An Evolutionary Approach to Combinatorial Optimization Problems

Sami Khuri
Department of Mathematics & Computer Science
San José State University
One Washington Square
San José, CA 95192-0103, U.S.A.
`khuri@sjsumcs.sjsu.edu`

Thomas Bäck and **Jörg Heitkötter**
Department of Computer Science
University of Dortmund
Systems Analysis Research Group, LSXI
D–44221 Dortmund, Germany
`{baeck,joke}@ls11.informatik.uni-dortmund.de`

Copyright © 1994 ACM Press. All rights reserved.
In the Proceedings of the 22nd ACM Computer Science Conference
Phoenix, Arizona, March 8–10, 1994; pages 66-73.

Abstract: The paper reports on the application of genetic algorithms, probabilistic search algorithms based on the model of organic evolution, to NP-complete combinatorial optimization problems. In particular, the subset sum, maximum cut, and minimum tardy task problems are considered. Except for the fitness function, no problem-specific changes of the genetic algorithm are required in order to achieve results of high quality even for the problem instances of size 100 used in the paper. For constrained problems, such as the subset sum and the minimum tardy task, the constraints are taken into account by incorporating a graded penalty term into the fitness function. Even for large instances of these highly multimodal optimization problems, an iterated application of the genetic algorithm is observed to find the global optimum within a number of runs. As the genetic algorithm samples only a tiny fraction of the search space, these results are quite encouraging.

1 Introduction

About thirty years ago, several researchers independently developed the idea to use algorithms based on the model of organic evolution as an attempt to solve hard optimization and adaptation tasks on a computer. Nowadays, due to their robustness and wide applicability and the availability of powerful, even parallel hardware, the resulting research field of evolutionary computation receives steadily increasing attention from researchers of many disciplines. Evolutionary algorithms such as genetic algorithms, evolutionary programming, and evolution strategies (to mention the most widely known representatives — see [18] for more detailed information on their similarities and differences) are well suitable for scaling up the problem size and the number of search points maintained by the algorithms. The latter is especially true for fine-grained parallel implementations of evolutionary algorithms [8].

In this paper we apply a representative of evolutionary algorithms, the genetic algorithm, to instances of different NP-complete combinatorial optimization problems. Due to their representation scheme for search points, genetic algorithms are the most promising and easily applicable representatives of evolutionary algorithms for the problems discussed here. These are the maximum cut problem that allows no infeasible solutions, and the constrained subset sum and minimum tardy task problems, whose search space includes infeasible regions. Unlike many other approaches that handle similar constrained optimization problems by a knowledge-based restriction of the search space to feasible solutions, our approach uses a penalty function to cope with constraints. As a major advantage of this technique the genetic algorithm remains problem-independent as the evaluation function for search points (the fitness function) is the only problem-specific part of the algorithm. Consequently, a single software package can be applied to a variety of problems. To achieve this, the penalty function has to obey some general principles reported in Section 3.

The outline of the paper is as follows: Section 2 presents a short overview of the basic working principles of genetic algorithms. Section 3 first presents our general principles for constructing the penalty function as part of the fitness function. Following this, the three NP-complete problems subset sum (Section 3.1), maximum cut (Section 3.2), and minimum tardy task (Section 3.3), their fitness functions for the genetic algorithm, and the experimental results are reported. The paper concludes by summarizing our experiences gained from these experiments.

2 Genetic Algorithms

Genetic Algorithms, developed mainly by Holland in the sixties, are direct random search algorithms based on the model of biological evolution (see e.g. [5, 9]). Consequently, the field of Evolutionary Computation, of which genetic algorithms is part, has borrowed much of its terminology from biology. These algorithms rely on the collective learning process within a population of individuals, each of which represents a search point in the space of potential solutions of a given optimization problem. The population evolves towards increasingly better regions of the search space by means of randomized processes of selection, mutation, and recombination. The selection mechanism favors individuals of better objective function value to reproduce more often than worse ones when a new population is formed. Recombination allows for the mixing of parental information when this is passed to their descendants, and mutation introduces innovation into the population. Usually, the initial population is randomly initialized and the evolution process is stopped after a predefined number of iterations. This informal description leads to the rough outline of a genetic algorithm given below (see also [6, 18] for more detailed explanations):

Algorithm *GA* **is**
  $t := 0$;
  initialize $P(t)$;
  evaluate $P(t)$;
  while not *terminate* $P(t)$ **do**
    $t := t + 1$;
    $P(t) :=$ *select* $P(t-1)$;
    recombine $P(t)$;
    mutate $P(t)$;
    evaluate $P(t)$;
  od
end *GA*.

The procedure "evaluate" in this algorithm calculates the objective function value for all members of the argument population $P(t)$, i.e. the evaluation function represents the only problem-specific part of a genetic algorithm.

Genetic algorithms turn out to be of particular interest for the optimization problems discussed in this paper since they have demonstrated their wide and successful applicability in a large number of practical applications. Furthermore, they use a bit string representation of individuals $\vec{x} = (x_1, \ldots, x_n) \in \{0,1\}^n$, which is well applicable for the problems described in Section 3.

The mutation operator works on bit strings by occasionally inverting single bits of individuals. Mutation is a probabilistic operator, and the probability of a single bit mutation p_m is usually very small (e.g., $p_m \approx 0.001$ [10]). Recent theoretical work on the mutation rate setting gives strong evidence for an appropriate choice of $p_m = 1/n$ on many problems [3, 14], and this rule is adopted in what follows for parameterization purposes of genetic algorithms.

The purpose of recombination (crossover) consists in the combination of useful string segments from different individuals to form new, hopefully better performing offspring. A parameter p_c (the crossover rate) of the algorithm indicates the probability per individual to undergo recombination; a typical choice is $p_c = 0.6$ [10] or larger. Recombination works by selecting two parent individuals $\vec{x} = (x_1, \ldots, x_n)$, $\vec{y} = (y_1, \ldots, y_n)$ from the population, choosing a crossover point $\chi \in \{1, \ldots, n-1\}$ at random, and exchanging all bits after the χ^{th} one between both individuals to form two new individuals:

$$\begin{aligned} \vec{x}\,' &= (x_1, \ldots, x_{\chi-1}, x_\chi, y_{\chi+1}, \ldots, y_n) \\ \vec{y}\,' &= (y_1, \ldots, y_{\chi-1}, y_\chi, x_{\chi+1}, \ldots, x_n) \quad . \end{aligned} \tag{1}$$

This one-point crossover served as a standard operator for many years. However, it can be naturally extended to a generalized m-point crossover by sampling m breakpoints and alternately exchanging each second of the resulting segments [10]. In general, a two-point crossover seems to be a good choice, but the number of crossover points may also be driven to extreme by using uniform crossover. This operator decides for each bit of the parent individuals randomly whether the bit is to be exchanged or not [20], therefore causing a strong mixing effect which is sometimes helpful to overcome local optima.

The selection operator directs the search process by preferring better individuals to be copied into the next generation, where they are subject to recombination and mutation. Genetic algorithms rely on a probabilistic selection operator called proportional selection, which uses the relative fitness of individuals $\vec{x}_i$ to serve as selection probabilities: $p_s(\vec{x}_i) = \Phi(\vec{x}_i) / \sum_{j=1}^{\mu} \Phi(\vec{x}_j)$.

Sampling μ individuals (where μ denotes the population size) according to this probability distribution yields the next generation of parents. $\Phi(\vec{x}_i)$ is related to the fitness function $f(\vec{x}_i)$ we use in our work. If the problem under consideration is a maximization one, or if the fitness function can take negative values, then $\Phi(\vec{x}_i)$ is taken to be a linear function of $f(\vec{x}_i)$. This technique known as linear dynamic scaling is commonly used in genetic algorithms (see [5] (pp. 123–124) or [6]).

Whenever no different parameter setting is stated explicitly, all experiments reported here are performed with a standard genetic algorithm parameter setting: Population size $\mu = 50$, one-point crossover, crossover rate $p_c = 0.6$, mutation rate $p_m = 1/n$, proportional selection with linear dynamic scaling. Except for the mutation rate, all parameters are identical to those suggested in the widely used GENESIS software package by Grefenstette (see [7] respectively [4], pp. 374–377). A generalized and more flexible software package called GENEsYs [2] is used to perform the experiments in Section 3.

3 Application Problems

We choose three combinatorial optimization problems from the literature of NP-complete problems. We believe that they represent a broad spectrum of the challenging intractable problems. The three problems are ordered according to their degree of difficulty. Two problem instances of each optimization problem are introduced, and thoroughly studied. Each one of the three problems starts with a brief introductory section, followed by genetic algorithms' representational issues. We explain how the problem is encoded, the fitness function we use, and other specific particularities the problem being studied might have. That is followed by the first problem instance which is in general a challenging exercise for any heuristic, but nevertheless, of moderate size when compared to our second problem instance. While the typical problem size for the first instance is about twenty, the second problem instance comprises of populations with strings of length about one hundred.

In the absence of test problems of significantly large sizes, we proceed by a well known technique (see e.g. [1]), and introduce scalable test problems. These can be scaled up to any desired large size, and more importantly, the optimal solution can be computed. Thus, our experimental results can be compared to the optimum solution.

As expected, significant portions of the search space of some of the problem instances we tackle are infeasible regions. Rather than ignoring the infeasible regions, and concentrating only on feasible ones, we do allow infeasibly bred strings to join the population, but for a certain price. A penalty term incorporated in the fitness function is activated, thus reducing the infeasible string's strength relative to the other strings in the population. If the penalty is too harsh, the search is similar to the one that discards infeasible strings. On the other hand, a very mild penalty will fail to pressure the search towards feasible solutions. In designing fitness functions for problems that do have infeasible strings, we make use of the following two principles:

- The fitness functions use graded penalty functions. Two infeasible strings are not treated equally. The penalty is a function of the distance from feasibility. It has been argued that such penalty functions generally outperform other modes of penalties [15].

- The best infeasible string can never be better than even the weakest feasible string. Thus, our fitness function will always have an offset term to ensure the strict order between feasible and infeasible strings.

We would like to point out that due to our principles, and especially the second one, the infeasible string's lifespan is quite short. It participates in the search, but is in general left out by the selection process for the succeeding generation. Finally, our experiments tend to confirm that the nature of the exact penalty function is not of paramount importance. Extending the search to the exploration of infeasible regions is of greater importance. This result agrees with the recent findings of Smith, et al. [17].

3.1 The Subset Sum Problem

In the subset sum problem we are given a set W of n integers and a large integer C. We are interested in finding a subset S of W such that the sum of the elements in S are closest to, without exceeding, C. The subset sum problem is NP-complete. The partition problem can be polynomially transformed into it [11].

We note that S can be represented by a vector $\vec{x} = (x_1, x_2, \ldots, x_n)$ where $x_i \in \{0, 1\}$. The presence of w_i in S means that $x_i = 1$, while its absence is represented by a value of zero in the i^{th} component of $\vec{x}$.

The following is a formal definition of the subset sum problem in which we make use of Stinson's terminology for combinatorial optimization problems [19].

Problem instance: A set $W = \{w_1, w_2, \ldots, w_n\}$ where the w_i's are positive integers, and a large positive integer C.

Feasible solution: A vector $\vec{x} = (x_1, x_2, \ldots, x_n)$ where $x_i \in \{0, 1\}$, such that:
$\sum_{i=1}^{n} w_i x_i \leq C$ for $j = 1, 2, \ldots, n$

Objective function: A function $P(\vec{x}) = \sum_{i=1}^{n} w_i x_i$, where $\vec{x} = (x_1, x_2, \ldots, x_n)$ is a feasible vector.

Optimal solution: A vector that maximizes the objective function; i.e., a feasible vector that corresponds to the subset with maximum sum.

This problem has many applications where a certain target C has to be reached, but not exceeded. We are thus interested in minimizing the difference, $C - P(\vec{x})$, and the problem can be recast as a minimization one.

The function to be minimized is:

$$f_1(\vec{x}) \quad = \quad s \cdot (C - P(\vec{x})) + (1 - s) \cdot P(\vec{x}) \quad , \qquad (2)$$

where $s = 1$ when $\vec{x}$ is feasible ($C - P(\vec{x}) \geq 0$), and $s = 0$ when $\vec{x}$ is infeasible. This choice of f_1 guarantees that all infeasible solutions yield larger objective function values than feasible ones, and the penalty term decreases linearly as the distance to the feasible region becomes smaller.

We use two test problems of dimensions $n = 100$ and $n = 1000$, denoted by "sus100" and "sus1000", respectively. A series of $N = 100$ experiments are run on each problem. In each problem, every weight, w_i is randomly drawn from $\{1, \ldots, 1000\}$ and the target C is the total weight of a randomly chosen subset S of items. Each item has a probability of 0.5 of being chosen. Thus, by summing up the items in S, we obtain the optimal solutions: $C_{100} = 28087$ for "sus100", and $C_{1000} = 238922$ for "sus1000". We note that from the construction of the problems, we do expect to obtain exact solutions. In other words, if $\vec{x}^*_{100}$ and $\vec{x}^*_{1000}$ denote the optimal bit strings for "sus100" and "sus1000", respectively, then $P(\vec{x}^*_{100}) = C_{100}$, and $P(\vec{x}^*_{1000}) = C_{1000}$.

The next problem we study, "sus1000-2" of dimension $n = 1000$, admits no solution of value C. It is constructed according to the method described in [13] (p. 128). More precisely, the weights are all even numbers, $w_i \in \{2, 4, \ldots, 1000\}$ and C is chosen to be odd, $C = 10^3 \cdot n/4 + 1$.

Our results are summarized in Table 1. The first column for each problem instance gives the different best fitness values subtracted from its corresponding target value C, encountered after $2 \cdot 10^4$ function evaluations. The second column records the number of times each one of these values is attained. The first row in this table shows the globally optimal runs. For example, 95% of the runs on "sus1000" gave the optimum value of $f_1(\vec{x}) = 0$, while the remaining 5% yielded $f_1(\vec{x}) = 1$.

As can be seen from the results, the genetic algorithm performed very well with all problem instances, independently of the existence of solutions of value C. It is worthwhile noting that the computational effort does not seem to be dependent on the problem size n. Recall, that for "sus1000-2", the problem does not admit an objective function value of 250001. The genetic algorithm located a global optimum after $2 \cdot 10^4$ evaluations in roughly the same number of runs, independently of n.

3.2 The Maximum Cut Problem

The maximum cut problem consists in partitioning the set of vertices of a weighted graph into two disjoint subsets such that the sum of the weights of the edges with one endpoint in each subset is maximized. Thus, if $G = (V, E)$ denotes a weighted graph where V is the set of nodes and E the set of edges, then the maximum cut problem consists in partitioning V into two disjoint sets V_0 and V_1 such that the sum of the weights of the edges from E that have one endpoint in V_0 and the other in V_1, is maximized. This problem is NP-complete. The satisfiability problem can be polynomially transformed into it [11].

The following is a formal definition of the maximum cut problem in which we make use of Stinson's terminology for combinatorial optimization problems [19].

Problem instance: A weighted graph $G = (V, E)$. $V = \{1, \ldots, n\}$ is the set of vertices and E the set of edges. w_{ij} represents the weight of edge $\langle i, j \rangle$, i.e., the weight of the edge between vertices i and j. Assume that $w_{ij} = w_{ji}$, and note that $w_{ii} = 0$ for $i = 1, \ldots, n$.

Feasible solution: A set C of edges, the cut-set, containing all the edges that have one endpoint in V_0 and the other in V_1, where $V_0 \cup V_1 = V$, and $V_0 \cap V_1 = \emptyset$. In other words, V_0 and V_1 form a partition of V.

Objective function: The cut-set weight $W = \sum_{\langle i,j \rangle \in C} w_{ij}$, which is the sum of the weights of the edges in C.

Optimal solution: A cut-set that gives the maximum cut-set weight.

sus100		sus1000		sus1000-2	
$C_{100} - f_1(\vec{x})$	N	$C_{1000} - f_1(\vec{x})$	N	$C_{1000} - f_1(\vec{x})$	N
28087	93	238922	95	250000	100
28086	6	238921	5		
28085	1				

Table 1: Overall best results of all experimental runs performed for the subset sum problem instances. The first row contains the globally optimal objective function values subtracted from the target value C.

First we need to encode the problem in such a way that a genetic algorithm can be applied to it. We can use a binary string $(x_1, x_2, \ldots, x_n)$ of length n where each digit corresponds to a vertex. Each string then encodes a partition of the vertices, i.e. a potential solution to the problem. If a digit is 1 then the corresponding vertex is in set V_1, if it is 0 then the corresponding vertex is in set V_0. Each string in $\{0,1\}^n$ represents a partition of the vertices. For instance, if we only have 5 vertices then the string: `10101` corresponds to the partition where vertices 1, 3, and 5 are in V_1 and nodes 2, and 4 are in V_0.

The function to be maximized is:

$$f_2(\vec{x}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w_{ij} \cdot [x_i(1-x_j) + x_j(1-x_i)] \,. \quad (3)$$

Note that w_{ij} contributes to the sum only if nodes i and j are in different partitions.

We begin our experimental work by considering randomly generated graphs of modest sizes. With a sufficiently small number of vertices, the global optimum can be *a priori* computed by complete enumeration. The results obtained from our experiments can then be compared to the optimum solution.

In random graphs, each pair of nodes has a fixed probability of having an edge between them. By varying this probability to produce graphs with different "density" of edges, we construct two kinds of graphs with $n = 20$ vertices each.

A probability of 0.1 is used in the first case to construct a very sparse random graph entitled "cut20-0.1". The random dense graph "cut20-0.9" is generated with a probability of 0.9 of placing an edge between arbitrary pairs of vertices. Both graphs are generated with random weights uniformly chosen in the range $[0, 1]$. The experimental results for these graphs are presented at the end of this section.

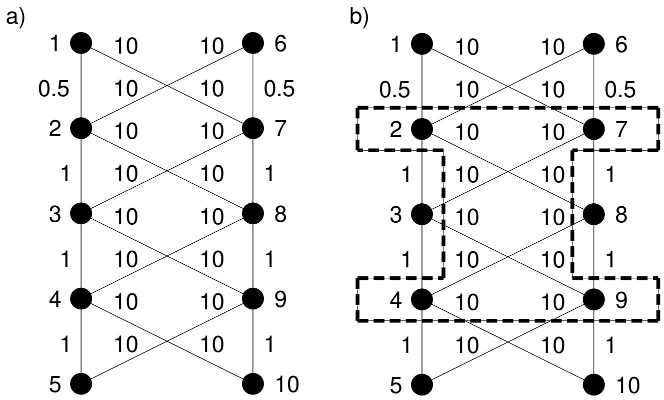

Figure 1: Example of a maximum cut for the graph structure proposed for generating test examples. The problem size is $n = 10$, the maximum cut value is $f^* = 87$.

In order to obtain large test problems for the application of genetic algorithms to instances of the maximum cut problem, we make use of the scalable weighted graph with $n = 10$ nodes shown in Figure 1a). The cut-set that yields the optimal solution can be computed from the construction. At first glance, one might be tempted to consider the bit string `1111100000` (or its complement `0000011111`), which partitions the set of vertices into a subset with five left nodes and another with five right nodes, as being the optimal solution. But this is only a local optimum with a function value of $f_2' = 80$. The dotted line partition of Figure 1b), represented by the bit string `0101010101` (or its complement) with objective function value $f_2^* = 87$, yields the optimum cut-set.

This graph can be scaled up, for any even n, to form arbitrarily large graphs with the same structure and an even number of nodes. The construction of a graph with n nodes consists in adding vertex pairs at the bottom of the graph and connecting them vertically by one edge of weight 1 per vertex and diagonally by one edge of weight 10 per vertex. According to this construction, the optimal partition is described by the $n/2$-fold repetition of the bit pattern `01` (or its complement) and has objective function value $f_2^* = 21 + 11 \cdot (n-4)$ for $n \geq 4$.

As can be seen from the previous paragraph, the optimal string for any graph will consist of repeated bit patterns of `01`$\cdots$. One might be tempted to believe that such regularity in the formulation of the problem instance might favor the workings of genetic algorithms. In order to defuse any doubts, we introduce a preprocessing step which consists in randomly renaming the vertices of the problem instance.

As a consequence, consecutive bit positions no longer correspond to vertices that are close to each other within the graph itself. For the experiments reported here, a graph of size $n = 100$ with randomly renamed vertices is used. This graph is identified in what follows by "cut100".

We perform a total of 100 experimental runs for each of the three test graphs, where the genetic algorithm is stopped after 10^4 function evaluations in case of the small graphs and after $5 \cdot 10^4$ evaluations in case of the large one. All results are summarized in table 2. The first column for each problem instance gives the different best fitness values encountered during the 100 runs. The second column records the number of times each one of these values is attained during the 100 runs. The values given in the first row of the table are the globally optimal solutions.

cut20-0.1		cut20-0.9		cut100	
$f_2(\vec{x})$	N	$f_2(\vec{x})$	N	$f_2(\vec{x})$	N
10.11981	68	56.74007	70	1077	6
9.75907	32	56.12295	1	1055	13
		56.03820	11	1033	30
		55.84381	18	1011	35
				989	12
				967	3
				945	1

Table 2: Overall best results of all experimental runs performed for the maximum cut problem instances. The first row contains the globally optimal objective function values.

A few important observations can be derived from table 2. First, we notice that the genetic algorithm is not affected by the "density" of the graphs with 20 nodes. Second, in both cases, the dense and the sparse, the global optimum is found by more than two thirds of the runs. These are good results especially when we realize that the genetic algorithm performs 10^4 function evaluations, and that the search space is of size 2^{20}. Thus, the genetic algorithm searches only about one percent of the search space.

Third, the same is true for the 100 node problem. The global optimum is found in six of the runs. The average objective function value of the remaining runs is $\bar{f}_2 = 1022.66$, which is about 5% from the global optimum. Only $(5 \cdot 10^4)/2^{100} \cdot 100\% \approx 4 \cdot 10^{-24}$ % of the search space is explored.

3.3 The Minimum Tardy Task Problem

The minimum tardy task problem is a task scheduling problem. Each task i from the set of tasks $T = \{1, 2, \ldots, n\}$ has a length l_i, the time it takes for its execution, a deadline d_i before which the task must be scheduled and its execution completed, and a weight w_i. The weight is a penalty that has to be added to the objective function in the event the task remains unscheduled. The lengths, weights, and deadlines of tasks are all positive integers. Scheduling the tasks of a subset S of T consists in finding the starting time of each task in S, such that at most one task at a time is performed and such that each task finishes before its deadline. The following is a formal definition of the minimum tardy task problem [19].

Problem instance:

$$\begin{array}{lcccclcl} \text{Tasks:} & 1 & 2 & \ldots & n & , & i & > 0 \\ \text{Lengths:} & l_1 & l_2 & \ldots & l_n & , & l_i & > 0 \\ \text{Deadlines:} & d_1 & d_2 & \ldots & d_n & , & d_i & > 0 \\ \text{Weights:} & w_1 & w_2 & \ldots & w_n & , & w_i & > 0 \end{array}$$

Feasible solution: A one-to-one scheduling function g defined on $S \subseteq T$, $g : S \rightarrow \mathbb{Z}^+ \cup \{0\}$ that satisfies the following conditions for all $i, j \in S$:

(1) If $g(i) < g(j)$ then $g(i) + l_i \leq g(j)$ which insures that a task is not scheduled before the completion of an earlier scheduled one.

(2) $g(i) + l_i \leq d_i$ which ensures that a task is completed within its deadline.

Objective function: The tardy task weight $W = \sum_{i \in T-S} w_i$, which is the sum of the weights of unscheduled tasks.

Optimal solution: The schedule S with the minimum tardy task weight W.

This task sequencing problem is also known as the maximum profit schedule problem where the weights are profits, and rather than summing the weights (profits) of the tardy tasks, i.e., the unscheduled ones, it is required to look for the set of scheduled tasks that yields the maximum weight (profit). In other words, the objective function is defined to be: $W = \sum_{i \in S} w_i$. Whether the minimum tardy task problem or its maximization version is considered, the problem remains NP-complete. The partitioning problem can be polynomially transformed into it [11].

Checking a subset S of T for feasibility is not as time consuming as one might think at first glance. To see whether the k tasks in S can be feasibly scheduled, one does not have to check all possible $k!$ different orderings. It can be shown that S is feasible if and only if the tasks in S can be scheduled in increasing order by deadline without violating any deadline [19]. If the tasks are not in that order, one needs to perform a polynomially executable preprocessing step in which the tasks are ordered in increasing order of deadlines, and renamed such that $d_1 \le d_2 \le \cdots \le d_n$. Thus, without loss of generality, we will assume that the tasks are ordered in increasing order of deadlines.

As was the case with the subset sum problem, S can be represented by a vector $\vec{x} = (x_1, x_2, \ldots, x_n)$ where $x_i \in \{0, 1\}$. The presence of task i in S means that $x_i = 1$, while its absence is represented by a value of zero in the i^{th} component of $\vec{x}$.

Example: Consider the following problem instance of the minimum tardy task problem:

Tasks:	1	2	3	4	5	6	7	8
Lengths:	2	4	1	7	4	3	5	2
Deadlines:	3	5	6	8	10	15	16	20
Weights:	15	20	16	19	10	25	17	18

Note that the deadlines are ordered increasingly.

a) $S = \{1, 3, 5, 6\}$, represented by `10101100`, is a feasible solution. The schedule is given by: $g(1) = 0$, $g(3) = 2$, $g(5) = 3$ and $g(6) = 7$. The objective function value amounts to: $\sum_{i \in T-S} w_i = w_2 + w_4 + w_7 + w_8$.

b) $S' = \{2, 3, 4, 6, 8\}$, given by `01110101`, is infeasible. We define $g(2) = 0$, and task 2 finishes at time $0 + l_2 = 4$ which is within its deadline $d_2 = 5$. We schedule tasks 3 at 4, i.e. $g(3) = 4$, which finishes at $g(3) + l_3 = 5$ which is within its deadline $d_3 = 6$. But task 4 cannot be scheduled since $g(4) + l_4 = 5 + 7 = 12$ and will thus finish after its deadline $d_4 = 8$. $\square$

Once more, the fitness function, unlike the objective function, will allow infeasible strings but with a certain penalty. We would also like to differentiate between the infeasible string of part b) of the example, $s =$ `01110101`, and other infeasible strings with the same prefix as s, such as $t =$ `01110000` for instance.

We believe that t is more infeasible than s, and should therefore carry a higher penalty. Thus, our fitness function does not stop as soon as it realizes that task 4 is unschedulable. It continues checking the string beyond task 4 to see if there are any other tasks that could have been scheduled. Thus, if we bypass task 4, we notice that tasks 6 and 8 could have been scheduled, $g(6) = 5$ and $g(8) = 8$. The fitness value of $s =$ `01110101` should then include the terms $w_1 + w_4 + w_5 + w_7$, while that of $t =$ `01110000` should have $w_1 + w_4 + w_5 + w_6 + w_7 + w_8$ among its terms. We also want our function to adhere to the principles stated earlier. No infeasible string should have a better fitness value than a feasible one. We thus make $\sum_{i=1}^{n} w_i$, the offset term, as part of the penalty term for all encountered infeasible strings.

Putting all of the above observations into one function, we obtain the following fitness function to be minimized for the tardy task problem:

$$\begin{aligned} f_3(\vec{x}) &= \sum_{i=1}^{n} w_i \cdot (1 - x_i) + (1 - s) \cdot \sum_{i=1}^{n} w_i \\ &+ \sum_{i=1}^{n} w_i x_i \cdot \mathbf{1}_{R^+} \left(l_i + \sum_{\substack{j=1 \\ x_j \text{ schedulable}}}^{i-1} l_j x_j - d_i \right) \quad . \end{aligned} \tag{4}$$

The third term keeps checking the string to see if a task could have been scheduled, as explained earlier. It makes use of the indicator function:

$$\mathbf{1}_A(t) = \begin{cases} 1 & \text{if } t \in A \\ 0 & \text{otherwise} \end{cases} \quad .$$

Also note that $s = 1$ when $\vec{x}$ is feasible, and $s = 0$ when $\vec{x}$ is infeasible.

Once again, we use a small problem of size $n = 20$, and a large one of size $n = 100$. For the former, we just extend the problem instance given in the example, from 8 to 20 tasks, and randomly allocate task lengths, deadlines

Tasks:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Lengths:	2	4	1	7	4	3	5	2	4	7	2	9	8	6	1	4	9	7	8	2
Deadlines:	3	5	6	8	10	15	16	20	25	29	30	36	49	59	80	81	89	97	100	105
Weights:	15	20	16	19	10	25	17	18	21	17	31	2	26	42	50	19	17	21	22	13

Table 3: Minimum tardy task problem instance of size $n = 20$.

(in increasing order), and weights. This problem, which is completely specified in table 3, is denoted by "mttp20" in what follows.

As was the case with the maximum cut problem, we construct a problem instance of very small size, $n = 5$, that can be scalable to form large size problems.

Tasks:	1	2	3	4	5
Lengths:	3	6	9	12	15
Deadlines:	5	10	15	20	25
Weights:	60	40	7	3	50

This problem instance can be used as a foundation to construct problem instances for any arbitrarily large number of tasks n where $n = 5t$ $(t \geq 1)$. We now describe how to construct a minimum tardy task problem instance of size $n = 5t$ from the 5-task model. The first five tasks of the large problem are identical to the 5-model problem instance. The length l_j, deadline d_j, and weight w_j of the j^{th} task, for $j = 1, 2, \ldots, n$, is given by: $l_j = l_i$, $d_j = d_i + 24 \cdot m$ and

$$w_j = \begin{cases} w_i & \text{if } j \equiv 3 \bmod 5 \text{ or } j \equiv 4 \bmod 5 \\ (m+1) \cdot w_i & \text{otherwise} \end{cases},$$

where $j \equiv i \bmod 5$ for $i = 1, 2, 3, 4, 5$ and $m = \lfloor j/5 \rfloor$. The tardy task weight for the globally optimal solution of this problem is $2 \cdot n$.

According to this description, a problem of size $n = 100$ ("mttp100") is generated and used for our experiments. Again, a total of 100 experiments is performed for both test problems. The number of function evaluations permitted for a single run is 10^4 for the small problem and $2 \cdot 10^5$ for the larger one (about $1.6 \cdot 10^{-23}$ % of the search space). Table 4 reports all results in the same way as in the previous sections.

The global optimum is located within two third of the runs even for the large problem instance. A second accumulation of results is observed for the local optimum of quality 329, which differs from the global optimum by five bits located at positions widely spread over the bit string. Similarly, even for the twenty task problem the local optimum of quality 46 differs from the global one by a Hamming distance of three. Again, these findings confirm the strong potential of genetic algorithms to yield a globally optimal solution with high probability in reasonable time even in case of hard multimodal optimization tasks when a number of independent runs is performed.

mttp20		mttp100	
$f_3(\vec{x})$	N	$f_3(\vec{x})$	N
41	79	200	67
46	19	243	7
51	2	329	24
		452	1
		465	1

Table 4: Overall best results of all experimental runs performed for the minimum tardy task problem instances. The first row contains the globally optimal objective function values.

4 Conclusions

In this work, we have explored the application of a special kind of evolutionary computation, genetic algorithms, to three combinatorial optimization problems. These algorithms, used as heuristics, performed very well on the three NP-complete problems. It is our belief that further investigation into these evolutionary algorithms will demonstrate their applicability to a wider range of NP-complete problems. The technology behind the parallelization of these techniques is growing at a very rapid pace. Massively parallel implementation of evolutionary algorithms provide an excellent means of scalability, especially with respect to population size. We believe that they will provide more adequate tools for tackling extremely large problems including applications that stem from the real world.

Acknowledgments

The first author is very grateful to Aïda Batarekh for all the valuable discussions they had on genetic algorithms and NP-hard problems. The second author gratefully acknowledges support by the DFG (Deutsche Forschungsgemeinschaft), grant Schw 361/5-2.

References

[1] D. H. Ackley. *A Connectionist Machine for Genetic Hillclimbing*. Kluwer Academic Publishers, Boston, 1987.

[2] Th. Bäck. GENEsYs 1.0. Software distribution and installation notes, Systems Analysis Research Group, LSXI, University of Dortmund, Department of Computer Science, Dortmund, Germany, July 1992. (Available via anonymous ftp to `lumpi. informatik.uni-dortmund.de` as file GENEsYs-1.0.tar.Z in /pub/GA/src).

[3] Th. Bäck. The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm. In Männer and Manderick [12], pages 85–94.

[4] L. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

[5] D. E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison Wesley, Reading, MA, 1989.

[6] J. J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, SMC–16(1):122–128, 1986.

[7] J. J. Grefenstette. *A User's Guide to GENESIS*. Navy Center for Applied Research in Artificial Intelligence, Washington, D. C., 1987.

[8] F. Hoffmeister. Scalable parallelism by evolutionary algorithms. In M. Grauer and D. B. Pressmar, editors, *Parallel Computing and Mathematical Optimization*, volume 367 of *Lecture Notes in Economics and Mathematical Systems*, pages 177–198. Springer, Berlin, 1991.

[9] J. H. Holland. *Adaptation in natural and artificial systems*. The University of Michigan Press, Ann Arbor, MI, 1975.

[10] K. A. De Jong. *An analysis of the behaviour of a class of genetic adaptive systems*. PhD thesis, University of Michigan, 1975. Diss. Abstr. Int. 36(10), 5140B, University Microfilms No. 76–9381.

[11] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computation*, pages 85–103. Plenum, New York, 1972.

[12] R. Männer and B. Manderick, editors. *Parallel Problem Solving from Nature 2*. Elsevier, Amsterdam, 1992.

[13] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Chichester, West Sussex, England, 1990.

[14] H. Mühlenbein. How genetic algorithms really work: I. mutation and hillclimbing. In Männer and Manderick [12], pages 15–25.

[15] J. T. Richardson, M. R. Palmer, G. Liepins, and M. Hilliard. Some guidelines for genetic algorithms with penalty functions. In Schaffer [16], pages 191–197.

[16] J. D. Schaffer, editor. *Proceedings of the 3rd International Conference on Genetic Algorithms and Their Applications*. Morgan Kaufmann Publishers, San Mateo, CA, 1989.

[17] A. E. Smith and D. M. Tate. Genetic Optimization using a Penalty Function. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 499–503. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[18] W. M. Spears, K. A. De Jong, T. Bäck, D. B. Fogel, and H. de Garis. An overview of evolutionary computation. In P. B. Brazdil, editor, *Machine Learning: ECML-93*, volume 667 of *Lecture Notes in Artificial Intelligence*, pages 442–459. Springer, Berlin, 1993.

[19] D. R. Stinson. *An Introduction to the Design and Analysis of Algorithms*. The Charles Babbage Research Center, Winnipeg, Manitoba, Canada, 2nd edition, 1987.

[20] G. Syswerda. Uniform crossover in genetic algorithms. In Schaffer [16], pages 2–9.