A Constructive Genetic Algorithm For The Generalised Assignment Problem

Research paper

Luiz A. N. LORENA lorena@lac.inpe.br

LAC/INPE- Instituto Nacional de Pesquisas Espaciais, Caixa Postal 515,

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

Marcelo G. NARCISO narciso@cnptia.embrapa.br

Embrapa Informática Agropecuária, Av. Dr. André tosello, s/n, Unicamp,

13083-970 – Campinas - SP, Brazil

J.E. BEASLEY j.beasley@ic.ac.uk

The Management School, Imperial College, London SW7 2AZ, England

Abstract. We present in this paper an application of the *Constructive Genetic Algorithm (CGA)* to the *Generalised Assignment Problem (GAP)*. 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, but with capacity constraints on the knapsacks. The *CGA* has a number of new features compared to a traditional *genetic algorithm*. These include a dynamic population size and the possibility of using heuristics. In our application of *CGA* to *GAP* we use a binary representation and an assignment heuristic which allocates items to knapsacks. Computational tests are presented using large publicly available problem instances taken from the literature.

Keywords: Constructive genetic algorithm, generalised assignment problem

1. Introduction

In this paper we consider an important combinatorial optimisation problem, the *Generalised Assignment Problem (GAP)*. This is the problem of assigning at minimum cost *n* items to *m* knapsacks (*n>m*), such that each item is assigned to exactly one knapsack subject to capacity constraints on the knapsacks.

Many real life applications can be formulated as a *GAP*, e.g. resource scheduling, the allocation of memory space in a computer, the design of a communications network with capacity constraints at each network node, assigning software development tasks to programmers, assigning jobs to computers in a network, vehicle routing problems, and

others, see De Maio and Roveda (1971), Balachandran (1976), Fisher and Jaikumar (1981) and Catrysse and Van Wassenhove (1992).

GAP is NP-hard, e.g. see Narciso and Lorena (1999). A number of algorithms in the literature are exact tree search algorithms [Ross and Soland (1975), Martello and Toth (1990)] and there are also a number of heuristics for the problem [Klastorin (1979), Fisher, Jaikumar and Van Wassenhove (1986), Jornsten and Varbrand (1990), Martello and Toth (1990), Catrysse and Van Wassenhove (1992), Lorena and Narciso (1996), Narciso and Lorena (1999)].

Genetic Algorithms (GA) have become popular in recent years as effective heuristics for NP-hard combinatorial optimisation problems such as the *GAP*. Today there are many variations on the general *GA* theme, but all such variations can be classified generically as **population heuristics** [Beasley (2002)], that is as heuristics which progressively evolve 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. We shall assume throughout this paper some familiarity on the part of the reader with genetic algorithms.

Holland (1975) originated *GA's*. For application to optimisation problems the first step in a *GA* is the definition of a representation – some way of encoding a solution to the problem under consideration, typically into a binary bit string. For example, if there are 7 binary (zero-one) variables in an optimisation problem then solutions can be represented as 0100010, 1100110, 1010101, etc – all of which are 7 digit binary bit strings. Solutions are evaluated though use of a *fitness function* and the fitness value associated with a solution indicates "how good" the solution is in terms of the original optimisation problem that is being considered.

Suppose now that, although we are sure of certain variable values in a solution, we are unsure of others. For example 101#1#1 represents the situation where we are sure of variable values for all except the fourth and sixth variables, unsure variable values being conventionally represented by #. 101#1#1 is an example of a **schema**. Clearly the schema:

101#1#1

comprises four solutions, namely:

1010101
1010111
1011101
1011111

which are formed by enumerating all combinations of possible values for each # in 101#1#1.

Holland put forward the *building block* hypothesis (schemata formation and conservation) as a theoretical basis for the *GA* mechanism. In this view avoiding disruption of good schemata is the basis for the good behaviour of a *GA*. One major difficulty with the building block hypothesis however is that schemata are only evaluated indirectly, via evaluation of specific solutions. This raises two problems:

1. a schema with v #'s has 2^v solutions (replace each # by zero or one) and inevitably, for computational reasons, we typically only gain limited information about the schema by examination of a limited number of the 2^v possible solutions

2. a solution with v decided variables is a member of 2^v-1 schemata (replace each decided variable by # or not) and so when using an evaluated solution we are attempting to use information about many schemata.

In this paper we use the word **structure** as a generic term to cover both solutions and schemata.

The *Constructive Genetic Algorithm (CGA)* [Furtado (1998), Ribeiro Filho and Lorena (1998), Lorena and Furtado (2001)] was proposed recently as an alternative to the traditional *GA* approach. One of the objectives of a *CGA* is the direct evaluation of schemata. In this paper we present a *CGA* for the *GAP*.

This paper is organised as follow. Application of the *CGA* to the *GAP* is presented in Section 2. In a *CGA* the original problem is regarded as a bi-objective optimisation problem that drives the evolutionary search for well adapted structures. The relevant aspects of our *CGA* for the *GAP* are explained: structure representation, the evolution process, selection, recombination and mutation, and *CGA* pseudo-code is presented. Section 3 presents computational tests using large publicly available problem instances from the literature, providing insights into *CGA* performance.

2. CGA for the GAP

2.1 GAP formulation

The *GAP* is best described using knapsack problems. Readers unfamiliar with knapsack problems (either single dimensional, bidimensional or multi-dimensional are referred to Martello and Toth (1979,1987,1990), Chu and Beasley (1998a), Lin (1998) and Beasley (2000). 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 as the capacity of knapsack i, assign each item j to exactly one knapsack i, not exceeding knapsack capacities. Defining x_{ij}=1 if item j is assigned to knapsack i, =0 otherwise, the *GAP* can be formulated as:

$$\text{minimise} \quad \sum_{i=1}^{m} \sum_{j=1}^{n} p_{ij}x_{ij} \qquad \{\text{minimise total cost}\}$$

$$\text{subject to:} \quad \sum_{j=1}^{n} w_{ij}x_{ij} \leq c_i \qquad \forall i \in M = \{1,\ldots,m\} \qquad \{\text{respect knapsack capacities}\}$$

$$\sum_{i=1}^{m} x_{ij} = 1 \qquad \forall j \in N = \{1,\ldots,n\} \qquad \{\text{all items assigned}\}$$

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

There have been relatively few papers presented in the literature applying genetic algorithms to the *GAP*. Chu and Beasley (1997) presented an algorithm where a solution was represented by an assignment of all items to knapsacks. Violation of knapsack capacities was dealt with in their genetic algorithm by use of unfitness (see Chu and Beasley (1998b)), which separates the measurement of constraint violation from the measurement of objective function quality (fitness). They presented computational results

for a large number of randomly generated problems involving up to 20 knapsacks and 200 items.

Wilson (1997), working independently from Chu and Beasley (1997), used the same representation. In his approach the initial population is composed of solutions that perform well with respect to the *GAP* objective function, but are infeasible. He then uses a genetic algorithm to evolve a feasible solution. Once a feasible solution has been found the genetic algorithm stops and that feasible solution is subjected to a local improvement procedure in an attempt to improve its *GAP* objective function value, whilst maintaining feasibility. He presented computational results for a large number of randomly generated problems involving up to 50 knapsacks and 500 items.

In order to apply the *CGA* to the *GAP* we will view the problem as a bi-objective optimisation problem. Note here that our *CGA* approach is distinctly different from the approaches adopted in Chu and Beasley (1997) and Wilson (1997). We first describe our structure representation.

2.2 Structure representation

For structure representation we used a sequence of n characters, where n is the number of items. This n character representation contains just three symbols:

1 to indicate that the item is a seed item and has been assigned to a knapsack
0 to indicate that the item is a non-seed item which will be assigned to a knapsack by a heuristic
to indicate that the item is temporarily out of the problem (an undecided item).

Seed items are initially assigned to the m knapsacks, exactly one per knapsack, so there are always precisely m 1's in each structure.

For example, considering a problem with n=7 items and m=3 knapsacks, we could have a structure S = (#1#01#1), where the 3 seed items are: item number 2 assigned to knapsack 1, item number 5 assigned to knapsack 2, and item number 7 assigned to knapsack 3. Item 4 has the label 0 and will be assigned to one of the knapsacks according to an assignment heuristic. Items 1, 3 and 6 have the label # and are temporarily not being considered. As S contains #'s it is a schema. By contrast the structure S =(0100101) is not a schema but a solution.

The following assignment heuristic is used to translate any structure S into a solution to the underlying *GAP*.

Assignment Heuristic - AH

---

1 – Assign the m items with label 1 to the m knapsacks, and update the knapsack capacities accordingly

2 – Assigning the other n-m items to the knapsacks (labels 0 and #):
- 2.1 – Solve the m knapsack problems separately exactly
- 2.2 – Update the knapsack capacities for the items assigned to exactly one knapsack
- 2.3 – Resolve the m knapsack problems separately exactly for the remaining items
- 2.4 – Update the knapsack capacities for the items assigned to exactly one knapsack

2.5 – For each item j remaining, assign it to the knapsack i^* that minimises w_{i^*j}
2.6 – If the solution obtained is not feasible for the *GAP*, restart the assignment of the n-m items (the m seed items were already assigned in step 1), assigning (if possible) item j to knapsack i^* that minimises w_{i^*j} and for which capacity is not violated

3 – If the solution is feasible for the *GAP* improve the solution with the second part of MMTH (see Lorena and Narciso (1996))

4 – Discard from knapsacks any items with label # in S

2.3 The bi-objective problem

Let C be the set of all 3^n structures that can be generated by the *0-1-#* representation we have adopted, and consider two functions f and g, defined as $f:C \to \Re_+$ and $g:C \to \Re_+$ such that $g(S) \geq f(S)\ \forall S \in C$. We define the double fitness evaluation of a structure due to functions f and g as *fg-fitness*.

For the *GAP* the function g represents the cost of items assigned to knapsacks after application of the assignment heuristic *AH*. Letting $C_i(S)_{AH}$ represent the items assigned to knapsack i after application of *AH* to structure S we have that $g(S) = \sum_{i=1}^{m} \sum_{j \in C_i(S)_{AH}} p_{ij}$. Note here that from step 4 in *AH* above any item j assigned a label # in S is not assigned to any knapsack at the end of *AH* and so is not included in $g(S)$.

For the *GAP* the function f represents the cost of items assigned to knapsacks after taking the solution produced by *AH* and moving a single item between knapsacks. To define f the following *MAH* heuristic is applied to S, producing an additional move of one item between two knapsacks:

Modified Assignment Heuristic – MAH

---

1. Apply *AH* to S
2. Over all the items in knapsacks presenting label 0 in S let j^* be the item with the most costly assignment (ties broken arbitrarily)
3. Let i^* be the knapsack corresponding to the least cost assignment of item j^* (ties broken arbitrarily)
4. Assign (move) item j^* to knapsack i^*

Letting $C_i(S)_{MAH}$ represent the items assigned to knapsack i after application of *MAH* to structure S we have that $f(S) = \sum_{i=1}^{m} \sum_{j \in C_i(S)_{MAH}} p_{ij}$. Clearly we have $g(S) \geq f(S)$. The difference $g(S) - f(S) \geq 0$ can be interpreted as the cost of a wrong assignment if the resulting *GAP*

solution is feasible. Considering our representation we have that *fg-fitness* values increase as the number of # labels decrease and therefore structures with few # labels have higher *fg-fitness.*

In our *CGA* for the *GAP fg-fitness* plays two roles:

- ***interval minimisation*** we would like to search for a $S \in C$ that minimises $g(S) - f(S)$, since obviously a good quality solution to the *GAP* cannot be improved by moving a single item between knapsacks.
- ***g maximisation*** we would like to search for a $S \in C$ that maximises $g(S)$, since we need to increase cost to ensure feasibility. A structure that has very few items assigned to knapsacks, e.g. because it is a schema in which most items are labelled #, will have low cost, but will not be a feasible solution to the *GAP*. This objective can be viewed as encouraging the process to move from schemata to solutions.

Hence our *CGA* implicitly considers the following *Bi-objective Optimisation Problem (BOP)*:

$$\begin{array}{ll} \textit{minimise} & g(S) - f(S) \\ \textit{maximise} & g(S) \\ \textit{subject to:} & g(S) \geq f(S)\ \forall S \\ & S \in C \end{array}$$

Zitzler and Thiele (1999), amongst others, have recognised that genetic algorithms (evolutionary algorithms) are a common approach to dealing with multiobjective problems. One genetic algorithm approach to dealing with such problems is simply to aggregate objectives together, but this does require a scaling between differing objectives to be defined. As Zitzler and Thiele (1999) have noted this "*requires profound domain knowledge that is often not available*". The approach taken in our *CGA* is not to aggregate $g(S)$ and $f(S)$ together but to consider them separately.

2.4 The evolution process

The *BOP* defined above is not directly considered as the set X is not completely known. Instead we consider an evolution process to attain the objectives (*interval minimisation* and *g maximisation*) of the *BOP*.

At the beginning of the process, two *expected values* are given to these objectives:

- for the *g maximisation* objective we use a value $g_{max} > \max_{S \in X} g(S)$ that is an upper bound on the objective value
- for the *interval minimisation* objective we use a value dg_{max}, obtained from g_{max} using a real number $0 < d \leq 1$.

The evolution process proceeds using an adaptive rejection threshold, which considers both objectives. Given a parameter $a \geq 0$ a structure S is discarded from the population if:

$$g(S) - f(S) \geq dg_{max} - a(g_{max} - g(S)) \qquad (1)$$

The right-hand side of equation *(1)* is the threshold for structure removal from the population and is composed of the expected value dg_{max} associated with interval minimisation and the measure $(g_{max} - g(S))$, which is the difference between g_{max} and $g(S)$ evaluations. For $a=0$ equation *(1)* is equivalent to comparing the interval length associated

with S against the expected length dg_{max}. When $a > 0$ structures containing a high number of # labels (i.e. structures which are schemata) have a higher probability of being discarded as, in general, they have higher differences $(g_{max} - g(S))$ since g_{max} is fixed and $g(S)$ is smaller for schemata than for solutions.

In our approach the value of the *evolution parameter* a is related to time in the evolution process. As initially good schemata need to be preserved for recombination a starts from zero and then slowly increases (in steps of e) from generation to generation. The population at evolution time a (denoted by P_a) is dynamic in size according to the value of the adaptive parameter a. The population can shrink to zero during the evolution process (extinction).

Rearranging equation *(1)* a structure S should be discarded from the population if

$$d(S) = [dg_{max} - (g(S) - f(S))]/[d(g_{max} - g(S))] \leq a$$

A key computational point here is that $d(S)$ (which we refer to as the **rank** of a structure) is fixed in value at the time the structure is created. The evolution parameter a on the other hand does increase over time. Hence we need only compute the rank $d(S)$ of a structure once and then continually compare it against the changing evolution parameter in deciding whether to discard the structure or not. Obviously the larger the rank the longer a structure will exist in the population, since a structure is only discarded once a becomes large enough to satisfy $d(S) \leq a$.

2.5 Selection, recombination and mutation

Selection of individuals can be made in several ways. *CGA* has been tested with a number of optimisation problems and in all cases an appropriate approach is that the population is kept ordered using a key value that involves for each structure its *fg-fitness* and its proximity to a feasible solution. Then, several times in a generation, two structures are randomly selected, one from the best part of the population and the other from the whole population, and these are recombined to form (one or more) new structures (see Lorena and Furtado (2001)).

The manner of recombination depends 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* approach use mutation over structures that represent feasible solutions for the problem (see Lorena and Furtado (2001)).

In our *CGA* for the *GAP* the population is ordered in increasing value of $D(S)=(1+d(S))/(n-n_{\#}(S))$, where $d(S)=(g(S)-f(S))/g(S)$ and $n_{\#}(S)$ is the number of # labels in S. Structures with small $n_{\#}(S)$ and/or with small $d(S)$ appear in first order positions.

The method used for selecting structures for recombination takes one structure from the n first positions in the ordered population (*base*) and the second structure from the whole population (*guide*). Before recombination the first structure is changed to generate a structure representing a 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).

Recombination merges information from both selected structures, but preserves the number of 1 labels (number of knapsacks) in the new generated structure.

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 $S_{base}(j) = \#$ *and* $S_{guide}(j)=0$ *then* $S_{new}(j) \neg 0$
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 structures are created by recombination. If a new structure 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 *k* with label 0 ***do***
    Interchange the labels on positions *j* and *k* generating an offspring S_{new} ;
      {offspring generation}
    Interchange the labels on positions *j* and *k* ; *{return to the original S}*
  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 structure insertion, the population is scanned to remove all structures S satisfying the condition $d(S) \leq \alpha$. As described earlier in this paper, the evolution parameter α is initially set to zero and slowly increased at each generation.

2.6 CGA pseudo-code
The pseudo-code for our *CGA*, Lorena and Furtado (2001), is:

Constructive Genetic Algorithm - CGA

$\alpha := 0$	*{initialise* α*}*
$\varepsilon := 0.01$	*{step value for time advance}*
Initialise P_a	*{initial population}*
Evaluate P_a	*{compute fg-fitness}*

For all $S \hat{I} P_a$ compute $d(S)$ *{rank computation}*
While (not stopping condition) ***do***
Discard from P_a all $S \hat{I} P_a$ satisfying $d(S) £ a$ *{evolution test}*
$a := a + e$ *{increment a}*
Produce P_a from P_{a-e} *{generate new population}*
Evaluate P_a *{compute fg-fitness}*
For all new $S \hat{I} P_a$ compute $d(S)$ *{rank computation}*
End_while

As the evolution parameter a increases, the population size initially increases and then starts to decrease until eventually the population becomes empty. Two stopping conditions are considered: stop when the population is empty, or when a pre-defined generation limit is reached.

To compute the upper bound g_{max}, at the very beginning of the process, a structure S representing a solution (i.e. containing no #'s) is randomly generated and g_{max} set equal to $g(S)$. For all of the computational results presented in this paper an initial population of size 200 was randomly created comprising structures with exactly m (number of knapsacks) label 1's and $n/5$ label 0's with the remainder of the structure being # labels.

3. Results

In this section we outline the performance of our *CGA* for the *GAP* when coded in *C* and run on a SUN ULTRA 2 200 MHz machine with 256 Mb RAM.

A set of large-scale problem instances were solved of dimensions, $m \times n$, (5×100), (5×200), (10×100), (10×200), (20×100) and (20×200), which were taken from OR-Library [Beasley (1990,1996)]. 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 *CGA* results (best feasible solution found over ten replications of *CGA*) compared with the best known solutions as reported in Chu and Beasley (1997). The *CGA* parameters were set to:

$d = 0.15$
$e = 0.1$ *for* $0 £ a £ 1$ *and* $e = 0.01$ *for* $a > 1$
stopping condition: maximum number of generations = 150 or the population is empty

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

Considering Table 1 we have that, over all problems, the average percentage deviation from the best known solution for our *CGA* [100(best *CGA* solution – best known solution)/(best known solution)] is 0.2724%, with the average computation time being 646 seconds. In addition for 11 of the 24 instances we find the best known solution.

The *CGA* solutions reported in Table 1 are very close to the best known solutions, obtained in the *GA* implementation of Chu and Beasley (1997) who ran their *GA* until five million

children had been generated. It can be conjectured that the computation time involved in our *CGA* is small compared to their *GA*. The computer times are not directly comparable as their *GA* was run on a different machine. We also conjecture that this computational time difference is the reason why the algorithm of Chu and Beasley (1997) outperforms our *CGA* (in terms of always finding equal, or better, solutions).

Table 1 Computational results

Problem	Best known solution	*CGA* solution	Number of generations	*CGA* times (seconds)
A 5×100	1698	***best***	51	253
A 5×200	3235	***best***	1	502
A 10×100	1360	***best***	87	308
A 10×200	2623	***best***	72	930
A 20×100	1158	***best***	1	350
A 20×200	2339	***best***	19	860
B 5×100	1843	***best***	150	302
B 5×200	3553	3601	150	432
B 10×100	1407	1410	150	165
B 10×200	2831	***best***	150	949
B 20×100	1166	***best***	150	474
B 20×200	2340	2347	150	683
C 5×100	1931	1941	150	195
C 5×200	3458	3460	150	405
C 10×100	1403	1423	150	203
C 10×200	2814	2815	150	498
C 20×100	1244	***best***	150	479
C 20×200	2397	***best***	150	1059
D 5×100	6373	6479	150	259
D 5×200	12796	12823	150	1253
D 10×100	6379	6390	150	497
D 10×200	12601	12634	150	1321
D 20×100	6269	6280	150	974
D 20×200	12452	12471	150	2158

best means that the *CGA* solution was the same as the best known solution

With regard to the sensitivity of our results with respect to d it is clear from equation *(1)* that the larger the value of d the larger the population will become. Our computational experience has been that for small d, the population increases slowly, but remains small. For large d the population grows large, presenting storage problems, but this eventually leads to good structures.

With regard to the sensitivity of our results with respect to e in order to investigate this we resolved the problems shown in Table 1, but with two different e schedules. Table 2 presents the results where $e = 0.01$ *for* $0 \leq a \leq 1$ and $e = 0.001$ *for* $a > 1$. Considering Table 2 we have that, over all problems the average percentage deviation from the best known solution for our *CGA* is 1.9090%, with an average computation time of 688 seconds. For 2 of the 24 instances we find the best known solution. It is clear that these results are not as good as those shown in Table 1.

Table 2 Computational results $\varepsilon = 0.01$ *for* $0 \leq \alpha \leq 1$ and $\varepsilon = 0.001$ *for* $\alpha > 1$

Problem	Best known solution	*CGA* solution	Number of generations	*CGA* times (seconds)
A 5×100	1698	1699	150	704
A 5×200	3235	***best***	1	519
A 10×100	1360	1361	150	332
A 10×200	2623	2625	150	947
A 20×100	1158	***best***	1	375
A 20×200	2339	2340	150	850
B 5×100	1843	1897	150	321
B 5×200	3553	3642	150	442
B 10×100	1407	1414	150	189
B 10×200	2831	2923	150	957
B 20×100	1166	1171	150	489
B 20×200	2340	2357	150	678
C 5×100	1931	1969	150	186
C 5×200	3458	3539	150	405
C 10×100	1403	1445	150	231
C 10×200	2814	2891	150	523
C 20×100	1244	1267	150	457
C 20×200	2397	2482	150	1105
D 5×100	6373	6602	150	272
D 5×200	12796	13084	150	1268
D 10×100	6379	6691	150	517
D 10×200	12601	12690	150	1356
D 20×100	6269	6507	150	921
D 20×200	12452	13019	150	2458

Table 3 presents the results where $\varepsilon = 1$ *for* $0 \leq \alpha \leq 1$ and $\varepsilon = 0.1$ *for* $\alpha > 1$. Considering Table 3 we have that, over all problems the average percentage deviation from the best known solution for our *CGA* is 1.8395%, with an average computation time of 707 seconds. For 3 of the 24 instances we find the best known solution. It is clear that these results are not as good as those shown in Table 1. With respect to Table 2 they are marginally better.

Table 3 Computational results $\varepsilon = 1$ *for* $0 \leq \alpha \leq 1$ and $\varepsilon = 0.1$ *for* $\alpha > 1$

Problem	Best known solution	*CGA* solution	Number of generations	*CGA* times (seconds)
A 5×100	1698	1699	150	897
A 5×200	3235	***best***	1	535
A 10×100	1360	***best***	1	341
A 10×200	2623	2624	150	976
A 20×100	1158	***best***	1	328
A 20×200	2339	2340	150	873
B 5×100	1843	1891	150	378
B 5×200	3553	3645	150	427
B 10×100	1407	1416	150	195
B 10×200	2831	2910	150	1023

B 20×100	1166	1175	150	512
B 20×200	2340	2357	150	691
C 5×100	1931	1963	150	222
C 5×200	3458	3513	150	413
C 10×100	1403	1452	150	248
C 10×200	2814	2901	150	510
C 20×100	1244	1265	150	413
C 20×200	2397	2473	150	1097
D 5×100	6373	6575	150	295
D 5×200	12796	13077	150	1231
D 10×100	6379	6681	150	497
D 10×200	12601	12701	150	1349
D 20×100	6269	6508	150	975
D 20×200	12452	13011	150	2543

4. Conclusions

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

Acknowledgements

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.

The authors would also like to acknowledge comments on an earlier version of this paper by the referee.

References

Balachandran, V. (1976) An integer generalized transportation model for optimal job assignment in computer networks, *Operations Research*, **24** 742-759.

Beasley, J.E. (1990) OR-Library: Distributing test problems by electronic mail, *Journal of the Operational Research Society*, **41** 1069-1072.

Beasley, J.E (1996) Obtaining test problems via Internet, *Journal of Global Optimization*, **8** 429-433.

Beasley, J.E. (2000) "Multidimensional knapsack problems". In *Encyclopaedia of Optimization*, Floudas, C.A and Pardalos, P.M. eds, Kluwer Academic Publishers, **Vol. III**, 517-521.

Beasley, J.E. (2002) "Population heuristics". In *Handbook of Applied Optimization*, Pardalos, P.M. and Resende, M. G.C. eds, Oxford University Press, Oxford, 138-157.

Catrysse, D. and Van Wassenhove, L.N. (1992) A survey of algorithms for the generalized assignment problem, *European Journal of Operational Research*, **60** 260-272.

Chu, P.C. and Beasley, J.E. (1997) A genetic algorithm for the generalised assignment problem, *Computers and Operations Research*, **24** 17-23.

Chu, P.C. and Beasley, J.E. (1998) A genetic algorithm for the multidimensional knapsack problem, *Journal of Heuristics*, **4** 63-86.

Chu, P.C. and Beasley, J.E. (1998) Constraint handling in genetic algorithms: the set partitioning problem, *Journal of Heuristics*, **4** 323-357.

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

Fisher, M.L. and Jaikumar, R. (1981) A generalized assignment heuristic for vehicle routing, *Networks*, **11** 109-124.

Fisher, M.L., Jaikumar, R. and Van Wassenhove, L.N. (1986) A multiplier adjustment method for the generalized assignment problem, *Management Science*, **32** 1095-1103.

Furtado, J.C. (1998) *Algoritmo Genético Construtivo na Otimização de Problemas Combinatoriais de Agrupamentos*. Ph.D. thesis, Instituto Nacional de Pesquisas Espaciais, São José dos Campos, Brazil.

Holland, J.H. (1975) *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press.

Jornsten, K.O. and Varbrand, P. (1990) Relaxation techniques and valid inequalities applied to the generalized assignment problem, *Asia-Pacific Journal of Operational Research*, **7** 172-189.

Klastorin, T.D. (1979) An effective subgradient algorithm for the generalized assignment problem, *Computers and Operations Research*, **6** 155-164.

Lin, E.Y.H (1998) A bibliographical survey on some well-known non-standard knapsack problems, *INFOR*, **36** 274-317.

Lorena, L.A.N. and Furtado, J.C. (2001) Constructive genetic algorithm for clustering problems, *Evolutionary Computation*, **9** 309-327.

Lorena, L.A.N. and Narciso, M.G. (1996) Relaxation heuristics for a generalized assignment problem, *European Journal of Operational Research*, **91** 600-610.

Martello, S. and Toth, P. (1979) "The 0-1 Knapsack Problem". In *Combinatorial Optimization* Christofides, N., Mingozzi, A., Toth, P. and Sandi, C. eds, John Wiley, 237-279.

Martello, S. and Toth, P. (1987) Algorithms for knapsack problems, *Annals of Discrete Mathematics*, **31** 213-257.

Martello, S. and Toth, P. (1990) *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley, New York.

Narciso, M.G. and Lorena, L.A.N. (1999) Lagrangean/surrogate relaxation for generalized assignment problems, *European Journal of Operational Research*, **114** 165-177.

Ribeiro Filho, G. and Lorena, L.A.N. (1998) A constructive algorithm for cellular manufacturing design. Paper presented at *EURO XVI - 16th European Conference on Operational Research*, July 1998, Brussels, Belgium.

Ross, G. T. and Soland, R. M. (1975) A branch and bound algorithm for the generalized assignment problem, *Mathematical Programming*, **8** 91-103.

Wilson, J.M. (1997) A genetic algorithm for the generalised assignment problem, *Journal of the Operational Research Society*, **48** 804-809.

Zitzler, E. and Thiele, L. (1999) Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach, *IEEE Transactions on Evolutionary Computation*, **3** 257-271.