Learning from Multiple Heuristics

Mehdi Samadi
Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E8
msamadi@cs.ualberta.ca

Ariel Felner
Information Systems Engineering Dept.
Deutsche Telekom Labs
Ben Gurion University
Beer-Sheva, Israel
felner@bgu.ac.il

Jonathan Schaeffer
Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E8
jonathan@cs.ualberta.ca

Abstract

Heuristic functions for single-agent search applications estimate the cost of the optimal solution. When multiple heuristics exist, taking their maximum is an effective way to combine them. A new technique is introduced for combining multiple heuristic values. Inspired by the evaluation functions used in two-player games, the different heuristics in a single-agent application are treated as features of the problem domain. An ANN is used to combine these features into a single heuristic value. This idea has been implemented for the sliding-tile puzzle and the 4-peg Towers of Hanoi, two classic single-agent search domains. Experimental results show that this technique can lead to a large reduction in the search effort at a small cost in the quality of the solution obtained.

Introduction

Heuristic evaluation functions play a key role in the effectiveness of solving single-agent search problems. Algorithms such as A* or IDA* are guided by the cost function $f(n) = g(n)+h(n)$, where $g(n)$ is the cost of reaching node n from the initial state and $h(n)$ estimates the cost of going from n to the goal. If $h(n)$ is admissible (never overestimates the actual cost) then these algorithms are guaranteed to find an optimal solution if one exists.

Single-agent search heuristic evaluation functions usually take one of two forms. The regular form is a single heuristic function that estimates the cost of moving from the current state to the goal. The second form adds together a number of smaller heuristic values, each of them estimating the cost of solving a disjoint subset of the problem. However, given multiple heuristics (any combination of regular heuristics and sums of disjoint heuristics), maximizing is still the method of choice to combine them (Korf and Felner 2002). Thus, for each state, the value of only *one* heuristic is used and knowledge from the rest is omitted from consideration.

In two-player games, heuristics are usually treated as features, each of them assigning a value to an aspect of a state. Usually, a linear combination is used to combine these feature values into a single heuristic assessment (e.g., (Campbell, Hoane, and Hsu 2002)). Knowledge from *all* of these heuristics is considered. The question is whether what works in two-player games can be effective for one-player (single-agent) applications.

There are differences between the one-player and two-player cases. In the two-player case, the features are designed to be as much as possible independent of each other. Hence a linear combination is fast and effective. The heuristic value obtained can be an over- or under- assessment; close is good enough. In contrast, for single-agent applications different heuristics are measuring the same thing (distance to the goal). Ideally the value would be admissible and as close as possible to the optimal value. Thus, the heuristics have a high correlation with each other and a linear combination of different heuristics won't be effective.

This paper treats single-agent search heuristics as features of the problem and, as in the two-player example, combines them to produce a single assessment. The heuristics are combined using an artificial neural network (ANN), since this better captures the high degree of overlap that can exist. In a sense, the ANN tries to approximate the optimal value by discovering the mutual influence and correlations between multiple heuristics.

The k different heuristics are treated as k features of the domain. The optimal solution cost is the target function. In a preprocessing phase, an ANN is built that learns a function to predict the solution cost given the k heuristics. For each state in a training set, its k heuristic values are fed into the ANN coupled with its pre-computed optimal solution. The resulting ANN is then used to supply the heuristic value for new instances of the problem domain. During a search (e.g. with IDA*), for each node s in the search tree the different k heuristics are computed and then used as inputs to the ANN. The output of the ANN (which is a prediction of the optimal solution) is used as $h(s)$ in the search algorithm. This ANN is similar to a pattern database (PDB) in the sense that it is built in a pre-calculation phase and is being consulted during the search to extract a heuristic value (but is much more space efficient than a PDB; all you need is a few hundred bytes to store an ANN). The contributions of this paper are:

1: A learned heuristic (with ANN in our case) evaluation function for single-agent search applications that combines multiple overlapping heuristics.

2: The ANN heuristic cannot guarantee admissibility. An effective method is introduced for adjusting the learning to increase the probability of producing an optimal solution.

3: A new idea is introduced, training using a relaxed version of the problem domain, that allows the ANN approach to be

effective for large domains where it is impractical to build a large training set.

4: Experiments on the sliding-tile puzzles and the 4-peg Towers of Hanoi demonstrate high-quality solutions (most are optimal) with a major improvement in execution speed. A comparison with weighted A* (WA*), a popular non-admissible heuristic modification to A*, shows the ANN can achieve better solutions with less effort.

This work introduces a new inadmissible heuristic that produces high quality solutions (most often optimal) with greatly reduced search effort. It does not depend on the search algorithm. Thus, for each domain in this paper we use the most appropriate search algorithm, including IDA*, recursive best-first search (RBFS) (Korf 1993) and frontier A* (FA*) (Korf et al. 2005). The AI literature concentrates on discovering optimal single-agent-search solutions. In reality, optimality is not needed for many real-world applications. Examples of these domains include pathfinding in computer games (Sturtevant and Buro 2005), robotic navigation (Likhachev, Gordon, and Thrun 2003), and planning (IPC 2008). In particular, the planning community recognizes that many important planning applications are so challenging that obtaining an optimal solution is not practical. Most planners strive for a solution—any solution—with solution quality being a secondary consideration.

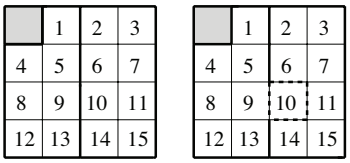
Application Domains

The sliding-tile puzzle consists of a square frame containing a set of numbered square tiles, and an empty position called the blank. The legal operators are to slide any tile that is horizontally or vertically adjacent to the blank into the blank position. State-of-the-art heuristics allow sub-second times for solving arbitrary instances of the 15-puzzle (4×4), however the 24-puzzle (5×5) remains computationally challenging.

The 4-peg Towers of Hanoi (TOH4) is an extension of the well-known 3-peg problem (Hinz 1997). It consists of 4 pegs and n discs of differing sizes. The task is to transfer all the discs from the initial peg to a goal peg. Only the top disc on any peg can be moved, and a larger disc can never be placed on top of a smaller disc. Unlike the 3-peg version, where a simple recursive algorithm exists, systematic search is currently the only method guaranteed to find optimal solutions for problems with 4 or more of discs.

Related Work

Learning single-agent-search heuristics from a given set of training data has been explored by others. (Ernandes and Gori 2004) uses a multi-layered ANN for learning heuristics in the 15-puzzle. Given a training set of states together with their optimal solution, they built a learning system that predicts the length of the optimal solution for an arbitrary state. Their features were the puzzle tiles, and the values of the features were the locations of the tiles. They biased their predicted values towards admissibility but their success was limited—optimal solutions were obtained in only 50% of the cases. The average time to solve a problem was faster than using the admissible Manhattan Distance heuristic by a factor of 500. Unfortunately, they did not use state-of-the-art heuristics (which can improve the search by many orders of magnitude) nor did they compare their approach to other non-admissible approaches.



(a) (b)

Figure 1: The 7-8 PDBs vs. 8-8 PDBs.

(Hou, Zhang, and Zhou 2002) use statistical learning techniques to predict the optimal solution for the (trivial) 8-puzzle (3×3). Again, the location of the tiles are used as the input to the system and the optimal solution length is the output. For an arbitrary instance of the puzzle, their initial results were *worse* than using Manhattan Distance. They modified their learning system to reduce the mean-squared error and obtain a more accurate heuristic function. They did not try biasing their predicted values towards admissibility.

The above efforts attempt to learn using primitive features of the application domain (tile locations and values). This can be misleading. Consider the 4×4 puzzle with all tiles in their goal locations, except that tiles 1, 2 and 3 are rotated to locations 2, 3 and 1. Here the values for the different features (tile locations) are similar to that of the goal state, with only three of the fifteen tiles out of place. Thus, this state seems to be very close to the goal ($h(s)$ is small). The small deviation (three swapped tiles) is deceptive to the learning algorithm since the solution length is large (18 in this case).

The work presented in this paper uses *heuristic values* as features of a state, not just the description of the states.

Learning From Heuristics

When maximizing over multiple heuristics, only one heuristic (the maximum) is taken and the knowledge contained in the other heuristics is ignored. Additive heuristics have the advantage that information from several heuristics are being considered, but are only admissible if they are completely disjoint (Korf and Felner 2002). For example, Figure 1(a) shows the state-of-the-art 7-8 partitioning of the tiles into two disjoint pattern databases (PDBs); adding preserves admissibility. In Figure 1(b) there are two groups of 8 tiles but tile 10 belongs to both groups. In this case adding the two PDB values loses admissibility because the moves of the overlapping tile will be counted twice. When overlapping features exist (tiles in our case) the individual heuristics can be more informed. However, adding them might significantly overestimate the optimal cost.

Our solution to this is to combine knowledge (even overlapping knowledge) from *all* of the available heuristics. Each heuristics is treated as one characteristic (or feature) of the problem state that has some internal wisdom about the optimal solution. The role of the learning technique is to discover the relative influence of each of the heuristics and their correlations to each other. A non-linear function (as might be discovered by, for example, an ANN) will compensate for inter-dependence between heuristics.

The basic algorithm that is used is as follows:

1: Identify a set of k heuristics that can be used to approximate the solution cost for a given domain state $(h_1, h_2, \ldots, h_k)$. For each state s, define the *heuristic vector* as $H(s) = \{h_1(s), h_2(s), \ldots, h_k(s)\}$. Any set of heuristics can be used, with no regard for admissibility. They can be a function of any part of the state—the entire state or just a small subset. In this paper, a variety of admissible PDB heuristics are used, allowing for a fair comparison against the current best admissible implementations for our two experimental domains.
2: Build a training set. A set of problem instances S is used as the training set. For each $s \in S$, the heuristic vector $H(s)$ is calculated and then a search is conducted (e.g., with the best existing heuristic) to find the optimal solution cost, $opt(s)$. This is done in a preprocessing phase.
3: A learning algorithm (an ANN in this case) is fed with the training set values for each $H(s)$ and $opt(s)$ and learns a target function that will predict $opt(s)$.
4: The learned evaluation function is then used to generate $h(s)$ for a search algorithm such as A*, IDA*, FA* or RBFS. When a state s is reached in the search, the heuristic vector $H(s)$ is calculated and fed into the ANN. The ANN's output value is used as $h(s)$.

Selecting the training set

The training set should reflect characteristics of the search space. A representative data set would include problem instances with a variety of solution costs. For the sliding-tile puzzles, for example, selecting random states will not be effective as it will tend to generate instances with large distances to the goal. Instead, the training data is generated by random walks backwards from the goal state (eliminating internal loops) to ensure that the data set includes a variety of solution costs (small and large).

ANN

The ANN used in this research is a connected feed forward neural network (Mitchell 1999). The hidden-layer neurons use the hyperbolic tangent activation function and the back propagation algorithm is used to train the neural network. Genetic algorithm were used in experiments to tune neural networks parameters.

Note that after the training phase, the ANN remains static. This does not have to be so. The ideas used by (Tesauro 1995) in his famous backgammon program are applicable here. One could use temporal differences to update the neural net after each problem solved, allowing the learning to be continuous. This is the subject of future work.

Biasing the ANN towards admissibility

Traditionally, the training phase is stopped when the *mean square error* (MSE) of the training data is below a predefined small threshold. It is defined as:

$MSE = \Sigma_{t \in T}(E(t)^2)/|T|$,

where T is the training set, $E(t) = \phi(t) - opt(t)$, opt is the target function, and ϕ is the learned function.

$E(t)^2$ is symmetric, so overestimation and underestimation have the same cost. Using this function with an ANN results in a heuristic that tries to be close to the optimal value without discriminating between being under (good) or over (undesirable). The error function can be modified to penalize positive values of $E(t)$ (overestimation), biasing the ANN towards producing admissible values. Define a new error function, $E'(t)$, as:

$E'(t) = (a + \frac{1}{1+\exp(-bE(t))})E(t)$

and use this instead of $E(t)$ in the MSE calculations. $E(t)$ and $E'(t)$ are compared in Figure 2. The new function is asymmetric; positive values of $E(t)$ (x axis) lead to a larger penalty (y axis) than for negative values. The parameters a and b determine the slope of the left side ($E < 0$) and the right side ($E > 0$), respectively. The new ANN is called penalty-enhanced ANN (PE-ANN) and it reduces the number of overestimating instances by a factor of roughly 4.

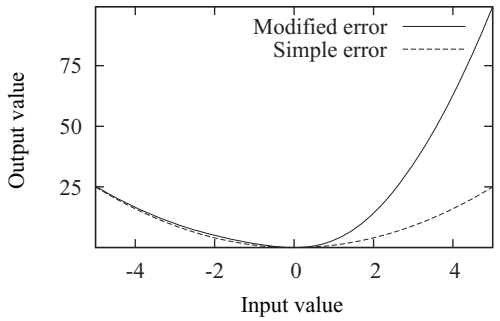


Figure 2: An example of simple and modified error function.

Experimental Results: 15-puzzle

Table 1 presents results for the 15-puzzle. The columns are: the size of $H(s)$, the heuristics used, the average heuristic value, the number of cases (out of 1,000) where the initial state heuristic was overestimated, the average cost of the solutions, the number of nodes generated, and the average running time (in seconds). The ANN used a training set of 10,000 instances. RBFS was used to solve the set of 1,000 problem instances from (Korf and Felner 2002). RBFS is a linear-space algorithm which expands nodes in a best-first order, even if the cost function is not monotonic—as is true for our applications. All experiments were performed on a AMD Athlon PC (2.2 GHz) with 2 GB of memory.

Line 1 presents the result of using IDA* with the heuristic of taking the maximum of the 7-8 disjoint PDBs and their reflection (Korf and Felner 2002). Line 2 uses the same heuristic but with the RBFS algorithm (slightly larger trees with more algorithm overhead). The next three lines show the results of using different heuristic vectors with the ANN. Line 3 shows the results of the ANN that combines $k = 4$ heuristics: the 7-tile PDB, the 8-tile PDB, and their reflections. Line 4 uses an ANN enhanced with a fifth feature, Manhattan Distance (MD). Compared to line 1, this version yields a 16-fold reduction in tree size, at the cost of less than 4% degradation in solution quality. Line 5 shows the results when 256 different disjoint 5-5-5 PDB partitions were used ($k = 3 \times 356 = 768$). With so many features to learn, the poor performance here (compared to line 4) is probably due to insufficient training.

k	Heuristic	Avg H	$h\uparrow$	Cost	Nodes	Time
Optimal benchmark results						
N/A	7-8 (IDA*)	45.63	0	52.52	36,710	0.019
N/A	7-8 (RBFS)	45.63	0	52.52	38,552	0.032
ANN						
4	7-8	56.52	524	55.55	2,915	0.002
5	7-8 + MD	54.72	482	54.26	2,241	0.001
768	5-5-5	60.08	592	58.84	3,482	1.215
PE-ANN						
4	7-8	50.82	94	52.72	14,852	0.014
5	7-8 + MD	50.56	78	52.61	16,654	0.021
Weighted 7-8 PDB						
N/A	W=1.05	-	-	52.56	27,486	0.028
N/A	W=1.07	-	-	52.64	21,762	0.021
N/A	W=1.10	-	-	52.82	13,826	0.013

Table 1: Results for the 15-puzzle.

Heuristic	Cost	Nodes	Time
100 random test cases			
max(120)	145.74	9,037,369	21.17
ANN(120)	145.92	78,246	0.34
PE-ANN(120)	145.82	184,182	0.61
The standard initial state			
max(120)	161	13,578,169	31.08
PE-ANN(120)	161	49,956	0.18

Table 2: Results for TOH4 (16 discs).

The next two lines (6 and 7) show the results for the penalty-enhanced ANN. Training with the penalty term significantly improved the solution quality—it is now within 0.1% of optimal—at the cost of significantly larger searches (but still a factor of two less than the benchmark).

Weighted A* (WA*) uses the cost function $f(n) = g(n) + w * h(n)$ where $W \geq 1$ (Pohl 1973). For $W > 1$ WA* is inadmissible, but typically a solution is discovered much faster. We compared the ANN heuristic to weighted heuristics using RBFS. The last lines of Table 1 show the results of the weighted PDB (WPDB) heuristic for different values of W. The PE-ANN results are comparable to that of weighted RBFS. PE-ANN produces slightly better solution costs but none of the results (lines 6-10) dominates the others on both the tree size and execution time dimensions.

Since the 7-8 PDB heuristic is very accurate and the search tree size using this heuristic is rather small, there is only little room for improvement. In the next sections we present the results on larger domains.

4-Peg Tower of Hanoi (16 Discs)

A 14-disc PDB is used for TOH4 (16 discs). There are 120 ways to choose 14 discs out of 16; $k = 120$ for our experiments.[1] The training set had $10,000$ entries. The optimal solution for each instance was found using Frontier A* (FA*) (Korf et al. 2005), a best-first algorithm that does not need to store the open list. FA* is commonly used to solve TOH4 (Felner, Korf, and Hanan 2004).

[1]A single 14-disc PDB provides all 120 options because only the relative size of the discs matter, not their absolute size.

W	Cost	Nodes	Time
100 random test cases			
1.1	145.94	5,993,461	14.11
1.2	146.56	4,072,095	9.82
1.3	146.18	2,060,251	4.92
1.4	147.47	1,455,607	3.82
standard initial state			
1.1	161	8,529,224	19.92
1.2	162	6,149,350	13.48
1.3	161	2,618,027	6.19
1.4	163	2,387,701	5.95

Table 3: WFA* results for TOH4 (16 discs).

Table 2 shows results for the 16-disc TOH4 on 100 random initial states. Line 1 presents results of an optimal solver which includes the 14-2 PDB heuristic, maximized over all 120 partitionings. The next lines show the ANN and PE-ANN results using the same 120 partitionings. Adding the penalty term method reduced the difference between the solution cost obtained to the optimal solution cost from 0.12% to 0.05%. The price was a 2-fold increase in the number of nodes searched. When compared to the optimal solution of line 1, the ANN gets a 115-fold reduction in search-tree size (49-fold for PE-ANN) with only a small (less than 1%) degradation in solution quality.

The last two rows give results for the standard initial state where all the discs are in the initial peg. Here, the ANN finds the optimal solution with an impressive 271-fold reduction in tree size (172-fold reduction in execution time).

Table 3 shows results for Weighted FA* on on these cases. Compared to the ANN results of Table 2, WFA* finds slightly worse quality solutions, but with a large loss in performance (18- to 76-fold depending on W) for the 100 random instances. PE-ANN has even better solution quality with a 7- to 32-fold performance gain over WFA*. PE-ANN also outperforms WFA* on the standard initial state with a 52-174-fold reduction in nodes and a 34-110-fold reduction in time (for the two admissible cases of $W = 1.1$ and $W = 1.3$).

Solving Larger Problems

An assumption that has been made thus far is that a large set of problem instances, coupled with their optimal solution costs, exist or is easily created. Generating large training sets is possible for small problems, such as the 15-puzzle and the 16-disc TOH4 problem, where optimal solutions to random instances can be found very fast. However, optimally solving larger problems becomes problematic due to time constraints. It takes two days to optimally solve a typical 24-puzzle problem using the best existing techniques. Solving 1,000 instances would take many CPU years.

To address this issue, the ANN can be trained using a relaxed version of the problem domain. Assume that P is the full problem. First, abstract P into a relaxed version of the problem, P'. Second, optimally solve training set instances of P'. Third, build an ANN for P' using the above methods. Finally, use this ANN as a lower bound for the full problem P. The advantage of this new method over ordinary PDBs is that much larger relaxed versions can be used.

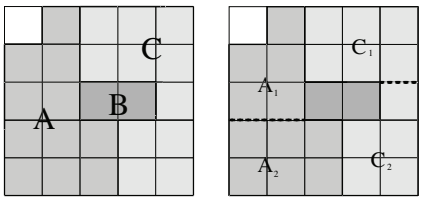


Figure 3: The 11-11-2 partitioning of the 24-puzzle.

Heuristic	Cost	Nodes	Time
6-6-6-6	100.78	360,892,479,670	156,007
PE-ANN			
11-11-2	101.41	118,465,980	111
weighted PDB			
6-6-6-6 W=1.1	101.66	8,920,346,297	7,084
6-6-6-6 W=1.2	103.58	133,032,445	107
6-6-6-6 W=1.3	106.58	10,554,813	9
6-6-6-6 W=1.4	110.30	1,400,431	1
weighted PE-ANN			
11-11-2 W=1.1	102.92	7,254,440	8
11-11-2 W=1.2	104.48	582,466	0.7

Table 4: Results for the 24-puzzle.

There is an important consideration in determining an effective relaxation abstraction for a problem. The more relaxation done to the problem, the easier the relaxed problem will be to solve (important for building a large training set). However, the more relaxed the problem, the less likely that solutions to the relaxed problem will be effective for the original problem. Clearly there is a tradeoff here.

24-puzzle

Figure 3(a) shows the 11-11-2 disjoint partitioning of the 24-puzzle, labeled A, B, and C, which was used in our relaxations. Building a PDB for an 11-tile subproblem is impractical as there are roughly 10^{16} different combinations. Instead, the 11-tile subproblem A is relaxed into disjoint 5-tile (A_1) and 6-tile (A_2) PDBs (and, hence, are additive), as shown in Figure 3(b). Similarly, the C_1 and C_2 PDBs were built for subproblem C. For each of the 11-tile subproblems, a PE-ANN was built each with a heuristic vector of $k = 4$ features (two sets of a 5-6 partitioning , one shown in the figure and another one).

Table 4 shows the results for the 24-puzzle averaged over the same 50 instances used by (Korf and Felner 2002). Line 1 reports results for IDA* and the 6-6-6-6 additive PDB heuristic (including their reflections about the main diagonal) used in (Korf and Felner 2002). RBFS is used in the remaining lines. Line 2 uses our 11-11-2 PE-ANN. The results show the PE-ANN achieved a dramatic improvement by a factor of over 3,000 in the tree size and run time. This comes at the cost of sacrificing 0.6% in the solution quality.

The next four rows show results for various values of W using the 6-6-6-6 PDB heuristic. WPDB can find poorer quality solutions than the ordinary 6-6-6-6 PDB, but with less computational effort. The PE-ANN produces a better solution quality (by less than 0.3%) with a 75-fold in the search effort. Since WA* is an effective algorithm, why not try weighting the ANN heuristic? Table 4 also shows results using a weighted ANN heuristic. Introducing a weighting factor degrades the solution quality by a small amount. In return, impressive performance gains are seen. Using a weighted PE-ANN with $W = 1.2$, the program builds trees that are almost one million times smaller than the optimal solver while increasing solution cost by less than 4%. This version also handily beats our best WPDB implementation, both in tree size and in execution time.

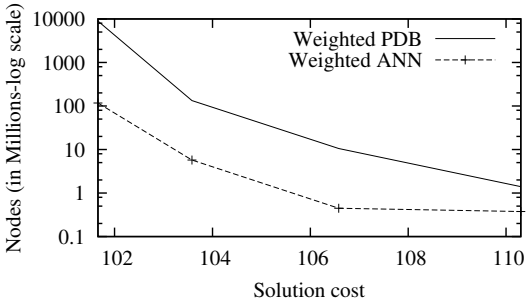


Figure 4: Comparing ANN and WPDB search-tree sizes.

Figure 4 compares the search-tree size as a function of solution quality for WANN and WPDB (in log scale). Clearly the ANN curve is below the WPDB curve meaning that better solutions are obtained for a given tree size or, conversely, less search is required for comparable quality solutions.

4-Peg Towers of Hanoi (17 and 18 Discs)

The relaxed ANN was also used for larger versions of TOH4. To solve the N-disc problem, an ANN for the M-disc problem, $M < N$, was used. The output of the M-disc problem is then used as a heuristic for the N-disc problem. Instances of the 17- and 18-disc TOH4 problem were solved by building the same 16-disc ANN described above for the largest 16 discs. The final heuristic for the 17- and 18-disc problems was the sum of the output of the 16-disc ANN and another PDB for the remaining smallest (1 or 2) discs.

Heuristic	Cost	Nodes	Sec
17 discs			
PDB (optimal)	184.2	$>$ 400,000,000	$>$ 950
WPDB (W=1.2)	185.1	34,444,992	81
WPDB (W=1.4)	185.7	10,404,654	24
WPDB (W=1.6)	189.2	5,568,465	14
ANN	184.4	14,736,538	45
WANN (W=1.1)	184.7	1,996,043	7
18 discs			
PDB (optimal)	217.1	$>$ 400,000,000	$>$ 950
WPDB (W=1.3)	219.4	332,030,484	762
WPDB (W=1.4)	220.6	183,839,127	424
WPDB (W=1.5)	221.1	87,492,338	203
ANN	217.2	261,449,786	725
WANN (W=1.1)	217.5	29,395,003	84

Table 5: Results on 10 random instances of TOH4.

Table 5 shows the results of solving 10 random instances of TOH4 with 17 and 18 discs. Non-trivial instances of these puzzles cannot be solved even when using 120 reflections of our 14-disc PDBs. The memory is exhausted once

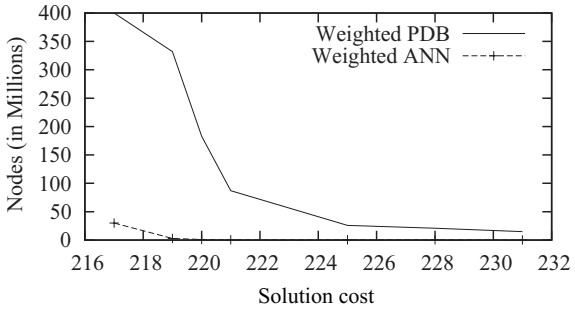


Figure 5: WPDB and WANN tree sizes for TOH4 (18 discs).

400,000,000 nodes are generated (FA* stores the frontier in memory). Thus, to obtain the optimal solutions reported in the table, we used sophisticated PDB compression methods (Felner et al. 2007) on a larger 16-1 PDB and a 16-2 PDB for the 17- and 18-disc problems.

These problems were solved with WFA*, using the relaxed 16-disc ANN just described and weighting this ANN (WANN). The table shows that WANN significantly outperforms WPDB. For example, WANN finds a solution that is within 0.1% of optimal while being at least 200 times faster (than the 400,000,000 bound). In contrast, WPDB expends more effort to find lesser-quality solutions. Similar results were obtained for the 18-disc problem.

Figure 5 compares the number of generated nodes as a function of the solution cost returned by WPDB and WANN for the 18-disc problem. Clearly, the WANN curve shows better performance than WPDB. Comparing identical quality (cost=219 for example), one sees that WPDB is many orders of magnitude slower. Comparing equal search effort (nodes=20 million, for example), WPDB has almost a 6% degradation in the solution cost.

All experiments used a 14-disc PDB and did not exploit the symmetric property of TOH. Our WANN can find close to optimal solution for the initial instance of up to the 30-disc TOH using only a 15-disc PDB. (Korf and Felner 2007) solved the same problem using a PDB of size 15, while exploiting the symmetric property of the problem (as well as using numerous other enhancements). The ANN solution was obtained much faster, used considerably less storage, but not guaranteed to be optimal in general.

Summary and Conclusions

This paper presents an improved heuristic for single-agent search for application domains that require high-quality solutions. By combining multiple heuristic values using an ANN, the resulting $h(s)$ value allows a search algorithm (such as IDA*, RBFS, and FA*) to quickly solve a problem with an optimal or near optimal solution. This property is particularly important for real-time domains, where the luxury of long search times is not practical.

WA* is a simple algorithm—essentially changing one line of code in A* (or FA* and RBFS). Building an ANN evaluation function is more work, but given that there are standard packages for ANNs, the implementation overhead is still relatively small. In return for this additional development effort, our results for two application domains show smaller search trees and faster execution time over WA* (in some cases, several orders of magnitude). These results were demonstrated using the sliding-tile puzzle and the Towers of Hanoi (additional results, not reported here, were obtained for the Top Spin puzzle).

The ideas presented in this paper can be applied to two-player adversarial search. Instead of using a single evaluation function in a game-playing program, one can combine evaluation functions and/or features from multiple programs. Given the proliferation of multi-core processors and the notorious poor parallel scalability of $\alpha\beta$ search, using additional resources to improve the quality of the evaluation function looks like a promising approach. Initial results for chess endgames are very encouraging.

Acknowledgments

The first author would like to thank Zohreh Azimifar and Maryam Siabani for their valuable discussions and suggestions. This research was supported by the Israel Science Foundation (ISF) under grant #728/06 to Ariel Felner and by iCORE.

References

Campbell, M.; Hoane, J.; and Hsu, F. 2002. Deep Blue. *Artificial Intelligence* 134(1-2):57–83.

Ernandes, M., and Gori, M. 2004. Likely-admissible and sub-symbolic heuristics. In *ECAI*, 613–617.

Felner, A.; Korf, R.; Meshulam, R.; and Holte, R. 2007. Compressed pattern databases. *JAIR* 30:213–247.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *JAIR* 22:279–318.

Hinz, A. M. 1997. The tower of Hanoi. In *Algebras and Combinatorics: ICAC'97*, 277–289.

Hou, G.; Zhang, J.; and Zhou, J. 2002. Mixture of experts of ANN and KNN on the problem of puzzle 8. Technical report, Computing Science Department University of Alberta.

IPC. 2008. International planning competition. `http://ipc.informatik.uni-freiburg.de`.

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134:9–22.

Korf, R. E., and Felner, A. 2007. Recent progress in heuristic search: A case study of the four-peg towers of hanoi problem. In *IJCAI*, 2324–2329.

Korf, R. E.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *JACM* 52(5):715–748.

Korf, R. E. 1993. Linear-space best-first search. *Artificial Intelligence* 62(1):41–78.

Likhachev, M.; Gordon, G. J.; and Thrun, S. 2003. ARA*: Anytime A* with provable bounds on sub-optimality. In *NIPS*.

Mitchell, T. M. 1999. Machine learning and data mining. *Communications of the ACM* 42(11):30–36.

Pohl, I. 1973. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *IJCAI-73*, 12–17.

Sturtevant, N. R., and Buro, M. 2005. Partial pathfinding using map abstraction and refinement. In *AAAI*, 1392–1397.

Tesauro, G. 1995. Temporal difference learning and td-gammon. *Commun. ACM* 38(3):58–68.