A Neural Network Based Guidance for a BRKGA: An Application to the Longest Common Square Subsequence Problem

Jaume Reixach[1(✉)], Christian Blum[1], Marko Djukanović[2], and Günther R. Raidl[3]

[1] Artificial Intelligence Research Institute (IIIA-CSIC), Campus of the UAB, 08193 Bellaterra, Spain
{jaume.reixach,christian.blum}@iiia.csic.es
[2] Faculty of Natural Sciences and Mathematics, University of Banja Luka, Mladena Stojanovića 2, 78000 Banja Luka, Bosnia and Herzegovina
marko.djukanovic@pmf.unibl.org
[3] Institute of Logic and Computation, TU Wien, Vienna, Austria
raidl@ac.tuwien.ac.at

Abstract. In this work we apply machine learning to better guide a biased random key genetic algorithm (BRKGA) for the longest common square subsequence (LCSqS) problem. The problem is a variant of the well-known longest common subsequence (LCS) problem in which valid solutions are square strings. A string is square if it can be expressed as the concatenation of a string with itself. The original BRKGA is based on a reduction of the LCSqS problem to the LCS problem by cutting each input string into two parts. Our work consists in enhancing the search process of BRKGA for good cut points by using a machine learning approach, which is trained to produce promising cut points for the input strings of a problem instance. In this study, we show the benefits of this approach by comparing the enhanced BRKGA with the original BRKGA, using two benchmark sets from the literature. We show that the results of the enhanced BRKGA significantly improve over the original results, especially when tackling instances with non-uniformly generated input strings.

Keywords: Genetic algorithms · Neural networks · Longest Common Subsequences · Beam search

Jaume Reixach and Christian Blum are supported by grants TED2021-129319B-I00 and PID2022-136787NB-I00 funded by MCIN/AEI/10.13039/501100011033. Günter R. Raidl is supported by the Vienna Graduate School on Computational Optimization (VGSCO), Austrian Science Foundation, project no. W1260-N35. Marko Djukanović is supported by the project entitled "Development of artificial intelligence models and algorithms for solving difficult combinatorial optimization problems" funded by the Ministry of Scientific and Technological Development and the Higher Education of the Republic of Srpska.

T. Stützle and M. Wagner (Eds.): EvoCOP 2024, LNCS 14632, pp. 1–15, 2024.
https://doi.org/10.1007/978-3-031-57712-3_1

1 Introduction

Recently, the use of machine learning (ML) for guiding metaheuristic algorithms has become popular in order to enhance performance [2]. In this work, we show how this idea can be beneficially applied within genetic algorithms (GAs). More specifically, information obtained through a ML approach is used to guide a biased random key genetic algorithm (BRKGA). In the context of combinatorial optimization, BRKGA work on a population of individuals indirectly representing solutions to the problem at hand by means of vectors of real values in $(0, 1)$. An important aspect of a BRKGA is the so-called decoder which maps every individual to a solution to the problem at hand. The methodology proposed in this work consists of biasing individuals towards vectors *learn*ed by a feed-forward neural network [1] before applying the decoder.

Our approach is evaluated using the Longest Common Square Subsequence (LCSqS) problem, a variation of the well-known Longest Common Subsequence (LCS) problem. Both problems are formally introduced in the following.

1.1 The LCSqS Problem

A string is considered a finite sequence of characters drawn from a finite set Σ, referred to as the alphabet. Given a string s, a subsequence of s is a string that can be derived from s by selectively removing zero or more characters while preserving the original order of the remaining ones. The longest common subsequence (LCS) problem entails, given a set of input strings $S = \{s_1, s_2, \ldots, s_m\}$ $(m \geq 2)$, the task of identifying the longest possible string that is as a subsequence of all the strings in S. This problem, known to be NP-hard when the number of input strings is not fixed [13], finds vital applications across diverse domains including bioinformatics, file plagiarism detection, and time series analysis [11,14,15,18].

In our present study, we concentrate on a specific variation of this problem known as the longest common square subsequence (LCSqS) problem as introduced by Inoue et al. [9]. A string s is classified as a square string if it can be expressed by concatenating a string s' with itself. The aim of the LCSqS problem is to find a longest common subsequence within the set S of input strings that is also a square string. Similarly to the LCS problem, the LCSqS problem possesses applications in bioinformatics, in particular facilitating the identification of internal structural similarities within molecular data [8].

Reduction to the LCS Problem. Before we explain the reduction, let us introduce additional notation. For a string s, we denote its length by $|s|$. For two integers $i, j \leq |s|$, $s[i, j]$ refers to a (continuous) part of string s that starts from the character at position i and ends with the character at position j; when $i = j$, the single-character string $s[i]$ is given, or when $i > j$, the empty string ε. Note that the starting character of each string holds position one.

A notable characteristic of the LCSqS problem, effectively leveraged by existing heuristic algorithms, is its reducibility to the LCS problem. Given a set of

input strings $S = \{s_1, s_2, \ldots, s_m\}$, let $\mathcal{P} = \{(p_1, p_2, \ldots, p_m) \in \prod_{i=1}^{m} \{1, \ldots, |s_i| - 1\}\}$ denote the set of all possibilities for partitioning each string of S into two parts. The LCSqS problem with input strings S can then be solved as follows. First, for every $p \in \mathcal{P}$ a solution s_p to the LCS problem with input strings $S_p = \{s_1[1, p_1], s_1[p_1 + 1, |s_1|], s_2[1, p_2], s_2[p_2 + 1, |s_2|], \ldots, s_m[1, p_m], s_m[p_m + 1, |s_m|]\}$ is computed. It is rather easy to see that the concatenation of $s_p^* = \arg\max_{p \in \mathcal{P}} s_p$ with itself, that is $s_p^* \cdot s_p^*$, gives an optimal solution to the original LCSqS problem [16]. Thus, the LCSqS problem with input strings S can be solved by finding the cut point vector $p \in \mathcal{P}$ that leads to the longest LCS solution for input strings S_p and concatenating this LCS solution with itself.

1.2 Literature Review

Numerous algorithms have been devised for addressing the LCS problem, owing to its many important practical applications. Exact solutions can be obtained through dynamic programming approaches, but their computational complexity is in $O(n^m)$, where m refers to the number of input strings, and n is the length of the longest input string. As this number of strings m grows, the practicality of these dynamic programming methods diminishes, leading to the adoption of heuristic and metaheuristic approaches. Among these, one of the most successful is beam search (BS), which was initially introduced in the context of the LCS problem by Blum et al. in [3].

When it comes to the LCSqS problem, Inoue et al. introduced exact dynamic programming algorithms in [9], which necessitate $O(n^6)$ time for the scenario involving two input strings. Conversely, Djukanović et al. [8] proposed two heuristic algorithms, with the most effective one being a hybrid approach combining reduced variable neighborhood search (RVNS) with BS. The so far leading heuristic algorithm is a Brkga searching through the space of cut points and using BS for solving the corresponding LCS problems [16]. The used BS approach is the one outlined by Djukanovíc et al. [7], which offers two distinct designs for its guiding heuristic function. In our work, we enhance this Brkga through the integration of a ML technique, and we refer to this enhanced version as Brkga-Learn. A brief overview of Brkga is provided in the following sections, while a more comprehensive explanation of this algorithm can be found in the referenced articles.

1.3 Our Contribution

As already mentioned, the original Brkga for the LCSqS problem from [16] searches the space of possible cut points heuristically, because a complete exploration of such a vast space is intractable in general. Each possible individual is mapped to a valid solution (consisting of a cut point for each input string) and then evaluated by applying the BS approach from [7] to the resulting LCS problem. Due to the high complexity of solving the LCS problem, this step of solving each LCS problem instance is done in a heuristic manner by BS. The

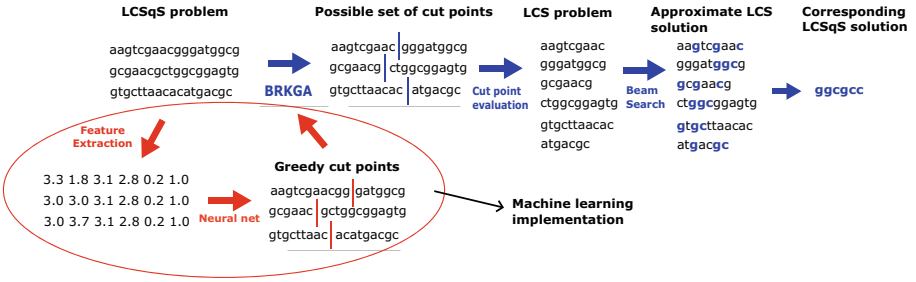

Fig. 1. General structure of the proposed BRKGA-LEARN approach

mapping process is performed within a decoder that also makes use of greedy information consisting of cut points deemed promising for every input string.

We introduce a novel approach to enhance the BRKGA for the LCSqS problem from [16] by replacing its original greedy mechanism with *learned* information provided by a ML model. Specifically, we advocate the utilization of a feed-forward *neural network* (NN). To determine a cut point for a given string s from the set S of all input strings, this NN takes as input features from s in combination with overarching features drawn from the entire set S. The overall structure of our proposed approach is illustrated in Fig. 1. First, given a set of input strings for the LCSqS problem, individual features for each input string and global features of S are extracted. Afterward, these are fed into the (previously trained) neural net, obtaining a cut point for each input string. Finally, these cut points are used by the decoder of the BRKGA, which uses them in order to replace greedy information.

The rest of this work is organized as follows. In Sect. 2, the original BRKGA is explained with a focus on its cut point search. Next, Sect. 3 develops the proposed learning mechanism, explaining the selected features and the training approach used for the NN. Section 4 presents a comprehensive experimental evaluation including a comparison of the proposed BRKGA-LEARN with the original BRKGA. Finally, in Sect. 5 some conclusions are derived and an outlook on future work is given.

2 The Original BRKGA

The top part of Fig. 1 illustrates the main idea of the BRKGA for the LCSqS problem. In the following, we will summarize the working mechanism of this algorithm. For an explanation of the BS we refer to the original publication [7].

In the context of any combinatorial optimization problem, a BRKGA works on a population in which each individual $\mathbf{v}$, represented by a vector of real values from $(0, 1)$, can be mapped to a valid solution to the problem at hand. In the case of the application to the LCSqS problem, an individual $\mathbf{v}$ is mapped to a cut point vector for the set of input strings S, that is, an element of the set of all possible cut point vectors $\mathcal{P}$ as defined in Sect. 1.1. The goal is to find an

Algorithm 1. The decoder of the BRKGA for the LCSqS problem

Input: Input strings $S = \{s_1, \ldots, s_m\}$, an individual $\mathbf{v}$, beam width β, and BS guidance function h

1: $\mathbf{v}' \leftarrow$ greedy_transformation($\mathbf{v}$)
2: $\mathbf{p}^{\mathbf{v}} \leftarrow$ map_to_cut_points($\mathbf{v}'$)
3: $S_{\mathbf{p}^{\mathbf{v}}} \leftarrow$ LCS problem instance induced by $\mathbf{p}^{\mathbf{v}}$
4: $t^{\mathbf{v}} \leftarrow$ beam_search_for_LCS_problem($S_{\mathbf{p}^{\mathbf{v}}}, \beta, h$)
5: **return** $t^{\mathbf{v}} \cdot t^{\mathbf{v}}$

individual $\mathbf{v}$ that maps to a cut point vector $\mathbf{p}^{\mathbf{v}} \in P$ that maximizes the length of the solution to the LCS problem with input strings $S_{\mathbf{p}^{\mathbf{v}}}$. The decoder that takes care of the mapping process is a critical part of the algorithm and will be explained below.

A BRKGA works as follows. First, a population of p_{size} individuals is initialized at random, by setting every individual to a random vector of values from $(0, 1)$. Afterward, the main loop is entered, in which the fitness of each individual is determined by applying the decoder and further evaluating the obtained solution. The population is then split into the following two parts:

1. The *elite* population $P_{\text{e}} \subset P$ that consists of the best p_{e} individuals of P.
2. The *non-elite* population, consisting of the remaining $p_{\text{size}} - p_{\text{e}}$ individuals of the current population P.

Here, $p_{\text{e}} < p_{\text{size}} - p_{\text{e}}$ is a algorithm parameter, called the number of elites. Another algorithm parameter, $p_{\text{m}} < p_{\text{size}} - p_{\text{e}}$, called the number of mutants, is then used to generate the next population of individuals in the following way. The elite population is passed to the next generation along with p_m mutant individuals, which are constructed randomly as in the case of the initial population. The remaining $p_{\text{size}} - p_{\text{e}} - p_{\text{m}}$ individuals are introduced through the process of mating. This consists of selecting two parents at random from the current population, one elite and one non-elite, and constructing a new individual by setting its i-th vector position to one of the parents' i-th vector positions, choosing for each position between the two parents depending on a parameter $\rho_e \in (0.5, 1]$, called the elite inheritance probability, which determines the probability of choosing the i-th vector position of the elite parent.

2.1 The Decoder

The decoder, which is shown in Algorithm 1, first applies a greedy transformation to individual $\mathbf{v}$ (line 1). Hereby, some greedy information is given by a vector $\mathbf{u}$ of the same dimension as $\mathbf{v}$ and the following expression is applied:

$$v'_i := v_i + \gamma \cdot (u_i - v_i) \tag{1}$$

Algorithm parameter $\gamma \in [0, 1]$ is the so-called *greedy rate* and controls the extent to which v_i is moved towards u_i, for all $i = 1, \ldots, m$. In case $\gamma = 1$, v_i

is simply replaced by u_i. In the other extreme, if $\gamma = 0$ the greedy information is not used at all. In [16], three different designs were proposed for u (see the subsequent section). Next, vector $\mathbf{v'} = (v'_1, v'_2, \ldots, v'_m) \in (0,1)^m$ is mapped to the cut point vector $\mathbf{p^v}$, where $p_i^v = \lfloor v'_i \cdot |s_i| \rceil$ for all $i = 1, \ldots, m$; see line 2 of Algorithm 1. Notation $\lfloor r \rceil$ refers to rounding value r to the closest integer. If p_i^v is 0 or $|s_i|$, the cut is set to 1 and $|s_i| - 1$ respectively, in order to only consider feasible cuts. Finally, BS is applied to the $2m$ input strings obtained after cutting the strings of S at the positions given by $\mathbf{p^v}$. The resulting LCS solution is concatenated with itself, producing a solution to the original LCSqS problem. After this is done, a measure of fitness is given to the individual, depending on its associated solution quality. The fitness measure used comprises two distinct values. The first value represents the length of the solution, while the second value is designed as a tie breaker to differentiate between individuals generating solutions of equal length. For a comprehensive description of this secondary fitness measure, which is elaborated upon with three different designs, we refer to the original article [16].

2.2 Different Designs for the Greedy Vector u

1. The first option consists in simply using $\mathbf{u} = (0.5, \ldots, 0.5)$. This is motivated by the fact that, in general, the middle point of every input string is potentially a good place for cutting the string into two pieces, as these cuts maximize the resulting strings minimum length obtained after cutting.
2. The second approach determines u_i for all $i = 1, \ldots, m$ by

$$u_i = \arg\min_{r \in [0,1]} \sum_{a \in \Sigma} \left| \left| s_i[1, \lfloor r \cdot |s_i| \rceil] \right|_a - \left| s_i[\lfloor r \cdot |s_i| \rceil + 1, |s_i|] \right|_a \right| \tag{2}$$

   Here, $\left| s_i[1, \lfloor r \cdot |s_i| \rceil] \right|_a$ and $\left| s_i[\lfloor r \cdot |s_i| \rceil + 1, |s_i|] \right|_a$ denote the numbers of occurrences of character a in the two strings obtained after cutting s_i after position $\lfloor r \cdot |s_i| \rceil$. This approach looks for the cut that maximizes the overall equilibrium of the quantity of each character at both sides of the cut. The value that minimizes the previous expression may not be unique. In this case a random value among the candidate values is chosen. With this design, the greedy value exploits some information about the distribution of characters within the strings to decide which cut appears most promising.
3. The last considered design determines u_i for all $i = 1, \ldots, m$ by

$$u_i = \arg\max_{r \in [0,1]} \left| \mathrm{LCS}\left(s_i[1, \lfloor r \cdot |s_i| \rceil], \; s_i[\lfloor r \cdot |s_i| \rceil + 1, |s_i|] \right) \right| \tag{3}$$

   Hereby, the LCS of the pair of substrings is determined with the dynamic programming approach from [17]. The motivation for this design is that the ultimate goal for deciding the cuts is maximizing the LCS length between the resulting 2 m strings. As it is intractable to maximize this value for all strings together, we choose each cut to be the one that maximizes the LCS

length between the resulting two parts of each string. Just as with the second design, the value that maximizes the latter expression may not be unique. In this case, the tie is broken randomly.

3 Machine Learning Based Guidance for BRKGA

Our approach presented in this paper, henceforth called BRKGA-LEARN, differs from the original BRKGA in the design of the greedy information vector u. More specifically, u_i is now determined by a feed-forward NN that receives features of the input string s_i together with some global features.

3.1 Features

Six features are used as input to the NN for each input string. The first two are specific to each string, while the last four are global ones. With this, we intend to capture information both about individual strings and the whole problem instance. Given a problem instance consisting of input strings $S = \{s_1, s_2, \ldots, s_m\}$, we denote by gv2_i and gv3_i the values for the second and third greedy value designs as outlined in the previous sub-section for the i-th string respectively.

The following features are extracted for every string s_i, $i = 1, 2, \ldots, m$:

$$X = \big(\text{gv2}_i,\ \text{gv3}_i,\ \overline{\text{gv2}},\ \overline{\text{gv3}},\ \sigma(\text{gv2}),\ \sigma(\text{gv3})\big)$$

Hereby, $\overline{\text{gv2}}$ and $\overline{\text{gv3}}$ denote the averages of the second and third greedy values across all strings in S and $\sigma(\text{gv2})$ and $\sigma(\text{gv3})$ are the corresponding sample standard deviations, respectively:

$$\overline{\text{gv2}} = \frac{\sum_{i=1}^{m} \text{gv2}_i}{m},\ \sigma(\text{gv2}) = \sqrt{\frac{\sum_{i=1}^{m} \big(\text{gv2}_i - \overline{\text{gv2}}\big)^2}{m-1}} \tag{4}$$

$$\overline{\text{gv3}} = \frac{\sum_{i=1}^{m} \text{gv3}_i}{m},\ \sigma(\text{gv3}) = \sqrt{\frac{\sum_{i=1}^{m} \big(\text{gv3}_i - \overline{\text{gv3}}\big)^2}{m-1}} \tag{5}$$

Feature values are standardized before being fed into the NN in order to have a mean of zero and a standard deviation of one. The NN comprises a single node in the output layer with a sigmoid activation function as the NN is applied to a single string at a time, representing a promising cut point in the form of a value in $(0, 1)$. Moreover, tuning indicated that one dense hidden layer consisting of just five nodes with a ReLU activation function is sufficient, as more complex networks did not yield significantly better results. Figure 2 provides a graphical representation of the used NN architecture.

The expressions for the ReLU and the sigmoid function are given below.

$$\text{ReLU}(x) = \max\{0, x\},\ x \in \mathbb{R} \tag{6}$$

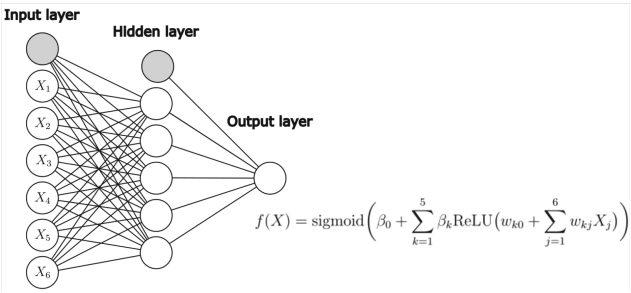

Fig. 2. A graphical representation of the employed feed-forward NN. $f(X)$ is the output of the neural network for the feature value vector $X = (X_1, X_2, \ldots, X_6)$. $\{w_{k,j}\}$ and $\{\beta_k\}$ are the 41 parameters of the neural net. Particularly, $\{w_{k,j}\}$ transform the features into the hidden-layer and $\{\beta_k\}$ transform the hidden-layer into the output. Morever, the lines from the top node in each layer represent the biases, which consist of parameters $\{w_{k0}\}$ and β_0.

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}, \ x \in \mathbb{R} \tag{7}$$

3.2 Neural Net Training

To train the NN we have considered the following two options:

1. The first option consists in generating toy LCSqS problem instances, finding the optimal cuts for their input strings by complete enumeration, and then using these to train the NN. With this method each string and optimal cut pair is used as a training example for the NN, which is then trained in a classical supervised fashion using the Adam [10] stochastic gradient descent-based optimizer. A critical question clearly is here whether or not such a NN will generalize to out-of-distribution inputs for larger problem instances.
2. Training the NN on full-size instances using a Genetic Algorithm (GA). Hereby, each individual represents a complete set of weights for the NN, and the population is evolved until overfitting is detected. This has been shown to be a valid alternative for training neural networks [5,6].

Overall, the first approach produced unsatisfactory results. This may be attributed to the limitation that finding optimal cuts through complete enumeration is only feasible for problem instances of modest size. In our study, we employed instances, each comprising five strings of length 50. It is likely that the model's inability to generalize effectively to larger instances was due to the small size of the training data. As a result, we decided to pursue the second approach, which does not depend on knowledge of the optimal cuts for the training instances.

To delve into further detail, the genetic algorithm (GA) utilized for the NN training is a random key genetic algorithm (RKGA), which operates very similarly

to the general process employed in a BRKGA. The primary distinction lies in the mating procedure. In the case of a RKGA, parental candidates are chosen randomly, whereas in a BRKGA, one parent is selected from the elites, and the other from the non-elite group. Additionally, in an RKGA, the offspring is derived by randomly selecting positions from the parents' vectors. The RKGA was applied using a population of 20 individuals, one elite individual, and seven mutants. Remember, in this context, that each individual is a vector of 41 real values that represent the NN weights.

Given a set of training LCSqS instances, the evaluation of an individual works as follows. First, a value u_i is obtained by supplying the NN (equipped with the weights of the individual) for every string s_i of every problem instance in the training set. Afterward, the decoder is applied in order to obtain the corresponding cut points. Lastly, BS is used to evaluate the obtained cut points. The average of the obtained solution lengths for all training instances is then used as a measure of fitness and associated to the corresponding individual. One may object that it would be better to run the whole BRKGA instead of just BS in order to evaluate an individual. However, this would lead to training times too large for practical purposes.

Once a new best individual is found during the search process of the training GA, a validation value is calculated in order to check for overfitting. This is done on the basis of a set of validation instances. The validation value is calculated by applying BRKGA for 10 min on the validation instances guided by the $\mathbf{u}$ vector obtained by the NN equipped with the corresponding individuals' weights. The average of the resulting LCSqS lengths is used as a validation value. Note that, in the case of validation it is computationally feasible to run BRKGA instead of BS, as new best individuals are found only sporadically.

Finally, when designing the training procedure one has to decide on an allowed domain for values for the NN weights. This is crucial since, during the generation of random individuals, they must be assigned random values within a specific interval. In our case we allowed weights to take real values from $(-1, 1)$. When allowing for larger values we observed that the NN fitted with random weights often produced output values close to either 0 or 1, which is undesirable as cut points too close to the start or end of strings usually lead to poor results.

In the BS for the NN training, we always applied a beam width of $\beta = 250$ and the UB_1 guidance function from [3]. For BRKGA, we used the parameter values determined for the experimental evaluation in the original BRKGA article, which were obtained by tuning depending on the benchmark set and string length.

4 Experimental Evaluation

In this section we experimentally evaluate BRKGA-LEARN and compare its performance to the one of the original BRKGA, the so far leading approach for solving the LCSqS problem. To do so, we train and run BRKGA-LEARN on two benchmark sets that were already used for the evaluation of the original BRKGA in [16]. The first benchmark set, named RANDOM, consists of instances with

strings generated uniformly at random. In contrast, the second set named Non-Random consists of instances with non-uniform strings generated by implanting appropriate patterns. For evaluating the original Brkga a third benchmark set consisting of real-world instances was used, which we do not consider here due to its strong structural similarity to benchmark set Random. For a more detailed explanation of the benchmark sets we refer to [16].

In order to keep the comparison fair, Brkga-Learn was given the same computation time for the application to each problem instance as Brkga. In particular, we used a computation time limit of 600 CPU seconds per instance. This time limit was also the one applied for each algorithm execution during parameter tuning and for the calculation of the validation value during training. Benchmark sets Random and Non-Random consist of 150 and 100 instances, respectively, for each $n \in \{100, 500, 1000\}$, where n denotes the length of the strings in the instances. Moreover, Non-Random instances can be split into two sets depending on a parameter *type*, which indicates the way in which the patterns were implanted.

Brkga-Learn was trained depending on n for instance set Random and depending on n and *type* for instance set Non-Random. Therefore, three separate training procedures were conducted for benchmark set Random, and six ones for benchmark set Non-Random. On the other hand, we performed the parameter tuning depending on n for both instance sets, following the same procedure as for the original Brkga. Moreover, we also used different equivalently generated instances for training, parameter tuning, and evaluation. Each Random training used fifteen instances for calculating training values and fifteen more for calculating validation values. One for every combination of m (number of strings) and $|\Sigma|$ (alphabet size). Similarly, Non-Random trainings used two sets of ten instances, with two instances for every possible value of m.

Brkga-Learn was implemented in C++ and training was executed in parallel using the OpenMP API [4], with the goal of speeding up the training process. Each training and evaluation run was executed on a cluster of machines with 10-core Intel Xeon processors at 2.2Ghz with 8Gb of RAM. The parallelism in the training runs was implemented in the calculation of the validation and training values. Each training uses 10 cores by distributing the training and validation instances within these. On the other hand, no parallelism was used for parameter tuning and the final experimental evaluation, just like in the case of the original Brkga. Early stopping was used for the training runs, meaning that they were run until the validation value decreased, which indicates a possible overfitting. Training runs lasted about one hour on average, with runs trained using larger instances requiring up to four hours.

Tables 1 and 2 show the results obtained from parameter tuning. Firstly, p_e, p_m, ρ_e, p_{size}, $of2$ are the proportion of elites, the proportion of mutants, the elite inheritance probability, the population size and the secondary objective function design, respectively. Secondly, γ is the greedy rate, which controls the amount of greedy information used, and finally, β and h are the parameters of the beam search, namely the beam width and the guiding function design.

Table 1. Parameter tuning results for benchmark set RANDOM.

	p_e	p_m	ρ_e	p_{size}	$of2$	γ	β	h
$n = 100$	0.15	0.13	0.35	706	1	0.51	113	UB_1
$n = 500$	0.21	0.03	0.45	329	3	0.92	53	UB_1
$n = 1000$	0.31	0.28	0.68	737	2	0.98	587	UB_1

Table 2. Parameter tuning results for benchmark set NON-RANDOM.

	p_e	p_m	ρ_e	p_{size}	$of2$	γ	β	h
$n = 100$	0.18	0.04	0.47	906	2	0.79	388	UB_1
$n = 500$	0.16	0.33	0.58	372	2	0.84	6	UB_1
$n = 1000$	0.23	0.02	0.56	218	1	0.87	6	UB_1

To perform parameter tuning we made use of the automatic configuration tool *irace* [12]. For benchmark set RANDOM, one tuning instance for every combination of m and $|\Sigma|$ was used. Similarly, for benchmark set NON-RANDOM, one instance for every combination of m and *type* was used. This means 15 tuning instances were used for each of the parameter tuning runs concerning the RANDOM benchmark set, while 10 instances were used for each NON-RANDOM one. Every tuning was allowed a budget of 5000 algorithm runs.

4.1 Benchmark Set RANDOM

The results obtained for benchmark set RANDOM are reported in Tables 3, 4 and 5. These contain the results for the instances with $n = 100$, $n = 500$ and $n = 1000$ respectively. For each combination of n, m and $|\Sigma|$ and for each algorithm we present the average length of the best solutions found ($\overline{|s|}$) and the average time required for finding these best solutions ($\bar{t}_{best}[s]$). As each group consists of ten instances, and each algorithm was applied ten times to each instance, the results for each of the 45 table rows average over 100 runs. In each row, the best result is shown in bold.

We can observe that results for this benchmark set differ not by much among the two solution approaches, although BRKGA-LEARN performs more often slightly better. Note that, for this set of instances, the original BRKGA used the first greedy information design which simply consists of biasing cuts towards the middle of every string.

As these instances consist of uniform strings, this approach already produces a good prediction on the optimal cut point, which did not leave our proposed guidance much room for improvement.

Table 3. Comparison of BRKGA-LEARN and BRKGA on RANDOM instances with string length $n = 100$.

m	$\|\Sigma\|$	BRKGA-LEARN		BRKGA	
		$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{s}$	$\bar{t}_{best}[s]$
10	4	**28.44**	98.36	28.34	19.06
10	12	**8.94**	11.73	8.00	0.06
10	20	**4.20**	0.04	4.00	0.00
50	4	19.02	144.95	**19.94**	85.65
50	12	**4.00**	0.03	**4.00**	1.25
50	20	**1.40**	0.39	0.20	0.00
100	4	15.72	168.17	**17.16**	68.45
100	12	**2.42**	30.20	2.20	5.38
100	20	**0.12**	12.91	0.00	0.00
150	4	13.16	160.99	**15.94**	53.37
150	12	**2.00**	0.02	**2.00**	0.20
150	20	**0.00**	0.00	**0.00**	0.00
200	4	12.02	34.86	**14.78**	92.28
200	12	**2.00**	0.29	1.60	1.08
200	20	**0.00**	0.00	**0.00**	0.00

Table 4. Comparison of BRKGA-LEARN and BRKGA on RANDOM Instances With string length $n = 500$.

m	$\|\Sigma\|$	BRKGA-LEARN		BRKGA	
		$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$
10	4	**159.56**	235.68	158.94	226.69
10	12	**59.90**	47.88	59.60	51.22
10	20	**36.12**	11.69	36.04	7.97
50	4	**126.40**	297.95	125.76	146.66
50	12	**40.06**	65.44	40.00	52.29
50	20	**22.00**	25.71	21.82	14.19
100	4	**117.26**	246.59	116.82	102.85
100	12	**34.30**	55.78	34.12	21.53
100	20	**18.00**	0.70	**18.00**	0.60
150	4	**112.50**	125.87	**112.50**	64.91
150	12	**32.00**	2.71	**32.00**	2.14
150	20	**16.02**	0.99	16.00	0.16
200	4	109.96	109.96	**110.00**	26.02
200	12	**30.08**	17.80	30.04	6.96
200	20	15.76	74.79	**15.90**	73.57

Table 5. Comparison of BRKGA-LEARN and BRKGA on RANDOM instances with string length $n = 1000$.

m	$\|\Sigma\|$	BRKGA-LEARN		BRKGA	
		$\overline{\|s\|}$	$\bar{t}_{best}[s]$	$\overline{\|s\|}$	$\bar{t}_{best}[s]$
10	4	**324.42**	155.60	323.98	164.38
10	12	125.44	78.44	**125.78**	110.02
10	20	77.68	46.47	**78.02**	107.96
50	4	**263.90**	88.78	263.74	96.49
50	12	**87.62**	99.69	**87.62**	87.74
50	20	50.16	17.68	**50.54**	43.04
100	4	**249.30**	70.03	248.84	78.81
100	12	78.36	44.55	**78.48**	58.38
100	20	**44.00**	3.68	**44.00**	1.76
150	4	**242.22**	62.90	242.06	73.12
150	12	74.26	40.61	**74.42**	49.93
150	20	41.26	100.94	**41.32**	87.43
200	4	**237.50**	64.25	237.32	69.46
200	12	**72.02**	19.07	72.00	13.19
200	20	39.88	124.41	**39.94**	77.83

4.2 Benchmark Set NON-RANDOM

The results for benchmark set NON-RANDOM are shown in Tables 6, 7 and 8, which follow the same structure as outlined in the last section.

Table 6. Comparison of BRKGA-LEARN and BRKGA on NON-RANDOM instances with string length $n = 100$.

m	*type*	BRKGA-LEARN		BRKGA	
		$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
10	1	**32.24**	97.06	32.14	114.59
10	2	30.56	66.05	**30.84**	95.25
50	1	**25.78**	151.84	24.98	227.49
50	2	**25.28**	100.28	24.86	183.25
100	1	**22.16**	107.93	19.34	180.61
100	2	**21.98**	120.53	20.08	149.02
150	1	**19.36**	127.27	16.76	154.21
150	2	**19.76**	161.05	16.68	147.78
200	1	**18.10**	136.83	14.72	145.22
200	2	**18.58**	120.13	14.70	160.61

Table 7. Comparison of BRKGA-LEARN and BRKGA on NON-RANDOM instances with string length $n = 500$.

m	*type*	BRKGA-LEARN		BRKGA	
		$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
10	1	66.18	39.68	**70.92**	129.82
10	2	**70.14**	60.28	64.78	117.03
50	1	58.58	160.06	**59.16**	287.05
50	2	**60.76**	218.02	55.32	187.35
100	1	**52.58**	205.24	49.30	321.75
100	2	**53.60**	222.17	51.00	277.44
150	1	**51.08**	286.49	46.52	344.74
150	2	**48.80**	285.49	48.78	315.31
200	1	**48.18**	327.75	43.62	354.31
200	2	**45.60**	368.48	43.60	407.89

Table 8. Comparison of BRKGA-LEARN and BRKGA on NON-RANDOM instances with string length $n = 1000$.

m	*type*	BRKGA-LEARN		BRKGA	
		$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}_{best}[s]$
10	1	90.50	96.65	**91.14**	113.29
10	2	90.92	130.95	**91.38**	102.84
50	1	**66.16**	154.49	65.40	284.90
50	2	**66.28**	157.41	63.94	255.81
100	1	**61.66**	228.93	59.60	346.65
100	2	**60.32**	223.04	57.88	266.50
150	1	**57.70**	286.83	55.74	306.60
150	2	**57.20**	258.09	54.38	195.92
200	1	**54.62**	282.27	52.46	72.89
200	2	**54.48**	268.08	52.06	149.85

In this case, BRKGA-LEARN obtains consistently and significantly better results than the original BRKGA with the exception of the instances with a very low number of input strings (m), for which the results are inconclusive. For all other instances, BRKGA-LEARN obtains better solutions than BRKGA, indicating a clear benefit from the proposed ML guidance in the case of non-uniform strings.

In order to measure the statistical significance of the differences between the obtained solution lengths of BRKGA-LEARN and BRKGA on the NON-RANDOM benchmark set, we employed the signed-rank Wilcoxon test [19]. It tests the one-sided alternative hypothesis that a solution value obtained by BRKGA-LEARN is in the expected case larger than the corresponding solution value obtained by BRKGA. We obtained a p-value of less than 10^{-4} indicating that the differences observed are statistically highly significant.

5 Conclusions and Future Work

This paper presented an example of how machine learning can be used to improve the performance of genetic algorithms. Particularly, we introduced a neural network based guidance for a biased random key genetic algorithm (BRKGA) applied to the longest common square subsequence (LCSqS) problem. BRKGA works on the basis of reducing from the LCSqS problem to the well-known longest common subsequence (LCS) problem. This is done by cutting each input string into two parts. The BRKGA is used to explore the set of possible cut points in the search for the best possible cut points. The presented machine learning guidance is implemented in order to leverage this search process. Given a string from a set of input strings, individual features for each input string are extracted, together with global ones. These are then fed into a neural network whose task it is to provide a presumably good cut point for the string. This information is then used inside BRKGA as greedy information.

We have experimentally evaluated the enhanced BRKGA (BRKGA-LEARN) against the original BRKGA using two sets of benchmark instances from the literature. BRKGA-LEARN has significantly improved the results of BRKGA for non-uniform instances. On the other hand, for the random instances, BRKGA turned out to not benefit as much from the machine learning guidance. The reason behind is that the original BRKGA naturally preferred greedy value for this benchmark set that simply consisted in biasing cut points towards the middle of the input strings.

As for future work, concerning the methodological aspects it would be interesting to try other machine learning models for regression as a replacement for the neural net and to extend the set of used features for the input strings. Concerning the training, another promising approach would be the application of reinforcement learning. Last but not least, the principles of the proposed learning-based approach are rather general, and it appears promising to apply this learning-based framework also to other BRKGA approaches for different hard optimization problems.

References

1. Bebis, G., Georgiopoulos, M.: Feed-forward neural networks. IEEE Potentials **13**(4), 27–31 (1994)
2. Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: a methodological tour d'horizon. Eur. J. Oper. Res. **290**(2), 405–421 (2021)
3. Blum, C., Blesa, M.J., Lopez-Ibanez, M.: Beam search for the longest common subsequence problem. Comput. Oper. Res. **36**(12), 3178–3186 (2009)
4. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
5. David, O.E., Greental, I.: Genetic algorithms for evolving deep neural networks. In: Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation, pp. 1451–1452 (2014)
6. Ding, S., Su, C., Yu, J.: An optimizing BP neural network algorithm based on genetic algorithm. Artif. Intell. Rev. **36**, 153–162 (2011)

7. Djukanovic, M., Raidl, G.R., Blum, C.: A beam search for the longest common subsequence problem guided by a novel approximate expected length calculation. In: Nicosia, G., Pardalos, P., Umeton, R., Giuffrida, G., Sciacca, V. (eds.) LOD 2019. LNCS, vol. 11943, pp. 154–167. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-37599-7_14
8. Djukanovic, M., Raidl, G.R., Blum, C.: A heuristic approach for solving the longest common square subsequence problem. In: Moreno-Diaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) Computer Aided Systems Theory – EUROCAST 2019. EUROCAST 2019. LNCS, vol. 12013, pp. 429–437. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45093-9_52
9. Inoue, T., Inenaga, S., Hyyrö, H., Bannai, H., Takeda, M.: Computing longest common square subsequences. In: 29th Annual Symposium on Combinatorial Pattern Matching, CPM 2018. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl Publishing (2018)
10. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
11. Luce, G., Frédéric Myoupo, J.: Application-specific array processors for the longest common subsequence problem of three sequences. Parallel Algorithms Appl. **13**(1), 27–52 (1998)
12. López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T., Birattari, M.: The irace package: iterated racing for automatic algorithm configuration. Oper. Res. Perspect. **3** (2011)
13. Maier, D.: The complexity of some problems on subsequences and supersequences. J. ACM **25**(2), 322–336 (1978)
14. Nakatsu, N., Kambayashi, Y., Yajima, S.: A longest common subsequence algorithm suitable for similar text strings. Acta Informatica **18**, 171–179 (1982)
15. Rahim Khan, M.A., Zakarya, M.: Longest common subsequence based algorithm for measuring similarity between time series: a new approach. World Appl. Sci. J. **24**(9), 1192–1198 (2013)
16. Reixach, J., Blum, C., Djukanovic, M., Raidl, G.: A biased random key genetic algorithm for solving the longest common square subsequence problem. SSRN: https://ssrn.com/abstract=4504431 or https://doi.org/10.2139/ssrn.4504431 (2023)
17. Wang, Q., Pan, M., Shang, Y., Korkin, D.: A fast heuristic search algorithm for finding the longest common subsequence of multiple strings. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 24, pp. 1287–1292 (2010)
18. Wang, Y.: Longest common subsequence algorithms and applications in determining transposable genes. arXiv preprint arXiv:2301.03827 (2023)
19. Woolson, R.F.: Wilcoxon signed-rank test. Wiley Encyclopedia of Clinical Trials, pp. 1–3 (2007)