A Relative Value Function Based Learning Beam Search for the Longest Common Subsequence Problem*

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 graph search algorithm frequently used to heuristically find good or near-optimal solutions to combinatorial optimization problems. Its most crucial component is a heuristic function that estimates the best achievable length-to-go from any state to a goal state. This function usually needs to be developed specificially for a problem at hand, which is a manual and time-consuming process. Building on previous work, we propose a Relative Value Function Based Learning Beam Search (RV-LBS) to automate this task at least partially by using a multilayer perceptron (MLP) as heuristic function, which is trained in a reinforcement learning manner on many randomly created problem instances. This MLP predicts the difference of the expected solution length from a given state to the expected average solution length of all states at a current BS level. To support the training of the MLP on the longest common subsequence (LCS) problem, a compact fixed-size encoding of the distribution of differences of remaining string lengths to the average string length at the current level is presented. Tests show that a MLP trained by RV-LBS on randomly created small-size problem instances is able to guide BS well also on larger established LCS benchmark instances. The obtained results are highly competitive to the state-of-the-art.

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

1 Introduction

Beam search (BS) is a prominent incomplete tree search frequently applied to find heuristic solutions to hard combinatorial optimization problems, e.g., packing [1], and string-related problems [2,4]. Starting from an initial state r, BS traverses a state graph in a breadth-first-search manner seeking a best path from r to a goal state t. To keep the computational effort within limits, BS selects

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

at each level only up to β most promising states to pursue further and discards the others. The selected subset of states is called the *beam* and parameter β the *beam width*. To do this selection, each state v obtained at a level is usually evaluated by a function $f(v) = g(v) + h(v)$, where function $g(v)$ represents the length of the path from the root state to state v and $h(v)$, called the heuristic function, is an estimate of the best achievable length-to-go from state v to the goal state. The β states with the best values according to this evaluation then form the beam.

Clearly, the quality of the solution BS obtains in general depending fundamentally on the heuristic function h. This function is typically developed in a manual, highly problem-specific way, frequently involving many computational experiments and comparisons of different options.

In a previous work [4], we presented a *Learning Beam Search* (LBS) framework to automate this task at least partially by using a machine learning (ML) model as heuristic function, which is trained offline on a large number of representative randomly generated problem instances in a reinforcement learning manner to approximate the expected length-to-go from a state to the target state. This approach was investigated on the well-known longest common subsequence (LCS) problem and a constrained variant thereof and yielded new state-of-the-art results for some benchmark instances.

The overall approach is inspired by the way Silver et al. [7] mastered chess, shogi, and Go with AlphaZero. In their approach, a neural network is trained to predict the outcome of a game state, called *value*, as well as to provide a *policy* for the next action to perform, and this network is used within a Monte Carlo Tree Search as guidance. Training samples are obtained by means of self-play. Regarding literature related to the guidance of BS by a ML model, besides our work, we are only aware of the work by Negrinho et al. [6], who examined this topic purely from a theoretical point of view.

Despite the success of our former LBS, we also recognized some weaknesses: (1) For instances with different numbers of input strings m and input string lengths n, individual ML models need to be trained. (2) Absolute values of the lengths-to-go of the candidate states in one level are actually not that relevant, but rather the differences among them as they already determine an ordering and therefore the beam selection. Note that absolute values may be relatively large in comparison to the differences, and a ML model trained to predict the absolute values may therefore make stronger errors in respect to small differences.

In this work, we address these weaknesses of the former approach regarding the LCS problem by proposing the *Relative Value Function Based Learning Beam Search* (RV-LBS). In it, a MLP is trained to predict a relative value indicating the difference of the expected solution length from a given state represented by a feature vector to the expected average solution length from all nodes at the current level. To support the training of the MLP, we also provide different features as input, among which is a compact fixed-size encoding of the distribution of differences of remaining string lengths to the average string length at the current level, making the approach in principle independent of the number of strings m.

2 Longest Common Subsequence Problem

We define a string s as a finite sequence of symbols from a finite alphabet Σ. Each string that can be obtained from s by deleting zero or more symbols from that string without changing the order of the remaining symbols is called subsequence. A *common subsequence* of a set of m non-empty strings $S = \{s_1, \ldots, s_m\}$ is a subsequence that occurs in all of these strings. The longest common subsequence (LCS) problem aims at finding a common subsequence of maximum length for S. For example, the LCS of strings AGACT, GTAAC, and GTACT is GAC. The problem is well-studied and has several applications, for example, in computational biology, where it is used to detect similarities between DNA, RNA, or protein sequences in order to derive relationships. For a fixed number m of input strings the LCS problem is polynomially solvable by dynamic programming in time $O(n^m)$ [3], where n denotes the length of the longest input string, while for general m it is NP-hard [5]. Dynamic programming becomes rapidly impractical when m grows. Therefore, many approaches have been proposed to heuristically solve the general LCS problem. 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 by Djukanovic et al. [2] and also on our LBS [4].

Notations. We denote the length of a string s by $|s|$, and the maximum input string length of a set of m non-empty strings S by n. The j-th letter of a string s is $s[j]$, where $j = 1, \ldots, |s|$. We use $s[j, j']$ to denote the substring of s starting with $s[j]$ and ending with $s[j']$ if $j \leq j'$ or the empty string ε otherwise. The number of occurrences of letter $a \in \Sigma$ in string s is $|s|_a$. To ensure an efficient "forward stepping" in the strings, we use the following data structure prepared in preprocessing. 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 a does not occur in s_i from position j onward.

3 State Graph

In the context of the LCS problem, the state graph is a directed acyclic graph $G = (V, A)$ with nodes V and arcs A. Each state (node) $v \in V$ 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 remaining relevant substrings $s_i[p_i^v, |s_i|],\ i = 1, \ldots, m$ of the input strings. These substrings form a LCS subproblem instance $I(v)$ induced by state v. The root node $r \in V$ has position vector $p^r = (1, \ldots, 1)$, and thus, strings $s_i[p^r, |s_i|] = s_i,\ i = 1, \ldots, m$. 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., $\ell(u, v) = a$. Appending letter $a \in \Sigma$ to a partial solution at state u is only feasible if $succ[i, p_i^v, a] > 0$ for $i = 1, \ldots, m$, and yields in this case state v with $p_i^v = succ[i, p_i^v, a] + 1,\ i = 1, \ldots, m$. As with each arc always exactly one letter is appended to a partial solution, the length (cost) of each arc $(u, v) \in A$ is one. States for which no feasible letter exist that can

be appended to a partial solution are jointly represented by the single terminal state $t \in V$ with $p^t = (|s_i| + 1)_{i=1,\ldots,m}$. 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. Filtering and dominance checks are applied in our BS exactly as explained in [2].

4 Relative Value Function Based Learning Beam Search

As RV-LBS is build upon LBS and the main LBS procedure is similar, we first describe the LBS.

The main LBS procedure starts with a randomly initialized MLP (for structure details see [4]), and an initially empty replay buffer R, which is realized as a first-in first-out (FIFO) queue of maximum size ρ and will contain the training data. The input provided to the MLP is a *feature vector* composed of the *remaining string lengths* $q_i^v = |s_i| - p_i^v + 1$, $i = 1, \ldots, m$, sorted according to non-decreasing values to reduce symmetries, and the *minimum letter appearances* $o_a^v = \min_{i=1,\ldots,m} |s_i[p_i^v, |s_i|]|_a$, $a \in \Sigma$ derived from a given node v. After initialization, a certain number z of iterations is performed. In each iteration, a new independent random problem instance is created and a BS with training data generation is applied. If the buffer R contains at least γ samples, the heuristic function h represented by the MLP is (re-)trained with random mini-batches from R.

The BS framework with training data generation receives as input parameters a problem instance I, heuristic function h, beam width β, and the replay buffer R to which new samples will be added. Initially, the beam B contains just the root state r created for the problem instance I. An outer while-loop performs the BS level by level until the beam B becomes empty. In each iteration, each node in the beam B is expanded by considering all feasible letters for the states the nodes represent and the set of successor nodes V_{ext} is created. Dominance checks and filtering are applied to reduce V_{ext} to only meaningful nodes. From each node in $v \in V_{\text{ext}}$ a training sample is produced with a certain small probability. To obtain a training sample, the subproblem instance $I(v)$ to which state v corresponds is determined, and an independent *Nested Beam Search* (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.

Although the MLP in the above-mentioned LBS approach approximates the expected LCS length from some nodes relatively well, it may also make stronger errors with respect to small differences among the nodes in V_{ext} to be evaluated, as the absolute values may be relatively large in comparison to the differences. We address this issue with our RV-LBS by re-defining the approximation goal of the heuristic function $h(v)$ as the difference of the expected solution length from a given node represented by a feature vector to the expected average solution

length from all successor nodes of the current beam. Formally:

$$h(v) \approx \mathrm{LCS}_{\exp}(v) - \frac{1}{|V_{\mathrm{ext}}|} \sum_{v' \in V_{\mathrm{ext}}} \mathrm{LCS}_{\exp}(v'), \tag{1}$$

where $\mathrm{LCS}_{\exp}(v)$ denotes the expected solution length from node v represented by its feature vector. By evaluating solutions in relation to other states at a current BS level we expect to obtain a more precise differentiation ranking of the states.

For obtaining training samples, we utilize NBS again to get a reasonable approximation of $\mathrm{LCS}_{\exp}(v)$ and therefore target values

$$t'_v = \mathrm{NBS}_g(v) - \frac{1}{|V_{\mathrm{ext}}|} \sum_{v' \in V_{\mathrm{ext}}} \mathrm{NBS}_g(v'), \tag{2}$$

where $\mathrm{NBS}_g(v)$ corresponds to the length-to-go approximation obtained from NBS for the subproblem instance induced by node v. To reduce the computational effort, we select only one level of each BS run uniformly at random, from which training samples are created for all nodes in V_{ext}.

To remain consistentce to the approximation goal as well as to get rid of the absolute remaining string lengths we perform an input *feature encoding* at each BS level by exploiting the following observation:

> If all remaining string lengths $q_i^v,\ i = 1, \ldots, m$ of the nodes $v \in V_{\mathrm{ext}}$ are large, the reduction of all q_i^v by the same cut-off value $b \geq 0$ does usually not make a significant difference for the choice of which nodes should be further pursued in the BS and consequently also our approximation goal.

For small string lengths, however, it may still be important to consider absolute lengths. We therefore calculate this cut-off value in dependence of $\overline{q_{\mathrm{ext}}}$ and a parameter λ as $b^{\mathrm{sl}} = \max(0, \overline{q_{\mathrm{ext}}} - \lambda|\Sigma|)$, where

$$\overline{q_{\mathrm{ext}}} = \frac{1}{m\,|V_{\mathrm{ext}}|} \sum_{i=1}^{m} \sum_{v \in V_{\mathrm{ext}}} q_i^v \tag{3}$$

is the average length of all remaining input strings for all nodes in V_{ext}.

Similar to the way for determining the cut-off value for the remaining string lengths, we calculate a cut-off value for the minimum numbers of letter occurrences. Let

$$\overline{o_{\mathrm{ext}}} = \frac{1}{|\Sigma|\,|V_{\mathrm{ext}}|} \sum_{a \in \Sigma} \sum_{v \in V_{\mathrm{ext}}} o_a^v \tag{4}$$

be the average number of minimum letter occurrences for all nodes in V_{ext} and all letters. The cut-off value in dependence of $\overline{o_{\mathrm{ext}}}$ and the parameter λ is determined by $b^{\mathrm{mlo}} = \max(0, \overline{o_{\mathrm{ext}}} - \lambda)$. The ultimate input features we provide to the MLP model for calculating $h(v)$ are:

$$(i)\ \hat{q}_i^v = q_i^v - b^{\mathrm{sl}}, \quad \forall v \in V_{\mathrm{ext}},\ i = 1, \ldots, m, \quad (ii)\ b^{\mathrm{sl}},$$
$$(iii)\ \hat{o}_a^v = o_a^v - b^{\mathrm{mlo}}, \quad \forall v \in V_{\mathrm{ext}},\ a \in \Sigma, \qquad (iv)\ b^{\mathrm{mlo}}, \quad (v)\ m.$$

Note that each cut remaining string length vector $\hat{q}_i^v$ in (i) is sorted according to non-decreasing values to reduce symmetries.

Downsampling. In order to enable RV-LBS to deal with different numbers of input strings, we compress the information of a remaining string lengths vector $q = (q_1, \ldots, q_m)$ into a smaller vector of constant size $m' < m$. As the smallest and thus the first values have higher impact and q_1 in particular also represents an upper bound on the length of the LCS on its own we directly keep $q_1, \ldots, q_k$ as features $q'_1, \ldots, q'_k$, for a small k. In our experiments, $k = 3$ turned out to be a reasonable choice. The remaining $q_{k+1}, \ldots, q_m$ are sampled down into $m' - k$ values $q'_{k+1}, \ldots, m'$. For this purpose, the original values are binned into $m' - k$ bins, rounding to the respective bin is done by the nearest integer, and the arithmetic mean values of each bin is determined.

A pseudocode for the BS with training data generation for RV-LBS is shown in Algorithm 1. Remember that this BS is called by the main (RV-)LBS procedure for many random instances. As the framework structure for generating training samples is very similar to that of LBS, we only describe the differences. In line 8 each node $v \in V_{\text{ext}}$ is augmented with the set of features obtained by applying the feature encoding- and downsampling approach with parameters λ and m' to each node v in relation to V_{ext}. If the buffer is provided and the current level is determined to create training samples, then target values for each node $v \in V_{\text{ext}}$ are calculated by Equation (2) and added together with the corresponding nodes v as training samples to the replay buffer.

Algorithm 1 Beam Search with optional training data generation for RV-LBS

1: **Input:** problem inst. I, heuristic function h, beam width β
2: only for training data generation: replay buffer R
3: **Output:** best found target node t
4: $B \leftarrow \{r\}$ with r being a root node for problem instance I
5: $t \leftarrow$ none // so far best target node
6: **while** $B \neq \emptyset$ **do**
7: $V_{\text{ext}} \leftarrow$ expand all nodes $v \in B$ by considering all valid letters
8: augment each $v \in V_{\text{ext}}$ with set features determined from v in relation to V_{ext}
9: update t if a terminal node v with a new largest $g(v)$ value is reached
10: **if** R given $\wedge$ level selected **then** // generate training samples?
11: **for** $v \in V_{\text{ext}}$ **do**
12: $t'_v \leftarrow \text{NBS}_g(v)$ // NBS call
13: **end for**
14: $\overline{t'} \leftarrow \sum_{v \in V_{\text{ext}}} t'_v / |V_{\text{ext}}|$
15: **for** $v \in V_{\text{ext}}$ **do**
16: add training sample $(v, t'_v - \overline{t'})$ to R
17: **end for**
18: **end if**
19: $B \leftarrow$ select (up to) β nodes with largest f-values from V_{exp}
20: **end while**
21: **return** t

5 Experimental Evaluation

The RV-LBS algorithm for the LCS problem was implemented in Julia 1.7 using the Flux package for the MLP. All tests were performed on a cluster of machines with AMD EPYC 7402 processor with 2.80 GHz in single-threaded mode with a memory limit of 32 GB per run. We applied the proposed algorithm on the benchmark set `virus`, which was already used in [2]. The set consists of single instances with number of input strings $m \in \{10, 15, 20, 25, 40, 60, 80, 100, 150, 200\}$ and the length of the input strings $n = 600$. The alphabet size $|\Sigma| = 4$ for all instances.

Preliminary tests of RV-LBS on randomly created problem instances of size $m \in \{10, 100\}$, $n = 100$, $|\Sigma| = 4$ led to the following configuration: no. of LBS iterations $z = 500000$, min. buffer size for learning $\gamma = 45000$, beam width $\beta = 50$, max. buffer size $\rho = 50000$, cut-off parameter $\lambda = 1$, downsampling parameter $m' = 7$. Figure 1 shows on the left-hand side the impact of the downsampling parameter m' on the LCS solution length and the right-hand side illustrates exemplary box plots for final LCS lengths obtained with different values for the cut-off parameter λ. LBS is used as baseline.

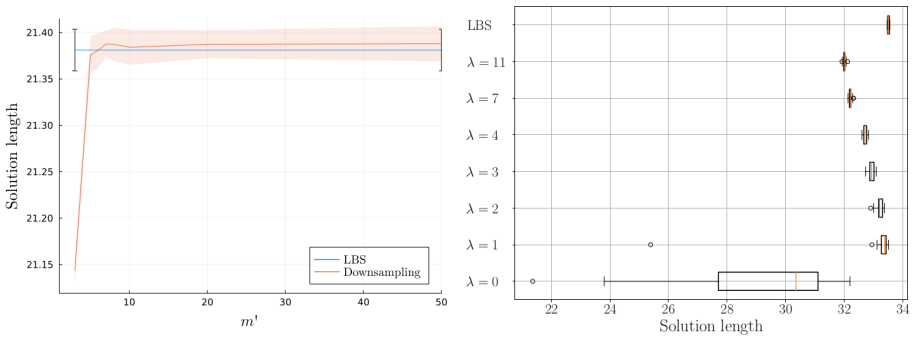

Fig. 1: Parameter calibration on randomly generated problem instances of size $n = 100$, $|\Sigma| = 4$, with $m = 100$ in the left and $m = 10$ in the right figure.

Finally, we trained 30 MLPs with RV-LBS, each on randomly generated problem instances of size $m = 10$, $n = 100$, $|\Sigma| = 4$, and used the best performing one thereafter in BS for solving the instances in the benchmark set `virus`. While all training with RV-LBS was done with $\beta = 50$, we followed [2] regarding test settings and applied BS on all benachmark instances with $\beta = 50$ to aim at low (computation) time and $\beta = 600$ to aim at high-quality solutions. Table 1 shows the obtained results. Column $|s_{\mathrm{RV-LBS}}|$ and $t_{\mathrm{RV-LBS}}$ present the respective solution qualities and runtimes obtained by a BS with the trained MLP and column $|s_{\mathrm{lit-best}}|$ and $t_{\mathrm{lit-best}}$ those from the literature [2]. Although only small-size instances were used for the training of the MLP, a BS with the trained MLP yields competitive results on the benchmark instances to the state-of-the-art.

Table 1: LCS results on benchmark set `virus`.

				low times ($\beta = 50$)				high-quality ($\beta = 600$)			
Set	$\|\Sigma\|$	m	n	$\|s_{\text{RV-LBS}}\|$	$t_{\text{RV-LBS}}$ [s]	$\|s_{\text{lit-best}}\|$	$t_{\text{lit-best}}$ [s]	$\|s_{\text{RV-LBS}}\|$	$t_{\text{RV-LBS}}$ [s]	$\|s_{\text{lit-best}}\|$	$t_{\text{lit-best}}$ [s]
virus	4	10	600	222.0	0.19	225.0	0.04	222.0	3.60	227.0	2.88
virus	4	15	600	194.0	0.33	201.0	0.23	200.0	3.58	205.0	2.24
virus	4	20	600	184.0	0.20	188.0	0.18	189.0	3.50	192.0	2.69
virus	4	25	600	191.0	0.25	191.0	0.06	194.0	4.08	194.0	2.20
virus	4	40	600	167.0	0.26	167.0	0.17	169.0	4.08	170.0	2.24
virus	4	60	600	161.0	0.30	163.0	0.04	163.0	4.15	166.0	2.38
virus	4	80	600	156.0	0.32	158.0	0.19	160.0	4.59	163.0	2.70
virus	4	100	600	153.0	0.39	156.0	0.19	156.0	5.48	158.0	0.90
virus	4	150	600	152.0	0.50	154.0	0.06	155.0	6.90	156.0	0.66
virus	4	200	600	151.0	0.63	153.0	0.09	153.0	7.90	155.0	1.22

6 Conclusions and Future Work

We presented a RV-LBS framework in which a MLP was trained to predict a relative value of a state and used this model thereafter in a BS. Moreover, we provided new features as input to the MLP to get rid of the absolute number of input strings and the string lengths. Training was done in a reinforcement learning manner by performing many BS runs on randomly created instances and calling a nested beam search to approximate the expected LCS length from a node. Although a MLP trained by RV-LBS on small-size instances was able to guide BS well on larger-size benchmark instances, we observed that the distribution of the encoded remaining string lengths at different BS levels obtained from larger-size instances has a larger standard deviation than smaller ones. Further normalization of features could be a promising direction.

References

1. Akeba, H., Hifib, M., Mhallah, R.: A beam search algorithm for the circular packing problem. Computers & Operations Research **36**(5), 1513–1528 (2009)
2. 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)
3. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997)
4. 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 International Publishing (2022)
5. Maier, D.: The complexity of some problems on subsequences and supersequences. Journal of the ACM **25**(2), 322–336 (1978)
6. 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)
7. 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)