Optimized Crossover for the Independent Set Problem

by
Charu C. Aggarwal
James B. Orlin
Ray P. Tai

WP #3787-95 January 1995

Optimized Crossover for the Independent Set Problem

Charu C. Aggarwal
Operations Research Center
Massachusetts Institute of Technology
Cambridge, MA 02139

James B. Orlin
Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA 02139

Ray P. Tai
EECS Department
Massachusetts Institute of Technology
Cambridge, MA 02139

January 15, 1995

Abstract

We propose a knowledge-based crossover mechanism for genetic algorithms that exploits the structure of the solution rather than its coding. More generally, we suggest broad guidelines for constructing the knowledge-based crossover mechanisms. This technique uses an optimized crossover mechanism, in which the one of the two children is constructed in such a way so as to have the best objective function value from the feasible set of children, while the other is constructed so as to maintain the diversity of the search space. We implement our approach on a classical combinatorial problem, called the independent set problem. The resulting genetic algorithm dominates all other genetic algorithms for the problem, and yields one of the best heuristics for the independent set problem in terms of robustness and time performance.

Keywords: Independent Set, genetic algorithms, crossover.

1 Introduction

Genetic algorithms were designed by Holland [14] for solving parameter optimization problems. The premise of genetic algorithms is that one can use principles from the process of evolution in order to design solution methods for many problems. In this context, evolution may be conceptualized as a continuously occurring optimization process involving biological species.

Genetic Algorithms were designed by exploiting the metaphor of evolution. Generation after generation, the individual species compete with each other to survive (Darwinian selection). Fitter members are more likely to mate with each other in subsequent generations. This mating leads to recombinations of the genetic materials of the fit members and often leads to a sequence of generations which are successively more fit. In addition, nature infrequently throws in unexpected variants in the form of mutations, resulting in a greater amount of diversity among the population members.

To transform the evolutionary metaphor into a way of developing heuristics, the essential idea is to treat each solution for a problem as an "individual", whose fitness is typically either the corresponding objective function value or a closely related function. The solutions for a problem are represented or *coded* as strings. The string representing an individual is called a *chromosome.* For the purpose of this paper, we shall assume that chromosomes are always in the form of bit-strings (that is, 0-1 vectors). The current collection of solutions is called the population. At each major iteration, the population is replaced by another population referred to as the subsequent generation. In order to move from one generation to the next, Holland proposed the following operators:

(1) **Selection:** This operator is motivated by the evolutionary mechanism implied by the well-known phrase "survival of the fittest". In the context of the genetic algorithm, it means that greater representation is given to the fitter strings in the current population of solutions. For the purpose of this paper, we shall use the simple fitness proportional rule, which stipulates that the expected number of copies of each individual in the next population is proportional to his fitness. More specifically, if $f(.)$ be the fitness function, and N be the number of population members, then the selection process is performed with the help of a roulette wheel having N slots. Each slot corresponds to a population member, and the width of a slot is proportional to its fitness. There are N fixed pointers uniformly spaced around the slotted wheel. The wheel is spun once, and positions marking these N pointers dictate the new population.

(2) **Crossover:** This operator is motivated by the biological process of meiosis. Suppose that x and y are parent chromosomes. We say that z is a potential child of x and y, if $z_i = x_i$ or $z_i = y_i$ for $i \in \{1, 2, \ldots n\}$. We let $CrossSpace(x, y)$ denote the set of potential children of x and y. In the genetic algorithm, a crossover is typically some well defined random finction $f(x, y)$, such that $f(x, y) \in CrossSpace(x, y)$. Holland [14] suggests choosing $z \in CrossSpace(x, y)$ as follows:

$$z_i = \begin{cases} x_i & \text{for } i = 1 \ldots k \\ y_i & \text{for } i = k+1 \ldots n \end{cases} \tag{1}$$

This kind of crossover is called a *single point crossover.*

(3) **Mutation:** This process is motivated by the evolutionary mechanism of mutation, where a chromosome undergoes a small but important modification. In genetic algorithms, mutations may be viewed as a selection from a neighborhood, as is common in simulated annealing and tabu search. (see, for example, [22]) For each solution x, there is an associated neighborhood $N(x)$. A mutation of x is a selection of some solution y from $N(x)$, possibly uniformly at

random, and possibly using some other approach. It is common (but not necessary) in genetic algorithms to have $N(x)$ consist of all vectors y such that y differs from x in exactly one bit.

Thus, the basic genetic algorithm may be summarized as follows:

```
Algorithm GA;
begin
  Initialize Population;
  Generation:=0;
  repeat
    Generation= Generation + 1;
    Selection(Population);
    Crossover(Population);
    Mutate(Population);
  until Termination_Criterion;
end
```

A primary contribution of Holland's paper and of the subsequent voluminous literature on Genetic Algorithms is the crossover operation. Most of the current methods of crossover determine a child of x and y by selecting $z \in CrossSpace(x, y)$, using a stochastic approach and *without reference to the objective function.* (We note that the same is true in standard implementations of simulated annealing, in which the objective function is used in the process of deciding whether to accept or reject the proposed neighbor of the current solution.) Some of the widely used crossover operations are 1-point crossover, 2-point crossover, and multipoint crossover. ([8]). There is also a literature on objective functions that are called "GA-deceptive" ([16], [17]) for which the above three crossover mechanisms are often quite ineffective. Here we propose a crossover mechanism which takes into account the objective function in a straightforward way. In particular, we refer to an optimized crossover of x and y as the selection of the most fit child in *CrossSpace(x,y).* We apply optimized crossover to the special case in which one is solving the independent set problem. Even though the independent set problem is NP-hard, finding an optimized crossover is polynomially solvable, and the resulting GA is computationally efficient, and obtains high quality solutions.

Since, an important goal of the crossover mechanism is to promote the recombination of solutions so as to encourage the rebuilding of fitter solutions, a natural technique would be to design the crossover specifically for the problem concerned, so as to encourage efficient interchange.

We now introduce the independent set problem. Suppose that $G = (N, A)$ is an undirected graph with a set N consisting of n nodes and a set A consisting of m arcs. An *independent set* is any set I of nodes, no two of which are adjacent; that is no arc $(i, j) \in A$ has both end points in I. The *maximum* independent set problem is that of finding a set of independent nodes with the largest cardinality. Finding such a set is NP-hard. (See, for example, Garey and Johnson [10]) A *maximal* independent set is a subset I of nodes such that $I \cup \{j\}$ is not independent for any $j \notin I$. Finding such a set is quite easy using a simple greedy heuristic. Obviously, a maximal independent set need not necessarily be maximum.

The maximum independent set problem is closely related to the maximum clique problem. A clique is exactly the reverse of an independent set, in the sense that for every pair of nodes i, j in the clique, $(i, j) \in A$. Thus, the maximum independent set problem and the maximum clique problem are equivalent in the sense that an algorithm for one problem can be used for the other,

by applying the algorithm on the complement[1] of the input graph. In the following discussion, we shall provide some background results for the independent set problem. Many of the algorithms are actually for the maximum clique problem, but since these can also be used for the independent set problem, we shall cite those algorithms as well. It may be noted that the independent problem is also closely related to the arc covering[2] and set packing[3] problems.

A number of algorithms have been proposed for the independent set problem. Many of these use partially enumerative techniques or branch and bound methods. Most of these require a worst-case complexity which is exponential, but in practice, they may do well for many classes of graphs. Some of these algorithms may be found in [4], [6], [7], [11], [12], [15], [18], [20], and [24], and a good survey of all algorithms related to this problem may be found in [21]. Several papers provide good heuristics for the problem, which are typically faster than exact methods, and are often relatively robust in terms of time requirements on different instances. Gibbons et.al. [13] developed a continuous heuristic (called the continious based heuristic) for the maximum clique problem, which is computationally competitive with most known heuristic procedures. Balas and Niehaus [5] have implemented a heuristic similar to our genetic algorithm which is very good in terms of the quality of solution, but is very time intensive.

Back and Khuri developed a traditional genetic algorithm for the independent set problem that used a "graded penalty function method. " Any set of nodes constituted a feasible solution, and the fitness function penalized the number of pairs of adjacent nodes in the set. We provide some limited comparisons of our approach to this algorithm in Section 5. However, Back and Khuri did not test their algorithm on the DIMACS benchmark examples, which are the primary examples used in our testing. As a result we do not provide a detailed comparison of their method to ours. In addition, we have developed a genetic algorithm based on "random keys" developed by Bean ([3]). We describe our random keys implementation and provide a detailed comparision of that approach to our optimized crossover approach in Section 4.

This paper is organized as follows. In the next section, we describe the optimized crossover in more detail. In Section 3, we describe how to implement our method for the particular case of the independent set problem. In Section 4, we present computational results. In the computational section, we compare our algorithm to three of the best known ones on the independent set problem. In Section 5, we present a brief conclusion and summary.

2 Optimized Crossover : General Principles

For parents x and y, an *optimum child* z is one that maximizes $\{f(z) : z \in CrossSpace(x, y)\}$. We refer to such a child as an O-child of parents x and y. We note that the selection of an O-child parallels Glover's suggestion [19] in tabu search to select the neighbor which has the highest objective function.

As is common with genetic algorithms, we replace the two parents in a crossover operation

[1]The complement of a graph $G = (N, A)$ is the graph $\overline{G} = (N, C - A)$, where C represents the set of arcs in a complete graph with node set N.

[2]An *arc cover* is a set of nodes V, which are such that every arc in the network has at least one end point in V. If V is a minimum arc cover in $G = (N, A)$ then $N - V$ is the maximum independent set.

[3]A set packing problem is defined over a number of sets $S_1, S_2, \ldots S_n$ each of which have elements drawn from a set of elements $E = \{1, \ldots n\}$. A set-packing P is a collection of sets $\{S_{i_1}, \ldots S_{i_k}\}$, such that no two sets have an element in common. A set packing problem may be represented as an independent set problem on a graph having a node corresponding to each set, and an arc joining two nodes if and only if the corresponding sets are not disjoint. A one to one correspondence exists between independent sets in this graphs and the possible set packings. In fact, the nodes corresponding to the independent set in the graph form a packing.

with two children. The firest child is called tge O-child, and the second is called the E-child. (or exploratory child). Suppose that x and y are binary n-vectors, and z is the O-child. We define the E-child w, such that $w_i = x_i + y_i - z_i$.

Equivalently, if $z_i = x_i$, then $w_i = y_i$, and if $z_i = y_i$, then $w_i = x_i$. In the case that $x_i = y_i$, we refer to x_i and y_i as the same allele. In case $x_i \neq y_i$, we refer to x_i and y_i as conflicting alleles. It follows that when x_i and y_i are conflicting alleles, then so are w_i and z_i. And when x_i and y_i are the same allele, then both the O-child and the E-child inherit the same allele.

The purpose of the E-child is to maintain the diversity of the population. Although the concept of diversity is one that does not have a counterpart in either tabu search or simulated annealing, it is widely discussed within the GA literature.

In the case that the problem to be solved is NP-hard, it is often the case that finding an O-child is also NP-hard. In particular, if x is the vector of all ones, and y is the vector of all zeros, then the O-child is the optimum for the entire population. Nevertheless, in such a case, it may be possible to generate a nearly optimum child in a small amount of time.

In the case that an E-child is infeasible, it may be possible to develop a feasible E-child using some repair mechanism. Briefly, a repair mechanism is a procedure by which one may transform an infeasible solution into a feasible one. In cases where a repair mechanism is not readily available, one could select one of the two parents to be the E-child.

3 Optimized Crossover for the Independent Set Problem

A key aspect of any genetic algorithm is the way in which the solutions are encoded. A natural way of coding the problem would be to represent each solution by a bit string of size n, where the bit in position i is a 0 or a 1, according as whether or not node i is or is not in the independent set.

Let us now see how the O-child and the E-child are generated. The idea of using the information from two independent sets in an optimum way so as to generate another independent set was conceived by Balas and Niehaus [5]. In the context of the independent set problem, generation of an O-child for the sets I_1 and I_2 means that we need to find the largest independent subset of $I_1 \cup I_2$. One can generate an O-Child, using the bipartite matching algorithm in the following well-known manner.

(1) **Step 1:** Let I_1 and I_2 be the two parent independent sets. Assume without loss of generality that $\mid I_1 \mid \leq \mid I_2 \mid$. Construct the subgraph of G as restricted to the nodes in I_1 and I_2. Further, this graph is undirected like the original graph G. All arcs have one endpoint in I_1 and the other in I_2. This subgraph is bipartite, because both the sets I_1 and I_2 are independent; hence $N_1 = I_1$ and $N_2 = I_2 - I_1$ forms one possible bipartite partition of the nodes.

(2) **Step 2:** Find a maximum matching $\mathbf{M}$ in this network. (See, for example, the book by Ahuja et. al. [1] for several efficient matching algorithms.)

(3) **Step 3:** Construct the (directed) auxiliary network $G' = (N_1 \cup N_2, A')$, which is exactly the same as the network defined in Step 1, except that the arcs are directed as well. For each arc (i, j) in the original network, such that $(i, j) \in \mathbf{M}$, let the orientation of the corresponding arc in the auxiliary network be such that the head end of the arc lies in N_1 and the tail end in N_2. For each arc $(i, j) \notin \mathbf{M}$, define its orientation such that the head end of the arc lies in N_2, and the tail end lies in N_1.

(4) **Step 4:** Starting at each unmatched node in N_1, run a depth-first search algorithm, and label all the visited nodes. Let L_1 and L_2 be the set of labeled nodes in N_1 and N_2 respectively. Then, the desired O-child is the set $L_1 \cup N_2 - L_2$.

The proof of correctness of the method is described in the paper by Balas and Niehaus [5]. They developed this method of optimized crossover (they referred to it more simply as a merging operation) for the independent set problem. Although the optimized crossover can be viewed from a genetic algorithm's viewpoint, Balas and Niehaus did not adopt this viewpoint and did not rely on other mechanisms such as selection and mutation that are traditional to genetic algorithms. It is an interesting empirical question as to whether these other features of genetic algorithms lead to an improved performance over the primary focus on the crossover operation. We answer this question in part in Section 5, in which relative computational results are presented. Providing a comprehensive answer is difficult in part because Balas and Niehaus focused primarily on the quality of the solution rather than the speed of performance. We focused much more on the empirical computational performance. As a result, our approach is, in general, 100 to 1000 times faster than the CLIQMERGE approach of Balas and Niehaus.

Once the O-child has been generated, the E-child is obtained by taking all the nodes in $I_1 \cup I_2$ which do not lie in the independent set corresponding to O-Child , and deleting nodes in a greedy manner until the resulting set is independent. Alternatively, it is also possible to use the parent which is least similar to O-child as the E-child.

Mutations were performed on the population in the usual way, by flipping a coin for each bit in the population. In case the coin flip was a success, the corresponding bits were flipped; otherwise not. The probability of success on such flips was defined as the mutation rate. Unfortunately, this sometimes results in strings that are infeasible. To regain feasibility, we applied the following repair algorithm. We delete some randomly chosen nodes from the solution until the resulting set was independent and then to added as many nodes as possible without violating the independence constraint.

The initial population was chosen heuristically in the following manner. For each member in the population a random permutation of the numbers $\{1, \ldots, n\}$ was generated. (n is the number of nodes in the network.) Then the nodes were examined greedily in that order, and added to the current set, if the independence condition was not violated. Thus, each element in the initial population is a *maximal* independent set. Moreover each O-child and E-child is a maximal independent set. The genetic algorithm applies the selection, crossover and mutation operators on this population iteratively.

4 Computational Results

We performed computational tests on the independent set problem by using a standard test bed of independent set (max clique) problems available from the second DIMACS implementation challenge. By using this widely available set of problem instances we could directly compare the computational results of our algorithm to that of many other researchers. We conducted an initial phase of testing in order to set the parameters associated with the genetic algorithm. In general, larger population sizes lead to improved solution quality and increased running times. After substantial testing, we settled on a population size that is equal to 25% of the number of nodes of the input graph. In determining the mutation rate, we found that the performance of the genetic algorithm steadily increased as the mutation rate was increased from .001 to .004. (see Figure 1.) However, the running time for mutation rates around .004 were increased substantially over those

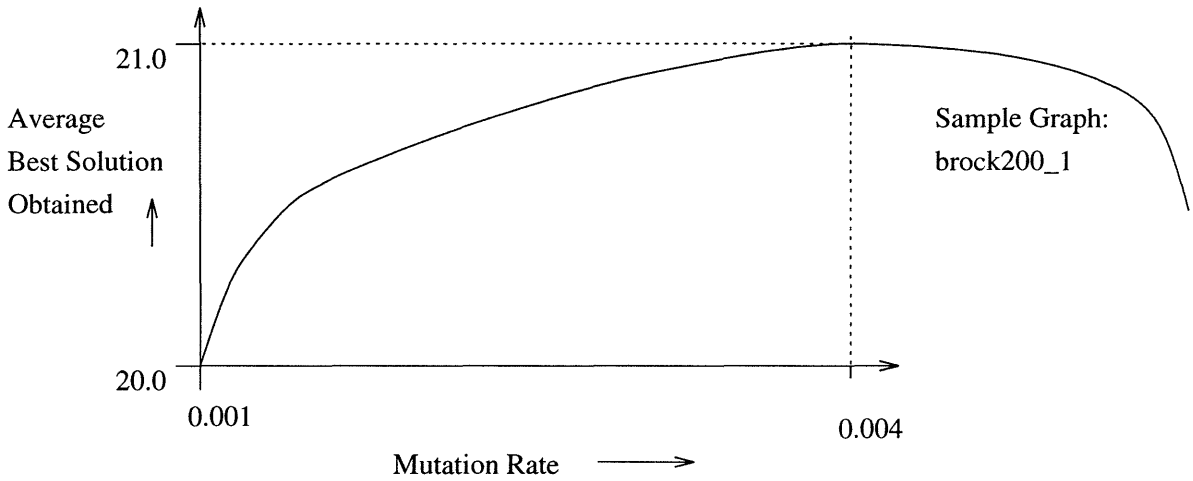

Figure 1: Dependence of the best solution obtained on the mutation rate.

of smaller mutation rates both because of a slow down of the convergence of the genetic algorithm and also because of the increased time to perform the mutations. As a result, we selected the more modest mutation rate of .002 for subsequent testing. Finally, we selected an elitist strategy, which means that the best solution in each generation was saved for the next.

In translating from the maximum clique problems into maximum independent set problems, one needs to take complements of each graph. Complementation creates dense graphs from sparse graphs and sometimes creates sparse graphs from dense graphs; since the input sizes are not fully comparable, the running times are not fully comparable; however, it should be noted that most of the DIMACS problems had the property that both the graph and its complement were dense, and in these cases the problem sizes are comparable.

We refer to our algorithm as OCH, for Optimized Crossover Heuristic. We ran OCH twice on each problem instance, and selected the better of the two solutions. In general, for a fixed total CPU time, this approach worked more effectively than either increasing the size of the population or increasing the number of generations.

All tests were performed on the DECstation running X windows, and the user time was measured for the execution of each graph. The code was written in C, and the compiler used was the Ultrix (version 4.2A) operating system compiler. Since the procedure for loading the graph converts from the maximum clique problem to the maximum independent set problem, the read times tend to be high. The time used for reading has not been included in the reporting times for the heuristic. Further, in order to compare our heuristic with that of other algorithms we needed to use a common standard. We chose this standard to be the DIMACS challenge machine. We ran a heuristic called dfmax on our machine, whose computational results on the SGI challenge machine were available from the DIMACS directories.[4] We ran the tests for a few instances in order to compare the relative speeds of our DEC machines, and the SGI challenge machines. We estimated that our machine was about 3.21 times slower than the SGI challenge machine on the average. In our presentation of the computational results, we shall compare our heuristic with those of all others in terms of SGI challenge times. While these translations to SGI challenge times are imperfect, it is a reasonably good first order approximation.[5]

[4]The heuristic also is available from the DIMACS DIRECTORIES.

[5]We note that 3.21 is actually the *average* slowdown on the various instances of the problem. We noted that the

4.1 Comparison with other genetic algorithms

As we have already stated, Bäck and Khuri have implemented a genetic algorithm, for the independent set problem. It is difficult to compare our results directly with theirs because their algorithm has not been implemented directly on the DIMACS challenge problems. Further, the sizes of the graphs which they have tested have been restricted to only 100 or 200 nodes. One interesting statistic is that their algorithm converges to the global optimum solution in less than 10% of the runs. (By comparison, the smallest graph on which we tested our solution was at least 200 nodes, and on such graphs the algorithm converged to the global optimum solution in every run.) We conjecture that their use of a penalty function method [23] results in a population largely consisting of infeasible solutions. Consequently, their algorithm might spend a large fraction of its time evaluating infeasible solutions.

We have implemented a different genetic algorithm which we call R-Key by using the traditional crossover and mutation mechanisms. The technique used by R-Key is essentially Jim Bean's [3] method of using random keys. Each node i of the graph is assigned a key value $Key(i)$ which initially is an integer selected uniformly at random from the interval $[0, 2^k - 1]$ for some choice of k. The keys determine the representation of the solution indirectly. To determine an independent set from the keys, one runs the greedy algorithm after first sorting the nodes in order of increasing keys. We describe the procedure in detail below.

The random keys approach has been applied to a wide range of problems including the traveling salesperson problem, the quadratic assignment problem, the knapsack problem, and some interesting special cases of the integer programming problem [3]. To our knowledge, this is the first implementation of the random keys approach to the independent set problem.

In order to simplify the description of the algorithm in pseudo-code, we let $\pi(j)$ denote the node whose key value is jth lowest. For example, if there are three nodes with key values .3, .4, and .2 respectively, then node 3 has the least key value, and $\pi(1) = 3$. Similarly, $\pi(2) = 1$, and $\pi(3) = 2$ in this case. One may view π as a measure of priority, and view $\pi(1)$ as the node whose priority is the highest. Using this view, for any particular assignment of keys, the random keys algorithm selects the (lexicographically) highest priority independent set.

A description of the way in which the fitness function of R-Key on a string $x = x_1 x_2 \ldots x_n$ is evaluated is as follows:

Function Fitness($Key(1), Key(2), \ldots, Key(n)$)
begin
  Sort the keys, and let $\pi(i)$ denote
    the index of ith smallest key;
  $S = \{\}$
  for $i = 1$ to n **do**
  begin
    Add node $\pi(i)$ to S provided that
      independence condition is not violated
  end
  return($\mid S \mid$)
end

slowdown was greater for problems of larger size. The estimate of 3.21 probably underestimates our relative running time on smaller problems, and overestimates our relative running time on larger problems. Therefore, an average value is not in our favor for the bigger problems.

Problem	Optimum	v_{OCH}	$v_{\text{R-Key}}$	t_{OCH}	$t_{\text{R-Key}}$
brock200-2	12	11	10	0.96	7.4
brock200-4	17	16	16	0.80	22.5
brock400-2	29	24	22	8.25	132.1
brock400-4	33	24	22	5.11	66.0
brock800-2	24	19	18	15.31	730.1
keller4	11	11	11	0.59	30.2
keller5	27	25	24	38.40	300.4

Table 1: Relative Performance of the genetic algorithm (OCH) with respect to the genetic algorithm R-Key *

A major advantage of the random keys approach is that it is a very simple mechanism of ensuring feasibility after crossover. This method creates a feasible solution for any set of key values including those obtained after crossover and mutation. In order to implement the operation, the keys used were integers in the range $[0, 2^k - 1]$ represented in binary. We used a two-point crossover mechanism, and the standard mutation operator, with a mutation rate of 0.02. As we can see from the computational results in Table 1, R-Key was unable to find the optimum solution for most of the cases, but performed reasonably well on most test problems.

In implementing R-Key , we ran each genetic algorithm three times and selected the best of the three solutions. We chose three solutions so that the running times of R-Key would be more directly comparable to OCH. (Our choice of three was not of particular significance.)

4.2 Comparison with CLIQMERGE

Balas and Niehaus [5] have used the the method of optimal merging (optimized crossover) in order to come up with a heuristic for the independent set problem. Their heuristic enumerates many more sets of parents than ours, and accordingly has a much greater running time. Our genetic algorithm using an optimized crossover mechanism is approximately 100 to 1000 times faster. We report the comparative running times in Table 2. While the running times of our algorithm are orders of magnitude faster, the quality of solution obtained is slightly worse. To check whether the improved performance of CLIQMERGE was due primarily to its searching a larger source, we ran OCH 20 times and selected the best solutions for a subset of DIMACS problem instances, and compared these results to CLIQMERGE. In each case tested, we obtained a solution quality which is equally good or better than that of CLIQMERGE, and has significantly less running time. The computation results are illustrated in Tables 2 and 3.

4.3 Comparison with the Continuous Based Heuristic of Gibbons et. al.

Gibbons et. al. have presented a deterministic heuristic, called the continuous based heuristic [13] for the independent set problem. This heuristic provides reasonably robust solutions in times comparable to OCH. Further, since this heuristic has already been tested comprehensively on the DIMACS test problems, it is relatively easy for us to compare our results to theirs. Here is a summary of the comparisons.

Problem	Optimum	v_{OCH}	$v_{\text{CLIQMERGE}}$	t_{OCH}	$t_{\text{CLIQMERGE}}$
brock200-2	12	11	11	0.96	68
brock200-4	17	16	16	0.80	135
brock400-2	29	24	25	8.25	917
brock400-4	33	24	25	5.11	876
brock800-2	24	19	21	15.31	2288
brock800-4	26	19	21	33.01	2252
keller4	11	11	11	0.59	69
keller5	27	25	27	38.40	4997
MANN-a27	126	126	126	8.19	19371
MANN-a45	345	343	344	110.36	32953

Table 2: Relative Performance of the Genetic Algorithm (OCH) with respect to CLIQMERGE *

Problem	Optimum	v_{OCH}	$v_{\text{CLIQMERGE}}$	t_{OCH}	$t_{\text{CLIQMERGE}}$
brock200-2	12	12	11	20.0	68
brock200-4	17	17	16	18.6	135
brock400-2	29	25	25	80.9	917
brock400-4	33	26	25	69.7	876
brock800-2	24	21	21	138.2	2288
brock800-4	26	22	21	1008.6	2252
keller4	11	11	11	8.9	69
keller5	27	27	27	43.89	4997
MANN-a27	126	126	126	175.6	19371
MANN-a45	345	345	344	3723.8	32953

Table 3: Performance of the Optimized Crossover Heuristic (OCH) with multiple executions *

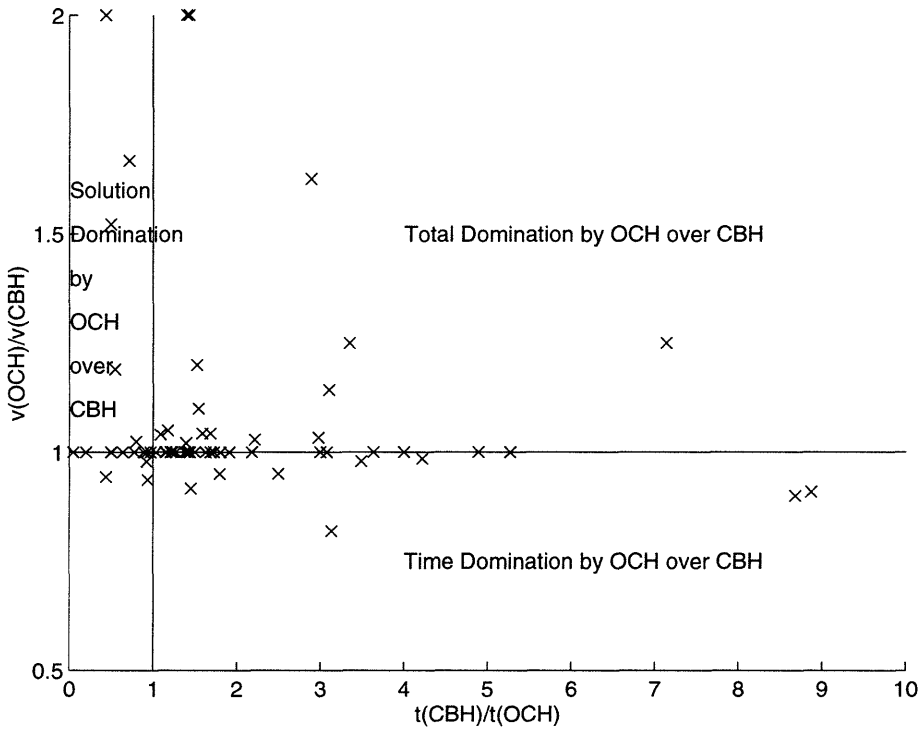

Figure 2: Relative performance of OCH and CBH

Graph	Nodes	Edges	v^*_{OCH}	v^*_{CBH}	v^*_{Opt}	t_{OCH}	t_{CBH}
c-fat-200-1	200	3235	12	12	12	0.59	1.02
c-fat-200-2	200	3235	24	24	24	1.12	0.72
c-fat-200-5	200	8473	58	58	58	1.15	0.58
c-fat-500-1	500	4459	14	14	14	6.07	8.6
c-fat-500-10	500	46627	126	126	≥ 126	16.2	12.6
c-fat-500-2	500	9139	26	26	26	5.01	6.08
c-fat-500-5	500	23191	64	64	64	8.00	7.43
johnson16-2-4	120	5460	8	8	8	0.16	0.27
johnson32-2-4	196	107880	16	16	16	8.20	9.49
johnson8-2-4	28	210	4	4	4	0.01	0.04
johnson8-4-4	70	1855	14	14	14	0.02	0.06
keller4	171	9435	11	10	11	0.59	0.91
keller5	776	225990	25	21	27	38.40	21.04
keller6	3361	4619898	DNR	DNR	≥ 59	DNR	DNR
hamming10-2	1024	518656	512	512	512	110.28	107.07
hamming10-4	1024	434176	33	35	≥ 40	50.34	21.96
hamming6-2	64	1824	32	32	32	0.02	0.03
hamming6-4	64	704	4	4	4	0.02	0.01
hamming8-2	256	31616	128	128	128	5.00	1.01
hamming8-4	256	20864	16	16	16	1.06	1.31
san1000	1000	250500	10	8	10	32.00	107.07
san200_0.7_1	200	13930	30	15	30	0.97	1.36
san200_0.7_2	200	13930	15	12	18	0.29	2.07
san200_0.9_1	200	17910	70	46	70	2.64	1.31
san200_0.9_2	200	17910	60	36	60	2.21	1.58
san200_0.9_3	200	17910	36	30	44	0.99	1.51
san400_0.5_1	400	39900	13	8	13	2.12	6.13
san400_0.7_1	400	55860	40	20	40	10.00	14.32
san400_0.7_2	400	55860	30	15	30	4.39	8.76
san400_0.7_3	400	55860	16	14	22	3.21	9.95
san400_0.9_1	400	71820	100	50	100	30.39	13.3

Table 4: Relative performance of OCH and CBH

Graph	Nodes	Edges	v^*_{OCH}	v^*_{CBH}	v^*_{Opt}	t_{OCH}	t_{CBH}
sanr200_0.7	200	13868	18	18	18	0.71	1.20
sanr200_0.9	200	17863	42	41	≥ 42	1.71	1.37
sanr400_0.5	400	39984	12	12	13	3.11	3.96
sanr400_0.7	400	55869	20	20	≥ 21	3.06	4.23
brock200_1	200	14834	21	20	21	0.85	1.00
brock200_2	200	9876	11	12	12	0.96	1.39
brock200_3	200	12048	14	14	15	0.77	4.06
brock200_4	200	13089	16	16	17	0.80	0.92
brock400_1	400	59723	24	23	27	2.41	3.83
brock400_2	400	59786	24	24	29	8.25	15.79
brock400_3	400	59681	24	23	31	3.25	5.48
brock400_4	400	59765	24	24	33	5.11	8.33
brock800_1	800	207505	19	20	23	30.46	54.65
brock800_2	800	208166	19	19	24	15.31	47.12
brock800_3	800	207333	19	20	25	22.13	55.17
brock800_4	800	207643	19	19	26	33.01	47.59
p_hat300-1	300	10933	8	8	8	2.01	3.63
p_hat300-2	300	21928	25	25	25	3.00	3.11
p_hat300-3	300	33390	36	36	36	1.20	5.87
p_hat500-1	500	31569	9	9	9	7.27	15.86
p_hat500-2	500	62946	36	35	36	6.12	13.56
p_hat500-3	500	93800	49	49	≥ 50	4.20	15.27
p_hat700-1	700	60999	9	11	11	15.2	47.6
p_hat700-2	700	121728	44	44	44	36.40	32.42
p_hat700-3	700	183010	62	60	≥ 62	13.87	41.31
p_hat1000-1	1000	122253	9	10	10	12.85	111.52
p_hat1000-2	1000	244799	45	46	≥ 46	91.32	84.38
p_hat1000-3	1000	371746	64	65	≥ 65	31.79	134.17
p_hat1500-1	1500	284923	10	11	12	40.23	356.81
p_hat1500-2	1500	568960	59	63	≥ 63	179.76	167.86
p_hat1500-3	1500	847244	92	94	≥ 94	180.54	630.08
MANN_a27	378	70551	126	121	126	8.19	8.93
MANN_a45	1035	533115	343	336	345	110.36	153.90
MANN_a81	3321	5506380	DNR	DNR	≥ 1100	DNR	DNR
MANN_a9	45	918	16	16	16	0.83	0.04

Table 5: Relative performance of OCH and CBH (contd.)

(1) Our heuristic was more *robust* than the Continuous Based Heuristic (CBH) in [13]. Although CBH performs well in general, it does not perform well on two of the nine classes of graphs available in the DIMACS directory in terms of the quality of solution obtained. Our algorithm performs well on both of these. In the case of the Sanchis graphs the quality of the solutions obtained by our genetic algorithm is significantly better. On the other graphs, the performance of our algorithm is competitive to CBH. Thus our algorithm performed robustly on each and every class of graph for which it was tested. We consider this a very important feature of our results, because quite a few heuristics for the independent set problem are known not to be too robust when faced with different kinds of graphs.

(2) The running times of OCH were typically less than those used by CBH. In some cases, the performance of our algorithm was significantly faster. On a few instances our method was slightly slower. Most importantly, we found that our algorithm ran much faster on the larger instances than CBH, while on the smaller instances the times required were relatively competitive. This is evidence of the fact that our algorithm dominates the Continuous Based Heuristic when the size of the instance is large. To compare the relative running times of OCH and CBH, side by side with the solution quality, we present a two dimensional plot of v_{OCH}/v_{CBH} versus t_{CBH}/t_{OCH} in Figure 2. As we can see, the graph is divided into four quadrants, denoting the areas in which OCH dominates over CBH, either totally, or in time only or solution quality only. The graph suggests that CBH and OCH are often comparable in one dimension, but OCH is usually far better than CBH in either CPU time or solution quality or both.

The details of the running times of the Continuous Based Heuristic and the Optimized Crossover Heuristic are illustrated in Tables 4 and 5.

5 Conclusion and Summary

In this paper we introduced the natural idea of optimized crossover for Genetic Algorithms. We tested it on the independent set problem, and presented computational results, which verifies the inherent power of this method, at least in this special case. It would only be fair at this stage to point out that it is not necessarily possible to apply the method to each and every kind of problem. Hence, we would like to lay down some general principles to be kept in mind while deciding whether or not a problem can be successfully solved by using optimized crossover.

It is not difficult to see that the time to actually perform the crossover is a very important component in the running time, and is likely to be the bottleneck. We observe that for dense graphs, the maximum independent set is typically $O(log(n))$, and hence, the corresponding matching problem for the optimized crossover mechanism can be solved in $O(log^3(n))$ time. (We observed that the running time in practice is $O(log^2(n))$, since only one or two extra augmentations were typically needed to convert the initial (maximal) matching into a maximum matching.) Thus, the time to perform an optimized crossover is asymptotically less than the time spent in implementing the crossover mechanism of the usual genetic algorithm crossover, which stores each independent set as a bit-string of size n. In general, the optimized crossover approach is well suited to cases in which the number of conflicting genes between the two parents is sufficiently small, since determining an optimized crossover may be formulated as an integer programming problem, in which the decision variables are the conflicting bits of the parents. A number of important 0-1 combinatorial optimization problems do fall into this category. When finding an optimized crossover is computa-

tionally prohibitive, it may be possible to design good approximation algorithms for the crossover itself, which may lead to good genetic algorithms in practice.

Acknowledgements

We wish to acknowledge ONR contract N00014-94-1-0099 as well as a grant from the UPS foundation. We also thank Joe Marks, Kim Kinnear, and Hitendra Wadhwa for helpful suggestions.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[2] T. Bäck and S. Khuri. *An evolutionary heuristic for the Maximum Independent Set Problem.* Proceedings of the First IEEE conference on evolutionary computation, IEEE press, pp. 531-535, 1994.

[3] J. C. Bean. *Genetics and Random Keys for Sequencing and Optimization.* Technical Report 92-43, University of Michigan, Ann Arbor, MI 48109-2117.

[4] E. Balas and C. S. Yu. *Finding the maximum clique in an arbitrary graph.* SIAM Journal of Computing. 15:4 pp., 1054-1068, 1986.

[5] E. Balas and W. Niehaus. *Finding Large Cliques by Bipartite Matching.* Management Science Research Report MSRR-597. Graduate School of Business Administration, Carnegie Mellon University.

[6] C. Bron and J. Kerbosch. *Finding all cliques of an undirected graph.* Communications of the ACM. 16:6, pp. 575-577, 1973.

[7] R. Carraghan and P. M. Paradalos. *An exact algorithm for the maximum clique problem.* Operations Research Letters, 9, pp. 375-382, 1990.

[8] K. A. De Jong. *Analysis of the behavior of a class of Genetic Adaptive Systems.* Ph. D. Dissertation, University of Michigan, Ann Arbor, MI, 1975.

[9] L. J. Eshelman, R. A. Caruana, J. D. Schaffer. *Biases in the Crossover Landscape.* Proceedings of the third International Conference on Genetic Algorithms. pages 10-19, June 1989.

[10] M. Garey, and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[11] M. Gendreau, J. C. Picard, and L. Zubieta. *An Efficient Implicit Enumeration Algorithm for the Maximum Clique Problem.* Springer Verlag, Lecture Notes in Economics and Mathematical Systems 304, A. Kurzhanski et. al. (ed.) 1988, 79-91.

[12] L. Gerhards and W. Lindenberg. *Clique detection for nondirected graphs: Two new algorithms.* Computing 21, pages 295-322, 1979.

[13] L. E. Gibbons, D. W. Hearn, and P. M. Paradalos *A Continuous Based Heuristic for the Maximum Clique Problem.* DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 00, 0000, 1991.

[14] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor MI 1975.

[15] E. Loukakis and C. Tsouros. *Determining the number of internal stability of a graph.* Internal Journal of Computing Mathematics 11, pp. 232-248, 1982.

[16] D. E. Goldberg . Simple genetic algorithms and the minimal deceptive problem. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing,* Chapter 6, pages 74-88. Morgan Kaufmann, Los Altos, CA, 1987.

[17] D. E. Goldberg. *Genetic Algorithms and Walsh Functions: Part II, Deception and Its Analysis.* Complex Systems (3), pages 153-171, 1989.

[18] L. E. Gibbons, P. M. Paradalos, and D. W. Hearn. *An Exact Algorithm With Accelerated Pruning for the Maximum Clique Problem.* DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 00, 0000, 1991.

[19] F. Glover. *Tabu Search- Part 1,* ORSA Journal of Computing, 190-206, 1989.

[20] P. M. Paradalos and G. P. Rodgers. *A Branch and Bound Algorithm for the Maximum Clique Problem.* Computers and Operations Research. 19:5, pp. 363-375, 1992.

[21] P. M. Paradalos and J. Xue. *The Maximum Clique Problem.* Journal of Global Optimization. 4: pp. 301-328, 1994.

[22] C. R. Reeves.(Ed.) *Modern Heuristic Techniques for Combinatorial Problems.* John Wiley and Sons Inc., 1993.

[23] J. T. Richardson, M. R. Palmer, G. Liepens, and M. Hilliard. *Some guidelines for Genetic Algorithms with penalty functions.* Proceedings of the 3rd International Conference on Genetic Algorithms. Morgan Kaufmann Publishers, San Mateo, CA, pages 191-197.

[24] R. E. Tarjan and A. E. Trojanoswski. *Finding a maximum independent set.* SIAM Journal of Computing. 6:3, pp. 537-546, 1977.