	6
C	00100000	1
D	00110100	3

A common selection method in GAs is fitness-proportionate selection, in which the number of times an individual is expected to reproduce is equal to its fitness divided by the average of fitnesses in the population.

A simple method of implementing fitness-proportionate selection is "roulette-wheel sampling" [6], which is conceptually equivalent to giving each individual a slice of circular roulette wheel equal in area to the individual's fitness. The roulette wheel is spun, the ball comes to rest on one wedge-shaped slice, and the corresponding individual is selected. in the above $n = 4$ example, the roulette wheel would be spun four times; the first two spins might choose chromosomes B and D to be parents, and the second two spins might choose chromosomes B and C to be parents. Once a pair of parents is selected, with probability p_c they cross over to form two offspring. If they do not cross over, then the offspring are exact copies of each parent. In the above example, Suppose the parents B and D cross over after the first bit position to form offspring E = 101101000 and F = 01101110, and parents B and C do not cross over, instead forming offspring that are exact copies of B and C. Next, each offspring is subject to mutation at each locus with probability p_m. For example, suppose offspring E is mutated at the sixth locus to form E` = 10110000, offspring F and C are not mutated at all, and offspring B is mutated at the first locus to form B` = 01101110. The new population will be the following:

Chromosome label	Chromosome string	Fitness
E`	10110000	3
F	01101110	5
C	00100000	1
B`	01101110	5

Note that, in the new population, although the best string (the one with fitness 6) was lost, the average fitness rose from 12/4 to 14/4.

3. The Knapsack Problem

The knapsack problem is a classic NP-Complete problem, it can be stated as follows: given a knapsack with capacity K_n and a set $U_n = \{u1, u_2,\ldots\ldots,u_n \}$of n items, with each item u_i having a weight w_i and a value v_i , find a subset $U_0 \subset U_n$ such that the total weight of U_0 is no more than K_n, and the total value of U_0 is maximized. In other words, the goal is to fill up hypothetical knapsack with the most expensive loot it can carry, for example a pile of just 50 items presents 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 study, it has been shown that a GA solves such problems, simplifying the illustration a smaller number of items, the pile contains fourteen items, so it provides 16,384 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 Bs etc. Figure1 lists all different items, their weights and values, and maximum number of each type that can fit into the knapsack.

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

Figure 1: Description of the knapsack items

4.1 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 of what have been discovered, but there is no mechanical technique for creating one. Like programming, creating a coding scheme part of the process is science and the other parties art, also like programming, it gets easier with experience. Early researchers used binary encoded starting exclusively, but higher order alphabets work without loss of efficiency and power. Usually the type of coding scheme that is best used depends on the particular problem.

The 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. Figure 2 illustrates the coding of fourteen items into a GA string. Each column in the figure 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 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.

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

Figure 2: The coding scheme for the knapsack example

4.2 Fitness Function

The most difficult and most important concept of genetic programming is the fitness function. The fitness function determines how well a program is able to solve the problem. It varies greatly from one type of program to the next. For example, if one were to create a genetic program to set the time of a clock, the fitness function would simply be the amount of time that the clock was wrong.

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). Thus 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.

In this study, we used a modified moderate penalty approach. The enhanced fitness function, as shown in figure 3, adds up the value of each item (i.e., variable *value*) and subtracts a moderate penalty for violators. The best value for the PENALTY, in this example, was three. This will subtracts three times the amount of excess weight. Figure 4 shows the resulting fitness of the three example strings previously defined.

```
Initialize PENALITY, MAXWEIGHT
compute weight, value
if weight > MAXWEIGHT then
   fitness = value –  PENALITY * (weight – MAXWEIGHT)
else
   fitness = value
```

Figure 3: Pseudo code of the enhanced fitness function

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

Figure 4: The results of the enhanced fitness function using a moderate penalty

4.3 Initialization

After the coding scheme and fitness function are integrated into the GA it becomes ready to run. The GA's first task is to create an initial population of strings. There are many ways to select an initial population; approaches range from randomly setting each character in a string to modify the result of a search made previously by a human. The knapsack example uses a modified weighted random design. Figure 5 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. The complement string has the opposite bit pattern of the original. Note that in the upper half of figure 5 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 figure 5 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 because 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. This rather small population was selected to better illustrate how GAs work.

Strings		1s Count	Prob
U	1 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	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
~W	1 1 0 1 1 1 0 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

Figure 5: Initializing a population of ten strings

4.4 Parent Selections

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. The population constructor then ranks the strings according to their fitness.

After the population constructor returns, the main program enters loop, and stays there until a solution is found. It's unlikely that any strings in the initial generation contain a solution; if no solution found, the program creates 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 best 25th out of 100 strings then it has a 75% chance of becoming a parent.

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 simply it is easy to implement. More important, I suspect that linear rank selection is inherently better behaved than proportional selection, because proportional selection requires 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, select a candidate string from the population, second determines if the candidate string will parent an offspring, by performing a weighted probability based on the string's rank. If the string does not quality to be a parent, then repeats the cycle and randomly selects another string from the population.

4.5 Parent 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. 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. Figure 6 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.

Parent 1	1	1	0	0	0	1	0	0	0	1	1	1	0
Parent 2	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
Child 1	1	0	1	0	0	1	0	1	0	0	1	1	0
Child 2	1	1	0	1	1	1	1	0	0	1	1	1	0

Figure 6: The uniform crossover operator

4.6 Mutation

Sometimes children undergo mutation. The knapsack example uses an interesting operator. 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 operator kicks in when the population shows Fitness diversity and adds new variety by introducing random mutation.

4.7 Finding the answer

The two children now can replace two older strings from the population. 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 a program determines the best answer, or at least, one of the better answers? One approach, used in this study, would be to solve the problem for a fixed solution that meets some predetermined and minimally acceptable value. A second approach would be to run the program until the rate of finding better answers drop off or the rate of improvement of those answers flattens out. The knapsack program ran until it finds the known answer, which is 24.

5. Conclusion

In this study we developed an enhanced solution to the knapsack problem using Genetic Algorithm. The solution of the problem is based on using a new fitness function which reduces the number of iterations needed, as a result reducing the computation time. To demonstrate the effect the proposed solution, an illustrative example was given and also ran using the traditional fitness function. The results show that finding the

optimal answer took, on average, about 29 iterations compared to 126 iterations using traditional fitness function. This gives an improvement of 77% in computation time.

خوارزميه وراثيه لحل مسألة "حقيبة الظهر" بفعالية عالية

عوني منصور الحموري

ملخص

في هذه الدراسة تم استخدام خوارزمية وراثية بشكل فعال لحل مشكلة معروفة بأسم "حقيبة الظهر". والهدف من هذه الدراسة تحسين حل المسألة باستخدام الخوارزمية الوراثية التي تعتمد على طريقة جديدة لحساب قيمة التماثل (Fitness). الخوارزمية الوراثية تستخدم طريقة البحث لتمثل بعض الظواهر الطبيعية، ويعتبر استخدام الخوارزمية الوراثية بسيط نسبياً لإيجاد حل نموذجي او حل تقريبي مفضل للمسائل التي يحتاج حلها وقتاً طويلاً. في هذه الدراسة تم استخدام مثال توضيحي يتكون من 14 عنصر من 5 أنواع مختلفة، وعليه فان عدد المحاولات الممكنة لتعبئة الحقيبة يصل إلى 16384. ونتيجة هذه الدراسة تم إثبات أن استخدام الأسلوب المقترح حسن الوقت اللازم لإيجاد الحل الأمثل بمقدار 77%.

References

[1] Bermemann H. J., ‘The evolution of intelligence: The nervous system as model of its environment,’ *Technical report, no. 1, contract no. 477(17)*, Dept.of Mathematics, Univ. of Washington, Seattle, July, (1958).

[2] Fogel D. B. and Ghozeil A., ‘Schema processing under proportional selection in the presence of random effects,’ *IEEE Transactions on Evolutationary Computation*, 1(4) (1997) 290-293.

[3] Grefenstette J. J., ‘Deception considered harmful,’ *Foundation of Genetic Algorithms* 2, ed. Whitley, L. D., San Mateo, CA: Morgan Kaufmann, (1993) 75-91.

[4] Goldberg D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, (1989).

[5] Holland J. H., ‘Outline for a logical theory of adaptive systems,’ *The Journal of the ACM (JACM)* , 9(3) (1962) 297-314.

[6] Holland J. H., *Hierarchical descriptions of universal spaces and adaptive systems* (Technical Report ORA Projects 01252 and 08226) Ann Arbor: University of Michigan, Department of Computer and Communication Sciences, (1968).

[7] Holland J. H., *Adaptation in natural and artificial systems.* Ann Arbor: The University of Michigan Press, (1975).

[8] Nix E.A. and Vose M. D., 'Modeling genetic algorithms with markov chains', *Annals of Mathematics and Artificial Intelligence,* 5(1) (1992) 79-88.

[9] Poli R., 'Why the schema theorem is correct also in the presence of stochastic effects,' In *Proceeding of the 2000 Congress on Evolutionary Computation,* 1, (2000) 487-492.

[10] Radcliffe Nicholas J., 'Schema Processing,' In :*Handbook of Evolutionary Computation* (Back, T., Fogel, D. B., and Michalewicz, Z.(eds.), Oxford University Press, (1997) B.2.5:1-10.

[11] Vose M. D. and Liepins G. E., 'Punctuated equilibria in genetic search,' *Complex System*, 5 (1991) 31-44.

[12] Ku S. and Lee B., 'A set-oriented genetic algorithm and the knapsack problem,' In *Proceedings of the 2001 congress on evolutionary computation*, 1, (2001) 650-654.

[13] Li K.L., Dai G.M and Li, Q.H., 'A genetic algorithm for the knapsack problem,' *International conference on machine learning and cybernetics*, 3 (2003) 1586-1590.

Since the Knapsack problem is a NP problem, approaches such as dynamic programming, backtracking, branch and bound, etc. are not very useful for solving it. Genetic Algorithms definitely rule them all and prove to be the best approach in obtaining solutions to problems traditionally thought of as computationally infeasible such as the Knapsack problem. Genetic Algorithms (GAs). Genetic Algorithms are computer algorithms that search for good solutions to a problem from among a large number of possible solutions. They were proposed and developed in the 1960s by John Holland, his students, and hi