J. Hlaváčová (Ed.): ITAT 2017 Proceedings, pp. 57–64
CEUR Workshop Proceedings Vol. 1885, ISSN 1613-0073, © 2017 R. Brunetto, O. Trunda

Deep Heuristic-learning in the Rubik's Cube Domain: an Experimental Evaluation

Robert Brunetto and Otakar Trunda

Charles University in Prague, Faculty of Mathematics and Physics
Malostranské náměstí 25, Praha, Czech Republic
robert@brunetto.cz otakar.trunda@mff.cuni.cz

Abstract: Recent successes of neural networks in solving combinatorial problems and games like Go, Poker and others inspire further attempts to use deep learning approaches in discrete domains. In the field of automated planning, the most popular approach is informed forward search driven by a heuristic function which estimates the quality of encountered states. Designing a powerful and easily-computable heuristics however is still a challenging problem on many domains.

In this paper, we use machine learning to construct such heuristic automatically. We train a neural network to predict a minimal number of moves required to solve a given instance of Rubik's cube. We then use the trained network as a heuristic distance estimator with a standard forward-search algorithm and compare the results with other heuristics. Our experiments show that the learning approach is competitive with state-of-the-art and might be the best choice in some use-case scenarios.

1 Introduction

Neural networks (NNs) have already proven to be able to cope with noisy and unstructured data like hand-written texts, images, sounds, classification of real-world objects based on incomplete description, and many others.

Recently, they also succeeded in several purely combinatorial domains like the game of Go. Nowadays, the program AlphaGo [15] that utilizes a deep neural net can beat top-class human players which was impossible just two years ago. No other approach is currently known to be able to play Go on such level. NNs are also a key component of the best Poker engine DeepStack [12] and several attempts have been made to use them for solving instances of Travelling Salesman Problem and other combinatorial problems [2].

In planning, the machine learning approaches are already being used in several ways, for example to select the best search algorithm, preprocess the problem or to promote searching of promising areas. See *International Planning Competition - Learning Track* [1] for more details.

Some attention has also been dedicated to heuristic learning, where the task is to automatically induce a heuristic function from training samples using a machine learning model. Models typically used in this area are very simple (like a decision tree or a shallow NN) and are not fine-tuned for the specific problem. In most cases, the are only used as a black box.

With recent rapid development of deep learning models, many new possibilities are now available in this area. Learning algorithms now exist for efficient training of deep feed-forward networks and also many other types of NNs have been developed and successfully used. For example, there are Deep Recurrent Networks like *LSTM*, Deep Convolutional Networks, Neural Turing Machines and others [2]. Using the CUDA framework, it is now possible to train such networks much faster on specialized graphic cards.

In this paper, we try to utilize some of such more complex models to learn an efficient heuristic function for solving the Rubik's cube puzzle. We work with the standard 3x3x3 cube, but our approach is in principle applicable to larger cubes as well.

1.1 Motivation

Our main motivation is to correct the error that heuristics make when estimating the distance. Admissible heuristics are often quite accurate near a goal state, but on states that are far from any goal state they significantly underestimate the real value [8]. The network should be able to find out in which situations such underestimation occurs and correct it appropriately.

Assuming that a set of admissible heuristics will be used as features, the network will play the role of a judge deciding which heuristics are trustworthy and which are not (in various circumstances).

Unlike in similar papers on heuristics learning, we include a simplified description of the state among the input features. Existing approaches only learn the goal-distance of the state using a few precomputed features that don't contain state description but mostly only heuristic estimates. We believe, that state description is important so that the network could distinguish between different types of states. Identifying types of states is crucial for the network to find out in which kinds of states the given heuristics underestimate and by how much.

[1]http://www.cs.colostate.edu/~ipc2014/

[2]http://www.asimovinstitute.org/neural-network-zoo/

2 Background

2.1 Rubik's Cube

The famous puzzle - Rubik's cube may serve as an example of a planning problem. The task is to find a sequence of actions leading form given initial state to specified goal state.

From a search perspective, the number of unique states of the standard 3x3x3 cube is $8! * 3^8 * 12! * 2^{12} * \frac{1}{12}$ which is about $4 * 10^{19}$. Every state can be solved optimally in at most 26 quarter-moves. There is a single goal state and the branching factor is 12 when using a symmetry breaking representation. We consider a model using quarter-moves meaning that it is allowed to rotate layers only by 90°. The half-move representation allows sides to be rotated by 180° in a single move.

To be able to describe some other properties of the cube, we first need to define a few additional notions. The 3x3x3 cube consists of 27 little cubes which we call *cubies*. Some faces of these cubies are colored, or more precisely they carry colored stickers. In the goal state, all faces of the large cube contains only stickers of the same color.

If we disassembled the cube, we would get 8 corner-cubies, each of which carrying 3 colored stickers, 12 edge-cubies, each with 2 stickers on them and 6 central-cubies each having 1 sticker.

A cube of any size can easily be solved suboptimally using a simple algorithm that runs in time $\mathcal{O}(n^2)$, where n is the size of the cube. Finding optimal solutions seems to be much harder, although the complexity class of this task has long been unknown. It has only recently been proven that solving Rubik's cube optimally is indeed NP-hard [4, 5].

Current state-of-the-art approaches for finding optimal or near-optimal solutions often utilize forward state space search using a pattern database as a heuristic [11, 16]. Pattern databases (PDBs) are precomputed solutions to smaller problems which are created from the original by abstracting-away some of its features. For example, if we only consider the 8 corner cubies of a standard 3x3x3 cube, we will get a 2x2x2 cube which is much easier to solve. The whole state space of this smaller problem may be enumerated, solved optimally for each state and stored in a database. For each state of the standard cube, we can then *project* it on the pattern by ignoring everything except the corners, look-up this state in the database and use its evaluation as a lower bound on the length of the plan. The set of features that we consider in the smaller problem is called *pattern* and features that are not included in the pattern are ignored.

A single pattern never contains all cubies. Using it as a heuristic leads to a state where all cubies contained in the pattern are correctly placed but others are not. In such states, the heuristic value is 0 and the algorithm has a hard time finding the goal state since it has no further guidance. For efficient searching it is necessary to combine several PDBs with different patterns or to combine PDB with other types of heuristics. There are many ways of combining the heuristics, from simple ones, like taking maximum, to more complex ones like *additive PDBs* or *cost partitioning* [10].

2.2 Heuristic Learning

The task of heuristic learning is to automatically create a heuristic function for given problem based on some training data. The learning is typically done a priory, where the training data are provided before the search. Heuristic can also be learned on-the-fly, where previous attempts to solve the problem serve as training data to learn future search strategy [7].

Several attempts have been made to utilize NNs for the heuristic learning task [1, 2, 14, 3, 17]. In the typical setting, a set of *features* is computed for every state in the training set as well as the optimal distance-to-go to the nearest goal state, and the network is then used to learn a mapping from features to distance estimate. After the learning process is finished, the network is used as a heuristic distance estimator together with an informed forward search algorithm like A* or IDA*.

Heuristics that are learned in this way provide no guarantees on admissibility. The goal of learning is so that the heuristic would be close to the real value but not necessarily always admissible - i.e. smaller than the real distance-to-go. Since search with an inadmissible heuristic doesn't typically guarantee finding optimal solutions, this approach is only suitable in cases where close-to-optimal solutions are sufficient. It is however possible to guarantee optimality even with an inadmissible heuristic by modifying the search strategy [9].

From the complexity perspective, the task of heuristic learning in general is hard. It is known that the task of finding optimal solutions to some planning problems like generalized 15-puzzle, Sokoban, and many more is NP-hard or even harder. It is also known that with an accurate-enough heuristic, the search time may be polynomial. Namely, if $\forall x : (h^*(x) - h(x)) \in \mathcal{O}(\log(h^*(x)))$ where x are states, h is heuristic and h^* is the real goal distance, then the search time is polynomial [13, p. 99]. It is therefore obvious that such heuristic cannot be computed in polynomial time unless some complexity classes collapse. The learned heuristic will probably not be able to solve large problems optimally in polynomial time but it may still outperform classic human-designed heuristics.

2.3 Notation

In the rest of the text, we use the following notation:

- S is the set of all states of the Rubik's cube
- $g \in S$ is the goal state
- $h^* : S \mapsto \mathbb{N}^0$ is goal-distance of states

- $F = (f_1, f_2, \ldots, f_n)$ is the set of real-valued features of states, where $\forall i : f_i : S \mapsto \mathbb{R}$
- $h_{NN} : F(S) \mapsto \mathbb{R}$ is the function that the neural network will approximate, where $F(S) = \{(f_1(s), f_2(s), \ldots, f_n(s)) \mid s \in S\}$

h_{NN} will then be used as a heuristic. When referring to this heuristic's value, we will write $h_{NN}(s)$ instead of $h_{NN}(f_1(s), f_2(s), \ldots, f_n(s))$. The goal of the learning is that $h_{NN}(s)$ is close to $h^*(s)$ for all states.

3 Getting the Training Data

There is a big issue of obtaining the training data. It is problematic to compute the real goal distances for a large number of training states due to enormous time complexity of such task. We have tried another way of generating training samples using backward search and random walks. This way we don't get exact goal-distances, but the obtained values should be close enough to them for most states.

3.1 Generating Samples by Random Walks

This approach works as follows:

1. Run a breadth-first search (BFS) from the goal state, store every visited state together with its goal-distance for as long as the memory suffices.
2. From the BFS frontier select K states at random.
3. From each of these K states, run a short random walk.
4. Among the states visited by random walks, select the desired number of states as a training set.

For the states visited by BFS, we know the exact goal distance. We have been able to generate 10^7 states which consumed about 5.5 GB of memory. This set contained all states with goal distance of 5 or lower and some states with goal distance of 6.

Random walks start from the BFS frontier and have length of 19. Random walk is forbidden to re-visit a state that it has visited previously, but two different random walks may intersect. The estimated goal distance of a node is then calculated as goal distance of the initial state of the random walk + the length of the random walk before it encountered the given state.

True goal distance of the state, where the random walk starts is known since it lies in the BFS frontier. For the other states that the random walk encounters, we only have an estimate of the goal distance. We believe, that for most states the estimate will be reasonably close to the real value, especially for states closer to the goal (i.e. closer to the BFS frontier) but the values are in general over-estimated.

Length of the random walks was set to 19 so that they could visit states that are as far as 25 from the goal state. (The BFS expands nodes to depth 6 and then a random walk goes for another 19 steps). The maximum distance to the goal of every state is 26, but the number of states that actually require 26 steps is very low so we ignore them in the training process.

If two random walks intersect, we recompute the goal distances of their states such that the triangle inequality holds. I.e. if some state is visited by a shorter walk, all its neighbours are informed of this shorter path and are recomputed if necessary. Then the information is passed on to their neighbours, their neighbours' neighbours etc. Experiments however show, that such intersections rarely happen.

A small number of random walks returns back to the set of states visited by BFS. Such random walks we discard from further use.

We generated a total of 100 000 samples evenly distributed in the state space.

3.2 Generating Samples Using an Optimal Solver

Our experiments showed that the random walks approach is not efficient enough. It introduces a significant noise into the training set which increases the training error of the network.

The noise is caused by the fact that a random walk of length k leads to a state whose real goal distance is typically smaller than k and cannot be exactly determined. Our experiments showed that the random walk over-estimates the goal distance by approximately 20%. The target value associated with the sample is in this case a random variable with mean of roughly $0.8 * k$ and a non-zero variance. It is impossible for the network to predict the noise caused by this variance which significantly increases the training error.

To counter this problem, we used an optimal solver to generate training samples with exact target values. We used the *Cube Explorer 5.13* software available on http://kociemba.org/cube.htm. The solver is quite fast on most inputs - it can optimally solve samples that are as far as 15 from the goal within a few seconds. There are however no guarantees on the runtime and some samples may require much more time to solve. For example, the superflip position, which is among the hardest takes more than 30 minutes to solve optimally with *Cube Explorer*.

We generated over 100 000 cube configurations by very long random walks from goal state. We then used the solver to find optimal solutions for each of these configurations. From the resulting optimal paths, we selected the training samples. We selected 3-4 configurations from every optimal path.

This way we generated a total of 345 396 training samples. Each of the samples is associated with a true goal distance. Generating the samples took about 48 wall-clock hours on 40 computers with 8 cores each. That gives about 15 000 CPU hours.

4 Feature Selection

We use a total of 22 features for every state. Some of the features are computed by PDB heuristics, some by other simpler heuristics and some of them describe the state directly.

We use 8 PDB heuristics with different patterns. The first pattern contains all 8 corner cubies and each of the other 7 patterns is composed by a set of 6 edge cubies. Every pattern contains different set of edge cubies and every edge cubie is included in some pattern.

All databases together contain about 150 million entries, take about 450 MB of memory and creating them took about 15 CPU hours.

Another five features are provided by simpler heuristics. These heuristics only count number of stickers that violate some conditions. They don't take into account cubies, i.e. they pretend that we can "unstick" some colored stickers from the cube and then stick them some place else. Heuristics count the number of such unstick-stick operations necessary to get the goal state of the cube. Heuristics work as follows:

- *errors*: Number of stickers that are not on the correct face. This number is then divided by 12 so that the resulting heuristic is admissible. (It is possible to move 12 stickers in a single move.)
- *distance*: The same as previous but stickers that need to go to the opposite face count as two because they require at least two moves to be placed correctly. The number is then also divided in order to be admissible.
- *pairs-different*: The total number of neighbouring pairs of stickers that are not in correct place. Neighbouring pair consists of two stickers on the same face that are next to each other (i.e. touching by an edge). Among all neighbouring pairs, we count those in which the two stickers have different colors. The number is then again normalized so that the estimate is admissible.
- *face-different*: The same as before but we consider every two stickers that are on the same face as a *pair* even if they are not next to each other.
- *face-error*: This only considers one specific face and counts how well it is completed.

The mentioned heuristics may look very similar but they actually work differently. For example, if we use the heuristics *pairs-different* and *face-different* to search for a solution of 8x8x8 cube, we get a very different kinds of states. Figures 1 and 2 show two states where the two heuristics got stuck respectively. The *pairs-different* heuristic (figure 1) created a number of connected components of the same color on each face, while *face-different* (figure 2) maximized the total number of colored stickers that are together on each face but it didn't create any coherent patterns.

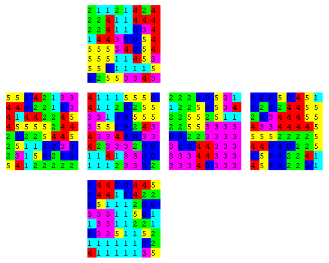
Figure 1: State found by the *pairDifferent* heuristic with coherent regions of the same color.

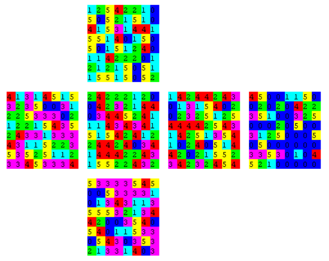
Figure 2: State found by the *faceDifferent* heuristic with much fewer coherent regions.

The rest of the features we use describe the state directly by specifying relations between cubies' current positions and their goal positions.

There are three types of cubies as described in 2.1: *centres, vertices* and *edges*. The central cubies never change their relative positions so we don't use them in this phase. For the other two types, we count every possible relation between positions of two cubies in the cube.

For example: two corner cubies may be in four different relative positions:

1. they are in the same place (i.e. they are both the same cubie)
2. they are on the same edge but not the same cubie
3. they are in the same face but not on the same edge
4. they are not in the same face (i.e. opposite corners of the cube)

In the given state, we determine the target position of every corner-cubie and then compute relation between the cubie's current position and it's target position. We then count the number of cubies that are in relations 1, 2, 3 and 4 respectively and use these numbers as features. For example, if we get numbers (8, 0, 0, 0), it means that in the given state, all corners are already in their respective correct positions.

We do the same for edge-cubies. Edges might be in 5 different relative positions, so this gives us another 5 features. (The number of edges that are in relation of $1,\ldots,5$ with their target positions.)

These features are not based on heuristics or estimating goal-distance. Instead they provide a description of the state. All these features combined should provide enough information such that the network recognizes types of states in which given heuristics underestimate the real goal-distance and will be able to correct the estimate.

5 Network Design and Training

The network consists of 6 layers in total. The input layer contains 22 neurons that reads the 22 features we use. All features are real numbers within range $[0,20]$. The four hidden layers contain 40, 36, 36 and 10 neurons respectively and all of them are feed-forward layers with *tanh* as their activation function. The output layer contains 1 neuron that computes the response of the network using linear activation. The response should be roughly within $[0,25]$ since in all training samples the target value was inside this interval.

The architecture and layers' sizes were designed according to "best practices" for the networks with similar number of inputs. We tried several other architectures (a cascade network, different numbers of hidden layers) but this one achieved best results.

To create and train the network, we used *Matlab Neural Network Toolkit*. We trained the network on a computer with processor *Intel Core i7 920 (4x 2.66 GHz + Hyper-Threading)*, *12 GB RAM*, graphic card *Nvidia GeForce 210* that supports CUDA. The training took about 5 hours.

Other training parameters like *batch-size*, *learning rate* and so on were kept on the default values suggested for this kind of network by the framework.

5.1 Training Results

We only present training results obtained by the exact sample-generation strategy using the optimal solver. The strategy using random walks also works but leads to larger training error.

The accuracy of fit is depicted in figure 3. Histogram shows number of samples that are within a specified distance from their respective targets. Ideally, all samples should be in the column marked 0. We see that the network's answers are slightly biased since most samples lie in the column 1. This means that the network underestimates the true value.

Figure 4 shows more precisely the distribution of training error. The horizontal axis enumerates intervals and the height of columns represents the percentage of samples whose training error (in absolute value) lies within the specified interval. We can see that for more than 60 % of samples, the error of estimation was less than 2. The mean absolute error was 1.94 with median of 1.22. The mean square error on the test set was 4.8 and median square error was 1.48. With the random walk sample-generation strategy the network only achieved mean square error of 8.9.

The training results show that for most samples the network was able to reasonably estimate their goal-distance

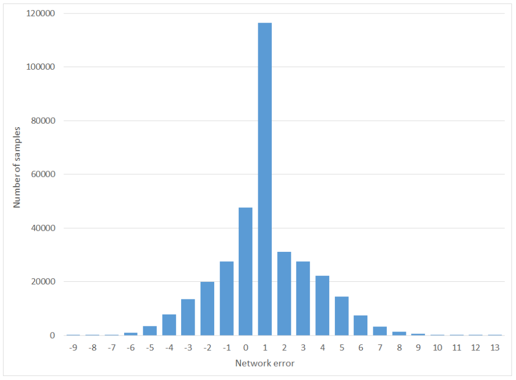

Figure 3: Histogram of accuracy of fit.

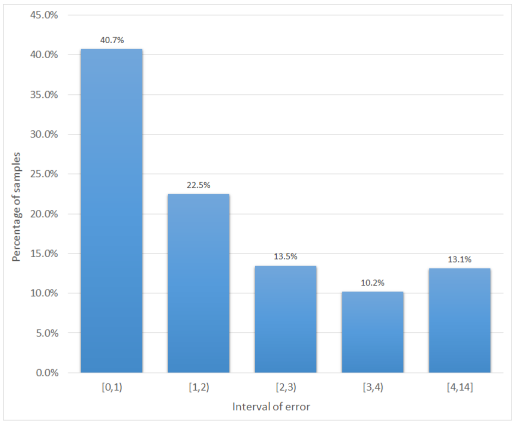

Figure 4: Distribution of training error of samples

but there is a few samples where the error is quite high. There is almost no difference between performance on training set and test set which suggests that the network is not over-fitted and should be able to generalize well.

Figure 5 shows a more thorough analysis of the network's performance. On the bottom axes, there are error of the network and the target value (which is the goal-distance of the sample). Height of the column represents the number of samples that falls into the respective category.

We can see that on samples that are close to the goal, the network is very accurate, has almost zero variance and low bias. On samples that are further from goal the network's answers became inaccurate. The variance increases and there is a slight bias towards under-estimating the real value.

6 Experiments

We used the trained network as a heuristic with IDA* algorithm on several randomly generated Rubik's cube instances. We compared the performance with other heuristics. *Iterative Deepening A* (IDA*)* is an algorithm similar to A* with the difference, that breadth-first search strategy is replaced by several cost-limited depth-first search runs.

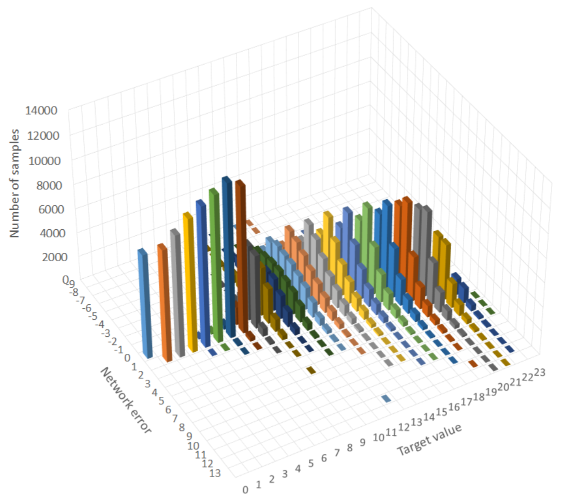

Figure 5: Distribution of training error with respect to target value.

The cost is computed in the same manner as with A* - using a heuristic estimator, and the limit successively increases until a solution is found. The algorithm provides the same guarantees as A* while requiring significantly less memory and slightly more time.

6.1 Experimental Setting

We generated 320 random instances of Rubik's cube and tried to solve them by IDA* with several heuristics. We generated samples by random walks starting in the goal state. We label the instances by length of the random walk that was used. Instances that were created by short random walks are easier because the instance is closer to the goal state. We generate instances by random walks of lengths 10, 14, 18, 22, 26, 30, 34 and 38. We run 40 walks of each length to get 320 instances in total.

In the experimental evaluation, we used six heuristics:

1. maximum of the five simple heuristics that we used as features (those that count stickers), denoted $h_{max(simple)}$
2. sum of these five heuristics, denoted $h_{sum(simple)}$
3. maximum of the eight PDB heuristics mentioned earlier, denoted $h_{max(PDB)}$
4. sum of these eight heuristics, denoted $h_{sum(PDB)}$
5. neural network as it was trained, denoted h_{NN}
6. neural network with a post-processing, denoted h_{NN+}

In the post-processing, we simply take maximum of the result of the network and the PDB heuristic, i.e. $h_{NN+} = \max(h_{NN}, h_{max(PDB)})$. Since the $h_{max(PDB)}$ is admissible, it makes no sense to estimate a value that is lower than $h_{max(PDB)}$. Furthermore, the network already has access to PDB estimates because it takes them as its input features.

We run each heuristic on all 320 problems with time limit of 5 minutes for each search instance. Experiments took about 115 CPU hours and run on 20 computers.

We measured several criteria:

- time required to solve the instance (capped at 300 seconds)
- number of expanded nodes during the search
- length of solution found
- minimal heuristic value of states that the algorithm encountered during the search

The heuristics $h_{max(simple)}$ and $h_{max(PDB)}$ are admissible. The sum of PDBs is not guaranteed to be admissible in this case and neither is the response of NN. The summation-based heuristics are more greedy and might be at least able to find sub-optimal solutions quickly.

6.2 Results

A table with detailed results can be downloaded at this link.

We present the results grouped by the length of random walk that was used to generate the problem instance. We call this the *difficulty of the instance*. The longer walk was used, the further from goal the instance is and therefore requires a longer sequence of actions to solve.

The number of solved instances by specific algorithms is shown in figure 6. We can see that with increasing difficulty of the problem the number of solved instances drops rapidly. This is mostly due to relatively strict time limit of 5 minutes for solving each instance.

We can also see that *NN* and *NN*+ heuristics solved the largest number of problems in most categories. In total, h_{NN} solved 130 problems, h_{NN+} 129 problems and $h_{max(PDB)}$ 126 problems. $h_{sum(PDB)}$ achieved much worse results and *Simple heuristics* scored the worst.

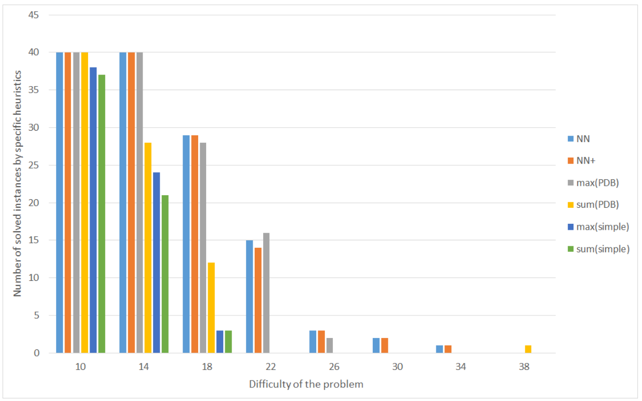

Figure 6: Number of problems solved by specific algorithms.

Figure 7 shows average number of expanded nodes of algorithms. Results are grouped by problem difficulty and only solved instances are considered. *Simple heuristics* have the largest number of expanded nodes (on problems that they have been able to solve) because they are the least informed ones.

PDBs have systematically higher number of expansion than both versions of neural networks. This suggests that both h_{NN} and h_{NN+} are more informed than $h_{max(PDB)}$. The effect is partially caused by the fact that evaluating h_{NN} is much slower than $h_{max(PDB)}$ and therefore the network is able to expand less nodes withing given time limit.

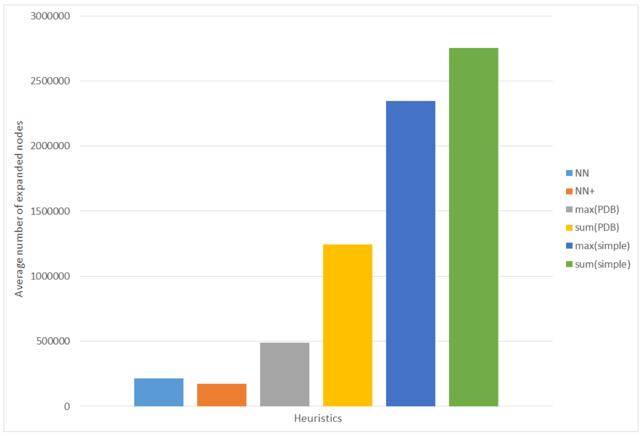

Figure 7: Expanded nodes of solved instances for specific algorithms.

Figure 8 analyzes run-times of individual heuristics. Problems are divided into categories according to how much time it took to solve them. The horizontal axis represents those categories as time intervals and height of columns shows how many problems fall into such category.

We can see that both networks as well as PDBs have been able to solve significant number of problems in time lesser than 1 second (for each problem). The run-time of h_{NN}, h_{NN+} and $h_{max(PDB)}$ are comparable. The time-performance of the other three heuristics is much worse.

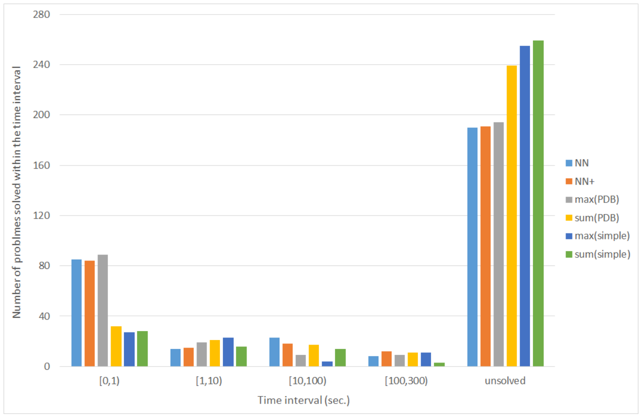

Figure 8: Histogram of run-time demands of problems.

Figure 9 shows average length of solution found by specific heuristics. Only solved instances are considered. The figure shows that *simple heuristics* found best solutions on average in categories 14 and 18. This is however caused by the fact that *simple heuristics* were only able to solve small problems that have shorter solutions and unsuccessful attempts are not considered in the average.

$h_{sum(PBD)}$ exhibits the worst average quality of solutions, but it has been able to solve the largest problem. This is caused by the fact that $h_{sum(PBD)}$ is the most greedy of all heuristics used. $h_{max(PBD)}$ has been able to find solutions with better average quality than both versions of the network. This is not surprising, since PDBs solved very similar set of problems as the NNs did and $h_{max(PBD)}$ is admissible so it guarantees finding optimal solutions. On average, NN found solutions that are 8.75% longer than optimal solutions found by PDBs. (Measured on instances that were solved by both h_{NN} and $h_{max(PDB)}$.

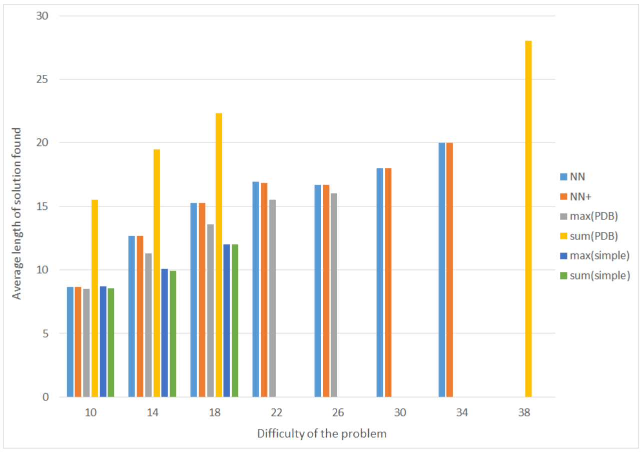

Figure 9: Average length of solution of solved instances for specific algorithms.

In figure 10 we can see the average of minimal heuristic value of states encountered during the search. Only unsolved instances are considered, because on solved problems the minimal heuristic value is always 0. We can see some interesting results here. $h_{sum(PDB)}$ was able to find states very close to the goal on most unsolved problems. It exhibits the lowest overall values even though the heuristic is very greedy and over-estimates the true value significantly.

On average, h_{NN} found states with better heuristic values than h_{NN+}. This is counter-intuitive as h_{NN+} is more informed. The result is most likely caused by the fact that h_{NN} under-estimated the true value of some states during the search, so it only "thought" it found close-to-goal states but it wasn't really the case. h_{NN+} is less prone to such under-estimating.

7 Conclusion

We trained and experimentally tested a neural network to estimate goal distances of Rubik's cube problems.

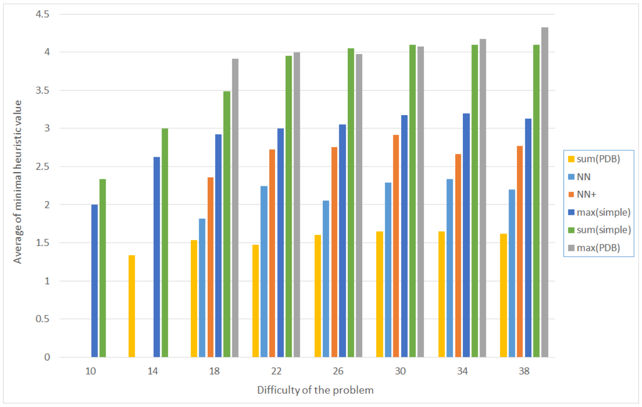

Figure 10: Average of minimal heuristic value found on unsolved problems.

The network is competitive with state-of-the-art pattern databases.

Our experiments show that PDBs are very fast and reasonably accurate while the network's evaluation is much slower because it involves computing the input features. Network therefore has lower search-speed in terms of nodes expanded per second. The NN heuristic, however, compensates this by being more informed than PDBs and can solve slightly larger number of problems withing given time limit.

On unsolved task the network seems to be able to find states that are closer to goal than $h_{max(PDB)}$. PDBs on the other hand guarantee optimality of solutions which the network does not.

We believe that the network represents an interesting and viable way of combining several heuristics together and for some use-case scenarios it may be the best choice. An ideal use-case scenario for NN is situation where we solve many problems from the same domain, there is enough time to prepare for the search (to train the network) and optimal solutions are not strictly required.

As a future work, we would like to find a balance between accuracy and speed of the network. By using a small set of suitable features, it should be possible to train network that is accurate and it's evaluation is still fast.

Acknowledgement

This research was supported by SVV project number 260 453.

References

[1] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C. Holte. Bootstrap learning of heuristic functions. In Ariel Felner and Nathan R. Sturtevant, editors, *Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010*. AAAI Press, 2010.

[2] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural Combinatorial Optimization with Reinforcement Learning. In *Proceedings of the Fifth International Conference on Learning Representations*, 2017.

[3] Hung-Che Chen and Jyh-Da Wei. Using neural networks for evaluation in heuristic search algorithm. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2011.

[4] E. D. Demaine, S. Eisenstat, and M. Rudoy. Solving the Rubik's Cube Optimally is NP-complete. *ArXiv e-prints*, June 2017.

[5] Erik D. Demaine et al. Algorithms for solving rubik's cubes. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Proceedings of Algorithms – ESA 2011: 19th Annual European Symposium*, pages 689–700. Springer Berlin Heidelberg, 2011.

[6] Dieter Fox and Carla P. Gomes, editors. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*. AAAI Press, 2008.

[7] Youssef Hamadi, Eric Monfroy, and Frédéric Saubion. *Autonomous search*. Springer-Verlag, 2012.

[8] Malte Helmert and Robert Mattmüller. Accuracy of admissible heuristic functions in selected planning domains. In Fox and Gomes [6], pages 938–943.

[9] Erez Karpas and Carmel Domshlak. Optimal search with inadmissible heuristics. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2012.

[10] Thomas Keller, Florian Pommerening, Jendrik Seipp, Florian Geißer, and Robert Mattmüller. State-dependent cost partitionings for cartesian abstractions in classical planning. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, 2016.

[11] Richard E. Korf. Finding optimal solutions to rubik's cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, pages 700–705. AAAI Press, 1997.

[12] Matej Moravčík et al. DeepStack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

[13] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[14] Mehdi Samadi, Ariel Felner, and Jonathan Schaeffer. Learning from multiple heuristics. In Fox and Gomes [6], pages 357–362.

[15] D. Silver, A. Huang, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[16] Nathan R. Sturtevant, Ariel Felner, and Malte Helmert. Exploiting the rubik's cube 12-edge PDB by combining partial pattern databases and bloom filters. In Stefan Edelkamp and Roman Barták, editors, *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014*. AAAI Press, 2014.

[17] Jordan Thayer, Austin Dionne, and Wheeler Ruml. Learning inadmissible heuristics during search. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2011.