This is a post-peer-review, pre-copyedit version of an article published in Liefooghe A & Paquete L (eds.) Evolutionary Computation in Combinatorial Optimization. Lecture Notes in Computer Science, 11452. EvoCOP 2019: European Conference on Evolutionary Computation in Combinatorial Optimization, Leipzig, Germany, 24.04.2019-26.04.2019. Cham, Switzerland: Springer International Publishing, pp. 131-146. The final authenticated version is available online at: https://doi.org/10.1007/978-3-030-16711-0_9

Quasi-Optimal Recombination Operator

Francisco Chicano[1][0000−0003−1259−2990], Gabriela Ochoa[2][0000−0001−7649−5669], Darrell Whitley[3], and Renato Tinós[4][0000−0003−4027−8851]

[1] University of Malaga, Spain *
chicano@lcc.uma.es
[2] University of Stirling, UK
gabriela.ochoa@cs.stir.ac.uk
[3] Colorado State University, USA
whitley@cs.colostate.edu
[4] University of Sao Paulo, Brazil
rtinos@ffclrp.usp.br

Abstract. The output of an optimal recombination operator for two parent solutions is a solution with the best possible value for the objective function among all the solutions fulfilling the gene transmission property: the value of any variable in the offspring must be inherited from one of the parents. This set of solutions coincides with the largest dynastic potential for the two parent solutions of any recombination operator with the gene transmission property. In general, exploring the full dynastic potential is computationally costly, but if the variables of the objective function have a low number of non-linear interactions among them, the exploration can be done in $O(4^\beta(n+m)+n^2)$ time, for problems with n variables, m subfunctions and β a constant. In this paper, we propose a quasi-optimal recombination operator, called Dynastic Potential Crossover (DPX), that runs in $O(4^\beta(n+m)+n^2)$ time in any case and is able to explore the full dynastic potential for low-epistasis combinatorial problems. We compare this operator, both theoretically and experimentally, with two recently defined efficient recombination operators: Partition Crossover (PX) and Articulation Points Partition Crossover (APX). The empirical comparison uses NKQ Landscapes and MAX-SAT instances.

Keywords: Recombination operator, dynastic potential, gray box optimization.

1 Introduction

Many binary recombination operators for genetic algorithms have the property of *gene transmission* [1]. When the solutions are represented by a set of variables taking values from a set (possibly different for each of them) with no other constraint among the variables, this property implies that any variable

* This research is funded by the Spanish Ministry of Economy and Competitiveness and FEDER under contract TIN2017-88213-R, and the University of Malaga.

in any child will take the value of the same variable in one of the parents. In particular, the variables having the same value for both parents will have the same value in all the children (i.e., the *respect* property [1] is obeyed). The other (differing) variables will take one of the values coming from a parent solution. The set of all the solutions that can be generated by a recombination operator from two parents is called *dynastic potential*. If we denote by $d(x, y)$ the Hamming distance (number of differing variables) between two solutions x and y, the largest dynastic potential of a recombination operator is $2^{d(x,y)}$. Uniform crossover has this dynastic potential. The dynastic potential of single-point crossover has size $2d(x, y)$ and the one of two-point crossover has size $2d(x, y) + \binom{d(x,y)-1}{2} = 1 + d(x, y)(d(x, y) + 1)/2$. In general, z-point crossover has a dynastic potential of size $O(d(x, y)^z)$ for $z << n$, with n variables.

Our goal in this paper is to design an *Optimal Recombination Operator* [2], which is one obtaining the best offspring from the largest dynastic potential. In the worst case, however, such a recombination operator is computationally expensive, since finding the best offspring in the largest dynastic potential is an NP-hard problem. For this reason, we design a *Quasi-Optimal Recombination Operator*, with worst time complexity $O(4^\beta(n + m) + n^2)$ where m is the number of subfunctions and β is an arbitrary constant. This operator will find the best offspring of the largest dynastic potential if the objective function has low espitasis, that is, if the number non-linear interactions among variables is small.

Our proposed operator, called Dynastic Potential Crossover (DPX), uses the variable interaction graph of the objective function to simplify the evaluation of the $2^{d(x,y)}$ solutions in the dynastic potential by using dynamic programming. The ideas for this efficient evaluation date back to Hammer's basic algorithm for variable elimination [3] and are also commonly used in operations over Bayesian networks [4]. Since it requires more information than just the objective function to do the job, this operator is framed in the so-called gray box optimization [5].

Recently defined crossover operators similar to ours are Partition Crossover (PX) [6] and Articulation Points Partition Crossover (APX) [7]. Although they were proposed to work with pseudo-Boolean functions, they can also be applied to the more general representation of variables defined over a finite alphabet. PX and APX also use the variable interaction graph of the objective function to efficiently compute a good offspring among a large number of them. PX and APX have $O(n^2 + m)$ time complexity and both of them obtained excellent performance in different problems [8, 7, 9, 10]. When combined with other gray box optimization operators, partition crossover was capable of optimizing instances with 1 million variables in seconds. We compare DPX with these two operators from a theoretical point of view and in the experimental section.

The paper is organized as follows. Section 2 presents the required background to understand the working principles of DPX. The proposed recombination operator is presented in Section 3. Section 4 describes the experiments and presents the results and, finally, Section 5 concludes the paper.

2 Background

We will work along the paper with functions defined over a set of variables x_i, each one taking values in a finite set, X_i, not necessarily the same for all the variables. We say that a function f of n variables has k-bounded epistasis if it can be written as a sum of m subfunctions f_l, each one depending on at most k variables:

$$f(x) = \sum_{l=1}^{m} f_l(x_{i_{l,1}}, x_{i_{l,2}}, \ldots, x_{i_{l,k}}), \tag{1}$$

where $i_{l,j}$ is the index of the j-th variable in subfunction f_l. In the case of binary variables, these functions have been named Mk Landscapes by Whitley et al. [5]. In *Gray Box Optimization*, the optimizer can evaluate the set of m subfunctions in Equation (1) (although their internal structure is unknown). This contrasts with Black Box Optimization, where the optimizer can only evaluate solutions and get their fitness value.

2.1 Variable Interaction Graph

The *Variable Interaction Graph* (VIG) [5] is a useful tool that can be constructed under Gray Box Optimization. It is a graph $VIG = (V, E)$, where V is the set of variables and E is the set of edges representing all pairs of variables (x_i, x_j) having *nonlinear interactions*. These nonlinear interactions can be captured in two ways. First, assuming that every pair of variables appearing together in a subfunction have a nonlinear interaction. A second approach is to apply the Fourier transform [11], and then look at every pair of variables to determine if there is a non-zero Fourier coefficient associated to a term with the two variables. This second method is more precise and not very expensive, because the Fourier transform can be constructed in $O(n)$ time for k-bounded epistasis functions.

An example of the construction of the variable interaction graph for a function with $n = 18$ variables (numbered from 0 to 17) and $k = 3$, is given below. We will refer to variables using numbers, e.g., $9 = x_9$. The objective function is the sum over the following 18 subfunctions:

$$\begin{array}{llll}
f_0(0,6,14) & f_5(5,4,2) & f_{10}(10,2,17) & f_{15}(15,7,13) \\
f_1(1,0,6) & f_6(6,10,13) & f_{11}(11,16,17) & f_{16}(16,9,11) \\
f_2(2,1,6) & f_7(7,12,15) & f_{12}(12,10,17) & f_{17}(17,5,2) \\
f_3(3,7,13) & f_8(8,3,6) & f_{13}(13,12,15) & \\
f_4(4,1,14) & f_9(9,11,14) & f_{14}(14,4,16) &
\end{array}$$

From these subfunctions, assume we extract the nonlinear interactions that are shown in Figure 1. In this example, every pair of variables that appear together in a subfunction has a nonlinear interaction.

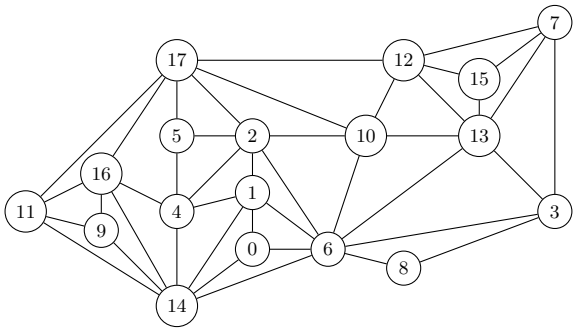

Fig. 1. Sample Variable Interaction Graph (VIG).

2.2 Recombination Graph

Let us assume that we have two solutions to recombine using the optimal recombination operator. We call these two solutions the *red* and the *blue* parents. All the variables with the same value in both parents will also share the same value in the offspring and the solutions in the dynastic potential will be in a hyperplane determined by the common variables. In the solution representation we will use digit 0 to denote that the variable has the same value as in the red parent and 1 to denote that the value is different. Thus, the red solution will be the string with all 0s. For example, let the two parents be

$$red = 000000000000000000 \quad \text{and} \quad blue = 111101011101110110$$

in our sample function of Section 2.1. Therefore, x_4, x_6, x_{10}, x_{14}, and x_{17} are identical in both parents. The rest of the variables are different. Both parents reside in a hyperplane denoted by $h = ****0*0***0***0**0$ where $*$ denotes the variables that are different in the two solutions, and 0 marks the positions where they have the same variable values.

We use the hyperplane $h = ****0*0***0***0**0$ to decompose the VIG in order to produce a *Recombination Graph*. We remove all the variables (vertices) that have the same "shared variable assignments" and also remove all edges that are incident on the vertices corresponding to these variables. This produces the recombination graph shown in Figure 2.

The recombination graph also defines a reduced evaluation function. This new evaluation function is linearly separable, and decomposes into q subfunctions defined over the recombining components. In our example:

$$g(x') = a + g_1(9, 11, 16) + g_2(0, 1, 2, 5) + g_3(3, 7, 8, 12, 13, 15),$$

where $g(x') = f|_h(x')$ and x' are restricted to a subspace of the hyperplane h that contains the parent strings as well as the full dynastic potential. The constant $a = f(x') - \sum_{i=1}^{3} g_i(x')$ depends on the common variables.

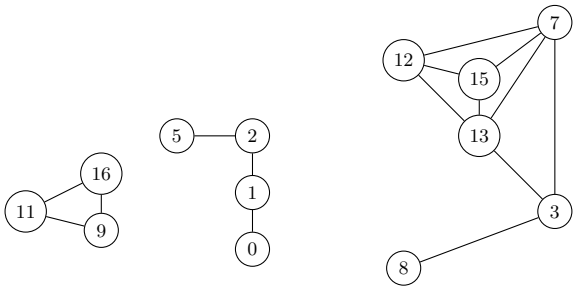

Fig. 2. Recombination Graph for the solutions (parents) $red = 0000000000000000000$ and $blue = 111101011101110110$.

The Partition Crossover operator (PX), defined by Tinós et al. [6] is based on this recombination graph. Every recombination graph with q connected components induces a new *separable* function $g(x')$ that is defined as:

$$g(x') = a + \sum_{i=1}^{q} g_i(x'). \tag{2}$$

Partition Crossover selects the parent yielding the best partial solution for each subfunction $g_i(x')$. All of the variables in the same recombining component in the recombination graph must be inherited together from one of the two parents.

Articulation Points Partition Crossover (APX) [7] goes further and finds the *articulation points* of the recombination graphs. They are variables whose removal increases the number of connected components. Variables x_1, x_2 and x_3 are articulation points in our example (see Figure 2). Then, APX efficiently simulates what happens when the articulation points are removed, one at a time, from the recombination graph by flipping the articulation point in any of the parent solutions before applying PX, and the best solution is returned as offspring. With the appropriate data structures, this can be done in $O(n^2 + m)$, the same complexity of PX.

3 Dynastic Potential Exploration

The proposed Dynastic Potential Crossover Operator (DPX) takes the idea of PX and APX even further. DPX starts from the recombination graph, like the one in Figure 2, and tries to exhaustively explore all the possible combinations of the parent values in the variables of each connected component to find the optimal recombination regarding the hyperplane h defined by the blue and red parents. This exploration is not done by brute force, but using dynamic programming. Following with our example, in order to compute the best combination for the variables x_9, x_{11} and x_{16}, we need to enumerate the 8 ways of taking each variable from each parent, and this is not better than brute force. However, component x_0, x_1, x_2, x_5, forms a thread. In this case we can store in a table which is the

best option for variable x_0 when any of the two possible values for variable x_1 are selected and we can store in the same table what is the value of the sum of subfunctions depending only on x_0 and x_1 (and possibly common variables eliminated in the recombination graph). After this step, we can consider that variable x_0 has been removed from the problem and we can proceed in the same way with the rest of the variables in the order x_1, x_2 and x_5. At the end, only 12 evaluations are necessary, while a brute force would require 16 evaluations.

The idea of variable elimination using dynamic programming dates back to the 1960's and Hammer's basic algorithm [3]. It is well-known that the complexity of this approach is $O(N2^t)$, where t is the treewidth of the graph. Computing the treewidth of a graph is an NP-hard problem [4]. Thus, heuristics should be applied to find an elimination order for the variables. The problem of variable elimination has also been studied in other contexts, like Gaussian Elimination [12] and Bayesian Networks [4]. In fact, we follow the works done for computing the *junction tree* in Bayesian Networks. In order to do this, we first need a *chordal graph* and then compute the *clique tree* (or junction tree), which will fix the order in which the variables are eliminated using Dynamic Programming. Our contribution in this work consists in applying these ideas to the recombination operator. The high level pseudocode of the proposed DPX is presented in Algorithm 1. In the next subsections we will detail each of these steps.

Algorithm 1 Pseudocode of DPX

Input: two parents x and y
Output: one offspring z
1: Compute the Recombination Graph of x and y as in [6]
2: Apply Maximum Cardinality Search to the Recombination Graph [12]
3: Apply the fill-in procedure to make the graph chordal [12]
4: Apply the Clique Tree construction procedure [13]
5: Assign subfunctions to cliques in the clique tree
6: Apply Dynamic Programming to find the offspring (see Algorithm 2)
7: Build z using the tables filled by Dynamic Programming

3.1 Chordal Graphs

A *chordal graph* is a graph where all the cycles of length 4 or more have a chord (edge joining two nodes not adjacent in the cycle). All the connected components in Figure 2 are chordal graphs. Tarjan and Yannakis [12] provided algorithms to test if a graph is chordal and add new edges to make it chordal if it is not. Their algorithms run in time $O(n + e)$, where e is the number of edges in the graph. In the worst case the complexity is $O(n^2)$. The first step to check the chordality is to number the nodes using *Maximum Cardinality Search* (MCS). This algorithm numbers each node in descending order, choosing always the

unnumbered node with a higher number of numbered neighbors and solving the ties arbitrarily. Figure 3 (left) shows the result of applying MCS to the third connected component of Figure 2.

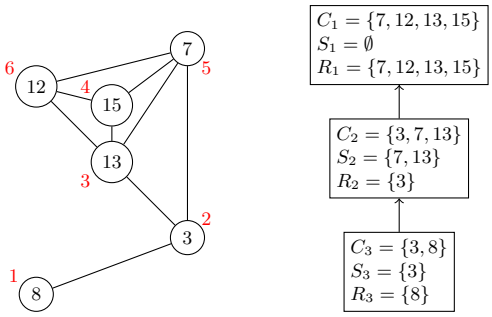

Fig. 3. Maximum Cardinality Search applied to the third connected component of Figure 2 (left) and clique tree with the sets of separators and residues (right).

If the graph is chordal then MCS will provide a numbering of the nodes such that for each triple of nodes u, v and w, with $(u, v), (u, w) \in E$ and u has a lower number than v, w, it happens that $(v, w) \in E$. If this is not the case, the graph is not chordal. A *fill-in* algorithm tests this condition and adds the required edges to make the graph chordal. This algorithm runs in $O(n + s')$ time, where s' is the number of edges in the final chordal graph. Again, in the worst case, the complexity is $O(n^2)$. These two steps, MCS and fill-in, can be computed to each connected component separately or to the complete recombination graph with the same result. The algorithms are applied in Lines 2 and 3 of Algorithm 1.

3.2 Clique Tree

Dynamic Programming is based on the exhaustive exploration of the cliques[5] in the chordal graph. The maximum size of a clique in the chordal graph is an upper bound of its treewidth, and determines the complexity of applying dynamic programming to find the optimal solution. A clique tree of a chordal graph is a tree where the nodes are cliques and for any variable appearing in two of such cliques, the path among the two cliques in the tree is composed of cliques containing the variable (junction tree property). We can also identify a clique tree with a tree-decomposition of the chordal graph [4]. This clique tree will determine the order in which the variables can be eliminated.

Starting from the chordal graph provided in the previous steps, we apply an algorithm by Galinier et al. [13] to find the clique tree (Line 4 in Algorithm 1).

[5] We will use the term *clique* to refer to a maximal complete subgraph, as the cited literature does. However, the term clique is sometimes used to refer to a complete subgraph (not necessarily maximal).

This algorithm runs also in $O(n + e')$ time and finds all the $O(n)$ cliques of the chordal graph. The cliques will be denoted with C_i, where i is an index that increases when a clique is discovered by the algorithm. An edge joining two cliques in the clique tree is labelled with a *separator*, which is the intersection of the variables in both cliques. A clique C_i is parent of a clique C_j if they are joined by an edge and $i < j$. In each clique C_i the *residue*, R_i, is the set of variables that are not in the separator with its parent. In each clique C_i, the residue, R_i, and the separator with the parent, S_i, forms a partition of the variables in C_i. It is not hard to prove that each variable is in the residue of one clique only. In Figure 3 (right) the residues and separators for all the cliques of the third connected component of Figure 2 are shown.

After computing the clique tree, all the subfunctions f_l depending on a nonempty set V of differing variables must be assigned to one (and only one) clique C_i containing V (Line 5 in Algorithm 1). They will be evaluated when this clique is processed. There can be more than one clique where the subfunction can be assigned. All of them are valid for a correct evaluation.We denote with F_{C_i} the set of subfunctions assigned to clique C_i.

The optimal offspring is found by iteratively reducing the variables in the residue of the cliques (Line 6 in Algorithm 1 and Algorithm 2). The clique tree must be traversed in post-order in order to do this. During the clique evaluation, for each combination of variables in the separator S_i (Line 2 in Algorithm 2), all the combinations of variables in the residue R_i are considered (Line 4 in Algorithm 2) and evaluated over the subfunctions assigned to the clique (Lines 6-8) and their child cliques (Lines 9-11). The evaluation in post-order makes it possible to have the `value` array of the child cliques filled when they are evaluated. The best combination of the variables in R_i for each combination of the variables of S_i is stored in the array `variable` in Line 14. This array will be used in the reconstruction of the offspring solution (Line 7 in Algorithm 1). In Algorithm 2 we assume that value 0 for a variable means the value in the red parent and a 1 means the value in the blue parent. The term x_V for V a set of variables, will denote a vector with the variables in V.

The operator described is an Optimal Recombination operator: it finds the best offspring from the largest dynastic potential. The time required to evaluate one clique in Algorithm 2 is $O((|F_{C_i}| + |children(C_i)|)2^{|C_i|})$, where $children(C_i)$ is the set of child cliques of C_i. The number of children is bounded by n and the number of subfunctions m is bounded by $O(n^k)$ due to the k-bounded epistasis of f. However, the exponential factor is a threat to the efficiency of the algorithm. In the worst case C_i can contain all the variables and the factor would be 2^n.

3.3 Limiting the Complexity

In order to avoid the exponential runtime, we propose to limit the exploration in Lines 2 and 4. Instead of iterating over all the possible combinations for all the variables in S_i and R_i we fix a bound β on the number of variables that will be exhaustively explored. The remaining variables will jointly take only two values, each one coming from one of the parents. This reduces the exponential part of the

Algorithm 2 Optimal Offspring Computation

```
1: for all cliques C_i of the clique tree in post-order do
2:    for x_{S_i} ∈ {0,1}^{|S_i|} do
3:       value[x_{S_i}] = −∞
4:       for x_{R_i} ∈ {0,1}^{|R_i|} do
5:          aux = 0
6:          for f ∈ F_{C_i} do
7:             aux = aux + f(x)
8:          end for
9:          for children cliques C' of C_i do
10:            aux = aux + value[x_{C'}];
11:         end for
12:         if aux > value[x_{S_i}] then
13:            value[x_{S_i}] = aux
14:            variable[x_{S_i}] = x_{R_i}
15:         end if
16:      end for
17:   end for
18: end for
```

complexity of Algorithm 2 to $2^{2\beta}$. Since β is a predefined constant decided by the user of the algorithm, the exponential factor turns into a constant. The operator is not anymore an optimal recombination operator, and this is the reason why we call it *quasi-optimal*. In the cases where $\beta \geq |C_i|$ for all the cliques, the operator will still return the optimal offspring. The next theorem presents the complexity of DPX.

Theorem 1. *Given a function in the form of* (1) *with* m *subfunctions, the complexity of DPX with a constant bound* β *for the number of exhaustively explored variables is* $O(4^{\beta}(n+m)+n^2)$.

Proof. We have seen in Section 3.1 that the complexity of Maximum Cardinality Search, the fill-in procedure and the clique tree construction is $O(n^2)$. The assignment of subfunctions to cliques can be done in $O(n+m)$ time, using the variable ordering found by MCS to assign the subfunctions that depends on each visited variable to the only clique where the variable is a residue. The complexity of the dynamic programming computation is:

$$\begin{aligned} O\left(\sum_i (|F_{C_i}| + |children(C_i)|)2^{|C_i|}\right) &= O\left(2^{2\beta}\sum_i (|F_{C_i}| + |children(C_i)|)\right) \\ &= O\left(4^{\beta}(m + \sum_i |children(C_i)|)\right) \\ &= O(4^{\beta}(m+n)), \end{aligned}$$

where we used the fact that the sum of the cardinality of the children for all the cliques is the number of edges in the clique tree, which is the number of cliques minus one, and the number of cliques is $O(n)$. The reconstruction of the offpsring solution requires to read all the `variable` tables until building the solution. The complexity of this procedure is $O(n)$. □

In many cases, the number of subfunctions m is $O(n)$ or $O(n^2)$. This is true, in particular, when the function has k-bounded epistasis. In these cases, the complexity of DPX reduces to $O(4^\beta n^2)$.

3.4 Theoretical Comparison with (A)PX

It is clear that DPX is no worse than PX, since it considers each connected component in the recombination graph and, in the worst case, it will do the same as PX and will pick the variables from one of the parent solutions. We wonder, however, if this happens with APX. If β is large enough for a given recombination, it cannot be worse than any recombination operator with the property of gene transmission and, in particular, cannot be worse than APX. If β is not that large and the limit in the exploration (Subsection 3.3) is applied, it could happen that articulation points are not explored as they are in APX. One possible threat to the articulation points exploration in DPX is that they disappear after making the graph chordal. The next result proves that articulation points survive the fill-in procedure and inspires a mechanism to reduce the probability that a solution explored in APX is not explored in DPX.

Theorem 2. *Articulation points of a graph are kept after the fill-in procedure.*

Proof. Proving that all articulation points survive the fill-in procedure is equivalent to proving that all the edges added by the fill-in procedure join vertices of one single bi-connected component. If an edge (v, w) is added joining vertices of two different bi-connected components, then two paths would exist to go from v to w: the original path traversing at least one articulation point a and the new edge. But, in this case, the articulation point a could be removed from the graph. In the other direction, adding edges to a bi-connected component never removes articulation points.

We assume that MCS has been applied to the graph. We denote with $\gamma(v)$ the number assigned by MCS to node v. Let us prove the claim by contradiction. Imagine that edge (v, w) is added in the fill-in procedure, where v and w are in different bi-connected components. The definition of fill-in (see [12]) implies that there is a path among v and w where all the intermediate nodes have a γ value lower than v and w. In particular, since v and w are in different bi-connected components, all the paths between them include the same set of articulation points and for all of them the value of γ is lower than $\min(\gamma(v), \gamma(w))$. MCS numbers the nodes in a connected component in decreasing order and in such a way that all the numbered nodes are connected. Thus, in all the bi-connected components the first node numbered by MCS is an articulation point, with the only exception of the bi-connected component where the numbering starts. This

implies that in one of the bi-connected components, say the one of v, there is an articulation point a_v with $\gamma(a_v) > \gamma(v)$ that was the first numbered in that bi-connected component. Regarding the bi-connected component of w, if it is the one where the numbering started, then there must be an alternative path from w to v though a_v. But this means that w and v belongs to the same bi-connected component, what is a contradiction. If the bi-connected component of w is not where the numbering started, there must be an articulation point a_w with $\gamma(a_w) > \gamma(w)$ where the numbering started in that bi-connected component. Once again, there must be an alternative path between v and w through a_w and a_v, contradicting the fact that v and w are in different bi-connected components. Then, the fill-in procedure will not add edge (v, w). □

The previous theorem implies that articulation points of the original recombination graph are also articulation points of the chordal graph. Articulation points of a chordal graph are minimal separators of cardinality 1 (see [13]) and they will appear as separators S_i in some cliques C_i. They are, thus, identified during the clique tree construction. In each clique C_i when β variables are chosen to be exhaustively explored (Lines 2 and 4 of Algorithm 2) we choose the articulation points first. This way, articulation points can be exhaustively explored with higher probability. The only thing that can prevent articulation points from being explored is that many of them appear in one single clique. This situation is illustrated in Figure 4. For $\beta \leq 1$, the clique of articulation points is evaluated only in the two parent solutions, and the same happens with the other cliques, giving a total of 16 explored combinations. However, APX would explore 20 combinations in this situation (see Eq. (6) in [7]).

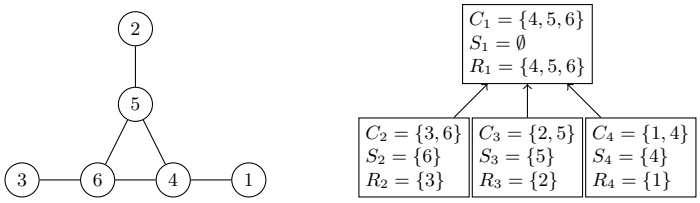

Fig. 4. Pathological component (left) in a recombination graph where DPX with $\beta \leq 1$ explores less solutions (16) than APX (20) and its clique tree (right).

4 Experiments

In order to experimentally analyze the performance of DPX, we included it in the Deterministic Recombination and Iterated Local Search (DRILS) algorithm [7]. We think this allows us to explore the performance of the operator in a real scenario, rather than generating random solutions and providing them to the operator. DRILS [8] uses a first improving move hill climber to reach a local

optimum. Then, it perturbs the solution by randomly flipping αN bits, where α is the so-called *perturbation factor*. It then applies local search to the new solution to reach another local optimum and applies crossover to the last two local optima, generating a new solution that is improved further with the hill climber. This process is repeated until a time limit is reached.

In our case, the recombination operator is DPX, but we also present results with PX and APX in Subsection 4.2 to compare the operators. In all the runs we set a time limit of 60s (1 minute). Since the algorithms are stochastic, we performed 10 independent runs for each instance and algorithm. We tested DRILS with DPX in NP-hard problems: random NKQ Landscapes with $K \geq 2$, which allows us to parameterize the density of edges in the VIG by changing K; and MAX-SAT instances of the MAX-SAT Evaluation 2017. Random NKQ ('Quantized' NK) landscapes [14] can be seen as Mk landscapes with one subfunction per variable ($m = n$). Each subfunction f_l depends on variable x_l and other $K = k - 1$ random variables, and the codomain of each subfunction is the set $\{0, 1, \ldots, Q - 1\}$, where Q is a positive integer. The values of the subfunctions are randomly generated. Random NKQ landscapes are NP-hard when $K = k - 1 \geq 2$. The computer used for the experiments is a multicore machine with four Intel Xeon CPU (E5-2670 v3) at 2.3 GHz, a total of 48 cores, 64 GB of memory and Ubuntu 16.04 LTS. The source code of all the algorithms can be found at https://github.com/jfrchicanog/EfficientHillClimbers.

4.1 DPX Statistics

In a first experiment, we compute statistics about DPX. In particular, in Tables 1 and 2 we count the average number of connected components identified in the recombination graph (Comp.), the average logarithm (in base 2) of the number of explored solutions (Exp.), the percentage of applications where the full dynastic potential is explored (Full) and the average runtime in milliseconds (Time). We used instances of random NKQ Landscapes with n =10 000 and n =100 000 variables. The value for K varies from 2 to 5, $Q = 64$ and β varies from 2 to 5. For each combination of the parameters n and K we generated 10 random instances and run DRILS with DPX 10 times. Thus, the numbers in the tables are averages over 100 runs (the percentage of full explorations counts all the applications of crossover in the 100 runs). The perturbation factor (α) in DRILS was set to $\alpha = 0.05$ in the cases $K = 2, 3$ and $\alpha = 0.01$ in the cases $K = 4, 5$. These values were taken from the recommendations in [8].

We observe in the tables that the percentage of applications of DPX where the full dynastic potential is explored is high, almost always around or above 90%, except in the case of N =100 000 and $K = 3$. This percentage should increase with β and it normally does, being the exceptions not significant. But the fact that the value is high for low values of β (2 or 3) is an indication that the cliques found in the recombination graph are small, with size 2 or 3 in most of the cases. One can imagine that this corresponds to threads of variables with some triangles sometimes. This corresponds with the plots presented by Chen et al. in [9]. Due to this high percentage of success we can trust that the logarithm

Table 1. DPX Statistics for n =10 000 variables.

	Comp.	Exp.	Full (%)	Time (ms)		Comp.	Exp.	Full (%)	Time (ms)
$K=2$					$K=4$				
$\beta=2$	64	212	99.9	6.8	$\beta=2$	13	44	99.3	6.2
$\beta=3$	64	210	100.0	3.9	$\beta=3$	12	42	100.0	3.5
$\beta=4$	64	210	100.0	3.9	$\beta=4$	12	42	100.0	3.4
$\beta=5$	64	210	100.0	3.9	$\beta=5$	12	42	100.0	3.6
$K=3$					$K=5$				
$\beta=2$	50	232	94.7	8.4	$\beta=2$	11	42	99.0	7.1
$\beta=3$	50	225	99.8	4.7	$\beta=3$	11	39	99.9	3.8
$\beta=4$	50	226	99.9	4.7	$\beta=4$	11	39	99.9	3.8
$\beta=5$	50	226	99.9	4.8	$\beta=5$	11	40	100.0	4.0

Table 2. DPX Statistics for n =100 000 variables.

	Comp.	Exp.	Full (%)	Time (ms)		Comp.	Exp.	Full (%)	Time (ms)
$K=2$					$K=4$				
$\beta=2$	668	2 249	99.9	75.5	$\beta=2$	140	567	91.2	69.2
$\beta=3$	668	2 249	100.0	74.1	$\beta=3$	140	566	99.1	68.8
$\beta=4$	668	2 248	100.0	72.5	$\beta=4$	141	571	99.3	71.0
$\beta=5$	670	2 261	100.0	81.6	$\beta=5$	142	587	99.2	82.2
$K=3$					$K=5$				
$\beta=2$	505	2 693	63.2	110.6	$\beta=2$	121	570	89.6	75.9
$\beta=3$	505	2 691	94.0	109.6	$\beta=3$	121	570	89.6	75.9
$\beta=4$	506	2 702	94.7	113.3	$\beta=4$	122	575	96.9	77.2
$\beta=5$	505	2 726	94.8	126.1	$\beta=5$	123	596	96.8	91.9

of the number of explored solutions (column "Exp." in the tables) is a good measure of the number of differing variables in the parent solutions. If we divide this number by the number of components we find a value between 3 and 4. This must be the average number of variables in each connected component.

Both the number of components and the differing variables are approximately multiplied by 10 when we compare the 10K variable instances with the 100K variable instances. We observe, however, that these values are similar for $K = 2, 3$ and are divided by 4 or 5 when $K = 4, 5$. The reason is the perturbation factor α, which is also divided by 5 in these instances.

The runtime is in the order of a few milliseconds for 10K variables and 70 to 100 milliseconds for 100K variables. This runtime should increase with β but we observe some exceptions for low β. The reason has to do with the procedures in DPX used to identify the group of variables that will be exhaustively explored and the one for which only the parent solutions will be evaluated. From the results of the tables, we conclude that a value for β of 3 or 4 is the best one for these instances. Two other parameters affecting the runtime are the pertur-

bation factor α, because it will determine the number of differing variables (the higher the value the higher the runtime), and K, since it will add edges to the recombination graph. This is why we observe that runtime increases from $K = 2$ to $K = 3$ and from $K = 4$ to $K = 5$. Anyway, this runtime is small compared to the number of solutions that are explored. If we take the results for N =100K and $K = 3$ as an example, DPX is exploring 2^{2693} solutions in 110 ms. This is equivalent to exploring around 10^{800} solutions per nanosecond (ns) if a black box approach is used.

4.2 Comparison with PX and APX for NKQ Landscapes

In this section we compare DPX with PX and APX. Table 3 shows a comparison regarding three aspects: exploration capacity, runtime and performance inside DRILS. The first two aspects depend only on the crossover operators and the third one depends also on the algorithm (DRILS). For the exploration capacity we show the logarithm in base 2 of the number of explored solutions by each operator. We observe how DPX has the largest exploration capacity, around the square of the one of APX (the logarithm is around double) and between the fourth and fifth power compared to the one of PX. In terms of runtime, DPX requires more time than PX and APX, as expected, and this time is between 20% and 70% higher than PX and APX. Finally, we compare the performance of DRILS using each of the crossover operators. For each instance (ten per value of K) we compare the medians of the algorithms after 1 minute of computation and we apply the Mann-Whitney test (with significant level 0.05) to check if the differences are statistically significant. The numbers followed by a black triangle ($\blacktriangle$), white triangle ($\triangledown$) and equal sign ($=$) are the numbers of instances in which DRILS with DPX is statistically better, worse or similar to DRILS with the operator of that column (PX or APX). The performance comparison suggests that DPX is improving the search of DRILS only for $K = 3$. In the other cases the other two operators (specially APX) are better. A complete explanation of this observation requires further research, but we can guess that DPX can be too greedy, providing a solution which is a (near) local optimum difficult to escape from. It also requires more time to run and this time is used in the other versions of DRILS to escape from the local optima. Both ideas can be checked with a Local Optimal Network (LON) analysis, which we defer to future work.

4.3 Comparison with PX and APX for MAX-SAT

In Table 4 we compare PX, APX and DPX using MAX-SAT instances from the MAX-SAT Evaluation 2017[6]. We used the same instances as in [7][7] to allow an easy comparison. They are 160 unweighted and 132 weighted instances.

We observe how DPX required on average twice the time required by PX and APX in each run, in the order of 2 to 5 milliseconds. However, the performance of DRILS using DPX is significantly better in most of the instances than

[6] http://mse17.cs.helsinki.fi/benchmarks.html.
[7] The list of instances is at https://github.com/jfrchicanog/EfficientHillClimbers.

Table 3. Comparison of PX, APX and DPX for N =100 000 variables. The value for α depends on K as described in the text and in DPX we used $\beta = 4$.

	Exploration			DRILS Performance		Runtime (ms)		
K	PX	APX	DPX	PX	APX	PX	APX	DPX
2	662	1 311	2 248	1▲ 0▽ 9 =	0▲ 8▽ 2 =	46	55	73
3	503	1 105	2 702	10▲ 0▽ 0 =	2▲ 0▽ 8 =	73	67	113
4	138	286	571	0▲ 4▽ 6 =	0▲ 9▽ 1 =	52	55	71
5	119	254	575	0▲ 9▽ 1 =	0▲ 10▽ 0 =	52	63	77

Table 4. Comparison of PX, APX and DPX for MAX-SAT instances (weighted and unweited). In all the cases $\alpha = 0.3$ in DRILS and $\beta = 4$ in DPX.

	DRILS Performance		Runtime (μs)		
Instances	PX	APX	PX	APX	DPX
Unweighted	126▲ 2▽ 32 =	96▲ 19▽ 45 =	1 060	849	1 907
Weighted	102▲ 14▽ 16 =	90▲ 17▽ 25 =	1 713	2 365	5 171

the performance when PX or APX is used. In particular, DPX is statistically better than APX and PX in 96 and 126 unweighted instances, respectively. The difference in the weighted instances is not so high, but still large enough to be promising. Interestingly, weighted MAX-SAT instances must have a fitness landscape similar to NKQ Landscapes while unweighted instances have a different fitness landscape, with many plateaus difficult to escape from. DRILS with DPX seems to work better than DRILS with PX and APX in such a plateau-based landscape. We defer to future work the detailed analysis of this performance and the big difference with NKQ Landscapes.

5 Conclusions

In this paper we propose a new gray box crossover operator, DPX, with the ability to obtain the best offspring out of the full dynastic potential if the density of interactions among the variables is low. We have provided theoretical results proving that DPX is no worse than Partition Crossover (PX) and usually no worse than Articulation Points Partition Crossover (APX). We also compared these three operators inside the DRILS algorithm in NKQ Landscapes and MAX-SAT, certifying its exploration ability.

An interesting future line of research is to analyze the operator using Local Optima Networks and the shape of the connected components of the recombination graph to understand the reasons for the observed different performance in NKQ Landscapes and MAX-SAT. It would also be interesting to check the performance of the operator in other algorithms and to develop a competitive MAX-SAT solver based on it.

References

1. Radcliffe, N.J.: The algebra of genetic algorithms. Annals of Mathematics and Artificial Intelligence 10(4), 339–384 (Dec 1994)
2. Eremeev, A.V., Kovalenko, J.V.: Optimal recombination in genetic algorithms. CoRR abs/1307.5519 (2013), http://arxiv.org/abs/1307.5519
3. Hammer, P.L., Rosenberg, I., Rudeanu, S.: On the determination of the minima of pseudo-boolean functions. Stud. Cerc. Mat. 14, 359–364 (1963)
4. Bodlaender, H.L.: Discovering treewidth. In: Vojtáš, P., Bieliková, M., Charron-Bost, B., Sýkora, O. (eds.) SOFSEM 2005: Theory and Practice of Computer Science. pp. 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
5. Whitley, D., Chicano, F., Goldman, B.W.: Gray box optimization for mk landscapes (nk landscapes and max-ksat). Evolutionary Computation 24, 491 – 519 (Jan-09-2016 2016)
6. Tinós, R., Whitley, D., Chicano, F.: Partition crossover for pseudo-boolean optimization. In: Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII. pp. 137–149. FOGA '15, ACM, New York, NY, USA (2015)
7. Chicano, F., Ochoa, G., Whitley, D., Tinós, R.: Enhancing partition crossover with articulation points analysis. In: Proceedings of GECCO. pp. 269–276. GECCO '18, ACM, New York, NY, USA (2018)
8. Chicano, F., Whitley, D., Ochoa, G., Tinós, R.: Optimizing one million variable NK landscapes by hybridizing deterministic recombination and local search. In: Genetic and Evolutionary Computation Conference, GECCO 2017. pp. 753–760 (2017)
9. Chen, W., Whitley, D., Tinós, R., Chicano, F.: Tunneling between plateaus: Improving on a state-of-the-art maxsat solver using partition crossover. In: Proceedings of GECCO. pp. 921–928. GECCO '18, ACM, New York, NY, USA (2018)
10. Tins, R., Zhao, L., Chicano, F., Whitley, D.: Nk hybrid genetic algorithm for clustering. IEEE Transactions on Evolutionary Computation 22(5), 748–761 (Oct 2018)
11. Terras, A.: Fourier Analysis on Finite Groups and Applications, Cambridge U. Press, Cambridge. Cambridge University Press (1999)
12. Tarjan, R.E., Yannakakis, M.: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. SIAM Journal on Computing 13(3), 566–579 (Aug 1984)
13. Galinier, P., Habib, M., Paul, C.: Chordal graphs and their clique graphs. In: Nagl, M. (ed.) Graph-Theoretic Concepts in Computer Science. pp. 358–371. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
14. Newman, M.E.J., Engelhardt, R.: Effect of neutral selection on the evolution of molecular species. Proc. R. Soc. London B pp. 1333–1338 (1998)