Runtime Analysis of Random Local Search on Jump Function with Reinforcement Based Selection of Auxiliary Objectives

Denis Antipov, Arina Buzdalova
ITMO University
49 Kronverkskiy av., Saint-Petersburg, Russia, 197101
Email: antipovden@yandex.ru, abuzdalova@gmail.com

***Abstract*—In certain optimization problems, aside from the target objective, auxiliary objectives can be used. These auxiliary objectives may be either helpful or not. Often we can not determine whether an auxiliary objective is helpful. In this work we consider the EA+RL method that dynamically chooses auxiliary objectives in random local search using reinforcement learning. This method's runtime has already been theoretically analysed on different monotonic functions, and it was shown that EA+RL can exclude harmful auxiliary objectives from consideration. EA+RL has also shown good results on different real-world problems. However, it has not been theoretically analysed whether this method can efficiently optimize non-monotonic functions using simple evolutionary algorithms and reinforcement learning agents.**

In this paper we consider optimization of the non-monotonic JUMP function with the EA+RL method. We use two auxiliary objectives. One of them is helpful during the first phase of optimization and another one is helpful during the last phase. On other stages they are constant, so they neither help nor slow optimization down. We show that EA+RL has at least $\Omega(\frac{\ell}{n})$ probability of solving this problem in polynomial time using random local search, which is impossible for the conventional random local search without learning. We also propose a modification of EA+RL that is guaranteed to find the optimum.

I. INTRODUCTION

Single-objective optimization can often be accelerated by using auxiliary objectives [2], [13], [16], [17], [19]. There exist different approaches of using auxiliary objectives [7], [12], [13]. Usually auxiliary objectives help to pass through plateaus or to escape from local optima.

There are different ways to obtain auxiliary objectives. Many theoretical works analysed different problems where auxiliary objectives arise from the decomposition of the target objective [2], [11], [16], [17]. Sometimes it is possible to generate auxiliary objectives [5], but in this case it is a challenging task to learn which of the generated objectives help the optimizer and which ones prevent it from reaching its goal. Moreover, sometimes helpfulness of an auxiliary objectives can change during optimization. It caused the uprise of methods of dynamic selection of auxiliary objectives [7], [12]. One of the methods that can dynamically learn which objectives help the optimizer reach the global optimum is EA+RL [20].

A. EA+RL: Selecting Objectives with Reinforcement Learning

EA+RL is a method of selecting objectives in multiobjective optimization problems with only one target objective and a bunch of auxiliary objectives that may or may not improve the optimization process. EA+RL has two components: a single-objective optimization algorithm and a learning agent. On each iteration of EA+RL, the optimization algorithm performs one iteration optimizing the objective selected by the learning agent, then it sends the reward of this iteration (usually the difference of target objective values) to the learning agent which updates its selection strategy.

The EA+RL method has shown good results on practical problems [5] and has been analysed theoretically on different model problems [1], [4]. In [4] authors investigated the ability of the EA+RL method to dynamically adapt to changing helpfulness of the auxiliary objectives. However, they achieved only preliminary results. The lack of understanding of the behaviour of the EA+RL method in the situation of changing helpfulness of the objectives is not the only gap in theoretical knowledge about this method. Another question that has not been observed in the theoretical field is how efficiently the EA+RL method can escape from local optima.

B. Research Questions

We consider two main research questions regarding EA+RL efficiency: (i) How successful is EA+RL in escaping local optima? (ii) What is EA+RL's behaviour in the situation, when there is more than one auxiliary objective and each objective is helpful only in particular phase of optimization while being useless during the other phases.

The first question is a common question for most random search heuristics. And the second question is specific for online methods of objective selection, including EA+RL. The situation, when auxiliary objectives can change their helpfulness is quite common in practical problems [3], [15]. And the desirable behaviour of the EA+RL method in such situations is to adapt to the changes and to relearn to select helpful objectives.

C. Optimizing JUMP Function Using the EA+RL Method

To answer the first research question, we should consider optimizing a non-monotonic function by the EA+RL method.

978-1-5090-4601-0/17/$31.00

One of the simplest examples of a non-monotonic function is the JUMP function that was described in [10], [14]. JUMP is a pseudoboolean function, i.e. its domain is a space of binary strings of a fixed length. We analyse the runtime of EA+RL method optimizing JUMP to get more insights about the ability of this method to escape local optima.

The previous works that have considered the JUMP function in the context of evolutionary computation have already shown the complexity of this function. In [10] it has been proven that $(1+1)$ EA needs $\Theta(n^{l+1})$ fitness evaluations for every integer $l \in [1, \lfloor \frac{n-1}{2} \rfloor]$. It has been also proven that a certain EA with crossover can solve JUMP with $O(n \log^3 n)$ fitness evaluations if l is a constant and with $O(n^{2c+1} \log n)$ fitness evaluations if $l = \lceil c \log n \rceil$, where c is a constant. Summing up the results of all previous works, we can say that the JUMP function is not easy to optimize with evolutionary algorithms.

Also, unbiased [8] and unrestricted [6] black-box complexity of JUMP has been considered. The work [6] has shown the best known algorithms that solve JUMP in linear time, however, they were not evolutionary algorithms.

In our work we consider the EA+RL method that optimizes JUMP function using two auxiliary objectives that change their helpfulness during the optimization. Let us notice that we do not focus on the methods of obtaining helpful objectives in practical problems, our aim is to answer the second research question, i.e. analyse how EA+RL relearns helpfulness of auxiliary objectives.

The structure of the rest of this paper is following. In Section II we introduce the target objective and auxiliary objectives for the EA+RL method, as well as the algorithm itself. In Section III we show that this algorithm has finite runtime only with probability not greater than 0.5 and show the upper bounds on the algorithm's runtime in cases when it finds the optimum. In Section IV we improve this method, so that this modification always has finite expected runtime, and we prove an upper bound on its runtime. Next, in Section V we compare the obtained theoretical bounds with empirical results. In the conclusion we sum up our results and discuss the possibilities for further research.

II. PROBLEM STATEMENT

In this paper we analyse the expected runtime of an optimization algorithm with an objective selection heuristic. First we will introduce the algorithm itself and then the objectives that the algorithm operates with.

A. Algorithm Description

The EA+RL method consists of two interacting parts: an optimization algorithm and a reinforcement learning *agent*. EA+RL is an iterative algorithm. On every iteration, the learning agent chooses one objective to be optimized. Then the learning agent sends the chosen objective to the optimization algorithm, and the optimization algorithm performs one iteration optimizing the selected objective. After that, the optimization algorithm sends to the learning agent the *reward*, that depends on the difference of the target objective values in the new generation and the previous one, and the *state* of the algorithm, that depends on the new generation. Based on this data, the learning agent updates its selection strategy, and the next iteration begins.

Algorithm 1 RLS controlled by EA+RL using the Q-learning algorithm

$x \leftarrow$ current individual, vector of N zeros
$Q \leftarrow$ transition quality matrix, $N \times 3$, filled with zeros
MUTATE$(x) \leftarrow$ mutation operator: inverts one random bit
while JUMP$(x) < N$ **do**
  $s \leftarrow$ JUMP(x)
  $y \leftarrow$ MUTATE(x)
  $f \leftarrow$ the random objective such that $Q(s, f)$ is maximum (may be JUMP, LEFTBRIDGE or RIGHTBRIDGE)
  if $f(y) \geq f(x)$ **then**
    $x \leftarrow y$
  end if
  $s' \leftarrow$ JUMP(x)
  $r \leftarrow s' - s$
  $Q(s, f) \leftarrow (1 - \alpha)Q(s, f) + \alpha(r + \gamma \cdot \max_j Q(s', j))$
end while

We have chosen the random local search (RLS) as an optimization algorithm. There is a single individual in a population encoded as a bit string. On each iteration RLS switches one random bit in the individual and substitutes the old individual with the new one if and only if the fitness of the new individual is not worse than the fitness of the old one.

The learning agent uses the reinforcement learning algorithm called *Q-learning*. Agent stores the quality of each objective a in each state of the algorithm s as the function $Q(s, a)$. States of the algorithm correspond to values of the target function calculated each time on a current individual. In the state s, the learning agent chooses the objective a that has the largest $Q(s, a)$. If there are several such objectives, agent chooses one objective uniformly at random.

Just after the initialization $Q(s, a) = 0$ for all states and objectives. After the agent receives the reward r from the optimization algorithm, it modifies Q in the following way: $Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_j Q(s', j))$, where $\alpha \in (0, 1]$ is a learning rate and $\gamma \in (0, 1)$ is a discount factor.

The pseudocode of the algorithm is shown in Algorithm 1, the objectives used there are described in Section II-B.

B. Definition of Objectives

In this paper we consider *pseudo-Boolean* functions that map bit strings of a fixed length to integers. The most famous class of such functions is ONEMAX [21]. ONEMAX(x) function has a hidden bit string z and returns the number of the bits coinciding in x and z. The considered algorithm does not work differently with different z, so z can be set as a bit string of all ones. In the rest of the paper ONEMAX(x) actually equals the number of ones in x.

Another considered class of pseudo-Boolean functions is JUMP, which has an integer parameter $l \in [1, \lfloor \frac{n-1}{2} \rfloor]$, where

n is the length of the bit string in the argument. It is equal to ONEMAX when $\text{ONEMAX}(x) \in [l+1, n-l-1] \cup \{n\}$, otherwise it returns zero. Formally, it can be defined in the following way:

$$\text{JUMP}(x) = \begin{cases} \text{ONEMAX}(x), & l < \text{ONEMAX}(x) < n-l, \\ n, & \text{ONEMAX}(x) = n, \\ 0, & \text{in the rest of the cases.} \end{cases}$$

We can see from the definition that JUMP has one global optimum in the string x that consists only of one-bits. Also it has local optima in all x that have exactly $n-l-1$ one-bits.

Parameter l is bounded above by the value $\lfloor \frac{n-1}{2} \rfloor$, as for greater values JUMP turns into the needle function (the one that has a non-zero value only for single bit string x). This case can not help to find an answer to the both research questions formulated in Section I-B: it does not have any local optima to observe EA+RL behaviour in it, also the whole search space except the optimal solution is a plateau, so EA+RL method can not perform any learning in such situation.

We also have two auxiliary objectives that the $EA + RL$ method can choose during the optimization. They are LEFTBRIDGE and RIGHTBRIDGE. We can say that they complement the target function:

$$\text{LEFTBRIDGE}(x) = \begin{cases} \text{ONEMAX}(x), & \text{ONEMAX}(x) \leq l, \\ 0, & \text{otherwise} \end{cases}$$

$$\text{RIGHTBRIDGE}(x) = \begin{cases} \text{ONEMAX}(x), & \text{ONEMAX}(x) \geq n-l, \\ 0, & \text{otherwise} \end{cases}$$

The choice of these objectives is justified by the second research question whether EA+RL can relearn the helpfulness of the objectives. LEFTBRIDGE seems to help the optimization process when a current individual x has $\text{ONEMAX}(x) \leq l$ and RIGHTBRIDGE seems to be helpful when $\text{ONEMAX}(x) \geq n-l$. Notice that both auxiliary objectives and the target objective has the same underlying bit string $z = (1, .., 1)$. This fact is also justified by the need to answer the second research question, as other underlying bit strings could make these objectives useless or even harmful throughout the optimization process. Moreover, in practical problems designers of algorithms usually try to find auxiliary objectives that complement the target objective, what is intuitively equal to the same underlying bit string.

The same parameter l for all three considered objectives can be explained in the following way. If the parameter l was greater for the auxiliary objectives, this case would be easier for the EA+RL method, as there would be zones in a search space, where at least two objectives are helpful. At the same time, lesser l would cause the presence of the zones where only random walk can be performed and selection of objectives does not matter, as every objective would have a plateau in those zones. So we decided to consider the "hardest case of all the sensible cases".

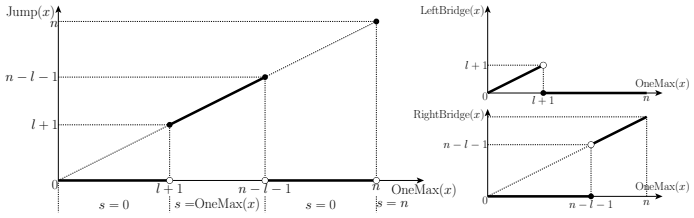

Fig. 1. Illustration of the JUMP function, auxiliary objectives and reinforcement learning states

III. RUNTIME ANALYSIS OF EA+RL

Runtime of an evolutionary algorithm is the number of fitness evaluations performed by the algorithm before it finds the optimal solution. In our case, the runtime equals the number of iterations, as the algorithm performs exactly one fitness evaluation on each iteration. In this section we evaluate the upper bounds on the expected runtime of the algorithm that has been described in Section II-A. In order to do this, we first split a typical run of the algorithm into three phases:

1) From the first iteration till the moment when a bit string x such that $\text{JUMP}(x) \neq 0$ is found for the first time.
2) From the end of the first phase till the moment when the algorithm finds a bit string x such that $\text{ONEMAX}(x) = n-l$ for the first time.
3) From the end of the second phase till the moment of finding the optimal bit string.

We illustrate the JUMP function in Fig. 1. During the first phase the algorithm is always in the state $s = 0$. The second phase begins with the state $s = l+1$ and the algorithm can visit states $s \in [l+1, n-l-1] \cup \{0\}$. On the third phase the algorithm is in the state $s = 0$ again, but in the worst cases it can return to the states visited in the second phase. This case will be considered separately.

We will perform the analysis of each phase separately to find the upper bounds on the expected number of iterations for each phase. After that we will use the linearity of mathematical expectation and sum up the expected runtime of every phase to get the expected runtime of the algorithm: $T = T_1 + T_2 + T_3$.

A. The First Phase: Overcoming The First Plateau

In the first phase, the algorithm wades through the first plateau of JUMP. In this phase, the algorithm is stuck in the state $s = 0$ and tries to escape it. As the learning agent has not learned anything yet, it has all $Q(0, i) = 0$ for every i. It means that the learning agent chooses objectives uniformly. But we know that two objectives has a plateau on the first phase – JUMP and RIGHTBRIDGE. So the optimization algorithm can accept both bit strings with more one-bits and with less one-bits, if one of these objectives has been chosen. Another objective LEFTBRIDGE does not let the optimization algorithm accept a bit string x which has less one-bits than the parent individual. The latter fact helps the algorithm to get to the end of the first phase faster.

To find the expected runtime for this phase we will use the multiplicative drift theorem [9].

Theorem 1 (First phase runtime)**.** *The expected runtime of the first phase is at most* $\frac{3}{2}n(\ln(l+1)+1)$.

Proof: Consider the potential function $\Phi(x) = l + 1 - \textsc{OneMax}(x)$. It becomes zero exactly at the moment of the end of the first phase. $\Phi(x)$ has the minimum non-zero value of 1 and the maximum value of $(l+1)$. Also to use the drift theorem we need to consider the expectation of difference of the potential function. Let Φ_t be the value of $\Phi(x)$ on the t-th iteration. Consider the expected difference of Φ_t and Φ_{t+1} when $\textsc{OneMax}(x) = i \neq l$:

$$\begin{aligned} E(\Phi_t - \Phi_{t+1}|\Phi_t) &= \frac{n-i}{n}\cdot 1 - \frac{2i}{3n}\cdot 1 = \frac{3n-5i}{3n} \\ &\geq \frac{6l-5i}{3n} \geq \frac{3l-3i+3}{3n} = \frac{\Phi_t}{n}. \end{aligned} \tag{1}$$

The last inequation is valid, as $6l - 5i = 3l - 3i + (2l - 2i) + l \geq 3l - 3i + 3$. For the $\textsc{OneMax}(x) = l$ (when also $\Phi_t = 1$) we have:

$$\begin{aligned} E(\Phi_t - \Phi_{t+1}|\Phi_t = 1) &= \frac{2(n-i)}{3n}\cdot 1 - \frac{2i}{3n}\cdot 1 \\ &= \frac{2}{3}\frac{n-2i}{n} \geq \frac{2\Phi_t}{3n} \end{aligned} \tag{2}$$

As it follows from Eq. 1 and Eq. 2, $E(\Phi_t - \Phi_{t+1}|\Phi_t) \geq \delta\Phi_t$, where $\delta = \frac{2}{3n}$. Now applying the multiplicative drift theorem, we can estimate the expected runtime of the first phase: $T_1 \leq \frac{1}{\delta}(\ln(\Phi_{\max}/\Phi_{\min}) + 1) = \frac{3}{2}n(\ln(l+1)+1)$. ∎

There are two possible scenarios of the end of the first phase. It can happen either on the iteration when the learning agent has chosen RIGHTBRIDGE function or on the iteration when the learning agent has chosen the JUMP function. Though it does not affect the second phase of the algorithm, it causes the totally different behaviour on the last phase of the algorithm that will be considered in Section III-D.

B. The Second Phase: Learning Which Objectives are Helpful

During the first phase, the value of the target objective was not changing, so the agent did not learn anything, thus it chose objectives uniformly at random. In the second phase, the learning plays a greater role in the choices of the agent. To show it, we need the following lemma.

Lemma 1 (Learning lemma)**.** *If the learning parameters* α *and* γ *satisfy the inequality* $2\alpha(1-\gamma) > 1$*, then the algorithm will visit each learning state* $s \neq 0$ *at most 5 times.*

To prove the above lemma, we first need to prove two auxiliary lemmas. The first lemma states the bounds on the values of $Q(s,a)$ function which are stored in the learning agent. The second lemma demonstrates how learning helps to determine the objectives that can slow optimization down.

Lemma 2 (Maximum Q)**.** *During the second phase, the maximum value of* $Q(s,a)$ *for every learning state* $s \neq 0$ *and every auxiliary objective* a *is not greater than* $\frac{1}{1-\gamma}$ *and* $Q(0,a)$ *for every* a *is not greater than* $l + \frac{1}{1-\gamma}$.

Proof: We will use induction to prove this lemma. Initially, all the values of $Q(s,a)$ are zeros that is less then both $\frac{1}{1-\gamma}$ and $l + \frac{1}{1-\gamma}$.

Now let us assume that on some iteration the conditions of lemma are satisfied. It is easy to show that they will be also true on the next iteration, if the current iteration is not the last one on the second phase. If $s \neq 0$ and the next state $s' \neq 0$, then the algorithm will modify only one $Q(s,a)$:

$$\begin{aligned} Q(s,a) &= (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_a Q(s',a)) \\ &\leq (1-\alpha)\frac{1}{1-\gamma} + \alpha\left(1 + \gamma\frac{1}{1-\gamma}\right) = \frac{1}{1-\gamma} \end{aligned} \tag{3}$$

If the current state $s = l+1$ and the next state $s' = 0$, then:

$$\begin{aligned} Q(l+1,a) &= (1-\alpha)Q(l+1,a) \\ &\quad + \alpha(-(l+1) + \gamma \max_a Q(0,a)) \\ &\leq \frac{1-\alpha}{1-\gamma} + \alpha\left(-(l+1) + \gamma\left(l + \frac{1}{1-\gamma}\right)\right) \\ &= \frac{1 + \alpha(1-\gamma)(\gamma l - l - 2)}{1-\gamma} \leq \frac{1}{1-\gamma} \end{aligned} \tag{4}$$

If the current state $s = 0$ and the next state $s' = l+1$, then:

$$\begin{aligned} Q(0,a) &= (1-\alpha)Q(0,a) \\ &\quad + \alpha(-(l+1) + \gamma \max_a Q(l+1,a)) \\ &\leq (1-\alpha)\left(l + \frac{1}{1-\gamma}\right) + \alpha\left((l+1) + \frac{\gamma}{1-\gamma}\right) \\ &= l + \frac{1}{1-\gamma}. \end{aligned}$$

In the last case, when $s = s'$:

$$\begin{aligned} Q(s,a) &= (1-\alpha)Q(s,a) + \alpha\gamma \max_a Q(s,a) \\ &< \max_a Q(s,a). \end{aligned}$$

∎

Lemma 3 (Obstructive objective)**.** *If* $2\alpha(1-\gamma) > 1$*, then if the algorithm in the second phase decreases the number of ones in the state* $s \neq 0$*, the same objective in this state will never be chosen again.*

Proof: To start with, let us notice that $Q(s, \textsc{Jump}) \geq 0$ for each state, because when the JUMP function is selected, the reward can not be negative, therefore $\max_a Q(s,a) \geq Q(s, \textsc{Jump}) \geq 0$.

Next, notice that if the algorithm passes from the state s to the state s' ($0 < s' < s$), then in Eq. 3 $r = -1$ and thus $Q(s,a) < 0$. And if the algorithm passes from the state $s = l+1$ to the state $s' = 0$, then the last inequation in Eq. 4 can be strengthened using the Lemma condition $2\alpha(1-\gamma) > 1$: $Q(l+1,a) \leq \frac{1+\alpha(1-\gamma)(\gamma l - l - 2)}{1-\gamma} < 0$.

Therefore, if the number of one-bits in the current individual was decreased in the state $s \neq 0$ after an objective a was selected to be optimized (notice that it can only be LEFTBRIDGE or RIGHTBRIDGE), then the inequality holds: $Q(s,a) < 0 \leq$

$Q(s, \text{JUMP})$. Along with the fact that $Q(s, \text{JUMP})$ will always stay non-negative, it means that the objective a will never be selected in the state s.

■

Now we are ready to prove Lemma 1.

Learning Lemma: Using Lemma 3, we can say that in every state $s \neq 0$ the number of one-bits in the current bit string can be decreased only two times. This means that the state s can be reached from the state $s+1$ only twice. The state s can also be reached from $s-1$ only three times, as it is possible to raise from the state $s-1$ only one time more than it is possible to fall there from s. Thus, the state s can be visited only 5 times.

■

After we have proven Lemma 1, we can find the upper bound on the runtime of the second phase using linearity of mathematical expectation.

Theorem 2 (Second phase runtime)**.** *The expected runtime of the second phase is not greater than* $5n \ln \frac{n-l-1}{l+1} + 2n + 15$.

Proof: The expected runtime of the second phase is the sum of the expected runtimes that algorithm spends in each state during the second phase. Algorithm visits states $s = l+1, ...n-l-1$ and $s = 0$ during the second phase.

For the states $s = l+1, ...n-l-2$ the probability to escape $p_{\text{esc}} \geq (n-s)/n$. Thus, the expectation of the number of iterations that algorithm spends in the state s during the second phase is the expectation of the number of iterations during one visit multiplied by the number of visits that has been stated in Lemma 1 to be not greater than 5: $T_s \leq 5n/(n-s)$.

For the state $s = 0$ we can estimate the upper bound on the iterations spent there per visit using the drift theorem and the potential function Φ again. When algorithm falls to the state $s = 0$ during the second phase, the number of ones in the current individual equals l, thus $\Phi_0 = 1$. The expected difference of the potential function can be upper bounded by Φ_t/n. Therefore, the algorithm will spend not more than n iterations per visit. A simple consequence from Lemma 3 is that there will be not more than two visits of the state $s = 0$ during the second phase, so $T_{s=0} \leq 2n$.

Finally, the state $s = n-l-1$ is a special case. If during the second phase there will be two decreases of the number of one-bits performed in this state, then the algorithm will get stuck in this state, and its runtime will be infinite. This case is considered in more details in Section III-C. Otherwise, the probability to leave this state is not less than $1/3$ and thus the algorithm will spend there not more than 3 iterations per visit. There can not be more than 5 visits, so totally algorithm spends in the state $s = n-l-1$ not more than 15 iterations during the second phase.

Summing up the received expectations we receive the result: $T_2 \leq 5 \sum_{s=l+1}^{n-l-2} \frac{n}{n-s} + 15 + 2n < 5n \ln \frac{n-l-1}{l+1} + 2n + 15$. ■

C. The Probability to Finish the Second Phase

The algorithm gets stuck in the state $s = n-l-1$ if it falls from the state $s = n-l-1$ to the state $s = n-l-2$ twice before finishing the second phase. To find the probability p of avoiding this case when the algorithm came to the state $s = n-l-1$ at the first time, we can write the following equation:

$$p = p_{\text{choose}} \cdot p_{\text{flip}+1} + p_{\text{choose}} \cdot p_{\text{flip}-1} \cdot p' + p_{\text{ntl}} \cdot p. \quad (5)$$

Here $p_{\text{choose}} = \frac{2}{3}$ is the probability to choose either LEFTBRIDGE or RIGHTBRIDGE, $p_{\text{flip}+1} = \frac{l+1}{n}$ and $p_{\text{flip}-1} = \frac{n-l-1}{n}$ are the probabilities to increase and decrease the number of one-bits during mutation respectively, $p_{\text{ntl}} = \frac{1}{3}$ is the probability not to leave the state $s = n-l-1$ and p' is the probability to finish the second phase if the algorithm learned not to choose RIGHTBRIDGE or LEFTBRIDGE.

The probability p' is the sum of the probability to move from the current state $s = n-l-1$ to the state $s = 0$ and the probability to stay in the current state and leave it during the future iterations: $p' = \frac{l+1}{2n} + \frac{1}{2}p'$, thus, $p' = \frac{l+1}{n}$ If we substitute p' into Eq. 5 with this value, we will get $p = \frac{l+1}{n}\left(1 + \frac{n-l-1}{n}\right) = 1 - \frac{(n-l-1)^2}{n^2}$.

D. The Third Phase: Probably Getting Stuck

As we said in Section III-A, behaviour of the algorithm during the third phase depends on which objective was selected by the learning agent in the end of the first phase.

Theorem 3. *If the learning agent has chosen* RIGHTBRIDGE *in the end of the first phase, then the expected runtime of the third phase* $T_3 \leq n(\ln l + 1)$.

Proof: As RIGHTBRIDGE has been chosen in the end of the second phase, it will always be chosen in the state $s = 0$, as $Q(0, \text{JUMP}) = Q(0, \text{LEFTBRIDGE}) = 0$ and $Q(0, \text{RIGHTBRIDGE}) > 0$. And during the third phase algorithm does not leave the state $s = 0$. RIGHTBRIDGE does not let the algorithm accept bit strings with less one-bits than in the current string, and the expected number of iterations until getting an individual with greater number of one-bits is $\frac{n}{n-i}$, where i is the number of one-bits in the current string. Summing up this estimation for all possible values of i, we get the expected runtime of the third phase: $T_3 = \sum_{i=n-l}^{n-1} \frac{n}{n-i} \leq n(\ln l + 1)$. ■

In the case of selecting RIGHTBRIDGE in the end of the first phase we can state a theorem:

Theorem 4. *The total expected runtime of the algorithm* $T \leq \frac{5}{2}n(\ln(l+1)+1) + 5n \ln \frac{n-l+1}{l+1} + 2n + 15 = O(n \ln n)$ *for every integer value of* $l \in [1, \lfloor \frac{n-1}{2} \rfloor]$, *if* RIGHTBRIDGE *has been chosen in the end of the first phase.*

This theorem simply follows from Theorems 1, 2 and 3.

In the other case, when the JUMP function was chosen in the end of the first phase, the algorithm will return to the state $s = n-l-1$ with very high probability. This happens because the probability to decrease the number of one-bits is greater than the probability to increase it in the third phase. And as $Q(0, \text{JUMP}) > 0$, the JUMP function will be chosen as the objective to be optimized, which lets the algorithm accept bit strings with less number of one-bits in the third phase. After

the state $s = n - l - 1$ is visited for the second time, the algorithm gets stuck in this state, because both LEFTBRIDGE and RIGHTBRIDGE have negative $Q(n-l-1, a)$.

Therefore, we have shown that the probability to get stuck in the state $s = n - l - 1$ is $1 - \frac{(n-l-1)^2}{n^2}$ in the second phase and $\frac{1}{2}$ in the third phase. The modified version of the algorithm that finds the global optimum in all cases will be introduced in the next section.

IV. MODIFICATION OF THE EA+RL METHOD

Consider a simple modification of the algorithm. According to this modification, the algorithm is restarted, if the value of the target objective has not changed over m iterations. A modified EA+RL method based on a similar idea was considered in [18].

In our modification m is $(c+1)n(\ln n + 1)$, where c is some constant that will be chosen in the end of this section. After the restart, the optimization algorithm forgets the current individual and the learning agent forgets everything it has learned. The main result of this section is presented in Theorem 5.

Theorem 5 (Runtime of EA+RL modification)**.** *The upper bound on the expected runtime of the algorithm modification is* $O\left(\frac{n^2 \log n}{l}\right)$.

A. Probability of Restart

To prove Theorem 5, we need to prove the following lemma first.

Lemma 4. *The probability that the algorithm will find an optimum before the restart occurs* $p_{end} \geq C_0 \left(1 - \frac{(n-l-1)^2}{n^2}\right)$, *where* $C_0 = (1 - e^{-\frac{2}{3}c+\frac{1}{3}})\frac{1}{2}\left(1 - \frac{1}{e^c n^{c+1}}\right)\left(\frac{1}{3}\right)^{\frac{5}{(e^2 n)^{c+1}}}(1 - e^{-c})$.

Proof: To find this probability, let us consider different scenarios of the restart.

At first, the restart can occur during the first phase, if this phase took too much time. As the multiplicative drift theorem was used, we can also use tail bounds to find the probability that the first phase will take more than m iterations:

$$\begin{aligned} p_1 &= \Pr[T_1 \geq m | \Phi_0 = l+1] \\ &= \Pr[T_1 \geq (c+1)n(\ln n + 1) | \Phi_0 = l+1] \\ &\leq \Pr\left[T_1 \geq \left(\frac{2}{3}c - \frac{1}{3} + 1\right)\frac{3}{2}n\left(\ln(l+1)+1\right)\right] \leq e^{-\frac{2}{3}c+\frac{1}{3}}. \end{aligned}$$

Next, if the algorithm has not been restarted during the first phase, there are two cases of passing to the second phase. If the algorithm chooses JUMP with the probability of $1 - p_{1\to 2} = 1/2$, then the current run will end with a restart, because the algorithm will get stuck in the local optimum in the end of the second phase.

The probability to restart during a second phase can be estimated as $p_2 = p_{s=0}^2 \prod_{i=l+1}^{n-l-1} p_{s=i}^5$, where $p_{s=i}$ is the probability to restart while visiting the state $s = i$. Powers correspond to the maximum number of visits for each state. We can bound $p_{s=0}$ by tail bounds from the drift theorem using Φ as a potential function: $p_{s=0} \leq 1 - \frac{1}{e^c n^{c+1}}$. $p_{s=i} \geq (1-(i/n)^m)$, as it is a probability that algorithm will not stay in the state $s = i$ for m iterations, and the probability not to leave the current stay is not less than i/n for any iteration.

So the production can be bounded below in the following way:

$$\begin{aligned} &\prod_{i=l+1}^{n-l-1}\left(1 - \left(\frac{i}{n}\right)^m\right)^5 \geq \left(1 - \left(\frac{n-l-1}{n}\right)^m\right)^{5(n-2l-1)} \\ &\geq \left(1 - e^{-\frac{m(l+1)}{n}}\right)^{5n} = \left(1 - (en)^{-(l+1)(c+1)}\right)^{5n} \\ &\geq (1/3)^{\frac{5n}{(en)^{(l+1)(c+1)}}} \geq (1/3)^{\frac{5}{(e^2 n)^{c+1}}}. \end{aligned}$$

The inequality between the second and the third lines is justified by inequalities $\left(1 - (en)^{-(l+1)(c+1)}\right)^{(en)^{(l+1)(c+1)}} \geq (1 - e^{-2})^{e^2} \geq 1/3$.

Therefore, we get that the probability not to restart at the second phase is $p_2 \geq (1 - \frac{1}{e^c n^{c+1}})\left(\frac{1}{3}\right)^{\frac{5}{(e^2 n)^{c+1}}}$.

In the end of the second phase, there is a probability that the algorithm will not move to the third state. We found this probability in Section III-C. It is $p_{2\to 3} = 1 - \frac{(n-l-1)^2}{n^2}$.

The last chance to restart the algorithm is on the third phase. We also can find the probability of this, using the multiplicative drift theorem as for the first phase. Consider the potential function $\Psi(x) = n - i$, where $i = \text{ONEMAX}(x)$. The expected difference $E(\Psi_t - \Psi_{t+1} | \Psi_t) = \frac{n-i}{n} = \frac{\Psi_t}{n}$. $\Psi_0 = l + 1$. $\Psi_{\min} = 1$. So using the drift theorem we can say that $p_3 = \Pr[T_3 \geq (c+1)n(\ln n + 1)] \leq \Pr[T_3 \geq (c+1)n(\ln l + 1)] \leq e^{-c}$.

Summing up this section, the only way not to restart the algorithm requires all of the following independent events: 1) no restart on the first phase ($p_1 \geq 1 - e^{-\frac{2}{3}c+\frac{1}{3}}$); 2) choosing RIGHTBRIDGE in the end of the first phase ($p_{1\to 2} = \frac{1}{2}$); 3) no restart on the second phase ($p_2 \geq (1 - \frac{1}{e^c n^{c+1}})\left(\frac{1}{3}\right)^{\frac{5}{(e^2 n)^{c+1}}}$); 4) ending the second phase ($p_{2\to 3} \geq 1 - \frac{(n-l-1)^2}{n^2}$); 5) no restart on the third phase ($p_3 \geq 1 - e^{-c}$).

So the probability not to restart will be

$$\begin{aligned} p_{\text{end}} \geq (1 - e^{-\frac{2}{3}c+\frac{1}{3}})\frac{1}{2}\left(1 - \frac{1}{e^c n^{c+1}}\right)\left(\frac{1}{3}\right)^{\frac{5}{(e^2 n)^{c+1}}} \\ \times \left(1 - \frac{(n-l-1)^2}{n^2}\right)(1 - e^{-c}). \end{aligned} \tag{6}$$

$(1 - e^{-\frac{2}{3}c+\frac{1}{3}})\frac{1}{2}(1 - e^{-c})$ is some constant C_1, if c is a constant, and $\left(1 - \frac{1}{e^c n^{c+1}}\right)\left(\frac{1}{3}\right)^{\frac{5}{(e^2 n)^{c+1}}}$ is $(1 - o(1))$, which proves the lemma. ■

B. Expected Runtime of EA+RL Modification

To start with, we should say that the expected number of runs (including the successful one) is the inverse p_{end} from Eq. 6: $N = 1/p_{\text{end}} \leq 2n^2/(C_0(n^2 - (n-l-1)^2))$.

Recall that the runtime of the algorithm when it finds the optimum is $T \leq \frac{5}{2}n(\ln l + 1) + 5n \ln \frac{n-l}{l+1} + n$. The following lemma bounds the runtime of a run that ends with a restart.

Lemma 5. *If a run ends with a restart, then the expected runtime of this run* $T_r \le \frac{3}{2}n(\ln(l+1)+1) + 5n\ln\frac{n-l-2}{l+2} + 4n + 19 + (c+1)n(\ln n + 1)$.

Proof: The longest scenario of the restart is the following. The algorithm goes through the first and the second phases. While passing from the first phase to the second phase, the agent selects JUMP. Then the algorithm gets to the state $s = n-l-1$, falls to the state $s=0$ and returns from it twice. Finally, the algorithm gets stuck in the state $s = n-l-1$ and spends there m iterations before the restart.

The expected runtime of this case is the sum of the following expected values:

- Total runtime of the first and the second phases: $T_1+T_2 \le \frac{3}{2}n(\ln(l+1)+1) + 5n\ln\frac{n-l-2}{l+2} + 2n + 15$.
- Two falls from the state $s = n-l-1$ to the state $s=0$, $T_{\text{before fall}} \le 4$.
- Two returns from the state $s=0$ to the state $s = n-l-1$: $T_{\text{fall}} \le 2n$. The proof of this fact uses the multiplicative drift in the same way as in Theorem 2.
- The number of iterations spent in the same state before the restart occurs: $T_{\text{stuck}} = m = (c+1)n(\ln n + 1)$.

The only thing left to prove the lemma is to sum up these bounds. ∎

Lemma 4 and Lemma 5 give us the proof of Theorem 5:

Runtime of EA+RL *modification:* T and T_r are both $O(n\log n)$. So the runtime of the algorithm modification is

$$\begin{aligned}
T_{\text{mod}} &= (N-1)T_r + T \\
&\le \left(\frac{n^2}{C_0(n^2-(n-l-1)^2)} - 1\right)\left(\frac{3}{2}n(\ln(l+1)+1)\right. \\
&\left. + 5n\ln\frac{n-l-1}{l+1} + 4n + 19 + (c+1)n(\ln n+1)\right) \\
&+ \left(\frac{5}{2}n(\ln(l+1)+1) + 5n\ln\frac{n-l-1}{l+1} + 2n + 15\right) \\
&= \left(\frac{n^2}{C_0(n^2-(n-l-1)^2)} - 1\right)O(n\log n) + O(n\log n) \\
&= O\left(\frac{n^2}{(l+1)(2n-l-1)}n\log n\right) = O\left(\frac{n^2\log n}{l}\right)
\end{aligned}$$

∎

This result depends on the value of l. When this value is $\Omega(n)$, the algorithm has a runtime of $O(n\log n)$, that means that the "extreme JUMP" is the easiest case for the modified algorithm. At the same time, the case when $l = \Theta(1)$ leads to the more than quadratic runtime while being the easiest case for the most simple evolutionary algorithms.

Preliminary experiments showed that for the different values of l the value of c which provides the lowest value of T_{mod} lies in $[3.2; 3.85]$ for different values of l. As it was mentioned before, the highest expected number of the restarts is achieved when $l=1$. In this case, the best value is $c = 3.85$, so it has been selected for the final version of the algorithm.

V. COMPARISON WITH EMPIRICAL RESULTS

We performed a set of experiments to show the precision of our theoretical results. We run the modified algorithm for

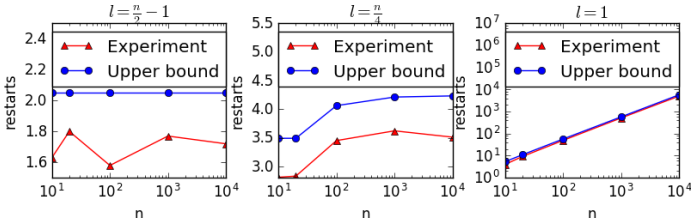

Fig. 2. Upper bound on the expected number of restarts and number of restarts averaged over 1000 runs

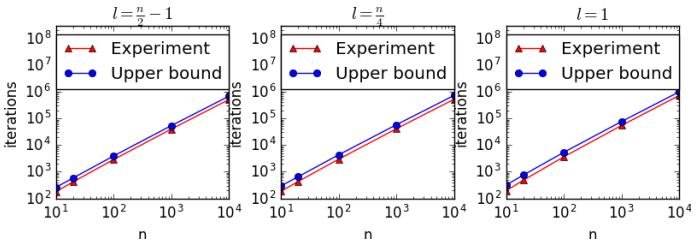

Fig. 3. Upper bound on the expected number of iterations and number of iterations averaged over 1000 runs for the runs ended with a restart

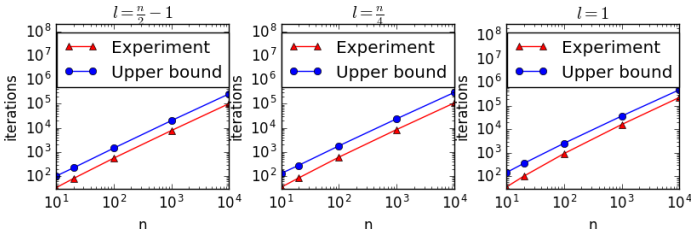

Fig. 4. Upper bound on the expected number of iterations and number of iterations averaged over 1000 runs for the runs ended with finding the optimum

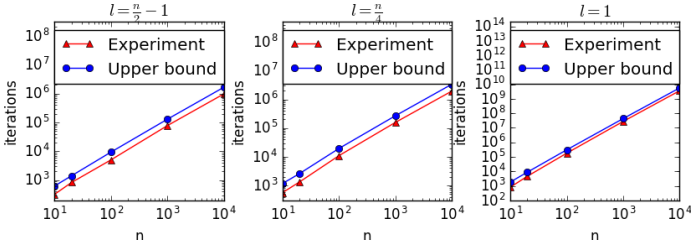

Fig. 5. Upper bound on the expected total number of iterations and number of iterations averaged over 1000 runs of the modified algorithm

$n = 10, 20, 100, 1000, 10000$ and for $l = \frac{n}{2}-1, \frac{n}{4}, 1$ for every value of n. For every pair of n and l we performed 1000 runs of the algorithm.

Table I and the plots in Fig. 2–5 illustrate the results of the experiments. The plots demonstrate that all received bounds are always above the real values and are really close to them for all values of n.

The results also show that the received bound on the expected number of iterations for the unsuccessful runs does not exceed the 1.5 factor of the experimentally received value. As for the successful runs, the upper bound is about 2.5 times greater than the real value.

The one more interesting fact is that we have never reproduced restart during the first or the third phase. The only two reasons of restarts were the two falls back from the state

TABLE I
PERCENT OF RESTARTS CAUSED BY SELECTING JUMP IN THE END OF THE FIRST PHASE AMONG ALL THE RESTARTS

n	10			20			100		
l	4	2	1	9	5	1	49	25	1
%	79.4	67.1	58.5	79.7	67.9	55.2	80.1	65.4	50.9

n	1000			10000		
l	499	250	1	4999	2500	1
%	80.3	63.7	50.0	81.6	64.3	50.0

$s = n - l - 1$ and the choice of the JUMP function in the end of the first phase. This means that the probability to make an unneeded restart is small enough, and the algorithm actually restarts only after it has been stuck in the state $s = n-l-1$. In Table I we show the percentage of the latter reason of restart among all the restarts.

VI. CONCLUSION AND FUTURE WORK

In this work we discovered that EA+RL can not efficiently manage with local optima of the JUMP function because of getting stuck in more than a half of runs. However, we proposed an EA+RL modification with restarts that solved the problem in a polynomial time. This modification should be improved in the future work, as in this paper our choice of the number of iterations before algorithm restarts is based on the theoretical analysis of the problem that is hard to be performed on most real-world problems.

We also analysed whether EA+RL method relearns which objective is helpful in the case when helpfulness of auxiliary objectives changes during optimization. We intuitively expected that the method relearns just at that optimization stage when an objective becomes helpful. But actually, in the considered problem relearning occurs because EA+RL occasionally learns to select the proper auxiliary objective before it becomes helpful. This means that EA+RL may mistakenly learn to select objectives that will be harmful in the later phases of the optimization. However, the restart mechanism helps the method to detect such "false learning" and begin learning from scratch, which guarantees finite expected runtime.

To the best of our knowledge, it was the first time when the analysis of selecting auxiliary objectives with EA+RL for the non-monotonic function was considered and an efficient method for the JUMP problem with auxiliary objectives was proposed. This method may be useful for optimization of other non-monotonic functions with auxiliary objectives.

To sum up, the future work is to expand the algorithm on some more practical problems by finding the way of selecting a restart strategy. Also we are interested in other strategies of escaping local optima like penalizing objectives that do not change the target fitness value for a long time. Another direction of future work is further investigation of learning and finding out if EA+RL can relearn multiple times.

VII. ACKNOWLEDGMENTS

This work was financially supported by the Government of Russian Federation, Grant 074-U01, and by RFBR according to the research project No. 16-31-00380 mol_a.

REFERENCES

[1] Antipov, D., Buzdalov, M., Doerr, B.: Runtime Analysis of (1+1) Evolutionary Algorithm Controlled with Q-learning using Greedy Exploration Strategy on OneMax+ZeroMax Problem. In: Evolutionary Computation in Combinatorial Optimization, pp. 160–172. No. 9026 in Lecture Notes in Computer Science (2015)

[2] Brockhoff, D., Friedrich, T., Hebbinghaus, N., Klein, C., Neumann, F., Zitzler, E.: On the effects of adding objectives to plateau functions. IEEE Transactions on Evolutionary Computation 13(3), 591–603 (2009)

[3] Buzdalov, M., Buzdalova, A.: Adaptive selection of helper-objectives for test case generation. In: 2013 IEEE Congress on Evolutionary Computation. vol. 1, pp. 2245–2250 (2013)

[4] Buzdalov, M., Buzdalova, A.: OneMax helps optimizing XdivK: Theoretical runtime analysis for RLS and EA+RL. In: Proceedings of Genetic and Evolutionary Computation Conference Companion. pp. 201–202. ACM (2014)

[5] Buzdalov, M., Buzdalova, A., Petrova, I.: Generation of tests for programming challenge tasks using multi-objective optimization. In: Proceedings of Genetic and Evolutionary Computation Conference Companion. pp. 1655–1658. ACM (2013)

[6] Buzdalov, M., Doerr, B., Kever, M.: The unrestricted black-box complexity of jump functions. Evolutionary Computation 24(4), 719–744 (2016)

[7] Buzdalova, A., Buzdalov, M.: Increasing efficiency of evolutionary algorithms by choosing between auxiliary fitness functions with reinforcement learning. In: Proceedings of the International Conference on Machine Learning and Applications. vol. 1, pp. 150–155 (2012)

[8] Doerr, B., Doerr, C., Kötzing, T.: Unbiased black-box complexities of jump functions. In: Proceedings of Genetic and Evolutionary Computation Conference. pp. 769–776 (2014)

[9] Doerr, B., Johannsen, D., Winzen, C.: Multiplicative drift analysis. Algorithmica 64(4), 673–697 (2012)

[10] Droste, S., Jansen, T., Wegener, I.: On the analysis of the (1+1) evolutionary algorithm. Theor. Comput. Sci. 276(1-2), 51–81 (2002)

[11] Handl, J., Lovell, S.C., Knowles, J.D.: Multiobjectivization by decomposition of scalar cost functions. In: Parallel Problem Solving from Nature – PPSN X, pp. 31–40. No. 5199 in Lecture Notes in Computer Science, Springer (2008)

[12] Jensen, M.T.: Helper-objectives: Using multi-objective evolutionary algorithms for single-objective optimisation: Evolutionary computation combinatorial optimization. Journal of Mathematical Modelling and Algorithms 3(4), 323–347 (2004)

[13] Knowles, J.D., Watson, R.A., Corne, D.: Reducing local optima in single-objective problems by multi-objectivization. In: Proceedings of the First International Conference on Evolutionary Multi-Criterion Optimization. pp. 269–283. Springer-Verlag (2001)

[14] Lehre, P.K., Witt, C.: Black-box search by unbiased variation. In: Proceedings of Genetic and Evolutionary Computation Conference. pp. 1441–1448. ACM (2010)

[15] Lochtefeld, D.F., Ciarallo, F.W.: Deterministic helper-objective sequences applied to Job-Shop scheduling. In: Proceedings of Genetic and Evolutionary Computation Conference. pp. 431–438. ACM (2010)

[16] Neumann, F., Wegener, I.: Minimum spanning trees made easier via multi-objective optimization. Natural Computing 5(3), 305–319 (2006)

[17] Neumann, F., Wegener, I.: Can single-objective optimization profit from multiobjective optimization? In: Multiobjective Problem Solving from Nature, pp. 115–130. Natural Computing Series, Springer Berlin Heidelberg (2008)

[18] Petrova, I., Buzdalova, A., Buzdalov, M.: Improved selection of auxiliary objectives using reinforcement learning in non-stationary environment. In: Proceedings of the International Conference on Machine Learning and Applications. pp. 580–583 (2014)

[19] Segura, C., Coello, C.A.C., Miranda, G., Léon, C.: Using multi-objective evolutionary algorithms for single-objective optimization. 4OR 3(11), 201–228 (2013)

[20] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, USA (1998)

[21] Witt, C.: Optimizing linear functions with randomized search heuristics – the robustness of mutation. In: Proceedings of the 29th Annual Symposium on Theoretical Aspects of Computer Science. pp. 420–431 (2012)