Heuristic Approaches for Solving the Multidimensional Knapsack Problem (MKP)

R. PARRA-HERNANDEZ N. DIMOPOULOS
Department of Electrical and Computer Eng.
University of Victoria
Victoria, B.C.
CANADA

Abstract: - There are exact and heuristic algorithms for solving the MKP. Solution quality and time complexity are two main differences among exact and heuristic algorithms. Solution quality is a measure of how close we are from the optimal point and time complexity is a measure of the required time to reach such point. The purpose of this paper is to report the solution quality obtained from the evaluation of three heuristic approaches presented elsewhere in the literature and to report the solution quality obtained from the evaluation of modified versions of those approaches. A genetic algorithm and an ant system are two of the heuristic evaluated.

Key-Words: - Knapsack problem, Ant System, Genetic Algorithm.

1 Introduction

As it is known, the Multidimensional Knapsack Problem (MKP) can be expressed as follows

$$\max \sum_{j=1}^{n} p_j x_j$$

$$subject \quad to \quad \sum_{j=1}^{n} r_{ij} x_j \le b_i \quad i = 1,...,m \qquad (1)$$

$$x_j \in \{0,1\} \quad j = 1,...,n$$

where

x_j represents the j-variable,
r_{ij} represents the required i-resource by the x_j variable,
b_i represents the total amount of i-resource,
p_j represents the value of the x_j variable.

Exact and heuristic approaches have been proposed for solving the MKP, some of them are [1], [6], [8] and [9]. An excellent review of the MKP and its associated exact and heuristic algorithms is given in [3]. As it is pointed out in [3], for the same m, as n increases the problem become harder and take more time to solve. Likewise, if m increases while fixing n, the difficulty increases as well. In the problems evaluated in this paper, n is the variable that increases.

For real time applications, the closer to the optimal point and the faster that point is reached the better. It is always possible that an optimal point may not be found within a reasonable amount of computational effort. Because heuristic algorithms are faster than exact algorithms, heuristic approaches are needed for solving problems that are too large to be solved, in a finite amount of time, by optimal solution procedures.

[3] and [8] (GA and MKHEUR) are known heuristic approaches for solving the MKP. The heuristic approach called Ant Colony System (ACS) appears to be interesting as well; the idea behind it was proposed by Dorigo [4][5]. To the best of our knowledge there is just one reported work on ant systems applied to the MKP [7].

The heuristic approaches MKHEUR, GA and ACS were tested and their performance compared. Unfortunately, the code was not made available to us; the algorithms were coded based on the description of the methods outlined in the corresponding papers.

Modifications to the mentioned heuristics were proposed and coded as well. Comparisons and evaluation performance, based on solution quality and time complexity, are given for the modified approaches. The numerical results were obtained using MATLAB®.

The paper is organized as follows. In section 2, descriptions of MKHEUR, GA and ACS are given; the performance produces by each approach is reported in the same section. In section 3 modifications to the original approaches are suggested and evaluated. Finally, conclusion from this work is given in section 4.

2 Heuristic approaches: MKHEUR, GA and ACS

2.1 Multiknapsack Heuristic: MKHEUR

The procedure discussed in [8] uses a surrogate constraint to determine the order in which variables are fixed equal to one. The surrogate problem can be defined as follows

$$\max \sum_{j=1}^{n} p_j x_j$$

$$subject \quad to \quad \sum_{j=1}^{n}\sum_{i=1}^{m} w_i r_{ij} x_j \le \sum_{i=1}^{m} w_i b_i \qquad (2)$$

$$x_j \in \{0,1\} \quad j = 1,...,n$$

where

$w=\{w_1,...,w_m\}$ set of surrogate multipliers.

The surrogate problem (2) is usually solved to obtain an upper bound on the original MKP. Using $u_j = p_j / \sum_{i=1}^{m} w_i r_{ij}$ a pseudo-utility ratio is obtained and used to determine the order of fixing equal to one variables.

The MKHEUR procedure is as follows:

1. Determine a set of surrogate multipliers.
2. Calculate the pseudo-utility ratios. Sort and renumber variables according to decreasing order of these ratios.
3. Fix variables equal to one according to the order determined in Step 2. If fixing a variable equal to one causes violation of one of the constraints, fix that variable to zero and continue. Denote the feasible solution determined in this step as X_0.
4. For each variable fixed equal to one in X_0, fix the variable equal to zero and repeat Step 3 to defined a new feasible solution. Denote these feasible solutions as X_z ($z=\{1, ..., q\}$; q equals the number of variables equal to one in X_0)

As Pirkul pointed out, the first solution obtained in Step 3 (X_0) is generally not optimal. This observation and the observation that optimal solutions differ from X_0 by only a few variables, led to Step 4. This last step attempts to capture optimal solutions by forcing variables, which have values one in X_0 to zero. The $X_z=\{x_1, ..., x_n\}$ $\forall z \in \{0, ..., q\}$ that max $\sum_{j=1}^{n} p_j x_j$ is *the solution* under MKHEUR.

2.2 Genetic Algorithm: GA

In this approach a heuristic operator that utilizes problem-specific knowledge is incorporated into the standard genetic algorithm. The basic steps of a GA are

1. Generate an initial population.
2. Obtain value of individuals in the population.
3. Repeat: select parents from population
   recombine parents to produce a child
   obtain value of the child
   replace population by the child
   Until: a satisfactory solution has been found.

In [3] an n-bit string representation, where n is the number of variables in the MKP, is used. A value of *0* or *1* at the j-bit implies that $x_j=0$ or *1* in the solution, respectively. A bit string represents a member or individual in the population. If a bit string represents an infeasible solution, a repair operator is used to convert an infeasible solution into a feasible one. The repair operator consists of two phases.

The first phase is called DROP. It examines each variable in increasing order of the pseudo-utility u_j and changes the variable from one to zero if feasibility is violated. The pseudo-utility is given by

$$u_j = p_j / \sum_{i=1}^{m} \lambda_i r_{ij} \qquad (3)$$

where

λ_i Lagrange multipliers of the LPR-MKP solution

LPR-MKP stands for Linear Programming Relaxation of the MKP. The LPR-MKP is expressed by (1) but using the relaxation

$$0 \le x_j \le 1 \qquad j = 1,...n \qquad (4)$$

The second phase is called ADD. It reverses the process by examining each variable in decreasing order of u_j and changes the variable from zero to one as long as feasibility is not violated. The objective of the DROP phase is to obtain a feasible solution from an infeasible one. The ADD phase improves, if possible, the solution quality of a feasible solution.

The first step in the GA approach is the creation of an initial feasible population. Then, a tournament selection is performed in order to select the parents who will have a child in the GA. After the child is created, it suffers a mutation. In the mutation, some randomly selected bits are changed. If the child after the mutation is infeasible, the repair operator is used.

If the child is a non-duplicated child, i.e., the child is not a member of the population, then it replaces the population member that $\min \sum_{j=1}^{n} p_j x_j$. On the other hand, if the child is the same as a member of the population it is discarded and the tournament selection is performed again. This process tends to improve the solution quality of the population as time goes by.

2.3 Ant Colony System: ACS

Dorigo tried to reproduce the ant behaviour in the ACS algorithm. Real ants are able to find a path from a food source to a destination nest by exploiting pheromone information. While walking, ants deposit pheromone on the ground, and follow, in probability, pheromone previously deposited by other ants. As it is explained in [7], in the MKP there are not links or paths to be followed, instead there are variables that seems to be independent.

The ACS works as follows: each ant generates a complete solution (local solution) by accepting variables according to a *probabilistic state transition rule*. The rule is given by

$$P_k(x_j) = \tau(x_j)\eta(x_j)^{\beta} / \sum r(x)\eta(x)^{\beta} \quad \forall x \in J_k \quad (5)$$

where

x_j	feasible j-variable
$P_k(x_j)$	probability of variable x_j to be taken by ant k
$\tau(x_j)$	pheromone value in variable x_j
$\eta(x_j)$	pseudo-utility value of x_j variable
β	relative importance of pheromone versus pseudo-utility
J_k	set of all feasible variables that can be taken by ant k

There is not a single rule to assign a value for τ and η. In our implementation, the initial value of $\tau(x_j)$ equals the value of x_j given by the LPR-MKP solution; $\eta(x_j)$ equals the pseudo-utility value given by (3).

Before accepting a variable, the ant evaluates the variables that have not been taken by the ant yet, and generates a set of feasible variables. A feasible variable is one that satisfies the ant resources constraint. The initial amount of ant resources is given by $\boldsymbol{b}=[b_1, ..., b_m]$. When the ant accepts the x_j variable the amount of its resources decreases by $\boldsymbol{r}=[r_{1j}, ..., r_{mj}]$ (see (1)).

Based on the pheromone and pseudo-utility value, a probability value using (5) is generated for each of the feasible variables; then a random selection, based on the probability obtained, is performed. It is evident, from (5), that ants prefer to accept variables that have a high pseudo-utility value $\eta(x_j)$ and a high amount of pheromone $\tau(x_j)$. When an ant accepts a variable, it decreases the pheromone amount on that variable by applying a *local updating rule*. The idea is to make the variable less desirable and therefore it will be chosen with a lower probability by the other ants.

A local solution is reached when an ant either allocates its initial resources or, although there are available resources, it is not feasible to accept more variables, i.e., resources are not enough. After each ant reaches a local solution, an ant-value is generated using

$$ant.value(r) = \sum_{j=1}^{n} p_j x_j \quad r = 1,...,R \quad (6)$$

where

R represents the number of ants

The ant with the highest *ant.value(r)* is chosen, and the pheromone amount of the variables taken by this ant is increased. This is done by applying a *global updating rule*. The goal of the global updating rule is to make the variables, which belong to the best solution, more attractive. The variables taken by this ant are recorded in a global memory. This complete one cycle of the searching process.

A new cycle is performed and, again, the ant with the highest value is chosen. If this value is higher than the one obtained in the last cycle, then the global memory is updated with the new variables. This process is done cycle after cycle and stops until either the number of maximum cycles is reached or a certain criterion performance is obtained.

2.4 Numerical results

As suggested in [8], the dual variables from the LPR-MKP solution were used as the surrogate multipliers. The performance of MKHEUR, GA, and ACS is given in Tables 1 and 2. The approaches were tested in 4 sets of problems. Each set has 10 problems. The problems were taken from the *OR-Library* (address www.ms.ic.ac.uk/info.html).

The *OR-Library* test problems evaluated are contained in the files: mknapcb7 (first 10 problems: Set 1), mknapcb8 (first 10 problems: Set 2) and mknapcb9 (first 20 problems: Sets 3 and 4). The number of variables (n) goes from 100 to 500 and the number of constraints is 30 in all cases. For an in

detail description about how the problems were generated and how they are organized in the mentioned files, see [3] and *OR-Library* web site. The heuristics were tested in a Pentium II 400 MHz Personal Computer running Linux OS.

The performance of the approaches is measured by the percentage gap between the best solution found by the heuristic and the optimal LPR-MKP solution, i.e., gap=100*(optimal LPR value - best solution value) / (optimal LPR value).

Table 1. Heuristic Performances (gap)

	MKHEUR	GA	ACS
Set (n)	% gap mean value		
1 (100)	5.20	3.71	4.40
2 (250)	2.14	1.56	2.09
3 (500)	1.05	0.86	1.14
4 (500)	0.44	0.38	0.50

Table 2. Heuristic Performances (time)

	MKHEUR	GA	ACS
Set (n)	time (secs)		
1 (100)	0.50	>3000	<25
2 (250)	1.47	>3000	<100
3 (500)	4.50	>3000	<300
4 (500)	5.68	>3000	<500

Because the GAP and ACS are probabilistic algorithms, dispersion values are given in Table 3. GA and ACS were run 100 times for each set. For the GA, each time 10^6 non-duplicated children were allowed. 100 feasible members were created for the initial population. For the ant system, each time 10 ants were used and 50 iterations performed. β equals log(10e2)/log(max(p.utility.value)/min(p.utility.valu e)). Parameters needed in the local and global updated rules were set equal to 0.1

Table 3. Heuristic Performances (dispersion values)

	MKHEUR	GA	ACS
Set (n)	Variance		
1 (100)	0	0.2409	0.4200
2 (250)	0	0.0477	0.0625
3 (500)	0	0.0125	0.0160
4 (500)	0	0.0043	0.0355

From Table 1 is clear that GA is the best heuristic approach based on solution quality. Nevertheless, GA is the heuristic that takes the longest (see Table 2). For small size problems (<250 variables) ACS appears to be a good choice. If time matters, as it usually does in real time applications, then MKHEUR appears to be the option.

3 Modified Heuristic Approaches

The results obtained in the last section shows that MKHEUR is a good heuristic based on solution quality and time complexity. Nevertheless, GA produces better solution quality if greater computation time is allowed.

GA performance can be speeded up if more knowledge about the behaviour of the MKP solution is incorporated into the algorithm. GA starts with the generation of an initial population. Throughout the algorithm the population performance is improved. After a certain performance is reached, almost all the individuals of the population have the same characteristics (variables among population members have almost the same values). The differences are among a *subset* of variables. The pseudo-utility values of this subset fall around a band in between the highest and lowest value of the pseudo-utility. This means that almost always all the variables with the highest pseudo-utility are taken and almost all the variables with the lowest pseudo-utility are rejected. The MKHEUR solutions (Step 4 section 2.1) could be used to generate the initial population in the GA algorithm. The population will take the highly desirable characteristics from the MKHEUR solutions. This will save the time originally needed, in the GA algorithm, to start improving the population performance.

The solution behaviour of the knapsack problem was originally notice by Balas and Zemel, [2], while working on single-constraint knapsack problems. It was noticed that the solution obtained by packing the knapsack in the decreasing order of utility ratios differed from the optimal solution by only a few variables. Differing variables were always around the point where the heuristic solution switched from ones to zeros (after renumbering variables in decreasing value of their ratios). A core problem was defined as *"the problems in those variables, whose cost/weight ratio falls between the maximum and minimum c/a ratio for which x has a different value in an optimal solution to KP from that in an optimal solution to LKP*" (KP- single constraint problem; LKP-linear programming relaxation of KP). A similar statement for the MKP core problem is given in [8].

This knowledge about the characteristics of the solution could be used in the next heuristic as follows

1. Reduce the original problem. As it was mentioned before, the variables with the lowest pseudo-utility values are almost always rejected; these variables represent approximately 15% of the original problem size. After the reduction is done, MKHEUR is applied in the new problem. Then a re-mapping is performed in order to get the solution, so far, in terms of the original variables.
2. Apply a heuristic rule to get an estimation of the core problem size and
3. Apply GA only on the core problem

3.1 MKHEURv2

For the part concerned to the reduction of the original problem, the following preprocessing steps need to be done:

1. Determinate a set of Lagrange multipliers
2. Calculate the pseudo-utility ratios. Sort and renumber variables according to decreasing order of those ratios.
3. Fix variables equal to one according to the order determined in the last step. If fixing a variable equal to one causes violation of one of the constraints, fix that variable equal to zero and continue. Denote the feasible solution determined as X. Calculate the pseudo-utility mean value of X ($\tilde{x}$).
4. Create a set of variables called Pseudo Rejected Variables (PRV) (last 15% of the original MKP variables with the lowest pseudo-utility ratio).
5. Merge PRVs based on their pseudo-utility ratio. Create *new variables* made of packed PRVs. The number of PRVs needed to create a *new variable* depends on the pseudo-utility value of each PRV. The pseudo-utility value of each PRV is added up until the total pseudo-utility of the new variable being packed is greater than $\tilde{x}$. The total pseudo-utility value obtained is the pseudo-utility value of the *new variable*. In the same way, the amount of resources needed by each PRV being packed is added up. The total amount of resources is the one required by the *new variable*.

It has to be noticed that a new problem has been created. The size of this one is smaller than that of the original problem. The new problem dimension is *n-rv+nv*, i.e., original dimension minus number of rejected variables plus new variables. The next step is to apply MKHEUR to the new problem. Finally, a mapping is performed in order to obtain the solution found in terms of the original variables. As it is clearly evident, the complexity of this modified version (MKHEURv2) is greater than that of MKHEUR, but as the number of variables increases the complexity pays off. The time used by MKHEURv2 to solve large-size problems is lesser than the one needed using MKHEUR. Moreover, MKHEURv2 seems to produce better solution quality than MKHEUR. A comparison performance is given in Table 4.

Table 4. MKHEUR and MKHEURv2 performance

	MKHEUR		MKHEURv2	
Set (n)	% gap	time (sec)	% gap	time (sec)
1 (100)	5.20	0.50	5.02	0.82
2 (250)	2.14	1.47	2.02	1.99
3 (500)	1.05	4.50	0.97	5.2
4 (500)	0.446	5.68	0.442	5.9

3.2 MKHEURv2+GA

MKHEURv2 was used to create the initial population needed for the GA algorithm. Because the population has highly desirable characteristics, the number of non-duplicated children for GA was reduced from 10^6 to 10^3. However, before merging MKHEURv2 and GA the dimension of the core problem has to be defined. The core problem dimension was set equal to the dimension of the original problem minus 80% of the first consecutive variables equal one in solution X (section 3.1 Step 3) and minus the variables used as PRVs. The results obtained are given in Table 5.

Table 5. GA, MKHEURv2 and MKHEURv2+GA performance

	GAP (10^6)	MKHEURv2	MKHEURv2 + GA (10^3)
Set (n)		% gap	
1 (100)	3.71	5.02	3.610
2 (250)	1.56	2.02.	1.601
3 (500)	0.86	0.970	0.963
4 (500)	0.38	0.442	0.420
5 (1000)	0.28	0.371	0.371

For this section a new set of problems was created. The set has 10 problems, each one has 30 constraints and it is made of Set 3 plus Set 4. Each set was evaluated 100 times. It is interesting to notice that, although the number of non-duplicated

children in the MKHEURv2+GA algorithm has been reduced dramatically, the performance of MKHEURv2+GA is similar as GA allowing 10^6 non-duplicated children (problems with 250 variables or less).

3.3 ACS + MKHEUR

It is also possible to merge ACS and MKHEUR in order to try to obtain an approach that produces a better performance than any one of these approaches alone. In section 2.3 is reported that the LPR-MKP solution is needed for the pseudo-utility ratios estimation; this is the main step in the MKHEUR algorithm. The MKHEUR missing step can be implemented at the end of the ACS algorithm. When a cycle is completed in the ACS, the best local solution is recorded in memory. In the ACS+MKHEUR approach, the best local solution is considered to be X_0 from Step 3 of MKHEUR (see 2.1), and then Step 4 is applied over this X_0 given by the ants. This was done and the solution quality obtained is slightly better then the one reported for ACS alone (about 1% better). Because the improvement was minuscule the ACS + MKHEUR approach was not further explored

4 Conclusion

The MKP can be used to express interesting problems such as resource allocation. The decision about what algorithm use to solve the MKP is problem depended. For real time applications, time is a critical factor to be considered and heuristic methods are used more often over exact methods. MKHEUR produces a good solution quality without requiring an excessive computational time. The LPR-MKP solution part takes more than 90% of the time needed by MKHEUR. Therefore, any improvement in that direction is welcome. Using knowledge about the solution behaviour of the MKP, a mapping of the problem is suggested and done, and the dimension of the MKP is reduced. Then, MKHEUR is applied and then a re-mapping of the problem is performed in order to express the solution in terms of the original variables.

Because MKHEURv2 works over a smaller dimension of the original problem, it is faster and the solution quality seems to be better than that of MKHEUR. It has to be pointed out that for small size problems, MKHEURv2 is slower because the time needed to perform the mapping is greater then the time needed for solving the LPR-MKP. As the problem size increases, MKHEURv2 becomes a better candidate for solving the MKP. If more computational time is allowed, then the GA algorithm can be applied on a defined core section and further improvement could be obtained.

Heuristic approaches are divided in two sets, the probabilistic ones and the non-probabilistic. GA and ACS are probabilistic approaches, therefore dispersion values must be considered if a heuristic approach is to be chosen. It is difficult to obtain reliable dispersion values for probabilistic approaches due to its nature. GA and ACS were run 100 times on the same problems. It appears that ACS produces a smaller dispersion value than GA when the latter is allowed to generate up to 10^3 non-duplicate children.

References:

[1] Balas, E. and Martin, C., Pivot and Complement– A Heuristic for 0-1 programming, *Management Science*, Vol. 26, 1980, pp. 86-96.

[2] Balas, E., and Zemel, An algorithm for large Zero-One knapsack problems, *Operations Research*, Vol. 28, 1980, pp. 1130-1154.

[3] Chu, P., and Beasley, J., A Genetic Algorithm for the Multidimensional Knapsack Problem, *Journal of Heuristics*, Vol. 4, 1998, pp. 63-86.

[4] Dorigo, M., *Optimization, learning and natural algorithms*. PhD Thesis. Politecnico di Milano. Italy. 1992.

[5] Dorigo, M., and Maniezzo, V., and Colorni, A., The ant system: optimization by a colony of cooperating agents, *IEEE Transactions on Systems, Man, and Cybernetics - Part B*, Vol. 26, 1996, pp. 29-41.

[6] Gavish, B. and Pirkul, H., Efficient Algorithms for Solving the Multiconstraint Zero-One Knapsack Problem to Optimality. *Mathematical Programming*, Vol. 31, 1985, pp. 78-105.

[7] Leguizamon, G., and Michalewicz, Z., A new version of ant system for subset problems. *Congress on Evolutionary Computation*, Vol. 2, 1999, pp. 1459-1464.

[8] Pirkul, H., A Heuristic solution procedure for the Multiconstraint Zero-One Knapsack Problem, *Naval Research Logistics*, Vol. 34, 1987, pp. 161-172.

[9] Toyota, Y., A simplified algorithm for obtaining approximate solution to Zero-One programming problems, *Management Science*, Vol. 21, 1975, pp. 1417-1427.