Learning To Estimate Search Progress Using Sequence Of States

#7

Abstract

Many problems of interest can be solved using heuristic search algorithms. When solving a heuristic search problem, we are often interested in estimating search progress, that is, how much longer until we have a solution. Previous work on search progress estimation derived formulas based on some relevant features that can be observed from the behavior of the search algorithm. In this paper, rather than manually deriving such formulas we leverage machine learning to automatically learn more accurate search progress predictors. We train a Long Short-Term Memory (LSTM) network, which takes as input sequences of states expanded by the search algorithm, and predicts how far along the search we are. Importantly, our approach still treats the search algorithm as a black box, and does not look into the contents of search states. An empirical evaluation shows our technique outperforms previous search progress estimation techniques.

Introduction

Many interesting and challenging real-world problems can be defined as search problems, where the objective is to find a path in a state space (which is often implicitly defined by a black-box successor generator) from some given initial state to a goal state. Often, we can use a heuristic function which estimates the distance from any given state to the goal to better guide this search. Many heuristic search algorithms have been developed to solve such problems, the most well-known of which is probably A^* (Hart, Nilsson, and Raphael 1968).

Search algorithms can have unpredictable behavior, as we typically do not know the topology of the search space (if we did, perhaps we would not need to use a search algorithm). While we do have some results when the search problem is described symbolically (Hoffmann 2011), these results do not apply for black-box search problem. Other work has addressed estimating search effort in terms of number of expanded nodes (Lelis, Stern, and Sturtevant 2014; Belov et al. 2017) or the cost of an optimal solution (Lelis et al. 2014) – we discuss these and others in more detail later. Importantly, these approaches are all *offline* – they give their predictions before search starts.

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

We are interested in predicting remaining search effort *online*, that is, in estimating how much progress the search algorithm has already made. This problem was first introduced by Thayer, Stern, and Lelis (2012), who also derived some formulas for estimating search progress based on relatively simple search features, which we explain in more detail later.

Search progress estimation is a useful tool not just for letting a user know whether they have time to make a cup of coffee before search completes, but also in making decisions about the search. For example, remaining search time estimates are used in situated temporal planning (Cashmore et al. 2018) to prioritize search nodes which have a higher probability to lead to a timely solution. Such estimates can also be used to decide when to perform a restart in a Branch-and-Bound search (Anderson et al. 2019) for solving a Mixed Integer Program.

In this paper, we leverage recent developments in machine learning to automatically learn more complex search predictors based on a given training set. Specifically, we view the search algorithm as box that emits a sequence of states (we treat the state which the search algorithm expands as the state being emitted), and train a Long Short-Term Memory (LSTM) network (Hochreiter 1998) to predict remaining search effort.

We conducted an empirical evaluation on a large set of planning domains, comparing our technique to previous work (Thayer, Stern, and Lelis 2012). The results indicate that our technique achieves state-of-the-art performance in search progress estimation.

Background

In this section we review some of the necessary background on LSTM networks and heuristic search.

LSTM Networks

Long Short-Term Memory (LSTM) networks (Hochreiter and Schmidhuber 1997) are a specific type of Recurrent Neural Network (Medsker and Jain 2001) – a deep learning approach which is especially suitable for learning from sequences. LSTMs were first introduced in order to solve the vanishing gradient problem in conventional Recurrent Neural Networks (Hochreiter 1998), and have been extensively used in natural language processing, e.g., (Sutskever, Vinyals, and Le 2014; Gers, Schmidhuber, and Cummins

2000). We now briefly review how LSTM networks are built, but refer the interested reader to a detailed survey (Yu et al. 2019) for more details.

An LSTM network consists of several LSTM cells, arranged in sequence. Each LSTM cell has a hidden state, and 3 gates: input, forget and output, that manage the reading, writing and memory update, respectively. The cell learns gate weights and uses them to change the input and the hidden state based on the previous input in the sequence.

The cell processes input x_t together with the hidden state h_{t-1} (which is the result of processing the previous input) by first passing them to the forget gate, which has a Sigmoid activation function. The output of the forget gate is a number between 0 and 1 for each element in the cell state C_t – the information we are passing to the next LSTM cell.

Next, the processing continues through the input gate, which has 2 activation functions, Sigmoid and tanh. The input is the same as the forget layer x_t and h_{t-1}. The Sigmoid activation decides which values to update and the tanh activation creates the values themselves. The outputs from both of these are multiplied.

Finally, processing continues to the output gate that gets x_t and h_{t-1} as inputs and goes through a Sigmoid activation which decides what parts will be output. The output also goes through a tanh activation, which is multiplied by the output of the Sigmoid activation.

This is all captured by the following equations, which describe the operation of each LSTM cell:

- forget gate: $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
- input gate: $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
- new cell state value: $\widetilde{C_T} = tanh(W_c \cdot [h_{t-1}, x_t] + b_C)$
- update cell state: $C_T = f_t * C_{t-1} + i_t * \widetilde{C_T}$
- output gate: $o_t = \sigma(W_0[h_{t-1}, x_t] + b_0)$
- hidden state: $h_t = o_t * tanh(C_t)$

Heuristic Search

As previously mentioned, many problems of interest can be modeled as state-space search problems (Bonet and Geffner 2001). A state space is formally represented as a tuple $T = \langle S, S_0, S_G, A, f, c\rangle$, where:

- S is a finite and non-empty set of states,
- $S_0 \in S$ is the initial state,
- $S_G \subseteq S$ is a non-empty set of goal states,
- $A(s) \subseteq A$ denotes the actions applicable in each state $s \in S$,
- $f : S \times A \rightarrow S$ is the state transition function, such that applying action a in state s leads to state $f(s, a)$, and
- $c(s, a)$ is the cost of performing action a in state s

Uninformed search algorithms such as Breadth-First Search (BFS) can be used to solve small state-space search problems. However, to solve larger problems, we typically also use a heuristic which estimates the distance from any given state to the goal. Formally, a heuristic is a function mapping the states S to some real number (or infinity to indicate a dead-end), that is $h : S \rightarrow \mathbb{R}_+ \cup \{\infty\}$.

Most search algorithms define some priority function $f(s)$ over the states. Typically, $f(s)$ is obtained from $h(s)$, the heuristic estimate of the distance to go from s, and $g(s)$, the cost of the best known path to s so far. For example, A* (Hart, Nilsson, and Raphael 1968) expands the node with the lowest $f_{A^*}(s) = g(s) + h(s)$, while Greedy Best First Search (GBFS) (Doran and Michie 1966) expands the node with the lowest $f_{GBFS}(s) = h(s)$. We will use f, g and h as features for each search node.

Search Progress Estimation

We can now formally describe the problem we address in this paper – search progress estimation. Our definition here is adapted from previous work (Thayer, Stern, and Lelis 2012). Let $\mathbb{A}$ be a search algorithm, and $P = \langle T, h\rangle$ be a heuristic search problem, which consists of a search space T and a heuristic function h. Denote by $\mathrm{E}_{\mathbb{A}}(P)$ the number of nodes expanded by search algorithm $\mathbb{A}$ while attempting to solve a search problem P. Let $\mathrm{Rem}_{\mathbb{A}}(P, \mathrm{Exp})$ be the number of remaining nodes that are going to be expanded by $\mathbb{A}$ when solving P after $\mathbb{A}$ has already expanding Exp nodes, this is $\mathrm{Rem}_{\mathbb{A}}(P, \mathrm{Exp}) = \mathrm{E}_{\mathbb{A}}(P) - \mathrm{Exp}$.

Definition 1 (Search Progress)**.** *The search progress of algorithm $\mathbb{A}$ solving problem P after expending Exp nodes is:*

$$Prog_{\mathbb{A}}(Exp) = \frac{Exp}{Exp + Rem_{\mathbb{A}}(Exp)}$$

As Definition 1 shows, search progress is a number between 0 and 1, which indicates what fraction of the total search effort to solve the problem we have already expended. For example if we expanded node number 1,000 out of 10,000 nodes that are going to be expanded in total, the search progress will be $1,000/10,000 = 0.1$. This allows comparing search progress between problems of different sizes on the same scale.

Learning to Predict Search Progress

We now describe our technique for search progress estimation, which is based on machine learning. We begin by defining the input features to our predictor. First, note that any expansion-based forward search algorithm works by expanding nodes in sequence. Thus, we can view the search algorithm as a black-box that outputs a sequence of nodes to expand. This motivates us to use a Recurrent Neural Network (as we discuss next), looking at the last k nodes that were expanded. However, for each of these nodes, it is also important to know something about where it came from – the path from the initial state to that node. Thus, for each of these k nodes, we also look at their ancestors up m layers up. In this paper, we used $k = 30$ and $m = 2$, resulting in collecting features from $k \cdot (m+1) = 90$ nodes, as illustrated in Figure 1.

For each of these $k \cdot (m+1)$ nodes, we collect the following features:

- $g(n)$ – the cost of the best known path to node n.

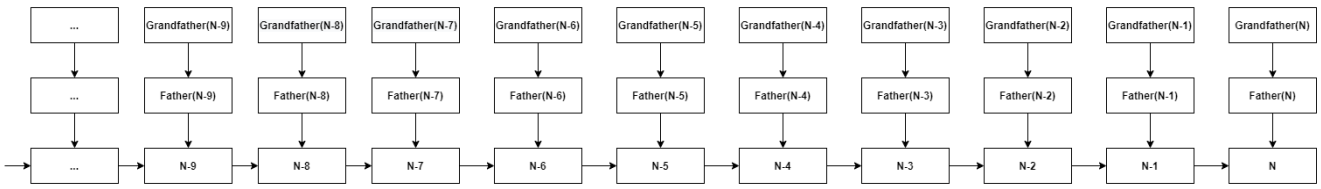

Figure 1: Illustration of the Nodes Used as Input for Prediction.
N is the node that was expanded last, $N-1$ the node expanded before it, and so on. Father() and Grandfather() are obtained by following the parent pointers for the indicated node.

- $h(n)$ – the heuristic estimate of the cost from n to the goal.
- $f(n)$ – the f-value for n the search algorithm uses.
- $b(n)$ – the branching factor of n, that is, the number of successors node n has.
- $N(n)$ – the serial number of n, that is, how many nodes were expanded before n. We remark that this number was previously used to measure the expansion delay (Dionne, Thayer, and Ruml 2011).

Additionally, we use some global features, which capture a snapshot of the current state of the search:

- $h_0 = h(S_0)$ – the heuristic value of the initial state. This value stays constant throughout the search, but is important for estimating the cost of the solution that will be returned.
- $h_{\min}$ – the minimal h-value we have seen so far among the expanded nodes.
- $Nh_{\min}$ – the number of nodes we expanded since the last time $h_{\min}$ was updated.
- $f_{\max}$ – the maximum f value we have seen so far.

We record the value of these features at the time each node was expanded, so we have k values for each of these features. These input features can be seen as a matrix of real numbers, of dimension $5k(m+1)+4k$, which are used as the input (features) to our network. Note that for the first k states, we use feature values of 0 for the undefined nodes (i.e., nodes with a negative serial number).

Network Structure: Our network[1] consists of 15 LSTM layers, followed by a fully connected layer which reduces the dimension by half (from 450 to 225), then a dropout layer with 50% dropout and Rectified Linear Units (ReLU) activations, and finally another fully connected layer which outputs our prediction – a single number. The parameters here (number of LSTM layers, dimensions, dropout rate, etc.) were determined by manual tuning over preliminary data. Automatically setting these parameters is future work.

[1]The code will be made publicly available upon publication of this paper.

Training: During training, we sample 1,000 nodes uniformly at random from each problem in our dataset, in order to avoid bias to larger problems. We use the node serial number to obtain the true label of search progress – for node n the true label is $N(n)/N(g_n)$, where g_n is the goal node found by the search algorithm. We used Adam as an optimiser with a learning rate of 0.001 and a batch size of 1024, and Mean Squared Error (MSE) as the loss function.

Related Work

Having described our proposed technique, we can now compare it to some related work. Some previous work attempted to predict search effort in specific domains, e.g., predicting the number of nodes expanded by A* for the 15-puzzle (Breyer and Korf 2008). Other work used sampling to estimate search effort (Lelis et al. 2014; Belov et al. 2017; Hutter et al. 2014) or the cost of a solution (Lelis, Stern, and Sturtevant 2014) in a more general setting. However, as previously mentioned, these are all *offline* methods, which attempt to make a prediction before search starts.

We now review in more detail the only prior work we are aware of that addresses *online* search progress estimation (Thayer, Stern, and Lelis 2012). Several search progress estimators were proposed, which we now describe, and later use as baselines in our empirical evaluation:

Velocity-Based Search Speed Estimator (VeSP) The Velocity-Based Search Speed Estimator (VeSP) estimator calculates the velocity during the search by the average velocity as:

$$V = \frac{h_0 - h_{\min}}{\text{Exp}}$$

It then uses the velocity V and the current $h_{\min}$ to estimate the remaining search effort (number of expanded nodes) SE_V as follows:

$$SE_V = \frac{h_{\min}}{V}$$

Finally, it uses this estimate to predict search progress by using:

$$VeSP(\text{Exp}) = \frac{\text{Exp}}{\text{Exp} + SE_V}$$

Vacillation-Based Search Speed Estimator (VaSP) The Vacillation-Based Search Speed Estimator (VaSP) differs

from VeSP by using node serial numbers to estimate the *expansion delay* (Dionne, Thayer, and Ruml 2011), that is, the average number of expansions between when a node is generated and when it is expanded, denoted $\overline{\Delta e}$.

The expansion delay is combined with $h_{\min}$ to predict remaining search effort as:

$$SE_e = \overline{\Delta e} \cdot h_{\min}$$

As in VeSP, the estimate of search progress is:

$$VaSP(\text{Exp}) = \frac{\text{Exp}}{\text{Exp} + SE_e}$$

Path-Based Progress Estimator (PBP) We first describe the Naive PBP (NPBP), which estimates search progress at a given node n by looking at the ratio between the cost until the current node ($g(n)$) and the total estimated cost ($g(n) + h(n)$)

$$NPBP(Exp) = \frac{g(n)}{g(n) + h(n)}$$

The main problem with NPBP is that it depends on the current node, but in many cases there are other open nodes with higher progress. Thus the PBP estimator returns the maximum progress achieved among expanded nodes so far.

The fPBP estimator attempts to address this shortcoming by using an accurate estimate of the cost of an optimal solution, denoted OPT. It uses $f_{\min}$, the minimal f-value currently on the open list to estimate search progress as:

$$fPBP(Exp) = \frac{f_{\min} - h_0}{OPT - h_0}$$

In this paper, as we do not assume access to an accurate estimate of OPT, we compare only to PBP.

Distribution-Based Progress Estimator (DBP) The Distribution-Based Progress Estimator (DBP) estimates the search progress using data observed during search. It relies on counting how many nodes are expanded by the search for each value of $d(n)$ – an estimate of remaining plan *length*. We remark that in unit-cost domains $d(n) = h(n)$, and where action costs are non-uniform, $d(n)$ can usually be estimated alongside $h(n)$. Define $c[d_i]$ as the count of how many nodes had value $d(n) = d_i$ during search. DBP fits a second degree polynomial to c, in order to estimate $\hat{c}$. The DBP estimate is then defined as:

$$Prog^*(\text{Exp}) = \frac{\text{Exp}}{\sum_{i=1}^{m} \hat{c}[d_i]}$$

where m is the highest d-value with a count higher than 0.

Empirical Evaluation

Having described our technique and the previous work on search progress estimation, we can now compare them empirically. We now describe this empirical evaluation.

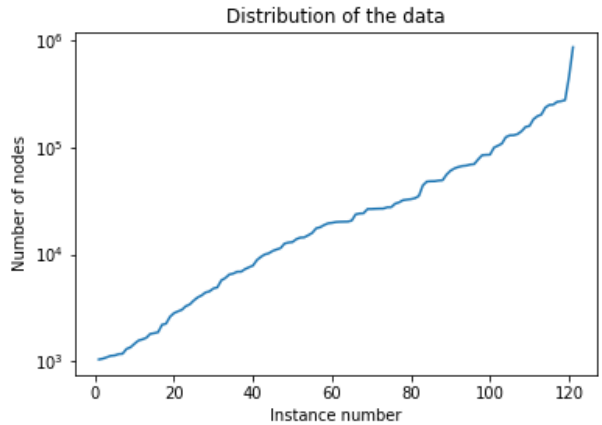

Figure 2: Distribution of the data

Benchmarks

In order to compare search progress estimates on various problems with different heuristics and different search algorithms, we chose to use planning benchmarks from past International Planning Competitions, as this allows us to implement our technique only once. Specifically, we extended Pyperplan (Alkhazraji et al. 2020) to output the features of the nodes it expands during search. This data is then used for training and testing of search progress estimators, as we describe next.

Our dataset started with planning problems from 21 IPC domains – specifically, the benchmarks which are part of the Pyperplan repository. These domains have a total of 605 planning problem instances. For each instance, we ran Pyperplan using A* and the LMCUT heuristic (Helmert and Domshlak 2009) with a time limit of 24 hours and a memory limit of 1,000,000 expanded nodes. This resulted in 331 solved instances. Of those, we omitted the instances which were solved using less than 1,000 expanded nodes, as these are solved so quickly that search progress estimation is useless, and it would have made sampling 1,000 nodes from each instance difficult. This left us with 121 solved planning problems from 19 domains, which we used as our final dataset.

Figure 2 shows the distribution of the number of expended nodes for each instance. We can see that 43 of the instances had between 1,000 to 10,000 expanded nodes, 58 had between 10,000 to 100,000 nodes, and 20 had between 100,000 to 1,000,000 nodes.

Experimental Setup

We compare our technique to the techniques introduced by previous work (Thayer, Stern, and Lelis 2012), which we described in detail above: VaSP, VeSP, PBP, and DBP. For VaSP, we used a moving average over the last 200 nodes to estimate expansion delay. We did not compare to fPBP, as we did not have a good estimate of the optimal solution cost, and so used the standard PBP estimator. For DBP we used NumPy to fit the polynomial to the data at hand.

All of our experiments were run on a server with 72 Intel Xeon E5-2695 CPUs (utilizing at most 18 processes in parallel). The deep learning was implemented in PyTorch, and run on 2 Nvidia Tesla M60 GPUs.

As our technique is based on learning from a training set, we examine three different training/test regimes:

Other Domains (OD): In this regime, when we test on problems from some domain, we train on all problems from all *other* domains. This simulates the setting where we need to estimate search progress on a problem from a completely unknown domain.

Same Domain (SD): In this regime, when we test on problems from some domain, we train only on problems from the same domain. To have some meaningful learning, we only used domains with more than 15 instances, leaving us only 3 domains. For each domain, we split its instances into the even- and odd-numbered problems, to obtain two sets of roughly the same size. We then trained a predictor on one set, and evaluated its performance on the other, giving us 6 data points in total.

Other Domains Tune Same (ODTS): In this regime, we first train a predictor based on instances from the other domains. We then take this trained model, and fine tune it on one half of the instances from the target domain (split, as before, to even and odd numbered problems). During this fine tuning we run the same training algorithm, except that the learning rate is 0.0001 and the batch size is 2048. As fine-tuning requires less data than SD, we used domains with more than 6 instances – giving us 6 domains and 12 data points in total.

We measure the accuracy of each search progress estimate prediction (for a given state) by the absolute error between the prediction and the true progress (which we have, since we use only solved problems). The accuracy of a search progress estimator over a given problem is the average accuracy of its predictions over all the states expanded by the algorithm, that is the mean absolute error (MAE). The accuracy for a set of problems (all problems in a domain, or all problems overall) is the average accuracy across these problems (giving equal weight to each problem, regardless of the number of states it required expending), that is, the average mean absolute error. We also report the average of domain averages (that is, giving the same weight to each domain, regardless of how many problems it contains).

We remark here that we used MSE rather than MAE as the loss function during training, as MSE gives us a smoother function. Nevertheless, as we discuss next, our technique achieves start-of-the-art performance.

Empirical Results

We now describe the results for our empirical evaluation. First, Table 1 compares our technique using the OD regime to the estimators proposed in previous work. OD is the closest to these in terms of assumptions, as it does not assume any knowledge about the problem it is used on.

In the table we can see OD gives the best accuracy in 11 out of 19 domains. In the other 8 domains, VaSP is the most accurate in 4 domains, while VeSP and PBP were the most accurate in two domains each. Overall, OD was the most accurate, with an average mean absolute error of 23.4%, compared to the best predictor from previous work, VeSP, with an average mean absolute error of 27.35%. The results when averaging over individual problems rather than domains are very similar.

Domain	VaSP	VeSP	PBP	DBP	OD
Airport(5)	29.38	19.44	**19.30**	17.19	25.01
Blocks(13)	26.55	26.02	31.59	45.31	**23.49**
Depot(3)	26.18	27.75	30.88	38.12	**23.55**
Elevators(11)	23.91	24.61	29.50	33.96	**21.97**
Freecell(1)	22.17	**20.65**	32.38	33.42	24.26
Gripper(4)	38.88	**18.50**	21.60	20.67	23.02
Logistics(5)	39.11	24.75	25.33	29.49	**20.88**
Openstacks(5)	**21.56**	33.90	37.30	37.27	25.16
Parcprinter(6)	34.33	34.87	35.11	40.60	**23.46**
Pegsol(15)	**23.39**	27.96	38.90	40.72	23.77
PSR-small(15)	28.19	26.52	41.95	40.65	**22.81**
Rovers(3)	24.81	32.11	34.07	39.35	**23.49**
Satellite(2)	30.53	22.91	**22.89**	32.93	24.77
Scanalyzer(3)	23.28	25.16	29.40	27.03	**22.31**
Sokoban(16)	**24.37**	26.48	38.95	38.17	24.52
TPP(1)	30.55	39.93	40.85	41.95	**24.48**
Transport(5)	**21.83**	23.42	30.25	33.93	22.62
WW(5)	46.63	38.66	38.65	43.24	**22.96**
Zenotravel(3)	26.90	26.05	29.96	29.25	**22.12**
Avg domain(19)	28.56	27.35	31.89	34.91	**23.40**
Avg problem(121)	27.66	27.06	33.93	36.93	**23.36**

Table 1: Average Mean Absolute Error (in percent) for Each Domain/Estimator Combination with A* and LMCUT

To get a better sense of the behavior of these estimators during search, Figure 3 shows plots of the absolute error during search, for 3 problems of different sizes. For each point on the x-axis, which indicates the true search progress, we plot the absolute error of each predictor at that state. As we can see, VaSP has a lot of noise. This is explained by the fact that every change in the value of $h_{\min}$ caused a major change in the estimation. With respect to the other estimators, our method had better accuracy until about the 60–70 percent point, and from there the prediction was least accurate and missed the end of the problem. The difference between our model and the baselines at the end of the search is not surprising, as the baselines depend more closely on h, and thus when h is close to zero, the estimates predict we are close to end of search.

Having seen that our technique, in the OD regime, outperforms previous work, we now turn to evaluating the impact of more specific training data by comparing the OD regime to the SD and ODTS regimes. Recall that in the SD regime our network is trained only on instances from the same domain, while in ODTS it is trained on instances from all other domains, and then fine-tuned on the domain being evaluated. Also recall that these regimes can only be used on domains which have enough data (15 solved instances for SD, 6 for ODTS), and thus the comparison is on fewer domains.

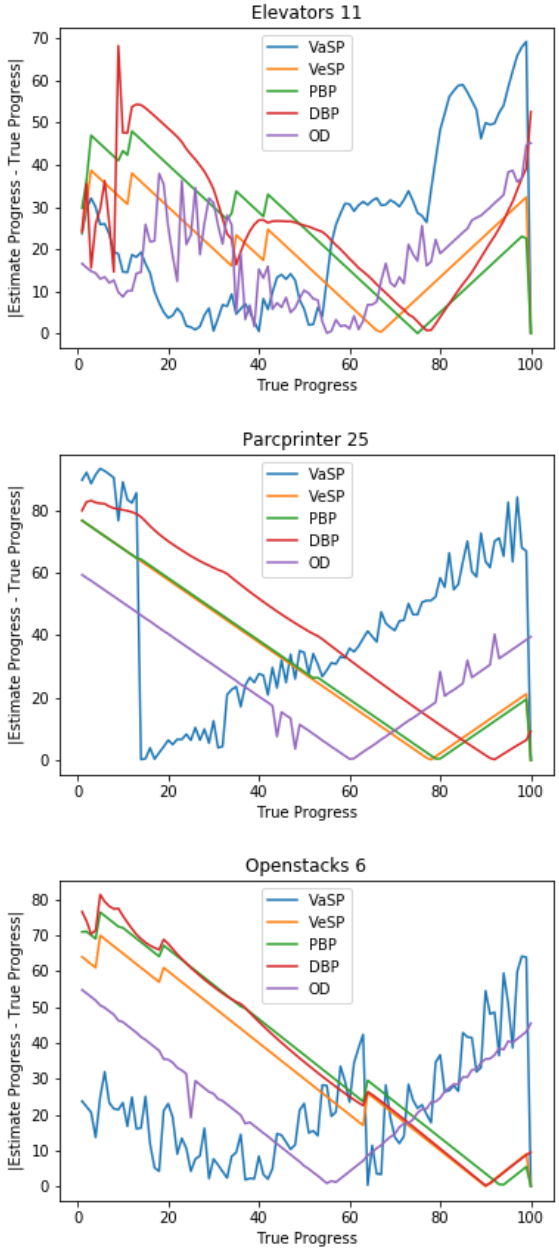

Figure 3: Absolute Error During Search in 3 Different Problems: Elevators 11 (top, 4,500 nodes), Parcprinter 25 (middle, 124,826 nodes), and Openstacks 6 (bottom, 866,801 nodes)

Domain	OD	SD	ODTS
Blocks	**23.49**	-	23.62
Elevators	21.97	-	**21.33**
Parcprinter	23.46	-	**23.45**
Pegsol	23.77	25.23	**22.59**
PSR-small	22.81	20.83	**20.31**
Sokoban	24.52	24.30	**23.55**
Avg domain(3)	23.70	23.45	**22.15**
Avg problem(46)	24.30	23.71	**22.18**

Table 2: Average Mean Absolute Error (in percent) with A* and LMCUT

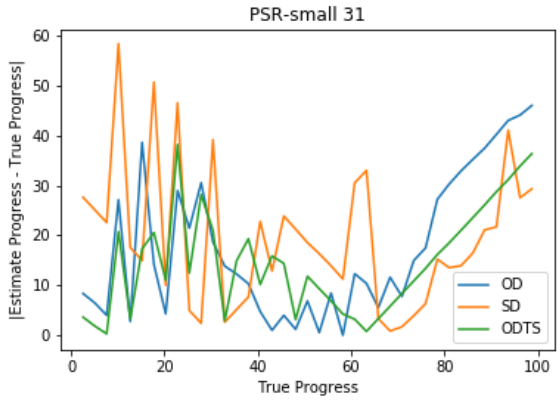

Figure 4: Absolute Error During Search in PSR-small 31 (3,950 nodes)

Table 2 shows the results for this evaluation. Interestingly, ODTS outperforms SD in all cases, showing that the training data from other domains is important. ODTS also outperforms OD in 5 out of 6 domains as well as overall, showing that domain-specific tuning does help.

Finally, Figure 4 plots the absolute error of the 3 different regimes on a specific instance, PSR-small 31. As we can see, all 3 predictors started with unstable estimates, and got more and more accurate in the middle. Near the end all predictors become less and less accurate and missed the fact they are near the goal. Other than this, we can see that ODTS was generally more accurate, while OD and SD were each better than the other in different parts of the search.

Discussion and Future Work

In this paper we have showed a novel approach to search progress estimation using deep learning. We have shown that our approach outperforms previous state-of-the-art approaches, and that it benefits from having access to better training data. It is also interesting (and perhaps sad) to note that the previous search progress estimators (Thayer, Stern, and Lelis 2012), which were developed manually using extensive expertise in heuristic search and human creativity, are outperformed by a machine learning algorithm with very limited prior knowledge.

Besides being an interesting problem, search progress es-

timation can be used to make decisions about search. For example, as previously mentioned, estimates of remaining search time have been used in situated temporal planning (Cashmore et al. 2018) and in Branch-and-Bound search (Anderson et al. 2019). Other applications are in anytime search (Dionne, Thayer, and Ruml 2011) or metareasoning (Shperberg et al. 2019, 2020). However, using search progress estimates to make decisions about search has a problem with self-reference – the predictions affect search decisions, which affect search time, which then affect the true search progress. We intend to address this challenge in future work by attempting to learn a series of predictors which converge to a fixed point which exhibits good performance.

Finally, we remark that we employed fairly simple machine learning tools here. In future work we will examine whether we can improve performance even more by using different architectures of neural networks, or more informative features.

References

Alkhazraji, Y.; Frorath, M.; Grützner, M.; Helmert, M.; Liebetraut, T.; Mattmüller, R.; Ortlieb, M.; Seipp, J.; Springenberg, T.; Stahl, P.; and Wülfing, J. 2020. Pyperplan. https://doi.org/10.5281/zenodo.3700819. doi:10.5281/zenodo.3700819. URL https://doi.org/10.5281/zenodo.3700819.

Anderson, D.; Hendel, G.; Le Bodic, P.; and Viernickel, M. 2019. Clairvoyant restarts in branch-and-bound search using online tree-size estimation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 1427–1434.

Belov, G.; Esler, S.; Fernando, D.; Le Bodic, P.; and Nemhauser, G. L. 2017. Estimating the size of search trees by sampling with domain knowledge. In *IJCAI*, 473–479.

Bonet, B.; and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2): 5–33.

Breyer, T.; and Korf, R. 2008. Recent results in analyzing the performance of heuristic search. In *Proceedings of the First International Workshop on Search in Artificial Intelligence and Robotics (held in conjunction with AAAI)*, 24–31.

Cashmore, M.; Coles, A.; Cserna, B.; Karpas, E.; Magazzeni, D.; and Ruml, W. 2018. Temporal Planning while the Clock Ticks. In *ICAPS*, 39–46.

Dionne, A. J.; Thayer, J. T.; and Ruml, W. 2011. Deadline-Aware Search Using On-Line Measures of Behavior. In *SOCS*, 39–46.

Doran, J. E.; and Michie, D. 1966. Experiments with the graph traverser program. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 294(1437): 235–259.

Gers, F. A.; Schmidhuber, J. A.; and Cummins, F. A. 2000. Learning to Forget: Continual Prediction with LSTM. *Neural Computation* 12(10): 2451–2471.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Syst. Sci. Cybern.* 4(2): 100–107. doi:10.1109/TSSC.1968.300136. URL https://doi.org/10.1109/TSSC.1968.300136.

Helmert, M.; and Domshlak, C. 2009. Landmarks, critical paths and abstractions: what's the difference anyway? In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 19.

Hochreiter, S. 1998. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6(02): 107–116.

Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural computation* 9(8): 1735–1780.

Hoffmann, J. 2011. Analyzing Search Topology Without Running Any Search: On the Connection Between Causal Graphs and h+. *J. Artif. Intell. Res.* 41: 155–229.

Hutter, F.; Xu, L.; Hoos, H. H.; and Leyton-Brown, K. 2014. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence* 206: 79–111.

Lelis, L. H.; Stern, R.; Felner, A.; Zilles, S.; and Holte, R. C. 2014. Predicting optimal solution cost with conditional probabilities. *Annals of Mathematics and Artificial Intelligence* 72(3): 267–295.

Lelis, L. H.; Stern, R.; and Sturtevant, N. R. 2014. Estimating search tree size with duplicate detection. In *Seventh Annual Symposium on Combinatorial Search*.

Medsker, L. R.; and Jain, L. 2001. Recurrent neural networks. *Design and Applications* 5.

Shperberg, S. S.; Coles, A.; Cserna, B.; Karpas, E.; Ruml, W.; and Shimony, S. E. 2019. Allocating Planning Effort When Actions Expire. In *AAAI*, 2371–2378. AAAI Press.

Shperberg, S. S.; Coles, A.; Karpas, E.; Shimony, S. E.; and Ruml, W. 2020. Trading Plan Cost for Timeliness in Situated Temporal Planning. In Bessiere, C., ed., *IJCAI*, 4176–4182. ijcai.org.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems-Volume 2*, 3104–3112.

Thayer, J. T.; Stern, R.; and Lelis, L. H. 2012. Are We There Yet?-Estimating Search Progress. In *SOCS*.

Yu, Y.; Si, X.; Hu, C.; and Zhang, J. 2019. A review of recurrent neural networks: LSTM cells and network architectures. *Neural computation* 31(7): 1235–1270.