Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2002), *16*, 23–30. Printed in the USA.
Copyright © 2002 Cambridge University Press 0890-0604/02 $12.50
DOI: 10.1017.S0890060401020030

Solving the 0/1 knapsack problem using an adaptive genetic algorithm

ZOHEIR EZZIANE
Faculty of Science Research Laboratory, Mathematics & Computer Science Department,
United Arab Emirates University, United Arab Emirates

(Received January 19, 1999; Accepted June 14, 2001)

Abstract

Probabilistic and stochastic algorithms have been used to solve many hard optimization problems since they can provide solutions to problems where often standard algorithms have failed. These algorithms basically search through a space of potential solutions using randomness as a major factor to make decisions. In this research, the knapsack problem (optimization problem) is solved using a genetic algorithm approach. Subsequently, comparisons are made with a greedy method and a heuristic algorithm. The knapsack problem is recognized to be NP-hard. Genetic algorithms are among search procedures based on natural selection and natural genetics. They randomly create an initial population of individuals. Then, they use genetic operators to yield new offspring. In this research, a genetic algorithm is used to solve the 0/1 knapsack problem. Special consideration is given to the penalty function where constant and self-adaptive penalty functions are adopted.

Keywords: 0/1 Knapsack Problem; Genetic Programming; Optimization; Self-Adaptive Penalty Functions

1. INTRODUCTION

The integer-knapsack problem seek find a way to fill a knapsack of some given capacity with some elements of a given set of available items of various types in the most profitable way. The input to the problem consists of:

- C, the total weight capacity of the knapsack,
- a positive integer N, the number of items types,
- a vector Q, where $Q[i]$ the available number of items of type i,
- a vector W, where $W[i]$ the weight of each item of type i, satisfying $0 < W[i] \le C$,
- a vector P, where $P[i]$ the profit gained by storing an item of type i in the knapsack,
- all input values are nonnegative integers.

Reprint requests to: Zoheir Ezziane, Faculty of Science Research Laboratory, Mathematics & Computer Science Department, United Arab Emirates University, Al Anin, P.O. Box 17172, United Arab Emirates. E-mail: z.ezziane@uaeu.ac.ae

The problem is to fill the knapsack with elements whose total weight does not exceed C, such that the total profit of the knapsack is maximal. The output is vector F, where $F[i]$ contains the number of items of type i that are put into the knapsack.

The integer-knapsack problem is a special case of the knapsack problem (KP), where discrete items are replaced by materials. The difference is that instead of working with integer numbers, real numbers are used to represent any quantity of material i which does not exceed the available space in the knapsack.

The 0/1 knapsack problem is to find a way to fill a knapsack of a capacity C with a certain number of elements, each having a weight and a profit, in the most profitable way. Thus, the 0/1 KP is the task for a given set of weights $W[i]$, profits $P[i]$, and capacity C, to find a binary vector $x = (x[1], \ldots, x[n])$, such that

$$\text{Maximize } \kappa(x) = \sum_{i=1}^{n} x[i] \times P[i] \tag{1}$$

$$\text{subject to } \sum_{i=1}^{i=n} x[i] \times W[i] \le C. \tag{2}$$

KP is NP-hard (Garey & Johnson, 1979). The problem has been intensively studied in the last 20 years both because of its theoretical interest and its wide practical applicability in operations research, computer science, engineering and management science. Because of the increasing number of the potential applications, numerous algorithms have been developed to solve the KP especially for large problem sizes. Several faster algorithms have been produced which provide good solution and approximate solutions. For example, Sahni (1975) introduced approximation algorithms to the 0/1 KP. Ibarra and Chul (1975) developed a fully polynomial approximation scheme for the KP. Martello and Toth (1988) designed a quite effective algorithm for large-size problems, which is based on the use of a greedy algorithm for solving large KP. Horowitz et al. (1994) also presented a very simple algorithm solving the KP using a greedy method. This paper addresses solving 0/1 KP using a genetic algorithm guided by a self-adaptive penalty function.

2. GENETIC ALGORITHMS

The concept of a genetic algorithm (GA), introduced by John Holland (1973), is a probabilistic heuristic search procedure that mimics the adaptation that nature uses to find an optimal state. A GA uses past and current information to direct the search with expected improved performance and achieve reliable results. It can be viewed as a general purpose optimization method and it has been successfully applied to a large variety of search, optimization, and machine learning tasks. The genetic algorithm approach has been found quite effective in obtaining the solution of a large variety of complex optimization problems, such as multidimensional knapsack (Chu & Beasley, 1998), subset-sum (Spillman, 1995), bin packing (Falkenauer, 1996; Hussain & Sastry, 1997) and timetabling (Kragelund, 1997). In obtaining the solution, the GA adopts a strategy of search based on an intelligent randomization process, and uses a fitness function as criterion.

A genetic algorithm, as is shown in Figure 1, starts with a population of randomly generated individuals. After evaluating all the population, individuals are selected for the application of a crossover operator. Given two parents (selected parents), a crossover operator generates an offspring. Another genetic operator called mutate introduces some random information to the offspring. The newly created offspring replace some or all of the current population, depending on the selection scheme, constructing the new population.

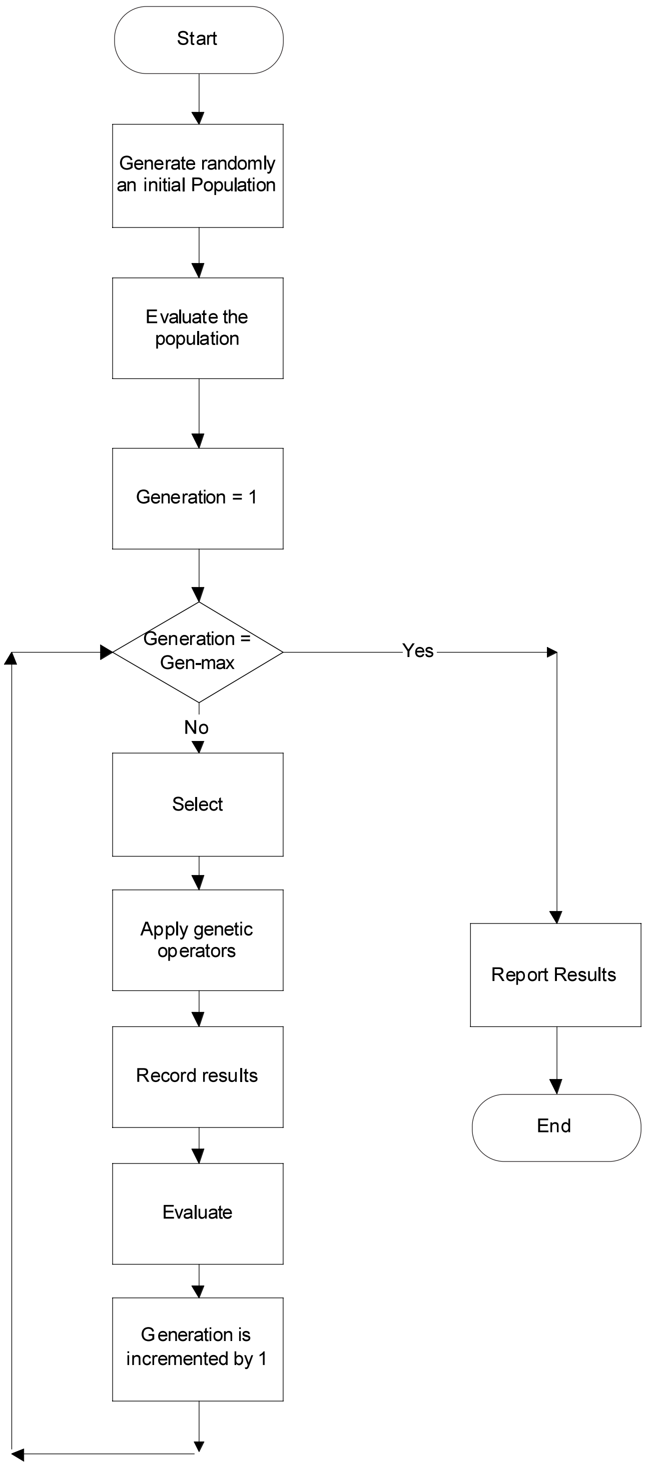

Fig. 1. Flowchart of genetic programming.

3. A GENETIC ALGORITHM FOR THE 0/1 KNAPSACK PROBLEM

When constructing a genetic algorithm for a specific problem, there are five components which need to be defined. The first is the genotype, which represents the individual representation scheme. The second is the evaluation function that plays the role of the environment, rating individuals based on their "fitness" value. The third is the breeding (mating) process. The fourth is the mutation process. The fifth is the assignment of values to various control parameters used by the GA. The last three components could also be considered as the internal workings of a GA.

3.1. Genotype

The natural representation of the 0/1 knapsack problem would be the binary representation, in which every bit represents the existence or not of a certain element of the KP. Thus, each element identification is given by the bit index. Hence a typical population of two individuals for a four-element KP would be:

1001	put elements 1 and 4 in the knapsack,
0110	put elements 2 and 3 in the knapsack.

3.2. Evaluation function

A feasible vector solution x needs to satisfy constraint (2), otherwise it is infeasible. Hence, a penalty is applied to all infeasible solutions in order to decrease their corresponding "fitness." Therefore, two types of evaluation functions used in this research are based on static (constant) and self-adaptive penalty functions. The standard evaluation function for each individual is given by the following expression:

$$\mathit{Evaluation}\ (x) = \sum_{i=1}^{i=n} (x[i] \times P[i]) - \mathit{Pen}(x) \tag{3}$$

$$\mathit{Maximum\ Profit\ Possible} = \text{Max}\ \kappa = \sum_{i=1}^{i=n} P[i]. \tag{4}$$

A vector solution x is optimal when $\mathit{Evaluation}\ (x) = \text{Max}\ \kappa$.

With a simple change of variable, a better evaluation function is designed for a better performance monitoring.

$$\mathit{New\text{-}Evaluation}\ (x) = 1 - ((\text{Max}\ \kappa - \mathit{Evaluation}\ (x))/\text{Max}\ \kappa). \tag{5}$$

Here, a solution x is optimal when $\mathit{New\text{-}Evaluation}\ (x) = 1.0$, and the worst solution is when $\mathit{New\text{-}Evaluation}\ (x) = 0.0$, where x represents the binary vector (individual) of KP elements;

where

W$[i]$ = random $[0..1]$; W$[i]$ represents the weight of element i;

P$[i]$ = W$[i]$ + 0.5; P$[i]$ represents the profit of element i (profits and weights are correlated);

$$\text{Capacity} = 0.5 \times \left(\sum_{i=1}^{i=n} W[i]\right);\ \text{capacity of the knapsack.} \tag{6}$$

Constant Penalty Function:

```
{
Pen = 0; Temp = $\left(\sum_{i=1}^{i=n} (x[i] \times W[i])\right) - \mathit{Capacity}$
If (Temp $\geq$ 0) Then Pen = SQRT(Temp)        (7)
Return Pen
}
```

Self-adaptive Penalty Function:

```
{
    Pen = 0; Temp = $\left(\sum_{i=1}^{i=n} (x[i] \times W[i])\right)$
    Weight-Factor = Temp/Capacity        (8)
    Loop-Iteration = (Integer)(Ceiling(a $\times$ Weight-Factor + b))        (9)
                    // a and b are to be determined
    Temp = Temp $-$ Capacity
    If (Temp $\geq$ 0) Then
            pen = SQRT(Temp)
            For (i=1; i $\leq$ Loop-Iteration; i++)
                    Pen = SQRT(Pen)
    Return Pen
}
```

What makes a vector solution x infeasible is that the total weight of the individual x has exceeded the knapsack capacity. Since the capacity is considered to be half the total possible knapsack weight, it means that $\sum_{i=1}^{i=n}(x[i] \times W[i]) = \rho * \mathit{Capacity}$, where $1 < \rho \leq 2$.

When $\rho = 2$, this indicates that the total weight of the solution vector x is twice the capacity of the knapsack. Here a maximum penalty is applied at the individual x, which is translated by setting a minimum value to *Loop-Iteration*.

When $\rho = 1 + \epsilon$, where $\epsilon \ll 1$, this indicates that the total weight of vector x is slightly larger than the knapsack capacity. Here a minimum penalty is applied at the individual x, which makes *Loop-Iteration* to be assigned a maximum value. Assume $\epsilon = 0.01$ (1% of the total knapsack capacity). The linear function between *Loop-Iteration* and ρ is determined as follows:

$$\mathit{Loop\text{-}Iteration}\ (\rho) = a * \rho + b. \tag{10}$$

The assignments of the parameters (*Loop-Iteration*, a, and b) are explained in the next section.

3.2.1. *Setting values for Loop-Iteration, a and b*

For large knapsack size, say 20,000 elements, $W[i] = 1.0$ and $x[i] = 1.0$ for all elements i. This represents the worst case scenario. Knapsack capacity [Eq. (6)] becomes $0.5 \times 20{,}000 = 10{,}000$.

$$\mathit{Weight\text{-}Factor}\ [\mathit{Eq.}\ (8)] = \left(\frac{20{,}000}{10{,}000}\right) = 2.0$$

$$\mathit{Pen}\ [\mathit{Eq.}\ (7)] = \mathit{SQRT}\ (20{,}000.0 - 10{,}000.0) = 100.0.$$

Here maximum penalty is applied to the individual. Therefore, make *Loop-Iteration* minimum, that is, 1. Hence, $Pen = SQRT\,(100.0) = 10.0$ (returned penalty).

The scenario for a minimum penalty happens when the total weight of the individual x is slightly larger than the knapsack capacity (say, by 1%), which makes $\rho = 1.01$. Here a minimum penalty is returned by the self-adaptive penalty function as follows:

$$\begin{aligned} &\textit{Weight-Factor} = 1.01 \\ &\qquad Pen = SQRT(10{,}100.0 - 10{,}000.0) \\ &\qquad\quad = SQRT(100.0) = 10.0. \end{aligned}$$

Performing SQRT four times will return $Pen = 1.1547$ (returned penalty).

Thus, for a large knapsack of 20,000 elements, the adaptive penalty function used in this research computes the minimum and the maximum penalties as approximately 1 and 10, respectively. Putting those values into Eq. (10), the following equations are obtained:

$$\rho(1.01) = 4 \tag{11}$$

$$\rho(2) = 1. \tag{12}$$

After solving the system of equations (10), (11), and (12), it yields $a = -3.03$ and $b = 7.06$.

Table 1 lists some values for *Weight-Factor* together with the corresponding *Loop-Iteration* which is used in the executions of the GA.

3.3. Internal workings of the GA

3.3.1. Selection

The selection scheme is responsible for selecting two parents (individuals). Goldberg (1989) reported various selections types, such as stochastic remainder, elitism, crowding factor model, tournament, and roulette wheel. The one used in this research paper is elitism. Elitism is a mechanism which is employed in a GA that ensures that the most highly fit individuals of the population are passed on to the next generation without being altered by genetic operators.

3.3.2. Crossover

The power of GAs arises from crossover. Crossover causes a structured, yet randomized exchange of genetic material between solutions, with the possibility that the “fittest” solutions generate “better” ones. A crossover operator should preserve as much as possible from the parents while creating an offspring. Crossover happens only with some probability P_c. The crossover operator used here is the simple one-point crossover.

The way the one-point crossover works as follows: an integer *Pos* is selected uniformly at random within the range $[1, \textit{Individual-Length} - 1]$, then two new offspring are created by swapping all bits between positions $Pos + 1$ and and *Individual-Length*. For example, using the character “|” to note random position *Pos*, consider the selected individuals (parents) Ind_1 and Ind_2 from the KP initial population:

$$\begin{aligned} Ind_1 &= 11000|011 \\ &\qquad\qquad (Pos = 5) \\ Ind_2 &= 01010|010 \end{aligned}$$

The resulting crossover yields two new individuals (offspring) Ind_1^* and Ind_2^*:

$$Ind_1^* = 11000010$$

$$Ind_2^* = 01010011$$

3.3.2. Mutation

Since the crossover operator preserves information already existing in the parents, but if is the only genetic operator used in the GA, it will restrain the diversity of the population. Mutation involves the modification of each bit of an individual with some probability P_m. Although the mutation operator has the effect of destroying the structure of a potential solution, chances are it will yield a better solution. Mutation in GAs restores lost or unexplored genetic material into the population to prevent the premature convergence of the GA. It has also been stated in Hussain and Sastry (1997) that mutation acts like a local improvement operator.

Uniform mutation is used in this research which simply works here as follows:

Let $Ind_1 = 11000011$ be the individual to be mutated at position 3.

The new individual after mutation is $Ind_1^* = 11100011$.

Crossover operators such as multipoint, order-based, position-based, and mutation operators such as nonuniform, scramble sublist, smooth, and many more are listed in Michalewicz (1996).

Table 1. *Loop-Iteration versus Weight-Factor*

	Weight-Factor						
	1.0001	1.001	1.01	1.10	1.4	1.7	1.9
Loop-Iteration	5	5	4	4	3	2	2

3.4. Control parameters

For all the experimental analyses and discussions, the probability of crossover $P_c = 0.8$, the probability of mutation $P_m = 0.01$, and maximum number of generations is variable, depending on the size of the KP, whereas the population size is 30.

4. EXPERIMENTS AND RESULTS

The results were averaged in all of the conducted experiments. SGA represents the GA with static (constant) penalty function, and AGA represents the GA with self-adaptive penalty function. Once the SGA and AGA start executing, the parameters are kept identical for the entire experiment.

The GA approach is compared with greedy method (GM) and heuristic algorithm (HA). GM makes locally optimal choices at each step in the hope that these choices will produce a globally optimal solution. The following variables are used in both the GM and the HA:

$p[1..n]$ and $w[1..n]$ contain the profits and weights respectively of the n objects, ordered such that $(p[i]/w[i]) \geq (p[i+1]/w[i+1])$.

$P_{\max}$ represents the profit of the solution and *cap* is the knapsack size.

The general guidelines of the GM for the knapsack problem is as follows:

```
Algorithm GreedyKnapsack
// $sol[1..n]$ is the solution vector.
//For example at the end of GM, assume $sol = [10010010]$.
//This solution indicates that only elements 1, 4, and 7 are
  included in the solution for a
//maximum profit equal to $P_{\max}$ with respect to the knap-
  sack capacity.
  {
    For $(i = 1; i \leq n; i{+}{+})$ $sol[i] = 0$;
          // Initialize the solution vector
    $temp = cap$; $P_{\max} = 0$;
    For $(i = 1; i \leq n; i{+}{+})$
    {
      If $(w[i] > temp)$ break
            // Check if the weight exceeded the KP capacity
      $sol[i] = 1$;        $temp = temp - w[i]$;
            $P_{\max} = P_{\max} + p[i]$;
    }
  }
```

Horowitz et al. (1994) proposed the heuristic algorithm for the knapsack problem called the epsilon approximation.

```
Algorithm EpsilonApprox$(p, w, cap, n, k)$
// $k$ is a nonnegative integer that defines the order of the
   algorithm
// In the following experiments, $k = 2$.
{
$P_{\max} = 0$;
for all combinations $I$ of size $\leq k$ and $weight \leq cap$ do
    // All $\sum_{i=0}^{k} C_i^n$ different subsets $I$ consisting of at most $k$
       of the $n$ elements are generated.
    // If the currently generated subset $I$ is such that
       $\sum_{i \in I} w[i] >$ cap, it is considered as infeasible.
    // Otherwise, the space remaining in the knapsack $(cap -
       \sum_{i \in I} w[i])$ is filled using the function Lbound.
    {
        $P_I =$
        $P_{\max} = \max(P_{\max}, P_I + \text{Lbound}(I, p, w, m, n)$;
    }
Return $P_{\max}$;
}// End of EpsilonApprox
```

The Function Lbound is given as follows:

```
Function Lbound$(I, P, W, Cap, N)$
{
    $Sum = 0$;    $Temp = Cap - \sum_{i \in I} w[i]$;
    for $i = 1$ to $N$ do
            if $(i \notin I)$ and $(W[i] \leq Temp)$ then
                    {
                        $Sum = Sum + P[i]$    $Temp = Temp - W[i]$;
                    }
            return $Sum$;
}
```

4.1. Analysis of the first experiment

Figure 2 shows that the HA outperforms the GM at every attempt, and sometimes got better results than the SGA (100 G, i.e., after 100 generations). Meanwhile, it is noticeable that after 100 generations, SGA and AGA were exchanging the leading role. But after 1000 generations, it is always AGA that obtains the best results.

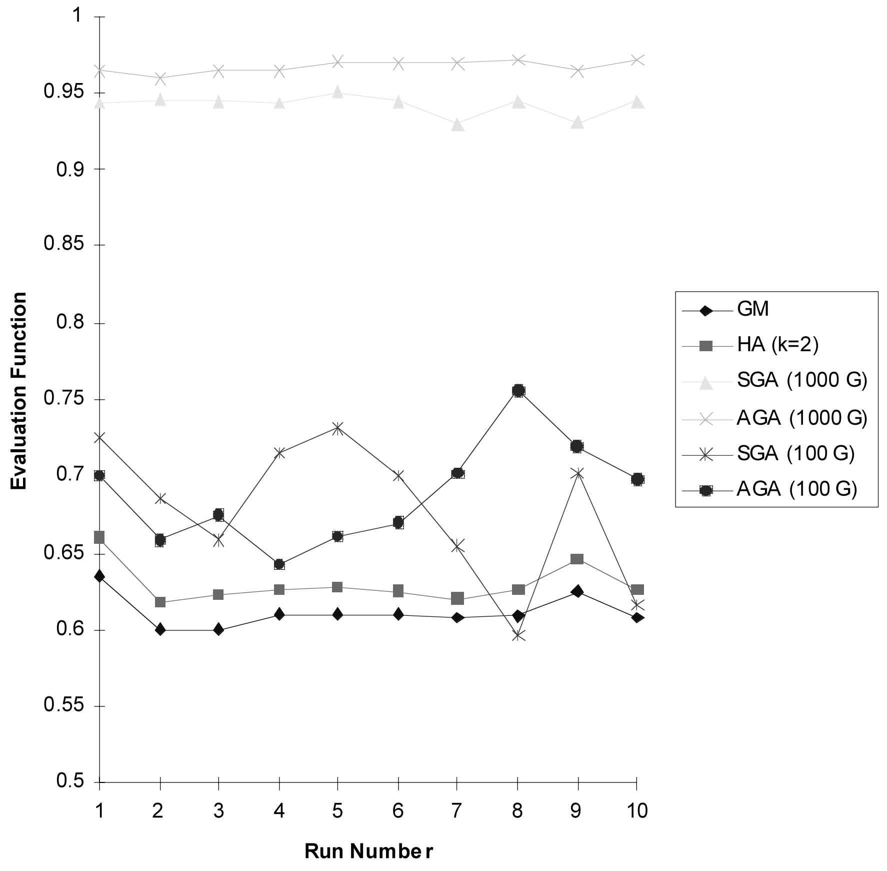

Fig. 2. 100-element 0/1 KP.

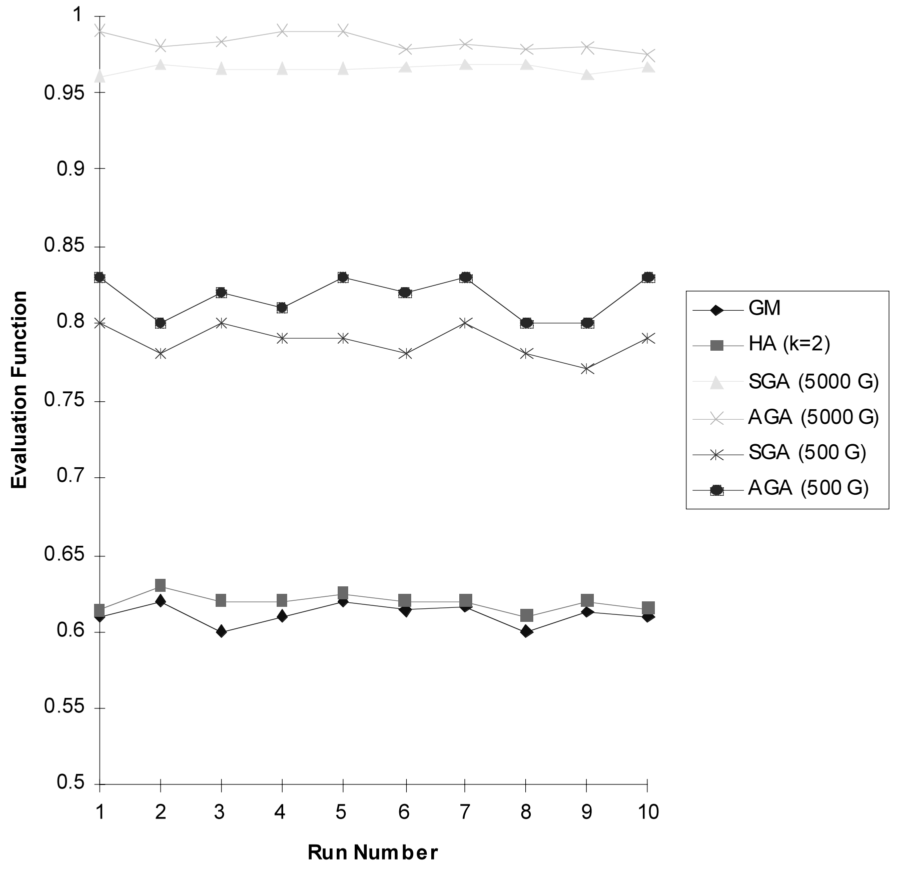

Fig. 3. 500-element 0/1 KP.

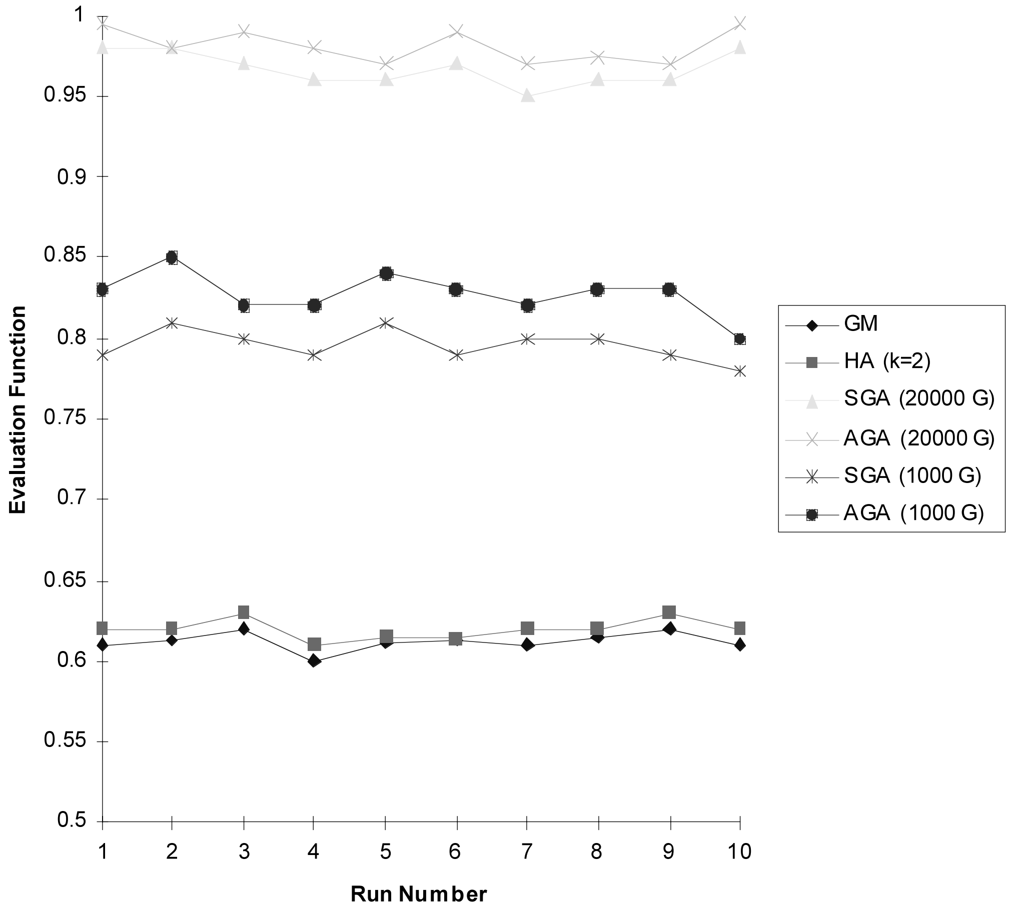

Fig. 4. 1000-element 0/1 KP.

4.2. Analysis of the second experiment

When the KP size was changed to 500 elements, the HA got better results than the GM but less than the genetic approach. Here as it is depicted in Figure 3, AGA performed much better than the SGA after 500 generations as well as after 5000 generations.

4.3. Analysis of the third experiment

During the last experiment, the KP size was changed to 1000 elements (Fig. 4). The AGA produced the best performance after the two snapshots (i.e., 1000 and 20,000 generations). This increase in the number of generations is twofold. First, the size of the KP is set to 1000 and second, a convergence to a close to 100% performance (perfect match) was targeted.

Those experiments were conducted various numbers of times, with the same parameter settings, and the AGA performance was exactly unchanged, that is, solutions with evaluation function reaching 0.995. At this point, the solution to the KP yielded so far is quite sufficient. After investigating further, and replacing the constant penalty function by the self-adaptive penalty function, an improved solution was observed. Note that in this research, no hill-climbing (local search) technique was added to the AGA in order to improve its performance.

5. CONCLUSIONS

This work applied a genetic algorithm to optimize to the 0/1 knapsack problem. The first GA used a constant penalty function, and the second used an adaptive penalty function. As has been observed during the experiments conducted in this paper, the AGA was able to generate the best results, for different KP sizes, when compared with SGA, GM, and HA.

Self-adaptation (or adaptation of parameters) is one of the most promising areas of genetic algorithms; it tunes the program during its execution while solving the problem. Furthermore, the concept of adapting control parameters during the program run fits also in the arena of artificial life and evolutionary computation. When this concept is used in solving combinatorial optimization problems, it yields better performance.

REFERENCES

Chu, P.C., & Beasley, J.E. (1998). Genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics 4(1)*, 63–86.

Falkenauer, E. (1996). A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics 2*, 5–30.
Garey, M.R., & Johnson, D.S. (1979). *Computers and intractability: A guide to the theory of NP-Completeness*. San Francisco: W.H. Freeman & Co.
Goldberg, D.E. (1989). *Genetic algorithms in search, optimization and machine learning*. Addison Wesley, Reading, Massachusetts.
Holland, J.H. (1973). Genetic algorithms and the optimal allocations of trials. *SIAM Journal of Computing 2(2)*, 88–105.
Horowitz, E., Sahni, S., & Rajasekaran, S. (1994). *Computer algorithms*. New York: Computer Science Press, W.H. Freeman & Co.
Hussain, S.A., & Sastry, V.U.K. (1997). Application of genetic algorithm for bin packing. *International Journal of Computer Mathematics 63*, 203–214.
Ibarra, O.H., & Chul, E.K. (1975). Fast approximation algorithms for the knapsack and the sum of subset problems. *Journal of the ACM 22(4)*, 463–468.
Kragelund, L.V. (1997). Solving a timetabling problem using hybrid genetic algorithms. *Software-Practice and Experience 27(10)*, 1121–1134.
Martello, S., & Toth. P. (1988). A new algorithm for the 0-1 knapsack problem. *Management Science 34*, 633–644.
Michalewicz, Z. (1996). *Genetic algorithms + data structures = evolution programs*, 3rd ed., Springer.
Sahni, S.K. (1975). Approximate algorithms for the 0/1 knapsack problem. *Journal of the ACM 22(1)*, 115–124.
Spillman, R. (1995). Solving large knapsack problems with a genetic algorithm. *Proc. IEEE Int. Conf. Systems, Man, and Cybernetics*, pp. 632–637. Vancouver, BC, Canada. Piscataway, NJ: IEEE.

Zoheir Ezziane received his Bachelor's degree in engineering from the Université des Sciences et de la Technologie de Houari Boumediene (USTHB) in Algeria in 1988, his Master's degree in computer science from the University of Central Florida in 1991, and his Ph.D. in engineering from Florida Atlantic University in 1994. He was a research associate at the Material Handling Research Center in Boca Raton, Florida, from 1992 to 1994, where he designed and implemented two software packages, the Consumer Integrated Packaging & Logistics and the Neural Network Warehouse Management System. Since 1995, he has been a lecturer and researcher in the Mathematics and Computer Science Department at the United Arab Emirates University. His research interests include machine learning, genetic algorithms, data mining, and network security.