A Policy-Based Learning Beam Search for Combinatorial Optimization*

Rupert Ettrich, Marc Huber, and Günther R. Raidl

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

Abstract. Beam search (BS) is a popular incomplete breadth-first search widely used to find near-optimal solutions to hard combinatorial optimization problems in limited time. Its central component is an evaluation function that estimates the quality of nodes encountered on each level of the search tree. While this function is usually manually crafted for a problem at hand, we propose a Policy-Based Learning Beam Search (P-LBS) that learns a policy to select the most promising nodes at each level offline on representative random problem instances in a reinforcement learning manner. In contrast to an earlier learning beam search, the policy function is realized by a neural network (NN) that is applied to all the expanded nodes at a current level together and does not rely on the prediction of actual node values. Different loss functions suggested for beam-aware training in an earlier work, but there only theoretically analyzed, are considered and evaluated in practice on the well-studied Longest Common Subsequence (LCS) problem. To keep P-LBS scalable to larger problem instances, a bootstrapping approach is further proposed for training. Results on established sets of LCS benchmark instances show that P-LBS with loss functions "upper bound" and "cost-sensitive margin beam" is able to learn suitable policies for BS such that results highly competitive to the state-of-the-art can be obtained.

Keywords: Beam Search · Machine Learning · Reinforcement Learning · Longest Common Subsequence Problem

1 Introduction

Beam search (BS) is a prominent incomplete, i.e., heuristic, graph search algorithm widely used to tackle hard planning and discrete optimization problems in limited time. Starting from a root node r, BS traverses a state graph in a breadth-first search manner but restricts the search by selecting at each level only up to β most promising nodes to pursue further and discards the others. The subset of selected nodes at the current level is referred to as the *beam*, and

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

parameter β as *beam width.* To select the β most promising nodes at each level, every node v from this level is typically evaluated by an evaluation function $f(v) = g(v) + h(v)$, where $g(v)$ represents the cost of the path from the root node r to the current node v and $h(v)$, called heuristic function, is an estimate for the cost of a best path from the current node v to a goal node. The β nodes with the best values according to this evaluation then form the beam.

Clearly, the quality of the solution BS obtains depends in general fundamentally on the evaluation function f. This function is usually developed manually for a specific problem, and its effectiveness relies on a good understanding and careful exploitation of the problem structure and possibly properties of expected instances. In practice, it is sometimes difficult to come up with an effective evaluation function that strikes the right balance between good BS guidance and reasonable computational effort.

The main contribution of this work is the investigation of a *Policy-Based Learning Beam Search* (P-LBS) that learns a *policy* for BS to select the β most promising nodes at each level of a BS, replacing the traditional approach of evaluating each node independently with the hand-crafted evaluation function f and afterwards selecting the nodes remaining on the beam based on their f-values. It builds upon our earlier Learning Beam Search (LBS) [11] framework, in which a machine learning model is used as heuristic function h as part of f and trained offline in a reinforcement learning manner on a large number of representative randomly generated problem instances to approximate specifically the expected cost-to-go from a node to a goal node. By learning a policy that is applied to all the nodes at a current level together in order to do the node selection and not insisting on approximating the actual cost-to-go, we allow now for greater flexibility and alternative modeling and training approaches.

In earlier work, Negrinho et al. [16] already described the learning of beam search policies for structured prediction problems by imitation learning and analyzed different variants from a purely theoretical perspective. In their approach, an abstract *scoring function* replaces the classical evaluation function f, which is not expected to approximate real solution costs anymore but shall just express how promising a node is in relation to the others at the current level. These scores thus induce a policy over the nodes, nodes are ranked accordingly, and the best-ranked nodes are accepted for the beam—just as in classical BS. Imitation learning is done on representative problem instances for which exact solutions, i.e., optimal paths, are assumed to be known. While Negrinho et al. [16] suggested and studied different loss functions for training with respect to theoretical convergence, no practical experiments were done.

Building on LBS, P-LBS again relies on *reinforcement learning* and does not need problem instances with known optimal solutions for training. We iteratively apply a BS with an initially randomly initialized neural network (NN) model as scoring function on many randomly generated representative problem instances. In each BS iteration, a subset of the BS tree levels is selected for generating training data. A training sample consists of all nodes encountered on a selected BS level. Two different approaches, *beam-unaware* and *beam-aware*, are investi-

gated to label training data. In *beam-unaware* training, the node in a training sample that lies on the path from the root node to the finally best solution node obtained by BS is labeled with one and all other nodes with zero. In *beam-aware* training, we perform a *nested beam search* (NBS) on each subinstance induced by each node of a training sample in order to approximate values for the true cost-to-go. Based on these values, we rank the nodes of a sample accordingly, and consider this ranking as the training target. Following Negrinho et al. [16], we consider different surrogate loss functions for the actual training.

To achieve reasonable scalability to larger problem instances, we stop the NBS executions when they reach a maximum level $d \in \mathbb{N}$ in their search trees, and evaluate the returned nodes by the so far trained NN to obtain suitable training targets for the new training data. This approach resembles a form of bootstrapping as known in reinforcement learning [22]. Produced training samples are stored in a FIFO replay buffer and used to continuously train the NN, intertwined with the P-LBS's further training data production.

While the general principle of P-LBS is quite generic, we test its effectiveness on the well-known NP-hard Longest Common Subsequence (LCS) problem [6]. Experiments show that policies trained by P-LBS are able to guide BS on established LCS benchmark instances well such that results being competitive to the state-of-the-art can be obtained.

Section 2 reviews related work. In Section 3 we present the general P-LBS framework, different loss functions, and the bootstrapping approach for speeding up the training data generation. The general NN architecture used as scoring function in P-LBS is described in Section 4. Section 5 introduces the LCS problem, its specific state graph, and the features used for the NN. Results of computational experiments are discussed in Section 6. Finally, we conclude in Section 7, where we also outline promising future work.

2 Related Work

In recent years there has been a growing interest at leveraging machine learning (ML) techniques to better solve discrete optimization problems. Under the umbrella term *learning to search* much work has been done in different directions for improving classical tree search [3]. We focus here particularly on beam search, which is a conceptually simple and classical incomplete search strategy for obtaining a heuristic solution in controlled time. It was originally introduced in the context of speech recognition [14], but since then has been widely applied to many combinatorial problems including scheduling, packing, and various string problems from bioinformatics such as the LCS problem, for which it frequently yields state-of-the-art results [6].

In the context of prediction tasks and sequence to sequence learning, BS is frequently used to derive better or feasible solutions than just by applying a simple greedy solution construction, see, e.g., [8,21]. These approaches rely on ML models that are trained independently of the BS beforehand on the basis of given labeled data, imitation learning, or occasionally reinforcement

learning. The BS is then applied as a decoder in the actual application (test time). Typically, such approaches suffer from ignoring the existence of the beam during training.

In contrast, beam-aware learning algorithms use BS at both, training and application/test time. A first approach by Collins and Roark [4] is perceptron-based and updates the parameters when the best node does not score first in the beam. On the other hand, Daumé et al. [5] described an approach that updates the parameters only when the best node falls out of the beam.

While further work on beam-aware algorithms exists in the context of prediction tasks and sequence to sequence learning, see, e.g., [23], most approaches do not expose the learned model to its own consecutive mistakes at train time: when a transition leads to a beam where the assumed best node is excluded, the algorithms either stop or reset to a beam with the best node included. To our knowledge, only Negrinho et al. [16] described an approach to learn beam search policies that addresses this issue. They formulate the task as learning a policy to traverse the combinatorial search space of beams. A scoring function is learned to match the ranking induced by given oracle costs from an assumed expert strategy. The authors proposed and analyzed several loss functions and data collection strategies that consider the beam also at train time and proved novel no-regret guarantees for learning BS policies.

In the context of classical combinatorial optimization, we are only aware of our LBS [11] sketched already in the introduction as a method where a guidance function is learned and used within a BS. This approach also exposes its learned model to its own mistakes by using the model in the BS for further training data generation and performing training in an interleaved way. However, it cannot be considered an actual beam-aware approach, as the model is specifically trained to approximate the cost to go, and the respective labels are obtained by independent NBS calls. In [12], we refined the original LBS specifically for the LCS problem by making the model independent of the number of strings and relying on a relative value function in which a cut-off is applied to the values of nodes at the same level.

LBS as well as the new P-LBS are both based on principles inspired by AlphaZero [20], although AlphaZero relies on Monte Carlo Tree Search (MCTS) and not BS. AlphaZero has proven to be very successful in the board games Go, chess, and shogi, with its predecessor AlphaGo being the first computer program that was able to beat a human Go champion. In the MCTS a neural network is used to evaluate game states and to provide a policy over possible moves. Training data is continuously produced by self-play in a reinforcement learning manner and stored in a replay buffer for training. AlphaZero has also been adapted to solve various combinatorial optimization problems like 3D packing problems [13], minimum vertex cover and maximum cut [1], or graph coloring [10].

3 Policy-Based Learning Beam Search

Solving a combinatorial optimization problem can be formulated as search in a state graph $G = (V, \mathcal{A})$ with nodes V and arcs $\mathcal{A}$. Each node $v \in V$ represents a problem-specific state, e.g., a partial assignment of values to the decision variables. Nodes $u, v \in V$ are connected by an arc $(u, v) \in \mathcal{A}$ if there is a valid problem-specific action that can be performed to transform state u into state v, for example, the assignment of a specific value to a so far unassigned decision variable of state u. Let label $\tau(u, v)$ denote this action transforming state u into state v. We assume each arc $(u, v) \in \mathcal{A}$ has associated cost $c_\mathrm{a}(u, v)$ that are induced by the action w.r.t. the objective function of the problem. State graph G has a dedicated root node $r \in V$ representing the initial state, in which typically all decision variables are unassigned. Moreover, there are one or more goal nodes $T \subset V$, which have no outgoing arcs and represent valid final states, e.g., in which all decision variables have feasible values. A complete solution is represented by a path from r to a goal node $t \in T$, referred to as r–t path, and we assume that the arc costs are defined in such a way that the objective value of the solution corresponds to the sum of the path's arc costs.

As already pointed out in the introduction, classical BS explores such a state space in an incomplete breadth-first search manner to find one or more heuristic solutions. Nodes are considered level by level, and at each level only up to β nodes are selected as beam to continue with. Now, let V_ext be the set of all nodes that have been derived as successors of the current beam. Moreover, let $f_\mathrm{s} \colon (V, 2^V) \to \mathbb{R}$ be a *scoring function* so that $f_\mathrm{s}(v, V_\mathrm{ext})$ assigns each node $v \in V$ a real-valued score *in relation to all the other nodes in* V_ext. Thus, the score of a node is not determined independently for each node but under consideration of V_ext. The score obtained by evaluating $f_\mathrm{s}(v, V_\mathrm{ext})$ for each $v \in V_\mathrm{ext}$ induces a policy over the nodes in V_ext, where higher values shall indicate a higher probability of a node leading to a best goal node. In P-LBS this scoring function f_s replaces the classical node-individual evaluation function f of BS and is realized in the form of a neural network that will be described in Section 4.

The core idea of P-LBS is to train function f_s via "self-play" similarly as in AlphaZero [20] by iterated application on many random instances generated according to the properties of the instances expected in the future application. A pseudocode of the P-LBS framework is shown in Alg. 1. It starts with a randomly initialized NN as scoring function f_s, and an initially empty replay buffer R which will contain the training data. The buffer is realized as first-in-first-out (FIFO) queue of maximum size ρ. The idea hereby is to remove older, outdated training samples when the scoring function has already been improved. After initialization, a certain number z of iterations is performed. In each iteration, a new independent random problem instance I with root node r is created and a BS with the current scoring function f_s is applied. This BS returns a best goal node t, and also the set L of node sets V_ext encountered at each level during the search. Next, from each set $V_\mathrm{ext} \in L$ a training sample is derived with probability α/L, where parameter α controls the expected number of samples produced per instance.

Algorithm 1 Policy-Based Learning Beam Search (P-LBS)

1: **Input:** nr. of iterations z, beam width β, NBS beam width β', replay buffer size ρ, min. buffer size for training γ, nr. of training samples per instance α
2: **Output:** trained scoring function f_s
3: $f_\mathrm{s} \leftarrow$ scoring function (randomly initialized NN)
4: $R \leftarrow \emptyset$ // replay buffer: FIFO of max. size ρ
5: **for** z iterations **do**
6: $\quad I, r \leftarrow$ create representative random problem instance with root node r
7: $\quad t, L \leftarrow$ BeamSearch(I, β, f_s) // best found goal node t,
8: $\quad\quad$ // set L of node sets V_ext encountered at each level
9: $\quad$ **for** $V_\mathrm{ext} \in L$ **do**
10: $\quad\quad$ **if** rand() $< \alpha/|L|$ **then** // generate training sample
11: $\quad\quad\quad$ **for** $v \in V_\mathrm{ext}$ **do**
12: $\quad\quad\quad\quad$ **if** beam-unaware **then**
13: $\quad\quad\quad\quad\quad c_v \leftarrow \begin{cases} 1, & \text{if node } v \text{ lies on } r\text{–}t \text{ path} \\ 0, & \text{otherwise} \end{cases}$
14: $\quad\quad\quad\quad$ **else if** beam-aware **then**
15: $\quad\quad\quad\quad\quad t'_v \leftarrow$ BeamSearch($I(v), \beta', f_\mathrm{s}$) // NBS call $\rightarrow$ best goal node
16: $\quad\quad\quad\quad\quad c_v \leftarrow g(t'_v)$
17: $\quad\quad\quad\quad$ **end if**
18: $\quad\quad\quad$ **end for**
19: $\quad\quad\quad$ add training sample $(V_\mathrm{ext}, \{c_v\}_{v \in V_\mathrm{ext}})$ to R
20: $\quad\quad$ **end if**
21: $\quad$ **end for**
22: $\quad$ **if** $|R| \geq \gamma$ **then**
23: $\quad\quad$ train f_s with batches of randomly sampled data from R
24: $\quad$ **end if**
25: **end for**
26: **return** f_s

For *beam-unaware* training, target values for the nodes in V_ext are derived by mapping a node $v \in V_\mathrm{ext}$ to $c_v = 1$ if node v lies on the best solution path r–t (ties are broken randomly in case there are multiple such paths with equal cost) and to $c_v = 0$ otherwise. For *beam-aware* training, each node $v \in V_\mathrm{ext}$ is mapped to a target value (i.e., approximate oracle cost) c_v obtained by performing an independent nested beam search (NBS) with beam width β' and scoring function f_s on the problem subinstance $I(v)$ induced by node v. However, as each NBS call is in general computationally expensive, we apply a bootstrapping approach (details below) to keep P-LBS scalable. All training samples derived are added to the replay buffer R.

At the end of each P-LBS iteration, if the replay buffer has reached a minimum fill level of γ, the scoring function f_s is incrementally trained with batches of data sampled uniformly at random from R using one of the following loss functions.

3.1 Loss functions.

Let $c = (c_v)_{v \in V_{\text{ext}}}$ be the vector of all target values of the nodes in V_{ext}. Moreover, given a training sample (V_{ext}, c), let $s_v = f_{\text{s}}(v, V_{\text{ext}})$ be the score obtained by evaluating our learnable scoring function f_{s} for each $v \in V_{\text{ext}}$ and $s = (s_v)_{v \in V_{\text{ext}}}$. Moreover, let $\hat{\sigma}$ be a permutation of V_{ext} that sorts the scores in s in descending order such that $s_{\hat{\sigma}(1)} \geq s_{\hat{\sigma}(2)} \geq \cdots \geq s_{\hat{\sigma}(|V_{\text{ext}}|)}$, and let σ^* be a permutation of $|V_{\text{ext}}|$ that sorts the target values in c in descending order such that $c_{\sigma^*(1)} \geq c_{\sigma^*(2)} \geq \cdots \geq c_{\sigma^*(|V_{\text{ext}})}$. We consider the following loss functions originally proposed by Negrinho et al. [16], as well as one introduced by ourselves called cost-sensitive marginal beam (cmb).

perceptron first (pf): $\ell(s, c) = \max(0, s_{\hat{\sigma}(1)} - s_{\sigma^*(1)})$
This loss is positive if the node with the highest target value does not correspond to the highest score node.

perceptron last (pl): $\ell(s, c) = \max(0, s_{\hat{\sigma}(\beta)} - s_{\sigma^*(1)})$
The loss is positive if the node with the highest target value falls out of the beam.

margin last (ml): $\ell(s, c) = \max(0, 1 + s_{\hat{\sigma}(\beta)} - s_{\sigma^*(1)})$
A penalty is given if the highest target value node is not among the β best nodes in s, but also a smaller penalty may be given if the highest target value node is placed low in the beam.

cost-sensitive margin last (cml):

$$\ell(s, c) = (c_{\sigma^*(1)} - c_{\hat{\sigma}(\beta)}) \max(0, 1 + s_{\hat{\sigma}(\beta)} - s_{\sigma^*(1)})$$

The previous ml loss is here weighted by the difference between the highest target value and the target value of the node at place β in the beam according to $\hat{\sigma}$.

cost-sensitive margin beam (cmb):

$$\ell(s, c) = \sum_{i=1}^{\beta-1} \max(0, c_{\sigma^*(i)} - c_{\hat{\sigma}(\beta)}) \max(0, 1 + s_{\hat{\sigma}(\beta)} - s_{\sigma^*(i)})$$

We suggest this additional variant of cml, in which the sum of the weighted ml losses for the first $(\beta - 1)$ elements in the beam is calculated. A penalty is given if any of the $(\beta - 1)$ first nodes in the beam according to c falls out of the beam according to s. This penalty is weighted as in the cml loss for each of the $(\beta - 1)$ first nodes in the beam.

log loss neighbors (lln): $\ell(s, c) = -s_{\sigma^*(1)} + \log \left(\sum_{i=1}^{|V_{\text{ext}}|} \exp(s_i) \right)$

Here we normalizes over all elements in V_{ext}. A higher penalty is given if there are nodes with higher scores than the score of the highest target value node.

log loss beam (llb): $\ell(s,c) = -s_{\sigma^*(1)} + \log\left(\sum_{i\in I}\exp(s_i)\right)$

Here, I denotes the index set that contains the index of the highest target value node and the indices of the β elements with the highest scores in s. This loss function is similar to the lln loss, but normalization is done only over the nodes in the beam according to the scores.

upper bound (ub): $\ell(s,c) = \max(0, \delta_{\beta+1}, \ldots, \delta_k)$

Here, $\delta_j = (c_{\sigma^*(1)} - c_{\sigma^*(j)})(s_{\sigma^*(j)} - s_{\sigma^*(1)})$ for $j = \beta+1, \ldots, |V_{\text{ext}}|$. Negrinho et al. [16] showed that this loss function is a convex upper bound for the expected beam transition cost.

Preliminary tests indicated that it is beneficial to use for the loss calculation not necessarily the beam width for which BS is intended to be finally applied, but an independent value proportional to $|V_{\text{ext}}|$. Therefore, the beam width considered in the loss functions is $\lceil |V_{\text{ext}}| \cdot \xi \rceil$, where $\xi \in (0,1]$ is a control parameter.

3.2 Bootstrapping

In beam-aware training, a training sample for a node set V_{ext} is obtained by executing NBS on each subinstance $I(v)$ induced by a node $v \in V_{\text{ext}}$. Depending on the beam width and the specific instance to be solved, these NBS executions can become computationally expensive. To keep beam-aware training scalable and reduce the computational effort, NBS executions are stopped when they reach a maximum level $d \in \mathbb{N}$. For simplicity, we assume in the following maximization and that goal nodes deeper in the search tree are always better, as it is the case in our benchmark, the LCS problem. An extension to the general case needs to consider the g-values of nodes but is otherwise straight-forward. Each depth-limited NBS call returns then either a set of nodes if level d is reached and the execution stopped or a best goal node otherwise. To determine the target costs for the nodes in V_{ext}, let $M \subseteq V_{\text{ext}}$ be the set of nodes for which the respective NBS calls finish with a goal node before or at level d, and let $N \subseteq V_{\text{ext}}$ be the set of nodes for which the NBS calls are stopped prematurely at level d. Moreover, let $\text{NBS}(I(v))$ be the set of nodes that is returned from level d for $v \in N$. Three cases are now distinguished:

1. $M = V_{\text{ext}},\ N = \emptyset$. In this case no early stopping occurred, and the target value of node $v \in M$ is set to $c_v = g(\text{NBS}(I(v)))$.
2. $M = \emptyset,\ N = V_{\text{ext}}$. Let $V'_{\text{ext}} = \{\text{argmax}_{u\in\text{NBS}(I(v))} f_{\text{s}}(u, \text{NBS}(I(v))) \mid v \in N\}$ be the set of nodes with highest f_{s}-values from each node set returned by $\text{NBS}(I(v))$. Moreover, let $v' \in V'_{\text{ext}}$ be the node that corresponds to $v \in V_{\text{ext}}$, i.e., the node $v' = \text{argmax}_{u\in\text{NBS}(I(v))} f_{\text{s}}(u, \text{NBS}(I(v)))$. The target values are then set to $c_v = f_{\text{s}}(v', V'_{\text{ext}})$.
3. $M \neq \emptyset,\ N \neq \emptyset$. Let M' be the set of goal nodes that represent the solutions returned by the NBS calls executed on subinstances $I(v)$ for $v \in M$. Set V'_{ext} is derived analogously to the previous case as the node set representing the

most promising partial solutions for the nodes in N. The target value for a node $v \in V_{\text{ext}}$ is then determined as $c_v = f_{\text{s}}(v', V'_{\text{ext}} \cup M'_v)$, where

$$v' = \begin{cases} \text{NBS}(I(v)) & \text{if } v \in M \\ \text{argmax}_{u \in \text{NBS}(I(v))} f_{\text{s}}(u, \text{NBS}(I(v))) & \text{if } v \in N \end{cases}.$$

Additional post-processing may help in problem-specific scenarios. For example, in case of uniform arc costs as in our LCS benchmark problem, the nodes in N should always be ranked higher than the nodes in M, because the respective NBS calls on the subinstances induced by nodes in M finish at an earlier level than the NBS calls on subinstances induced by nodes in N. The oracle cost corresponding to nodes in N can then simply all be increased by the same value in order to be ranked above all nodes in M while still maintaining their relative positioning among the nodes in N.

4 Neural Network Architecture

The NN used as scoring function f_{s} in P-LBS must fulfill an important property: It must be able to deal with inputs of variable size as $|V_{\text{ext}}|$ in general varies and we aim at scoring each node in dependence of all nodes in V_{ext}.

Let the input to the NN be a vector of vectors $(x_v)_{v \in V_{\text{ext}}}$, where x_v is a problem-specific feature vector representing the state associated with node v. Moreover, also $g(v)$, the cost from the r–v path, are appended as an additional feature in x_v. Figure 1 illustrates the NN architecture realizing f_{s}. The NN is a feedforward network with layers $j = 0, \ldots, 3$ described in the following. Hereby, $A^{(j)}$ denotes a weight matrix and $b^{(j)}$ a bias vector for each layer j. Weights and biases are shared within each layer among the components for the individual nodes' feature vectors.

Layer 0: The inputs x_v are first embedded by a linear transformation

$$h_v^{(0)} = A^{(0)} x_v + b^{(0)} \qquad \forall v \in V_{\text{ext}}.$$

Layer 1: The embeddings $h_v^{(0)}$ of the individual nodes are then pooled to obtain a constant-size global embedding for V_{ext}. We do this simply by averaging, i.e.,

$$h^{(1)} = \frac{1}{|V_{\text{ext}}|} \sum_{v \in V_{\text{ext}}} h_v^{(0)}.$$

Layer 2: Now, the node-individual embeddings from layer 0 are combined with the the global embedding from layer 1 by concatenation and used subsequently as inputs for a per-node linear transformation followed by a ReLU activation:

$$h_v^{(2)} = \text{ReLU}(A^{(2)}(h_v^{(0)} \mid\mid h^{(1)}) + b^{(2)}) \qquad \forall v \in V_{\text{ext}}.$$

Layer 3: A final linear transformation is used to compute the scores s_v in the form of logits

$$s_v = h_v^{(3)} = A^{(3)} h_v^{(2)} + b^{(3)} \qquad \forall v \in V_{\text{ext}}.$$

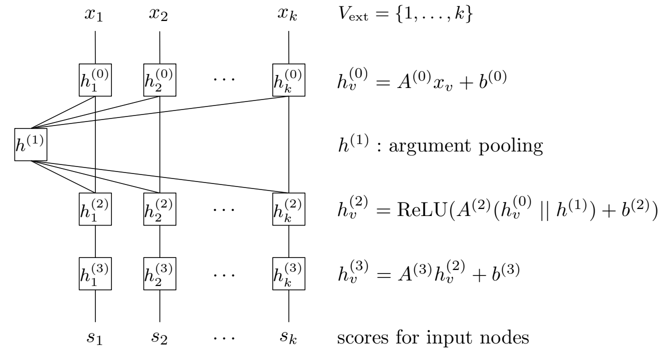

Fig. 1: Four-layer feedforward NN architecture for P-LBS.

5 Case Study: Longest Common Subsequence Problem

A string is a finite sequence of symbols taken from an alphabet Σ. A subsequence of a string is a string that is obtained by deleting zero or more symbols. A *common* subsequence of a set of strings $\mathcal{S} = \{S_1, \ldots, S_m\}$ is a string that is a subsequence of every string in $\mathcal{S}$. The longest common subsequence (LCS) problem aims at finding a common subsequence of maximum length for $\mathcal{S}$. 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 [18], database query optimization [17], and image processing [2]. For a fixed number m of strings the LCS problem is polynomially solvable by dynamic programming in time $O(n^m)$ [9], where n denotes the length of the longest input string, while for general m it is NP-hard [15]. The current state-of-the-art heuristic approaches for large m and n are based on BS with a theoretically derived function EX that approximates the expected length of the result of random strings from a partial solution [6] and also on our LBS [11].

Notations. We denote the length of a string S by $|S|$, and the maximum length of all input strings in $\mathcal{S}$ by n. The j-th letter of a string shall be $S[j]$. By $S[j, j']$ we denote the substring of S starting with $S[j]$ and ending with $S[j']$ if $j \leq j'$ or the empty string ε otherwise. As in previous works [6,11], the following data structure is prepared in preprocessing to enable an efficient "forward stepping" in the strings. For each $i = 1, \ldots, m$, $j = 1, \ldots, |S_i|$, and $a \in \Sigma$, $succ[i, j, a]$ stores the minimal position j' such that $j' \geq j \wedge S_i[j'] = a$ or 0 if letter a does not occur in S_i from position j onward.

State Graph for the LCS Problem. In the state graph $G = (V, \mathcal{A})$ for the LCS problem, a node $v \in V$ represents a state 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 remaining relevant substrings $S_i[p_i^v, |S_i|]$ for $i = 1, \ldots, m$. These substrings form the LCS subproblem instance $I(v) = S_i[p_i^v, |S_i|]$, for $i = 1, \ldots, m$ induced by node v. The root node $r \in V$ has position vector $p_i^v = (1, \ldots, 1)$. Hence, $I(v)$ corresponds to the original LCS

instance. An arc $(u, v) \in \mathcal{A}$ refers to transitioning from state u to state v by adding a valid letter $a \in \Sigma$ to a partial solution, and consequently, arc (u, v) is labeled by this letter, i.e., $\tau(u, v) = a$. Extending a partial solution at state u by letter $a \in \Sigma$ is only valid, if $succ[i, p_i^u, a] > 0$ for $i = 1, \ldots, m$ and yields state v with $p_i^v = succ[i, p_i^u, a] + 1$ for $i = 1, \ldots, m$. States that allow no feasible extension are represented by a single goal node $t \in V$ with $p_i^t = |S_i| + 1$ for $i = 1, \ldots, m$ that has no outgoing arcs. As with each arc always exactly one letter is appended to a partial solution, the cost of each arc $(u, v) \in \mathcal{A}$ is one. As the objective of the LCS problem is to find a maximum length string, $g(v)$ corresponds to the number of arcs of the longest identified r–v path.

Node features and training. As features to represent a state we only use the *remaining string lengths* $|S_i| - p_i^v + 1$, $i = 1, \ldots, m$, on which also the heuristic from [6] is based. To prevent possible difficulties in learning symmetries, the remaining string lengths are always sorted according to non-decreasing values before providing them as feature vector x_v. In the NN, the hidden vectors $h_v^{(j)}$ for $j = 0, 1$, have size ten, whereas the hidden vectors $h_v^{(2)}$ have size 20, and the weight matrices and bias vectors were dimensioned accordingly.

As in previous work [11], the ADAM optimizer with step size 0.001 and exponential momentum decay rates 0.9 and 0.999 is applied for training. In each P-LBS iteration, two mini-batches of eight random samples are selected from the replay buffer R and used for learning. The loss of a single sample is obtained by one of the loss functions from Section 3.1, and the loss of a batch is determined by the mean loss of the individual batches.

6 Experimental Evaluation

We implemented P-LBS in Julia 1.7 using Flux for the NN. All experiments were executed on an Intel Xeon E5-2640 processor with 2.40 GHz and a memory limit of 20 GB. Two in the literature commonly used benchmark sets for the LCS problem are considered to empirically analyze and evaluate P-LBS. The first set referred to as `rat` was introduced in [19], and consists of 20 single instances composed of sequences from rat genomes. Each of these instances differs in the combination of the alphabet size $|\Sigma| \in \{4, 20\}$, number of input strings $m \in \{10, 15, 20, 25, 40, 60, 80, 100, 150, 200\}$. The length of the strings is $n = 600$. The second benchmark set denoted as `ES` is from [7] and consists of 50 random instances for each combination of $|\Sigma| \in \{2, 10, 25\}$, $m \in \{10, 50, 100\}$, where $n = 1000$ for instances with $|\Sigma| \in \{2, 10\}$, and $n = 2500$ for instances with $|\Sigma| = 25$. Preliminary tests led to the following P-LBS configuration that turned out to be suitable for all our benchmark sets unless stated otherwise: nr. of P-LBS iterations $z = 2000$, P-LBS and NBS beam widths $\beta = \beta' = 50$, NBS depth limit $d = 5$, beam width parameter for loss calculation $\xi = 0.1$, max. buffer size $\rho = 500$, min. buffer size for learning $\gamma = 250$, nr. of training samples generated per instance $\alpha = 5$, ten restarts with randomly initialized NN weights and final adoption of the NN yielding the best result on 30 independent random instances.

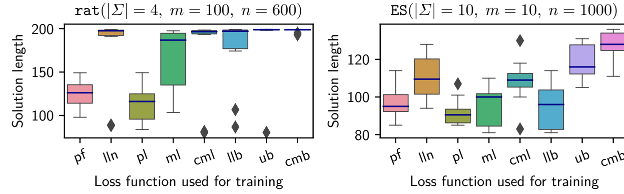

Fig. 2: Impact of the loss function in P-LBS on the solution lengths of BS on `rat` and `ES` instances.

Loss functions. One of our main goals is to analyze the impact of the different loss functions from Section 3.1 in practice, as they were so far only theoretically considered in [16]. For this purpose, ten P-LBS runs were performed for ($|\Sigma| = 4$, $m = 100$, $n = 600$) and ($|\Sigma| = 10$, $m = 10$, $n = 1000$), and the learned scoring functions were used in BS to solve the respective `rat` and `ES` instances. Figure 2 shows the obtained solution lengths for each loss function pf, lln, pl, ml, cml, llb, ub, and cmb as box plots. Loss functions pf and lln were used in the conjunction with beam-unaware training, whereas all other loss functions were used with beam-aware training. We can clearly see that loss functions pf, pl, ml, and llb perform significantly worse than cml, cmb, lln, and ub. Therefore, we use only loss functions cml, cmb, lln, and ub in the further experiments.

NBS depth limit. The choice of depth limit d in the NBS calls has a considerable impact on the runtime of P-LBS. Thus, we want to use a depth limit in the NBS calls that is as small as possible, but at the same time, large enough to produce robust models leading to high-quality predictions. In order to examine this aspect, we performed ten P-LBS runs each for different depth limits d in the NBS calls. Figure 3 shows exemplary box plots for final LCS lengths and training times on a representative `rat` instance, obtained by BS with scoring functions trained via P-LBS using the different depth limits. As one may expect, higher values for d lead to a more stable convergence of the NN, reflected by the smaller standard deviation and generally larger solution lengths seen in the left subfigure. The right subfigure shows the runtimes of P-LBS with $z = 2000$ iterations. We can see that our bootstrapping approach works well already for quite moderate depth limits $d \geq 5$ and can save much time. We therefore apply $d = 5$ in the remaining experiments.

Evaluation on benchmark instances. Finally, we evaluate our approach with each of the remaining loss function alternatives on all instances from benchmark sets `rat` and `ES` and also compare it to state-of-the-art methods from the literature. For this purpose, NNs were trained for each combination of $|\Sigma|$, m, and n occurring in the benchmark instances on random instances using P-LBS with each loss function. All training with P-LBS was done using beam width

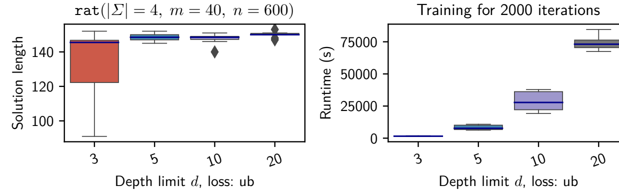

Fig. 3: Impact of depth limit d in NBS calls on the solution length of BS on a `rat` instance.

$\beta = \beta' = 50$, except for `rat` instances with $|\Sigma| = 20$, for which $\beta = \beta' = 20$ and $z = 1000$ were used due to computational budget limitations. Concerning the final testing, we followed [6] and applied BS on all benchmark instances using $\beta = 50$ to aim at low (computation) time and $\beta = 600$ to aim at high-quality solutions, respectively. Table 1 shows the obtained results. Columns $|g_{\text{lln}}|$, $|g_{\text{cml}}|$, $|g_{\text{ub}}|$, and $|g_{\text{cmb}}|$ list the average solution lengths obtained by BS with the NNs trained by P-LBS with loss functions lln, cml, ub, and cmb, respectively. Additionally, respective average solution qualities of LBS from [12] are shown in columns $|g_{\text{LBS}}|$. So far best-known average solution lengths reported in [6] are listed in columns $|g_{\text{lit-best}}|$. Average runtimes of the BS with the trained NNs (with loss function cmb) and corresponding ones from LBS are provided in columns $t_{\text{cmb}}[s]$ and $t_{\text{LBS}}[s]$.

The results show that BS with the trained NNs with loss function ub yields for both, low time and high-quality experiments, in six out of nine instance groups on benchmark set `ES` higher average solution lengths than lln, cml and cmb, while BS with the trained NNs with loss functions ub and cmb achieves on many instance groups on benchmark set `rat` higher average solution lengths than lln and cml. This coincides with our previous loss function analysis, where loss functions ub and cmb yielded higher solution lengths than lln and cml. Furthermore, the high variance in the results obtained by BS with the trained NNs with loss functions lln and cml on benchmark set `rat` indicates that these loss functions produce less robust models than ub and cmb. We conclude that loss functions ub and cmb are most suitable for training NNs to guide BS on the LCS problem. While BS with the trained NNs with loss function ub yields in six out of nine instance groups for low time and in seven out of nine for high-quality experiments higher average solution lengths compared to LBS on benchmark set `ES`, loss functions ub and cmb perform slightly worse than LBS on benchmark set `rat`.

Comparing BS with the trained NNs with loss functions ub and cmb to the so far best-known solutions, ub and cmb yield results being competitive on benchmark set `ES` but perform slightly worse on benchmark set `rat`. In total, BS with the trained NNs with loss functions lln, cml, ub and cmb could achieve in five out of 29 instance groups new best solutions for low time and in two out of 29 for high-quality. Concerning runtimes, we can conclude that they are lower than those of LBS, as more node features were used in LBS.

Table 1: LCS results on benchmark sets `rat` and `ES` obtained by BS with NNs trained by P-LBS (with loss functions lln, cml, ub, and cmb), and LBS [12]. Respective so far best-known average solution lengths reported in [6] are listed in columns $\lvert g_{\text{lit-best}}\rvert$.

				low time ($\beta = 50$)								high quality ($\beta = 600$)							
Set	$\lvert\Sigma\rvert$	m	n	$\lvert g_{\text{lln}}\rvert$	$\lvert g_{\text{cml}}\rvert$	$\lvert g_{\text{ub}}\rvert$	$\lvert g_{\text{cmb}}\rvert$	$t_{\text{cmb}}[s]$	$\lvert g_{\text{LBS}}\rvert$	$t_{\text{LBS}}[s]$	$\lvert g_{\text{lit-best}}\rvert$	$\lvert g_{\text{lln}}\rvert$	$\lvert g_{\text{cml}}\rvert$	$\lvert g_{\text{ub}}\rvert$	$\lvert g_{\text{cmb}}\rvert$	$t_{\text{cmb}}[s]$	$\lvert g_{\text{LBS}}\rvert$	$t_{\text{LBS}}[s]$	$\lvert g_{\text{lit-best}}\rvert$
`rat`	4	10	600	198.0	199.0	199.0	199.0	0.280	199.0	0.550	**201.0**	**205.0**	201.0	204.0	203.0	7.542	**205.0**	8.591	**205.0**
`rat`	4	15	600	***186.0**	178.0	182.0	182.0	0.318	184.0	0.660	184.0	183.0	182.0	184.0	184.0	7.485	**185.0**	9.097	**185.0**
`rat`	4	20	600	162.0	159.0	***170.0**	167.0	0.298	169.0	0.620	169.0	**173.0**	167.0	172.0	171.0	7.388	**173.0**	8.082	**173.0**
`rat`	4	25	600	***169.0**	167.0	166.0	166.0	0.344	166.0	0.766	167.0	169.0	167.0	170.0	170.0	8.392	**171.0**	9.295	**171.0**
`rat`	4	40	600	147.0	143.0	**152.0**	150.0	0.204	**152.0**	0.844	**152.0**	145.0	150.0	153.0	153.0	8.649	**156.0**	10.064	**156.0**
`rat`	4	60	600	145.0	144.0	147.0	149.0	0.284	149.0	0.868	**150.0**	150.0	149.0	151.0	149.0	9.027	**152.0**	12.129	**152.0**
`rat`	4	80	600	110.0	134.0	132.0	132.0	0.304	**138.0**	1.056	**138.0**	132.0	137.0	140.0	138.0	9.451	140.0	12.564	**142.0**
`rat`	4	100	600	122.0	119.0	129.0	134.0	0.508	**135.0**	0.483	**135.0**	128.0	134.0	**138.0**	131.0	9.521	137.0	13.650	**138.0**
`rat`	4	150	600	101.0	117.0	123.0	126.0	0.746	**127.0**	1.176	**127.0**	117.0	122.0	114.0	128.0	10.159	**130.0**	11.625	**130.0**
`rat`	4	200	600	105.0	104.0	115.0	115.0	1.438	121.0	1.572	**123.0**	111.0	115.0	**124.0**	118.0	10.529	**123.0**	14.117	123.0
`rat`	20	10	600	70.0	***71.0**	70.0	70.0	1.420	70.0	1.108	70.0	**71.0**	**71.0**	**71.0**	**71.0**	11.774	**71.0**	10.104	**71.0**
`rat`	20	15	600	**62.0**	61.0	**62.0**	60.0	1.788	62.0	1.117	**62.0**	62.0	62.0	**63.0**	62.0	12.457	**63.0**	12.048	**63.0**
`rat`	20	20	600	53.0	52.0	52.0	52.0	1.184	**54.0**	1.059	**54.0**	***55.0**	54.0	54.0	***55.0**	11.557	54.0	13.704	54.0
`rat`	20	25	600	50.0	50.0	**51.0**	50.0	0.983	**51.0**	1.152	**51.0**	**52.0**	**52.0**	**52.0**	51.0	7.662	**52.0**	13.073	**52.0**
`rat`	20	40	600	43.0	43.0	44.0	45.0	1.206	**49.0**	0.529	**49.0**	47.0	47.0	47.0	47.0	10.368	**49.0**	16.005	**49.0**
`rat`	20	60	600	43.0	43.0	43.0	45.0	1.948	**46.0**	1.945	**46.0**	46.0	45.0	45.0	46.0	16.698	**47.0**	19.734	**47.0**
`rat`	20	80	600	40.0	38.0	39.0	38.0	1.858	42.0	1.953	**43.0**	40.0	40.0	40.0	40.0	16.986	43.0	24.741	**44.0**
`rat`	20	100	600	37.0	37.0	36.0	**38.0**	1.711	**38.0**	2.007	**38.0**	38.0	39.0	38.0	39.0	13.746	39.0	24.441	**40.0**
`rat`	20	150	600	34.0	34.0	35.0	**37.0**	1.828	**37.0**	2.457	**37.0**	**37.0**	**37.0**	**37.0**	**37.0**	15.408	**37.0**	28.719	**37.0**
`rat`	20	200	600	33.0	33.0	33.0	33.0	1.956	**34.0**	2.048	**34.0**	**34.0**	**34.0**	**34.0**	**34.0**	18.582	**34.0**	32.118	**34.0**
`ES`	2	10	1000	608.74	605.06	608.84	604.62	0.41	606.80	0.71	**609.80**	614.42	612.18	614.68	611.64	8.107	613.35	11.248	**615.06**
`ES`	2	50	1000	518.54	529.24	531.46	529.30	0.67	529.76	1.02	**535.02**	523.38	532.96	535.76	533.24	13.046	534.29	16.684	**538.24**
`ES`	2	100	1000	503.64	508.40	511.36	508.54	1.09	514.62	1.72	**517.38**	507.96	512.34	513.94	512.32	17.777	516.85	22.103	**519.84**
`ES`	10	10	1000	198.70	198.56	198.80	199.00	0.94	198.94	1.17	**199.38**	202.60	202.42	202.42	202.60	18.081	202.10	23.249	**203.10**
`ES`	10	50	1000	134.60	133.80	134.22	133.64	1.15	134.02	1.89	**134.74**	136.28	135.64	136.12	135.58	16.080	135.56	19.720	**136.32**
`ES`	10	100	1000	119.46	120.82	120.94	120.88	1.39	121.20	1.20	**122.10**	121.26	122.12	122.38	122.14	17.258	122.67	24.349	**123.32**
`ES`	25	10	2500	***230.90**	230.49	230.49	230.76	4.62	229.39	5.15	230.28	235.38	235.28	235.69	***235.69**	76.820	235.20	77.550	235.22
`ES`	25	50	2500	136.62	136.84	137.18	136.84	8.09	133.88	7.82	**137.9**	138.56	138.92	139.04	138.66	74.902	137.44	105.956	**139.5**
`ES`	25	100	2500	119.76	120.56	120.64	120.46	10.33	119.70	16.74	**121.74**	121.22	121.92	122.12	121.94	116.728	121.71	159.843	**122.88**

7 Conclusions and Future Work

We proposed a general Policy-Based Learning Beam Search (P-LBS) framework for learning BS policies to solve combinatorial optimization problems. Instead of the traditional approach of evaluating each node independently with a hand-crafted evaluation function in BS, we learn a policy for selecting the nodes to continue with in the next BS level. Learning is performed by utilizing concepts from reinforcement learning, in particular the self-play of AlphaZero: P-LBS generates training data on its own by executing BS with the so far trained policy on many representative randomly generated problem instances. While different loss functions for learning a BS policy have been suggested but only studied from a theoretical point of view in the literature, we compare and evaluate them in the practical scenario of solving the prominent LCS problem. Reasonable scalability to larger problem instances could be achieved by utilizing bootstrapping. Our case study on the LCS demonstrates that P-LBS with loss functions ub and cmb is able to learn BS policies such that highly competitive results can be obtained.

One weakness we recognized in P-LBS using beam-unaware training is that the BS in our implementation returns exactly one best goal node and r–t path disregarding the fact that multiple best goal nodes with equal objective values and different r–t paths may exist. As a result, nodes in a training sample are labeled with zeroes, although these nodes possibly lie on a path of another best goal node. In future work, it would be promising to adapting the BS so that all found equally good goal nodes and corresponding r–t paths are considered. General improvement potential of P-LBS lies in using a more advanced graph neural network as policy to get rid of the dependency of specific instance sizes. Finally, we are interested in applying P-LBS to further problems of different nature to investigate the full potential of P-LBS.

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. Bezerra, F.: A longest common subsequence approach to detect cut and wipe video transitions. In: Proc.. 17th Brazilian Symposium on Computer Graphics and Image Processing. pp. 154–160. IEEE Press (2004)
3. Chang, K.W., Krishnamurthy, A., Agarwal, A., Daumé, H., Langford, J.: Learning to search better than your teacher. In: Proc. of the 32nd Int. Conf. on Machine Learning. vol. 37, pp. 2058–2066 (2015)
4. Collins, M., Roark, B.: Incremental parsing with the perceptron algorithm. In: Proc. of the 42nd Annual Meeting on Association for Computational Linguistics. pp. 111–es (2004)
5. Daumé, H., Marcu, D.: Learning as search optimization: approximate large margin methods for structured prediction. In: Proc. of the 22nd Int. Conf. on Machine Learning. pp. 169–176. ACM Press (2005)
6. 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)
7. Easton, T., Singireddy, A.: A large neighborhood search heuristic for the longest common subsequence problem. Journal of Heuristics **14**(3), 271–283 (2008)
8. Graves, A., Jaitly, N.: Towards end-to-end speech recognition with recurrent neural networks. In: Proc. of the 31st Int. Conf. on Machine Learning. pp. 1764–1772. PMLR (2014)
9. Gusfield, D.: Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press (1997)
10. 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)
11. Huber, M., Raidl, G.R.: Learning beam search: Utilizing machine learning to guide beam search for solving combinatorial optimization problems. In: Nicosia, G., et al. (eds.) Machine Learning, Optimization, and Data Science. LNCS, vol. 13164, pp. 283–298. Springer (2022)
12. Huber, M., Raidl, G.R.: A relative value function based learning beam search for the longest common subsequence problem. In: Moreno-Díaz, et al. (eds.) Proc. of the 18th Int. Conf. on Computer Aided Systems Theory (to appear)
13. 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)
14. Lowerre, B.T.: The harpy speech recognition system. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA (1976)
15. Maier, D.: The complexity of some problems on subsequences and supersequences. Journal of the ACM **25**(2), 322–336 (1978)
16. 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)
17. Ning, K., Ng, H.K., Leong, H.W.: Analysis of the relationships among longest common subsequences, shortest common supersequences and patterns and its application on pattern discovery in biological sequences. Int. Journal of Data Mining and Bioinformatics **5**(6), 611–625 (2011)
18. Ossman, M., Hussein, L.F.: Fast longest common subsequences for bioinformatics dynamic programming. Int. Journal of Computer Applications **975**, 8887 (2012)
19. 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)
20. 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)
21. 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)
22. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press (2018)
23. Xu, Y., Fern, A.: On learning linear ranking functions for beam search. In: Proc. of the 24th Int. Conf. on Machine Learning. pp. 1047–1054. ACM Press (2007)