Learning Beam Search: Utilizing Machine Learning to Guide Beam Search for Solving Combinatorial Optimization Problems*

M. Huber and G. R. Raidl

Algorithms and Complexity Group,
Institute of Logic and Computation, TU Wien, Austria
{mhuber,raidl}@ac.tuwien.ac.at

Abstract. Beam search (BS) is a well-known incomplete breadth-first-search variant frequently used to find heuristic solutions to hard combinatorial optimization problems. Its key ingredient is a guidance heuristic that estimates the expected length (cost) to complete a partial solution. While this function is usually developed manually for a specific problem, we propose a more general Learning Beam Search (LBS) that uses a machine learning model for guidance. Learning is performed by utilizing principles of reinforcement learning: LBS generates training data on its own by performing nested BS calls on many representative randomly created problem instances. The general approach is tested on two specific problems, the longest common subsequence problem and the constrained variant thereof. Results on established sets of benchmark instances indicate that the BS with models trained via LBS is highly competitive. On many instances new so far best solutions could be obtained, making the approach a new state-of-the-art method for these problems and documenting the high potential of this general framework.

Keywords: Beam Search · Combinatorial Optimization · Machine Learning · Longest Common Subsequence Problem

1 Introduction

Beam search (BS) is a prominent graph search algorithm frequently applied to heuristically solve hard planning and discrete optimization problems in limited time. In this context, it traverses a state graph from a root node, representing an initial state, in a breadth-first-search manner to find a best path to a target node. To keep the computational effort within limits, BS evaluates the reached nodes at each level and selects a subset of only up to β most promising nodes to continue with; the other nodes will not be pursued further, making BS an incomplete search. The subset of selected nodes at a current level is called *beam*,

* This project is partially funded by the Doctoral Program "Vienna Graduate School on Computational Optimization", Austrian Science Foundation (FWF), grant W1260-N35.

and parameter β *beam width*. In this way, BS continues level by level until there are no nodes to further expand. A shortest or longest path from the root node to a target node is finally returned as solution. As we consider here maximization problems, we assume w.l.o.g. that the goal is to find a longest path.

Clearly, the way how nodes are evaluated and selected for the beam plays a crucial role for the solution quality. Typically, the length of the longest path to a node so far is considered, and a heuristic value that estimates the maximum further length to go in order to reach a target node is added. This latter heuristic value is calculated by a function also called *guidance function* or guidance heuristic. It is typically developed in a manual, highly problem-specific way, frequently involving many computational experiments and comparisons of different options. Finding a promising guidance function is often challenging as the function not only needs to deliver good estimates but also needs to be fast as it is evaluated for each node in the BS.

The key idea of this work is to use a machine learning (ML) model as guidance function in BS, more specifically a neural network (NN), to approximate the maximum further length to go from a current node to reach a target node. In such an approach, it is a challenge to train the ML model appropriately. Classical supervised learning would mean that labeled training data is available in the form of problem-specific nodes (states) plus *real/exact* maximum path lengths to target nodes. Such data would only be obtainable with huge computational effort and for smaller problem instances. With the BS, however, we primarily want to address large problem instances that cannot practically be solved exactly. Concepts from *reinforcement learning* come to our rescue: In our *Learning Beam Search* (LBS) we start with a randomly or naively initialized ML model and create training data on the fly by performing the search many times on representative, randomly created problem instances. Better estimates for the maximum lengths to go than the ML model usually delivers are determined for subsets of reached nodes by means of *nested BS calls*. This generated training data is buffered in a FIFO replay buffer and used to continuously train the ML model, intertwined with the LBS's further training data production.

While the general principle of this LBS is quite generic, we consider here two well-known NP-hard problems as specific case studies: the Longest Common Subsequence (LCS) problem and the Constrained Longest Common Subsequence (CLCS) problem. Our experimental results show that for both problems, LBS automatically trained on independent random instances is able to compete with the so far leading approaches and in many cases obtains better solutions in comparable runtimes.

Section 2 reviews related work. In Section 3, we present the new LBS in a problem-independent way. The LCS and CLCS problems are introduced in Section 4. The problem-specific state graphs and how the guidance functions are specifically realized by NNs are described in Sections 5 and 6, respectively. Results of computational experiments are discussed in Section 7. Finally, we conclude in Section 8, where we also outline promising future work.

2 Related Work

The increasing popularity of ML also affected classical combinatorial optimization. There is a growing interest in utilizing ML to better solve hard discrete problems. While end-to-end ML approaches to combinatorial optimization also have been attempted by a number of researchers and appear promising, see, e.g., [5], these approaches are usually still not competitive with state-of-the-art problem-solving techniques. However, a broader range of approaches has been suggested to improve classical optimization methods with ML components.

In the context of tree search techniques, one approach is *imitation learning*, i.e., to learn a heuristic by imitating an expert's behavior. In this direction, He et al. [11] proposed to speed up a branch-and-bound by learning a node selection and pruning policy from solving training problems given by an oracle that knows optimal solutions. Concerning variable branching in mixed integer programming, Khalil et al. [15] suggested a ML framework that attempts to mimic the decisions made by strong branching through solving a learning to rank problem. Moreover, Khalil et al. [16] introduced a framework for learning a binary classifier to predict the probability of whether a heuristic will succeed at a given node of a search tree. Training data is collected by running a heuristic at every node at the search tree, gathering the binary classification labels. A general learning to search framework that uses a retrospective oracle to generate feedback by querying the environment on roll-out search traces to improve itself after initial training by an expert was suggested by Song et al. [26].

AlphaGo and its successor AlphaZero gained broader recognition as agents excelling in the games of Go, chess, and shogi [25]. They are based on Monte Carlo tree search in which a deep NN is used to evaluate game states, i.e., to estimate their values in terms of the probabilities to win or lose. Additionally, the NN provides a policy in terms of a probability distribution over the next possible moves. Training is done via reinforcement learning by self-play. Thus, training data is continuously produced by simulating many games against itself, stored in a replay buffer, and used to continuously improve the NN. We apply a similar principle also in our LBS. Several researchers adapted AlphaZero to address combinatorial optimization problems: For example, Laterre et al. [18] applied it to a 3D packing problem, Abe et al. [1] to problems on graphs including minimum vertex cover and maximum cut, and Huang et al. [12] to graph coloring. The latter two approaches used different kinds of graph neural networks as ML models. Mittal et al. [21] suggested another form of heuristic tree search for various graph problems that is guided by a graph neural network. Here, training is done on the basis of smaller instances with known solutions in a supervised fashion, but results indicate that the approach generalizes well to larger instances not seen during training. In the more general context of metaheuristics, a recent survey on utilizing ML can be found in [14].

Beam search was originally proposed in the context of speech recognition [19]. Since then it has been applied in a variety of areas including machine translation [27] and syntactic parsing [29]. Concerning combinatorial optimization problems, many applications exist in particular in the domains of scheduling,

see, e.g., [23,8,3], and string-related problems originating in bioinformatics, see, e.g., [13,7,6], but also packing [2].

Concerning specifically the guidance of BS by a ML model, we are only aware of the work by Negrinho et al. [22], who examined this topic from a pure theoretical point of view. They formulated the approach as learning a policy for an abstract structured prediction problem to traverse the combinatorial search space of beams and presented a unifying meta-algorithm as well as novel no-regret guarantees for learning beam search policies using imitation learning.

3 Learning Beam Search

We consider a discrete maximization problem that can be expressed as a longest path problem on a (possibly huge) directed acyclic state graph $G = (V, A)$ with nodes V and arcs A. Each node $v \in V$ represents a problem-specific state, for example, the partial assignment of values to the decision variables in a solution. An arc $(u, v) \in A$ exists between nodes $u, v \in V$ if and only if state v can be obtained from state u by a valid problem-specific action, such as the assignment of a specific feasible value to a so far unassigned decision variable in state u. Let label $\ell(u, v)$ denote this action transitioning from state u to state v. There is one dedicated root node $r \in V$ representing the initial state, in which typically all decision variables are unassigned. Moreover, there are one or more target nodes $T \subset V$, which have no outgoing arcs and represent valid final states, e.g., in which all decision variables have feasible values. Note that this definition of the state graph also covers classical branching trees. Each arc $(u, v) \in A$ has associated a length (or cost) $c(u, v)$. Any path from the root node r to a target node $t \in T$ represents a feasible solution, and we assume that its length, which is the sum of the path's edge lengths, corresponds to the objective value of the solution. As we consider a maximization problem, we seek a longest r–t path, over all $t \in T$.

Our LBS builds upon classical BS, i.e., a breadth-first-search in which at each level a subset of at most β nodes, called the beam B, is selected and pursued further. This selection is performed by evaluating each node u of the current level with the evaluation function $f(v) = g(v) + h(v)$, where $g(v)$ corresponds to the length of a longest so far identified path from the root r to node v, and $h(v)$ is a heuristic guidance function estimating the maximum further length to go to some target node. Note that in an implementation values $g(v)$ are stored with each node v as well as a reference to a predecessor node $u = \mathit{pred}(v)$ on a maximum length path, and thus, $g(v) = g(u) + c(u, v)$; only the root node has no predecessor. In this way, once a target node t is reached, a maximum length r–t path within the investigated part of graph G can be efficiently identified, and the corresponding solution is obtained via the respective arc labels.

As already stated in the introduction, the heuristic guidance function $h(v)$ estimating the length to go is usually crafted manually in a problem-specific way. In our LBS, however, we use an ML model. Still, problem-specific aspects will play a role in the choice of the specific model, in particular, which features are

Algorithm 1 Learning Beam Search (LBS)

Input: nr. of iterations z, beam width β, exp. nr. of training samples per instance α, NBS beam width β', replay buffer size ρ, min. buffer size for training γ
Output: trained guidance function h
$h \leftarrow$ untrained guidance function h (ML regression model)
$R \leftarrow \emptyset$ // replay buffer: FIFO of max. size ρ
for z iterations **do**
    $I \leftarrow$ create representative random problem instance
    Beam Search with training data generation (I, β, R, α, β')
    if $|R| \geq \gamma$ **then**
        (re-)train h with data from R
    end if
end for
return h

derived from a problem-specific state and which kind of ML is actually used. But for now, it is enough to assume that $h(v)$ is a learnable function mapping a state to a scalar value in $\mathbb{R}$.

The core idea of LBS is to train function h via self-learning by iterated application on many random instances generated according to the properties of the instances expected in the future application. The principle is comparable to how learning takes place in AlphaZero [25]. A pseudocode for the main part of the LBS is shown in Alg. 1. It maintains an initially empty replay buffer R which will contain the training data. This buffer is realized as a first-in first-out (FIFO) queue of maximum size γ. The idea hereby is to also remove older, outdated training samples when the guidance function has already been improved. A certain number (z) of iterations is then performed. In each iteration, a new independent random problem instance is created and the actual BS applied. This BS, however, is extended by a training data generation that adds in the expected case α new training samples with labels to the replay buffer R; details on this data generation will follow below. After each BS run, a check is performed to determine if the buffer R already contains a minimum number of samples γ, and if this is the case, the guidance function is (re-)trained with data from R. As this training is performed in each iteration, it is usually enough to do a small incremental form of training if the ML model provides this possibility. More specifically, we will use a neural network and train for one epoch over R with mini-batches of size 32. The improved guidance function is then immediately used in the next BS call.

Algorithm 2 shows the actual BS, which is enhanced by the optional training data generation via *nested beam search* (NBS) calls. It receives as input parameters a specific problem instance I to solve, the beam width β, and when training data should be generated the replay buffer R to which the new samples will be added, the expected number of samples to generate α, and a possibly different beam width β' for the NBS. The procedure starts by initializing the beam B with the single root node created for the problem instance I. The outer while-loop

Algorithm 2 Beam Search with optional training data generation

```
1: Input: problem instance I, beam width β,
2: only when training data should be generated: replay buffer R, exp. nr. of samples α,
   NBS beam width β'
3: Output: best found target node t
4: B ← {r} with r being a root node for problem instance I
5: t ← none  // so far best target node
6: while B ≠ ∅ do
7:     V_ext ← ∅
8:     for v ∈ B do
9:         expand v by considering all valid actions, add obtained new nodes to V_ext
10:    end for
11:    for v ∈ V_ext do
12:        evaluate node by f(v) = g(v) + h(v)
13:        filter dominated nodes (optional, problem-specific)
14:        if v ∈ T ∧ t = none ∨ g(t) < g(v) then
15:            // new best terminal node encountered
16:            t ← v
17:        end if
18:        if R given ∧ rand() < α/n_nodes then  // generate training sample?
19:            t' ← Beam Search (I(v), β')  // NBS call
20:            add training sample (v, g(t')) to R
21:        end if
22:    end for
23:    B ← select (up to) β nodes with largest f-values from V_ext
24: end while
25: return t
```

performs the BS level by level until B becomes empty. In each iteration, each node in the beam is expanded by considering all feasible actions for the state the node represents and creating respective successor nodes. These are added to set V_{ext}. Each node in V_{ext} is then evaluated by calculating $g(v)$, $h(v)$ as well as the sum $f(v)$. Optionally and depending on the specific problem, domination checks and filtering can be applied to reduce V_{ext} to only meaningful nodes. Next, line 14 checks if a training sample should be created from the current node v, which is done with probability α/n_{nodes} when the replay buffer R has been provided. Hereby, n_{nodes} is an estimate of the total number of (non-dominated) nodes a whole BS run creates so that we can expect to obtain about α samples. More specifically, in our implementation we initially set $n_{\text{nodes}} = 0$ for the very first LBS iteration, actually producing no training data but counting the number of overall produced nodes, and update n_{nodes} for each successive iteration by the average number of nodes produced over all so far performed LBS iterations. Thus, n_{nodes} is adaptively adjusted. To actually obtain a training sample for a current node v, the sub-problem instance $I(v)$ to which state v corresponds is determined, and an independent NBS call is performed for this subproblem with beam width β'. This NBS returns the target node t' of a longest identified path

from node v onward, and thus $g(t')$ will typically be a better approximation to the real maximum path length than $h(v)$. State v and value $g(t')$ are therefore together added as training sample and respective label (target value) to the replay buffer.

Computational complexity. Let us assume that the expansion and evaluation of one node takes the problem-specific time T_{node} and the maximum height of the BS tree is H. One NBS call then requires time $O(\beta' \cdot H \cdot T_{\text{node}})$. Considering that LBS performs z iterations and in each makes in the expected case α NBS calls, we obtain that LBS runs in $O(z \cdot (\beta + \alpha \cdot \beta') \cdot H \cdot T_{\text{node}})$ total time.

4 Case Studies

We test the general LBS approach specifically on the following two problems.

The Longest Common Subsequence (LCS) Problem. A string is a sequence of symbols from an alphabet Σ. A subsequence of a string s is a sequence derived by deleting zero or more symbols from that string without changing the order of the remaining symbols. A *common subsequence* of a set of m non-empty strings $S = \{s_1, \ldots, s_m\}$ is a subsequence that all these strings have in common. The LCS problem seeks a common subsequence of maximum length for S. For example, the LCS of strings AGACT, GTAAC, and GTACT is GAC.

The LCS problem is well-studied and has many applications in particular in bioinformatics, where it is used to find relationships among DNA, RNA, or protein sequences. For $m = 2$ strings the problem can be solved efficiently [10], while for general m it is NP-hard [20]. Many heuristics have been proposed for the general LCS problem, and most so far leading ones rely on BS. See [7] for a state-of-the-art method and a rigorous comparison of methods. The BS proposed in [7] utilizes a sophisticated guidance function that approximates the expected LCS length for the remaining input string lengths assuming uniform random strings.

Notations. For a string s, we denote its length by $|s|$. Let $n = \max_{s_i \in S} |s_i|$ be the maximum input string length. The j-th letter of a string s is $s[j]$, with $j = 1, \ldots, |s|$. By $s[j, j']$ we refer to the substring of s starting with $s[j]$ and ending with $s[j']$ if $j \leq j'$ or the empty string ε else. Let $|s|_a$ be the number of occurrences of letter $a \in \Sigma$ in string s.

As in previous work [7], we prepare the following data structure in preprocessing to allow an efficient "forward stepping" in the strings. For each $i = 1, \ldots, m$, $j = 1, \ldots, |s_i|$, and $c \in \Sigma$, $succ[i, j, c]$ stores the minimal position j' such that $j' \geq j \wedge s_i[j'] = c$ or 0 if c does not occur in s_i from position j onward.

The Constrained Longest Common Subsequence (CLCS) Problem. This problem extends the LCS problem on m input strings by additionally considering a *pattern string* P that *must* appear as subsequence in a solution.

For $m = 2$ input strings besides the pattern string, this problem can again be solved efficiently, see, e.g., [28], but for general m the problem also is NP-hard.

Concerning heuristics to address large instances of this general variant, there exists an approximation algorithm [9], which, however, is in practice clearly outperformed by the BS approaches in [6]. One of these BSs is of similar nature as the above-mentioned BS for the LCS problem [7] as it also utilizes an expected length calculation, however, it required a careful extension to consider the pattern string.

As additional data structure, a table $embed[i, j]$, $i = 1, \ldots, m$, $j = 1, \ldots, |P|$ that stores the right-most position j' in input string s_i such that $P[j, |P|]$ is a subsequence of $s_i[j', |s_i|]$ is prepared here during preprocessing.

5 State Graphs for the LCS and CLCS Problems

The state graph $G = (V, A)$ searched by our LBS for solving the LCS problem corresponds to the one used in former work [7]. We therefore only briefly summarize the main facts. A state (node) v is represented by a *position vector* $p^v = (p_i^v)_{i=1,\ldots,m}$ with $p_i^v \in 1, \ldots, |s_i| + 1$, indicating the still relevant substrings of the input strings $s_i[p^v, |s_i|]$, i=1,. . . ,m. Note that these substrings form the LCS subproblem instance $I(v)$ induced by node v, for which LBS may perform an independent NBS call to obtain a target value for training. The root node $r \in V$ has position vector $p^r = (1, \ldots, 1)$, and thus, $I(v)$ corresponds to the original LCS instance. An arc $(u, v) \in A$ refers to transitioning from state u to state v by appending a valid letter $a \in \Sigma$ to a partial solution, and thus, arc (u, v) is labeled by this letter, i.e., $\ell(u, v) = a$. In other words, appending letter $a \in \Sigma$ to a partial solution at state u only is feasible if $succ[i, p_i^v, a] > 0$ for $i = 1, \ldots, m$, and yields in this case state v with $p_i^v = succ[i, p_i^v, a] + 1,\ i = 1, \ldots, m$. States that allow no feasible extension are jointly represented by the single terminal node $t \in V$ with $p^t = (|s_i| + 1)_{i=1,\ldots,m}$. As the objective is to find a maximum length string, and with each arc always one letter is appended to a partial solution, the length (cost) of each arc $(u, v) \in A$ is here $c(u, v) = 1$, and thus, $g(v)$ corresponds to the number of arcs of the longest identified r–v path.

In case of the CLCS problem, we also need to consider pattern string P. The position vector is therefore extended by an additional value p_{m+1}^v indicating the position from which on P is not yet covered by the partial solutions leading to state v. A letter $a \in \Sigma$ is only feasible as extension, if the state that would be obtained by it still allows to cover the remaining pattern string, i.e., if $succ[i, p_i^v, c(a)] + 1 \leq embed[i, p_{m+1}^v]$ for $i = 1, \ldots, m$.

For both, the LCS and the CLCS problem, dominance checks and filtering are performed in our LBS exactly as described in [7] and [6], respectively.

6 ML Models for the LCS and CLCS Problems

In principle, any ML regression model may be considered for LBS as guidance function $h(v)$. Clearly, the model needs to be flexible enough, and providing the possibility of incremental learning is a particular advantage in the context of the LBS. Therefore, we consider here for both of our test problems a simple dense

feedforward NN with two hidden layers, both equipped with ReLU activation functions. The output layer consists of a single neuron without activation function only – remember that its value is supposed to approximate the maximum further length to go from state v.

Formally, we have defined $h(v)$ to directly receive a state v as input. However, it makes sense to consider the actual input used for the NN more carefully. As an intermediate step, we transform the raw state (and problem instance) information into a more meaningful *feature vector*, which is then actually provided to the NN.

The well-working guidance heuristic from [7] is based on the *remaining string lengths* $|s_i| - p_i^v + 1$, $i = 1, \ldots, m$, only. Therefore, we also use them as features for our NN. Note that the order of the strings and therefore also these values are irrelevant. To avoid possible difficulties in learning these symmetries, we avoid them by always sorting the remaining string lengths before providing them as input to the NN.

In case of the CLCS problem, we additionally have the position p_{m+1}^v in the pattern string P as part of the state, and consequently, we also provide $|P| - p_{m+1}^v + 1$ as an additional feature.

Moreover, earlier guidance heuristics for the LCS problem rely on the *minimum numbers of letter appearances* $\min_{i=1,\ldots,m} |s_i[p_i^v, |s_i|]|_c$, $c \in \Sigma$, from which also a (usually weak) lower bound on the solution length may be calculated. Therefore, we also provide these values for both problems as further features to the NN.

The NN is initialized with random weights. Once the replay buffer has reached the minimum fill level of γ samples, incremental training is done in each LBS iteration by sampling one mini-batch of 32 random samples from the replay buffer and applying the ADAM optimizer with step size 0.001 and exponential decay rates for the moment estimates 0.9 and 0.999 as recommended in [17]. As loss function we use the mean squared error.

7 Experimental Evaluation

We implemented LBS in Julia 1.6 using the Flux package for the NN. All experiments were performed in single-threaded mode on a machine with an Intel Xeon E5–2640 processor with 2.40 GHz and a memory limit of 20 GB. Benchmark instances are grouped by the alphabet size $|\Sigma|$, the number of input strings m, and the maximum string length n. LBS was applied to train a NN for each combination of $|\Sigma|$, m, and n. Remember that this learning takes place on the basis of independent random instances that LBS creates on its own. Finally, the benchmark instances are used to evaluate the performance of the BS using the correspondingly trained NN as guidance function. Preliminary tests led to the following LBS configuration that turned out to be suitable for all our benchmarks unless stated otherwise: no. of LBS iterations $z = 1000$, min. buffer size for learning $\gamma = 3000$, LBS and NBS beam widths $\beta = \beta' = 50$, max. buffer size $\rho = 5000$, and exp. nr. of training samples generated per instance $\alpha = 60$. In the

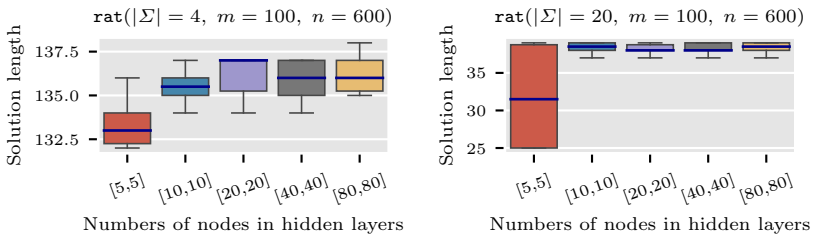

Fig. 1. Impact of the numbers of nodes in the hidden layers on the solution length of LBS on `rat` benchmark instances.

following, we first discuss the experiments for the LCS problem and then those for the CLCS problem.

7.1 LCS Experiments

For the LCS problem, two frequently used benchmark sets are considered. The first one denoted as `rat` was introduced in [24] and consists of 20 instances composed of sequences from rat genomes. The sequences of these instances are close to independent random strings, each sequence has length $n = 600$, but all instances differ in their combinations of values for $|\Sigma|$ and m. The second benchmark set `BB` from [4] consists of 80 random instances for eight different combinations of $|\Sigma|$ and m (ten instances per combination) and all have string lengths up to $n = 1000$. These instances stand out in that the strings of each exhibit large similarities. Consequently, we generated the random instances within the LBS in the same manner, to obtain suitable NNs specifically for this kind of instances.

A parameter of major importance is the number of hidden nodes in the NN. Clearly, the network size has a direct impact on the computation times of the BS as the guidance function needs to be evaluated for each non-terminal node. Thus, we want to make the NN as small as possible, but at the same time, large enough to get high-quality predictions. In order to examine this aspect, we made tests with different NN configurations. Figure 1 shows exemplary box plots for final LCS lengths obtained from ten LBS runs per NN configuration on selected `rat` instances. We conclude that 20 nodes in both hidden layers are a robust choice. Smaller NNs are sometimes too restrictive, occasionally implying significantly worse results. Therefore, we use this configuration in all further experiments.

Next, we investigate the impact of the beam width on the solution quality, performing again ten runs per configuration. Figure 2 shows respective boxplots. The same beam width has been used for the LBS (β), for the NBS calls (β'), as well as for the final testing on the `rat` benchmark instances. As one may expect, larger beam widths in general yield better results. In particular, using NBS beam widths of $\beta' \leq 30$ turned out to yield clearly inferior results. Therefore, we set $\beta = \beta' = 50$ in all further experiments if not indicated otherwise.

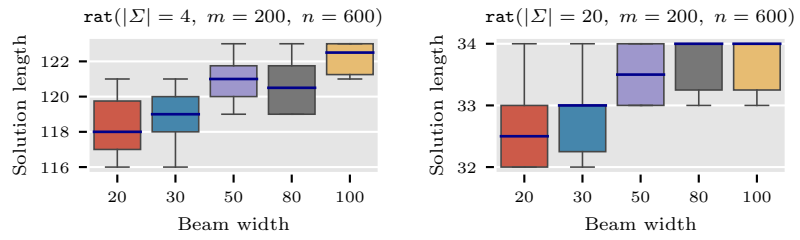

Fig. 2. Impact of beam width $\beta = \beta'$ in training and testing on `rat` instances.

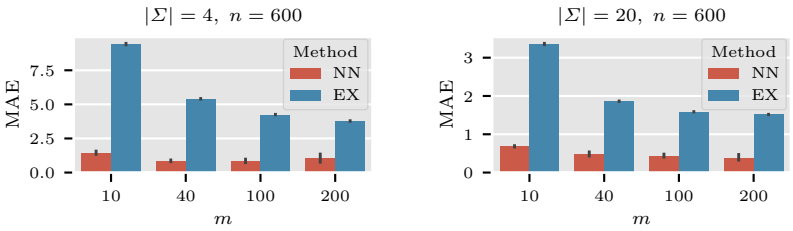

Fig. 3. Mean absolute error of the trained NNs and EX on test samples created by a BS with EX guidance function.

Of interest also is how well a NN trained by LBS actually approximates the real LCS length. As we cannot obtain exact LCS lengths for instances of interesting size, we approximate them by applying the so far leading BS with the approximate expected length (EX) guidance function from [7]. More specifically, to consider instances with a broad range of different input string lengths, we generated 10000 labeled test samples by LBS using EX as guidance function in the outer BS as well as in the NBS calls instead of an NN. This was done for $|\Sigma| \in \{4, 20\}$, $m \in \{10, 40, 100, 200\}$, and $n = 600$. Ten NNs were then trained by LBS for each configuration, and these NNs as well as EX were tested on the generated data sets. Figure 3 shows obtained Mean Absolute Errors (MAEs); standard deviations are indicated by the small black lines. We can observe that the NNs approximate the LCS lengths much better than EX, and differences are particularly large for smaller m. The MAE of EX is about four to six times as large as the MAE of the NNs.

Finally, we compare our approach to the state-of-the-art methods from the literature. While all training with LBS was done with $\beta = \beta' = 50$, we do the tests on the benchmark instances following [7] with two different beam widths: aiming for *low (computation) time* with $\beta = 50$ and aiming for *high quality* with $\beta = 600$. Table 1 shows obtained results. For our LBS, the average solution length $|s_{\text{LBS}}|$ and the runtime of the BS with the trained NN t_{LBS} are listed for each instance group. Columns $|s_{\text{BS-EX}}|$ and $t_{\text{BS-EX}}$ show the respective solution qualities and runtimes for the BS from [7] with the EX guidance function. For a

Table 1. LCS results on benchmark sets `BB` and `rat`.

				low times					high quality				
Set	$\|\Sigma\|$	m	n	$\|s_{\text{LBS}}\|$	t_{LBS} [s]	$\|s_{\text{BS-EX}}\|$	$t_{\text{BS-EX}}$ [s]	$\|s_{\text{lit-best}}\|$	$\|s_{\text{LBS}}\|$	t_{LBS} [s]	$\|s_{\text{BS-EX}}\|$	$t_{\text{BS-EX}}$ [s]	$\|s_{\text{lit-best}}\|$
BB	2	10	1000	651.2	0.855	635.1	0.824	**662.9**	673.1	12.044	673.5	9.180	**676.5**
BB	2	100	1000	***556.1**	1.550	525.1	1.765	551.0	***565.8**	22.979	536.6	18.368	560.7
BB	4	10	1000	***540.1**	1.262	453.0	0.954	537.8	**545.4**	18.112	545.2	12.467	**545.4**
BB	4	100	1000	***381.3**	2.591	318.6	2.174	371.2	***392.9**	35.331	329.5	24.233	388.8
BB	8	10	1000	462.4	1.452	338.8	1.270	**462.6**	**462.7**	28.232	**462.7**	19.155	**462.7**
BB	8	100	1000	***267.4**	4.319	198.0	3.257	260.9	***274.8**	60.682	210.6	36.785	272.1
BB	24	10	1000	**385.6**	5.430	**385.6**	4.172	**385.6**	**385.6**	67.455	**385.6**	48.177	**385.6**
BB	24	100	1000	***148.2**	10.314	95.8	9.399	147.0	**149.5**	153.194	113.3	138.174	**149.5**
rat	4	10	600	199.0	0.550	198.0	1.138	**201.0**	**205.0**	8.591	**205.0**	4.240	204.0
rat	4	15	600	***184.0**	0.660	182.0	1.134	182.0	**185.0**	9.097	**185.0**	7.276	184.0
rat	4	20	600	**169.0**	0.620	168.0	2.082	**169.0**	***173.0**	8.082	172.0	4.120	170.0
rat	4	25	600	166.0	0.766	**167.0**	1.182	166.0	***171.0**	9.295	170.0	4.766	168.0
rat	4	40	600	***152.0**	0.844	146.0	1.172	151.0	***156.0**	10.064	152.0	5.265	150.0
rat	4	60	600	149.0	0.868	**150.0**	1.315	149.0	**152.0**	12.129	**152.0**	12.016	151.0
rat	4	80	600	***138.0**	1.056	137.0	1.368	137.0	140.0	12.564	**142.0**	13.292	139.0
rat	4	100	600	***135.0**	0.483	131.0	1.408	133.0	**137.0**	13.650	**137.0**	7.739	135.0
rat	4	150	600	**127.0**	1.176	**127.0**	2.734	125.0	***130.0**	11.625	129.0	16.841	126.0
rat	4	200	600	**121.0**	1.572	**121.0**	1.733	**121.0**	**123.0**	14.117	**123.0**	19.567	**123.0**
rat	20	10	600	**70.0**	1.108	**70.0**	2.501	**70.0**	**71.0**	10.104	**71.0**	7.579	**71.0**
rat	20	15	600	**62.0**	1.117	**62.0**	2.660	61.0	**63.0**	12.048	**63.0**	13.448	62.0
rat	20	20	600	***54.0**	1.059	53.0	2.553	53.0	**54.0**	13.704	**54.0**	7.970	**54.0**
rat	20	25	600	***51.0**	1.152	50.0	2.545	50.0	**52.0**	13.073	**52.0**	13.573	51.0
rat	20	40	600	***49.0**	0.529	47.0	2.872	48.0	**49.0**	16.005	**49.0**	8.801	**49.0**
rat	20	60	600	**46.0**	1.945	**46.0**	3.234	**46.0**	**47.0**	19.734	46.0	13.413	**47.0**
rat	20	80	600	42.0	1.953	41.0	2.236	**43.0**	43.0	24.741	43.0	23.051	**44.0**
rat	20	100	600	**38.0**	2.007	**38.0**	3.932	**38.0**	39.0	24.441	**40.0**	25.239	39.0
rat	20	150	600	***37.0**	2.457	36.0	2.481	36.0	**37.0**	28.719	**37.0**	29.312	**37.0**
rat	20	200	600	**34.0**	2.048	**34.0**	3.189	**34.0**	**34.0**	32.118	**34.0**	26.838	**34.0**

fair time comparison, we re-implemented this approach in our Julia-framework and list the times measured by us, while the solution lengths correspond to those reported in [7]. Last but not least, so far best known solution lengths from other approaches, as also reported in [7], are shown in column $|s_{\text{lit}-\text{best}}|$. Best solution lengths are printed bold, and new best ones obtained by LBS are additionally marked with an asterisk. In 13 out of 28 cases from the low time experiments and in 7 out of 28 cases from the high quality experiments, new best results could be achieved by LBS. In the remaining cases, the quality of the LBS solutions either matched so far best results or were only by a small amount behind. Concerning runtimes, we can conclude that they are very similar to those of BS-EX.

7.2 CLCS Experiments

For the CLCS problem, we use the benchmark set from [6]: ten instances for each combination of $|\Sigma| \in \{4, 20\}$, $m \in \{10, 50, 100\}$, and $n \in \{100, 500, 1000\}$, and ratios of $\frac{n}{|P|} \in \{4, 10\}$ concerning the pattern strings. Note that it is guaranteed that the pattern string appears in the input strings in the way the instances were created, for details on the creation see [6]. We compare the results of the following seven methods from the literature with those obtained by the LBS: the approximation algorithm from [9] (Approx), and Greedy, Random, BS-UB, BS-Prob, BS-EX, and BS-Pat from [6]. In all BS approaches, the same beam width $\beta = 2000$ was used for the tests on the benchmark instances. Results are shown in Table 2. Here, in ten out of 36 cases, new best results could be achieved

Table 2. Results for the CLCS problem on benchmark instances from [6].

$\frac{n}{\lvert\Sigma\rvert}$	$\lvert\Sigma\rvert$	m	n	$\lvert s_{\text{LBS}}\rvert$	t_{LBS} [s]	$\lvert s_{\text{Approx}}\rvert$	$\lvert s_{\text{Greedy}}\rvert$	$\lvert s_{\text{Random}}\rvert$	$\lvert s_{\text{BS-UB}}\rvert$	$\lvert s_{\text{BS-Prob}}\rvert$	$\lvert s_{\text{BS-EX}}\rvert$	$\lvert s_{\text{BS-Pat}}\rvert$
4	4	10	100	**34.5**	0.198	28.6	32.2	31.4	**34.5**	**34.5**	**34.5**	**34.5**
4	4	50	100	**27.5**	0.009	26.4	26.9	26.9	**27.5**	**27.5**	**27.5**	**27.5**
4	4	100	100	**26.5**	0.006	25.9	26.2	26.1	**26.5**	**26.5**	**26.5**	**26.5**
4	4	10	500	***183.2**	12.080	134.3	160.4	153.8	179.3	182.4	181.1	168.6
4	4	50	500	147.9	9.912	130.1	139.5	138.1	146.2	**148.3**	146.3	142.7
4	4	100	500	140.6	10.387	128.9	135.8	134.5	140.4	**140.8**	140.3	137.3
4	4	10	1000	***366.5**	27.570	264.7	317.4	308.1	350.3	361.7	361.4	330.8
4	4	50	1000	***296.6**	23.909	257.4	277.3	274.5	291.9	296.4	289.5	284.2
4	4	100	1000	**282.5**	27.273	256.4	270.7	268.1	279.7	**282.5**	279.0	273.3
4	20	10	100	**25.0**	0.005	**25.0**	**25.0**	**25.0**	**25.0**	**25.0**	**25.0**	**25.0**
4	20	50	100	**25.0**	0.005	**25.0**	**25.0**	**25.0**	**25.0**	**25.0**	**25.0**	**25.0**
4	20	100	100	**25.0**	0.005	**25.0**	**25.0**	**25.0**	**25.0**	**25.0**	**25.0**	**25.0**
4	20	10	500	**125.0**	0.006	**125.0**	**125.0**	**125.0**	**125.0**	**125.0**	**125.0**	**125.0**
4	20	50	500	**125.0**	0.009	**125.0**	**125.0**	**125.0**	**125.0**	**125.0**	**125.0**	**125.0**
4	20	100	500	**125.0**	0.012	**125.0**	**125.0**	**125.0**	**125.0**	**125.0**	**125.0**	**125.0**
4	20	10	1000	**250.0**	0.012	**250.0**	**250.0**	**250.0**	**250.0**	**250.0**	**250.0**	**250.0**
4	20	50	1000	**250.0**	0.014	**250.0**	**250.0**	**250.0**	**250.0**	**250.0**	**250.0**	**250.0**
4	20	100	1000	**250.0**	0.031	**250.0**	**250.0**	**250.0**	**250.0**	**250.0**	**250.0**	**250.0**
10	4	10	100	**34.6**	2.628	22.9	29.6	26.5	**34.6**	**34.6**	34.3	32.1
10	4	50	100	***25.1**	3.307	19.8	21.8	21.0	24.9	25.0	24.3	23.5
10	4	100	100	**23.0**	3.668	18.9	20.8	19.6	**23.0**	**23.0**	21.9	21.5
10	4	10	500	***186.3**	21.081	121.4	163.7	147.9	182.2	185.0	184.8	165.9
10	4	50	500	***143.4**	26.723	114.2	129.5	123.6	138.7	142.9	141.8	131.2
10	4	100	500	***133.8**	37.620	111.3	122.0	118.3	129.2	133.3	132.0	124.3
10	4	10	1000	***377.2**	43.442	245.5	329.1	294.8	365.0	375.8	376.3	330.4
10	4	50	1000	***290.9**	56.531	233.5	266.5	254.9	279.6	289.2	290.4	266.0
10	4	100	1000	***272.7**	79.228	230.3	253.2	246.8	262.3	270.9	272.1	255.2
10	20	10	100	**10.2**	0.004	**10.2**	10.1	**10.2**	**10.2**	**10.2**	**10.2**	**10.2**
10	20	50	100	**10.0**	0.005	**10.0**	**10.0**	**10.0**	**10.0**	**10.0**	**10.0**	**10.0**
10	20	100	100	**10.0**	0.005	**10.0**	**10.0**	**10.0**	**10.0**	**10.0**	**10.0**	**10.0**
10	20	10	500	**53.1**	0.007	51.0	52.5	52.7	**53.1**	**53.1**	**53.1**	**53.1**
10	20	50	500	**50.0**	0.006	**50.0**	**50.0**	**50.0**	**50.0**	**50.0**	**50.0**	**50.0**
10	20	100	500	**50.0**	0.008	**50.0**	**50.0**	**50.0**	**50.0**	**50.0**	**50.0**	**50.0**
10	20	10	1000	**105.4**	0.011	101.0	103.9	104.6	**105.4**	**105.4**	**105.4**	**105.4**
10	20	50	1000	**100.0**	0.009	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
10	20	100	1000	**100.0**	0.012	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

by LBS, and it scores worse in only two out of 36 cases; in the remaining cases, the solutions values from LBS are equal to the so far best known ones.

8 Conclusions and Future Work

We presented a general learning beam search framework to solve combinatorial optimization problems for which the solution space can be represented by a state graph. Instead of the frequently challenging manual design of a meaningful guidance function, we train a regression model to approximate the real length to go from a state and use this model thereafter in a BS. Training is done in the spirit of reinforcement learning by performing many BS runs on randomly created instances and calling a nested beam search to obtain labeled training data. Our case studies on the LCS and the CLCS problems clearly show that this learning approach can be highly effective. On many benchmark instances new best solutions could be obtained, making this approach a new state-of-the-art method for the considered two problems.

Clearly, the proposed LBS is not entirely problem-agnostic: Still, it is important to use a suitable state space, to derive meaningful features from states, and to choose an appropriate ML model for a problem at hand. Moreover, note

that in our implementation for the LCS and CLCS, individual models need to be trained for specific choices of $|\Sigma|$, m, and n. In future work specifically for the LCS and CLCS problems, we aim at relying on different features that just describe the distribution of remaining input string lengths, in order to learn models that are independent of m and possibly also n.

General improvement potential for LBS lies in the fact that the guidance function actually does not need to approximate the length to go well, but only needs to provide scores for ranking the solutions in the beam. Can this flexibility be used to come up with alternative optimization targets and loss functions for the training, yielding overall better results? Parallelization and the utilization of GPUs are further natural possibilities to speed up in particular the learning. Last but not least, we aim at applying LBS to further problems and to also investigate other ML models than NNs.

References

1. Abe, K., Xu, Z., Sato, I., Sugiyama, M.: Solving NP-hard problems on graphs with extended AlphaGo Zero. arXiv:1905.11623 [cs, stat] (2020)
2. Akeba, H., Hifib, M., Mhallah, R.: A beam search algorithm for the circular packing problem. Computers & Operations Research **36**(5), 1513–1528 (2009)
3. Blum, C., Miralles, C.: On solving the assembly line worker assignment and balancing problem via beam search. Computers & Operations Research **38**(1), 328–339 (2011)
4. Blum, C., Blesa, M.J.: Probabilistic beam search for the longest common subsequence problem. In: Stützle, T., et al. (eds.) Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics. pp. 150–161. Springer (2007)
5. Dai, H., Khalil, E.B., Zhang, Y., Dilkina, B., Song, L.: Learning combinatorial optimization algorithms over graphs. In: Advances in Neural Information Processing Systems. vol. 31, pp. 6348–6358. Curran Associates, Inc. (2017)
6. Djukanovic, M., Berger, C., Raidl, G.R., Blum, C.: On solving a generalized constrained longest common subsequence problem. In: Olenev, N., et al. (eds.) Optimization and Applications, LNCS, vol. 12422, pp. 55–70. Springer (2020)
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., et al. (eds.) Proc. of the 5th Int. Conf. on Machine Learning, Optimization and Data Science. LNCS, vol. 11943, pp. 154–167. Springer (2020)
8. Ghirardi, M., Potts, C.N.: Makespan minimization for scheduling unrelated parallel machines: A recovering beam search approach. European Journal of Operational Research **165**(2), 457–467 (2005)
9. Gotthilf, Z., Hermelin, D., Lewenstein, M.: Constrained LCS: Hardness and approximation. In: Ferragina, P., et al. (eds.) Combinatorial Pattern Matching. pp. 255–262. Springer (2008)
10. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997)
11. He, H., Daumé, H.C., Eisner, J.M.: Learning to search in branch-and-bound algorithms. In: Ghahramani, Z., et al. (eds.) Advances in Neural Information Processing Systems. vol. 27. Curran Associates, Inc. (2014)

12. Huang, J., Patwary, M., Diamos, G.: Coloring big graphs with AlphaGo Zero. arXiv:1902.10162 [cs] (2019)
13. Huang, L., Zhang, H., Deng, D., Zhao, K., Liu, K., Hendrix, D.A., Mathews, D.H.: LinearFold: linear-time approximate RNA folding by 5'-to-3' dynamic programming and beam search. Bioinformatics **35**(14), i295–i304 (2019)
14. Karimi-Mamaghan, M., Mohammadi, M., Meyer, P., Karimi-Mamaghan, A.M., Talbi, E.G.: Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art. European Journal of Operational Research (2021), 10.1016/j.ejor.2021.04.032
15. Khalil, E.B., Bodic, P.L., Song, L., Nemhauser, G., Dilkina, B.: Learning to branch in mixed integer programming. In: Proc. of the 26th Int. Joint Conf. on Artificial Intelligence. pp. 724–731. AAAI Press (2016)
16. Khalil, E.B., Dilkina, B., Nemhauser, G.L., Ahmed, S., Shao, Y.: Learning to run heuristics in tree search. In: Proc. of the 26th Int. Joint Conf. on Artificial Intelligence. pp. 659–666. Melbourne, Australia (2017)
17. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: Proc. of the 3rd Int. Conf. on Learning Representations. San Diego, CA (2015)
18. Laterre, A., Fu, Y., Jabri, M.K., Cohen, A.S., Kas, D., Hajjar, K., Dahl, T.S., Kerkeni, A., Beguir, K.: Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization. In: AAAI 2019 Workshop on Reinforcement Learning on Games. AAAI Press (2018)
19. Lowerre, B.: The Harpy Speech Recognition System. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA (1976)
20. Maier, D.: The complexity of some problems on subsequences and supersequences. Journal of the ACM **25**(2), 322–336 (1978)
21. Mittal, A., Dhawan, A., Manchanda, S., Medya, S., Ranu, S., Singh, A.: Learning heuristics over large graphs via deep reinforcement learning. arXiv:1903.03332 [cs, stat] (2019)
22. Negrinho, R., Gormley, M., Gordon, G.J.: Learning beam search policies via imitation learning. In: Bengio, S., et al. (eds.) Advances in Neural Information Processing Systems. vol. 31, pp. 10652–10661. Curran Associates, Inc. (2018)
23. Ow, P.S., Morton, T.E.: Filtered beam search in scheduling. International Journal of Production Research **26**, 297–307 (1988)
24. Shyu, S.J., Tsai, C.Y.: Finding the longest common subsequence for multiple biological sequences by ant colony optimization. Computers & Operations Research **36**(1), 73–91 (2009)
25. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D.: A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science **362**(6419), 1140–1144 (2018)
26. Song, J., Lanka, R., Zhao, A., Bhatnagar, A., Yue, Y., Ono, M.: Learning to search via retrospective imitation. arXiv:1804.00846 [cs, stat] (2019)
27. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems. vol. 27. Curran Associates, Inc. (2014)
28. Tsai, Y.: The constrained longest common subsequence problem. Information Processing Letters **88**, 173–176 (2003)
29. Weiss, D., Alberti, C., Collins, M., Petrov, S.: Structured training for neural network transition-based parsing (2015)