Evolving Monotone Conjunctions in Regimes Beyond Proved Convergence

Pantia-Marina Alchirch[1](✉)[0000−0002−9596−7884], Dimitrios I. Diochnos[2][0000−0002−2934−606X], and Katia Papakonstantinopoulou[1,2][0000−0002−4674−9303]

[1] Athens University of Economics and Business (TESLAB), Hellas
{marina.alchirch,katia}@aueb.gr
[2] University of Oklahoma, USA
{diochnos,katia}@ou.edu

Abstract. Recently it was shown, using the typical mutation mechanism that is used in evolutionary algorithms, that monotone conjunctions are provably evolvable under a specific set of Bernoulli$(p)^n$ distributions. A natural question is whether this mutation mechanism allows convergence under other distributions as well. Our experiments indicate that the answer to this question is affirmative and, at the very least, this mechanism converges under Bernoulli$(p)^n$ distributions outside of the known proved regime.

Keywords: Evolvability · Genetic programming · Monotone conjunctions · Distribution-specific learning · Bernoulli$(p)^n$ distributions

1 Introduction

Automating the creation of computer programs that perform intelligent operations has been driving the research in evolutionary programming and in machine learning – though the approaches used are oftentimes different. Slightly more than a decade ago, these two fields came closer with the introduction of the framework of *evolvability* by Leslie Valiant [16].

Evolvability formulates evolution as a learning process and is a framework for a special type of local search method that ultimately develops *individuals* (that is, computer programs) that have high fitness within their environment. In other words, the goal is to develop a function that has high predictive accuracy on an unknown function c that we want to learn from training examples.

We continue the study of a simple and intuitive class of Boolean functions, that of *monotone conjunctions*, within the framework of evolvability.

1.1 Monotone Conjunctions and Representation

A monotone conjunction is a function that combines a set of variables with a Boolean AND. For example, the function $f = x_1 \wedge x_2 \wedge x_5$ returns TRUE if the first, second and fifth variable are satisfied simultaneously on a truth assignment

$a = (a_1, \ldots, a_n) \in \{0,1\}^n$, otherwise it returns FALSE. When we are working in a space with n Boolean variables, an intuitive representation for monotone conjunctions is that of a bitstring of length n, where a 1 (resp. 0) in a particular bit indicates the presence (resp. absence) of the specific variable in the function. For example, when $n = 8$, we can represent the function $f = x_1 \wedge x_2 \wedge x_5$ as: $\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{0}$. With $|h|$ we denote the *size* of a monotone conjunction h; the number of variables that are contained in h. Hence, in our example, $|f| = 3$.

On the Importance of Conjunctions within Machine Learning Conjunctions, as well as disjunctions, are perhaps the most basic classes of Boolean functions that act as building blocks for more complex functions. Even though these classes of functions are simple, nevertheless they have exponentially many functions on n Boolean variables and therefore provide a basic testbed for various ideas, as well as for understanding general bounds that are proved in the context of machine learning. Furthermore, learning algorithms for such basic classes of functions may provide insights for more sophisticated algorithms or even extend naturally to algorithms for richer classes of functions in certain contexts.

As an example, within the *Probably Approximately Correct (PAC)* model of learning [15], learning functions that are disjunctions of a *constant* number k of conjunctions can be achieved with a learning algorithm that is merely used for learning conjunctions [11]. The idea is that a disjunction of k conjunctions $f_1 \vee f_2 \vee \cdots \vee f_k$ can be converted to a conjunctive formula, where each clause has at most k literals[3] via the distributive law as shown below:

$$f_1 \vee f_2 \vee \cdots \vee f_k = \bigwedge_{u_1 \in f_1, u_2 \in f_2, \ldots, u_k \in f_k} (u_1 \vee u_2 \vee \cdots \vee u_k)\,.$$

Therefore, for every selection (allowing repetitions) of k literals $(u_1, u_2, \ldots, u_k)$ over the original set of n Boolean variables $\{x_1, \ldots, x_n\}$, one can create a new variable $y_{u_1,u_2,\ldots,u_k}$ whose value is defined by $y_{u_1,u_2,\ldots,u_k} = u_1 \vee u_2 \vee \cdots \vee u_k$. Hence, an efficient distribution-independent algorithm for learning conjunctions from the set $\{x_1, \ldots, x_n\}$, may also learn such richer functions efficiently, but this time over the broader set of the y variables, which are $(2n)^k$ in total.

1.2 Related Work and Motivation

Using a simulation argument, a hallmark result in evolvability is one by Vitaly Feldman where it has been shown that evolvability is equivalent to learning using correlational statistical queries under a fixed distribution [5]. However, this simulation result, as has also been pointed out by Feldman, is not necessarily the most intuitive or efficient approach for designing evolutionary algorithms. At the same time intuitive evolutionary mechanisms are desirable and sought for; see, e.g., [9, 12]. In this context, it is perhaps not surprising that one of the simplest, non-trivial, classes of Boolean functions, that of monotone conjunctions, has received a lot of attention and their evolvability has been studied.

[3] A literal is a Boolean variable or its negation.

In particular, Leslie Valiant gave a *swapping-type* algorithm for learning monotone conjunctions when the distribution was uniform over $\{0,1\}^n$, when he introduced evolvability [16]. We outline this *swapping algorithm* in Section 3.1. The analysis of this algorithm was simplified in [3]. Eventually it was shown that this algorithm converges for Bernoulli$(p)^n$ distributions (defined in Section 3), characterized by any $p \in (0,1)$, where the uniform distribution is a special case obtained for $p = 1/2$. Meanwhile, another direction of research towards the learnability of monotone conjunctions has explored the power of *parallel* statistical queries by means of *recombination* [7], and of *horizontal gene transfer* [14].

On the other hand, the problem of learning monotone, or not, conjunctions has been studied within genetic programming (GP) as well. In this direction [10, 9] have explored tree-like representations for learning monotone conjunctions under the uniform distribution in the realistic (for machine learning and evolvability) case, where the number of training examples are upper bounded by some polynomial of the input parameters. There has also been done additional work on exploring the learnability of monotone conjunctions, but some of these algorithms may have unrealistic assumptions for the framework of evolvability. For example, the algorithm in [13] uses a genetic approach in which the updates depend on the number of bits in which the candidate solution and the input differ. As another example, in the case of [6] it is assumed that the learner has knowledge of the exact fitness value of various hypotheses.

Along the lines of genetic programming, another mechanism that has been studied is the one inspired by the standard mutation mechanism that is encountered in (1+1) evolutionary algorithms (EAs). This mechanism considers all the bits in the bitstring representation of a monotone conjunction (recall the discussion from Section 1.1) and tosses a coin that succeeds with probability $1/n$ in each bit. Whenever the coin toss succeeds, the bit at the particular coordinate is flipped. This algorithm has been shown to converge under product distributions where each variable is satisfied with the same probability p (called Bernoulli$(p)^n$ distributions; see Section 3), when p takes values in $(0, 1/3] \cup \{1/2\}$ [2]. A natural question that we try to answer in this paper is the following one:

> *Does the mutation mechanism that is inspired by the (1+1) EA allow the evolvability of monotone conjunctions under a broader set of distributions, compared to what is currently provably known?*

Our experimental findings indicate that the answer is *affirmative*.

Structure of the Paper Section 2 summarizes the computational models that come together in our work. Section 3 provides details on the problem that we study as well as a brief discussion on a related algorithm to our work, from where we draw inspiration on providing specific values to certain parameters that govern the evolutionary mechanism that we study. Section 4 provides details on the implementation of our method and how we define successful executions. Section 5 presents the values of certain parameters that we use in the experiments as well as discusses the results of our experimental study. Section 6 concludes our work with a summary and ideas for future work.

2 Computational Models Relevant to Our Work

We now describe briefly the computational models that are relevant to our work. Before we do that, however, we make a remark on the terminology.

Remark 1 (Terminology). A candidate solution of an optimization problem, in EAs/GP is typically called an *individual*. On the other hand, in machine learning, a candidate solution to a learning problem, is typically called a *hypothesis* (or a *model*). One may use these terms interchangeably and in particular in our case these correspond to Boolean functions (or, if you prefer, to *computer programs*).

2.1 Evolutionary Algorithms and Evolving Programs

Evolutionary algorithms is a class of algorithms that develop solutions to optimization problems of interest. The development of these solutions proceeds in an iterative manner, such that candidate solution(s) from one iteration to the next are typically obtained after applying modification operators on the representation of the candidate solution(s) of the previous iteration. The function that is being optimized is called a *fitness function*. The idea is that the higher the fitness value of a particular individual (solution) is, the better the individual is for our purposes; i.e., as a solution to the optimization problem that we solve. The simplest mechanism that creates such solutions is shown in Algorithm 1, where we see that given an individual (candidate solution) x encoded as a bitstring of length n, a *mutated* version x' is obtained from x after tossing n times a coin that succeeds with probability $1/n$, and upon success of each coin toss, the respective bit in the binary representation of x is flipped. This modification mechanism is

Algorithm 1: The (1+1) Evolutionary Algorithm

Input: A function f to be optimized over $\{0,1\}^n$.
Output: A solution x, candidate for optimizing f.

```
x ← random string from {0,1}^n;
repeat
    Compute x' by flipping each bit of x independently with probability 1/n;
    if f(x') ≥ f(x) then x ← x';
until some termination condition is met;
```

called a *mutation* as it tries to mimic in an elegant and compact way the way mutations occur in nature, and thus allows this algorithmic scheme to explore the binary search space in a randomized way. If x' is at least as fit as its *parent* x, then x' is selected to be the solution used for the next generation; otherwise, x is selected for one more generation. This way, the different solutions that we obtain across the different *iterations* (also known as *generations*) monotonically increase the fitness values that correspond to them. The interested reader may find additional discussion and several interesting results in [4].

A closely related field to EAs is that of genetic programming (GP) [8]. Similarly to EAs, the goal of GP is to develop a solution/individual that maximizes a fitness function. However, in the case of GP, the individuals correspond to different *functions* (computer programs), rather than to mere numerical points or truth assignments, from the domain of the fitness function. The most usual representation of these individuals is with the use of tree structures, as then on one hand such a representation is convenient (in a manner similar to decision trees) and on the other hand it is easy to define modification operators inspired by nature, such as mutation and recombination, and give rise to new individuals to be considered as candidate solutions that may survive in the next generation.

Our work in this paper falls under the broader umbrella of supervised machine learning, where the goal of the learner is to develop a function that approximates well some ground truth function. In other words, the goal of the learner is very well aligned with the goal of GP. On the other hand, the functions that we consider in our case have a very natural representation using bitstrings, as it was discussed in Section 1.1, and thus we can ultimately use Algorithm 1.

2.2 Supervised Machine Learning and Evolvability

In supervised machine learning the learner is typically presented with a set $\mathcal{S} = \{(x_i, c(x_i))\}_{i=1}^m$ of *training examples* that exhibit the behavior of some *ground truth* function c on certain instances of the domain $\mathcal{X}$. Based on this information, the learner forms a *hypothesis* (or a *model*) h that approximates the ground truth c. For example, one typical approach for selecting a hypothesis h from a set of possible hypotheses $\mathcal{H}$, is that of *empirical risk minimization*, where the $h \in \mathcal{H}$ that is selected is the function that has the best predictive accuracy on the training examples $\mathcal{S}$ that were given to the learner.

Evolvability on the other hand is a special framework for supervised machine learning. In particular, in evolvability, the learner only gets to know how well their hypotheses approximate the ground truth function c, based on *aggregate* information that is computed from training examples. This information is equivalent to a noisy estimate of the risk (error rate) of the various hypotheses. The idea is that evolvability casts the whole process of evolution as a learning problem and the modifications that occur on the individuals (hypotheses) during the evolution should be favoring the fittest ones for survival to the next generation. In that sense, the encoding of the individuals at the genotype level cannot depend on individual experiences, but rather on some aggregate signal that is received from the environment, which thus describes how fit the particular individual is for this environment. Algorithms for evolvability are called *ecorithms*.

After this brief high-level discussion on supervised learning and evolvability, we will now provide more details for the framework of evolvability. We are looking at Boolean functions where the output values TRUE and FALSE are represented by 1 and -1 respectively. Evolvability works in a local search fashion, where at each step of the evolution a (parent) hypothesis h is considered together with the hypotheses that are obtained after applying a mutation operator on

the particular (parent) hypothesis h, forming a neighborhood $N(h)$. Eventually, each hypothesis in the neighborhood $N(h)$ is evaluated using a *fitness* function, called *performance*, and ultimately this function is driving the search. For a *target function* c that we are trying to learn[4] and a distribution D over $\{0,1\}^n$, the performance of a hypothesis h, also called the *correlation* of h and c, is

$$Perf_D(h,c) = \mathbf{E}_{x\sim D}\left[h(x)\cdot c(x)\right]. \tag{1}$$

Note that from the above definition we also have that:

$$Perf_D(h,c) = \sum_{x\in\{0,1\}^n} h(x)c(x)\mathbf{Pr}_{x\sim D}(x) = 1 - 2\cdot\mathbf{Pr}_{x\sim D}\left(h(x)\neq c(x)\right). \tag{2}$$

An approximate value $\widehat{Perf}_S(h,c)$ of $Perf_D(h,c)$ is obtained empirically for each hypothesis using a sample S; we denote this value with ν_h for brevity. Then, for a real constant t, called *tolerance*, we obtain the sets:

$$\begin{cases} \texttt{Bene} = \{h'\in N(h) \mid \nu_{h'} > \nu_h + t\} \\ \texttt{Neut} = \{h'\in N(h) \mid \nu_{h'} \geq \nu_h - t\} \setminus \texttt{Bene}. \\ \texttt{Del} \ \ = \{h'\in N(h) \mid \nu_{h'} < \nu_h - t\} \end{cases} \tag{3}$$

Hence, for the next iteration, a hypothesis from the set `Bene` is selected, should `Bene` $\neq \emptyset$. Otherwise, a hypothesis from `Neut` is selected; note that `Neut` $\neq \emptyset$ since `Neut` always contains the parent hypothesis h. Thus, while the set `Del` of deleterious mutations is needed for partitioning the neighborhood $N(h)$, it is of little interest as no hypothesis will ever be selected from `Del`. *The goal of the evolution* is to produce in $poly(1/\varepsilon, 1/\delta, n)$-time a hypothesis h such that

$$\mathbf{Pr}\Big(Perf_D(h,c) < Perf_D(c,c) - \varepsilon\Big) < \delta. \tag{4}$$

3 The Learning Problem that we Study

We are interested in learning monotone conjunctions in the framework of evolvability. In particular, we focus on Bernoulli$(p)^n$ distributions $\mathcal{B}_{n,p}$ over $\{0,1\}^n$. These distributions are specified by the probability p of setting each variable x_i equal to 1. Thus, a truth assignment $(a_1,\ldots,a_n)\in\{0,1\}^n$ has probability $\prod_{i=1}^n p^{a_i}(1-p)^{1-a_i}$. Given a monotone conjunction c that we want to learn and a hypothesis h, we can partition the variables that appear in either c or h as shown below:

$$c = \bigwedge_{i=1}^{m} x_i \wedge \bigwedge_{k=1}^{u} y_k \quad \text{and} \quad h = \bigwedge_{i=1}^{m} x_i \wedge \bigwedge_{\ell=1}^{r} w_\ell. \tag{5}$$

Therefore, the x's are *mutual* variables, the y's are called *undiscovered* (or *missing*) variables, and the w's are the *wrong* (or *redundant*) variables. Variables

[4] The function c is also called *ideal function*, as it represents the *ideal behavior* in a certain environment.

in the target c are called *good*, otherwise they are called *bad*. Given this decomposition, we can calculate the quantity $\mathbf{Pr}_{x\sim\mathcal{B}_{n,p}}(h(x) \neq c(x))$ under a $\mathcal{B}_{n,p}$ distribution, with the following two observations:

- $h(x) = +1$ and $c(x) = -1$: This happens on truth assignments where the x_i's are satisfied, the w_ℓ's are satisfied, and at least one of the y_k's is falsified. Therefore, this will happen with probability $p^m p^r (1 - p^u)$.
- $h(x) = -1$ and $c(x) = +1$: Similar analysis implies that this will happen with probability $p^m p^u (1 - p^r)$.

Adding the above two we get: $\mathbf{Pr}_{x\sim\mathcal{B}_{n,p}}(h(x) \neq c(x)) = p^{m+r} + p^{m+u} - 2p^{m+r+u}$. As a consequence, (2) reduces to,

$$Perf_{\mathcal{B}_{n,p}}(h, c) = 1 - 2p^{m+r} - 2p^{m+u} + 4p^{m+r+u}\,. \tag{6}$$

Definition 1 (Short, Medium, Long). *Given integers q and ϑ, a monotone conjunction f is* short *when $|f| \leq q$,* medium *when $q < |f| \leq q + \vartheta$, and* long *otherwise.*

Definition 1 partitions the class of functions that we want to learn in three groups and will allow us to define a criterion (Criterion 1) that we will use in order to determine if a particular experimental run is successful or not. We will also need the following definition.

Definition 2 (Best q-Approximation). *A hypothesis h is called a best q-approximation of c if $|h| \leq q$ and $\forall h' \neq h, |h'| \leq q : Perf_D(h', c) \leq Perf_D(h, c)$.*

3.1 A Related Algorithm: The Swapping Algorithm

Before we discuss details of our implementation, we briefly describe a related algorithm to our work, the *swapping algorithm* for monotone conjunctions, that was introduced by Valiant in [16].

The swapping algorithm has been shown to converge [1] under Bernoulli$(p)^n$ distributions that are characterized by any $0 < p < 1$ using the general evolutionary scheme that was described in Section 2.2, where at every step of the evolution the neighborhood $N(h)$ is partitioned into the sets `Bene`, `Neut`, and `Del`, and selection first favors the set `Bene`, otherwise the set `Neut`. The algorithm is important for our work because we intend to use some of its parameters and ideas in the evolutionary mechanism that we want to study.

In the swapping algorithm, the neighborhood $N(h)$ of a monotone conjunction h is the set of monotone conjunctions that arise by *adding* a variable (neighborhood $N^+(h)$), *removing* a variable (neighborhood $N^-(h)$), or *swapping* a variable with another one (neighborhood $N^\pm(h)$), plus the conjunction itself. Thus, $N(h) = N^-(h) \cup N^+(h) \cup N^\pm(h) \cup \{h\}$. As an example, let $h = x_1 \wedge x_2$, and $n = 4$. Then, $N^-(h) = \{x_1, x_2\}$, $N^+(h) = \{x_1 \wedge x_2 \wedge x_3, x_1 \wedge x_2 \wedge x_4\}$, and $N^\pm(h) = \{x_3 \wedge x_2, x_4 \wedge x_2, x_1 \wedge x_3, x_1 \wedge x_4\}$. Note that $|N(h)| = \mathcal{O}(n\,|h|)$ in general. Finally, for the parameters q and ϑ that appear in Definition 1, the swapping algorithm uses the following values:

$$q = \lceil \log_{1/p}(3/\varepsilon) \rceil \quad \text{and} \quad \vartheta = \lfloor \log_{1/p}(2) \rfloor\,. \tag{7}$$

4 Implementation

Regarding the implementation, our starting point is the algorithm for the evolution of monotone conjunctions that was used in [2], which in turn is based on the (1 + 1) EA (Algorithm 1). Our evolutionary mechanism is shown in detail in Algorithm 2. This algorithm is known to converge [2] to a hypothesis that

Algorithm 2: MUTATOR function based on the (1 + 1) EA

$q \leftarrow \lceil \log_{1/p}(3/\varepsilon) \rceil$;
$h' \leftarrow$ MUTATE(h);
if $p < 1/3$ **then** $t \leftarrow p^{q-1} \min\{4p^q/3, 1-3p\}$;
else if $p = 1/3$ **then** $t \leftarrow 2 \cdot 3^{-1-2q}$;
else if $p = 1/2$ **then** $t \leftarrow 2^{-2q}$;
else if $p > 1/3$ ***and*** $p < 1/2$ **then**
  $\vartheta \leftarrow 0$;
  $\Lambda \leftarrow 1 - 2p$;
  $\mu \leftarrow \min\{2p^{q+\vartheta}, \Lambda\}$;
  $t \leftarrow p^{q-1}\mu(1-p)$;
else
  $k \leftarrow \lfloor \log_{1/p}(2) \rfloor$;
  $\vartheta \leftarrow k$;
  $\Lambda \leftarrow \min\{\left|2p^k - 1\right|, \left|1 - 2p^{k+1}\right|\}$;
  $\mu \leftarrow \min\{2p^{q+\theta}, \Lambda\}$;
  $t \leftarrow p^{q-1}\mu(1-p)$;
if $|h'| > q$ **then return** h;
$\nu_h \leftarrow$ EVALUATEHYPOTHESIS(h);
$\nu_{h'} \leftarrow$ EVALUATEHYPOTHESIS(h');
if $\nu_{h'} > \nu_h + t$ **then return** h';
else if $\nu_{h'} >= \nu_h - t$ **then return** USELECT(h, h');
else return h;

satisfies (4), which is the goal for evolution, for Bernoulli$(p)^n$ distributions that are characterized by $p \in (0, 1/3] \cup \{1/2\}$. However, we are interested in studying this evolutionary mechanism, at the very least, for other values of p that characterize Bernoulli$(p)^n$ distributions and it is this particular case that we explore in this paper. Below we explain the functions that appear in Algorithm 2.

EVALUATEHYPOTHESIS returns the performance $Perf_D(h, c)$ of a hypothesis h. In the experiments we do that using (6), by using the values m, u, and r, of the mutual, undiscovered, and redundant variables.

The function MUTATE takes as input the bit vector that represents the initial hypothesis, flips each bit with probability $1/n$, and returns the new mutated hypothesis. This is the mutation mechanism that was described in Algorithm 1.

The function USELECT is responsible for selecting uniformly at random a hypothesis from the two that are passed as parameters. In particular, the two

hypotheses are h and h', where h is the initial hypothesis and h' the mutated one that occurred from function MUTATE.

Finally, we would like to make the following remark. As discussed in Section 2.2, the evolutionary mechanism has access to a noisy value $\widehat{Perf}_S(h,c)$, that is obtained from an appropriately large sample S, as a proxy for the true value $Perf_D(h,c)$ for some hypothesis h. However, by using (6) directly in EVALUATEHYPOTHESIS we obtain the true value *exactly*. We argue that this should not be a problem, as the neighborhood is split into the sets `Bene`, `Neut`, and `Del` based on the tolerance t. The idea is that when one may try in the future to prove rigorously our experimental findings from Section 5, it should be enough to identify the minimum non-zero difference in the performance between *any two* hypotheses in the hypothesis space. Assuming this value is equal to Δ, then by setting the tolerance equal to $\Delta/2$ and requiring approximation of each $Perf_D(h,c)$ to be done within $\Delta/2$ of their true value, then the sets `Bene`, `Neut`, and `Del`, will be entirely correct in the partitioning of the hypotheses in the neighborhood, to beneficial, neutral, and deleterious.

4.1 Setting the Parameters q and ϑ

Two important parameters that we use in Algorithm 2 are the parameters q and ϑ and the values that we use are given by (7). Regarding q, its value has been the same in [3, 1, 2] and therefore it is only natural to maintain this definition in our work as well. Regarding ϑ, we introduce it because it was useful for proving the convergence of the swapping algorithm when $p \geq 1/2$. One of the ideas from [1] is that when the function c that we want to learn is of medium size (i.e., $q < |c| \leq q + \vartheta$) and the distribution $\mathcal{B}_{n,p}$ is governed by some $p \geq 1/2$, then convergence is proved when a hypothesis h is formed that is a best q-approximation of c (per Definition 2). Hence, our hope is that this phenomenon will transcend from the swapping algorithm where it has been proved to work, to the (1+1) EA mechanism that we explore in this work.

4.2 Guessing a Good Value for the Tolerance t

Beyond q and ϑ for which we use the values of (7), another important parameter for evolution is that of the tolerance t. As the algorithm that we use (Algorithm 2) comes from [2], we use the values indicated by [2] in the proved regime; i.e., when $p \in (0, 1/3] \cup \{1/2\}$. In the unproved regime (i.e., when $p \notin (0, 1/3] \cup \{1/2\}$) we attempt to use the tolerance that is indicated in [1] which allows the swapping algorithm to converge for every $p \in (0, 1)$ that characterizes the Bernoulli$(p)^n$ distribution that governs the instances. In particular, the tolerance in [1] is

$$t_{\text{swapping}} = p^{q-1}\mu(1-p)\,, \tag{8}$$

where $\mu = \min\{2p^{q+\vartheta}, \Lambda\}$. Regarding the quantity Λ, if $0 < p < 1/2$ we have

$$\Lambda_{p<1/2} = 1 - 2p\,. \tag{9}$$

When $p \in [1/2, 1)$, the quantity Λ is defined by first looking if p is of the form $2^{-1/k}$, with $k \in \{1, \ldots, n\}$, or if p belongs to a sub-interval of $[1/2, 1)$ of the form $(2^{-1/k}, 2^{-1/(k+1)})$; in other words, we care about the two consecutive points from the family of points $2^{-1/k}$ (with $k \in \{1, \ldots, n\}$) that contain p. It is this latter case which corresponds to the values of $p > 1/2$ that we examine in this work in the unproved regime (i.e., $p \in \{0.6, 0.7, 0.8, 0.9\}$). Note that the interval of interest $(2^{-1/k}, 2^{-1/(k+1)})$ is obtained for $k = \lfloor \log_{1/p}(2) \rfloor$. Eventually, a quantity that is good enough for our purposes is to set

$$\Lambda_{p>1/2} = \min\{\left|2p^k - 1\right|, \left|1 - 2p^{k+1}\right|\}. \tag{10}$$

In other words, using (9) and (10) we can define

$$\Lambda = \begin{cases} \Lambda_{p<1/2}\ , \text{ if } 1/3 < p < 1/2, \\ \Lambda_{p>1/2}\ , \text{ if } 1/2 < p < 1\,. \end{cases} \tag{11}$$

Now, one can use (11) in (8) and compute the desired tolerance that will be used for the experiments depending on the p that we want to test.

4.3 Successful Executions

The following criterion was used for proving convergence in [1] and we adopt it.

Criterion 1 (Success Criterion) *We define a single run to be successful if we accomplish the following:*

(a) When c is short, identify c precisely.
(b) When c is medium, generate a best q-approximation of c.
(c) When c is long, generate a hypothesis h such that $\mathsf{Perf}_{\mathcal{B}_{n,p}}(h, c) \geq 1 - \varepsilon$.

Therefore, for a given Bernoulli$(p)^n$ distribution and a given target c, we run Algorithm 2 in an endless loop until we satisfy our Criterion 1. In fact, we consider such an execution successful if we satisfy Criterion 1 *for 10 consecutive iterations*, thus signifying that the solution that we have found has some notion of stability and therefore it is not the case that we satisfy perhaps Criterion 1 during one iteration but then in a subsequent iteration the hypothesis drifts away and evolves to a solution that has performance less than $1 - \varepsilon$.

Remark 2 (On the Strictness of the Success Criterion). Criterion 1 is probably more strict than what is really needed in some cases. To see this, consider the following situation: say, $p = 0.2$, $\varepsilon = 0.01$ ($\Rightarrow q = 4$), and the target function that we want to learn is $c = x_1 \wedge x_2 \wedge x_3 \wedge x_4$. Then, according to Criterion 1, this is case (a), and we would like to evolve h such that $h = c$. However, the hypothesis $h' = x_5 \wedge x_6 \wedge x_7 \wedge x_8$ is very different from c (as none of the variables that appear in h also appears in c) but nevertheless, using (6), we see that it holds $\mathsf{Perf}_{\mathcal{B}_{n,p}}(h', c) = 1 - 2 \cdot 0.2^4 - 2 \cdot 0.2^4 + 4 \cdot 0.2^8 \approx 0.99361$. In other words, even if h' does not satisfy the stringent requirement of case (a) of our criterion for successful execution, since $\varepsilon = 0.01$ it nevertheless satisfies (4) which is really the goal of evolution, as it has performance at least $1 - \varepsilon$.

5 Experimental Results and Discussion

Using Algorithm 2 we perform experiments[5] for $\mathcal{B}_{n,p}$ distributions such that $p = j/10$, where $j \in \{1, 2, \ldots, 9\}$. By testing the values $p \in \{0.1, 0.2, 0.3, 0.5\}$ we can understand the rate of convergence when p is in the regime of proved convergence (based on [2]). Moreover, we can also use these numbers as baselines for forming conclusions regarding the rate of convergence when we perform experiments under distributions $\mathcal{B}_{n,p}$ that are characterized by values of p, when p is outside of the known regime of $(0, 1/3] \cup \{1/2\}$ where we have proved convergence.

5.1 Details on the Experimental Setup

Dimension of the Instance Space. In all of our experiments we set the dimension of our instance space to be equal to $n = 100$. This value of $n = 100$ allows a rich hypothesis space while at the same time it allows the repetitive execution of Algorithm 2 in a fairly reasonable amount of time for our experiments.

Target Sizes that we Test. For each p value mentioned above we generate targets that have sizes taken from the sets S_a, S_b, and S_c shown below:

$$\begin{cases} S_a = \{1, 2, q/4, q/2, 3q/4, q-1, q\} \\ S_b = \{q+1, q+\vartheta/2, q+\vartheta\} \\ S_c = \{q+\vartheta+1, q+\vartheta+(n-q-\vartheta)/4, \\ \qquad q+\vartheta+2(n-q-\vartheta)/4, \\ \qquad q+\vartheta+3(n-q-\vartheta)/4, n\}. \end{cases} \tag{12}$$

In particular, the target sizes from the set S_a are used for testing case (a) of Criterion 1, the target sizes from the set S_b are used for testing case (b) of Criterion 1, and the target sizes from the set S_c are used for testing case (c) of Criterion 1. Note that when $p < 1/2$, then $\vartheta = 0$. In such a case we consider the set S_b to be empty. That is, the target size that is indicated as having size $q+1$ is available in the set S_c where now $q + \vartheta + 1 = q + 0 + 1 = q + 1$.

Epochs (Repetitions). For each pair $(p, |c|)$ that we test, we perform 100 different *epochs (repetitions)* starting from the empty hypothesis (i.e., h is a bitstring of length n where each entry has the value of 0) until convergence. The epochs smooth the experimental results and allow us to better understand the average case of execution.

Numerical Values of the Parameters q, ϑ, and t. Table 1 summarizes the values that the parameters q, ϑ, and t obtain, when $p = j/10$ for $j \in \{1, 2, \ldots, 9\}$.

5.2 High-Level Summary of Results

We note that in every single one of our experiments we were able to satisfy Criterion 1. Table 2 presents the average number of iterations that was necessary so that we can satisfy Criterion 1 in every case that we tested.

[5] Source code available at: `https://gitlab.com/marina_pantia/evolvability_code`

Table 1: Values of q, ϑ, and tolerance t corresponding to each probability p that we tested in our experiments. Note that the values for the tolerance t should be multiplied by 10^{-6}. In every case the dimension of the instance space is $n = 100$.

p	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
q	3	4	5	7	9	12	16	26	55
ϑ	0	0	0	0	1	1	1	3	6
t (10^{-6})	13.3	17	26.2	8.05	3.81	3.79	6.62	2.33	1.09

Table 2: Average number of iterations until convergence (as computed using 100 epochs), depending on the target size and the probability used. Note that when $p < 1/2$, then $\vartheta = 0$ and therefore in these situations it is the case that $q + 1 > q + \vartheta/2$ as well as $q + 1 > q + \vartheta$. Therefore, some values may be repeated or appear out of order. However, for uniformity we keep these rows everywhere in accordance to the presentation of the sets S_a, S_b, and S_c in (12) from Section 5.1. In addition, when $p \in \{0.5, 0.6, 0.7\}$, then $\vartheta = 1$ and hence $q + 1 = q + \vartheta/2 = q + \vartheta$ as we use rounding in order to treat decimals (i.e., $\text{round}(\vartheta/2) = \text{round}(1/2) = 1$).

target size $\lvert c\rvert$	probability p								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	9355.3	8240.16	6109.52	4494.06	1846.62	292.16	263.67	231.91	264.03
2	18373.48	13234.84	8878.63	6757.35	3460.26	1275.46	1189.7	413.5	432.55
$q/4$	9355.3	8240.16	6109.52	6757.35	3460.26	2377.29	2285.82	1251.94	1147.72
$q/2$	18373.48	13234.84	8878.63	13631.78	6485.19	4388.57	4522.01	2340.74	1482.84
$3q/4$	18373.48	22613.09	22200.21	15894.43	11988.98	7071.61	6480.98	3718.52	2071.57
q	39011.15	33743.87	40434.23	37737.39	26736.85	15716.76	15174.42	6951.12	3826.76
$q+1$	12.56	13.56	14.82	15.67	24303.88	23337.84	28692.53	7449.52	3608.87
$q+\vartheta/2$	39011.15	33743.87	40434.23	37737.39	24303.88	23337.84	28692.53	9728.63	4553.08
$q+\vartheta$	39011.15	33743.87	40434.23	37737.39	24303.88	23337.84	28692.53	14651.58	9382.01
$q+\vartheta+1$	12.56	13.56	14.82	15.67	17.2	20.4	27.54	39.87	102.61
$q+\vartheta+(n-q-\vartheta)/4$	12.49	13.78	14.4	15.04	17.15	20.53	25.37	38.43	100.97
$q+\vartheta+2(n-q-\vartheta)/4$	12.67	13.68	14.93	14.79	16.88	20.8	25.24	38.15	92.61
$q+\vartheta+3(n-q-\vartheta)/4$	13.03	13.9	14.92	14.45	17.31	20.72	25.4	38.95	97.44
n	12.22	13.93	15.04	14.77	17.21	20.87	25.92	37.27	94.81

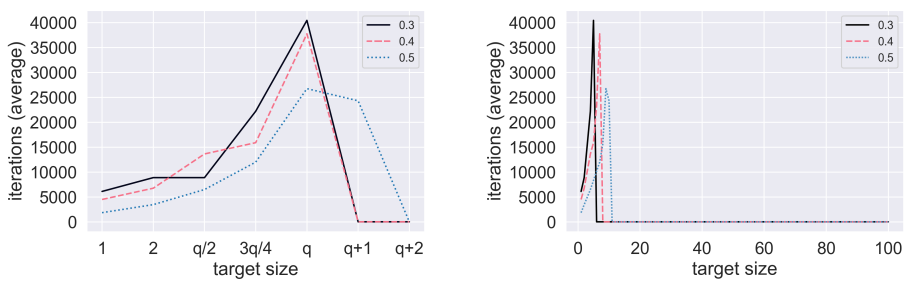
On the Convergence against Long Targets. As it can be seen from Table 2, case (c) in Criterion 1, corresponding to $|c| > q + \vartheta$, is perhaps the easiest one to accomplish. The intuitive reason from the work of [1] is that q and ϑ have been selected in such a way, so that *any* hypothesis h that has size q, *regardless* of its composition of *good* and *bad* variables among the q variables that it contains, will satisfy the equation $Perf_{\mathcal{B}_{n,p}}(h, c) \geq 1 - \varepsilon$. In particular, the intuitive idea is that c and h contain enough many variables and hence they make positive predictions on a small subspace of the n-dimensional hypercube $\{0, 1\}^n$. As a consequence c and h agree almost everywhere, since almost everywhere they make a negative prediction. As a further consequence, their correlation is at least $1 - \varepsilon$. Indeed, in our experiments when $|c| > q + \vartheta$, regardless of the underlying value of p that governs the distribution, we see that h satisfies the criterion $Perf_{\mathcal{B}_{n,p}}(h, c) \geq 1 - \varepsilon$ very quickly.

On the Convergence against Short and Medium Targets. Moreover, quite remarkably, based on the results shown in Table 2 for short and medium targets, the evolutionary mechanism that we study appears to be converging faster to a solution that satisfies Criterion 1 when p is outside of the known proved regime of $(0, 1/3] \cup \{1/2\}$. As characteristic examples one can compare the entries corresponding to $p = 0.4$ versus $p = 0.3$, or $p = 0.8$ versus $p = 0.5$.

The conclusions that we draw for the different values of p that we test are similar. Therefore in Section 5.3 below we focus on one particular case where $p = 0.4$, while in Section 5.4 we complement Table 2 and the discussion of Section 5.3 by showing boxplots with more refined information on the convergence rate of every case that we tested.

5.3 Details on the Convergence when $p = 0.4$

Figure 1a presents the average number of iterations against target sizes up to $q + 2$ when $p \in \{0.3, 0.4, 0.5\}$ and $n = 100$. Table 1 informs us that for $p = 0.3$,



(a) Iterations until convergence when $n = 100$ and $p = 0.4$, compared to $p = 0.3$ and $p = 0.5$ where it is known that the algorithm converges efficiently. On the horizontal axis we see target sizes as a function of q, so that we can study better case (a) of Criterion 1.

(b) Iterations until convergence when $n = 100$ and $p = 0.4$, compared to $p = 0.3$ and $p = 0.5$ where it is known that the algorithm converges efficiently. We can see the convergence rate when $p = 0.4$ across the entire spectrum of possible target sizes.

Fig. 1: Iterations until convergence when $n = 100$ and $p \in \{0.3, 0.4, 0.5\}$. In Figure 1a we focus in the situation where $|c| \in \{1, 2, q/2, 3q/4, q, q + 1\}$ which covers case (a) of Criterion 1, even though q is different for the different p values; see, e.g., Table 1. When $|c| > q + 1$ the convergence is very fast for all cases. In Figure 1b we see the complete picture for target sizes between 1 and $n = 100$.

$p = 0.4$, and $p = 0.5$ we have $q = 3$, $q = 4$, and $q = 5$ respectively. Even though these values of q are different, nevertheless, they all correspond to the situation where the target is short – case (a) of Criterion 1 – and for this reason we decided to put labels on the x axis that are related to q. Of course, since the q values are different, one can also consider a plot similar to Figure 1b and be able to see the complete picture for target sizes between 1 and $n = 100$ in each case. Regardless

if one uses Figure 1a or Figure 1b, we observe that (i) the algorithm converges for every target size, and (ii) the rate of convergence when $p = 0.4$ is very similar to what we observe for $p = 0.3$ and $p = 0.5$ where it has been proved that the algorithm converges. Similar results are discussed below for other values of p.

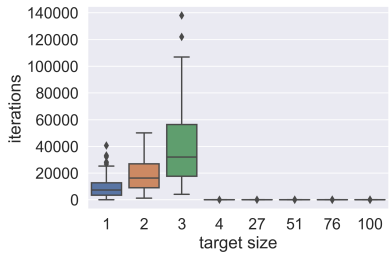
5.4 Further Details on the Experiments of Every $(p, |c|)$ Pair Tested

We complement Table 2 and the discussion of Section 5.3 by providing further statistics for the executions of Algorithm 2. Figure 2 presents *boxplots* regarding the number of iterations that was needed so that Criterion 1 was satisfied for every $p = i/10$, with $i \in \{1, 3, 4, 5, 6, 7, 8, 9\}$ and for every target size that belonged to one of the sets S_a, S_b, and S_c that were described in (12). (Due to space limitations we omitted the case for $p = 0.2$.) Each boxplot shows the median value for the execution of the algorithm regarding a particular $(p, |c|)$ pair. Furthermore, the thick part of the boxplot indicates the range of values that belong between the 25th and the 75th percentile. The whiskers are drawn so that they are in 1.5 times the inter-quartile range and finally in some cases we may also see some outliers which correspond to executions that took unexpectedly long/short time.

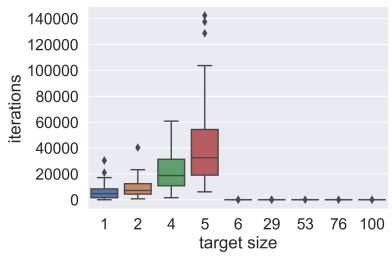
5.5 Discussion

As a summary, for every value $p = j/10$ with $j \in \{1, 2, \ldots, 9\}$ that characterizes a Bernoulli$(p)^n$ distribution, Algorithm 2 satisfied the goal of evolution and converges to a function that satisfies (4). This is true against *any target function that we tested*. Moreover, the average case analysis indicates that the running time needed to converge to such a good solution is in fact comparable to the running time that is needed (on average) by a simpler variant of this algorithm, that is obtained when the algorithm is tested against values of $p \in (0, 1/3] \cup \{1/2\}$, where it is has been proved that the algorithm converges efficiently [2].

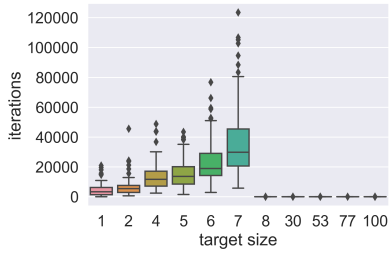
Implications. One first implication of these experimental results is that the (1+1)-EA variant that we examined, appears to be equally powerful as the swapping algorithm which provably evolves monotone conjunctions for Bernoulli$(p)^n$ distributions governed by any $p \in (0, 1)$ satisfying (4). A second implication is that the success criterion that we set beforehand (Criterion 1) indeed appears to capture fairly accurately what is happening on successful executions that also generate stable solutions. As a consequence of these two, a third implication is that the experimental convergence that we explored motivates future work for a formal approach on rigorously proving the convergence of the algorithm under Bernoulli$(p)^n$ distributions for values of $p \in (0, 1)$ outside of the known proved regime, which is $(0, 1/3] \cup \{1/2\}$ based on [2]. Fourth, somehow surprisingly, the experimental results suggest that the convergence of the algorithm is actually *faster* in the unknown regime compared to the known one, when the target is short or medium (i.e., for target sizes that are expected to be difficult); e.g., compare the results between $p = 0.8$ and $p = 0.5$ in Table 2.



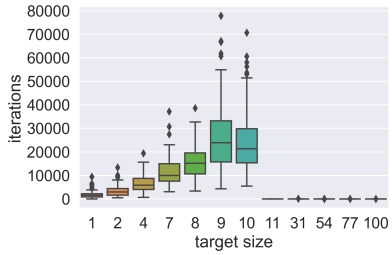
(a) $p = 0.1$, $q = 3$ and $\vartheta = 0$.



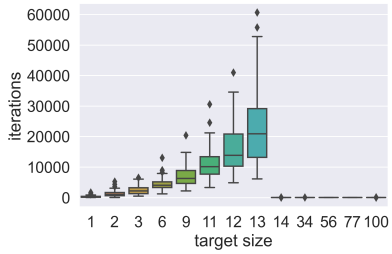
(b) $p = 0.3$, $q = 5$ and $\vartheta = 0$.



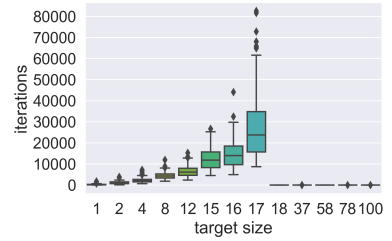
(c) $p = 0.4$, $q = 7$ and $\vartheta = 0$.



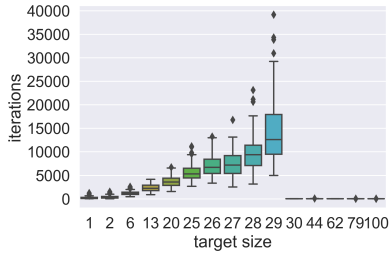
(d) $p = 0.5$, $q = 9$ and $\vartheta = 1$.



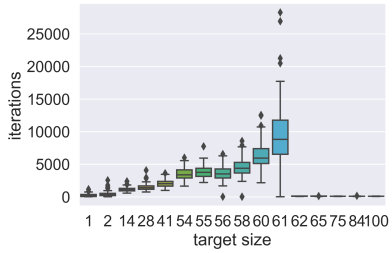
(e) $p = 0.6$, $q = 12$ and $\vartheta = 1$.



(f) $p = 0.7$, $q = 16$ and $\vartheta = 1$.



(g) $p = 0.8$, $q = 26$ and $\vartheta = 3$.



(h) $p = 0.9$, $q = 55$ and $\vartheta = 6$.

Fig. 2: Boxplots of iterations needed until convergence when $n = 100$ and for probabilities $p \in \{0.1, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$. The x-axis corresponds to target sizes generated according to the sets S_a, S_b, and S_c that are presented in Section 5.1; these sizes depend on p as they depend on the parameters q and ϑ which ultimately depend on p. Furthermore, when $|c| > q + \vartheta$, the convergence is very fast and thus the deviation from the median of the iterations is insignificant.

6 Conclusions

We studied the evolvability of monotone conjunctions under Bernoulli$(p)^n$ distributions using the standard mutation mechanism that appears in (1+1) EAs. We extended the algorithm introduced in [2] by drawing inspiration from the convergence properties of the swapping algorithm under such distributions [1]. Our experiments indicate that the extension we proposed allows the formation of hypotheses that approximate well *any target function* c under *arbitrary* Bernoulli$(p)^n$ distributions since the computed solutions in our experiments were *stable* and more importantly *satisfied the goal of evolution required by (4) in every combination* $(p, |c|)$ *that we tested*. In the future, it would be interesting to prove rigorously this experimental result, as well as explore the convergence of this (1+1) EA-based mutation mechanism under distributions beyond Bernoulli$(p)^n$.

References

1. Diochnos, D.I.: On the Evolution of Monotone Conjunctions: Drilling for Best Approximations. In: ALT. pp. 98–112 (2016)
2. Diochnos, D.I.: On the Evolvability of Monotone Conjunctions with an Evolutionary Mutation Mechanism. Journal of Artificial Intelligence Research **70**, 891–921 (2021)
3. Diochnos, D.I., Turán, G.: On Evolvability: The Swapping Algorithm, Product Distributions, and Covariance. In: SAGA. Lecture Notes in Computer Science, vol. 5792, pp. 74–88. Springer (2009)
4. Droste, S., Jansen, T., Wegener, I.: On the analysis of the (1+1) evolutionary algorithm. Theoretical Computer Science **276**(1-2), 51–81 (2002)
5. Feldman, V.: Evolvability from learning algorithms. In: STOC. pp. 619–628 (2008)
6. Kalkreuth, R., Droschinsky, A.: On the Time Complexity of Simple Cartesian Genetic Programming. In: IJCCI. pp. 172–179. ScitePress (2019)
7. Kanade, V.: Evolution with Recombination. In: FOCS. pp. 837–846 (2011)
8. Koza, J.R.: Genetic programming - on the programming of computers by means of natural selection. Complex adaptive systems, MIT Press (1993)
9. Lissovoi, A., Oliveto, P.S.: On the Time and Space Complexity of Genetic Programming for Evolving Boolean Conjunctions. Journal of Artificial Intelligence Research **66**, 655–689 (2019)
10. Mambrini, A., Oliveto, P.S.: On the Analysis of Simple Genetic Programming for Evolving Boolean Functions. In: EuroGP. Lecture Notes in Computer Science, vol. 9594, pp. 99–114. Springer (2016)
11. Pitt, L., Valiant, L.G.: Computational limitations on learning from examples. Journal of the ACM **35**(4), 965–984 (1988)
12. Reyzin, L.: Statistical Queries and Statistical Algorithms: Foundations and Applications. CoRR **abs/2004.00557** (2020)
13. Ros, J.P.: Learning Boolean Functions with Genetic Algorithms: A PAC Analysis. In: FOGA. pp. 257–275 (1992)
14. Snir, S., Yohay, B.: Prokaryotic evolutionary mechanisms accelerate learning. Discrete Applied Mathematics **258**, 222–234 (2019)
15. Valiant, L.G.: A theory of the learnable. Communications of the ACM **27**(11), 1134–1142 (1984)
16. Valiant, L.G.: Evolvability. Journal of the ACM **56**(1), 3:1–3:21 (2009)