Enhancing Partition Crossover with Articulation Points Analysis

Francisco Chicano
University of Málaga
Malaga, Spain
chicano@lcc.uma.es

Gabriela Ochoa
University of Stirling
Stirling, Scotland, UK
gabriela.ochoa@cs.stir.ac.uk

Darrell Whitley
Colorado State University
Fort Collins, Colorado, USA
whitley@cs.colostate.edu

Renato Tinós
University of São Paulo
Ribeirão Preto, São Paulo, Brazil
rtinos@ffclrp.usp.br

ABSTRACT

Partition Crossover is a recombination operator for pseudo-Boolean optimization with the ability to explore an exponential number of solutions in linear or square time. It decomposes the objective function as a sum of subfunctions, each one depending on a different set of variables. The decomposition makes it possible to select the best parent for each subfunction independently, and the operator provides the best out of 2^q solutions, where q is the number of subfunctions in the decomposition. These subfunctions are defined over the connected components of the recombination graph: a subgraph of the objective function variable interaction graph containing only the differing variables in the two parents. In this paper, we advance further and propose a new way to increase the number of linearly independent subfunctions by analyzing the articulation points of the recombination graph. These points correspond to variables that, once flipped, increase the number of connected components. The presence of a connected component with an articulation point increases the number of explored solutions by a factor of, at least, 4. We evaluate the new operator using Iterated Local Search combined with Partition Crossover to solve NK Landscapes and MAX-SAT.

CCS CONCEPTS

• **Mathematics of computing → Combinatorial optimization**;
• **Theory of computation → Random search heuristics**;

KEYWORDS

Partition Crossover, Gray-box optimization, articulation points, pseudo-Boolean optimization

ACM Reference Format:
Francisco Chicano, Gabriela Ochoa, Darrell Whitley, and Renato Tinós. 2018. Enhancing Partition Crossover with Articulation Points Analysis. In *GECCO '18: Genetic and Evolutionary Computation Conference, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205455.3205561

1 INTRODUCTION

Pseudo-Boolean optimization problems are encoded as binary strings and produce real numbers as output. The class of k-bounded pseudo-Boolean optimization problems are those where the nonlinearity of the objective function is restricted to at most k interactions. In these problems, the objective function can be expressed as a sum of M subfunctions. In Gray-Box optimization [7], the optimizer is given access to these M subfunctions. The optimizer does not need to know the specific application, nevertheless useful problem structure can be extracted from the subfunctions.

Two recent Gray-Box optimization advances have resulted in improved algorithms for solving k-bounded pseudo-Boolean problems. The first advance is the use of lookahead methods that can identify improving moves in constant time [2], which makes traditional random mutation operators unnecessary. The second advance is the development of Partition Crossover [6], a deterministic greedy form of recombination that analytically decomposes parents into *recombining components*. These recombining components, in turn, decompose the evaluation function into linearly separable subfunctions during recombination. If q recombining components are found, Partition Crossover finds the best of 2^q offspring in linear time. Combinations of these two techniques have produced hybrid algorithms able to solve adjacent NK landscapes to optimality on instances with up to one million variables [1]. For random NK landscapes, optimality cannot be assessed, but the proposed hybrid algorithms outperform the previous state-of-the-art.

The performance of Partition Crossover is related to the number of connected components it can find in the *recombination graph*, a subgraph of the objective function variable interaction graph containing only the differing variables in the two parents. This paper proposes an improvement over Partition Crossover, consisting in flipping the *articulation points* of the recombination graph. These points correspond to variables that, once flipped, increase the number of connected components (and subfunctions). The presence of an articulation point in a connected component increases the number of explored solutions by a factor of, at least, 4, compared to the original Partition Crossover. The improvements of the proposed operator are evaluated within the recent state-of-the-art Gray-Box Iterated Local Search combined with Partition Crossover, DRILS algorithm [1], to solve NK landscapes of up to one million variables and a set of real-world MAX-SAT instances.

The rest of the paper is organized as follows. Section 2 overviews relevant background. Section 3 describes the articulation point

analysis, including the foundations and implementation of the new Partition Crossover. Section 4 describes the experimental studies conducted on both NK Landscapes and MAX-SAT instances and discusses the results obtained. Finally, Section 5 outlines our key findings and potential future developments.

2 BACKGROUND

A *pseudo-Boolean function* is a real-valued function of Boolean variables. A k-bounded pseudo-Boolean function f of N variables is written as a sum of M subfunctions, each one depending on at most k variables:

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

where f_l is a subfunction depending on k decision variables and $i_{l,j}$ is the index of the j-th variable in subfunction l. These functions have been named Mk Landscapes by Whitley et al. [7]; examples are NK Landscapes (with $k = K + 1$), MAX-kSAT, and Unconstrained Quadratic Optimization (with $k = 2$). In Gray Box Optimization, the optimizer is given access to the set of M subfunctions in Equation (1).

NKQ ('Quantized' NK) landscapes [4] can be seen as Mk landscapes with one subfunction per variable ($M = N$). Each subfunction f_i depends on variable x_i and other $K = k-1$ variables, and the codomain of each subfunction is the set $\{0, 1, \ldots, Q-1\}$, where Q is a positive integer. The subfunctions are randomly generated, but the dependencies of the variables in each subfunction are generated according to a given model. Two NK models are widely used. The adjacent model, in which subfunction f_i depends on consecutive variables (x_i, x_{i+1}, ..., x_{i+k-1}); and the random model, in which f_i depends on x_i and other $K = k - 1$ random variables. Other models in between can be defined [7], but the adjacent and random models are extreme as one is very easy to solve and the other is very hard to solve. Adjacent NKQ landscapes can be optimized in polynomial time $O(N)$ using dynamic programming [9]. Random NKQ landscapes, however, are NP-hard when $K = k - 1 \geq 2$.

2.1 Variable Interaction Graph

The variable interaction graph (VIG) [7] 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 Boolean 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. This is almost always true for NK landscapes. A second approach is to convert the k-bounded pseudo-Boolean function into a Walsh polynomial [3], and then look at every pair of variables to determine if they are indexed by a Walsh coefficient. This second method is both more precise and more efficient because the Walsh polynomial can be constructed in $O(N)$ time.

An example of the construction of the Variable Interaction Graph for a random NK landscape with $N = 18$ variables (numbered from 0 to 17) and $K = 2$ ($k = 3$), is given below. We will refer to variables using numbers, e.g., $9 = x_9$. The NK landscape sums over the following 18 subfunctions:

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

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.

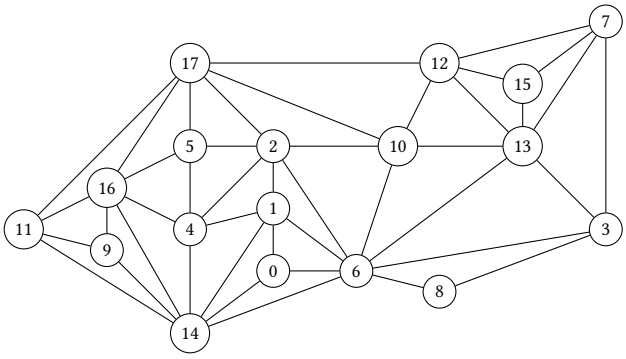

Figure 1: Sample Variable Interaction Graph (VIG).

2.2 Partition Crossover

The Variable Interaction Graph can be used to implement a deterministic recombination operator: Partition Crossover (PX) [6]. When the parents are locally optimal, Partition Crossover acts as a tunneling algorithm that can move directly from local optima to local optima with high probability. Partition Crossover is a form of *greedy, deterministic recombination*. It takes two solutions (parents), extracts the variable assignments they share, and uses these shared variable assignments to decompose both the VIG and the evaluation function. Considering the example in Figure 1, let the two parents be

$$P_1 = 000000000000000000 \text{ and } P_2 = 111100011101110110$$

Therefore, $x_4 = x_5 = x_6 = x_{10} = x_{14} = x_{17} = 0$ in both parents. Otherwise, $x_i = 0$ in P_1 and $x_i = 1$ in P_2 for all of the other bits. Both parents reside in a hyperplane denoted by $h = ****000***0***0**0$ where $*$ denotes the bits that are different in the two solutions, and 0 marks the positions where they have the same bit values (again, without loss of generality).

We use the hyperplane $h = ****000***0***0**0$ 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 bits. This produces the recombination graph shown in Figure 2.

We can search for connected components of the recombination graph to identify the *recombining components*. The decomposition shown in Figure 2 results in $q = 3$ recombining components. All of the variables that appear together in the same recombining component in the recombination graph must be inherited together

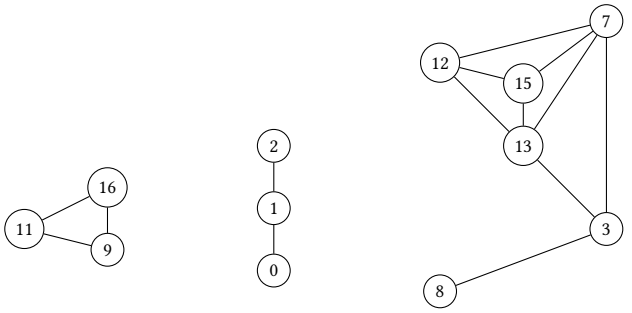

Figure 2: Recombination Graph for the solutions (parents) P_1 = 0000000000000000 and P_2 = 1111000111011101 10.

from one of the two parents. 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.

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

where $g(x') = f|_h(x')$ and x' are restricted to a subspace of the hyperplane h that contains the parent strings P_1 and P_2 as well as all of their potential offspring under Partition Crossover. The constant $a = f(x') - \sum_{i=1}^{3} g_i(x')$ depends on the common variables. We can now see how Partition Crossover works. Every recombination over q recombining components induces a new *separable* function $g(x')$ that is defined as:

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

Since $g(x')$ is a separable function, Partition Crossover can be greedy and select which parent yields the best partial solution for each subfunction $g_i(x')$. The following Partition Crossover Theorem was originally proven to hold for the Traveling Salesman Problem [8]. Tinós et al. [6] have proven the following result also holds for all k-bounded pseudo-Boolean functions.

Theorem 2.1 (The Partition Crossover Theorem). *Given q linearly separable recombining components with bounded epistasis, Partition Crossover returns the best of $2^q - 2$ reachable solutions distinct from parent solutions P_1 and P_2 in $O(N)$ time.*

3 ARTICULATION POINTS ANALYSIS

The performance of Partition Crossover is related to the number of connected components it can find in the recombination graph, because the operator implicitly explores a number of solutions which is exponential in the number of connected components. We propose here an improvement over Partition Crossover, consisting in flipping some variables in one of the parent solutions in order to break the connected components of the recombination graph, increasing the number of connected components. A node in a graph whose removal can break a connected component is called *articulation point* [5] (see Figure 3). By finding and evaluating the articulation points of the recombination graph, our proposed operator is able to explore an exponentially larger set of solutions with the same asymptotic cost as the original Partition Crossover, that is, $O(m)$ where m is the number of edges in the recombination graph. The new operator is called Articulation Points Partition Crossover (APX). In short, for each variable which is an articulation point of the recombination graph, APX computes the increase in the objective function of assigning the same value to that variable in both parents and applying Partition Crossover. This computation is independently performed for each connected component and all the contributions are added to give the overall contribution. If there is no articulation point in the recombination graph or removing an articulation point does not increase the objective value, the operator works as the original PX. In the following sections we detail the theoretical background of the operator.

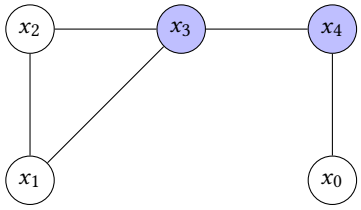

Figure 3: Example of articulation points. Nodes x_3 and x_4 are articulation points of the graph.

3.1 Finding Articulation Points

Articulation points in a graph can be found using an algorithm due to Tarjan [5]. This algorithm is a slight modification of a Depth First Search (DFS) exploration of the graph. The algorithm can also be used to find the connected components required for PX. Let's call *DFS tree* the exploration tree that is obtained after a DFS exploration of a graph. Then, a node v is an articulation point if any of the following two conditions hold [5]:

- the node is the root of the DFS tree and it has more than one child, or
- the node is not the root of the DFS tree and it has a child subtree with all its edges incident in nodes found not earlier than v in the DFS tree.

These conditions can be used to implement an algorithm to find all the articulation points of a graph $G(V, E)$. The complexity of this algorithm is: $O(|V| + |E|)$. In the case of a k-bounded pseudo-Boolean function, $|E|$ is proportional to $|V|$ and the complexity is $O(|V|) = O(N)$.

3.2 Evaluating Articulation Points

Removing an articulation point is not always useful, since it implies flipping a variable in one of the parent solutions and this could decrease the objective value, yielding an offspring that may not improve the parent solutions. Let x and y denote the parent solutions, $G(V, E)$ the recombination graph and $C \subseteq V$ one connected component of G. We will denote with 1_C a binary string with 1 in the positions of the variables in C and 0 in the remaining positions (of variables not in C). We will use F to denote the set of subfunctions f_i whose sum is f. Given a set of variables C, we denote with F_C the subset of subfunctions of F that depend on a variable in C. Let

us call g_C to the sum of subfunctions that depend on a variable[1] in C, that is, $g_C = \sum_{h \in F_C} h$.

The original Partition Crossover selects the values for the variables in C from one of the parents (x or y) in such a way that g_C is maximum. If we call z the offspring of PX, this means: $g_C(z) = \max(g_C(x), g_C(y))$. Thus, it explores two possible values for the variables in C for each component. For example, if the graph in Figure 3 is one connected component in the recombination graph of $x = 00000\ldots$ and $y = 11111\ldots$, PX takes for this component the first five variables from x or y, depending on the value of $g_{\{0,1,2,3,4\}}(x)$ and $g_{\{0,1,2,3,4\}}(y)$ (two combinations).

In the case of APX, given a connected component C, with articulation points set $AP(C)$, and an articulation point $a \in AP(C)$, it explores the values for the variables in C where a is flipped in x or y, breaking C into d_a disjoint connected components. All the new components can inherit the values of any of the parents independently. Figure 4 shows a hypothetical connected component, C, and highlights one articulation point a. We denote with C_i ($i = 1, 2, \ldots, d_a$) the connected components in which C is broken down if a is removed, and we call them *connected sub-components*. In the figure, the value of $d_a = 4$.

Using our previous example based on Figure 3, there are two articulation points in the recombination graph (variables 3 and 4). We can break the graph in variable 3 if we do $y_3 = 0$ or $x_3 = 1$. In any of these two cases there are two connected sub-components: $\{1, 2\}$ and $\{0, 4\}$. APX can decide the source (x or y) for the variables in each connected sub-component independently. Thus, a total of eight different combinations for variables 0 to 4 are explored and the best one of them is taken. These combinations are 00000, 00010, 01110, 10011, 11111, 11101, 01100, 10001. Something similar happens when variable 4 is considered. It breaks the recombination graph in two connected sub-components, and the eight combinations analyzed are 00000, 00001, 10001, 01111, 11111, 11110, 01110, 10000. There are four combinations in common between the two sets (00000, 11111, 01110, 10001), so the total number of combinations analyzed for the first five variables is 12. This is six times the number of combinations analyzed by the original PX.

The next lemma provides an expression for the value of $g_C(z)$ after applying APX when C has one single articulation point.

Lemma 3.1. *Given two solutions x and y whose offspring by APX is z, and a connected component C of the recombination graph of x and y with one single articulation point $a \in AP(C)$ joining d_a connected sub-components, the value of $g_C(z)$ is:*

$$g_C(z) = \max_{t \in \{x,y\}} \left(\sum_{h \in F_{a^*}} h(t \oplus 1_a) + \sum_{i=1}^{d_a} \Delta_i(t) \right) \quad (3)$$

where $\Delta_i(t) = \max(g_{C_i}(t \oplus 1_a), g_{C_i}(t \oplus 1_C))$ is the maximum value that the connected sub-component C_i can take if variable a is flipped (removed from the graph), and $F_{a^*} = F_{\{a\}} - \cup_{i=1}^{d_a} F_{C_i}$ is the set of subfunctions depending on a but not on any other variable in C. The previous value is the best of 2^{d_a+1} combinations for the variables in C. We denote with $\oplus$ the bitwise exclusive OR operator for binary strings.

[1]The g_C functions defined here are a generalization of the g_i functions introduced in Section 2.2.

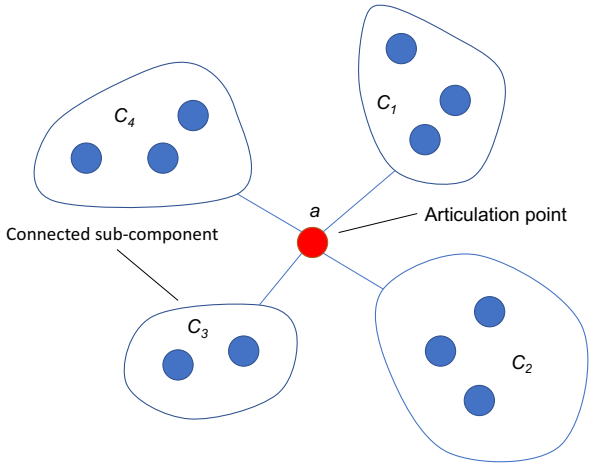

Figure 4: One connected component in the recombination graph. We can appreciate one of the articulation points in the center and the connected sub-components joined by the articulation point.

Proof. APX behaves like PX applied to two situations: 1) variable a is flipped in x and 2) variable a is flipped in y. Then, it takes the best offspring of these two scenarios and the value of $g_C(z)$ is the maximum value obtained by g_C in both of them.

Let us consider case 1). If variable a is flipped in x, for each connected sub-component C_i we have to take the best of two possibilities: the variables in C_i take the values of x, and the variables in C_i take the values of y. The latter coincide with $x \oplus 1_C$ for the variables in C_i, what allows us to write the contribution of this sub-component as $\Delta_i(x) = \max(g_{C_i}(x \oplus 1_a), g_{C_i}(x \oplus 1_C))$. These contributions are summed to be included in the value of g_C. Observe, however, that not all the subfunctions in F_C appear in the union of the F_{C_i}. In particular, the subfunctions depending on a but not on any other variable of C will not be summed in the previous expression. Thus, we need to explicitly add these subfunctions by including the term $\sum_{h \in F_{a^*}} h(x \oplus 1_a)$. This case explores 2^{d_a} combinations for the variables of C, since each sub-component C_i take the decision in an independent way.

Case 2) is exactly the same as case 1), but using y instead of x. It also explores 2^{d_a} new combinations for the variables of C, and the sum of the two cases is 2^{d_a+1}. □

The computation of Lemma 3.1 can be done for each articulation point in a connected component C. In general, if r articulation points are identified in a connected component, there are 2^r potential decompositions to analyze in that connected component. This could be inefficient and ineffective in many cases. On the other hand, we observed in preliminary experiments that the number of articulation points per connected component tends to be low (from 1 to 3) and the articulation points form a line with high probability (see Figure 7). In these situations, many of the 2^r potential decompositions can provide similar values for the offspring objective function evaluation. For this reason, APX only allows one articulation point to be removed per connected component. It

selects the one providing the maximum value for g_C. We defer the analysis of flipping multiple articulation points in a connected component to future work. The next theorem presents the expression for the objective function of an offspring z of APX and the number of implicitly explored solutions (from which the offspring is the best).

Theorem 3.2. *Given two solutions x and y whose offspring by APX is z, the value of $f(z)$ is*

$$f(z) = \sum_{C \in CC(G)} g_C(z) + \sum_{h \in F - F_{x \oplus y}} h(z) \tag{4}$$

where $CC(G)$ is the set of connected components of the recombination graph of x and y, and $F - F_{x \oplus y}$ is the set of subfunctions that only depend on variables with the same value in x and y. The expression of $g_C(z)$ is:

$$g_C(z) = \max_{\substack{a \in AP(C) \\ t \in \{x,y\}}} \left(\sum_{h \in F_{a^*}} h(t \oplus 1_a) + \sum_{i=1}^{d_a} \Delta_{C_i^a, a}(t) \right) \tag{5}$$

where $\Delta_{C_i^a, a}(t) = \max\left(g_{C_i^a}(t \oplus 1_a), g_{C_i^a}(t \oplus 1_C)\right)$ is the contribution of the connected sub-component C_i^a when articulation point a is removed and $F_{a^*} = F_{\{a\}} - \cup_{i=1}^{d_a} F_{C_i^a}$ is the set of subfunctions that depend on a but not on any other variable in C. The number of solutions that APX implicitly explores is:

$$E(x,y) = 2^{|CC(G)|} \prod_{C \in CC(G)} \left(1 - e_C + \sum_{a \in AP(C)} \left(2^{d_a} - 1\right)\right), \tag{6}$$

where e_C is the number of edges in the connected component C joining two articulation points. Observe that $2^{|CC(G)|}$ is the number of solutions implicitly explored by the original PX.

Proof. Equation (5) is a consequence of Lemma 3.1 where the maximum over all the articulation points in a connected component is taken. Equation (4) is a sum of the contribution to the objective value of each connected component plus the sum of the evaluation of the subfunctions that were not considered in the evaluation of the connected components because they do not depend on any variable in a connected component (they only depend on variables with the same values in both parents).

Regarding the number of solutions implicitly explored, the independent analysis of each articulation point (Lemma 3.1) provides a count of 2^{d_a+1} combinations of variables, but we have to subtract from the count the combinations that are explored in the analysis of two different articulation points. Two combinations implicitly explored by the analysis of any articulation point are those of parent solutions x and y. We have to remove these two combinations from any 2^{d_a+1} count and add 2 to the final sum (to take them into account). If two articulation points are joined by an edge, there are two additional common combinations generated by choosing a different parent for the variables at each side of the edge (see Figure 5). Thus, we have to subtract $2e_C$ to the count of explored combinations, where e_C is the number of edges joining two articulation points in C.

Given two articulation points, if they are not joined by an edge, then the only two common combinations explored by both are the parent combinations. Figure 6 helps to see this. Let us suppose that

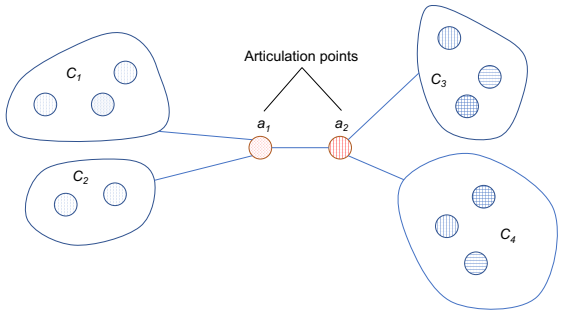

Figure 5: Two articulation points joined by an edge. The analyses of a_1 and a_2 explore four common combinations, where the variables at each side of the edge select one of the two parents independently.

we are exploring a combination common to the analysis of a_1 and a_2. Then, all the variables in the sub-component C_3 must be taken from the same parent, including a_2 and v. The analysis of a_2 reveals that a_1 and the sub-components C_1 and C_2 must be taken from the same parent as variable v in sub-component C_3. Thus, all the variables in the component must be taken from the same parent and we are exploring one of the parent combinations.

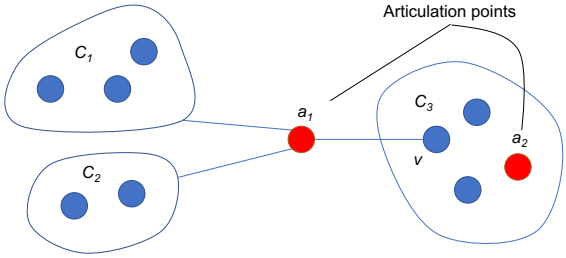

Figure 6: Two articulation points not joined by an edge. The analyses of a_1 and a_2 explore only two common combinations: the ones found in the parent solutions.

In summary, the number of explored combinations in one connected component C is:

$$\sum_{a \in AP(C)} (2^{d_a+1} - 2) - 2e_C + 2 = 2\left(1 - e_C + \sum_{a \in AP(C)} (2^{d_a} - 1)\right). \tag{7}$$

The product of Equation (7) extended to all the connected components of G is Equation (6). □

A more practical (easier to remember) lower bound for the number of explored solutions is provided in the next corollary.

Corollary 3.3. *Given two solutions x and y, a lower bound of the number of solutions implicitly evaluated in APX is:*

$$E(x,y) \geq 2^{|CC(G)|} \prod_{\substack{C \in CC(G) \\ |AP(C)| > 0}} 2(1 + |AP(C)|) \tag{8}$$

Proof. According to Eq. (6), each connected component with articulation points contribute to the count with an additional factor of $\left(1 - e_C + \sum_{a \in AP(C)}(2^{d_a} - 1)\right)$. We can easily compute a lower bound of this expression by considering that $d_a \geq 2$ (a direct consequence of the definition of articulation point) and $e_C \leq |AP(C)| - 1$. The latter can be explained by the fact that articulation points and biconnected components (the subgraphs in between) form a tree [5]. The maximum number of edges joining two articulation points is the number of edges of a tree formed by the articulation points, and this is $|AP(C)| - 1$. Thus, we have $2^{d_a} - 1 \geq 3$ and

$$\begin{aligned}\left(1 - e_C + \sum_{a \in AP(C)} (2^{d_a} - 1)\right) &\geq (1 - e_C + 3|AP(C)|) \\ &\geq (1 - |AP(C)| + 1 + 3|AP(C)|) \\ &\geq 2(1 + |AP(C)|),\end{aligned}$$

what finishes the proof. □

Equation (8) clearly shows that the number of implicitly explored solutions is much higher than that of PX. In APX, each connected component with articulation points contributes with an additional factor to the count of explored solutions of, at least, $2(1 + |AP(C)|)$. This means, that any connected component with one articulation point, increases the count in, at least, a factor of 4 compared to PX. A component with two articulation points increases the count in, at least, a factor of 6 compared to PX.

The computation of the best articulation point to remove (if any) and the best decision for each sub-component in a connected component of the recombination graph can be done during the DFS exploration. The computation effort required to do this analysis only increases the time in a constant factor compared to PX. There is no change in the asymptotic behaviour of the operator run time, which is $O(N)$ for k-bounded pseudo-Boolean functions and $O(N^2)$ in the general case.

4 EXPERIMENTS

In order to experimentally analyze the performance of APX, we included it in the Deterministic Recombination and Iterated Local Search (DRILS) algorithm. DRILS [1] 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 Partition 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. The pseudocode is shown in Algorithm 1.

In addition to the original DRILS algorithm, we implement a variant where the Partition Crossover operator in Line 4 of Algorithm 1 is replaced by APX. This version is called DRILS+APX in the rest of the paper. 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 used NP-hard problems to measure the performance of APX: Random NKQ Landscapes with $K \geq 2$ and MAX-SAT.

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 memory usage was limited to 3GB during all the executions. The source code of DRILS and DRILS+APX is available at https://github.com/jfrchicanog/EfficientHillClimbers.

Algorithm 1 DRILS

```
1: current ← hillClimber(random());
2: while not stopping condition do
3:    next ← hillClimber (perturb(current));
4:    child ← PX(current, next);
5:    if child = current or child = next then
6:       current ← next;
7:    else
8:       current ← hillClimber(child);
9:    end if
10: end while
```

4.1 APX Statistics

In a first experiment, we compute statistics about the new operator. In particular, we count the number of connected components identified in the recombination graph, the number of articulation points, the number of connected sub-components joined by the articulation points (d_a), and the number of explored solutions in one recombination. To collect these data, we used random NKQ Landscapes, where $N = 10^5$ variables and $N = 10^6$ variables, K goes from 2 to 5 and $Q = 64$. For each combination of the parameter N and K we generated 10 random instances and run DRILS+APX 10 times. The perturbation factor (α) in DRILS was set to $\alpha = 0.05$ in the case $K = 2, 3$ and $\alpha = 0.01$ in the case $K = 4, 5$. These values were taken from the recommendations in [1].

Table 1 shows averages over all the recombinations appearing in all the runs for each combination of N and K. In the case of the number of explored solutions, we compute the binary logarithm and provide the average. This makes it possible to easily compare the number of solutions explored by APX and PX, since the number of components (third column) is the binary logarithm of the number of solutions explored by PX.

Table 1: APX Statistics.

N	K	#Comp.	#APs	d_a	$\log_2 E(x,y)$
10^5	2	662	687	2.25	1 311
	3	503	1 151	2.37	1 105
	4	138	196	2.33	286
	5	119	218	2.36	254
10^6	2	7 774	10 836	2.28	15 987
	3	4 515	21 793	2.35	9 454
	4	1 748	6 281	2.38	3 907
	5	1 105	7 207	2.34	2 341

We can observe in Table 1 that the number of articulation points can be similar to the number of components, but it can also be several times larger, indicating that each connected component can

have several articulation points. The trend in the number of articulation points is not as clear as the number of components. While the number of components decrease with K, the number of articulation points can increase or decrease with K. The average degree of the articulation points is slightly larger than 2 (its minimum value). It is not common to see high degrees; the maximum value we observed in the experiments was 13. High values are more probable when K is high. Regarding the number of explored solutions, we observe that the logarithm is around two times the number of components. This means that APX implicitly explores a set of solutions with a size that is around the square of the number of solutions explored by PX. According to the data in Table 1, this number is between $2^{254} = 10^{76}$ and $2^{15\,987} = 10^{4\,813}$.

Figure 7 shows the recombination graph for two local optima during a run of DRILS+APX. For visualization purposes, we chose one of the smallest recombination graphs we observed in the experiments, for an instance with $N = 10^5$ variables and $K = 2$.

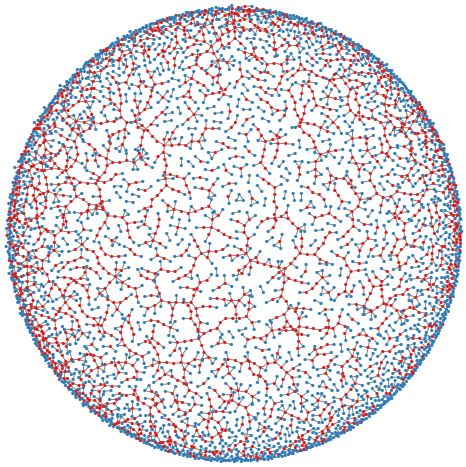

Figure 7: Example of recombination graph during a DIRLS+APX run for an NK instance with $N = 10^5$ and $K = 2$, showing the connected components and articulation points (red nodes). The graph contains 858 components and 4 339 nodes, of which 1 825 are articulation points (42%).

4.2 Performance Comparison in NKQ Landscapes

The goal of the second experiment is to determine when APX is beneficial when compared to the original PX. Table 2 shows in the third and fourth columns the number of instances where each algorithm (DRILS and DRILS+APX) statistically outperforms the other. The fifth column reports the number of instances where there is no statistically significant difference between the algorithms. We used the Mann-Whitney test for the comparison, and marked a difference as significant when the test reports a p-value below 0.05.

We can observe that DRILS+APX is statistically better than DRILS in 40 instances, DRILS is better than DRILS+APX in 4 instances and they are both similar in 36 instances.

Figure 8 shows the average fitness over time obtained by DRILS and DRILS+APX for a concrete NKQ configuration with $N = 10^6$ and $K = 3$ (average over 100 samples). We can clearly see how DRILS+APX outperformed DRILS after a few seconds.

Table 2: Number of NKQ instances where any of the algorithms statistically outperforms the other or the two are similar. The average runtime of one execution of APX and PX is also shown.

		DRILS performance			Runtime (ms)	
N	K	APX	PX	Sim.	APX	PX
10^5	2	10	0	0	55	46
	3	10	0	0	67	73
	4	2	0	8	55	52
	5	1	1	8	63	52
10^6	2	2	3	5	1 383	970
	3	5	0	5	1 785	2 485
	4	9	0	1	1 360	1 439
	5	1	0	9	1 633	1 559

Columns sixth and seventh of Table 1 also report the average runtime of one application of APX and PX. The numbers are in the same order of magnitude although sometimes PX is faster and sometimes is slower than APX. Thus, we can claim that the extra computation does not have a big impact in the runtime.

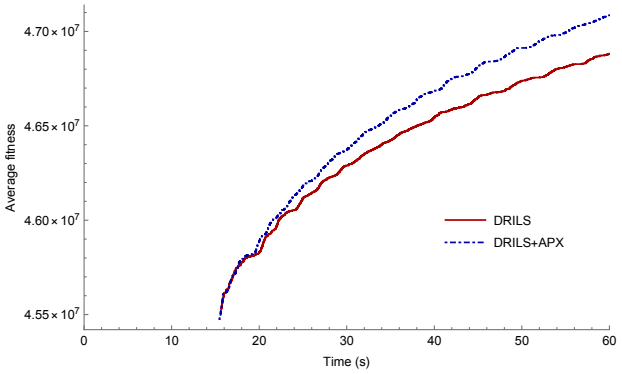

Figure 8: Average fitness over time obtained by DRILS and DRILS+APX in all the instances and all the runs of NKQ Landscapes for $N = 10^6$ and $K = 3$.

4.3 Performance Comparison in MAX-SAT

We used the weighted and unweighted benchmarks for incomplete solvers of the MAX-SAT Evaluation 2017[2]. From the 194 instances in the unweighted benchmark, our implementation worked with 160, running out of memory in the remaining 34 instances. In the case of the weighted benchmark the implementation worked on 132 out of the 156 instances. This section reports the results obtained over these 292 MAX-SAT instances. Table 3 reports the number of instances where each algorithm statistically outperforms the other

[2] http://mse17.cs.helsinki.fi/benchmarks.html.

(columns three and four). The fifth column reports the number of instances where there is no statistically significant difference (using Mann-Whitney with significance level 0.05) between the algorithms. Three different values for the perturbation factor (α) were used: 0.10, 0.20 and 0.30.

Table 3: Number of MAX-SAT instances where any of the algorithms statistically outperforms the other or the two are similar. The last two columns report the runtime of APX and PX.

		DRILS performance			Runtime (μs)	
Instances	α	APX	PX	Sim.	APX	PX
Unweighted	0.10	78	1	81	463	454
	0.20	82	2	75	684	729
	0.30	85	2	73	849	1 060
Weighted	0.10	26	19	87	1 425	882
	0.20	49	14	69	1 859	1 416
	0.30	77	5	50	2 365	1 713

DRILS+APX seems to be better in the unweighted instances than in the weighted ones, compared to DRILS. Unweighted instances are expected to have more plateaus than weighted ones, and plateaus seem to be problematic for the traditional PX. We also observe that DRILS+APX outperforms DRILS more clearly for higher values of the perturbation factor. The runtime of APX and PX is similar in order of magnitude (hundreds of microseconds), but it is higher in the case of APX for the weighted instances. This is an indication that the work spent in the analysis of articulation points is not useful most of the time for these instances. In the unweighted instances, the runtime of APX is lower than that of PX for a high value of the perturbation factor.

5 CONCLUSIONS

We propose an improved version of Partition Crossover, Articulation Points Partition Crossover (APX). This new operator increases the number of explored solutions in an exponential factor with just a small constant increment in computational time. The core idea of APX is to flip variables in the parent solutions that are articulation points in the recombination graph. As a result, the number of connected components increases, and the variables in the new components can be selected from one of the parents independently of the other components. Empirical results on both Random NKQ Landscapes and MAX-SAT provide evidence that the new APX operator increases the performance of a recent state-of-the-art search Gray-Box algorithm for pseudo-Boolean optimization.

Future work on APX includes a detailed analysis of the possibility of flipping more than one articulation point per connected component. We have included APX in one particular algorithm (DRILS), but the operator is independent of the algorithm and can be included in GAs. Regarding DRILS, we can use the information of the recombination graph and articulation points to guide the random walk after finding a local optimum. In particular, this guide could be essential in plateaus, a scenario for which preliminary theoretical results on APX provide an encouraging message.

ACKNOWLEDGEMENTS

Funding was provided by the Fulbright program, the Spanish Ministry of Education, Culture and Sport (CAS12/00274), the Spanish Ministry of Economy and Competitiveness and FEDER (TIN2014-57341-R and TIN2017-88213-R), the University of Málaga, Andalucía Tech, the Air Force Office of Scientific Research, (FA9550-11-1-0088), the Leverhulme Trust (RPG-2015-395), the FAPESP (2015/06462-1) and CNPq (304400/2014-9).

REFERENCES

[1] Francisco Chicano, Darrell Whitley, Gabriela Ochoa, and Renato Tinós. 2017. Optimizing one million variable NK landscapes by hybridizing deterministic recombination and local search. In *Genetic and Evolutionary Computation Conference, GECCO 2017*. 753–760. https://doi.org/10.1145/3071178.3071285

[2] Francisco Chicano, Darrell Whitley, and Andrew M. Sutton. 2014. Efficient identification of improving moves in a ball for pseudo-boolean problems. In *Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014*, Dirk V. Arnold (Ed.). ACM, ACM, NY, USA, 437–444. https://doi.org/10.1145/2576768.2598304

[3] Robert B. Heckendorn, Soraya Rana, and Darrell Whitley. 1999. Polynomial Time Summary Statistics for a Generalization of MAXSAT. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1 (GECCO'99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 281–288. http://dl.acm.org/citation.cfm?id=2933923.2933952

[4] M. E. J. Newman and Robin Engelhardt. 1998. Effect of neutral selection on the evolution of molecular species. *Proc. R. Soc. London B* (1998), 1333–1338.

[5] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM Journal of Computing* 1, 2 (1972), 146–160.

[6] Renato Tinós, Darrell Whitley, and Francisco Chicano. 2015. Partition Crossover for Pseudo-Boolean Optimization. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII (FOGA '15)*. ACM, New York, NY, USA, 137–149. https://doi.org/10.1145/2725494.2725497

[7] Darrell Whitley, Francisco Chicano, and Brian W. Goldman. 2016. Gray Box Optimization for Mk Landscapes (NK Landscapes and MAX-kSAT). *Evolutionary Computation* 24 (Jan-09-2016 2016), 491 – 519. https://doi.org/10.1162/EVCO_a_00184

[8] Darrell Whitley, Doug Hains, and Adele Howe. 2009. Tunneling Between Optima: Partition Crossover for the Traveling Salesman Problem. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO '09)*. ACM, New York, NY, USA, 915–922. https://doi.org/10.1145/1569901.1570026

[9] Alden Wright, Richard Thompson, and Jian Zhang. 2000. The computational complexity of NK fitness functions. *IEEE Transactions on Evolutionary Computation* 4, 4 (2000), 373–379.