
Journal of Intelligent Manufacturing, 14, 599–616, 2003

A genetic algorithm-based solution methodology for modular design

ALI K. KAMRANI[1] and RICARDO GONZALEZ[2]

[1]*Department of Industrial Engineering, The University of Houston, Houston, TX 77204, USA*
[2]*Manufacturing Systems Engineering Department, The University of Michigan, Dearborn, MI 48128-1491, USA*

Combinatorial optimization problems usually have a finite number of feasible solutions. However, the process of solving these types of problems can be a very long and tedious task. Moreover, the cost and time for getting accurate and acceptable results is usually quite large. As the complexity and size of these problems grow, the current methods for solving problems such as the scheduling problem or the classification problem have become obsolete, and the need for an efficient method that will ensure good solutions for these complicated problems has increased. This paper presents a genetic algorithm (GA)-based method used in the solution of a set of combinatorial optimization problems. A definition of a combinatorial optimization problem is first given. The definition is followed by an introduction to genetic algorithms and an explanation of their role in solving combinatorial optimization problems such as the traveling salesman problem. A heuristic GA is then developed and used as a tool for solving various combinatorial optimization problems such as the modular design problem. A modularity case study is used to test and measure the performance of the developed algorithm.

Keywords: Modular design, similarity coefficients, combinatorial optimizations, genetic algorithms, heuristic methods

1. Introduction

The family of combinatorial optimization problems is characterized by having a finite number of feasible solutions. These problems abound in everyday life, particularly in engineering design. In principle, finding the optimal solution for a finite problem could be done by simple enumeration. However, real life problems are much more complicated and enumeration is frequently an impossible technique to use because the number of feasible solutions can be enormous.

Modular design is a design technique that can be used to develop complex products using similar components. Components used in a modular product must have features that enable them to be coupled together to form a complex product. Modular design can also be viewed as the process of producing units that perform discrete functions, which will then be connected together to provide a variety of functions. Modular design emphasizes the minimization of interactions between components, which enables components to be designed and produced independently from each other. Each component, designed for modularity, is supposed to support one or more functions. When components are structured together to form a product, they will support a larger or general function.

Developing a modular product can be classified as a combinatorial optimization problem as each company works endlessly for the optimality of their products. The family of combinatorial optimization problems is characterized by having a finite number of feasible solutions. These problems abound in everyday life, particularly in engineering design. In principle, finding the optimal solution for a finite problem could be done by simple enumeration. However, real life problems are much more complicated and

enumeration is frequently an impossible technique to use because the number of feasible solutions can be enormous.

Combinatorial optimization problems are derived from combinatorics, a branch of mathematics concerned with the problem of arranging and selecting discrete objects. However, as Law (1976) points out, while combinatorics tries to find if a particular arrangement of objects exists in a solution set, combinatorial optimization problems go a step further, trying to determine if a given arrangement can be an optimal solution to the problem at hand. Several methods and algorithms have been used in the solution of various combinatorial optimization problems. Sait and Youssef (1999) divided them into two major categories: exact algorithms, and approximation algorithms. The first category includes linear programming methods, dynamic programming, branch-and-bound, among many others. Linear programming approaches formulate the problem at hand as either a maximization or a minimization of a certain objective subject to a number of constraints. Dynamic programming is a search method suitable for optimization problems whose solution can be obtained as the result of a sequence of decisions or steps. Branch-and-bound searches the solution space tree trying to find the optimal solution to the problem. Most exact algorithms have the common problem that in nature, they all are enumerative; a fact that creates a problem when one tries to solve a real-life problem that has a lot of constraints and difficulties. These problems can have a very large number of solutions, and simply enumerating them and finding the best one is not efficient and sometimes impossible. Approximation algorithms are a partial solution to the problems found with enumeration techniques. They constitute the second category of algorithms discussed by Sait and Youssef (1999). Usually known as heuristic methods, these algorithms give a viable option when trying to solve very complex problems. They search only a portion of the solution space heuristically and find good solutions to a problem based on a number of constraints. These techniques are more efficient, as they give solutions ''faster'' than the enumeration methods discussed before. However, there is a trade-off in using heuristics instead of simple enumeration techniques. The trade-off consists in ''giving up'' the possibility of getting an optimal solution in order to achieve acceptable results in a reasonable amount of time. A third option for solving difficult combinatorial optimization problems are genetic algorithms (GAs). These are in essence, an extension to the approximation algorithms just discussed, but unlike those algorithms, GAs search the solution space more broadly, giving the user a better chance of getting an optimal solution in less time than the other two options.

This paper is organized as follows. The following section provides an introduction to modular design and its approach to product design. An introduction to GAs coupled with an explanation of their role in solving combinatorial problems such as the modular design problem is then given. A heuristic GA is then developed and presented in the next section. The algorithm developed in this paper is capable of solving various kinds of combinatorial optimization problems, but an emphasis is given to the modular design problem. Finally, a case study is presented to show how the heuristic GA developed in the previous section works. The results of this experiment are then discussed and a series of conclusions are presented to emphasize the benefits and drawbacks of using GAs in various optimization problems.

2. Design for modularity (DFMo)

The concept of modularity can provide the necessary foundation for organizations to design products that can respond rapidly to market needs and allow the changes in product design to happen in a cost-effective manner. Modularity can be applied to the design processes to build modular products and modular manufacturing processes. According to Salhieh *et al*. (1999) and Shirly (1992), modular products are products that fulfill various overall functions through the combination of distinct building blocks or modules, in the sense that the overall function performed by the product can be divided into sub-functions that can be implemented by different modules or components. Using the concept of modularity in product design focuses on decomposing the overall design problem into functionally independent sub-problems, in which interaction or interdependence between sub-problems is minimized. Thus, a change in the solution of one problem may lead to a minor modification in other problems, or it may have no effect on other sub-problems. That is, the modular design concept attempts to establish a design decomposition technique that reduces the interaction

between design components (or modules) to reduce the complexity and development time of a product. Kamrani and Salhieh (2000) introduced a new methodology for design for modularity (DFMo). The roadmap of this methodology is shown in Fig. 1.

2.1. *DFMo approach to product design*

To ensure that the voice of the customer is incorporated in the design process, the concept of DFMo has been extended to include the voice of customer in the needs analysis phase. This was facilitated by utilizing the concepts of Kano's model (Kano *et al*., 1984) and quality function deployment (QFD) (Shillito, 1994) as shown in Fig. 2.

2.2. *Needs analysis*

Several sources of information can be used to identify the needs. Such sources include potential customers, the company for which the design is being made, the competition, and any authorities that can impose restrictions on the product (standards, safety, etc.). Information collected from the customer are analyzed and organized according to Kano's model as following:

- *Expected requirements*: These are the basic requirements that customers expect to exist self-evidently. These requirements are satisfied through basic product/service characteristics.
- *Unspoken requirements*: These are product features that customers do not talk about and, though silent, are important and cannot be ignored.
- *Spoken requirements*: These are specific product/service features customers say they want in a product.
- *Unexpected requirements*: These are unspoken features of a product that make the product unique and distinguish it from the competition.

Existing product(s) are then decomposed into sub-

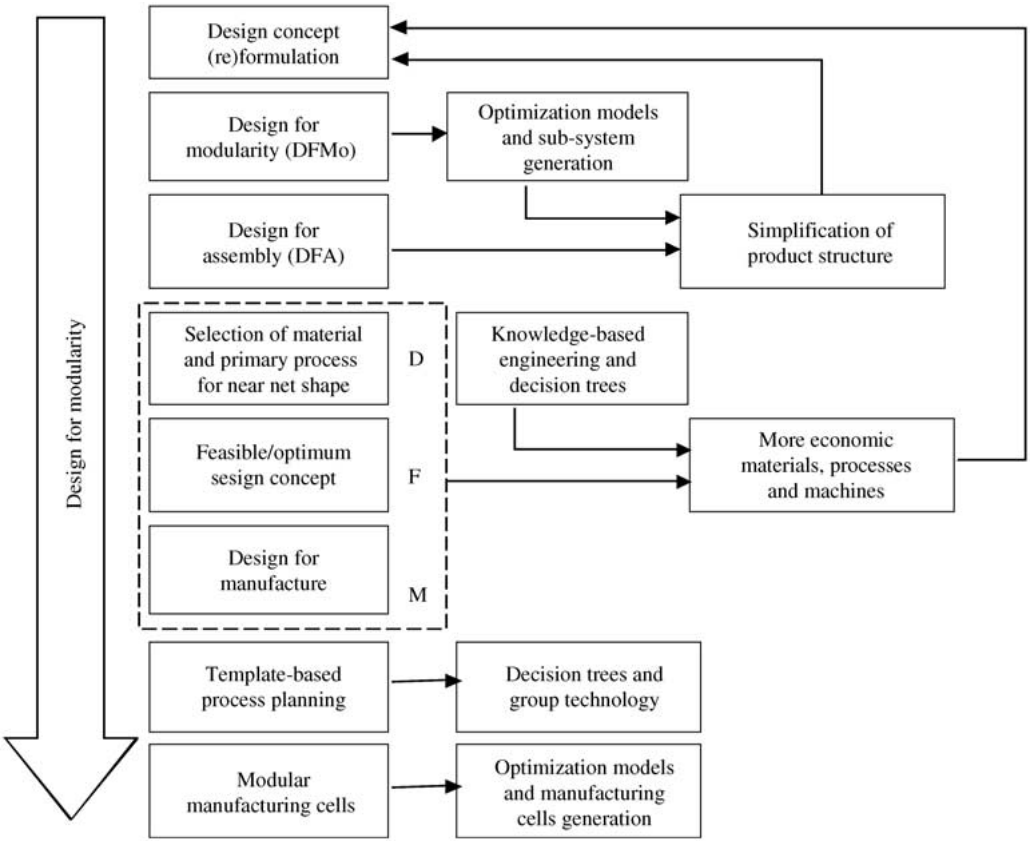

Fig. 1. Design for modularity life cycle.

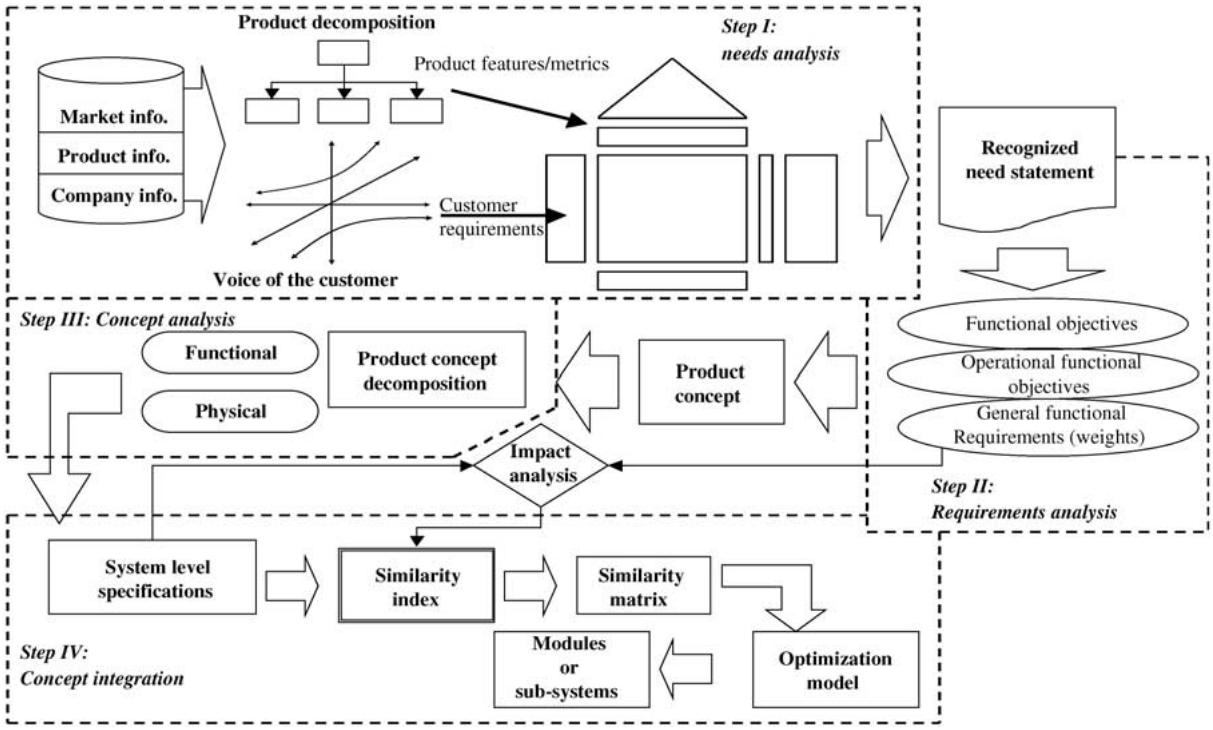

Fig. 2. QFD/DFMo integrated product design.

systems and/or sub-assemblies (physical elements) in order to identify the features. These physical elements are described using metrics or product features. Metrics describe the physical characteristics of the product or component that allow the product to fulfill its' function. Metrics form the basis of the design effort as design teams try to assign values to metrics and ensure that the product functionality and requirements has not been violated. It is important to identify the metrics in a manner that will allow the designer the freedom to be creative and the flexibility to choose among different design approaches. The house of quality is used for further analysis of the requirements and specifications.

2.3. *Requirement analysis*

Functional objectives (FOs) are an abstraction of the product function required to satisfy customer needs. FOs can be thought of as the basic operations or transformations that must be performed by the system to satisfy customers' primary needs. FOs are often somewhat general, but they should always employ action phrases such as ''to transform'', ''to support'', or ''to lift''. Operational functional requirements (OFRs) are detailed and specific information representing a set of constraints that the design must possess in order to satisfy the product intended function. OFRs are usually presented in the form of ranges. General functional requirements (GFRs) are the criteria set by the designer, based on the needs analysis, to evaluate the resulting design. FRS is those requirements that satisfy the customers' secondary needs, which could form a critical factor for the customer when comparing different competitive products that accomplish the same function. Several GFRs may exist for a product, some are more important than others, therefore different weights should be assigned to different requirements. Customer needs are considered an essential factor in weight assignment. Using a benchmarking study of competitive products could make weight assignment. Alternatively, it could be an input of the design team based on previous knowledge of the importance of such requirements.

2.4. *Product concept analysis*

Product/concept analysis is the decomposition of the product into its basic functional and physical elements. These elements must be capable of achieving the product's functions. Functional elements are defined as the individual operations and transformations that contribute to the overall performance of the product. Physical elements are defined as the parts, components, and subassemblies that ultimately implement the product's function. Product concept analysis consists of product physical decomposition and product functional decomposition. In product physical decomposition, the product is decomposed into its basic physical components which, when assembled together, will accomplish the product function. Physical decomposition should result in the identification of basic components that must be designed or selected to perform the product function. Product functional decomposition describes the products overall functions and identifies components functions. Also, the interfaces between functional components are identified.

2.5. *Product/concept integration*

System level specifications (SLS) are the one-to-one relationship between components with respect to their functional and physical characteristics. Functional characteristics are a result of the operations and transformations that components perform in order to contribute to the overall performance of the product. Physical characteristics are a result of the components' arrangements, assemblies, and geometry that implement the product function. Physical and functional characteristics, forming the SLS, are arranged into a hierarchy of descriptions that begins by the component at the top level and ends with the detailed descriptions at the bottom level. Bottom level descriptions (detailed descriptions) are used to determine the relationships between components, 1 if the relationship exists and 0 otherwise. This binary relationship between components is arranged in a vector form, ''system level specifications vector'' (SLSV). Figure 3 illustrates the hierarchy structure of the physical and functional characteristics.

SLS identified in the previous step affects the GFRs in the sense that some specifications may help satisfy some GFRs, while other specifications might prevent the implementation of some desired GFRs. The impact of the SLS on GFR's should be clearly identified which will help in developing products that will meet, up to a satisfactory degree, the GFRs stated earlier. The impact will be determined based on, (-1: Negative Impact, 0: No Impact, $+1$: Positive Impact).

The degree of association between components should be measured and used in grouping components into modules. This can be done by incorporating the GFR weights; in addition to the SLSV and their impacts on the GFRs to provide an associativity measure between components. The general form of the associativity measure is as follows,

SLSV (C1 and C2) SLS and FRs

$$(S)_{1x1} = (1 \quad 0 \quad \cdot \quad \cdot \quad a_n)_{1xn} * \begin{pmatrix} 1 & \cdot & \cdot & \cdot & b_{1,m} \\ 0 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ b_{n,1} & \cdot & \cdot & \cdot & b_{n,m} \end{pmatrix}_{nxm}$$

Weights for FRs

$$* \begin{pmatrix} 1 \\ 0.9 \\ \cdot \\ \cdot \\ c_{mx1} \end{pmatrix}_{mx1}$$

The associativity measures associated with components are arranged in a component vs. component matrix.

3. Genetic algorithms

Many combinatorial optimization problems from the manufacturing systems world are very complex and hard to solve using conventional optimization techniques. As it has been suggested, enumeration techniques are usually obsolete if a problem is extremely big. Since real life problems are in most cases enormous, new techniques have to be developed and used to solve these large and difficult problems. Many algorithms have been proposed since the development of the approximation techniques mentioned in the introduction. These techniques include simulated annealing, tabu search, and GAs. The rapid development of computer science, along with the rapid development of artificial intelligence techniques has created a great interest among engineers in

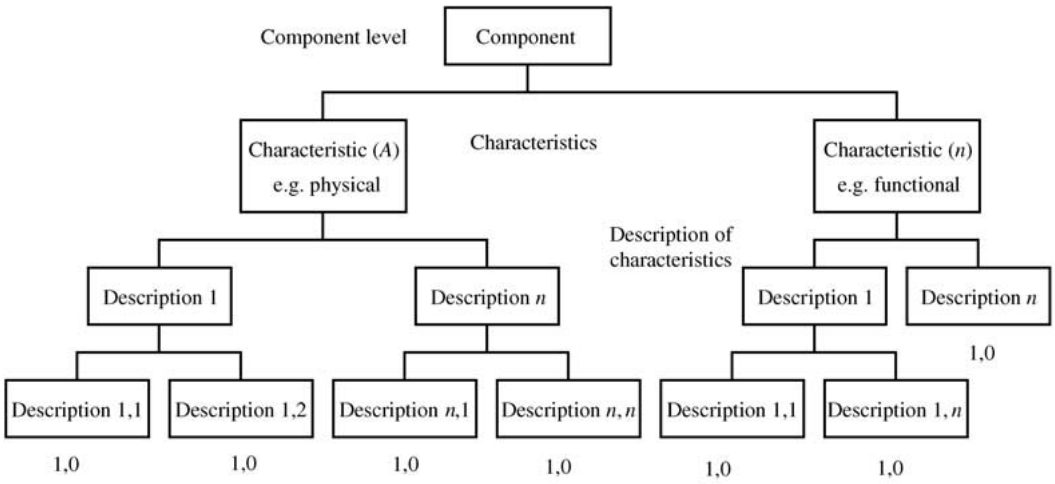

Fig. 3. System level specification decomposition hierarchy.

imitating living beings to solve those kinds of difficult problems. Michalewicz (1992) explained that the simulation of the natural evolutionary process of human beings results in stochastic optimization techniques called evolutionary algorithms. The idea is that by simulating the thought process that the human uses to solve difficult problems, the algorithm will be able to solve the same kind of problems. Moreover, and unlike the normal local search algorithm, evolutionary algorithms (and more specifically GAs) will try to take into account a wider range of possible solutions, which will then in turn reduce the probability of not finding an optimal solution to a given combinatorial problem.

Holland (1975), one of the pioneers in the area of evolutionary algorithms, stated that GAs are stochastic search techniques based on the mechanism of natural selection and natural genetics. They start with a set of random solutions called population. Each individual or solution in the population is called a chromosome. As Goldberg (1989) explained, a chromosome is a string of symbols (usually, a binary bit string), which represents a solution to the problem being solved and discussed. The initial population has to be determined by the user. Each combinatorial optimization problem is different, so special attention has to be given to the definition of a solution. Solutions can be represented as binary bit strings, number strings, word strings, among many other options. However, if a representation does not accurately represent what the solution actually means, the algorithm will perform poorly, and no useful information will be obtained. In essence, the algorithm will search for an optimal solution in an inaccurate solution space.

Chromosomes evolve via successive iterations called generations. During each iteration, the chromosomes are evaluated using a fitness function that will eventually decide if a chromosome passes to the next generation or dies. After an iteration of the algorithm, a new generation is created with new chromosomes called offspring. They are formed by either merging two chromosomes from the current generation using the crossover operation or by modifying a single chromosome using the mutation operation. The new generation is then formed by selecting some of the chromosomes according to their fitness value and by rejecting others that do not qualify as valuable individuals. Fitter chromosomes are more likely to live longer. In essence, they have a higher probability of survival. According to Holland (1975), after a few runs (a few generations) the algorithm should converge to the best chromosome, which hopefully, represents the optimal solution to the problem. However, as many researchers have emphasized, this solution may not be the optimal one. This is due to the fact that the algorithm may be ''lost'' in an area that does not have the best feasible solution and/or the initial solution given by the user does not accurately represent a possible solution to the problem at hand. Figure 4 illustrates the basic process of a GA. In this case, the original population is composed of four individuals, which are divided into two groups, forming two pairs of ''parents.'' Parents experience

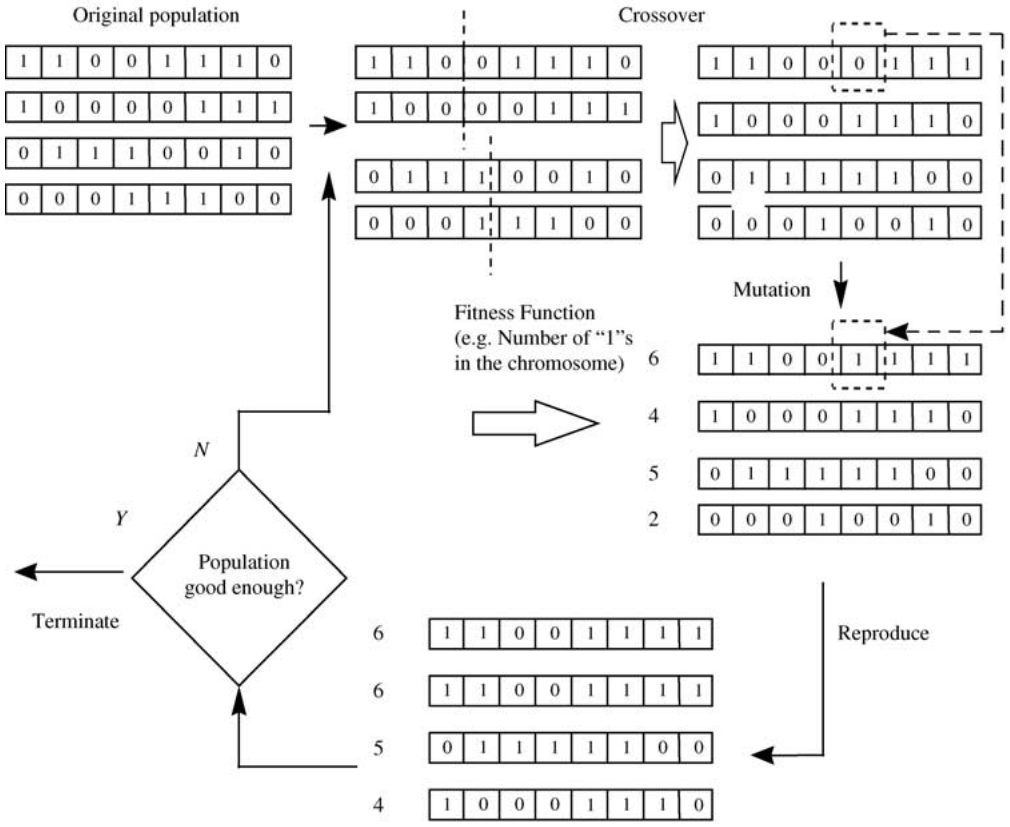

Fig. 4. Basic structure of a GA.

crossover (in this case, single-point crossover which will be explained later) and become new chromosomes (called offsprings). After crossover, one of the chromosomes undergoes mutation (one of its attributes changes from 0 to 1).

When the crossover and mutation operations have done their job, a fitness function is applied to the chromosomes to determine the fitness of each individual. In this case, the number of ones on a chromosome determines the fitness value. The usage of these values determines which chromosomes will survive and which chromosomes will die. In this case, the inferior individual (the one that has a fitness of 2) is eliminated from the population and the superior individual (the one with fitness 6) is duplicated. This new population is then tested. If the population is not good enough, another generation will be created and the process is then repeated until a good population is found (in essence, the creation of new generations mimics the evolutionary process). At this point, an optimal solution is found and no more iterations are needed, so the process is terminated. Holland described the basic structure of a genetic algorithm in 1975. This structure follows the same basic process shown in Fig. 4. The crossover operation used by Holland (1975) was the single-point approach, and the selection probability of chromosomes for reproduction is determined by the chromosome's fitness value. Based on Holland's algorithm, Obitko (1998) summarized the algorithm in his work. The algorithm shown below is highly adaptive, and many combinatorial optimization problems can and have been solved using this basic outline:

Procedure: *Genetic Algorithm*

(1) [*Initialization*] Generate random population of n chromosomes.

(2) [*Evaluation*] Evaluate the fitness $f(x)$ of each chromosome x in the population.

(3) [*New population*] Create a new population by repeating the following steps until the new population is complete:

- [*Selection*] Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected).
- [*Crossover*] Given a crossover probability, crossover the parents to form new offspring. If no crossover is performed, offspring is an exact copy of parents.
- [*Mutation*] With a mutation probability, mutate new offspring at given positions in a chromosome.
- [*Accepting*] Place new offspring in a new population.

(4) [*Replace*] Use new generated population for a further run of algorithm (new generation).

(5) [*Test*] If the end condition is satisfied (usually, required fitness value is obtained), STOP, and return the best solution in current population.

(6) [*Loop*] Otherwise go to step 2 (a new generation will be created).

For combinatorial optimization problems, the evaluation process needs to be carefully defined. The performance of the fitness function is crucial for the algorithm's performance. If the evaluation technique is too general, the algorithm may never reach an optimal solution because the fitness function will not measure the chromosomes properly. On the other hand, if the process is too strict, the algorithm will be unable to find any kind of optimal solution. It will just run forever trying to satisfy some impossible conditions. This may be very costly if one takes into account the cost associated with the use of GAs in combinatorial optimization problems.

The three most important steps in a GA will now be presented. These steps, coupled with the initialization and evaluation of the solutions, give GAs an edge in the solution of hard combinatorial optimization problems. They introduce new information that would not be otherwise searched and inspected by regular optimization methods, a benefit that can prove vital in the use of GAs as plausible optimization techniques.

3.1. *Genetic and evolution operations*

The initialization and evaluation processes are very important in the performance of a GA during the solution of a combinatorial optimization problem. However, the three steps that really measure the algorithm's performance are the crossover and mutation steps, and the selection step. These three steps can be divided into two kinds of operations that are performed within any GA:

(1) Genetic operations: crossover and mutation.
(2) Evolution operation: selection.

The genetic operations mimic the process of heredity of genes to create new offspring at each generation, while the evolution operation mimics the process of Darwinian evolution to create populations from generation to generation. Both operations introduce a wide range of possibilities to the problem being studied. By mixing and mutating existing chromosomes in the solution space, the algorithm explores solutions to the combinatorial problem that would never be explored by a regular optimization technique.

3.1.1. *Crossover*

Crossover may be the most important operation performed by a GA for the solution of a combinatorial optimization problem. It usually operates on two chromosomes at a time and generates offspring by combining them. A cut-point is randomly chosen and the segments are combined to create two new chromosomes. Cheng and Gen (1997) explained that the cut point could be one or many. If the crossover operation follows a one cut-point approach, then both chromosomes are mixed together: the ''head'' of one is paired with the ''tail'' of the other one and so, two new chromosomes are created (check by looking back at Fig. 4). If there is more than one cut point the new chromosome will be an intercalated version of both parents. The single-point crossover approach was first denominated by Holland *et al*. (1986) as the canonical approach. In the traveling salesman problem (TSP), this approach may lead to solutions that do not include some cities, or that may repeat several cities. This is highly undesirable, as these solutions cannot be accepted. Cheng and Gen (1997) embedded a repairing procedure in their algorithm to resolve this problem. However, the canonical approach is blind, a characteristic that may be very costly in terms of the repair procedure and the time it will take for the algorithm to find a solution to the combinatorial optimization problem (and more specifically to the TSP). The second approach, called the heuristic approach is a very problem dependant approach as different heuristics can be used depending on the

combinatorial optimization problem being studied. Many authors however, have used the Heuristic crossover method presented by Grefenstette (1987) for solving the TSP problem. This approach follows the nearest neighbor heuristic for finding the best path from one place to another. Crossover may be the most important feature in a genetic algorithm. The performance of the algorithm is closely related with the performance of the crossover operation. A very critical characteristic of the crossover operation is the crossover rate, which controls the number of chromosomes that undergo the crossover operation. This rate should be carefully defined so that an optimal solution can be found in a reasonable amount of time. A high crossover rate allows a more deep exploration of the solution space and reduces the chances of settling for a false optimum; but if the rate is too high, the result is wastage of computation time in exploring bad regions of the solution space. In the sample shown at the end of the section the crossover rate used is 0.60, meaning that about 60% of the chromosomes experience crossover.

3.1.2. *Mutation*

Mutation is an operation that produces spontaneous random changes in some chromosomes. It serves one of two roles: replacing genes lost from the population during the selection process (so that they can be tried in a new context), or providing the genes that were not present in the initial population. There are two basic ways of doing mutation. They are bit flipping and random assignment. In bit flipping, bits in a chromosome are changed with a certain probability. Random assignment assigns zeros or ones at random within the chromosome disregarding the values that were already there. As for the crossover operation, a mutation rate has to be carefully defined for each problem. This mutation rate is defined as a percentage of the total number of genes in the population. This rate controls the pace at which new genes are introduced into the population for trial. If the rate is too low, many useful genes will never be tried out; but if the rate is too high, there will be much random perturbation and the offspring will start losing their resemblance to the parents. In the experiment performed at the end of the section the mutation rate used is 0.15, meaning that about 15% of the total number of genes will experience mutation.

3.1.3. *Selection*

The principle behind GAs, as it was mentioned earlier, is Darwinian natural selection. Selection deals with the problem of selecting the ''valuable'' individuals or chromosomes that will survive and pass to the next generation. Selection is the driving force in any genetic algorithm, and thus, selection pressure is critical on the performance of a GA in the solution of a combinatorial optimization problem. Michalewicz (1992) stated that usually, low-selection pressure is indicated at the start of the GA search in favor of a wide exploration of the search space, while high-selection pressure is recommended at the end in order to exploit the most promising regions in the search space. The selection procedure may create a new population for the next generation based on all parents and offspring or only on part of them. Selection may replace parent chromosomes by their offspring. Cheng and Gen (1997) explained that when all parents are replaced by their offspring, a *generational replacement* has taken place. This occurrence is not recommended and is highly undesirable, as GAs are blind in nature and so, offspring chromosomes may be worst than their parents. This would lead the GA to some unwanted places when looking for an optimal solution. Over the past few years, some solutions have been suggested to overcome the problem of experiencing a generational replacement during the solution of a combinatorial optimization problem. Holland (1975) gave a first suggestion that involved a random selection of a parent whenever an offspring was born. This chosen parent was then replaced by the offspring that was just created. De Jong (1975) suggested another solution to the generational replacement problem. It is called the crowding strategy, which selects the parent to be replaced depending on its similarities with the just created offspring (the most similar parent dies and gives space for the new offspring). The selection of chromosomes is not completely determined by their fitness values. One advantage that GAs bring to the solution of various combinatorial optimization problems is the fact that they search the solution space more broadly, meaning that apparently ''weak'' solutions are not disregarded right away, as they may lead the algorithm to good final solutions. Holland (1975) proposed the most common method for the selection of chromosomes that pass from generation to generation. It is called the *roulette wheel selection*

method, the most recognized method for selection among stochastic techniques. The *roulette wheel selection* method determines the selection probability for each chromosome by proportionally assigning a portion of the roulette to a chromosome depending on its fitness value. After all probabilities have been found, a roulette with the values is created. The selection process is then started by ''spinning'' the wheel as many times as required until a full population is selected. The probability that a chromosome will be chosen for survival is directly related to its fitness value, as chromosomes with higher fitness values occupy most of the roulette. In essence, chromosomes with high fitness values will be selected more times from this ''biased'' roulette.

4. A heuristic-based GA method

Throughout this work, the need of developing an efficient algorithm that will produce acceptable answers for a wide variety of combinatorial optimization problems has been stressed. In this section, an efficient heuristic GA is developed to solve various instances of the diagnosis problem, more specifically, of the modular design problem. The heuristic algorithm developed in this paper can be used to solve a vast variety of combinatorial optimization problems, as the differences among these problems lie only on the chromosome representation of the answers. The objective of the problem can be different as well, but a few modifications to the algorithm would have to be made. These changes however, would not affect the actual heuristic algorithm. They would only affect the complexity of the developed user interface for the problem. The objective of the problem in this paper is to maximize the sum of all the similarities between components, so that they can be grouped together to form a modular product. The heuristic developed in this paper, as well as many other heuristic GAs (Chu and Tsai, 1998; Venugopal and Narendran, 1992, among many others), try to give good solutions to hard combinatorial optimization problems that are too difficult or impossible to solve with conventional solving methods.

4.1. *Overview of the heuristic GA*

Developing a heuristic GA involves various steps. The proposed algorithm developed for this paper follows a total of 10 steps that can be divided into four basic points:

- *Initialization*:
  1. Define the chromosomes and their representation.
  2. Determine the population size, crossover probability, and mutation probability.
  3. Generate initial population of solutions that satisfy the problem constraints.
- *Reproduction*:
  4. Compute the fitness value of each chromosome.
  5. Calculate the total fitness of the population.
  6. Find the reproduction probability for each chromosome.
  7. Calculate the cumulative reproduction probability for each chromosome.
  8. Choose the best chromosomes for the next generation.
- *Crossover*:
  9. Do crossover on the selected chromosomes based on a crossover probability defined in the initialization procedure.
- *Mutation*:
  10. Perform mutation on the chromosomes based on a mutation probability defined in the initialization procedure.

This list gives a basic outline of the various steps one has to follow in order to develop an efficient heuristic GA. As the objectives and the nature of each problem is different, one has to pay close attention to each of the 11 steps to develop clever heuristics that will enhance the performance and the usability of the algorithm. This work closely studies each step, developing and proposing different methods that improve the usability of the GA developed for this paper.

4.1.1. *The chromosomal representation*

As it is discussed by Chu and Tsai (1998), the first step in the implementation of a heuristic algorithm involves the representation of the problem to be solved with a finite-length string called chromosome. As it has been explained before, each element of the chromosome (each gene) represents a decision variable, feature, or parameter of the problem. The chromosome representation involves two related decisions. The first decision has to do with the problem of finding an effective way of encoding the

problem in terms of a string chromosome. This decision can lead to the determination of the size of each chromosome, which is completely dependent on the nature of the problem. The second decision involves the selection of a string format for each gene in the chromosome. The value of a gene can be either a binary number or an integer. The binary format, which is the genetic one, has been widely used by many researchers. Among them, Karr and Gentry (1993), Cooper and Vidal (1993), and Epsy *et al*. (1992), developed different genetic algorithms for fuzzy control problems using the binary format. Austin (1990) used the same format for neural network applications that solved the XOR problem. Maniezzo (1994) studied the topology and weight distribution of neural networks using binary genes.

The integer format (used in this work) is domain-specific, and has been widely used by many researchers in the areas of facility layout, TSPs, scheduling, among many other problems. Chan and Tansri (1994) studied the crossover operator using integer format genes and chromosomes. Tam (1992) used this format to solve various facility layout problems. Biegel and Davern (1990) analyzed the job-shop scheduling problem using the same kind of genes. For the problem that will be presented in the next section, the integer format is used. Each gene will have an integer value, which will represent the classification of components into groups depending on their similarities. Encoding for this problem is done as follows: the chromosome strings consist of n integer genes, each of which represent a component in the family of components $(1, \ldots, n)$. Each value in a chromosome stands for the group number to which a component is assigned. For example, if there are 10 components to be divided into three groups, then the chromosome can be represented as (1, 1, 3, 2, 1, 2, 2, 3, 3, 3). Here, this chromosome indicates that group 1 has components 1, 2, and 5; group 2 has components 4, 6, and 7; and group 3 has components 3, 8, 9, and 10. The size of the chromosome depends only on the size of the matrix being studied, so if a problem has 20 components, the size of the chromosome will be 20 (20 genes). Also, since the similarities between components can and will be different, the integer value of each gene is allowed to be any number between 1 and the number of components being grouped (n), which will basically mean that any group from 1 to n could be selected in the process.

4.1.2. *Determination of systems parameters*

Some parameters, including population size, maximum number of generations, and the probability of crossover and mutation, should be decided before the algorithm starts to find a solution. These parameters are very sensitive to the computational performance of the algorithm, and although Holland (1975), Goldberg (1989), and Grefenstette (1986) have all proposed some standard values for these parameters, it is usually with extensive experimentation that these parameters are best set. The population size directly affects the computation time of the algorithm, as having a large population size will mean that the algorithm will have to perform more calculations. A default population size of 20 chromosomes is set. It is important to note that a higher population size will give the algorithm a higher chance of success, as the solution space searched will be larger. However, if good solutions are required in a reasonable amount of time, the population size should be smaller. This will allow the algorithm to quickly find a good solution to the problem being studied. The crossover probability is set to be 0.65. This basically means that about 65% of the chromosomes in each generation will experience crossover. If the crossover probability is set to 100%, then all offspring will be created by crossover. If it is 0%, the whole new generation will be made from exact copies of chromosomes from old populations. Crossover is made in hope that new chromosomes will have good parts of old chromosomes and maybe the new chromosomes will be better. However, it is always good to allow some chromosomes to survive without change in the next generation, an occurrence called elitism. The mutation probability is set to be 0.20. This means that about 20% of the genes in a population will experience mutation. If the mutation probability is set to a 100%, all the chromosome will be changed, on the other hand, if it is set to 0%, nothing will change. Mutation occurs to prevent falling into local optima, but it should not occur very often, as the GA will become just a random search method. The number of generations is directly affected by the performance of the algorithm. A maximum number of generations can be set, but the algorithm will usually find a solution to the problem before this number is ever reached. This means that many extra computations will be performed after the algorithm has already found a solution. The heuristic algorithm developed examines

the fitness of each population, and stops whenever the global fitness cannot be greater than what it already is. This means that the algorithm has found an optimal solution, at which point, the solution is given to the user while the algorithm starts another run to investigate the grouping of components with a different number of groups.

4.1.3. *Generation of initial populations*

After the problem is represented, a set of initial solutions (chromosomes) called population has to be determined. This first population will represent the first generation, which will then evolve until a good solution to the problem being studied is found. In order to obtain feasible solutions, each chromosome has to satisfy the problem constraints. Chu and Tsai (1998) proposed the *replacement policy* as a way of making sure that the chromosomes created for the first generation (and for further generations as well) were valid. Validity can be asserted in many ways, but one easy method checks the problem constraints given by the problem with each created chromosome. Chu and Tsai (1998) discussed two different methods that have been widely used and that are part of their replacement policy. The first method is called the variable restriction method, which only chooses the chromosomes that meet the feasible region of constraints. The second method is called the penalty function method, which allows chromosomes to violate the constraints by giving them a penalty, which will then in turn lower their probability of survival to zero.

In this work, a modification of the variable restriction method is utilized. A random number generator creates the initial population by randomly generating numbers between 1 and n (where n is the size of the chromosome). After each chromosome is created, the algorithm checks which groups were created, and with that information, the algorithm checks the validity of the chromosome by comparing it with the specified problem constraints. If the chromosome is valid, no changes are made, but if it violates one of the constraints specified in the problem, the necessary changes are made to ensure the validity of the chromosome. Due to the chromosomal representation chosen for the problem, the only problem that can occur is the assignment of components to groups that have not been created. That problem is easily resolved by creating the groups that are represented within the chromosome. For example, if the groups selected are 2, 5, and 8, and the chromosome is (5, **2**, 8, 8, **2**, 5, 5, **8**, 2, 2), one has to make sure that position 2 in the chromosome has a 2, position 5 in the chromosome has a 5, and position 8 in the chromosome has an 8. In this case, the chromosome would not be valid, and the algorithm would change the fifth position value from a 2 to a 5 to make the chromosome a valid one. This modification of the variable restriction method is used after each population is created, so that only valid chromosomes are studied and manipulated.

4.1.4. *Selection and use of the fitness function*

In order to find better solutions, a fitness function is needed for evaluating and selecting good generations of chromosomes. The fitness function should give domain-specific information about the value of each chromosome. For this reason, it is usually a good idea to define it in the form of a mathematical formulation, either a maximization or a minimization of some parameters and constraints.

In the case of this heuristic GA, the objective function of the p-median model is selected as the fitness function. Each chromosome, created by either the initialization of the GA, or by the creation of a new generation, will have a fitness value. Since the objective of this model is to maximize the sum of the similarities, the fitness values are continuously increasing, until an optimal solution is found. Once the fitness function is defined and used for the first time (after the first generation of chromosomes are created), the algorithm starts the selection and reproduction process (steps 5–8). Using the fitness values of each chromosome, the roulette wheel selection process that was explained in the previous section begins by finding the total fitness of the population (step 5). Then, and using the roulette principles, a reproduction probability is assigned to each chromosome (based on their fitness values), and the roulette is filled using the respective cumulative probabilities of each chromosome (steps 6 and 7). Since the space on the wheel is totally dependent on the fitness value of each chromosome, fitter chromosomes will have a larger space in this bias roulette, increasing their chances of survival. The algorithm then randomly generates a number between 0 and 1, and a chromosome is chosen as a parent based on that random number and the cumulative reproduction probability that was calculated in step 7.

In the previous subsection, a case for keeping the

best chromosomes in the new generation was made. This is called elitism, and the heuristic developed for this paper accounts for that occurrence. First, the best three chromosomes of each population are reproduced and chosen as part of the next generation of chromosomes. However, there is still a chance that either crossover and/or mutation will modify the chromosomes. As it will be explained in the next subsections, the best chromosome will be passed on from generation to generation, as a way of ensuring the algorithm's success. At the end of the reproduction cycle, the new group of chromosomes is ready for crossover and mutation. The new chromosomes are the basis of a new generation; a generation that will be altered by crossover and mutation, and that will be examined after the two genetic operators have done their part in introducing new areas of unvisited solution space to the problem.

4.1.5. *Crossover*

After the better-fitted chromosomes are selected, each pair of chromosomes is selected sequentially to exchange information according to the crossover probability previously determined. A random number is generated, and if the number lies below the crossover probability, crossover is performed. A mode of single-point crossover is used in this heuristic; a mode for which the cut point for doing crossover is randomly selected at each instance. All but one chromosome are subject to crossover. After reproduction gives a new generation of chromosomes, the best chromosome is assured to be a part of the next generation unchanged, as it is always a good idea to keep the best solution within a population. Elitism ensures that the best chromosome will not suffer crossover and/or mutation. The algorithm will also benefit from elitism, as it will have a better chance of finding good solutions to the problem without losing the best solution found so far.

After the crossover cycle is over, and all chromosomes have been exposed to crossover, the new generation is subjected to mutation. A crossover cycle is over whenever each pair of sequential chromosomes have either experienced crossover (because their probability of crossover fell within the 65%), or have been passed without change because their probabilities fell outside the crossover probability. Note that if a pair of chromosomes experience crossover, the next chromosomes to be sequentially selected will not include any of these two chromosomes, as it is not desired to double-crossover a single chromosome.

4.1.6. *Mutation*

After crossover changes the chromosomes in the new population, mutation in each chromosome occurs according to some previously set mutation probability, which in this case is set to 20%. The mutation operator occasionally alters genes within a chromosome by changing the value of a single gene within a chromosome. The heuristic GA in this case alters the genes based on the mutation probability as well as on the groups or modules that have already been selected for each chromosome. This ensures that new chromosomes will be explored, and thus, the solution space will be broader. It is important to note that the changes due to mutation lie within the groups selected for each chromosome. In essence, this means that once a gene is randomly selected for mutation, the change will only occur within the number of groups being used. This ensures that the number of groups within a population will not increase, as this is a highly undesirable occurrence when the algorithm is studying a number of groups. Once all chromosomes are subjected to mutation (except the best one), the new generation of chromosomes is set. The new run will not require an initialization procedure, as the generation that just experienced reproduction, crossover, and mutation, acts as the new generation. However, another mode of mutation is introduced at this point, as the chromosomes of this new generation are subjected to a validity check. An illegal chromosome should never be passed to the next generation (replacement policy), and so, the required changes to each chromosome are made in order to ensure their validity.

The next section uses the developed GA to solve a modular design problem. As it will be shown in the next section, the algorithm is very efficient and performs well in the solution of this problem, giving the user fast and good solutions to the problem at hand.

5. Case study: A classification problem

In order to be able to group components into families a similarity matrix between components has to be defined. Using the similarity matrices developed by Kamrani and Salhieh (2000), the GA will group the

Table 1. Similarity matrix for modular design

	SS1C1	SS1C2	SS1C3	SS1C4	SS2C1	SS2C2	SS2C3	SS3C1	SS3C2	SS3C3	SS3C4	SS3C5	SS3C6	SS4C1	SS4C2	SS4C3	SS4C4
SS1C1		6	2	2	5	2	2	1	1	2	2	2	2	5	2	2	2
SS1C2	6		1	2	2	5	2	2	2	1	1	2	2	2	5	2	3
SS1C3	2	1		1	2	5	2	2	2	1	1	2	2	2	2	5	2
SS1C4	2	2	1		2	2	5	2	2	2	2	4	4	2	2	2	5
SS2C1	5	2	2	2		2	2	4	4	2	2	2	2	5	2	2	2
SS2C2	2	5	5	2	2		2	2	2	4	4	2	2	2	5	5	2
SS2C3	2	2	2	5	2	2		2	2	2	2	4	4	2	2	2	5
SS3C1	1	2	2	2	4	2	2		1	2	2	2	2	1	2	2	2
SS3C2	1	2	2	2	4	2	2	1		2	2	2	2	1	2	2	2
SS3C3	2	1	1	2	2	4	2	2	2		1	2	2	2	1	1	2
SS3C4	2	1	1	2	2	4	2	2	2	1		2	2	2	1	1	2
SS3C5	2	2	2	4	2	2	4	2	2	2	2		1	2	2	2	1
SS3C6	2	2	2	4	2	2	4	2	2	2	2	1		2	2	2	1
SS4C1	5	2	2	2	5	2	2	1	1	2	2	2	2		2	2	2
SS4C2	2	5	2	2	2	5	2	2	2	1	1	2	2	2		1	2
SS4C3	2	2	5	2	2	5	2	2	2	1	1	2	2	2	1		2
SS4C4	2	3	2	5	2	2	5	2	2	2	2	1	1	2	2	2	

components of a speed reducer in order to achieve an optimal modular design. From there, the engineer will have the tools to develop a modular facility for the production of the product being studied, which will in turn, reduce costs and the time it takes to get the product into the market. The similarities between components are calculated by taking into account their functional and physical characteristics. The similarity matrix used in this case is shown in Table 1 (SS stands for Sub-System, and C stands for a Component within the sub-systems. There are 4 sub-systems and a total of 17 components).

The GA developed in the previous section begins by randomly generating the initial population of solutions for the matrix shown. Since the first generation of solutions group all the components into one group, the algorithm just finds the column on the matrix that yields the highest fitness values.

The fitness value for the best chromosome is 48. The grouping for this case, at it was mentioned earlier, does not give the user any valuable information as all the components are grouped into one family. After 8 generations, the algorithm reaches the best solution for this case, a solution with a total fitness (for the generation) of 960, which is the sum of all the fitness values of each chromosome in a generation composed of 20 individuals. Figure 5 shows the improvement made by the GA after every generation. A maximum of 80 generations is allowed, but in this case, the algorithm only needs 8 generations mainly because of the simplicity of the problem being studied (classification of components into one group).

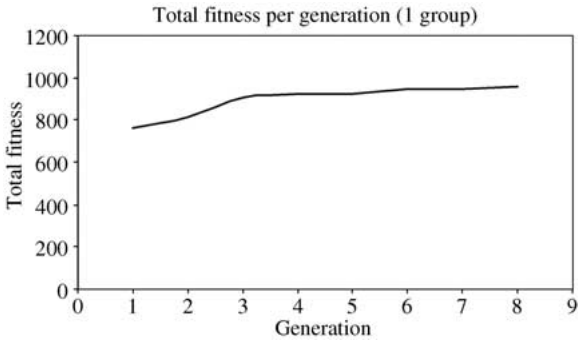

Fig. 5. GA behavior for a one-group solution.

The second run of the algorithm is more interesting since the GA works to group the components into two separate groups given their similarities. At the end of this run the fitness values of each chromosome should be greater than the ones given in the previous run (48 was the best chromosome for the one group run). However, if the groupings do not yield a better fitness value, this will mean that grouping the components into two groups does not give the best overall optimal solution. It is important to note that some groupings, even if they are not the optimal ones, will give companies what they are looking for, as they may be satisfied with good solutions that will improve their operations.

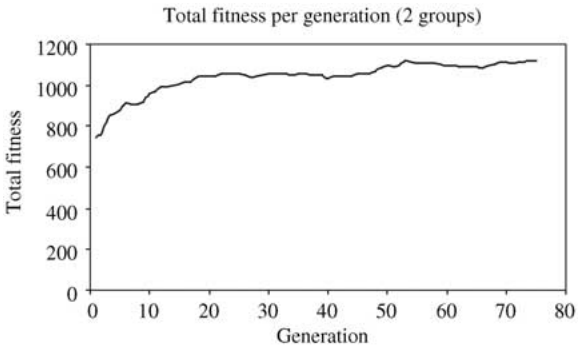

Fig. 6. GA behavior for a two-group solution.

As it is shown on Fig. 6, the genetic algorithm does in fact give a better solution in this case. The fitness function value is 56, and the best grouping is (4, 6, 6, 4, 4, 6, 4, 6, 6, 6, 6, 4, 4, 6, 6, 6, 4). This basically means that for example components 1 and 4 within the first sub-system (SS1C1 and SS1C4) are together in one group (positions one and four are both fours), and components 2 and 3 (within the same sub-system) are together in another group. The rest of the groupings for this solution follow the physical proximity of the components being studied. Figure 6 shows the improvement per generation of the total fitness of the algorithm. Table 2 shows the final groupings on the matrix. A total of 75 generations were needed this time, as the problem got more interesting and complicated.

The solution given by the GA for two groups is a very good solution, but it is not the optimal one, as there are several high similarities left out of the two groups (an occurrence that may imply that the solution is not optimal). One can see obvious faults that cannot be solved with just two groups. It is apparent from the grouping that the addition of at least another group would improve the arrangement of components, as the assignment of the components into two groups can be very restrictive.

The third run of the GA groups the components into a total of three groups. Again, if the fitness of the final arrangement yields a lower number than the one given by the previous runs, one can conclude that a three group arrangement may not be the optimal arrangement one is looking for. However, and as it was mentioned earlier, the physical characteristics of the components being studied implies that at least three groups should be used when arranging the components of the system under study.

This new run of the GA brings a new improvement with respect to the previous runs. In this case, the GA yields a fitness value of 64, the highest one so far, and a total fitness of 1263. As Fig. 7 shows, this run uses up all 80 generations to get the optimal solution for the problem, a solution that will end up being the overall optimal solution for the speed reducer.

The grouping given by the GA is (5, 6, 6, 7, 5, 6, 7, 5, 5, 6, 6, 7, 7, 5, 6, 6, 7). Table 3 shows the actual grouping given by the GA. This grouping of components is the optimal one up to this point, and if one looks at the way these groups were formed, one

Table 2. Grouping for a two-group solution

	SS1C1	SS1C4	SS2C1	SS2C3	SS3C5	SS3C6	SS4C4	SS1C2	SS1C3	SS2C2	SS3C1	SS3C2	SS3C3	SS3C4	SS4C1	SS4C2	SS4C3
SS1C1		2	5	2	2	2	2	6	2	2	1	1	2	2	5	2	2
SS1C4	2		2	5	4	4	5	2	1	2	2	2	2	2	2	2	2
SS2C1	5	2		2	2	2	2	2	2	2	4	4	2	2	5	2	2
SS2C3	2	5	2		4	4	5	2	2	2	2	2	2	2	2	2	2
SS3C5	2	4	2	4		1	1	2	2	2	2	2	2	2	2	2	2
SS3C6	2	4	2	4	1		1	2	2	2	2	2	2	2	2	2	2
SS4C4	2	5	2	5	1	1		3	2	2	2	2	2	2	2	2	2
SS1C2	6	2	2	2	2	2	3		1	5	2	2	1	1	2	5	2
SS1C3	2	1	2	2	2	2	2	1		5	2	2	1	1	2	2	5
SS2C2	2	2	2	2	2	2	2	5	5		2	2	4	4	2	5	5
SS3C1	1	2	4	2	2	2	2	2	2	2		1	2	2	1	2	2
SS3C2	1	2	4	2	2	2	2	2	2	2	1		2	2	1	2	2
SS3C3	2	2	2	2	2	2	2	1	1	4	2	2		1	2	1	1
SS3C4	2	2	2	2	2	2	2	1	1	4	2	2	1		2	1	1
SS4C1	5	2	5	2	2	2	2	2	2	2	1	1	2	2		2	2
SS4C2	2	2	2	2	2	2	2	5	2	5	2	2	1	1	2		1
SS4C3	2	2	2	2	2	2	2	2	5	5	2	2	1	1	2	1	

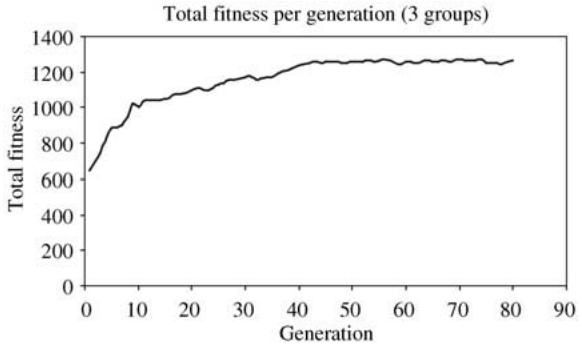

Fig. 7. GA behavior for a three-group solution.

can possibly affirm that this grouping may be the overall optimum one. Although the solution for three groups given by the GA seems to be the optimal one, it is very difficult to assert the optimality of the solution. Moreover, for bigger problems it may be impossible to know if a solution is the optimal one or just a good solution for the problem. In this case, since the problem is not very big, one could try using an optimization program such as LINDO to assert the optimality of the solution given by the GA.

The solution for the three-group classification is in fact, the optimal solution (LINDO gives the same grouping). However, and as it was discussed before, the GA cannot be 100% sure that this is the optimal overall solution mainly because of the blind nature of the procedure. This can be a drawback, but in this case it is actually a benefit, because the algorithm stores the best solution for a three-group arrangement and tries to find better and perhaps more specific solutions in the following runs. A company may want to work with a five-group arrangement for example, even if the arrangement is not the optimal one. Since a program such as LINDO many not be readily available, and the problem may be too large and complicated for regular programs to handle, one has to study and analyze the solutions given by the genetic algorithm. The fourth run of the GA gives the user a new arrangement of the components using four groups. Again, if the solution does not reflect an improvement against the three-group solution, and if the difference is quite large between the fitness values, one can start concluding that in fact, the three-group classification is the optimal one.

The total fitness (as well as the fitness of the chromosomes) of the new run goes down to almost 900, while for the previous run it was over 1200. Moreover, the fitness value of the best chromosome goes down from 64 to just 47, which is actually lower than the fitness value of the first run, in which the algorithm grouped all components into one group. This sudden drop in fitness, along with the poor grouping of the components (1, 1, 6, 1, 5, 6, 5, 5, 5, 6, 6, 15, 5, 5, 15, 6, 15), gives the user a ''preview'' of what is about to happen with the following groupings (five, six, seven, eight, and nine groups) and their fitness values (see Table 4 for the four-group classification).

Table 3. Grouping for a three-group solution

	SS1C1	SS2C1	SS3C1	SS3C2	SS4C1	SS1C2	SS1C3	SS2C2	SS3C3	SS3C4	SS4C2	SS4C3	SS1C4	SS2C3	SS3C5	SS3C6	SS4C4
SS1C1		5	1	1	5	6	2	2	2	2	2	2	2	2	2	2	2
SS2C1	5		4	4	5	2	2	2	2	2	2	2	2	2	2	2	2
SS3C1	1	4		1	1	2	2	2	2	2	2	2	2	2	2	2	2
SS3C2	1	4	1		1	2	2	2	2	2	2	2	2	2	2	2	2
SS4C1	5	5	1	1		2	2	2	2	2	2	2	2	2	2	2	2
SS1C2	6	2	2	2	2		1	5	1	1	5	2	2	2	2	2	3
SS1C3	2	2	2	2	2	1		5	1	1	2	5	1	2	2	2	2
SS2C2	2	2	2	2	2	5	5		4	4	5	5	2	2	2	2	2
SS3C3	2	2	2	2	2	1	1	4		1	1	1	2	2	2	2	2
SS3C4	2	2	2	2	2	1	1	4	1		1	1	2	2	2	2	2
SS4C2	2	2	2	2	2	5	2	5	1	1		1	2	2	2	2	2
SS4C3	2	2	2	2	2	2	5	5	1	1	1		2	2	2	2	2
SS1C4	2	2	2	2	2	2	1	2	2	2	2	2		5	4	4	5
SS2C3	2	2	2	2	2	2	2	2	2	2	2	2	5		4	4	5
SS3C5	2	2	2	2	2	2	2	2	2	2	2	2	4	4		1	1
SS3C6	2	2	2	2	2	2	2	2	2	2	2	2	4	4	1		1
SS4C4	2	2	2	2	3	2	2	2	2	2	2	2	5	5	1	1	

Table 4. Grouping for a four-group solution

	SS1C1	SS1C2	SS1C4	SS1C3	SS2C2	SS3C3	SS3C4	SS4C3	SS2C1	SS2C3	SS3C1	SS3C2	SS3C6	SS4C1	SS3C5	SS4C2	SS4C4
SS1C1		6	2	2	2	2	2	2	5	2	1	1	2	5	2	2	2
SS1C2	6		2	1	5	1	1	2	2	2	2	2	2	2	2	5	3
SS1C4	2	2		1	2	2	2	2	1	5	2	2	4	2	4	2	5
SS1C3	2	1	1		5	1	1	5	2	2	2	2	2	2	2	2	2
SS2C2	2	5	2	5		4	4	5	2	2	2	2	2	2	2	5	2
SS3C3	2	1	2	1	4		1	1	2	2	2	2	2	2	2	1	2
SS3C4	2	1	2	1	4	1		1	2	2	2	2	2	2	2	1	2
SS4C3	2	2	2	5	5	1	1		2	2	2	2	2	2	2	1	2
SS2C1	5	2	1	2	2	2	2	2		2	4	4	2	5	2	2	2
SS2C3	2	2	5	2	2	2	2	2	2		2	2	4	2	4	2	5
SS3C1	1	2	2	2	2	2	2	2	4	2		1	2	1	2	2	2
SS3C2	1	2	2	2	2	2	2	2	4	2	1		2	1	2	2	2
SS3C6	2	2	4	2	2	2	2	2	2	4	2	2		2	1	2	1
SS4C1	5	2	2	2	2	2	2	2	5	2	1	1	2		2	2	2
SS3C5	2	2	4	2	2	2	2	2	2	4	2	2	1	2		2	1
SS4C2	2	5	2	2	5	1	1	1	2	2	2	2	2	2	2		2
SS4C4	2	3	5	2	2	2	2	2	2	5	2	2	1	2	1	2	

The grouping shown on Table 4 is a more specific grouping of the components. However, as it can be seen on the matrix, there are a lot of high similarity values outside the groupings. This implies that the solution is not optimal, although it may give the user a much more specific arrangement of components. Without the help of LINDO, one can see that the best solution for the speed reducer has a total of three groups.

6. Conclusions

This paper presents an introduction to the combinatorial optimization problem, the various methods used to solve these kinds of problems, and a sample traveling salesman problem solved using GAs. An overview on the history and the purpose of combinatorial optimization problems was presented first. These kinds of problems abound on a daily basis throughout the industry, and the techniques readily available to users are becoming more and more obsolete. They either take too long to solve these difficult problems or they cannot solve them at all. GAs present a great alternative for users trying to solve very large and complex problems because they tend to find good solutions avoiding the common problems that for example, local search may experience. These algorithms have shown great potential in various fields, as they are very adaptive, they work well with all types of combinatorial optimization problems, and they do not need a lot of mathematical requirements in order to arrive at a good, reasonable solution. GAs are for the most part very efficient. Their complexity is rather low, so the size of problems that can be solved using GAs is a lot greater than the maximum size of problems that can be solved using other techniques. Greater strides have to be made in order to make GAs completely reliable, as they may be the most viable and logic option to be used in the solution of hard combinatorial optimization problems. The heuristic GA developed in this paper is a very efficient yet simple algorithm for solving combinatorial optimization problems. The algorithm is capable of handling harder and larger problems, as it is very flexible in terms of the size of the populations being used, the crossover and mutation probabilities and their methods. For more complicated problems, the algorithm can be easily modified to include one or more of the crossover and mutation methods discussed in this paper. The solution of complex problems will benefit from the use of these methodologies, as single-point crossover, for example, does not always yield the results one may expect from the crossover operation.

References

Austin, S. (1990) Genetic solutions to XOR problems. *AI Expert*, **5**, 53–57.

Biegel, J. E. and Davern, J. J. (1990) Genetic algorithms and job shop scheduling. *Computers and Industrial Engineering*, **19**, 81–91.

Chan, K. C. and Tansri, H. (1994) A study of genetic crossover operations on the facilities layout problem. *Computers and Industrial Engineering*, **26**, 537–550.

Cheng, R. and Gen, M. (1997) *Genetic Algorithms and Engineering Design*, Wiley-Interscience Publication.

Chu, C. H. and Tsai, C. C. (1998) A heuristic genetic algorithm for manufacturing cell formation. *Group Technology and Cellular Manufacturing: Methodologies and Applications*, 285–310.

Cooper, M. G. and Vidal, J. J. (1993) Genetic design of fuzzy controllers. *Proceedings of the Second International Conference on Fuzzy Theory and Technology.*

De Jong, K. (1975) *An analysis of the behavior of a class of genetic adaptive systems*, Ph.D. thesis, University of Michigan, Ann Arbor.

Epsy, T., Vimbrack, E. and Aldridge, J. (1992) Application of genetic algorithms to tuning fuzzy logic systems. *NASA International Workshop on Neural Networks and Fuzzy Logic*, 237–248.

Goldberg, D. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Massachusetts.

Grefenstette, J. (1986) Optimization of control parameters for genetic algorithm. *IEEE Transactions on Systems, Manufacturing, and Cybernetics*, **16**, 122–128.

Grefenstette, J. (1987) Genetic algorithms for the traveling salesman problem. *Proceedings of the First International Conference on Genetic Algorithms*, Lawrence Erlbaum Associates, New Jersey, pp. 160–168.

Holland, J. H. (1975) *Adaptation in Natural and Artificial Systems*, MIT press, Boston.

Holland, J. H., Smith, D. and Oliver, I. (1986) A study of permutation crossover operators on the traveling salesman problem. *European Journal of Operational Research*, 224–230.

Karr, C. L. and Gentry, E. J. (1993) Fuzzy control of pH using genetic algorithms. *IEEE Transactions on Fuzzy Systems*, **1**, 46–53.

Kamrani, A. and Salhieh, S. (2000) *Product Design for Modularity*, Kluwer Academic Publishers.

Kano, N., Seraku, N., Takahashi, F. and Tsuji, S. (1984) Attractive quality and must-be quality. *Hinshitsu: The Journal of the Japanese Society for Quality Control*, 39–48.

Law, A. G. (1976) Theory of approximation, with applications. *Proceedings of a Conference Conducted by the University of Calgary and the University of Regina*, Alberta, Canada.

Maniezzo, V. (1994) Genetic evolution of the topology and weight distribution of neural networks. *IEEE Transactions on Neural Networks*, **5**, 39–53.

Michalewicz, Z. (1992) *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, Berlin Heidlberg.

Obitko, M. (1998) *Introduction to Genetic Algorithms*, Czech Technical University, URL: http://cs.felk.cvut.cz/ ∼ xobitko/ga/.

Sait, S. and Youssef, H. (1999) *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems*, IEEE Computer Society, Washington.

Salhieh, S. M. (1998) *Decomposition Methodology for Complex Product Development*, University of Michigan-Dearborn.

Salhieh, S. and Kamrani, A. (1999) Macro level product development using design for modularity. *Robotics and Computer Integrated Manufacturing Journal*, **15**, 319–329.

Shillito, L. M. (1994) *Advanced QFD: Linking Technology to Market and Company Needs*, John Wiley & Sons, New York.

Shirly, G. V. (1992) Modular design and the economics of design for manufacturing, in Susman G. (ed.), *Integrating Design and Manufacturing for Competitive Advantage*, Oxford University Press.

Tam, K. Y. (1992) Genetic algorithms, function optimization, and facility layout design. *European Journal of Operations Research*, **63**, 322–346.

Venugopal, V. and Narendran, T. T. (1992) A genetic algorithm approach to the machine component grouping problem with multiple objectives. *Computers and Industrial Engineering*, **22**, 469–480.

Wall, M. (1995) *Introduction to Genetic Algorithms*, Massachusetts Institute of Technology, Boston, MA, URL: http://lancet.mit.edu/ ∼ mbwall/presentations/IntroToGA.