2012 4th International Conference on Computer Research and Development
IPCSIT vol.39 (2012) © (2012) IACSIT Press, Singapore

Meta-heuristics for Multidimensional Knapsack Problems

Zhibao Mian[+]

Computer Science Department, The University of Hull, Hull, UK

Abstract. In this research, we present the use of genetic algorithms and constraint handling techniques used in genetic algorithms to solve Multidimensional Knapsack Problems (MKP). We also investigate and compare how the use of different genetic algorithm operators and parameters could obtain different solutions for the MKP. Three proposed genetic algorithms (basic GA, GA with penalty function1, and GA with penalty function2) are implemented. The performance of these genetic algorithms is compared based on some small MKP. Finally, a well-known MKP is modified as a fictitious capital budgeting problem, and the results are reported. The experiment results show that genetic algorithms are capable to solve MKP and capital budgeting problems.

Keywords: Genetic Algorithm; Multidimensional Knapsack Problem; Constraint Handling Techniques; Capital Budgeting Problem

1. Introduction

The Multidimensional Knapsack Problem (MKP) is a well-known NP-hard problem [1], which arises in several practical problems such as capital budgeting problem. The MKP can be defines as follows:

$$\text{Maximize } \sum_{j=1}^{n} p_j x_j \quad (1)$$

$$\text{Subject to } \sum_{j=1}^{n} r_{ij} x_j \le b_i, \ \mathrm{i} = 1, \ldots, \mathrm{m}, \quad (2)$$

$$x_j \in \{0,1\}, \ \mathrm{j} = 1, \ldots, \mathrm{n}. \quad (3)$$

Where

n is the number of items,

m is the number of constraints,

$p_j \ge 0$ is the profit of the *jth* item,

$r_{ij} \ge 0$, for $\mathrm{i} = 1, \ldots, \mathrm{m}$, is the weight of the *jth* item,

and $b_i \ge 0$, for $\mathrm{i} = 1, \ldots, \mathrm{m}$, is the capacity of the knapsack.

There are m constraints described in equation (2) and each of them is called a knapsack constraint, hence, the MKP is also called the m-dimensional knapsack problem.

Many practical problems can be defined as a MKP. For example, in capital budgeting problem, project j has profit p_j and consumes r_{ij} units of resource i. The goal is to find a subset of n projects such that the total profit is maximized without violating all resource constraints.

In last two decades, a number of papers involving the use of Genetic Algorithms (GAs) to solve MKP. Genetic algorithms, sometimes called natural selection, are search methods based on the principles of survival of the fittest. It was introduced and originally developed extensively by John Holland [6] in the

[+] Corresponding author. Tel.: + (44)1482465045; fax: +(44)1482466666.
E-mail address: Z.Mian@2009.hull.ac.uk

middle of 1970s. The readers are referred to [4, 5] for a detailed steps of construction of a basic genetic algorithm.

2. Introduction of constraint handling techniques

Constraints must not be violated. However, either some encoding methods or after the use of recombination and mutation operators, may cause illegal individuals. In many of the applications where GAs are intended to be the problem of finding a feasible solution is itself NP-hard [2]. Michalewicz [7] also indicated that each evolutionary computation technique used to a specified problem should address the issue of handling unfeasible individuals.

Two constraint handling techniques, i.e., the use of penalty functions and repair algorithms will be implemented in our proposed GAs.

(1) Penalty functions. One most advantage of penalty function is that infeasible solutions are allowed in a population so that one feasible and highly fit solution might be generated by the combination of two infeasible individuals.

(2) Repair algorithm. In MKP, it is relatively easy to repair an infeasible solution and transform it into a feasible one. Repair functions are not only used to make an infeasible solution feasible, but also to improve the fitness of solutions.

3. GAs for multidimensional knapsack problem

In this section, a complete implementation of genetic algorithms will be introduced. The experiments are based on those standard MKP test problems from OR-Library issued by [2].

3.1. Population size and initial population

For population size, it is expected that the bigger the population is the better the results will be found. However, a bigger population also leads to an additional computation time per generation. We will discuss the differences between population size equals to 100 and 200 later on.

We adopt a random initialisation scheme in order to achieve sufficient diversification. In the first instance, each of the initial feasible solution was randomly constructed. However, it is very difficult to find a feasible solution for those large test problems. We consider it is due to the fact that finding a feasible solution is itself NP-hard. Hence, a primitive heuristic was used where a solution with all zeros is initially generated and a variable is randomly selected repeatedly and set to one if the solution is feasible. The heuristic terminates while the selected variable cannot be added to the solution. The pseudo-code for the initialisation as follows (see Algorithm 1).

Algorithm 1. Initialise P(0) for the MKP

Let: $I = \{1, \ldots, \mathrm{m}\}$ and $J = \{1, \ldots, \mathrm{n}\}$;
Let: $R_i =$ the accumulated resources of constraint i in S; /* S is solution set*/
for $k = 1$ to N do
    set $S_k[j] \leftarrow 0$, $\forall j \in J$;
    set $R_i = 0$;
    set $T \leftarrow J$; /* T is a dummy set*/
    randomly select a $j \in T$ and set $T \leftarrow T - j$;
    while $R_i + r_{ij} \le b_i$, $\forall i \in I$ do
        set $S_k[j] \leftarrow 1$;
        set $R_i \leftarrow R_i + r_{ij}, \forall i \in I$;
        randomly select a $j \in T$ and set $T \leftarrow T - j$;
    end while
end for

3.2. Selection operators

Parent selection is the task of assigning reproductive opportunities to each individual in the population. Two selection strategies, i.e., roulette-wheel selection and tournament selection, are implemented in our experiments and the experimental results will be compared in section 3.

3.3. Crossover operators

The idea behind the crossover is let offsprings to inherit some both parents' properties. Two crossover operators are tested in our experiment, i.e., one-point crossover and uniform crossover.

3.4. Mutation operator

Once a child solution has been generated from crossover, a mutation procedure is performed to change some selected bits in the child solution at random.

3.5. Replacement

After applying the above three operators, the new offsprings have been generated. However, how to replace the old population (generation) with the new offsprings is a question. We adopt the steady-state-no-duplicates strategy, which replaces the worst one of the current population by the one new generated individual, and checks that no duplicate offsprings are added to the population. This adds the computational overhead but more search space is explored.

3.6. Penalty functions

It is very likely that one good solution might be generated by the combination of two infeasible solutions. We tested two penalty functions (penalty function 1 and 3) proposed in [9], and the experiment results will be analysed and discussed in section 3.

3.7. Construct the parameters and operators in GAs

Table I shows the parameters and strategies used in our experiments. Since there is not one set of parameters that is suitable for all types of problems, we have to test during the experiment in order to find out the best set. Some of the parameters and strategies will be set the same as those showed in the literature [2, 3, and 8] since they have been improved to generate better solutions.

Table. I: Parameters used in GAs

parameters / strategy	*setting*	
population size	100	
initialisation	initialise algorithm	
selection (two options)	roulette wheel	tournament ($k = 2$)
crossover (two options)	one point	uniform
crossover probability	0.9	
mutation	bit flip	
mutation probability	*2/n*	
replacement strategy	steady-state-no-duplicates	
stopping criteria	run over 10^6 generations	
improvements	penalty function, repair operator, local improvement operator	

4. Experiments and analyssis

4.1. Selection and crossover operators selection

We have conducted two selection methods and two crossover methods. The aim of this experiment is to find out one combination of these two methods in order to generate the best solution. Fig. 1 shows the results based on one largest problem.

In Fig. 1, the tournament selection with uniform crossover increases fast and produces the best solutions. Hence, this combination will be chosen in our experiment.

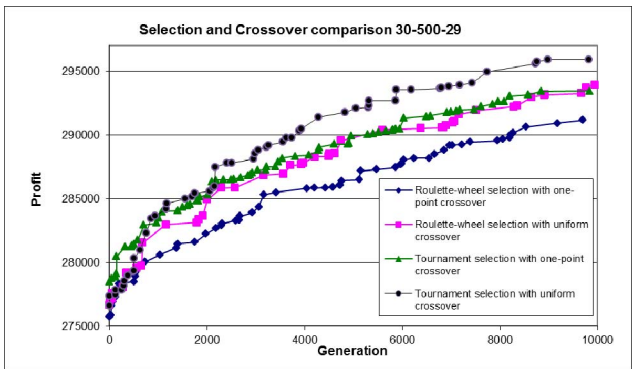

Fig. 1: Test case m=30, n=500, problem 29, profit with different selection and crossover operators.

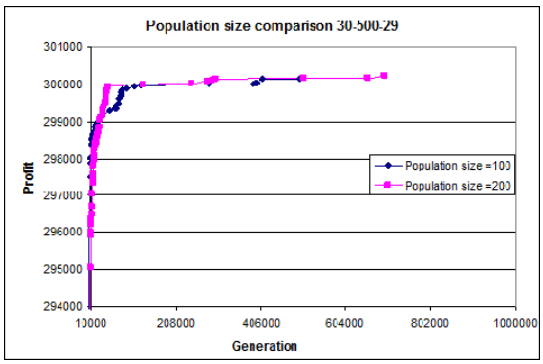

Fig. 2: Test case m=30, n=500, problem 29, profit with different population sizes.

4.2. Population size selection

In this section, different population sizes (100 and 200) are tested based on some of largest problems. The fragment data for profit with different population sizes from 10,000 to 1,000,000 generations are displayed in Fig. 2.

In Fig. 2, it can be seen that the genetic algorithm with population size equals to 200 increases faster and generates better solutions than those of population size equals to 100. However, there is no significant gap between these two sizes. We decide to choose 100 as the population size to save experiment time.

4.3. Algorithms selection

Three GAs are implemented in this experiment, i.e., the basic GA, GA with penalty function 1, and GA with penalty function 2. The aim of this section is to choose one of the best algorithms to solve the MKP. Fig. 3 shows the experiment results in solution quality base on one small problem.

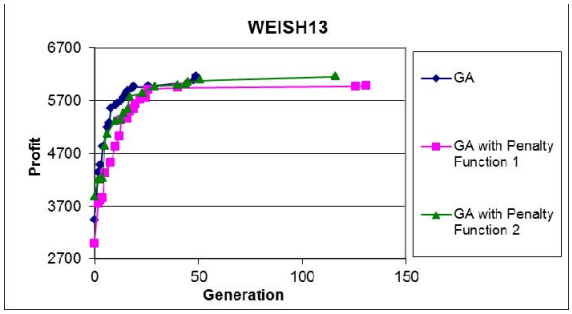

Fig. 3: Different GAs comparison regarding to profit based on problem WEISH13.

For small problems, in most cases, classical GA produces better performance since it has the fastest increase at beginning and finds the optimal solution very early. GA with penalty function 1 generally has the worst performance. GA with penalty function 2 also performed well in some cases. For example, in Fig. 3, the optimal solution (profit: 6159) is found by both GA and GA with penalty function 2.

5. Case study

In this section, we use the classical GA to solve a simple fictitious capital budgeting problem. In order to simulate the real world capital budgeting problems, we add two more constraints to those original constraints in MKP. One is called dependency constraint (e.g., if project 1 is chosen then project 5 should be chosen as well, and vice versa), the other is called exclusive constraint (e.g., if project 1 is selected then project 5 should not be selected, and vice versa).

We simply change the problem SENTO1 (in OR-Library, see the WWW address http://people.brunel.ac.uk/~mastjjb/jeb/orlib/files/mknap2.txt) by adding the above constraints to a real capital budgeting problems. The experiment result is shown as follows:

Firstly, before the extra constraints are added to this problem, we found the well-known solution values (7772) and the optimal solution is displayed as follows:

01001 00110 10110 10110 10001 01001 00000 01000 00000 11000 00100 10010

Next, the dependency constraint (project 1 and 5 are dependent in solutions) was added. The best solution values (7703) were found after 10 runs. The best solution is shown below, where both project 1 and 5 are selected:

11001 00100 00110 10110 10001 01001 00000 11000 00000 11000 00100 10010

Finally, the exclusive constraint (project 1 and 5 are exclusive in solutions) was added. Below is the optimal solution (7772) we obtained.

01001 00110 10110 10110 10001 01001 00000 01000 00000 11000 00100 10010

It is the same as the original optimal solution, because this solution is also feasible in exclusive case.

Note that, this is a simple example showing how to use GAs to solve real world project selection and capital budgeting problems based on MKP. It is sure that there are more other complicated constraints in real world problems.

6. Summary and future work

All three proposed GAs perform well for small problems. Although the best performance is obtained from the classical GA, it is hard to say which is better compared with others. Hence, further comparison is needed in large problems.

Note that, although classical GA performs well for small problems, some of the optimal solutions are not found. This means the performance of classical GA need to improve as well. Future work will be based on testing the performance (both solution quality and CPU consumed time) of GAs with local improvement.

7. References

[1] Freville A., The multidimensional 0–1 knapsack problem: An overview, *European Journal of Operational Research 155 (2004)*, pp 1–21.

[2] Chu P. and Beasley J., A genetic algorithm for the multidimensional knapsack problem, *Journal of Heuristics 4 (1998)*, pp 63–86.

[3] Raidl G.R., An improved genetic algorithm for the multiconstraint knapsack problem, in: *Proceedings of the 5th IEEE International Conference on Evolutionary Computation*, 1998, pp 207–211.

[4] Burke E. and Kendall G., Search Methodologies: Introductory tutorials in optimization and decision support technique, chapter 4. 2005, New York: Springer.

[5] Negnevitsky M., Artificial intelligence: A guide to intelligent systems. 2005, Harlow, England; New York: Addison-Wesley.

[6] Holland J.H., Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. 1975, Ann Arbor: University of Michigan Press.

[7] Michalewicz Z., A survey of constraint handling techniques in evolutionary computation Method. *In 4th Annual Conference on Evolutionary Programming*. 1995, pp 135-155.

[8] Aickelin U., Genetic Algorithms for multiple-choice Optimisation Probems, in European Business Management School. 1999, University of Wales.

[9] Djannaty F. and Doostdar S., A Hybrid Genetic Algorithm for the Multidimensional Knapsack Problem, *Int. J. Contemp. Math. Sciences*, Vol. 3, 2008, no. 9, pp 443 – 456.