Performance Analysis of Genetic Algorithm for Solving the Multiple-Choice Multi-Dimensional Knapsack Problem

Khandaker Moinur Rahman¹, Syed Ishtiaque Ahmed²
¹Department of Computer Science and Engineering, IBAIS University, Dhaka, Bangladesh
Email: khandaker.moinur.rahman@gmail.com
²Department of Computer Science and Engineering, BUET, Dhaka, Bangladesh
Email: ishtiaque@csebuet.org

Abstract—In this paper, we approximately solved the multiple-choice multi-dimensional knapsack problem (known as MMKP), an NP-Hard combinatorial optimization problem using genetic algorithm. The performance of genetic algorithm has been evaluated on several problem instances. Parameters of genetic algorithm have been varied to analyze the effect upon result. This performance has been compared to other existing algorithms for solving MMKP. Encouraging results have been obtained.

Index Terms—genetic algorithm, knapsack problem, admission control, multimedia system

I. INTRODUCTION

In the MMKP, let there be $n$ groups of items. Group $i$ has $l_i$ items. Each item of the group has a particular value and it requires $m$ resources. The objective of the MMKP is to pick exactly one item from each group so that total value of the collected items is maximized, subject to $m$ resource constraints of the knapsack [7]. In mathematical notation, let $v_{ij}$ be the value of the $j$th item of the $i$th group, $r_{ij} = (r_{ij1}, r_{ij2}, \ldots, r_{ijm})$ be the required resource vector for the $j$th item of the $i$th group and $\bar{R} = (R_1, R_2, \ldots, R_m)$, be the resource bound of the knapsack. Now, the problem is to find $V = \sum_{i=1}^{n} \sum_{j=1}^{l_i} x_{ij} v_{ij}$, (objective function), so that,

$$\sum_{i=1}^{n} \sum_{j=1}^{l_i} x_{ij} r_{ij} \leq R_k$$

(resource constraints), where, $V$ is the value of the solution, $k = 1, 2, \ldots, m$, $x_{ij} \in \{0,1\}$ are the picking variables, and $\sum_{j=1}^{l_i} x_{ij} = 1$.

II. RELATED WORKS AND OUR WORKS

Several approaches exist for solving KP and its variants. Most of the researches on knapsack problems deal with the simpler constraint version ($m = 1$ and $n = 1$). To our knowledge, very few papers dealing directly with the MMKP are available. Moser et al. [6] have designed an approach based upon the concept of graceful degradation. Khan et al. [5] have tailored an algorithm based on the aggregate resources. Hifi et al. [2] proposed a guided local search-based heuristic. Rafael Parra-Hernández and Nikitas J. Dimopoulos [3] also proposed a new heuristic for MMKP. Finally, M Hifi, M Michrafy and A Sbihi proposed a Reactive Local Search-based (RLS) algorithm [1] using deblocking and degrading strategies. They also presented a modified version of RLS called MRLS. In this paper, we proposed the use of genetic algorithm for MMKP. In addition, we analyzed the performance of genetic algorithm and compared the experimental results obtained by other algorithms.

III. MAPPING OF THE GENETIC ALGORITHM TO MMKP

Here, Genetic Algorithms was used to solve the MMKP where one has to maximize the benefit of group of objects in a knapsack without exceeding its capacity.

A. Encoding of the chromosomes

To represent the whole population of chromosomes a tri-dimensional array (chromosomes $[\text{Size}][\text{number of groups}][\text{number of items}]$) was
used. Size stands for the number of chromosomes in a population. The second and third dimensions represent the groups and the items respectively that may potentially be included in the knapsack.

**B. Fitness Function**

Fitness of each chromosome was calculated by summing up the benefits of the items that were included in the knapsack, while making sure that the capacity of the knapsack was not exceeded.

**C. Group Selection**

The selection process was based on fitness. Array of chromosomes was sorted according to their fitness values in ascending order. The population was categorized into five groups: (0 ... Size/5), (Size/5 ... 2*Size/5), (2*Size/5 ... 3*Size/5), (3*Size/5 ... 4*Size/5) and (4*Size/5 ... Size). Next, a chromosome was randomly chosen from the first, second, third, fourth and fifth group with 5%, 10%, 15%, 25% and 45% probability respectively. Thus, the fitter a chromosome was, the more chance it had to be chosen for a parent in the next generation. This combination of grouping was found most effective.

**E. Crossover**

Single point crossover was used in this algorithm which randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two children from parents.

**F. Mutation**

Mutation was performed on random bit position of the chromosomes with probability ranging from 0.2% up to 1.50%.

**G. Elitism**

Elitism was used where two of the fittest chromosomes are copied without changes to the new population, so the best solutions found will not be lost.

**H. Proposed Algorithm**

1. Select an initial generation, $\omega = (x_1, x_2, \ldots, x_n)$ and sort its chromosomes according to fitness
   loop counter initially set to 0;
   no_improvement_count initially set to 0;
2. Repeat;
3.   prev_max = maximum fitness of current generation;
4.   copy two top scored chromosomes to the new generation without any change;
5.   Repeat;
6.   Set index = 0;
7.   random_value = Random (0,1);
8.   first_parent = Selected a single chromosome from categories of current generation using random_value;
9.   random_value = Random (0,1);
10.  second_parent = Selected a single chromosome from categories of current generation using random_value;
11.  perform crossover on first_parent and second_parent;
12.  index = index + 2;
13.  copy two children generated to the next generation;
14.  Until index = (population_size/2) –2;
15.  loop_counter = loop_counter + 1;
16.  Replace the current generation with the new generation;
17.  Sort current generation;
18.  Perform mutation on the current generation;
19.  cur_max = maximum fitness of current generation;
20.  If cur_max $\leq$ prev_max, then
21.     no_improvement_count = no_improvement_count + 1;
22.  Else no_improvement_count = 0;
23.  End If
24.  If(no_improvement_count $>$ improve_threshold), then break;
25.  Until (loop_counter = maximum number of generations)

**IV. PERFORMANCE OF GENETIC ALGORITHM**

A total of 13 problem instances [4] were considered. Table 1 evaluates the performance of genetic algorithm compared to other existing algorithms on the set of problem instances. Improvement of solution with increasing runtime for problem instance 111 is shown in Figure 1 as a sample. The population size and mutation ratio were also varied for all 13 problem instances. Solution of genetic algorithm always improved with increasing runtime and increasing population size. But the same cannot be claimed in case of mutation ratio.

**V. CONCLUSION**

Although the performance of GA was not the best for MMKP in some of the cases compared to other state of the art algorithms but developing some heuristics can surely
Table 1: Performance comparison of Different Algorithms

<table>
<thead>
<tr>
<th>Problem Instances</th>
<th>Opt/Best</th>
<th>RLS</th>
<th>MRLS*</th>
<th>KLMA</th>
<th>Der_Algo</th>
<th>Genetic Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>I01</td>
<td>173</td>
<td>173</td>
<td>173</td>
<td>167</td>
<td>173</td>
<td>173</td>
</tr>
<tr>
<td>I02</td>
<td>364</td>
<td>364</td>
<td>364</td>
<td>354</td>
<td>356</td>
<td>361</td>
</tr>
<tr>
<td>I03</td>
<td>1602</td>
<td>1595</td>
<td>1602</td>
<td>1533</td>
<td>1553</td>
<td>1598</td>
</tr>
<tr>
<td>I04</td>
<td>3597</td>
<td>3564</td>
<td>3597</td>
<td>3437</td>
<td>3502</td>
<td>3591</td>
</tr>
<tr>
<td>I05</td>
<td>3949.59</td>
<td>3905.90</td>
<td>3905.70</td>
<td>3899.10</td>
<td>3943.20</td>
<td>3905.70</td>
</tr>
<tr>
<td>I06</td>
<td>4799.30</td>
<td>4799.30</td>
<td>4799.30</td>
<td>4799.30</td>
<td>4799.30</td>
<td>4799.30</td>
</tr>
<tr>
<td>I07</td>
<td>24587*</td>
<td>24121</td>
<td>24587</td>
<td>23912</td>
<td>23983</td>
<td>24423</td>
</tr>
<tr>
<td>I08</td>
<td>36877*</td>
<td>36110</td>
<td>36877</td>
<td>35979</td>
<td>36007</td>
<td>36641</td>
</tr>
<tr>
<td>I09</td>
<td>49167*</td>
<td>48291</td>
<td>49167</td>
<td>47901</td>
<td>48048</td>
<td>48691</td>
</tr>
<tr>
<td>I10</td>
<td>61437*</td>
<td>60291</td>
<td>61437</td>
<td>59811</td>
<td>60176</td>
<td>60770</td>
</tr>
<tr>
<td>I11</td>
<td>73773*</td>
<td>72283</td>
<td>73773</td>
<td>71760</td>
<td>72003</td>
<td>72958</td>
</tr>
<tr>
<td>I12</td>
<td>86071*</td>
<td>84446</td>
<td>86069</td>
<td>84141</td>
<td>84160</td>
<td>85015</td>
</tr>
<tr>
<td>I13</td>
<td>98429*</td>
<td>96850</td>
<td>98429</td>
<td>96003</td>
<td>96103</td>
<td>97050</td>
</tr>
</tbody>
</table>

Figure 1: Solution vs. runtime (seconds) for problem instance I11

improve the performances. Our research clearly indicates that possibility as shown in the statistics presented in this paper.

REFERENCES