A Constructive Genetic Algorithm For The Generalized Assignment Problem

Luiz A. N. Lorena

LAC/INPE- Instituto Nacional de Pesquisas Espaciais

Caixa Postal 515

12201-970 - São José dos Campos – SP, Brazil

lorena@lac.inpe.br

Marcelo G. Narciso

Embrapa Informática Agropecuária

Av. Dr. André tosello, s/n, Unicamp

13083-970 – Campinas - SP, Brazil

narciso@cnptia.embrapa.br

J.E. Beasley

The Management School, Imperial College

London SW7 2AZ, England

j.beasley@ic.ac.uk

Abstract

We present in this paper an application of the *Constructive Genetic Algorithm (CGA)* to the *Generalized Assignment Problem (GAP)*. The *CGA* presents some new features compared to a traditional *genetic algorithm (GA)*, such as a population formed only by schemata, recombination among schemata, dynamic population, mutation in complete structures, and the possibility of using heuristics in schemata and/or structure representation. The *GAP* can be described as a problem of assigning *n* items to *m* knapsacks, *n*>*m*, such that each item is assigned to exactly one knapsack, subject to capacity constraints on the knapsacks. In our application of *CGA to GAP*, we regard the *GAP* as a clustering problem. A binary representation is used for schemata and structures, and an assignment heuristic allocates items to knapsacks. Schemata do not consider all the problem data. The schemata are recombined, and they can produce new schemata or structures. New schemata are evaluated and can be added to the population if they pass an evolution test. Structures can result from recombination of schemata or complementing of good schemata. They suffer mutation and the best structure generated is kept in the process. Computational tests have been performed using instances of large scale available in the literature.

1. Introduction

Genetic Algorithms (GA) have become popular in recent years as efficient heuristics for difficult combinatorial optimisation problems. The underlying foundation for such algorithms are the controlled evolution of a structured population. Today there are many variations on the general GA theme and all such variations can be classified generically as population heuristics [3], that is as heuristics that operate with a population of solutions. Such heuristics are in marked contrast to other approaches, such as tabu search and simulated annealing, that operate on just a single solution.

The traditional *GA* can usually be represented [17] by the pseudo-code:

Genetic Algorithm - GA

```
t := 0 ;
Initialize $P_t$ ;                 { initial population }
Evaluate $P_t$ ;
While (not stop condition) do
    t := t + 1 ;
    Select $P_t$ from $P_{t-1}$ ;   { selection operator }
    Recombine $P_t$ ;              { crossover and mutation operators }
    Evaluate $P_t$ ;               { evaluate fitness }
end_while
```

The *GA* works on a set of variables called *structures*. For application to optimization problems [4,14], the first step is the definition of a codification plan that allows a one to one mapping between solutions and structures. The following string can represent a structure $S_k = (s_{k1}, s_{k2}, ..., s_{kn})$, where n is the number of variables in the problem. A *fitness function* assigns a numeric value to each member of the current *population* (a collection of structures). The *genetic operators* used are *selection*, like tournament or biased roulette wheel, working together with a number of *crossover* and *mutation* operators. The best structure is kept after a predefined number of generations [10,17,18].

Holland [10] put forward the *building block* hypothesis (*schema* formation and conservation) as a theoretical basis for the *GA* mechanism. In his view avoiding disruption of good schema is the basis for the good behaviour of a *GA*. One major problem with building blocks however is that schemata are evaluated indirectly, via evaluation of their instances (structures), rather than directly, as an instance may typically represent several schemata.

The *Constructive Genetic Algorithm (CGA)* was proposed recently as an alternative to the traditional *GA* approach [9,20]. One of the objectives of a *CGA* is the direct evaluation of schemata. The population, formed only by schemata, is built, generation after generation, searching for a “highly informed” population. The basic form of a *CGA* [13] is:

Constructive Genetic Algorithm - CGA

```
$\alpha$ := 0 ;
$\varepsilon$ := 0.01;                                   { time interval }
Initialize $P_\alpha$ ;                                  { initial population }
Evaluate $P_\alpha$ ;                                    { proportional fitness }
For all $S_k$ Î $P_a$ compute $d(S_{.k})$               { rank computation }
end_for
While (not stop condition) do
    For all $S_k$ Î $P_a$ satisfying $\alpha < d(S_k)$ do    { evolution test }
        $\alpha := \alpha + \varepsilon$ ;
        Select $P_\alpha$ from $P_{\alpha-\varepsilon}$ ;          { reproduction operator }
        Recombine $P_\alpha$ ;                           { recombination operators }
        Evaluate $P_\alpha$ ;                            { proportional fitness }
    end_for
    For all new $S_k$ Î $P_a$ compute $d(S_k)$          { rank computation }
    end_for
end_while
```

Some steps in *CGA* are notably different from a classical *GA*. The *CGA* works with a *dynamic population*, composed exclusively of schemata, which increases after the use of *recombination operators* and can decrease as generations pass, guided by the *evolution parameter* α (increased by ε at each generation). Schemata recombination diversifies the population thereby generating new schemata or structures. At the time of its creation each schema receives a *proportional fitness* evaluation and a rank $d(S_k)$ used in the evolution test. A structure can be obtained after schemata recombination, or by complementing a good schema. They represent feasible solutions, suffer mutation and are compared to the best solution found so far, which is always retained. Another main difference between a classical *GA* and a *CGA* is the new *fg-fitness* process.

We examine in this paper a *CGA* application to the problem of the minimum cost assignment of n tasks to m agents ($n > m$), such that each task is assigned to only one agent subject to capacity constraints on the agents. This problem is an important combinatorial optimization problem, the *Generalized Assignment Problem (GAP)*.

Many real life applications can be modeled as a *GAP*, e.g. resource scheduling, the allocation of memory space in a computer, the design of a communication network with capacity constraints for each network node, assigning software development tasks to programmers, assigning jobs to computers in a network, vehicle routing problems, and others [1,4,6,7].

GAP is NP-hard [19]. A number of algorithms in the literature are exact tree search methods [16,21], and there are also a number of heuristics for the problem [4,8,11,12,15,16,19].

This paper is organized as follow. A review of the *Constructive Genetic Algorithm (CGA)* is presented in Section 2. Section 3 presents the *CGA* application to the *GAP*. Some other aspects of the *CGA* are also explained, the schemata and structure representation, recombination and mutation. Section 4 presents computational tests considering large scale instances from the literature, providing insights into *CGA* performance.

<u>*2. CGA review*</u>

We present in this section a review of a typical *CGA*. The main components of a *CGA* will be described in the following sequence:

- *fg-fitness*,
- the evolution parameter,
- selection, and
- stopping conditions.

<u>*2.1. fg-fitness*</u>

In a *CGA* population evaluation is performed by mapping the schemata space onto $\Re_+$. Each generation has an associated evolution time a. Let P_a be the population of schemata at evolution time a, and consider two functions defined as $f : P_a \rightarrow \Re_+$ and $g : P_a \rightarrow \Re_+$, such that $f(S_k)$ £ $g(S_k)$ for all $S_k \in P_a$. This double fitness of structure S_k is called *fg-fitness*. We also need to have defined a common upper bound $g_{max} > max[g(S_k)$ for each $S_k \in P_a]$.

For each structure $S_k \in P_a$ there is an associated *gap* (*proportional deviation* from $g(S_k)$) given by $d_k = \frac{g(S_k) - f(S_k)}{g(S_k)}$, $k=1,\ldots,m$. The absolute deviation from $g(S_k)$ is obtained by $d_k g(S_k)$. If d is an expected overall proportional deviation from g_{max}, dg_{max} is the expected absolute deviation from g_{max}.

Functions f and g will be particularized to the *GAP* in the next section. Considering that

S_k and S_j are two given schemata or schema and structure, if $d_k > d_j$ then S_j is better than S_k.

2.2. The evolution parameter

Given the definition of *fg-fitness*, schemata in the initial population can be evaluated. We now examine population evolution. *CGA* uses an automatic mechanism to maintain the population – the population growing in the initial generations and decreasing when good structures are found. This process is governed by the evolution parameter, to be presented in this section. To assign an appropriate value to this parameter, we need to use the common upper bound g_{max} and the *fg-fitness* deviations presented in Section 2.1. The process is as follows.

We would like to know how much time a schema S_k will survive in the population, producing offspring. The *fg-fitness* deviations give the answer. The schema will be evaluated comparing the absolute deviations $d_k g(S_k)$ and dg_{max}. We use an analogy with the well-known *A** [13] approach. Our goal is the expected absolute deviation from g_{max}, and the current absolute deviation from $g(S_k)$ is $d_k g(S_k)$. To reach the goal, an estimate of what we will additionally have in the future as absolute deviation (on offspring), parameterized by a non-negative real value a, can be given by $a\ d[g_{max} – g(S_k)]$. Then if $d_k g(S_k) + a\ d[g_{max} – g(S_k)]$ ³ dg_{max}, the structure S_k has no future and must be rejected.

The parameter a is estimated using the expression

$$a \geq \frac{dg_{\max} - d_k g(S_k)}{d[g_{\max} - g(S_k)]} = d(S_k)\,.$$

When it is created each schema receives the corresponding *rank* value $d(S_k)$ that will be compared to the current *evolution parameter* a. Hence from the moment of its creation we know how long it will survive. The higher the value of $d(S_k)$, the more survival and recombination time the schema will have.

2.3. Selection

Selection of individuals can be made in several ways. *CGA* has been tested with a number of optimization problems and in all cases an appropriate approach is that the population is kept ordered using a key value that considers the *fg-fitness* and its proximity to a feasible solution representation. Then, several times in a generation, two schemata are randomly selected, one from among the best part of the population and the other from the whole population, and these are recombined to form (one or more) new schemata or structures (see Lorena and Furtado [13]).

The recombination is made depending on the problem and the way the structure represents a solution. The main goal of recombination is population diversification. Structures representing feasible solutions can be generated not only by recombination, but also by complementation of a selected schema. The best results found with the *CGA* uses mutation over structures that represent feasible solutions for the problem (see Lorena and Furtado [13]).

2.4. Stopping conditions

As the evolution parameter increases, the population size initially increases and α then start to decrease until eventually the population becomes empty. So, two stopping conditions are considered: the process stops when the population is empty, or when a pre-defined generation limit is reached.

3. The CGA application to GAP

The *GAP* is best described using knapsack problems [16]. Given *n* items and *m* knapsacks, with p_{ij} as the cost associated with assigning item *j* to knapsack *i*, w_{ij} as the weight of assigning item *j* to knapsack *i*, and c_i the capacity of knapsack *i*, assign each item *j* to exactly one knapsack *i*, not exceeding knapsack capacities. Then the *GAP* can be formulated as

$$v(GAP) = \quad Min \sum_{i=1}^{m}\sum_{j=1}^{n} p_{ij}x_{ij}$$

$$(GAP) \qquad subject\ to \qquad \sum_{j=1}^{n} w_{ij}x_{ij} \le c_i\ ,\ i \in M = \{1,\dots,m\},$$

$$\sum_{i=1}^{m} x_{ij} = 1\ ,\ j \in N = \{1,\dots,n\},$$

$$x_{ij} \in \{0,1\}\ ,\ i \in M\ ,\ j \in N.$$

The application of the *CGA* to the *GAP* is made through an analogy with clustering problems. Each knapsack is a capacitated cluster to which items must be allocated.

3.1. Schema and structure representation

For schema and structure representation, we used a sequence of *n* symbols, where *n* is the number of items. Seed items are initially assigned to the *m* knapsacks, exactly one per knapsack. If the structure is a schema, some items are not considered, i.e. they are considered temporarily out of the problem. The structures have in each position one of the following three possible symbols:

1 - to indicate the seed item is assigned to a knapsack,

0 - to indicate a non-seed item assigned to a knapsack, and

# - to indicate items temporarily out of the problem.

A structure with #'s represents a schema. For example considering a problem with 10 items and 3 knapsacks, a structure could be represented by S_k = (#,1,#,0,1,0,0,#,1,0), where item number 2 was assigned to knapsack 1, item number 5 was assigned to knapsack 2, and item number 9 is assigned to knapsack 3. The items receiving labels 0 will be assigned to one of the knapsacks according to an assignment heuristic, and the items with labels # are out of the problem.

Suppose we have a given schema or structure S_k. The following assignment heuristic is used to complement the S_k representation:

Assignment Heuristic - AH

```
1 – Assign the m items with label 1 to the m knapsacks,

2 – Update the knapsack capacities,

3 – Assigning the other items to the knapsacks (labels 0 and #)
        3.1 – solve the m knapsack problems separately exactly
        3.2 – update the knapsack capacities for the items assigned to exactly one knapsack,
        3.3 – resolve the m knapsack problems separately exactly for the remaining items,
        3.4 – update the knapsack capacities for the items assigned to exactly one knapsack,
        3.5 – for each item j remaining, assign it to knapsack$_{i*}$ corresponding to the smallest $w_{i*j}$ .
        3.5 – If the obtained solution is not feasible to GAP, restart the assignments of the n-m items (the
        m seed items were already assigned in step 1), assigning item j to knapsack$_{i*}$ corresponding to the
        smallest $w_{i*j}$. If capacities are violated, assign if possible, item j to the knapsack corresponding to
        the next smallest $w_{ij}$ for which capacities are not violated.

4 – If the solution is feasible to GAP improve the solution with the second part of MMTH (see [15]), else
discard the schema and select a new one.

5 – Discard from knapsacks the items with labels # in $S_k$.
```

Knapsack problems are solved exactly using the algorithm of Horowitz and Sahni [16].

3.2. Particularizing the fg-fitness

Consider a structure or schema $S_k \in P_a$. For the *GAP*, after the application of the assignment heuristic *AH*, the clusters $C_i(S_k)_{AH}$ are identified, corresponding to the indices of items on knapsack$_i$, i=1,...,m. The function *g* is then defined by

$$g(S_k) = \sum_{i=1}^{m} \sum_{j \in C_i(S_k)_{AH}} p_{ij} \, .$$

To define the function *f* the following *MAH* heuristic is applied to S_k, producing an additional move of one item between two knapsacks:

Modified Assignment Heuristic – MAH

```
1. Apply AH to $S_k$.
2. Sort in non-increasing order the costs $p_{ij}$ corresponding to the items in knapsacks presenting label 0 in $S_k$.
3. Let $p_{i^*j^*}$ be the cost of the item at the first order position (item $j^*$ was assigned to knapsack $i^*$).
4. Sort in non-decreasing order the costs $p_{ij^*}$, i=1,...,m. Let $p_{i'j^*}$ be the cost of the item at the first order position.
5. Move item $j^*$ to knapsack $i'$.
```

After the *MAH* application, if S_k is an structure, the corresponding *GAP* solution may be infeasible. The new clusters $C_i(S_k)_{MAH}$ are used in the definition of function *f* as

$f(S_k) = \sum_{i=1}^{m} \sum_{j \in C_i(S_k)_{MAH}} p_{ij}$. Clearly we have that $f(S_k) \leq g(S_k)$.

To compute the upper bound g_{max}, at the very beginning of the process, a structure *S* representing a feasible solution (no #'s) is randomly generated and *g(S)* is taken as the g_{max} value.

For all the computational results presented in this paper an initial population was randomly created with 20% of positions in each structure with label 0, exactly *m* (number of knapsacks) with label 1, and the remaining positions having the label #.

3.4. Selection and recombination

The population was kept non-decreasing ordered according the following key $\Delta(S_k) = \frac{1+d_k}{n-n_\#}$, where $n_\#$ is the number of # labels in S_k. Schemata with small $n_\#$ and/or presenting small d_k are better and appear in first order positions.

The method used for selection takes one schema from the *n* first positions in the population (*base*) and the second schema from the whole population (*guide*). Before recombination, the first schema is complemented to generate a structure representing a feasible solution, i.e. all #'s are replaced by 0's. This complete structure suffers mutation and is compared to the best solution found so far (which is kept throughout the process). The recombination merges information from both selected schemata, but preserves the number of labels 1 (number of knapsacks) in the new generated schema.

Recombination

```
if $S_{base}(j) = S_{guide}(j)$ then $S_{new}(j) \neg S_{base}(j)$
if $S_{guide}(j)=\#$ then $S_{new}(j) \neg S_{base}(j)$
if $S_{base}(j) = \#$ or 0 and $S_{guide}(j)=1$ then
    $S_{new}(j) \neg 1$ and $S_{new}(i) \neg 0$ for some $S_{new}(i)=1$
if $S_{base}(j) = 1$ and $S_{guide}(j)=0$ then
    $S_{new}(j) \neg 0$ and $S_{new}(i) \neg 1$ for some $S_{new}(i)=0$
```

At each generation, exactly *n* new schemata are created by recombination. If a new schema does not represent a feasible solution, then it is inserted into the population; otherwise it suffers mutation and is compared to the best solution found so far. The following pseudo-code describes the mutation process:

Mutation

```
For each position j with label 1 do
    For each position l with label 0 do
        Interchange the labels on positions j and l generating an offspring $S_{new}$ ;
                                                        {offspring generation}
        Interchange the labels on positions j and l ; {returning to the original $S_{base}$ }
    End_for
End_for
```

The mutation process was limited to considering just ten new structures to avoid excessive computation time.

At each generation, after new schemata insertion, the population is scanned to remove all structures satisfying the condition $a \geq d(S_k)$. As described earlier in this paper, the evolution parameter a is initially set to zero and slowly increased at each generation.

4. Results

In this section we outline the *CGA* performance on the *GAP*. The *CGA* was coded in *C* and run on a *SUN ULTRA SERVER 2, 200 MHz* machine.

A set of large-scale instances were solved (of dimensions, m x n, (5 x 100), (5 x 200), (10 x 100), (10 x 200), (20 x 100) and (20 x 200), from OR-Library [2]). These comprise 24 instances of different sizes and types. Referring to *Table 1* the problems in classes *A, B* and *C* present increasingly constrained knapsacks. Class *D* comprises more difficult correlated problems.

Table 1 presents the best *CGA* results (best $g(S_k)$) for ten replications compared with the best known solutions reported in [5]. The *CGA* parameters are set to:

d = 0.15,
a starts at 0,
e = 0.1 for 0 £ a £ 1,
e = 0.01 for a > 1.
The stopping conditions: maximum number of generations = 150, or
the population is empty (a is big enough).

For problems in class A the best known solutions are optimal so the algorithm was terminated when those solutions were found.

The *CGA* solutions reported in *Table 1* are very close to the best known solutions, obtained in the *GA* implementation of Chu and Beasley [5] who ran their *GA* until 500000 distinct feasible solutions were found. It can be conjectured that the computational efforts of *CGA* are very small compared to their *GA*. The computer times are not directly comparable, as the *GA* was run on a different machine.

Table 1: Computational results

Problem	*Best known solution*	*CGA solution*	*Number of generations*	*CGA times (seconds)*
A 5x100	1698	1698	51	253
A 5x200	3235	3235	1	502
A 10x100	1360	1360	87	308
A 10x200	2623	2623	72	930
A 20x100	1158	1158	1	350
A 20x200	2339	2339	19	860
B 5x100	1843	1843	150	302
B 5x200	3553	3601	150	432
B 10x100	1407	1410	150	165
B 10x200	2831	2831	150	949
B 20x100	1166	1166	150	474
B 20x200	2340	2347	150	683
C 5x100	1931	1941	150	195
C 5x200	3458	3460	150	405
C 10x100	1403	1423	150	203
C 10x200	2814	2815	150	498
C 20x100	1244	1244	150	479
C 20x200	2397	2397	150	1059
D 5x100	6373	6479	150	259
D 5x200	12796	12823	150	1253
D 10x100	6379	6390	150	497
D 10x200	12601	12634	150	1321
D 20x100	6269	6280	150	974
D 20x200	12452	12471	150	2158

5. Conclusions

In this paper we have presented an application of the constructive genetic algorithm to the generalized assignment problem. Computational results were promising as compared to a previous genetic algorithm approach presented in the literature.

Acknowledgments:
The first author acknowledges Conselho Nacional de Desenvolvimento Científico e Tecnológico -CNPq (proc. 350034/91-5, 520844/96-3, 680082/95-6) and Fundação para o Amparo a Pesquisa no Estado de S. Paulo - FAPESP (proc. 95/9522-0 e 96/04585-6) for partial financial support.

References

[1] Balachandran, V. *An integer generalized transportation model for optimal job assignment in computer networks*. Operations Research, v. 24, n.4, p. 742-749, 1976.

[2] Beasley, J.E. *OR-Library: Distributing test problems by electronic mail.* Journal of Operational Research Society, v. 41, n. 11, p. 1069-1072, 1990.

[3] Beasley, J.E. *Population heuristics.* Technical Report – Imperial College, London, England, 1999.

[4] Catrysse, D. and Van Wassenhove, L.N. *A survey of algorithms for the Generalized Assignment Problem.* European Journal of Operational Research. v. 60, p. 260-272, 1992.

[5] Chu, P.C. and Beasley, J.E. *A genetic algorithm for the generalised assignment problem.* Computers and Operations Research. p. 17-23, 1997.

[6] De Maio, A. and Roveda, C. *An all zero-one algorithm for a certain class of transportation problems.* Operations Research. v. 19, p. 1406-1418, 1971.

[7] Fisher, M.L. and Jaikumar, R. *A generalized assignment heuristic for vehicle routing.* Networks. v. 11, p. 109-124, 1981.

[8] Fisher, M.L., Jaikumar, R. and Wassenhove, L.N.V. *A multiplier adjustment method for the generalized assignment problem.* Management Science. v. 32, p. 1095-1103, 1986.

[9] Furtado, J.C. *Algoritmo Genético Construtivo na Otimização de Problemas Combinatoriais de Agrupamentos.* Ph.D. thesis - INPE, 1998

[10] Holland, J.H. *Adaptation in natural and artificial systems*. MIT Press, p. 11-147, 1975.

[11] Jornsten, K. and Varbrand, P. *Relaxation techniques and valid inequalities applied to the generalized assignment problem.* Asia Pacific Journal of Operational Research. v. 7, p. 172-189, 1990.

[12] Klastorin, T.D. *An effective subgradient algorithm for the Generalized Assignment Problem.* Computers and Operations Research. v. 6, p. 155-164, 1979.

[13] Lorena, L.A.N. and Furtado, J.C. *Constructive genetic algorithm for clustering problems.* Submitted for publication – Evolutionary Computation. Presented at the Optimization 98 congress - Coimbra, Portugal - July 1998. Available from http://www.lac.inpe.br/~lorena/cga/cga_clus.PDF

[14] Lorena, L.A.N. and Lopes, L.S., *Genetic Algorithms Applied to Computationally Difficult Set Covering Problems.* Journal of the Operational Research Society 48, 440-445, 1997.

[15] Lorena, L.A.N. and Narciso, M.G. *Relaxation heuristics for a generalized assignment problem.* European Journal of Operational Research. v. 91, n. 1, p. 600-610, 1996.

[16] Martello, S. and Toth, P. *Knapsack Problems - Algorithms and Computer Implementations.* New York: John Wiley & Sons, USA, 1990.

[17] Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Berlin, 1996.

[18] Mitchell, M., *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, England, 1996.

[19] Narciso, M.G. and Lorena, L.A.N. *Lagrangean/surrogate Relaxation for Generalized Assignment Problems*. European Journal of Operational Research, 114(1), 165-177, 1999.

[20] Ribeiro Filho, G. and Lorena, L.A.N. *A constructive algorithm for cellular manufacturing design.* EURO XVI - 16th European Conference on Operational Research, 12 a 15/07/98, Brussels, Bélgica.

[21] Ross, G. T. and Soland, M.S. *A branch and bound algorithm for the generalized assignment problem.* Mathematical Programming, v. 8, p. 91-103, 1975.