A Hybrid Genetic/Optimization Algorithm for a Task Allocation Problem

Atidel Ben Hadj-Alouane
James C. Bean
Katta G. Murty
Department of Industrial & Operations Engineering
University of Michigan
Ann Arbor, MI 48109

Technical Report 93-30

December 1996

A Hybrid Genetic/Optimization Algorithm for a Task Allocation Problem *

Atidel Ben Hadj-Alouane
James C. Bean
Katta G. Murty

Department of Industrial and Operations Engineering
University of Michigan
Ann Arbor, MI 48109-2117

December 16, 1996

Abstract

We consider the problem of designing a distributed computing system for handling a set of repetetive tasks on a periodic basis. Tasks assigned to different processors need communication link capacity, tasks executing on the same processor do not. The aim is to develop a design of minimum total cost that can handle all the tasks. We compare the performances of a genetic algorithm, a commercial $0-1$ integer programming software and a hybrid approach from the literature, in solving real instances of the problem.

*This work was supported in part by the National Science Foundation under Grant DDM-9018515 and DDM-9308432 to the University of Michigan.

1 Introduction

The task allocation problem arises in distributed computing systems where a number of tasks (dealing with files, programs, data, etc.) are to be assigned to a set of processors (computers, disks, consoles, etc.) to make sure that all tasks are executed within a certain cycle time. The aim is to minimize the cost of the processors and the interprocessor data communication bandwidth installed in order to achieve this objective.

In this paper, we deal with a specific task allocation problem in the automobile industry discussed by K. N. Rao [22]. In the modern automobile, many tasks such as integrated chassis and active suspension monitoring, fuel injection monitoring, etc., are performed by a subsystem consisting of micro-computers linked by high speed and/or low speed communication lines. The cost of the subsystem is the sum of the costs of the processors (or microcomputers), and the installation costs of the data links that provide interprocessor communication bandwidth.

Each task deals with the processing of data coming from sensors, actuators, signal processors, digital filters, etc., and has a throughput requirement in KOP (thousand operations per second). Several types of processors are available. For each, we are given its purchase cost and its throughput capacity in terms of the KOP it can handle. The tasks are interdependent; to complete a task we may need data related to some other task. Hence, if two tasks are assigned to different processors, they may need communication link capacity (in bits/second). Tasks executing in the same processor do not have communication overhead. Also, each task may have to be allocated only to potential processors in a specified subset. The communication load between two tasks is independent of the processors to which they are assigned. This assumption of uniform communication load is valid for local networks such as the one considered in this paper ([3] and [16]) and is valid for the real auto industry problem tested.

We also assume that if there is need for communication between a pair of processors, it is provided by a direct link between them. This assumption is reasonable since (1) the number of processors in the system is small, and (2) the company preferred the simple architecture resulting from this assumption.

Define

m	$=$	number of tasks to be performed
n	$=$	number of processors under consideration; only a subset of these will be installed
P_t	$=$	set of processors to which task t may be assigned
a_t	$=$	throughput requirement (in KOP) of task t, $t = 1$ to m
s_p, b_p	$=$	cost (\$), and capacity in KOP of processor p, $p = 1$ to n
$c_{tt'}$	$=$	cost (\$) of installing communication bandwidth between task pair t, t' if they are assigned to different processors (this cost term may depend on whether the link needed is a low-speed or hi-speed communication link).

To model this problem, we define the following $0-1$ decision variables for $t = 1$ to m, $p = 1$ to n.

$$x_{tp} = \begin{cases} 1, & \text{if task } t \text{ is assigned to processor } p \\ 0, & \text{otherwise} \end{cases}$$

$$y_p = \begin{cases} 1, & \text{if processor } p \text{ is used (i.e. allotted some tasks)} \\ 0, & \text{otherwise} \end{cases}$$

In term of these decision variables, the model for the minimum cost design for the micro-computer architecture is the following quadratic $0-1$ integer program

$$\min \quad f(x,y) = \sum_{t=1}^{m-1} \sum_{t'=t+1}^{m} c_{tt'}(1 - \sum_{p=1}^{n} x_{tp}x_{t'p}) + \sum_{p=1}^{n} s_p y_p$$

$$s.\ t. \quad \sum_{t=1}^{m} a_t x_{tp} \le b_p y_p, \ for\ p = 1\ to\ n \tag{1}$$

$$(T) \qquad \sum_{t=1}^{m} x_{tp} \le m y_p, \ for\ p = 1\ to\ n \tag{2}$$

$$\sum_{p \in P_t} x_{tp} = 1, \ for\ t = 1\ to\ m \tag{3}$$

$$x_{tp}, y_p \in \{0,1\}, \ for\ all\ t,\ p.$$

The first constraint (1) guarantees that the total KOP requirements of all the tasks assigned to processor p does not exceed its KOP capacity of b_p. Constraint (2) ensures that a processor is purchased if it is allotted at least one task. Constraint (3) guarantees that each task is assigned to exactly one processor. In this formulation, constraint (2) is redundant since it is implied by constraint (1), where the throughput requirements,a_t, are positive. The reason for including constraint (2) is because the penalty transformation, as shown later in Section 2, involves the relaxation of constraint (1) only. In parallel, every solution generated by the algorithm developed in this paper satisfies constraint (2), and not nescessarily constraint (1).

Note that the objective function is quadratic and nonconvex. This problem can be transformed into a $0-1$ linear integer program at the expense of increasing the number of $0-1$ variables in the model substantially. The linear relaxation of this model exhibits a large duality gap (about 70% in preliminary experiments). When the number of processors is only two, it can be transformed into a minimum capacity cut problem [27] and solved efficiently by network flow techniques. However, for three or more processors, the problem has been shown to be NP-hard [7] [11] [21].

The uniqueness of this particular problem is that it introduces fixed cost for the processors. Most studies in the literature consider the processor cost as a function of the task(s) assigned to it ([4], [6], [16], [17], [21], and [25]), i.e., include only variable processor costs.

Most task allocation algorithms in the literature are either based on branch-and-bound, or are heuristics based on constructive initial assignment, or on iterative assignment-improvement (as classified in [20]). In [4] and [25], branch-and-bound algorithms are given for task assignment in a heterogeneous multiprocessor system with no capacity constraints. Several other branch-and-bound-based techniques address problems with various types of constraints such as preference constraints, exclusion constraints, bounds on the number of tasks assigned to each processor and also capacity constraints (see [6], [16], and [17]). Constructive assignment heuristic methods include the clustering algorithm and the Banded Q Heuristic for the uncapacitated heterogeneous system [20], the allocation technique for the uncapacitated homogeneous system [23], and the graph matching approach for the heterogeneous system under various constraints [24]. In [23] and [24], the objective is to balance processor load and minimize interprocessor communication cost. Only two iterative assignment-improvement techniques are encountered in the literature. The first is the transformation method proposed in [20]. The second is the genetic algorithm presented in [14], which addresses the uncapacitated heterogeneous system and uses traditional genetic operators (see Section 3.1 for an introduction to genetic algorithms).

In [2], a genetic algorithm (GA) approach is developed for the Multiple-Choice Integer Program (MCIP). Some constraints are relaxed using a nonlinear penalty function. Then a genetic algorithm is used to solve the dual problem defined in terms of the penalty coefficients. The task allocation problem (T) shares the multiple-choice constraints (3) with the MCIP, but has a nonlinear objective function that is easy to compute. This is the major requirement for an efficient genetic algorithm. In this paper, we expand the approach developed for the MCIP to find good solutions for the task allocation problem. Note that (T) is nonlinear and, hence, the algorithm to be discussed here is not a direct application of the results in [2].

To the best of our knowledge, the closest work related to our task allocation problem is the hybrid allocation algorithm presented in [9]. It combines notions of heuristics and implicit enumeration techniques to solve the special case of (T) where all costs of processors are zero. Computational results presented later in this paper compare that hybrid allocation algorithm and our genetic algorithm for some problem instances. We are aware of no other published works that address this formulation.

It is interesting to note the similarity between the structure of our penalty transformation (PT_λ), defined in the next section, and the uncapacitated hub location problem formulated in [5]. Each of the variables x_{tp} is analogous to the variable representing the assignment of origin/destination pair to a pair of hubs. The processor "usability" variables, y_p, are similar to those that represent whether a certain location is a hub. In addition, the objective function of the uncapacitated hub location problem is very similar to the objective of (PT_λ), in the special case where $\lambda = 0$. See [15] and [19] for algorithms tailored for the uncapacitated hub problem. Therefore, the algorithm presented here can be applied to the uncapacitated hub location problem, but it may not be competitive with codes designed specifically for the hub location problem because of capacity constraints. Much of the machinery in our approach exists to handle the capacity constraints specific to the task allocation problem.

The remainder of this paper is organized as follows. The second section presents the penalty transformation and expands the strong duality results of [2] to the task allocation problem. The third section describes the genetic algorithm developed for (T). The last section presents computational results for real problems from the auto industry.

2 Penalty Transformation

The common characteristic between the problem discussed in [2], (MCIP), and (T) is that they both have the multiple-choice structure (constraints (3)). We relax constraint (1) with a penalty function and calculate values of y (constraint (2)) directly from x. The penalty term added to the objective function is the sum of weighted squares of violations of constraint (1), given as:

$$p_\lambda(x,y) = \sum_{p=1}^{n} \lambda_p[\min(0, b_p y_p - \sum_{t=1}^{m} a_t x_{tp})]^2,$$

where $\lambda = (\lambda_p) \in \Re^n$, is ≥ 0. Then, the relaxed problem is the following $0-1$ nonlinear program

$$(PT_\lambda) \qquad \begin{array}{ll} \min & f(x,y) + p_\lambda(x,y) \\ s.\ to & \sum_{t=1}^{m} x_{tp}, \leq m y_p,\ \ for\ p = 1\ to\ n \\ & \sum_{p \in P_t} x_{tp} = 1,\ \ for\ t = 1\ to\ m \\ & x_{tp}, y_p \in \{0,1\},\ \ for\ all\ t,\ p. \end{array}$$

For the MCIP, the penalty term turns a linear problem into a nonlinear problem. Loss of linearity of the objective function is not particularly critical for a GA and empirical tests show that the approach works well. The addition of a nonlinear objective in (T) adds no complexity.

This relaxation differs from the common Lagrangian relaxation in that the penalty term is nonlinear here and linear in the Lagrangian case. The Lagrangian approach is not effective here since it can result in a large duality gap [8].

In [2], for the case of multiple-choice integer programs, it is shown that there exist finite values for multipliers (represented by the vector λ) such that the relaxed problem solves the original problem exactly. That is, the nonlinear relaxation has the property of strong duality, that is, no duality gaps. In this paper we state similar results for the quadratic integer program, (T). Let $v(\cdot)$ be the global optimal objective value of problem $(\cdot)$. The following theorem describes the relationship between $v(PT_\lambda)$ and $v(T)$ for a fixed λ.

Theorem 1 (Fixed λ)

a– For a given $\lambda \geq 0$, if (x, y) is a global minimum of (PT_λ) and (x, y) satisfies constraint (1), then (x, y) is a global minimum of (T).

b– For a given $\lambda \geq 0$ and $\varepsilon > 0$, if (x, y) is ε-optimal to (PT_λ) and (x, y) satisfies constraint (1), then (x, y) is ε-optimal to (T).

The next theorem establishes the existence of some value of the vector λ for which Theorem 1-a can be applied. This establishes *strong duality* for the dual $\max_{\lambda \geq 0} v(PT_\lambda)$.

Theorem 2 (Strong Duality) *Let S_λ be the set of global optimal solutions to (PT_λ). If (T) is feasible, then there exists $\bar{\lambda} \geq 0$ and $(x, y) \in S_\lambda$ such that, for all $\lambda \geq \bar{\lambda}$, $p_\lambda(x, y) = 0$ and (x, y) is optimal to (T).*

The proofs of the above two theorems are similar to those presented in [2]. Indeed, these results only depend on the boundedness and discreteness of the set of feasible solutions of (PT_λ). In Theorem 2, a sufficient value of $\bar{\lambda}$ is given as $\bar{\lambda} = \max\{\lambda_0; 0\}e_n$, where

$$\lambda_0 = \max_{(x,y)\in X, \alpha(x,y)\neq 0} \left\{ \frac{f(x^*,y^*) - f(x,y)}{\alpha(x,y)} \right\}, \tag{4}$$

$e_n = (1,1,\ldots,1)^T \in \Re^n$, (x^*,y^*) is an optimal solution to (T),
$X = \{(x,y) \in \{0,1\}^{(mn+n)}$: Constraints (2) and (3) are satisfied$\}$, and
$\alpha(x,y) = \sum_{p=1}^{n}[\min(0, b_p y_p - \sum_{t=1}^{m} a_t x_{tp})]^2$.

This result is used in the genetic algorithm described in the next section.

3 Genetic Algorithm Approach

3.1 Introduction

Genetic algorithms are random search techniques that imitate natural evolution. They are initiated by selecting a population of randomly generated solutions to the problem. They move from one generation of solutions to another by breeding new solutions using only objective evaluation and the so-called genetic operators. In this sense, it can be classified as an iterative assignment-improvement technique. In general, genetic algorithms work with an encoding of the variables rather than the variables themselves. Typically, a solution is represented with a *string* of bits (also called chromosome). Each bit position is called a *gene*, and the values that each gene can take are called *alleles* (for an elementary account of genetic algorithms see [18]; and for more details about encoding see [10]).

A basic genetic algorithm has three main operators. The first operator is *reproduction* where strings (or solutions) are copied to the next generation with some probability based on the quality of their objective function value. The second operator is *crossover* where randomly selected pairs of strings are mated, creating new ones. The crossover operator is described in detail in [1] and [10]. The third operator, *mutation*, is the occasional random alteration of the allele of a gene. It plays a secondary role in genetic algorithms since, in practice, it is performed with a very small probability (on the order of 1/1000).

Mutation diversifies the search space and protects from loss of genetic material that can be caused by reproduction and crossover.

3.2 A Genetic Algorithm for (PT_λ), λ Fixed

We use a direct encoding in which a solution is represented by a string of length equal to the number of tasks. The allele of a given gene (for a given task), takes the value of the index of the processor to which it is assigned. Hence, each gene, t, of the string can take any integer in P_t. Note that the variables y_p, $p = 1$ to n, are not included in the representation. The reason is that they can be easily determined given any task-processor assignment. Since y_p represents whether the processor p is used or not, it is equal to 1 if at least one gene has a value equal to p, otherwise it is set to zero.

Given a current generation, the next generation is created using the following operations. The size of the population is held constant always, the operations are carried out in this order until the next generation is complete.

1. *Copy* the N_c top solutions. The solutions of the current generation are sorted in increasing order of the objective function value from top to bottom, then the top N_c solutions are copied into the next generation. N_c is usually fixed to 10% of the population size. This approach, called elitist reproduction [10], replaces the traditional probabilistic reproduction. The advantage of using elitist reproduction is that the best solution is monotonically improving from one generation to the next.

2. *Mate* random pairs: first, randomly select with replacement, two strings (parents) from the entire current generation. Unlike the traditional GA, each parent is selected by giving each individual in the entire generation equal probability. Next, create two offspring using the Bernoulli crossover (referred to as parameterized uniform crossover in [26]). For each gene, toss a biased coin and use the outcome to determine whether or not to interchange the alleles of the parents. Once the offspring are created they are evaluated, and only the one with better ob-

jective value is included in the new generation. Formally, the mating operation can be described as follows. Consider the parent strings $S_1 = s_{11}s_{12}\dots s_{1m}$ and $S_2 = s_{21}s_{22}\dots s_{2m}$, where s_{ij} is the allele of the jth gene of the string S_i. Similarly, let $O_1 = \mathbf{o}_{11}\mathbf{o}_{12}\dots\mathbf{o}_{1m}$ and $O_2 = \mathbf{o}_{21}\mathbf{o}_{22}\dots\mathbf{o}_{2m}$ be the offspring created by mating S_1 and S_2. Note that, here, the bold style is used for $\mathbf{o}_{ij}$ since the latter denotes the gene of O_i and not the allele. Then, the probabilistic interchange of the m alleles can be modeled using m independent $0-1$ random variables, say $W_1, W_2, \dots W_m$, having each the Bernoulli distribution with parameter p_c. This parameter represents the probability of crossover for each gene.

By performing these m independent trials, the genes of the two offspring are built. Therefore, The offspring genes are functions of the random variables W_j's, given as follows.
For $j = 1, \dots, m$,

$$\mathbf{o}_{1j} = (1 - W_j)s_{1j} + W_j s_{2j}$$

$$\mathbf{o}_{2j} = W_j s_{1j} + (1 - W_j)s_{2j}$$

In our experiments, we have found that biasing the selection toward one parent (i.e., $p_c \neq 1/2$) works well for many problems. This seems to support the development of a generalized form of building block.
Continue this process until the required number of children for the next generation are produced.

3. Create *mutations* by randomly generating a small number of entirely new solutions ($N_m = 1\%$ of the population size) and including them in the new generation. A random solution consists of a string of m random numbers, say $r_1r_2\dots r_m$, where r_t is uniformly distributed on the set P_t for each $t = 1, \dots, m$. This operation is clearly different from the gene by gene mutation since it involves bringing new members to the population. In [1] this is referred to as "immigration" and is shown to have an important role in preventing premature convergence of the population, especially when it is used with the elitist reproduction as opposed to the proba-

bilistic reproduction.

Continue this process until the required number of immigrants for the next generation are produced.

The following pseudo-code shows how to perform N_g iterations of the genetic algorithm described above. Note that the best feasible solution found is updated at each new generation.

<u>**ALGORITHM** A_1</u>

Initialization. Choose a population size, N (experiments show that $N \in [m, 3m]$ works well in general), this will be held constant through all the generations.
Set best objective value so far to infinity. Randomly generate and evaluate N solutions as described above in the immigration operation. Let G_1 be the set of these solutions. Set $k = 1$.

```
Main Step.  While ($k < N_g$)
            BEGIN
            Set $G_{k+1} = \emptyset$

              1. Sort the solutions in $G_k$ in increasing order of objective value from top
                 to bottom.
                 Include the top $N_c$ solutions in $G_{k+1}$.
                 If top solution in $G_k$ is feasible, update best feasible solution so far.

              2. While ($|G_{k+1}| < (N - N_m)$)
                 BEGIN (Crossover)

                   • Randomly (uniformly) select two solutions, $P_1$ and $P_2$ from $G_k$.
                   • Mate $P_1$ and $P_2$ to produce offspring $O_1$ and $O_2$.
                   • Evaluate $O_1$ and $O_2$, and include in $G_{k+1}$ the one with
                     lower objective value.

                 END (Crossover)

              3. Randomly generate $N_m$ solutions and include them in $G_{k+1}$.

            Set $k = k + 1$.
```

3.3 A Genetic Algorithm for (T)

In this section we briefly describe the dual genetic algorithm for (T) modified from [2]. The basic idea is to run algorithm A_1 with an oscillating penalty parameter, λ. In this algorithm, the vector λ maintains equal components through all iterations. This means that all the multipliers are adjusted with the same amount. This simple approach seems to work well with the constraints involved in the task allocation problem. Varying the multpliers independently, along the lines of the subgradient algorithm [8], is one way of extending the GA approach to other types of problems.

Here, we fix λ at a certain value (empirically determined) and run algorithm A_1 while keeping track of the penalty value of the top solution at every generation. Periodically (every N_f generations), we check for possible alteration of the value of λ. There are three possible situations.

1. If the top solution had a nonzero penalty for all N_f past generations (i.e., infeasible), the value of λ is increased to give the solution a higher penalty for violating the capacity constraints; hence, push it to the bottom of the population.

2. If the top solution had a zero penalty for all N_f past generations, the value of λ is decreased to allow more infeasible solutions to compete for the top position; in this case, the purpose of changing λ is to diversify the search (each infeasible solution acts as a seed that has the potential to lead to better solutions not explored so far).

3. If the penalty value of the top solution has oscillated between zero and nonzero at least once in the past N_f generation, continue the algorithm with the same value of λ.

Note that in the first two situations, all solutions of the current generation are re-evaluated with the new λ before the algorithm continues.

The value of λ is increased by the multiplicative factor β_1, and decreased (divided) by the factor β_2, where β_1 and β_2 are different real constants chosen empirically (β_1,$\beta_2 > 1$). In general, β_1 is chosen to be larger than β_2 to allow for a fast move towards feasibility at the early stages of the algorithm.

In [2], for the MCIP, the initial value of λ used is based on an approximation of λ_0 from an optimal solution of the linear programming relaxation. However, (T) is a nonconvex quadratic problem for which even the continuous version is difficult to solve. Therefore a different approximation of λ_0 is used. This approximation uses a heuristic solution in which only the cost of processors is minimized, since it is usually higher than interprocessor communication costs (by a factor of 100). The tasks (taken in any order) are assigned successively to the least expensive processor without exceeding its capacity. This is similar to the First-Fit bin packing heuristic [13] except that the bins (or processors) are sorted in increasing order of cost. Let (x_a, y_a) be the resulting solution. Then the following approximation is used. $\lambda^{(1)} = \max\{\hat{\lambda}_0; \epsilon\}e_n$, where $\epsilon > 0$ and arbitrary small, and

$$\hat{\lambda}_0 = 1/\bar{N}_1 \sum_{i \in G_1, \alpha(x_i,y_i) \neq 0} \left(\frac{f(x_a, y_a) - f(x_i, y_i)}{\alpha(x_i, y_i)} \right),$$

where G_1 is the set of solutions in the initial generation, $\bar{N}_1$ is the number of solutions, in G_1, and with nonzero penalty (i.e., with $\alpha(x_i, y_i) \neq 0$).

If run long enough, the procedure described above will find an optimal solution to (T) with probability one (see [1] for more detail). However, it is typically stopped heuristically by setting a limit to the number of generations created. A pseudo-code describing this heuristic procedure is given below.

ALGORITHM A_2

```
Initialization. Choose two scalars $\beta_1 > \beta_2 > 1$. The values used here are $\beta_1 = 8$ and $\beta_2 = 5.6$.
                Set $\lambda^{(1)} = \max\{\hat{\lambda}_0, \epsilon\}e_n$ as described above.
                Choose a frequency, $N_f$, for altering $\lambda$ (here we use $N_f = m/2$).
                Choose a value for $N_{max}$, the maximum number of generations to be
                created (here we use $N_{max} = 5000$).
                Randomly generate a population of solutions (as described in
                Initialization step of Algorithm $A_1$) and evaluate objective values.
                Set $k = 1$.
```

```
Main Step.  While ($k < N_{max}$)
            BEGIN
            Create a new generation as in Main Step of Algorithm $A_1$.
            If the last $N_f$ consecutive generations have top solution with nonzero
            penalty, let $\lambda^{(k+1)} = \beta_1 \lambda^{(k)}$, and re-evaluate current generation with $\lambda^{(k+1)}$.
            If the last $N_f$ consecutive generations have top solution with zero
            penalty, let $\lambda^{(k+1)} = \lambda^{(k)}/\beta_2$, and re-evaluate current generation with $\lambda^{(k+1)}$.
            Otherwise $\lambda^{(k+1)} = \lambda^{(k)}$.
            $k = k + 1$.
            END
```

4 Computational Results

The GA approach was applied to several instances of the task allocation problem, (T), ranging from 15 to 41 tasks and up to 12 processors. All programming was done in C and the computation below is reported in seconds on an IBM RS/6000-320H.

Two tests were run: six real data sets from auto manufacturing where the GA is compared to a commercial integer programming code, and two data sets from Hughes Air-defense System where the GA is compared to the hybrid approach in [9].

In the first test set, three instances for each of the sizes 20×6 (20 tasks $\times$ 6 processors) and 40×12 were considered. The integer programming code used is IBM's OSL [12] applied to the linearization of the quadratic problem (T) [18] with over $\frac{m(m-1)n}{2}$ more binary variables, and mn more constraints than those in (T), the most efficient linearization for this problem. The resulting $0-1$ integer program is shown in the Appendix.

Table 1 presents the results from 60 runs of the GA on 6 different problem instances. Each column shows the name and size of the problem instance, and reports the outcome of 10 runs of the GA, each with a different random seed. Each GA run was terminated

Table 1: **Problems from the Automobile Microcomputer system**

Computational effort over 10 random seeds

			dat1	dat2	dat3	dat4	dat5	dat6
		Problem	dat1	dat2	dat3	dat4	dat5	dat6
		Size	20 x 6	20 x 6	20 x 6	40 x 12	40 x 12	40 x 12
GA	Best	min	13804	11946	11120	39680	36575	35821
	obj. value	med	13866	11946	11228	39869	37214	36427
	found	max	13903	11946	11864	41149	38767	36568
		min	135	421	274	2521	649	551
	Generations	med	1021	1146	748	3377	3782	4266
		max	3444	2933	1833	4863	4900	4857
		min	3.43	10.56	6.94	205.21	52.79	44.80
	CPU (sec)	med	25.93	28.74	18.95	274.89	307.61	346.84
		max	87.48	73.56	46.45	395.85	389.54	394.89
OSL	Solution		14839	13097	12288	43547	43673	39152
	Nodes		23055	2492	†	6691	†	582
	CPU (sec)		3600.04	486.46	†	10589.10	†	2050.53

† Shut down with no solution better than the initial bound after 18000 seconds of CPU time.

after 5000 generations. However, the table shows the number of generations and CPU times elapsed until the best solution was found. Population sizes of 100 and 120 were used for the 20×6 and 40×12 problems respectively. Each column also reports the best solution found by IBM's OSL package in a certain time frame (about one day real time). In the last two lines, the number of nodes and the associated CPU times, are those required by OSL's branch-and-bound routine to find the reported best solution value found.

Note that the GA was able to find better solutions for these problems, in 100 to 400 CPU seconds, than the $0-1$ IP software package OSL was able to find in 2000 to 11000 CPU seconds. Also, note that computation time of the GA increases with problem size in a reasonable manner. These results show that the GA outperforms OSL in all problem instances of Table 1. In fact, the GA was at least 7 times faster than OSL; moreover, it found solutions that are at least 4% better. Note that for one instance (problem dat3), OSL did not find any solution better than the initial bound (solution of the First-Fit bin packing heuristic) even after 18000 seconds of CPU time.

For the second test set, there is only one instance for each of the 15×5 and 41×4 problems. These two problems are part of an experimental study found in [9]. They are special cases of (T) since all processor costs are zero ($s_p = 0$ for $p = 1, 2, \ldots n$). The 41×4 problem was derived from an existing Hughes-built air defense system and the following variation was considered: eight objects were preallocated, specifically, $x_{61} = x_{10,1} = 1$, $x_{14,2} = x_{24,2} = 1$, $x_{31,3} = x_{34,3} = 1$, and $x_{21,4} = x_{41,4} = 4$. Therefore, for this specific problem, the genetic algorithm is slightly modified by fixing the alleles of the genes corresponding to the above preallocation. Note that fixed genes are not affected by the crossover operator. Although this modification does not seem very efficient (since unnecessary crossover of the fixed genes is still performed), computation shows that the GA still preserves its high performance.

As for the 15×5 problem, it was designed to be difficult by making the data traffic and the graph associated with it symmetric and without cycles of length three. In [9], these two problems are solved either optimally or heuristically using a Hybrid Allocation (HA) algorithm based on implicit enumeration.

Table 2 shows results from 20 runs of the GA on the two problems described above against those found in [9] using the HA approach. For each problem instance, GA runs were performed with different random seeds and were also terminated after 5000 generations. Population sizes of 100 and 120 were used for the 15×5 and 41×4 problems respectively. Computation for the HA algorithm was carried out on an Amdahl 470 using a PL/I compiler. Time in terms of CPU seconds used is not available; therefore only the number of iterations to find the best solution is reported.

¿From Table 2, note the GA solutions were at least as good as HA's. In fact, GA found better solutions than HA for the 15×5 problem: it found the optimal solution (known to be 16) in 9 out 10 runs in at most 7 seconds. However, the best solution found by the HA algorithm, using various branching and bounding strategies, has a value of 18.

Results of GA and HA for the 41×4 problem are comparable, since the HA algorithm

Table 2: **Problems from Hughes-built Air Defense system**

Computational effort over 10 random seeds

		Problem	15 x 5	41 x 4
GA	Best	min	16	53
	obj. value	med	16	53
	found	max	17	80
		min	76	26
	Generations	med	158	170
		max	397	4400
		min	1.31	2.02
	CPU (sec)	med	2.73	13.23
		max	6.87	342.60
HA	Solution		18	53
	Iterations		†	16787

† Different methods were used but no computational effort was reported.

found the optimal solution and so did the GA for most runs. Note that the HA algorithm found the optimal solution in 16787 iterations. However, it took 645 additional iterations to prove its optimality.

5 Conclusion

A genetic algorithm is proposed for the $0-1$ integer program with nonlinear objective function. The application of this GA approach to the task allocation problem shows excellent results for various real problems. In all instances considered, it outperformed OSL and HA in terms of solution quality and computational time. The simplicity of the GA's data structure, its robustness, and the ease with which it can be adopted to various platforms are also appealing. Moreover, this algorithm can be easily modified to handle other variations of the task allocation problem.

Acknowledgment

We would like to thank K. N. Rao for bringing this problem to our attention and providing us with some data. Our thanks also go to anonymous referees for their insightful comments and suggestions.

Appendix

A Linearization of (T)

$$
\begin{aligned}
\min \quad & \sum_{t=1}^{m-1} \sum_{t'=t+1}^{m} c_{tt'}\left(1 - \sum_{p=1}^{n} z_{tt'p}\right) + \sum_{p=1}^{n} s_p y_p \\
s.\ to \quad & \sum_{t'=t+1}^{m} z_{tt'p} + \sum_{t'=1}^{t-1} z_{t'tp} - (m-1)x_{tp} \le 0, \ for\ t = 1\ to\ m, p = 1\ to\ n \qquad (5) \\
& \sum_{t=1}^{m} a_t x_{tp} \le b_p y_p, \ for\ p = 1\ to\ n \\
& \sum_{t=1}^{m} x_{tp} \le m y_p, \ for\ p = 1\ to\ n \\
& \sum_{p=1}^{n} x_{tp} = 1, \ for\ t = 1\ to\ m \\
& x_{tp}, y_p \in \{0,1\}, \ for\ all\ t,\ p,
\end{aligned}
$$

where for $t = 1$ to m, $t' = t+1$ to m, and $p = 1$ to n,

$$
z_{tt'p} = \begin{cases} 1, & \text{if both tasks } t \text{ and } t' \text{ are assigned to processor } p \\ 0, & \text{otherwise.} \end{cases}
$$

Constraint (5) in the above formulation insures that the processor p to which task t is assigned, is not assigned more than $m-1$ other tasks.

References

[1] J. C. Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA Journal On Computing*, 6:154–160, 1994.

[2] A. Ben Hadj-Alouane and J. C. Bean. A genetic algorithm for the multiple–choice integer program. *Operations Research*, 45(1), 1997.

[3] A. Billionnet. Allocating tree structured programs in a distributed system with uniform communication costs. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):445–448, April 1994.

[4] A. Billionnet, M. C. Costa, and A. Sutter. An efficient algorithm for a task allocation problem. *Journal of the Association for Computing Machinery*, 39:502–518, 1992.

[5] J. F. Campbell. Integer programming formulations of discrete hub location problems. *European Journal of Operational Research*, 72:387–405, 1994.

[6] A. Dutta, G. Koehler, and A. Whinston. On optimal allocation in a distributed processing environment. *Management Science*, 28(8):839–853, August 1982.

[7] A. Fernández-Baca, D.and Medepalli. Approximation algorithms for certain assignment problems in distributed systems. Technical Report 91-17, Department of Computer Science, Iowa State University, Iowa City, IA, 1991.

[8] M. L. Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management Science*, 27(1):1–18, 1981.

[9] A. Gabrielian and D. B. Tyler. Optimal object allocation in distributed systems. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 88–95, 1984. San Francisco, Calif., May 14-18.

[10] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing Company, Inc., 1989.

[11] M. Gursky. Some complexity results for a multiprocessor scheduling problem. Private Communication from H. S. Stone, 1981.

[12] IBM Corporation. *Optimization Subroutine Library Guide and Reference*, 1990.

[13] D. S. Johnson. Fast algorithms for Bin-Packing. *Journal of computer and system sciences*, 8:272–314, 1974.

[14] Y. S. Kim, Y. C.and Hong. A task allocation using a genetic algorithm in multicomputer systems. *IEEE TENCON*, pages 258–261, 1993.

[15] J. G. Klincewicz. Heuristics for the p-hub location problem. *European Journal of Operational Research*, 53:25–37, 1991.

[16] V. M. Lo. Heuristic algorithm for task assignment in distributed systems. *IEEE Transactions on Computers*, 37(11):1384–1397, 1988.

[17] P. R. Ma, E. Y. S. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, C-31(1):41–47, January 1982.

[18] K. G. Murty. *Operations Research: Deterministic Optimization Models.* Prentice Hall, 1994. To appear.

[19] M. E. O'Kelly. A clustering approach to the planar hub location problem. *Annals of Operations Research*, 40:339–353, 1992.

[20] C. Price and S. Krishnaprasad. Software allocation models for distributed computing systems. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 40–48, 1984. San Francisco, Calif., May 14-18.

[21] G. S. Rao, H. S. Stone, and T. C. Hu. Assignment of tasks in a distributed processor system with limited memory. *IEEE Transactions on Computers*, C-28(4):291–299, April 1979.

[22] K.N. Rao. Optimal synthesis of microcomputers for GM vehicles. Technical report, 1992.

[23] A. K. Sarje and G. Sagar. Heuristic model for task allocation in distributed computer systems. In *IEE Proceedings. Part E, Computers and Digital Techniques*, volume 138, pages 313–318, September 1991.

[24] C. C. Shen and W. H. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transactions*, C-34:197–203, 1985.

[25] J. B. Sinclair. Efficient computation of optimal assignments for distributed tasks. *Journal of Parallel and Distributed Computing*, 4:342–362, 1987.

[26] W. M. Spears and De Jong K. A. On the virtues of parametrized uniform crossover. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 230–236, 1991.

[27] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3:85–93, January 1977.