Applying Evolutionary Algorithms to Combinatorial Optimization Problems

Enrique Alba Torres[1] and Sami Khuri[2]

[1] Universidad de Málaga, Complejo Tecnológico,
Campus de Teatinos, 29071 Málaga, Spain.
eat@lcc.uma.es
WWW home page: http://polaris.lcc.uma.es/~ eat/
[2] Department of Mathematics & Computer Science, San José State University,
One Washington Square, San José, CA 95192-0103, U.S.A.
khuri@cs.sjsu.edu
WWW home page: http://www.mathcs.sjsu.edu/faculty/khuri

Abstract. The paper describes the comparison of three evolutionary algorithms for solving combinatorial optimization problems. In particular, a generational, a steady-state and a cellular genetic algorithm were applied to the maximum cut problem, the error correcting code design problem, and the minimum tardy task problem. The results obtained in this work are better than the ones previously reported in the literature in all cases except for one problem instance. The high quality results were achieved although no problem-specific changes of the evolutionary algorithms were made other than in the fitness function. The constraints for the minimum tardy task problem were taken into account by incorporating a graded penalty term into the fitness function. The generational and steady-state algorithms yielded very good results although they sampled only a tiny fraction of the search space.

1 Introduction

In many areas, such as graph theory, scheduling and coding theory, there are several problems for which computationally tractable solutions have not been found or have shown to be non-existent [12]. The polynomial time algorithms take a large amount of time to be of practical use. In the past few years, several researchers used algorithms based on the model of organic evolution as an attempt to solve hard optimization and adaptation problems. Due to their representation scheme for search points, Genetic Algorithms (GA) are the most promising and easily applicable representatives of evolutionary algorithms for the problems discussed in this work.

The goal of this work is two-fold. First, a performance comparison of three evolutionary algorithms for solving combinatorial optimization problems is made. These algorithms are the generational genetic algorithm (genGA), the steady-state genetic algorithm (ssGA) [13], and the cellular genetic algorithm (cGA) [8]. Second, the paper reports the improvement achieved on already known results for similar problem instances. We compare the results of our experiments to those of [2] and [6].

The outline of the paper is as follows: Section 2 presents a short overview of the basic working principles of genetic algorithms. Section 3 presents the maximum cut problem, the error correcting code design problem, and the minimum tardy task problem. The problem's encoding, the fitness function, and other specific particularities of the problem are explained. The experimental results for each problem instance are described in Section 4. The paper summarizes our findings in Section 5.

2 The Evolutionary Algorithms

Our genGA, like most GA described in the literature, is generational. At each generation, the new population consists entirely of offspring formed by parents in the previous generation (although some of these offspring may be identical to their parents). In steady-state selection [13], only a few individuals are replaced in each generation. With ssGA, the least fit individual is replaced by the offspring resulting from crossover and mutation of the fittest individuals. The cGA implemented in this work is an extension of [10]. Its population is structured in a toroidal 2D grid and the neighborhood defined on it always contains 5 strings: the one under consideration and its north, east, west, and south neighboring strings. The grid is a 7×7 square. Fitness proportional selection is used in the neighborhood along with the one–point crossover operator. The latter yields only one child: the one having the larger portion of the best parent. The reader is referred to [1] for more details on panmictic and structured genetic algorithms.

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. We would like to point out that 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.

3 Combinatorial Optimization Problems

In this paper, we apply three evolutionary algorithms to instances of different NP-complete combinatorial optimization problems. These are the maximum cut problem the error correcting code design problem, and the minimum tardy task problem. These problems represent a broad spectrum of the challenging intractable problems in the areas of graph theory [9], coding theory [7], and scheduling [4]. All three problems were chosen because of their practical use and the existence of some preliminary work in applying genetic algorithms to solve them [2], [3], and [6].

The experiments for graph and scheduling problems are performed with different instances. The first problem instance is of moderate size, but nevertheless, is a challenging exercise for any heuristic. While the typical problem size for the first instance is about twenty, the subsequent problem instances comprise of populations with strings of length one hundred and two hundred, respectively. In the absence of test problems of significantly large sizes, we proceed by introducing scalable test problems that can be scaled up to any desired large size, and more importantly, the optimal solution can be computed. This allows us to compare our results to the optimum solution, as well as to the existing best solution (using genetic algorithms). As for the error correcting code design problem, we confine the study to a single complex problem instance.

3.1 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 since the satisfiability problem can be polynomially transformed into it [5].

We use a binary string $(x_1, x_2, \ldots, x_n)$ of length n where each digit corresponds to a vertex. Each string encodes a partition of the vertices. 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. The function to be maximized is:

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

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

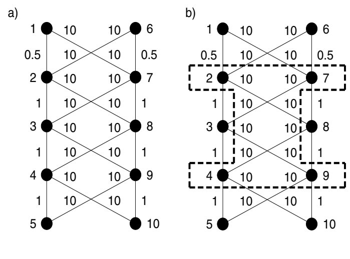

Fig. 1. Example of a maximum cut for the graph structure proposed for generating test examples. The problem size is $n = 10$, the maximum cut value is $f^* = 87$.

In this work, we consider the randomly generated sparse graph "cut20-0.1" and the randomly generated dense graph "cut20-0.9" found in [6]. In order to obtain larger problem instances, 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. The dotted line partition of Figure 1b is represented by the bit string `0101001010` (or its complement) with objective function value $f^* = 87$ and yields the optimum cut-set.

This graph can be scaled up, for any even value of 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 easily described by the concatenation of a copy of the $n/4$-fold repetition of the bit pattern `01`, followed by a 0, then another copy of the $n/4$-fold repetition of the bit pattern `01`, and finally a 0. Alternatively, one could take the complement of the described string. The string has objective function value $f^* = 21 + 11 \cdot (n - 4)$ for $n \geq 4$. 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 $n = 100$, "cut100", is used.

3.2 The Error Correcting Code Design Problem

The error correcting code design problem (ECC) consists of assigning codewords to an alphabet that minimizes the length of transmitted messages and that provides maximal correction of single uncorrelated bit errors, when the messages are transmitted over noisy channels. Note that the two restrictions are conflicting in nature. On one hand, we would like to assign codewords that are as short as possible, and on the other hand, good error correction is achieved by adding redundant bits so as to maximize the Hamming distance between every pair of codewords.

This study considers binary linear block codes. Such codes can be formally represented by a three-tuple (n, M, d), where n is the length (number of bits) of each codeword, M is the number of codewords and d is the minimum Hamming distance between any pair of codewords. An optimal code consists in constructing M binary codewords, each of length n, such that d, the minimum Hamming distance between each codeword and all other codewords, is maximized. In other words, a good (n, M, d) code has a small value of n (reflecting smaller redundancy and faster transmission), a large value for M (denoting a larger vocabulary) and a large value for d (reflecting greater tolerance to noise and error). As n increases, the search space of possible codes grows exponentially.

Linear block codes can either be polynomially generated, such as the Bose, Chaudhuri, and Hocquenghem (BCH) codes [7], or non-polynomially generated, by using some heuristic. Genetic algorithms can be used to design such codes [3]. Other researchers have used hybrids (e.g., simulated annealing and genetic algorithms) and parallel algorithms to achieve good codes [2].

In this study, we consider a problem instance that was tackled by [2], where $n = 12$ and $M = 24$, and use their fitness function with all three evolutionary algorithms. However, we do not parallelize our genetic algorithms, as is the case in their work. The function to be minimized is:

$$f(C) = \frac{1}{\sum_{i=1}^{M} \sum_{j=1; i \neq j}^{M} \frac{1}{d_{ij}^2}} \tag{2}$$

where d_{ij} represents the Hamming distance between codewords i and j in the code C (of n codewords, each of length M).

Note that for a code where $n = 12$ and $M = 24$, the search space is of size $\binom{4096}{24}$, which is approximately 10^{87}. It can be shown that the optimum solution for $n = 12$ and $M = 24$ has a fitness value of 0.0674. The optimum solution is illustrated in [2].

3.3 The Minimum Tardy Task Problem

The minimum tardy task problem is a task-scheduling problem. It is NP-complete since the partitioning problem can be polynomially transformed into it [5]. The following is a formal definition of the minimum tardy task problem [12]:

Problem instance:

Tasks:	1	2	...	n	,	i	$>$	0
Lengths:	l_1	l_2	...	l_n	,	l_i	$>$	0
Deadlines:	d_1	d_2	...	d_n	,	d_i	$>$	0
Weights:	w_1	w_2	...	w_n	,	w_i	$>$	0

Feasible solution: A one-to-one scheduling function g defined on $S \subseteq T$, $g : S \longrightarrow Z^+ \cup \{0\}$ 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.

A subset S of T is feasible if and only if the tasks in S can be scheduled in increasing order by deadline without violating any deadline [12]. 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 $d_1 \leq d_2 \leq \cdots \leq d_n$.

A schedule S can be represented by a vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ 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 $\boldsymbol{x}$. We use the fitness function described in [6] which allows infeasible strings and uses a graded penalty term.

For our experiments, we use three problem instances: “mttp20” (of size 20), “mttp100” (of size 100) and “mttp200” (of size 200). The first problem instance can be found in [6]. The second and third problem instances were generated by using a scalable problem instance introduced in [6].

4 Experimental Runs

We performed a total of 100 experimental runs for each of the problem instances. Whenever no 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 $p_c = 0.6$, bit-flip mutation, mutation rate $p_m = 1/n$ (where n is the string length), and proportional selection. All three algorithms were run on a uniprocessor machine. These were the settings used with the same problem instances reported in [2] and [6].

What follows is the convention used to present the results of the experimental runs. The first column for each evolutionary algorithm gives the best fitness value encountered during the 100 runs. The second column for each evolutionary algorithm 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 average number of iterations it took to obtain the maximum value. The first value recorded under $f(x)$ is the globally optimal solution. For example, Table 1 reports that genGA obtained the global optimal (whose value is 10.11981) ninety two times out of the 100 runs. The table also indicates that the optimum value was obtained after 2782.1 iterations when averaged over the 100 runs.

4.1 Results for the Maximum Cut Problem

Table 1. Overall best results of all experimental runs performed for “cut20-01”.

ssGA		genGA		cGA	
avg = 626.4		avg = 2782.1		avg = 7499	
$f(x)$	N	$f(x)$	N	$f(x)$	N
10.11981	79	10.11981	92	10.11981	16
9.76	21	10.05	1	10.0	24
		10.0	1	9.765	53
		9.89	1	9.76	7
		9.76	5		

We notice that genGA performs better than ssGA for the sparse graph (see Table 1), while ssGA gives better results for the dense graph (see Table 2). Due to the very small population size ($n = 50$), in which neighborhoods cannot develop properly, the cGA did not produce results as good as the two other algorithms. In other words, we believe that with such a small population size, cGA is still mainly in the exploration stage rather than the exploitation stage. As for "cut100", all three algorithms were unable to find the global optimum. When compared to [6], our ssGA and genGA performed better for the sparse and dense graphs. As for "cut100", our algorithms were not able to improve on the results of [6].

Overall, these are good results especially when we realize that the evolutionary algorithms explore only about 1% of the search space.

Table 2. Overall best results of all experimental runs performed for "cut20-0.9".

ssGA		genGA		cGA	
avg = 2007.1		avg = 4798		avg = 7274	
$f(x)$	N	$f(x)$	N	$f(x)$	N
56.74007	72	56.74007	50	56.74007	2
56.04	10	56.73	19	56.5	12
55.84	16	56.12	12	55.5	59
55.75	2	56.04	9	54.5	24
		55	10	53.5	3

Table 3. Overall best results of all experimental runs performed for "cut100".

ssGA		genGA		cGA	
$f(x)$	N	$f(x)$	N	$f(x)$	N
1077	0	1077	0	1077	0
1055	9	1055	0	1055	0
1033	19	1033	8	1033	4
1011	36	1011	9	1011	11
989	22	989	14	989	8
967	7	967	9	967	4
≤ 945	7	≤ 945	60	≤ 945	73

4.2 Results for the ECC Problem

For the ECC problem, ssGA outperformed both genGA and cGA. For this problem instance, cGA produced results comparable to those of ssGA. But as can be seen from the average values in Table 4, ssGA is substantially faster than cGA. Our algorithms performed better than the one reported in [2]. We believe that our algorithms outperformed theirs mainly because we used symmetric encoding, where once a string is processed, we assume that its complement too has been taken care of, thus producing substantial time savings.

4.3 Results for the Minimum Tardy Task Problem

We notice that while ssGA outperforms the two other heuristics for the 20-task problem instance, genGA gives much better results for the 100-task and 200-task problems. For "mttp20", the local optimum of 46 differs from the global one by a Hamming distance of three. Compared to the results of [6] for "mttp20", ssGA performs much better, genGA is comparable, while cGA's performance is worse.

With "mttp100", the global optimum (200) is obtained with the unique string composed of 20 concatenations of the string b=`11001`. The second best solution of 243, is obtained by the strings that have `11000` as prefix (with tasks three, four and five contributing a total of 60 units towards the fitness value). This prefix is followed by the substring `11101` (contributing 3 units towards the fitness value) and 18 copies of b=`11001` (each contributing 10 units towards the fitness value). Since there are 19 ways of placing `11101` among the 18 substrings `11001`, there are 19 strings of quality 243 $(60 + 3 + (18 \times 10))$. A second accumulation of results is observed for the local optimum of 329, which is obtained by the schedule represented by the string with prefix: `001001110111101`. The string is then completed by concatenating 17 copies of b=`11001`. This string too is unique. Compared to the results reported in [6] for "mttp100", both ssGA and genGA significantly outperform them. Once more, cGA lags behind.

For "mttp200", genGA is a clear winner among the three evolutionary algorithms. This problem instance was not attempted by [6].

Table 4. Overall best results of all experimental runs performed for the ECC problem instance.

ssGA		genGA		cGA	
avg = 7808		avg = 35204		avg = 30367	
$f(\boldsymbol{x})$	N	$f(\boldsymbol{x})$	N	$f(\boldsymbol{x})$	N
0.067	40	0.067	22	0.067	37
0.066	0	0.066	11	0.066	1
0.065	17	0.065	18	0.065	21
0.064	25	0.064	33	0.064	27
0.063	13	0.063	16	0.063	13
0.062	5	0.062	0	0.062	1

Table 5. Overall best results of all experimental runs performed for "mttp20".

ssGA		genGA		cGA	
avg = 871.4		avg = 2174.7		avg = 7064.2	
$f(\boldsymbol{x})$	N	$f(\boldsymbol{x})$	N	$f(\boldsymbol{x})$	N
41	86	41	73	41	23
46	10	46	11	46	7
51	4	49	8	49	9
		51	3	51	9
		56	1	53	6
		57	1	54	1
		61	1	56	12
		65	2	≥ 57	33

Table 6. Overall best results of all experimental runs performed for "mttp100".

ssGA		genGA		cGA	
avg = 43442		avg = 45426		avg = 15390	
$f(\boldsymbol{x})$	N	$f(\boldsymbol{x})$	N	$f(\boldsymbol{x})$	N
200	78	200	98	200	18
243	4	243	2	243	18
326	1			276	2
329	17			293	6
				316	1
				326	1
				329	37
				379	9
				$\geq$ 429	8

Table 7. Overall best results of all experimental runs performed for "mttp200".

ssGA		genGA		cGA	
avg = 288261.7		avg = 83812.2		avg = 282507.3	
$f(\boldsymbol{x})$	N	$f(\boldsymbol{x})$	N	$f(\boldsymbol{x})$	N
400	18	400	82	400	6
443	8	443	9	443	7
476	2	476	2	493	1
493	1	493	2	529	34
516	1	496	1	543	1
529	42	529	3	579	10
579	3	629	1	602	1
602	1			629	8
665	23			665	17
715	1			$\geq$ 679	15

5 Conclusion

This work explored the applications of three evolutionary algorithms to combinatorial optimization problems. The algorithms were the generational, the steady-state and the cellular genetic algorithm. The primary reason behind embarking on the comparison testing reported in this work was to see if it is possible to predict the kind of problems to which a certain evolutionary algorithm is or is not well suited. Two algorithms, ssGA and genGA, performed very well with the maximum cut problem, the error correcting code design problem and the tardy task scheduling problem. The third algorithm, namely cGA, usually lagged behind the other two. For all problem instances except one, genGA and ssGA outperformed previously reported results.

Overall, our findings confirm the strong potential of evolutionary 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.

We subscribe to the belief that one should move away from the reliance on individual problems only in comparing the performance of evolutionary algorithms [11]. We believe researchers should instead create test problem generators in which random problems with certain characteristics can be generated automatically and methodically. Example characteristics include multimodality, epistasis, the degree of deception, and problem size. With this alternative method, it is often easier to draw general conclusions about the behavior of an evolutionary algorithm since problems are randomly created within a certain class. Consequently, the strengths and weaknesses of the algorithms can be tied to specific problem characteristics.

It is our belief that further investigation into these evolutionary algorithms will demonstrate their applicability to a wider range of NP-complete problems.

References

1. E. Alba and J. M. Troya. A survey of parallel distributed genetic algorithms *Complexity* pages 31–52, vol. 4, number 4, 1999.
2. H. Chen, N. S. Flann, and D. W. Watson. Parallel genetic simulated annealing: a massively parallel SIMD algorithm. *IEEE transactions on parallel and distributed systems*, pages 805–811, vol. 9, number 2, February 1998.
3. K. Dontas and K. De Jong. Discovery of maximal distance codes using genetic algorithms. *Proceedings of the Tools for Artificial Intelligence Conference*, pages 805–811, Reston, VA, 1990.
4. P. Brucker, Scheduling Algorithms, Springer-Verlag, 2nd edition, 1998.
5. 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.
6. S. Khuri, T. Bäck, and J. Heitkötter. An evolutionary approach to combinatorial optimization problems. *Proceedings of the 22nd Annual ACM Computer Science Conference*, pages 66–73, ACM Press, NY, 1994.
7. S. Lin and D. J. Costello, Jr. Error Control Coding: Fundamentals and Applications', Prentice Hall, 1989.
8. B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms, *Proceedings of the 3rd ICGA*, pages 428–433, Morgan Kaufmann, 1989.
9. C. H. Papadimitriou, Computational Complexity, Addison Wesley, 1994.
10. J. Sarma and K. De Jong. An analysis of the effect of the neighborhood size and shape on local selection algorithms. *Lecture Notes in Computer Science*, vol. 1141, pages 236–244, Springer-Verlag, Heidelberg, 1996.
11. W. Spears. Workshop on test problems generators. *Proceedings of the International Conference on Genetic Algorithms*, Michigan, July 1997.
12. D. R. Stinson. *An Introduction to the Design and Analysis of Algorithms.* The Charles Babbage Research Center, Winnipeg, Manitoba, Canada, 2nd edition, 1987.
13. G. Syswerda, A Study of Reproduction in Generational and Steady-State Genetic Algorithms. *Proceedings of FOGA*, pages 94–101, Morgan Kaufmann, 1991.