Identification and Exploitation of Linkage by Means of Alternative Splicing

Philipp Rohlfshagen and John A. Bullinaria

School of Computer Science, University of Birmingham
Edgbaston, Birmingham B15 2TT, United Kingdom
P.Rohlfshagen@cs.bham.ac.uk, J.A.Bullinaria@cs.bham.ac.uk

Summary. Alternative splicing is an important cellular process that allows the expression of a large number of unique cell-specific proteins from the same underlying strand of DNA, and thereby drastically increases the organism's phenotypic plasticity. Its emergence is facilitated by the modular composition of genes into numerous semi-autonomous building blocks. In artificial evolution, such modular composition is usually unknown initially, but once learned may greatly increase the algorithm's efficiency.

In this paper, an abstract interpretation of alternative splicing is presented that emulates some of the properties of its natural counterpart. Two appoaches, both based upon a simple (1+1) evolutionary algorithm, are described and shown to work well on established benchmark problems. The first algorithm, eAS, is designed for cyclical dynamic optimisation problems: it systematically merges the problem variables into groups that capture the properties exhibited by a finite number of successive states and reuses that information when required. The second algorithm, iAS, employs a systematic search to identify a sub-set of variables for which simultaneous inversion affords an increase in fitness. This approach seems particularly useful for problems that have many local optima that are far apart in the search space. Results from a systematic series of experiments highlight the intrinsic attributes of each algorithm, and allow analysis in terms of the identification and exploitation of linkage.

1 Introduction

Evolutionary Algorithms (EAs) are abstract interpretations of biological evolutionary systems in which potential solutions to the problem of interest survive and reproduce according to some measure of their fitness. Repeated application of appropriate genetic operators and selection lead to increasing fitness from one generation to the next. EAs have become a popular choice of algorithm for a diverse range of optimisation problems, especially those where traditional methods tend to fail (e.g., those with highly rugged multi-modal search spaces). Numerous extensions have been suggested over recent years that improve the performance of the canonical framework in a variety of domains. In particular, understanding the identification and exploitation of linkage in these algorithms can lead to more efficient EAs [14].

The concept of linkage originates in genetics and describes the physical relationship between two genetic loci: genes located sufficiently close to one another on the same chromosome will be inherited as a single unit, known as a linkage group, with high probability. As genes are flexible units that have the ability to 'move' around the chromosome, one would expect interdependent genes to 'attract' one another. This concept has been extended to evolutionary computation (EC) where linkage is commonly used to describe the interdependency amongst the variables of a problem. This interdependency means that the optimal state (e.g., 0 or 1) of one variable S_i depends on the state of another variable S_j. Quantifying linkage is difficult in general, but becomes more straightforward if the problem is decomposable. If $S = \{S_1, S_2, \ldots, S_n\}$ describes a set of n variables S_i, the fitness function is said to be an *Additively Decomposable Function* (ADF) if it can be written as a sum of lower-order sub-functions:

$$f(S) = \sum_{i=1}^{m} f_i(S_{V_i}) \tag{1}$$

where m is the number of sub-functions, f_i is the i-th sub-function, and S_{V_i} is the variable sub-set defined by index set V_i. For example, if $V_i = \{1, 2, 3, 5\}$ then $S_{V_i} = \{S_1, S_2, S_3, S_5\}$. The set of

variables specified by V_i is a set of interdependent variables, also known as a *linkage set* or, more commonly, a *building block* (BB). For most real world applications, the linkage sets will overlap, but they still form a useful basis for analysis.

In evolutionary computation we are usually interested in optimising some objective fitness function $\Phi : \mathbb{X} \rightarrow \mathbb{R}$, where $\mathbb{X}$ represents the problem. The mapping from the problem's parameters to the elements in $\mathbb{X}$ is usually static and pre-determined. If no problem specific knowledge is used, the proximity of elements in $\mathbb{X}$ will not necessarily reflect the relationship between the parameters of the problem. Consequently, genetic operators acting on $\mathbb{X}$ are likely to be disruptive. The identification of linkage groups may allow this disruption to be limited, and hence not only enhances the algorithm's performance, but also leads to more scalable algorithms. This idea has already been exploited in numerous different ways, some notable examples being the Messy GA and Fast Messy GA [15, 17], and the Linkage Learning Algorithm [21], which attempt to evolve appropriate representations that capture the relationship between the variables of the problem. Another approach has been to use statistical models known as Estimation of Distribution Algorithms (EDAs), such as the Compact and Extended Compact GA [22, 23]. Finally, another class of approaches uses perturbations to detect dependencies amongst variables, such as in the Gene Expression Messy GA [29].

In this chapter, we study the identification and utilization of linkage in two EAs inspired by the cellular process of alternative splicing (AS). AS is a post-transcriptional process that occurs in the majority of cells of higher eukaryotes and is partly responsible for the proteomic diversity observed in these organisms. The dominant property of this process is the ability to express meaningful sub-sequences of the underlying encoding (DNA in this case) alternatively. In nature, this ability is facilitated by the occurrence of readily available BBs, known as exons and introns. Exons (and, as will be shown in section 2, introns) may be viewed as a tightly linked BBs that, by themselves, contain meaningful information about the search space (in effect, sub-solutions). In artificial systems, the structure of linkage sets is rarely available from the onset and needs to be identified during the execution of the algorithm. Therefore, in order to implement the mechanisms of AS, considerable effort has to be directed at the identification of modules in the search space.

The first algorithm proposed here, called explicit Alternative Splicing (eAS), is an implicit memory approach for cyclic dynamic environments. It is an extension of an algorithm we presented previously [37]. This algorithm utilises a single encoding that represents a tree of variable depth; and each new state encountered results in a new level being added to that tree. The problem variables are systematically placed into nodes of the tree that correspond to their value across all states encountered thus far. Special care is taken to prevent the genetic operators disturbing the acquired information. In other words, throughout the execution of eAS, the memory of previous states is not acquired and stored directly, but instead, variables common to sub-sets of states are identified and brought together within a single node. This leads to a very concise representation of the search space and allows the algorithm to successfully identify and preserve common (shared) genetic material across a succession of different environmental states when tested on a simple dynamic problem.

The second algorithm, called implicit Alternative Splicing (iAS), is proposed for binary encodings and relies upon a top-down search to identify sub-sets of binary variables that may be inverted successfully in their entirety. EAs often suffer from entrapment in local optima from which an escape is unlikely due to a general loss of diversity (i.e. premature convergence). iAS attempts to circumvent this issue by using large-scale perturbations of the encoding that are incrementally refined. In other words, iAS tries to perturb the underlying encoding without disrupting any of the tightly linked BBs discovered so far.

The remainder of this chapter is structured as follows: first, the process of AS in biological systems is presented in section 2, followed by a literature review of implementations that resemble AS in EC (section 3). The first algorithm, eAS, is outlined in section 4, including an overview of dynamic optimisation, a description of the algorithm, the experimental setup and results. Section 5 presents the second algorithm, iAS, following a similar structure to the previous section. Finally, the chapter is concluded in section 6 with a summary of the two approaches and a brief discussion of prospects for future work in this area. Throughout, special emphasis is given to the underlying concept of linkage.

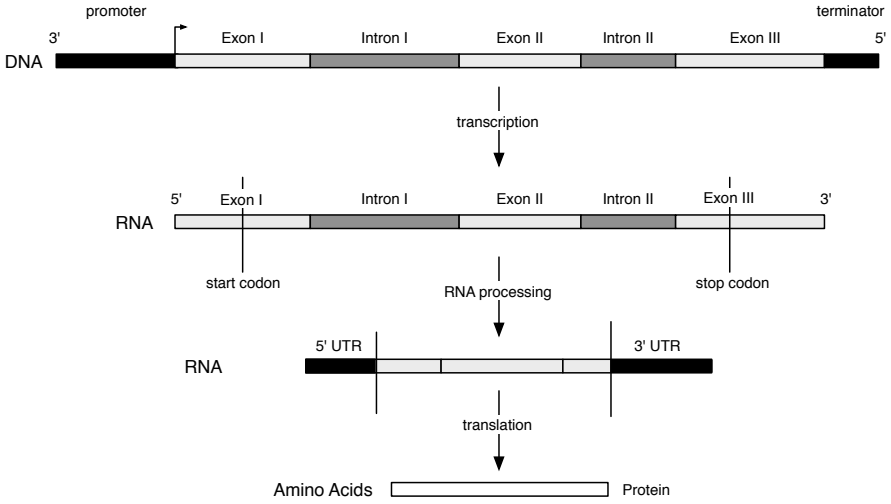


Fig. 1: Protein synthesis of an eukaryotic gene: DNA is transcribed to RNA, introns are spliced (RNA processing) and the resulting strand is translated to a polypeptide of amino acids (a protein).

2 Alternative Splicing in Nature

The physical carrier of genetic information in almost all living organisms is DeoxyriboNucleic Acid (DNA). The DNA of an organism contains all the information required (viewed in the context of the cell) to allow the development of all bodily functions. The DNA of multi-cellular eukaryotes typically contains multiple different regions that are demarcated using well specified sequences of nucleotides. In general, regions that have a dedicated function are known as genes. In classical genetics, a gene is a conceptual entity, a hereditary unit that determines or influences a certain physical characteristic. In molecular genetics, a gene is a physical entity, a well defined strand of DNA that contains instructions to synthesise a protein or other functional constructs. In prokaryotes (bacteria), a gene exclusively contains instructions to fulfill its purpose (e.g., synthesise a protein). In higher eukaryotes (plants and animals), on the other hand, genes usually contain non-coding regions that do not directly contribute towards the final protein product. These interrupted genes are composed of short, coding sequences called exons, and longer, non-coding sequences known as introns (see figure 1).

Prokaryotes usually have very condensed DNA that allows highly efficient replication. The DNA of eukaryotes, on the other hand, may contain numerous structural and regulatory elements that may affect gene expression: after transcription, but before translation, an intervening processing step is required to remove non-coding regions that do not contribute towards the protein product. This is known as RNA processing, and physically removes introns from the RNA before it is translated into a protein. Every gene is made up of at least one exon and always starts and ends with an exon, independent of the total number of exons and introns. Exons and introns always occur in an alternating fashion, with the introns usually occupying far larger regions of the gene than exons.

The classical pathway of gene expression removes and discards all introns, but AS may affect the splicing event so that introns may be retained or exons may be skipped. This may lead to numerous different transcripts from a single template. AS frequently takes place in the human genome and is now recognised as being one of the most fundamental sources of proteomic complexity. Extensive reviews of the mechanisms and developmental consequences of AS already exist [3, 34]. AS events are thought to be regulated by factors such as cell type or developmental stage [32] and occur in an estimated 60% of all genes in the human genome [24, 27]. It is therefore an important factor in accounting for the discrepancy between the size of the human genome and proteome. The five main splicing events are depicted and described in figure 2. The importance of AS is best illustrated by

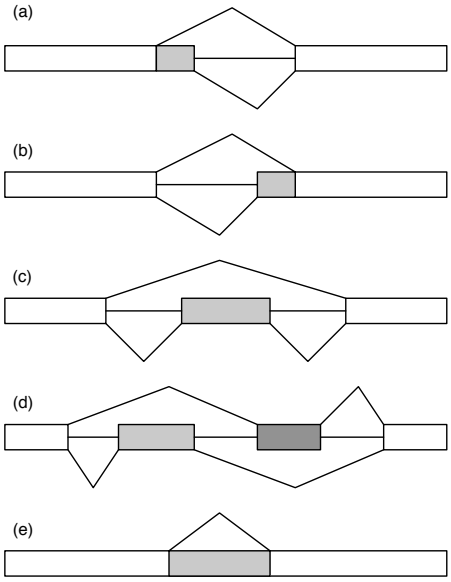


Fig. 2: The five most common forms of AS (the boxes resemble exons, the horizontal lines correspond to introns): (a) alternative 5' site, (b) alternative 3' site, (c) cassette exon, (d) mutually exclusive exons, and (e) retained intron. Adopted from [18].

the fact that AS determines the sex in *Drosophila melanogaster*: the female splice variant includes exon 4 and the male ones does not [2].

The following scenario provides a good illustration of an effect of AS that is highly relevant to the field of EC: AS is often associated with exon tandem duplication events [31]. Any exon that is tandem duplicated (that is, has an exact copy placed in parallel) may be regulated as an alternative element at first, and hence it does not interfere with the existing exons. The new exon is expressed only in a small fraction of transcripts (minor form), meaning that the original gene expression is largely preserved (major form). This effectively removes the selection pressure from the new exon and creates a neutral or near-neutral region which is free to accumulate mutations. If the minor form should yield an improvement in fitness, positive selection pressure will favour it, and it will subsequently increase in frequency. The alternative pathway then eventually becomes the major form, or a tissue-specific expression (see [35]). In other words, the temporary suspension of selection pressure caused by AS may allow the gene to escape local optima in the fitness landscape.

An interesting perspective is offered by Herbet and Rich [25] who classify prokaryotes as 'hard wired' because they usually consist of a single chromosome containing almost exclusively protein coding material. DNA therefore serves as a true template which is translated faithfully into proteins without the need for significant modification. These attributes are very much evident in current EAs. Eucaryotes, on the other hand, are 'soft-wired' as they use relatively little content of their genome, yet result in a far more complex translation by means of post-transcriptional regulation. Here a single template may be used to create vast numbers of similar or even significantly distinct proteins. We believe these insights from genetics may have a significant impact on the development of novel EAs.

3 Alternative Splicing in Evolutionary Computation

The majority of EAs, and Genetic Algorithms (GAs) in particular, were originally inspired by the field of population genetics, and most notably Fisher's genetical theory of natural selection [11]. In recent years, however, advances in molecular genetics, including the genome sequencing projects, have triggered an interest in abstractions that are more directly inspired by the biochemical information processing architecture of organic cells. For example, properties of the genetic code

have been exploited successfully by Karuptga and Gosh [28], and a simple implementation of RNA editing, a post-transcriptional process that selectively modifies individual nucleotides, has been presented by Huang and Rocha [26].

It appears that there have been no direct implementations of AS in the literature other than our own work on this subject [37, 38]. However, there have been sudies that explore the phenomenon of 'alternative expressions', one of the earliest being Levenick's Swappers [33]. Swappers are very simple encodings that consist of two parts, one of which is expressive (active) at any one time. This is somewhat similar to a dynamic exon-intron structure, and Levenick showed how such dynamic expression may be useful in accelerating the algorithm's rate of adaptation. A more elaborate and (more importantly) adaptive approach is the structured GA due to Dasgupta and McGregor [9] which uses a control sequence of meta-bits that determines the regions of the encoding to be expressed. Only one meta-bit (and thus one region) may be active at any one time. This encoding was proposed particularly to deal with cyclic environments: whenever a new environment is encountered, the control sequence is expected to express the part of the genome that implicitly stores that particular state.

Similarly, Collard et al. [8] proposed the Dual GA (DGA) which uses a standard binary encoding, but with an extra bit to determine whether the encoding is expressed as its dual, where the dual is defined simply as the inverse of the binary encoding. This meta-bit is, like the rest of the encoding, subject to crossover and mutation and thus adaptive. This work was extended by Gaspar et al. [12] to include multiple meta-bits that control different sections of the encoding: the Folding GA has several meta-genes that determine the state of all subsequent bits in the encoding until the next meta-gene. This encoding has been tested successfully in dynamic domains [13] and is discussed in further detail in section 5.1. Yang extends these concepts further and proposes a primal-dual encoding for use in static [41] and dynamic [40] environments. Here, primal chromosomes are defined as those individuals currently in the population, and a selection scheme is used that considers the individuals of least fitness to be chosen for dual mappings: whenever the dual produces a greater fitness than its primal counterpart, the primal is replaced by the dual. There are numerous further studies dealing with the concept of diploid or poly-ploid encodings, though that work is generally motivated by the concept of multiple alleles rather than the expression of alternative segments at the molecular level.

We know of no further work that explicitly addresses the utility of AS or closely related concepts in artificial evolution. This may be due to the fact that AS has only recently gained increasing attention due to a better understanding (from the recent genome sequencing projects in particular). Finally, it should be noted that the concept of linkage used here differs slightly from its generally accepted meaning in genetics, but not necessarily from its usual meaning in EC. In general, the design of EAs in EC follow the principles of population genetics, attempting to obtain a favourable distribution of alleles by means of crossover and mutation under selection pressure. The two algorithms presented in the remainder of this chapter assume that the underlying encoding represents a strand of DNA (or, to be more precise, a single gene) and not a series of genes as found on a chromosome. Thus the linkage groups or sub-structures do not correspond to groups of genes, but groups of nucleotides / amino acids (i.e. exons and introns). This is a vital distinction from a biological perspective, but irrelevant given the definition of linkage in EC (outlined above) where we are only interested in the interdependencies of the variables, independently of how they have been represented conceptually.

4 Explicit Alternative Splicing

Historically, EAs have been applied mainly to classes of static optimisation problems. Many problems are, however, dynamic in the real world, and it is thus not surprising to see increasing efforts directed towards Dynamic Optimisation Problems (DOPs) [5]. A DOP, simply stated, is a problem that changes over time $t = 1, 2, \ldots$. More formally:

$$F(\mathbf{X}, t) = f_t(\mathbf{X}) \text{ where } f_t(\mathbf{X}) = D(f_{t-1}(\mathbf{X}), t-1) \tag{2}$$

and where $D(f_i(\mathbf{X}), t)$ encapsulates the dynamics of the problem. The reason for this recent interest is straightforward: problems of great complexity are either impossible or very costly to solve. It

follows that in changing environments one should attempt to utilise the best solutions found so far to guide adaption to the shifted problem. This should in general be more efficient than a complete restart of the algorithm, which would be the simplest technique to deal with dynamics. In fact, such a brute force approach is essentially identical to static optimisation, except that the number of function evaluations (FEs) allowed is restricted by the dynamics of the search space. A more efficient approach is to increase the diversity of the population (e.g., via hyper-mutations [7] or random immigrants [19]), either throughout the execution of the algorithm, or whenever an environmental change has been detected. Another useful approach is to have multiple populations that keep track of the optima encountered in the past, and make use of them as appropriate. Alternatively, a central population could keep track of the overall search, while multiple smaller populations diverge to track promising regions in the search space [6]. Finally, the most recent trend attempts to exploit the concepts of anticipation and prediction, which is particularly useful if the current solution or action affects the dynamics of the search space (i.e. there is time-linkage [4]).

Most relevant to this work, however, is the use of memory which has been implemented either implicitly or explicitly. In the former case, the memory is embedded in the encoding, usually in the form of diploidy (e.g., [16]) or polyploidy (e.g., [20]). This approach has been used successfully for small numbers of distinct states, but the space requirements and the memory's loss of integrity over time prevent this technique from scaling to larger numbers of states. Subsequently, more research has focused on explicit memory schemes which attempt to maintain a diverse register of previously good solutions. In cases where the environment returns to a point similar or identical to a previously visited one, solutions from the register may be reused efficiently.

In general, the dynamics of a problem may be characterised by their magnitude and frequency. Here we only consider dynamics that do not alter the actual structure of the search space, but only shift the search space by some distance. Then the magnitude ρ corresponds to the distance between the global optima at times t and $t+1$. A value of $\rho = 0.2$, for example, means the global optimum has shifted a distance of $0.2n$ where n is the size of the problem. The frequency of change, τ, describes, in FEs, how often a change occurs.

4.1 Pseudo Rhythm

Prior to describing our proposed algorithm, it is important to stress a particular property of random (non-cyclical) DOPs. Randomly changing environments do not follow a strict rhythmic pattern. However, it is possible to formalise a notion of rhythm for acyclic environments using the principle of pseudo-periodicity [10], which was originally defined for pseudo-rhythmic random Boolean networks. Pseudo-periodicity allows one to estimate the correlation amongst a succession of M states. The correlation between any two binary states $\mathbf{X}(t)$ and $\mathbf{X}(t')$ at times t and t' is defined as

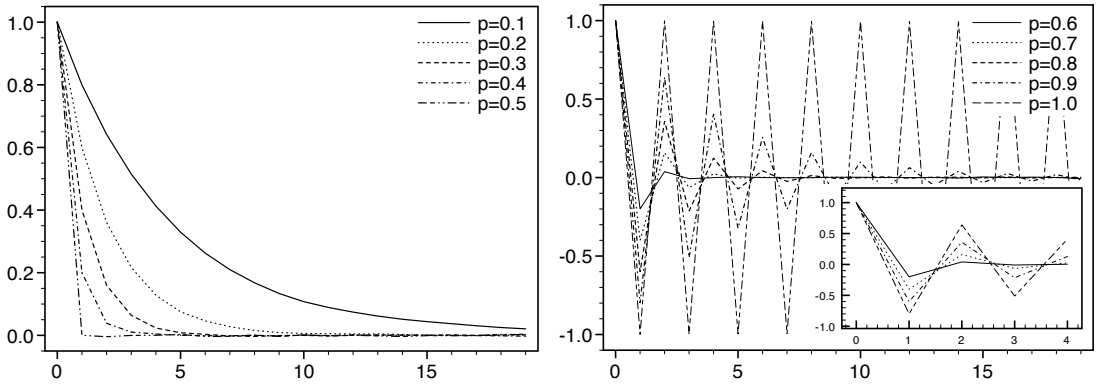
$$C(t,t') = \frac{1}{N}\sum_{i=1}^{N} X_i^*(t)X_i^*(t') \tag{3}$$

where $X_i^*(t)$ is the mapping of $X_i(t)$ onto $[-1,1]$. The overall estimate for a succession of states is then given by the auto-correlation

$$AC(k) = \frac{1}{M}\sum_{t=1}^{M} C(t,t+k) \tag{4}$$

for $k = 0,1,2,\ldots$. Further details have been discussed by Di Paolo [10] (also see [39]).

In order to explore the rhythmic activity of randomly changing environments, we generated a succession of 2000 binary states using the framework described in section 4.3 using different degrees of change ($\rho \in \{0.1, 0.2, \ldots, 1\}$) between successive states. The results of this basic analysis are shown in figure 3: there is no correlation amongst states for magnitudes of change equal to or below 0.5. However, once the magnitude of change affects the majority of bits, a rhythmic pattern emerges. This trend increases as the magnitude of change approaches a value of 1 (which corresponds to a cyclic environment). This implies that memory approaches may not only work well for strictly cyclical domains, but also for randomly changing domains where the change between successive states is sufficiently large. This property is investigated further in section 4.4.



(a) (b)

Fig. 3: Pseudo-rhythm of randomly changing environments: the y-axis shows the degree of correlation amongst states, the x-axis corresponds to the i-th successive state: (a) changes of magnitude 0.1-0.5; (b) changes of magnitude of 0.6-1.0. The inlay in figure (b) shows a magnification of the period up to the fourth successive state, excluding $\rho = 1.0$.

4.2 eAS: the Algorithm

The general idea of eAS is to have a single encoding that may be expressed in numerous different ways. This technique is essentially an implicit memory approach, because almost all the information required to reconstruct previously visited states is stored within a single encoding. The most significant difference from other implicit memory approaches is the way in which the memory is constructed and reused. The states visited are not stored in their entirety, but rather the algorithm attempts to capture the relatedness of a succession of states. The algorithm presented here is a refinement of an algorithm we presented earlier [37]. For the sake of completeness, we will briefly describe the original algorithm, eAS I, first.

eAS I

This algorithm combines the principles of explicit and implicit memory to represent multiple states concisely within a single non-redundant encoding. The encoding consists of two parts, a memory of p splicing patterns $\mathbf{Y} \in \{0,1\}^{p \times q}$ and a virtual gene $\mathbf{X} \in \{1,2,\ldots,q\}^n$ which is divided into q segments. The splicing patterns control which segments of the virtual gene contribute towards the phenotype. There are p such splicing patterns, each of length q, only one of which is active at any one time (with the index of the active splice denoted by σ). Furthermore, inactive splicing patterns are shielded from mutation (explicit memory) while the active pattern is mutated with low probability using a standard binary mutation operator. The problem variables are able to 'move' between segments by means of a mutation operator that simply places a variable from one segment into a randomly chosen one: the mutation operator considers every element in $\mathbf{X}$ and mutates element i with probability p_m by reassigning the value of X_i to a randomly chosen one from $\{1,2,\ldots,X_i-1,X_i+1,\ldots,q\}$. All variables within a segment that is 'expressed' are assigned a value of 1, all other variables are set to 0. A segment j is expressed if $Y_{\sigma j} = 1$.

The following simple example for $n = 10$, $p = 4$ and $q = 5$ will illustrate this. Row 1 shows the encoding (segment numbers for each variable) and rows 2-5 show the (active) splicing patterns on the left and the resultant phenotype on the right.

$$
\begin{aligned}
&(2,3,0,0,0,1,3,3,3,4)\\
(0_0,0_1,0_2,0_3,0_4) &\therefore (0,0,0,0,0,0,0,0,0,0)\\
(0_0,1_1,1_2,0_3,1_4) &\therefore (1,0,0,0,0,1,0,0,0,1)\\
(0_0,0_1,1_2,1_3,1_4) &\therefore (1,1,0,0,0,0,1,1,1,1)\\
(1_0,1_1,1_2,1_3,1_4) &\therefore (1,1,1,1,1,1,1,1,1,1)
\end{aligned}
$$

Whenever a change in the environment occurs, all p splicing patterns are evaluated and the currently best one (as judged by the fitness of the resulting phenotypes) is activated. Such an approach effectively allows one to view cyclic dynamic optimisation problems as static. There is one globally optimal solution that solves the problem independent of its current state and, once that solution is found, no further adaptation is required unless noise and uncertainty is introduced into the system. The encoding bears some noticeable similarity to the Messy GA [15] and Linkage Learning Algorithms [21]. However, there is no under- or over-specification and no crossover. The latter is partly substituted by the use of splicing patterns which have a similar effect as they affect multiple variables simultaneously.

The problem with this approach is that one needs to specify the number of segments and splices in advance, which might pose a problem: the number of splices corresponds to the number of unique states in the environment and the number of segments should be chosen according to the relatedness of the states encountered. This information is not usually available a priori. Nevertheless, we confirmed empirically that approximations for p and q are sufficient for our algorithm to significantly outperform a simple GA, as well as a GA with hyper-mutations or random immigrants, in certain types of dynamics for the problems considered [37].

eAS II

The algorithm presented now was developed to improve upon the previous algorithm and to allow for a more simplified application. The most significant difference is the elimination of the splicing patterns, and the number of segments required no longer has to be set a priori. Now the encoding consists of a single vector $\mathbf{X} \in \{0, 1, \ldots, 2^d - 1\}^n$ where $d \geq 1$ is determined dynamically throughout the execution of the algorithm, and has an initial value of $d = 1$. This encoding, with fitness $\psi = \Phi(\mathbf{X})$, essentially represents a tree. At first, when the algorithm is initiated, all variables may belong to either one of two groups, one group for those variables equal to 1, and the other for variables equal to 0. The integer encoding $\mathbf{X}$ is translated to a binary vector $\mathbf{Y} \in \{0, 1\}^n$ by:

$$\text{decode}(\mathbf{X})\text{: } Y_i = \lfloor X_i / 2^{d-d_c} \rfloor \bmod 2, \text{ for } i = 1, 2, \ldots, n \tag{5}$$

where $d_c \leq d$ is the level of the tree to be expressed. At this stage, the encoding is identical to the classical binary one. Once a change in the environment is detected, the number of groups is expanded by a factor of 2 and the variable d is incremented by 1. The four groups now class the variables according to variables that are 1 or 0 in both states encountered so far, and variables that are either 1 or 0 in one state but not the other. Figure 4 shows how the tree is incrementally built up: whenever a change occurs, all d levels of the current tree are decoded and evaluated. These scores are stored as $\boldsymbol{\phi}^n$ and are compared against a register, $\boldsymbol{\phi}^a$, that contains the scores of each expression when it was last active. The expression whose new score matches its old score is made active. The active expression, d_c, is corresponds to the level i of the tree for which $\boldsymbol{\phi}_i^n = \boldsymbol{\phi}_i^a$. If no match can be found, another level is added to the tree (expand) with a random assignment of variables. If there are multiple matches, one is chosen at random. It follows that a new level is only added to the tree if a new state is encountered. Otherwise, the (hopefully) correct expression is reused (see section 4.2). An upper limit d_{max} is used to prevent the tree from growing indefinitely.

The mutation operator, like all the other operators acting on the encoding, ensures that only the active expression is affected by mutation (i.e. all other expressions are unaffected and thus preserved). The mutation operator considers every single element in $\mathbf{X}$ and mutates it with probability p_m to a new value as follows:

$$X_i \leftarrow \begin{cases} X_i + 2^{d-d_c} & \text{if } X_i / 2^{d-d_c} \bmod 2 = 0 \\ X_i - 2^{d-d_c} & \text{otherwise} \end{cases} \tag{6}$$

Further details of this algorithm may be found in the pseudo-code (algorithm 1). An example of the encoding adapting to a two state cyclical *oneMax* with optima $\mathbf{1}$ and $\mathbf{0}$ illustrates it:

$$\begin{aligned}
\mathbf{X}(t) &= (0,1,1,1,1,1,0,1,0,1) \therefore \mathbf{Y} = (0,1,1,1,1,1,0,1,0,1),\ d = 1,\ d_c = 1 \\
\mathbf{X}(t+1) &= (0,2,2,3,2,2,1,2,0,3) \therefore \mathbf{Y} = (0,0,0,1,0,0,1,0,0,1),\ d = 2,\ d_c = 2 \\
\mathbf{X}(t+2) &= (2,2,2,3,2,2,3,2,2,3) \therefore \mathbf{Y} = (1,1,1,1,1,1,1,1,1,1),\ d = 2,\ d_c = 1 \\
\mathbf{X}(t+3) &= (2,2,2,2,2,2,2,2,2,2) \therefore \mathbf{Y} = (0,0,0,0,0,0,0,0,0,0),\ d = 2,\ d_c = 2
\end{aligned}$$

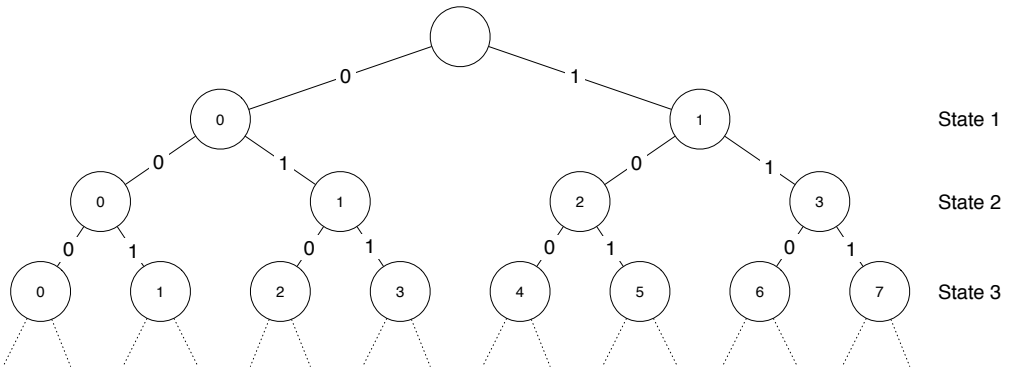


Fig. 4: The internal memory structure of eAS: each layer corresponds to a unique state encountered. The problem variables are sorted according to their state ($\{0, 1\}$) such that there is a strict and easily exploitable ordering. All variables in segment 4 of state 3, for example, are expressed as 1 in state 1, 0 in state 2 and 0 in state 3.

The encoding is shown at four different environmental stages where $f_t(\mathbf{X}) = f_{t+2}(\mathbf{X})$ and $f_{t+1}(\mathbf{X}) = f_{t+3}(\mathbf{X})$. The adaptation of one binary encoding leaves the other binary encoding intact and the encoding eventually settles into the correct state. Some notable advantages of this algorithm are:

- concise representation of multiple states
- operators (decode, mutate, expand, reduce) all operate in linear time ($O(n)$)
- memory grows as required

On the other hand, it is clear that the growth of the tree places a limit on the number of states that may be memorised: at any time, 2^y nodes are required to describe y states and it is clear that at least $2^y - n$ nodes do not actually contain any variables. It should be possible to use an actual tree that is constructed on the fly and where empty nodes are pruned as necessary. The disadvantage of this is that the operators would require more time to manipulate the data structure and an explicit memory structure might be a better option.

Finally, note that there are two fundamental assumptions underlying the design of this algorithm: first, that the algorithm is told when change has occurred, and second, that the algorithm is able to reliably identify previously encountered states. These issues are addressed next.

State Detection and Identification

In general, and in true black-box fashion, any EA is required to detect changes in the environment to ensure that not only an appropriate action is taken, but also that no expired fitness values are used. It is possible to divide the task of dynamic optimisation into three fundamental aspects as shown in figure 5. Here we assume the algorithm is told when a change occurs. Nevertheless, it would be relatively simple to detect change within the problems considered here using a small register of stationary points placed systematically across the search space (see, for example, Morrison ([36]). A periodic re-evaluation of these points should reveal any changes in the environment. In fact, the likelihood of detecting change using a single stationary point is identical to the likelihood that a previously encountered state is correctly identified as such: whenever a change in the environment has been signaled, eAS re-evaluates all d expressions to obtain the current fitness values. The algorithm assumes that a previous state has reoccurred given any of the d expressions produces a fitness value identical to the one produced when that expression was last active. The problem with this approach is that any given expression may produce identical fitness values for different environments and this may mislead the algorithm.

Let us assume that the problem is one of dynamic pattern matching, namely the *oneMax* problem with a dynamically changing target. Given all types of change, any target pattern can

Algorithm 1 Pseudo-code for the part of eAS that deals with changes in the environment.

```
//this method converts the encoding from an integer encoding X to a binary one Y
decode(X) : for i = 1,2,...,n : Y_i = ⌊X_i/2^{d-d_c}⌋ mod 2

//this method adds another level to the tree (to account for a novel state encountered)
expand(X) : for i = 1,2,...,n : X_i ← { 2X_i     if rand < 0.5
                                      { 2X_i + 1 otherwise      , d ← d + 1

//eliminates the lowest level of the tree if instead an existing level is to be reused
reduce(X) : for i = 1,2,...,n : X_i ← { X_i/2         if X_i mod 2 = 0
                                      { (X_i - 1)/2   otherwise        , d ← d - 1

//act upon change
if change detected then
  φ^n ← Φ(decode(X, i) for i = 1,2,...,d //evaluate all expressions and store them in φ^n
  φ^a_{d_c} = ψ //update the fitness value of the splice active just prior change
  d_c = -1 //reset the pointer to the currently active splice

  //check if the new state has been encountered before
  for i = 1,2,...,d do
    if φ^a_i = φ^n_i then
      d_c ← i
    end if
  end for

  //The state encountered has not been encountered previously
  if d_c = -1 then
    if d < d_max then
      expand(X)
      //test only if environment is acyclic, else this statement is always true
      if Φ(decode(X, d)) > max(φ^n) then
        d_c ← d
      else
        reduce(X)
        d_c ← index of max(φ^n)
      end if
    else
      d_c ← index of max(φ^n)
    end if
  end if
end if
```

change into any of $2^n - 1$ different patterns. In this simple scenario, the fitness of any encoding equals the Hamming distance δ between the encoding and the target pattern. In general, the probability that a new state encountered is indeed a state encountered previously is

$$p(\delta) = 1 - \frac{\left(\frac{n!}{\delta!(n-\delta)!}\right) - 1}{2^n - 1} \tag{7}$$

It is clear that if $\delta = 0$ or $\delta = n$, then $p(\delta) = 1$. Therefore, although there is a certain probability that an expression is reused in the wrong context, the continuous approximation of the expression to the target pattern reduces this probability over time:

$$\lim_{\delta \to 0} p(\delta) = 1 \tag{8}$$

As mentioned previously, in order to detect change, a small register of static random points may be used. The number of points required in this case depends upon the probability $p(\delta)$ given by

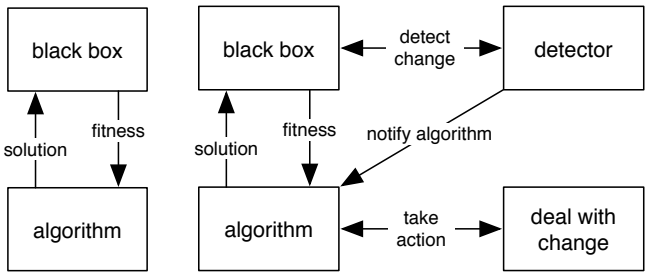


Fig. 5: EAs for DOPs require the ability to detect and react to changes in the fitness function. Two scenarios are shown, the case of static (a) and dynamic (b) optimisation.

equation 7. In order to guarantee the detection of change with probability of 0.995, we require

$$m = \frac{\log(1 - 0.995)}{\log(1 - p(n/2))}$$

random static points (we assume a random point has an expected distance of $n/2$ to the target pattern). In the case of the $n = 100$ *oneMax*, this corresponds to $m \approx 2$ points.

4.3 Experimental Setup

The algorithm has been tested on a well established benchmark problem that allows the modeling of different search space characteristics, some of which are known to be difficult for EAs. This problem is described next, followed by a brief discussion of the experimental settings.

Modular Base Function

This problem, which we shall call a Modular Base Function (MBF), is constructed using BBs of 4 bits, each of which contribute equally towards the total fitness according to some fitness vector. In this case, 25 BBs are used to construct a problem of size $n = 100$. The fitness value of any encoding is then given by:

$$\sum_{i=1}^{25} \xi_{x(i)} \quad \text{where} \quad x(i) = \sum_{j=1}^{4} X_{4i+j} \tag{9}$$

in which $\xi_{x(i)}$ returns an integer value according to the fitness vector $\boldsymbol{\xi}$. Here we use

$$\boldsymbol{\xi} \leftarrow \begin{cases} (0,1,2,3,4) \\ (0,2,2,4,4) \\ (3,2,1,0,4) \end{cases}$$

which correspond to the classical *oneMax* (1), a neutral landscape (2) and a fully deceptive one (3; see [42]). In other words, for each BB, the unitation function indexes the corresponding fitness vector to return a value that contributes towards the overall fitness of an encoding. It is clear that this function is identical to the ADF presented earlier with non-overlapping BBs of size 4 bits.

Dynamics

In order to introduce dynamics, we make use of the dynamic benchmark generator proposed by Yang (see [42]) upon which the following description is based. The DOP-generator can construct a dynamic environment from any stationary binary-encoded problem $f(\mathbf{X})$, $\mathbf{X} \in \{0,1\}^n$ using a bitwise exclusive-or ($\oplus$) operation that involves $\mathbf{X}$ and a binary mask $\mathbf{M}(k)$, where $k = \lfloor t/\tau \rfloor$ is

the index of the current environment. The dynamics are generated by performing the exclusive-or on each individual as follows:

$$f(\mathbf{X}, t) = f(\mathbf{X} \oplus \mathbf{M}(k)) \tag{10}$$

The mask $\mathbf{M}$ is incrementally generated using randomly or systematically constructed templates $\mathbf{T}(k)$ each of which contains $\rho \times n$ ones:

$$\mathbf{M}(k) = \mathbf{M}(k-1) \oplus \mathbf{T}(k) \tag{11}$$

The initial mask at time $k = 1$ is $\mathbf{M}(1) = \mathbf{0}$. It is possible to generate cycles of size $2K$ by generating $2K$ masks systematically first. Individuals are subsequently evaluated according to:

$$f(\mathbf{X}, t) = f(\mathbf{X} \oplus \mathbf{M}(k \bmod 2K)) \tag{12}$$

The masks are generated as follows: K binary templates $\mathbf{T}(0), \ldots, \mathbf{T}(K-1)$ are constructed randomly to form a partition of the search space. Each mask contains n/K exclusively selected bits that are assigned a value of 1. The masks are then generated according to:

$$\mathbf{M}(i+1) = \mathbf{M}(1) \oplus \mathbf{T}(i \bmod K), \quad \text{for } i = 0, 1, \ldots, 2K-1 \tag{13}$$

The templates up to $K-1$ are used to construct incrementally the mask $\mathbf{M}(K) = \mathbf{1}$ and are subsequently reused to generate up to $2K$ states where $\mathbf{M}(2K) = \mathbf{M}(0) = \mathbf{0}$. In this case, the number of states, $2K$, dictates the distance among successive states: n/K. Alternatively, the magnitude of change ρ dictates the number of successive states: $K = n/(\rho \times n)$. If, for example, $n = 100$ and $\rho = 0.1$, the cycle has a length of $2K = 20$ states. This is, however, only the upper limit and it is possible to generate smaller cycles by restricting the partition of the search space to a sub-space (i.e. some bits never change). Thus, if a cycle of length 8 is required with a distance between successive states of $\rho = 0.1$, the partition of the search space is simply restricted to $(0.1n * 8)/2$, which, in the case of $n = 100$ is 40 bits. The maximum cycle that may be generated given n is $2n$ where all successive states have a distance of 1.

The performance of the algorithm on this DOP may be calculated as follows:

$$F = \frac{1}{G} \sum_{i=1}^{G} \left(\frac{1}{N} \sum_{j=1}^{N} F_{BOG_{i,j}} \right) \tag{14}$$

where $F_{BOG_{i,j}}$ corresponds to the fitness of the Best Of Generation in generation i and trial j, G is the number of generations, and N the number of trials. As eAS is implemented as a $(1+1)$-EA, the value of F_{BOG} corresponds to the fitness after every 100 FEs.

Settings

The majority of experiments will focus on cyclical domains. However, given the analysis of rhythm in acyclic domains (section 4.1) and the presumably stronger presence of such scenarios in the real world, it is interesting to investigate how eAS would fare in randomly changing environments. The algorithm is tested on the dynamic MBF for all three fitness matrices and a problem of size $n = 100$. For each setting as shown in table 1, we ran the algorithm for 30 times, limiting the duration of each run to 30 cycles each of length c (total number of FEs per run is thus $30c\tau$). For the acyclical experiments, the algorithm is executed for a total of 50 changes (50τ FEs). A summary of parameter settings may be found in table 1. We do not take into account the FEs required to evaluate all expressions after each change. This omission is justified on the grounds that we do not compare eAS to another algorithm and that the FEs required after a change are usually insignificant in regard to the period between environmental changes.

4.4 Results and Analysis

Cyclical Dynamics

The general performance of eAS is summarised in table 2 using as performance measure equation 14. Figure 6 shows how eAS is able to memorise the dynamics relatively independent of the length

Parameter	**Values**
τ	$250^{1,2}, 500^{1,2}, 1000^{1,2}$
ρ	$0.1^{1,2}, 0.2^{1,2}, 0.5^{1,2}, 0.8^{2}, 1.0^{1}$
c^1	$\begin{cases} 4,8,12,16,20 & \text{if } \rho = 0.1 \\ 4,6,8,10 & \text{if } \rho = 0.2 \\ 2,4 & \text{if } \rho = 0.5 \\ 2 & \text{if } \rho = 0.8 \end{cases}$

Table 1: Experimental setup for eAS on cyclical (^{1}) and acylical (^{2}) dynamic environments.

		ρ=0.1					ρ=0.2				ρ=0.5		ρ=1.0
		c=4	c=8	c=12	c=16	c=20	c=4	c=6	c=8	c=10	c=2	c=4	c=2
250	ξ=1	98.51	98.51	98.49	98.50	98.50	98.52	98.52	98.51	98.48	98.54	98.50	98.53
	ξ=2	96.02	96.11	96.10	96.13	96.07	96.12	96.20	96.05	96.11	96.07	96.06	95.99
	ξ=3	77.76	77.91	78.53	78.42	78.59	77.67	77.99	77.93	78.20	77.80	78.09	77.76
500	ξ=1	99.15	99.15	99.15	99.15	99.15	99.15	99.16	99.15	99.15	99.16	99.15	99.19
	ξ=2	98.01	97.92	97.96	97.93	97.95	97.91	97.90	97.87	97.90	97.94	97.87	97.90
	ξ=3	78.72	78.50	78.53	78.42	78.59	78.61	78.46	78.52	78.46	78.25	78.23	78.59
1000	ξ=1	99.57	99.57	99.58	99.58	99.58	99.58	99.58	99.57	99.58	99.58	99.58	99.58
	ξ=2	98.96	98.97	98.97	98.97	98.97	98.97	98.96	98.96	98.95	98.94	98.96	98.92
	ξ=3	78.68	78.78	78.81	78.77	78.72	78.69	78.79	78.70	78.62	78.75	78.80	78.52

Table 2: The performance of eAS on the dynamic *oneMax* for different magnitudes of change ($\rho \in \{0.1, 0.2, 0.5, 1.0\}$) and different durations between successive changes ($\tau \in \{250, 500, 1000\}$). The maximum possible score is 100 in each case.

of the cycle. Only the initial period where the dynamics are 'learned' is affected. The performance for the *oneMax* and the neutral function is very good. In fact, as indicated by the table and the graphs, the average value would approximate the maximum of 100 if executed for sufficiently long. The performance on the deceptive problem is significantly worse, as expected. Nevertheless, the performance is again unperturbed by the length of the cycle. In all cases, the performance of eAS improves given longer periods between changes.

Figure 7 shows the performance of eAS on the neutral and deceptive MBF. The neutral case is similar to the *oneMax* although it takes longer for the algorithm to fully encapsulate the dynamics. The final fitness value obtained in the deceptive case is only locally optimal and oscillations are evident where the algorithm switches between solutions without further adaptation (figure 7 (b) where $FE > 5000$).

It is important to note that the memory only helps the algorithm to deal with the dynamics and not the base function. If other variation operators would have been employed, a better overall performance could be achieved. In fact, it is possible to combine the algorithm presented in the second half of this chapter, iAS, with this memory scheme for improved performance. This will be left for future work.

Acyclical Dynamics

We would expect eAS to do well in acyclic environments if the change is sufficiently large to result in a reliable pseudo-rhythm. However, most changes in the real world are expected to be small on average. The most obvious problem encountered by eAS in this case would be the continuous accumulation of states added to the memory: no state is likely to be repeated within a short period of time and the encoding will consistently add new levels to the tree until the maximum has been reached. This is the reason for the guardian **if** statement as shown in the pseudo-code: a new level is only added to the tree if the corresponding expression, which is random initially, produces a higher fitness value than any of the stored encodings. One would expect continuous optimisation in environments with small change (i.e. trees with a single level) and a mixture of restart and re-use in environments with larger changes (which eventually should adapt a pure re-use strategy).

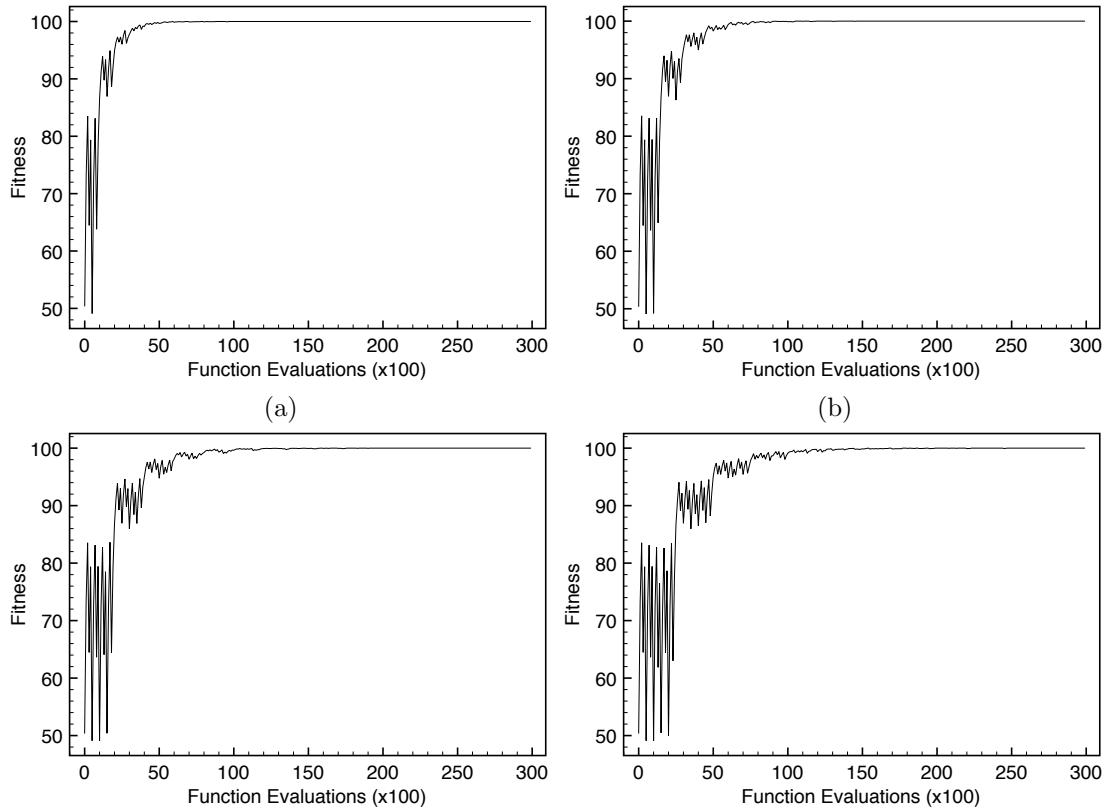


Fig. 6: Performance of eAS on dynamic *oneMax* for $\rho = 0.2$ and 4 (a), 6 (b), 8 (c) and 10 (d) states per cycle.

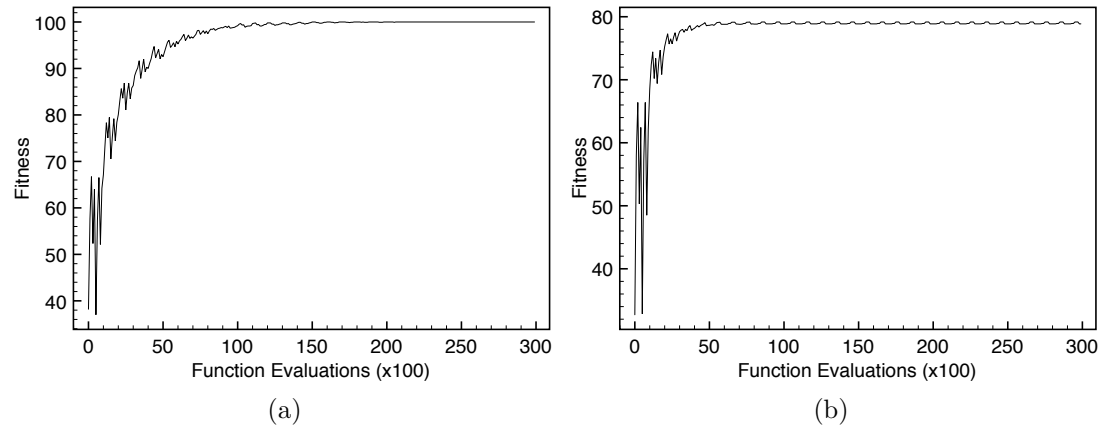


Fig. 7: Performance of eAS on fitness matrices 1 (a) and 2 (b) for a change of $\rho = 0.1$ and a period of $\tau = 250$.

For this experiment, we focus simply on the depth of the tree that evolved given different parameter settings. The results are shown in table 3 and, as expected, the algorithm simply continues to adapt without making use of memory if the changes are very small. Conversely, if the changes are fairly large, memory is used and, more importantly, reused efficiently. A value for $\rho = 0.8$ has an approximate period of 2, which is reflected by the depth of the tree that evolved. Finally, the

	250	500	1000
ξ=1	1.0	1.0	1.0
ρ=0.1 ξ=2	1.0	1.0	1.0
ξ=3	1.0	1.0	1.0
ξ=1	1.0	1.0	1.0
ρ=0.2 ξ=2	1.2	1.0	1.0
ξ=3	1.3	1.0	1.0
ξ=1	9.1	9.0	9.4
ρ=0.5 ξ=2	8.8	8.9	8.2
ξ=3	9.2	9.5	9.4
ξ=1	2.5	2.1	2.0
ρ=0.8 ξ=2	4.6	3.2	2.3
ξ=3	5.5	3.2	2.0

Table 3: Depth of tree constructed in acyclical environments for all fitness matrices and different magnitudes of change $\rho \in \{0.1, 0.2, 0.5, 0.8\}$ and durations $\tau \in \{250, 500, 1000\}$.

most difficult scenario for eAS is a change of magnitude 0.5 in which case there is no rhythm. It is interesting to note that the algorithm converges to a memory of approximately 9 states for $\rho = 0.5$ independent of the fitness vector or period τ. This value may be explained as follows: whenever the environment changes, the new optimum shifts a distance of $1/2n$. A randomly generated encoding is also expected to have a distance to the global optimum of $1/2n$. The initial likelihood that a new level is added to the tree is thus $1/2$. If, however, a new level is added and another change occurs, this likelihood changes to $1/3$ (we assume that all encodings evolved so far have, on average, a distance of $1/2n$ to the new optimum). In general, the probability of adding another level to the tree is thus $1/(d+1)$. If we simulate this happening for 50 times, empirical results confirm (averaged over 1000 trials) the expected value of d to be roughly 9.5.

4.5 Discussion

It is worth pointing out that the concept of linkage in this scenario, and possibly in other dynamic domains as well, differs from its traditional meaning. Here, we are not interested in the structural properties of the base state, but instead we are interested in the structural properties amongst a succession of states. The genetic operators need to be designed to find and preserve useful structures. It has been shown here how an abstract implementation of AS is able to find subsets of bits common to subsets of states, and to reuse that information when required. The encoding therefore provides a very compact representation of the entire dynamic landscape, and the dynamics are effectively removed from the problem as there is now a single static optimal solution. The performance of eAS has been demonstrated in a variety of different experiments, and the results seem to suggest that the ability of eAS to cope with cycles is independent of cycle length, although there is obviously an upper limit as to how many different states may be fully captured. An unsigned 8 byte integer representation (long) could capture cycles with up to 64 states. It is important to note that the suggested encoding does not enhance the algorithm's performance on the base problem (i.e. the static version of the problem), but only enhances the algorithm's ability to cope with the dynamics imposed on top of the base problem. Nevertheless, the suggested methodology may be applied to other algorithms.

5 Implicit Alternative Splicing

The second algorithm, implicit Alternative Splicing (iAS), differs from the previous approach in that BBs are found on-the-fly using a systematic search procedure. Hence this algorithm is referred to as implicit because the encoding itself does not imply any modularity of the search space. It is clear that any artificial equivalent of AS relies upon a proper choice of segmentation and a meaningful definition of 'alternative expression'. In eAS, this choice was dictated naturally by the

succession of states. This algorithm, on the other hand, requires additional definitions. We define an alternative expression of a binary segment simply as the segment that is maximum hamming distance from its source. More specifically, given a binary vector $\mathbf{X} \in \{0,1\}^n$, we define an inversion operator $\perp$ such that $\mathbf{X}^{\perp B}$ means: $X_b = 1 \oplus X_b,\ \forall b \in B \subseteq \{1,2,\ldots,n\}$. In other words, B is a set of indices indicating which bits in $\mathbf{X}$ are to be inverted. If B is the set of all indices in $\mathbf{X}$ we simply write $\mathbf{X}^{\perp}$ and call it the dual of $\mathbf{X}$. Any segment that may be inverted without negatively affecting the fitness of the underlying encoding will be called an exon. Subsequently, iAS attempts to find and invert (i.e. express alternatively) the largest exon possible. This objective is achieved by inverting randomly chosen segments of decreasing size in a top-down fashion, using intermediate fitness values as stepping stones for subsequent inversions (see below, and also [38]).

This abstract implementation essentially reduces iAS to the search of the largest possible binary sequence that may be inverted successfully. The first step of iAS is to create the dual of the encoding with some probability p_d. If the dual has a higher fitness than the original encoding, the dual replaces the original encoding. This is followed by a phase of recursive divisions and inversions: the initial focus is directed at a randomly chosen subset of indices $B \subseteq \{1,2,\ldots,n\}$ that is of size $0 < b_l \leq |B| \leq b_u \leq n$ where b_l and b_u are lower and upper bounds specified by the user. At the beginning of each iteration of iAS, the size of B is chosen randomly within those bounds. As the set of indices B is dynamically reassigned throughout a single iteration of iAS, we explicitly denote the initial set as B^0. The chosen bits are randomly divided into two, equally sized disjoint sets, α and β such that $\alpha \cup \beta = B$ and $\alpha \cap \beta = \emptyset$. Each of those sets is inverted in $\mathbf{X}$, one at a time, to produce two fitness values. This step is repeated q times with different, randomly chosen, partitions of B. If any of the $2q$ fitness values should result in a fitness superior to the original one, the process is terminated and the changes are applied immediately. Otherwise, iAS proceeds by taking the 'path of least resistance' and is executed repeatedly using as guidance those bits the inversion of which caused the least decline in fitness (and may thus contain promising inversions of smaller scale). Those bits are then again randomly divided q times and the above procedure is repeated until a better fitness value is found or the number of bits to be inverted is reduced to 1. If the inversion is neutral in regard to fitness, the process is terminated with some predetermined probability p_n. If q should be larger than the total number of unique divisions,

$$d_{unique} \leftarrow \begin{cases} 0 & \text{if } |B| < 2 \\ 0.5 \times (|B|!/((|B|/2)!)^2) & \text{if } |B| \bmod 2 = 0 \\ 0.5 \times ((|B|+1)!/(((|B|+1)/2)!)^2) & \text{otherwise} \end{cases} \tag{15}$$

an exhaustive search of unique divisions is performed.

The following illustrates the workings of iAS on a simple $n = 10$ *oneMax* problem. Let us assume we have a solution $\mathbf{X} = (1,1,1,1,1,1,0,1,1,0)$ with fitness $\Phi(\mathbf{X}) = 8$. First we generate the dual with probability p_d. The dual is $\mathbf{X}^{\perp} = (0,0,0,0,0,0,1,0,0,1)$, has a fitness of $\Phi(\mathbf{X}^{\perp}) = 2$ and will be ignored. We then randomly generate an initial B such as $B = \{0,2,3,4,6,7,8,9\}$. This subset of variables is then partitioned $q = 2$ times and each partition is used to determine which bits to invert:

$$\begin{aligned} \alpha_1 &= \{0,4,6,8\} \therefore \mathbf{X}^{\perp\alpha_1} = (\mathbf{0},1,1,1,\mathbf{0},1,\mathbf{1},1,\mathbf{0},0) \\ \beta_1 &= \{2,3,7,9\} \therefore \mathbf{X}^{\perp\beta_1} = (1,1,\mathbf{0},\mathbf{0},1,1,0,\mathbf{0},1,\mathbf{1}) \\ \alpha_2 &= \{2,3,4,7\} \therefore \mathbf{X}^{\perp\alpha_2} = (1,1,\mathbf{0},\mathbf{0},\mathbf{0},1,0,\mathbf{0},1,0) \\ \beta_2 &= \{0,6,8,9\} \therefore \mathbf{X}^{\perp\beta_2} = (\mathbf{0},1,1,1,1,1,\mathbf{1},1,\mathbf{0},\mathbf{1}) \end{aligned}$$

In this case, $\mathbf{X}^{\perp\beta_2}$ has a fitness of 8 and the algorithm terminates with probability p_n. If execution continuous, $B \leftarrow \{0,6,8,9\}$ and

$$\begin{aligned} \alpha_3 &= \{0,6\} \therefore \mathbf{X}^{\perp\alpha_3} = (\mathbf{1},1,1,1,1,1,\mathbf{0},1,0,1) \\ \beta_3 &= \{8,9\} \therefore \mathbf{X}^{\perp\beta_3} = (0,1,1,1,1,1,1,1,\mathbf{1},\mathbf{0}) \\ \alpha_4 &= \{0,8\} \therefore \mathbf{X}^{\perp\alpha_4} = (\mathbf{1},1,1,1,1,1,1,1,\mathbf{1},1) \\ \beta_4 &= \{6,9\} \therefore \mathbf{X}^{\perp\beta_4} = (0,1,1,1,1,1,\mathbf{0},1,0,\mathbf{0}) \end{aligned}$$

Again, the algorithm would terminate with probability p_n in cases $\mathbf{X}^{\perp\alpha_3}$ and $\mathbf{X}^{\perp\beta_3}$. If not terminated, the correct solution would have been uncovered in the third iteration of the procedure with $\Phi(\mathbf{X}^{\perp\alpha_4}) = n$. More details may be found in the pseudo-code for this algorithm (see algorithm 2).

Algorithm 2 Pseudo-code for a single iteration of the iAS algorithm.

```
//Testing the dual
if rand([0,1]) ≤ p_d then
  if Φ(X^⊥) > Φ(X) then
    X ← X^⊥
  end if
end if

//Generate initial set of bits considered for partitioning and inversion
randomly generate B ⊆ {1,2,...,n} | 0 < b_l ≤ |B| ≤ b_u ≤ n

//Generate random partitions and inversions
while |B| ≥ 2 ∧ terminate = false do
  if d_unique(B) < q then
    P ← all possible pairs of disjoint partitions (α, β) of B
  else
    P ← q random, equally-sized pairs of disjoint partitions (α, β) of B
  end if

  //Select the bits that produced the best inversion and assign them to B
  B ← P_i | Φ(X^⊥P_i) ≥ Φ(X^⊥P_j), ∀_{j∈P}

  //Apply the best inversion found so far if it improves the fitness of X
  if Φ(X^⊥B)) ≥ Φ(X) then
    if Φ(X^⊥B)) = Φ(X) ∧ rand([0,1]) ≤ p_n then
      X ← X^⊥B
      terminate =true
    else
      X ← X^⊥B
      terminate =true
    end if
  end if
end while
```

The reasons for this particular implementation follow directly from the exposition of AS in section 2: AS allows the emergence of new proteins by temporarily suspending selection pressure for certain splice forms which are subsequently free to accumulate mutations. This accelerated rate of change may ultimately produce a new protein (see [35]). A purely neutral approach is too costly for our purposes (due to the general lack of parallelism) and selection pressure is upheld to a certain degree while allowing a temporary decrease in fitness. The lack of problem specific knowledge implies that there is no evidence of modularity in the search space, and it is impossible to determine a priori what segments should be inverted. iAS thus systematically searches for such a segment and the top-down recursive technique allows for the largest segment to be found. There is no certainty that a successful inversion will be found, but testing q different divisions increases the likelihood of success. It is crucial that the initial number of bits, $|B^0|$, is chosen randomly (within bounds) as the size of all subsequent segments depends on this initial choice (as segments are always halved). The maximum number of FEs processed during a single iteration is:

$$\sum_{i=1}^{\lceil \log_2(n) \rceil} min\{q, d_{unique}(\lceil n/2^i \rceil)\} \tag{16}$$

given that $|B^0| = n$ and $p_d = 0$ and that the algorithm continues until $|B| = 1$. In the case of $n = 100$ and $q = 10$, this would be $2(4q + 3 + 1) = 88$ FEs.

5.1 Comparison to the Folding GA

The ability of iAS to invert numerous bits simultaneously makes the need for crossover redundant and some initial experiments have confirmed that there is indeed no significant advantage to using multiple individuals (we investigated the use of 2 individuals in [38]) in place of the (1+1) EA employed here. It is worth pointing out the importance of choosing a random distribution of bits to be inverted (and not continuous segments), and it is possible to draw a meaningful comparison between n-point crossover and uniform crossover: the effectiveness of n-point crossover depends upon the total length of the BBs (distance between first and last defining bit), while the effectiveness of uniform crossover is only dependent upon the actual number of bits within the BBs. If the bias of uniform crossover is chosen randomly every time the operator is applied, uniform crossover has the same attributes as n-point crossover without the restrictions imposed by the static ordering of variables inherent in the algorithm's encoding.

A similar comparison may be drawn between iAS and the Folding GA (FGA, [12]), the most closely related approach in the literature. The FGA, an extension of the dual GA, includes multiple meta-bits that adaptively control different sections of the encoding. These so-called meta-genes determine the state of all subsequent bits in the encoding up to the next meta-gene. An example taken from [13] is as follows. A transcription step T is applied to the encoding that affects all genes between any two meta-bits $\{\dot{0}, \dot{1}\}$ such that $T(\dot{0}\omega) = \omega$ and $T(\dot{1}\omega) = \bar{\omega}$ where $\bar{\omega}$ is the inverse of ω:

$$\begin{aligned} T([\dot{1}01\dot{0}0]) &= [100] \\ T([10\dot{1}1]) &= [100] \\ T([\dot{1}0\dot{0}0\dot{0}0]) &= [100] \end{aligned}$$

This technique is expected to work well on the fully deceptive MBF as a single mutation to any of the meta-bits allows the simultaneous inversion of multiple bits. However, as the distribution of BBs is generally unknown a priori, the meta-bits have to be inserted adaptively. If the deceptive BBs form consecutive groups of variables, this approach should fare well because only a few meta-bits are required to exert sufficient control over the locally optimal parts of the encoding. If, on the other hand, BBs are distributed randomly across the encoding, the problem may become intractable.

If, for example, the MBF is constructed as follows:

$$((v_1, v_2, v_3, v_4)_1, (v_5, v_6, v_7, v_8)_2, \ldots, (v_{n-3}, v_{n-2}, v_{n-1}, v_n)_{n/4})$$

each meta-bit exerts control over exactly one BB in the worst case scenario (locally and globally solved BBs in alternating fashion). If, on the other hand, the MBF is constructed randomly as follows:

$$((v_5, v_{15}, v_8, v_{10})_1, (v_{12}, v_1, v_{22}, v_{23})_2, \ldots, (v_2, v_{32}, v_{21}, v_{11})_{n/4})$$

a meta-bit may only affect a single variable in the worst case scenario. Thus, in order to overcome deception, four meta-bits have to be inverted simultaneously, which is just as hard as the original problem. The crucial difference between the FGA and iAS is, of course, that the folding GA adapts during the execution of the algorithm and hence does not rely upon the extensive amount of FEs required by iAS to locate meaningful structures in the search space.

5.2 Experimental Setup

Here we present a modification of the basic MBF problem as well as two additional problems on which iAS has been tested. The extended version of MBF is presented first, followed by a discussion of the NK fitness landscape and a brief outline of the multiple knapsack problem. Finally the experimental settings are described.

MBF - Extended Version

Here we extend the basic MBF problem to allow exclusively selected parts of the encoding to refer to different fitness matrices. This allows one to mix different search space attributes within the same problem. An MBF of type 0.33 - 0.33 - 0.33, for example, indicates that the search space is

composed to 1/3 of each of the 3 fitness vectors (in the order: *maxOnes*, neutral and deceptive). A type 1 - 0 - 0 would correspond to the classical *maxOnes* problem. Furthermore, the elements in $\mathbf{X}$ that belong to the same BB are distributed randomly across $\mathbf{X}$ (although there is still no overlap across all BBs).

NK Fitness Landscape

Kauffman developed the NK-fitness landscape model (NK; [30]) to study the effects of genetic interactions (epistasis). Each bit in the encoding contributes towards the encoding's fitness. The contribution depends upon the state of the bit itself and the state of all k bits that are linked to it, and the more the bits are dependant on one another, the more rugged the search space becomes. The fitness values for each bit are generated randomly in the range $[0, 1]$ for each of the 2^{k+1} possible states and are stored in a look-up table (for high values of k, these values may be generated on-the-fly). The final fitness is the average contribution of all bits:

$$F(\mathbf{X}) = \frac{1}{n}\sum_{i=1}^{n} F_i(X_i; X_{i_1}, \ldots, X_{i_k}) \tag{17}$$

where $\{i_1, \ldots, i_k\} \subset \{1, \ldots, i-1, i+1, \ldots, n\}$. For random neighbourhoods, this problem has been shown to be NP-complete for values of $k \geq 2$ (see [1]). This problem is again an example of an ADF where the component functions are defined by the relationship between variables. Unlike the MBF, there is overlap between the linkage groups proportional to k, making this problem increasingly difficult to solve.

Multiple Knapsack Problem

The Multiple Knapsack Problem (MKP) is a widely studied combinatorial optimisation problem that has several direct counterparts in industry, such as the cutting stock problem or resource allocation in distributed systems. Well-established benchmarks for this NP-hard problem are readily available allowing a direct comparison to other approaches in the literature. The MKP is a generalisation of the single knapsack problem: the objective is to fill a series of m knapsacks each of capacity c_j with any number of n items, each with weight W_{ij} and value V_i, in such a way that the combined value of all items is maximized without exceeding any of the knapsack's capacities. More formally, the aim is to maximize the fitness

$$\max\{\sum_{i=1}^{n} V_i X_i\} \quad | \quad \sum_{i=1}^{n} W_{ij} X_i \leq c_j \quad \forall j \tag{18}$$

where $X_i \in \{0, 1\}$. We studied the widely used SAC'94 suite of benchmark MKP problems, which may be found online at `http://elib.zib.de/pub/Packages/mp-testdata/ip/sac94-suite/`. It contains 55 problem instances which range in size from 15-105 objects and 2-30 knapsacks.

Settings

We tested the performance of iAS on three different problems. First, it was applied to the fully deceptive MBF in order to investigate the ability of iAS to reliably identify the BBs of the problem. Secondly, it was tested on a variety of other MBFs and Kauffman's NK fitness landscape. Here a comparison was made against the canonical GA (cGA) using a population size of 100, uniform crossover and tournament selection. The crossover and mutation probabilities were $p_c = 0.8$ and $p_m = 1/n$ respectively. Finally, iAS was tested on several instances of the MKP. In each case, the algorithm was executed 20 times for each problem, with a maximum of 20,000, 50,000 and 100,000 FEs. Whenever the global optimum was found within the limit allowed, the algorithm's run was terminated and the number of FEs required noted. The parameter settings used for all problems are shown in table 4.

Parameter Values	
q	$10^{2,3}$, 20^{1}
$\|B^0\|$	64^{1} 100^{1} $\{2,3,\ldots,100\}^{1}$ $\{50,51,\ldots,100\}^{1,2,3}$
p_n	$0.5^{1,2,3}$
p_d	$0.5^{1,2,3}$

Table 4: Experimental setup for iAS for problems 1 (^{1}), 2 (^{2}) and 3 (^{3}).

	$\|B^0\|=64$			$\|B^0\|=100$			$50<\|B^0\|<100$			$2<\|B^0\|<100$		
q	%	avg	FE	%	avg	FE	%	avg	FE	%	avg	FE
1	0	91.95	-	0	91.35	-	0	91.05	-	0	92.25	-
5	10	98.4	423350	0	93.45	-	0	96.45	-	0	95.7	-
10	70	99.6	330962	0	95.6	-	15	98.2	410294	5	98.05	355555
20	100	100	223895	5	98.25	469648	65	99.6	321232	30	99.05	371357
30	100	100	187486	30	99	330856	75	99.75	291386	70	99.65	368337
40	100	100	90103	80	99.75	350168	90	99.9	257587	90	99.9	290079
50	100	100	109580	80	99.75	329730	85	99.85	275812	65	99.6	278106
60	100	100	107851	80	99.75	336884	90	99.9	273186	95	99.95	258828
70	100	100	99680	65	99.65	347471	90	99.9	213651	95	99.95	248939
80	100	100	107223	60	99.55	320236	90	99.9	275824	70	99.7	293830
90	100	100	130937	85	99.75	330761	85	99.85	262796	75	99.7	337469
100	100	100	110571	55	99.55	354448	90	99.9	259557	65	99.65	255182

Table 5: Results for iAS on fully deceptive problem (BB=4) for different values of $|B^0|$ and q. The % symbol indicates the number of trials solved successfully.

5.3 Results

Results Problem 1: MBF

Table 5 shows the performance of iAS on the fully deceptive MBF: iAS is able to solve this problem if executed for a sufficiently large number of FEs (500,000 in this case). This type of problem may only be solved successfully if the algorithm is able to invert entire BBs instantaneously, and it is evident from table 5 that the choice of $|B^0|$ is crucial: values that are multiples or powers of 4 (the BB size) are most successful (64 in this case is the largest possible initial segment that is a power of 4). It is clear how problem specific knowledge may be employed in the choice of $|B^0|$. This effect is shown more clearly in figure 8 where the frequencies of segment sizes successfully inverted are shown. The highest peak, irrelevant of the initial segment size, corresponds to the BB size. In other words, iAS is able to successfully identify the BBs and invert them to find the global optimum. It is important that the initial segment size is as large as possible to guide the search and it is evident that $50 < |B^0| < 100$ performs better than $2 < |B^0| < 100$. A large initial segment not only increases the frequency of finding large exons, but also increases the overall success rate of finding smaller exons in subsequent processing steps.

It is also evident that the choice of q is crucial as well. The more divisions are tested at each stage, the higher the success rate of finding an exon. On the other hand, high values of q are computationally expensive. There is thus a trade-off between the number of times iAS may be executed and q. In this case, $40 < q < 70$ seems best. It should be stressed that a stochastic approach to exploring different divisions at each stage is the only appropriate choice for deceptive problems. Attempts to find the best possible division using local search, or indeed a GA, have failed because the search for a division seems to be as difficult as the original problem itself. Depending on the problem, however, different (problem-specific) heuristics may be employed.

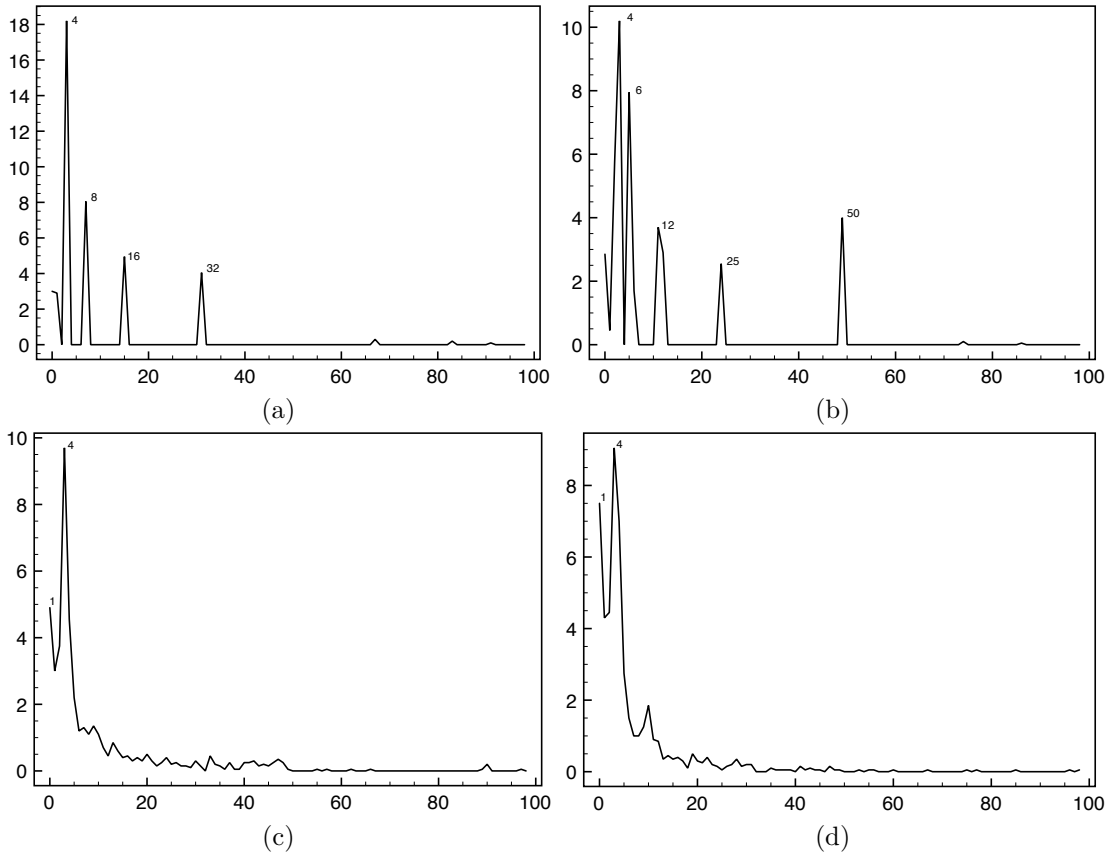


Fig. 8: Results for iAS on fully deceptive problem (BB=4). Frequency of successful inversions (non-successful inversions not shown) for different values of $|B^0|$: 64 (1), 100 (2), $50 < |B^0| < 100$ (3) and $2 < |B^0| < 100$ (4). The x-axes correspond to the number of bits inverted, the y-axes describe the number of successful inversions.

The second set of experiments on the MBF compares iAS to the cGA on a variety of different search space properties as shown in table 6. First, it is evident from the data that iAS, unlike the cGA, improves significantly in performance if given more resources (i.e. more FEs). Secondly, iAS performs significantly better than the cGA on all instances, and is able to solve all instances that have no more than 33% deception at least once. It should be noted that a high value of q is required to solve deceptive BBs but slows down the algorithm in the simpler cases. A re-run of iAS on case 1 and 2 using $q = 1$ requires only 473 and 568 FEs respectively.

Results Problem 2: NK Fitness Landscape

The results for the first part of this experiment are shown in Table 7: iAS is significantly better than the cGA in at least 70% of the cases, especially those with higher values of k. Again, as the limit on the FEs is increased, the differences in performance increase as well. This indicates that iAS makes better use of the available resources and suffers less from local optima entrapment.

The NK fitness landscape is an ADF where the size and overlap of linkage sets increases with k. High values of k thus imply a greater degree of difficulty as the optimal assignment for each subset of variables $\mathbf{X}_{V_i}$ is likely to interfere with most other assignments. iAS initially inverts a large subset of variables and this inversion is likely to affect most, if not all, linkage sets (for high values of k at least). Nevertheless, the performance of iAS seems to indicate that the incremental refinements made to the set of inverted variables is able to locate and isolate a set of variables the

Case	20,000 FEs (1) %	avg	FE	50,000 FEs (2) %	avg	FE	100,000 FEs (3) %	avg	FE	Sig 1v2	2v3	1v3	cGA Diff	Sig
1.00 - 0.00 - 0.00 (1)	100	100	1542	100	100	1542	100	100	1542	-	-	-	1743	*
0.00 - 1.00 - 0.00 (2)	100	100	1804	100	100	1804	100	100	1804	-	-	-	2206.1	*
0.00 - 0.00 - 1.00 (3)	0	90.9	-	0	91.95	-	0	94	-	x	*	*	-8	*
0.60 - 0.20 - 0.20 (4)	30	99.05	7952	80	99.75	24582	100	100	33693	*	x	*	-2.4	*
0.20 - 0.60 - 0.20 (5)	30	98.8	11458	70	99.6	24547	100	100	39088	*	x	*	-2.4	*
0.20 - 0.20 - 0.60 (6)	0	94.35	-	0	96.25	-	0	97.45	-	*	*	*	-4.15	*
0.33 - 0.33 - 0.33 (7)	0	97.3	-	20	98.7	39884	45	99.4	60362	*	*	*	-3.75	*

Table 6: Results for iAS on MBF for different distributions. The differences in performance compared to the cGA are shown in the right-most column (20,000 FEs). Column 'Sig' indicates if the differences are significant (*) and the % symbol indicates the number of trials solved successfully.

	20,000 FEs				50,000 FEs				100,000 FEs			
k	cGA	iAS	Diff	Sig	cGA	iAS	Diff	Sig	cGA	iAS	Diff	Sig
1	0.7105	0.7119	0.0014	x	0.7108	0.7120	0.0012	x	0.7112	0.7120	0.0009	x
2	0.7363	0.7403	0.0040	x	0.7374	0.7412	0.0038	x	0.7380	0.7421	0.0041	x
3	0.7448	0.7581	0.0134	*	0.7469	0.7609	0.0140	*	0.7483	0.7619	0.0137	*
4	0.7490	0.7634	0.0144	*	0.7517	0.7669	0.0152	*	0.7536	0.7700	0.0164	*
5	0.7473	0.7645	0.0172	*	0.7521	0.7657	0.0136	*	0.7530	0.7683	0.0153	*
6	0.7385	0.7535	0.0150	*	0.7419	0.7587	0.0169	*	0.7435	0.7615	0.0179	*
7	0.7388	0.7500	0.0111	*	0.7417	0.7535	0.0117	*	0.7438	0.7552	0.0114	*
8	0.7285	0.7457	0.0171	*	0.7308	0.7497	0.0189	*	0.7339	0.7508	0.0168	*
9	0.7251	0.7382	0.0131	*	0.7302	0.7426	0.0124	*	0.7307	0.7455	0.0148	*
10	0.7233	0.7370	0.0136	x	0.7287	0.7412	0.0124	x	0.7301	0.7442	0.0141	*

Table 7: Comparison of iAS and cGA on NK for different values of k given different limits on the number of FEs allowed. Column 'Sig' indicates if the differences are significant (*).

inversion of which is beneficial. In other words, the approach taken by iAS seems to work despite significant overlap (epistasis) of linkage sets in the problem space.

Results Problem 3: MKP

This section highlights the potential of iAS for other, more realistic, problems. In particular, it is shown how iAS may be used to solve constrained and permutation based optimisation problems. The constrained optimisation problem chosen is the MKP and iAS may be employed as usual. However, whenever a segment is inverted, the inversion is carried out as follows (starting with a feasible solution): first, all 1s are inverted to 0s. It is safe to do this because the exclusion of an item will never invalidate a solution. Secondly, all bits which were originally 0 are inverted (in random order), if possible. This approach ensures that the inverted segment is as close as possible to a fully inverted segment while obeying the constraints imposed by the problem. This is the simplest approach that ensures feasibility and more sophisticated techniques may be developed that should produce superior results. In particular, it is clear that problem specific knowledge such as the average value-weight ratio of all items, may be used here to determine the order of bits considered for inclusion. Nevertheless, as the results in table 8 show, iAS is able to solve at least one trial in almost all instances and again, iAS produces better results in almost all instances given a higher limit on the number of FEs allowed.

It is also possible to apply iAS to permutation based problems. The greatest difficulty in this case is the definition of what an 'alternative expression' is. This is straightforward in the binary case, but much less clear in the case of permutations because there is no single unique 'inversion'. In this case, the group membership as well as the order within the group matters. iAS may thus be extended to include another test at each step that evaluates a certain number of randomly

	20,000 FEs		100,000 FEs	
Instance	%	avg	%	avg
hp1	60	3405.15	75	3409.5
hp2	45	3151.85	70	3168.6
pb1	50	3069.45	65	3080.1
pb2	15	3142.1	45	3154.85
pb4	65	93882.55	85	94683.05
pb5	60	2132.2	85	2136.45
pb6	55	769.7	80	773.2
pb7	20	1027.8	50	1031.15
pet2	100	87061	100	87061
pet3	100	4015	100	4015
pet4	100	6120	100	6120
pet5	55	12394.5	100	12400
pet6	5	10565	40	10594.65
pet7	0	16448.35	5	16475.7
sent01	20	7755.7	20	7758.25
sent02	0	8701.45	0	8710.4
weing1	100	141278	100	141278
weing2	100	130883	100	130883
weing3	100	95677	100	95677
weing4	90	118986.4	100	119337
weing5	100	98796	100	98796
weing6	100	130623	100	130623
weing7	0	1094227.25	0	1095209.7
weing8	25	620728.1	35	621543.2
weish01	100	4554	100	4554
weish02	60	4534	60	4534
weish03	85	4105.55	100	4115

	20,000 FEs		100,000 FEs	
Instance	%	avg	%	avg
weish04	100	4561	100	4561
weish05	100	4514	100	4514
weish06	35	5546.8	55	5550.15
weish07	80	5562.35	100	5567
weish08	55	5602.55	100	5605
weish09	95	5244.3	100	5246
weish10	55	6322.85	80	6332.6
weish11	50	5611.35	95	5637.75
weish12	70	6320.6	100	6339
weish13	100	6159	100	6159
weish14	60	6938.35	80	6947.8
weish15	100	7486	100	7486
weish16	55	7287.2	75	7288.2
weish17	30	8623.85	95	8632.3
weish18	5	9561.35	35	9572.9
weish19	35	7672.25	65	7686.6
weish20	55	9442.3	80	9446
weish21	70	9063.6	90	9070.35
weish22	25	8908.4	30	8927.8
weish23	10	8319.35	25	8332.95
weish24	20	10202.35	80	10216.25
weish25	10	9919.45	35	9927.7
weish26	35	9547	55	9566.65
weish27	60	9779.45	90	9806.7
weish28	50	9453.75	70	9485.1
weish29	30	9360.4	70	9397.55
weish30	50	11178.75	70	11189.55

Table 8: Results for iAS on MKP using a partial repair function and two different limits on the number of FEs allowed. The % symbol indicates the number of trials solved successfully.

generated permutations within each half. In other words, at each stage, q divisions are tested and for each test, z permutations within each half are tested as well. The average fitness value of all z permutations is subsequently used to guide the search. This requires significantly more FEs, but may be beneficial for difficult problems that consist of relatively few variables, such as the quadratic assignment problem, or in cases where the number of FEs allowed is sufficiently large.

5.4 Discussion

iAS has been tested on three different problems and has been compared to the cGA on two of them. In both cases, iAS significantly outperformed the cGA in the majority of cases. The ability of iAS to invert large numbers of bits simultaneously allows the algorithm to escape from local optima. This is evident in the constant increase in performance once the limit on the number of FEs allowed is increased. Nevertheless, each iteration of iAS requires a significant cost in terms of FEs required to successfully invert a segment which could pose a problem, if resources are very limited.

The tests on the fully deceptive MBF seem to indicate that iAS works especially well on problems where deception leads to sub-optimal solutions that are maximum Hamming distance from the globally optimum sub-solution. However, this attribute is not necessarily restricted to fully deceptive problems as any local optimum in a binary search space is some Hamming distance away from the global optimum and it has been shown that iAS was able to solve at least some trials in almost all instances when tested on a constrained real-world problem. Further work is required

to investigate in more depth the kind of problems iAS may be expected to do well on. This will be left for future work.

6 Conclusions and Future Work

This chapter has presented two novel algorithms that are inspired by Alternative Splicing (AS), an important cellular process found in higher eukaryotes. Loosely speaking, AS affects the expression of individual genes by affecting the choice of modules (exons and introns) that contribute towards the protein. This modular composition of genetic information is strongly related to the concept of linkage in Evolutionary Computation (EC), and the idea of tightly linked informational Building Blocks (BBs) is evident in both algorithms presented.

The first algorithm, explicit Alternative Splicing (eAS), has been applied to a dynamic optimisation problem and systematically groups the variables of a problem according to their value across a series of finite states. This systematic grouping allows for the efficient reuse of acquired information in cases when the dynamic environment returns to a previously visited state. eAS is essentially an implicit memory approach that attempts to capture the state of a variable across multiple different states. Linkage groups in this case do not describe the sub-structures of the base problem, but instead capture the properties of a succession of different instances of the base problem. The ability to recall states from memory allows eAS to deal efficiently with dynamics, at least in the cases considered here. It does not, however, help the algorithm to solve the base problem, and it is expected that other, more sophisticated, operators may have a significant impact on the overall performance of eAS.

The second algorithm, implicit alternative splicing (iAS), uses a top-down search process to locate a segment for which inversion has a non-negative impact on the encoding's fitness. The search starts with a large, randomly chosen initial segment which is systematically reduced in size. At each iteration, the current segment is halved and each half is inverted and the resulting encoding is evaluated. These intermediate fitness values are used to guide the search towards a successful inversion. This algorithm is able to solve fully deceptive problems if given sufficient time. Furthermore, iAS significantly outperformed a canonical GA on two test problems, and managed to solve the majority of instances in the multiple knapsack problem using a partial random repair.

Both algorithms are based upon a simple (1+1) EA. The reason for choosing this particular underlying framework was mainly driven by the fact that individual encodings require multiple function evaluations (FEs) to produce a candidate solution (although in the case of eAS, multiple FEs are only required if a change has occurred). In nature, most cellular processes rely upon the massive parallelism evident in animal populations, but in EC such luxuries are usually unavailable and compromises have to be made (i.e. population size reductions). If resources are available, we would expect either algorithm to work well in a population based framework.

6.1 Future Work

We are currently expanding our work on the iAS algorithm. In particular, the algorithm seems highly suitable for use in dynamic domains, cyclic or not, as it should be able to deal efficiently with a variety of different transitions. Initial experiments have shown promise, but further testing is required to determine the full power of the approach. Furthermore, efforts are underway to analyse the behaviour of iAS analytically, and to draw some general conclusions regarding its running time for selected problems. The methodology employed in iAS is fairly general, and it should be possible to refine the performance further using local search or statistical processing to guide the choice of divisions. Finally, it may also be of interest to combine the two approaches to exploit the benefits each of them has to offer.

7 Acknowledgements

This work was supported by a Paul and Yuanbi Ramsay scholarship.

References

[1] L. Altenberg. NK fitness landscapes. In T. Bäck, D. B. Fogel, and Z. Michalewicz, editors, *The Handbook of Evolutionary Computation*, pages B2.7:2–B2.7:10. Oxford University Press, 1997.

[2] B. S. Baker. Sex in flies: The splice of life. *Nature*, 340:521–524, 1989.

[3] D. L. Black. Mechanisms of alternative pre-messenger RNA splicing. *Annual Review of Biochemistry*, 72:291–336, 2003.

[4] P. A. N. Bosman and H. L. Poutrè. Learning and anticipation in online dynamic optimization with evolutionary algorithms: the stochastic case. In *Proceedings of the 2007 Genetic and Evolutionary Computation Conference*, pages 1165–1172, 2007.

[5] J. Branke. *Evolutionary Optimization in Dynamic Environments.* Kluwer, 2001.

[6] J. Branke, T. Kaußler, C. Schmidt, and H. Schmeck. A multi-population approach to dynamic optimization problems. In I. C. Parmee, editor, *Adaptive Computing in Design and Manufacture 2000*, pages 299–308. Springer, 2000.

[7] H. G. Cobb. An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continuous, time-dependant nonstationary environments. Technical report, Naval Research Laboratory, Washington, USA, 1990.

[8] P. Collard and J.-P. Aurand. DGA: An efficient genetic algorithm. In *Proceedings of the Eleventh European Conference on Artificial Intelligence*, pages 487–491, 1994.

[9] D. Dasgupta and D. R. McGregor. Nonstationary function optimization using the structured genetic algorithm. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 145–154, Amsterdam, 1992. Elsevier.

[10] E. A. Di Paolo. Rhythmic and non-rhythmic attractors in asynchronous random boolean networks. *BioSystems*, 59:185–195, 2001.

[11] R. A. Fisher. *The Genetical Theory of Natural Selection.* Oxford: Clarendon Press, 1930.

[12] A. Gaspar, M. Clergue, and P. Collard. Folding genetic algorithms: the royal road toward an optimal chromosome's expressiveness. In *Second International ISCS Symposium on Soft Computing*, 1997.

[13] A. Gaspar and P. Collard. Time dependent optimization with a folding genetic algorithm. In *IEEE International Conference on Tools for Artificial Intelligence*, pages 207–214. IEEE Computer Society Press, 1997.

[14] D. E. Goldberg. *The Design of Innovation: Lessons from and for competent genetic algorithms.* Kluwer Academic Publishers, 2002.

[15] D. E. Goldberg, D. E. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis and first results. *Complex Systems*, 3:493–530, 1989.

[16] D. E. Goldberg and R. E. Smith. Nonstationary function optimization using genetic algorithms with dominance and diploidy. In J. J. Grefenstette, editor, *Second International Conference on Genetic Algorithms*, pages 59–68. Lawrence Erlbaum Associates, 1987.

[17] D.E. Goldberg, K. Deb, H. Kargupta, and G. Harik. Rapid, accurate optimization of difficult problems using fast messy genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 56–64, 1993.

[18] B. R. Graveley. Alternative splicing: increasing diversity in the proteomic world. *TRENDS in Genetics*, 17(2):100–107, February 2001. 4 stars.

[19] J. J. Grefenstette. Genetic algorithms for changing environments. In R. Manner and B. Manderick, editors, *Proceedings of the Second International Conference on Parallel Problem Solving from Nature 2*, pages 137–144, Amsterdam, 1992. Elsevier.

[20] B. S. Hadad and C. F. Eick. Supporting polyploidy in genetic algorithms using dominance vectors. In *Proceedings of the Sixth International Conference on Evolutionary Programming*, volume 1213, pages 223–234. Springer, 1997.

[21] G. Harik. *Learning Gene Linkage to Efficiently solve problems of bounded difficulty using genetic algorithms.* PhD thesis, University of Michigan, Ann Arbor, 1997.

[22] G. R. Harik. Linkage learning via probabilistic modeling in the ECGA. Technical Report 99010, University of Illinois at Urbana-Champain, 1999.

[23] G. R. Harik, F. G. Lobo, and D. E. Goldberg. The compact genetic algorithm. *IEEE*, 3(4):287, 1999.

[24] E. D. Harrington, S. Boue, J. Valcarcel, J. G. Reich, and P. Bork. Estimating rates of alternative splicing in mammals and invertebrates. *Nature Genetics*, 36(9):915–917, 2004.
[25] A. Herbet and A. Rich. RNA processing and the evolution of eukaryotes. *Nature Genetics*, 21:265–269, 1999.
[26] C.-F. Huang and L. M. Rocha. Exploration of RNA editing and design of robust genetic algorithms. In *Proceedings of the 2003 IEEE Congress on Evolutionary Computation*. IEEE Press, 2003.
[27] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409:860–921, 2001.
[28] H. Kargupta and S. Gosh. Towards machine learning through genetic code-like transformations. *Genetic Programming and Evolvable Machine Journal*, 3(3):231–258, 2002.
[29] H. Karuptga. The gene expression messy genetic algorithm. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, 1996.
[30] S. A. Kauffman. *The Origins of Order*. Oxford University Press, 1993.
[31] F. A Kondrashov and E. V. Koonin. Origin of alternative splicing by tandem exon duplicaton. *Human Molecular Genetics*, 10(23):2661–2669, 2001.
[32] A. N. Ladd and T. A. Cooper. Finding signals that regulate alternative splicing in the postgenomic era. *Genome Biology*, 3(11):1–16, 2002.
[33] J. R. Levenick. Swappers: Introns promote flexibility, diversity and invention. In F. L. Orlando, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, H. Jakiela, and R. E. Smith, editors, *Proceeding of Genetic and Evolutionary Computation Conference 1999*, volume 1, pages 361–368, San Francisco, CA, July 1999. Morgan Kaufmann.
[34] A. J. Lopez. Alternative splicing of pre-mRNA: Developmental consequences and mechanisms of regulation. *Annual Review of Genetics*, 32:279–305, 1998.
[35] B. Modrek and C. J. Lee. Alternative splicing in the human, mouse and rat genomes is associated with an increased frequency of exon creation and/or loss. *Nature Genetics*, 34(2):177–180, 2003.
[36] R. W. Morrison. *Designing Evolutionary Algorithms for Dynamic Environments*. Springer, Berlin, 2004.
[37] P. Rohlfshagen and J. A. Bullinaria. Alternative splicing in evolutionary computation: Adaptation in dynamic environments. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computing*, pages 8041–8048, Piscataway, NJ, 2006. IEEE.
[38] P. Rohlfshagen and J. A. Bullinaria. Implicit alternative splicing for genetic algorithms. In *Proceedings of the 2007 IEEE Congress on Evolutionary Computing*, pages 47–54. IEEE Press, 2007.
[39] P. Rohlfshagen and E. A. Di Paolo. The circular topology of rhythm in asynchronous random boolean networks. *BioSystems*, 73:141–152, 2004.
[40] S. Yang. Non-stationary problem optimization using the primal-dual genetic algorithm. In R. Sarker, R. Reynolds, H. Abbass, K.-C. Tan, R. McKay, D. Essam, and T. Gedeon, editors, *Proceedings of the 2003 IEEE Congress on Evolutionary Computation*, volume 3, pages 2246–2253, 2003.
[41] S. Yang. PDGA: the primal-dual genetic algorithm. In A. Abraham, M. Koppen, and K. Franke, editors, *Design and Application of Hybrid Intelligent Systems*, pages 214–223. IOS Press, 2003.
[42] S. Yang. A comparative study of immune system based genetic algorithms in dynamic environments. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1377–1384, 2006.