Bootstrap Learning of Heuristic Functions

Shahab Jabbari Arfaee
Computing Science Department
University of Alberta
Edmonton, AB Canada T6G 2E8
(jabbaria@cs.ualberta.ca)

Sandra Zilles
Computer Science Department
University of Regina
Regina, SK Canada S4S 0A2
(zilles@cs.uregina.ca)

Robert C. Holte
Computing Science Department
University of Alberta
Edmonton, AB Canada T6G 2E8
(holte@cs.ualberta.ca)

Abstract

We investigate the use of machine learning to create effective heuristics for search algorithms such as IDA* or heuristic-search planners. Our method aims to generate a strong heuristic from a given weak heuristic h_0 through bootstrapping. The "easy" problem instances that can be solved using h_0 provide training examples for a learning algorithm that produces a heuristic h_1 that is expected to be stronger than h_0. If h_0 is too weak to solve any of the given instances we use a random walk technique to create a sequence of successively more difficult instances starting with ones that are solvable by h_0. The bootstrap process is then repeated using h_i in lieu of h_{i-1} until a sufficiently strong heuristic is produced. We test our method on the 15- and 24-sliding tile puzzles, the 17- and 24-pancake puzzles, and the 15- and 20-blocks world. In every case our method produces a heuristic that allows IDA* to solve randomly generated problem instances extremely quickly with solutions very close to optimal.

Introduction

Modern heuristic search and planning systems require good heuristics. The main approach to creating heuristics for a state space is abstraction: from the state space description one creates a description of an abstract state space that is easier to search; exact distances in the abstract space give estimates of distances in the original space (Culberson & Schaeffer 1996; Haslum & Geffner 2000; Bonet & Geffner 2001). One limitation of this approach is that it is often memory-intensive. This has led to the study of compression schemes (Edelkamp 2002; Samadi *et al.* 2008), disk-based methods (Zhou & Hansen 2005), and distributed methods (Edelkamp, Jabbar, & Kissmann 2009). These methods extend the range of problems to which abstraction is applicable, but since combinatorial problems grow in size exponentially it is easy to imagine problems so large that, with the computers of the foreseeable future, even the best heuristics created by these systems will be too weak to enable arbitrary instances to be solved reasonably quickly.

A second limitation of abstraction is that it can only be applied to state spaces given in a suitable declarative form. Sometimes there is no such state-space description, for example, if a planner is controlling a system or computer game, or when such a description would be vastly less efficient than a "hard-coded" one, or when the state space is described declaratively but in a different language than the abstraction system requires. We call such representations opaque. With an opaque representation, a state space is defined by a successor function that can be called to compute a state's children but cannot otherwise be reasoned about. By definition, abstraction cannot be applied to create heuristics when the state space is represented opaquely.

A way of creating heuristics that sidesteps these limitations is to apply machine learning to a set of states with known distance-to-goal (the training set) to create a function that estimates distance-to-goal for an arbitrary state, *i.e.*, a heuristic function. This idea has been applied with great success to the 15-puzzle and other state spaces of similar size (Ernandes & Gori 2004; Samadi, Felner, & Schaeffer 2008) but could not be applied to larger spaces, *e.g.*, the 24-puzzle, because of the infeasibility of creating a sufficiently large training set containing a sufficiently broad range of possible distances to goal. To overcome this obstacle, (Samadi, Felner, & Schaeffer 2008) reverted to the abstraction approach: instead of learning a heuristic for the 24-puzzle directly they learned heuristics for two disjoint abstractions of the 24-puzzle and added them to get a heuristic for the 24-puzzle. This approach inherits the limitations of abstraction mentioned above and, in addition, is not fully automatic: the crucial choices of which abstractions to use and how to combine them are made manually.

Ernandes and Gori (2004) proposed a different way of extending the machine learning approach to scale to arbitrarily large problems, but never implemented it. We call this approach "bootstrap learning of heuristic functions" (bootstrapping, for short). The contribution of the present paper is to validate their proposal by supplying the details required to make bootstrapping practical and fully automatic and showing experimentally that it succeeds, without modification or manual intervention, on both small (*e.g.*, the 15-puzzle) and large problems (*e.g.*, the 24-puzzle). The measure of success of our system is not that it produces better heuristics than previous methods but rather that it produces heuristics that are almost as good from a much weaker starting point.

Bootstrapping is an iterative procedure that uses learning to create a series of heuristic functions. Initially, this procedure requires a (weak) heuristic function h_0 and a set of

states we call the bootstrap instances. Unlike previous machine learning approaches, there are no solutions given for any instances, and h_0 is not assumed to be strong enough to solve any of the given instances. A heuristic search algorithm is run with h_0 in an attempt to solve the bootstrap instances within a given time limit. The set of solved bootstrap instances, together with their solution lengths (not necessarily optimal), is fed to a learning algorithm to create a new heuristic function h_1 that is intended to be better than h_0. After that, the previously unsolved bootstrap instances are used in the same way, using h_1 as the heuristic instead of h_0. This procedure is repeated until all but a handful of the bootstrap instances have been solved or until a succession of iterations fails to solve a large enough number of "new" bootstrap instances (ones that were not solved previously).

If the initial heuristic h_0 is too weak to solve even a few of the given bootstrap instances within the given time limit we enhance h_0 by a random walk method that automatically generates bootstrap instances at the "right" level of difficulty (easy enough to be solvable with h_0, but hard enough to yield useful training data for improving h_0).

As in the earlier studies (Ernandes & Gori 2004; Samadi, Felner, & Schaeffer 2008), which may be seen as doing one step of the bootstrap process with a very strong initial heuristic, the learned heuristic might be inadmissible, *i.e.*, it might sometimes overestimate distances, and therefore IDA* is not guaranteed to find optimal solutions with the learned heuristic. With bootstrapping, the risk of excessive suboptimality of the generated solutions is much higher than with the one-step methods because on each iteration the learning algorithm might be given solution lengths larger than optimal, biasing the learned heuristic to even greater overestimation. The suboptimality of the solutions generated is hence an important performance measure in our experiments.

We test our method experimentally on six problem domains, the 15- and 24-sliding-tile puzzles, the 17- and 24-pancake puzzles, and the 15- and 20-blocks world, in each case (except the 15-puzzle) starting with an initial heuristic so weak that the previous, one-step methods would fail because they would not be able to generate an adequate training set in a reasonable amount of time. In all the domains, bootstrapping succeeds in producing a heuristic that allows IDA* to solve randomly generated problem instances extremely quickly with solutions that are very close to optimal. On these domains our method systematically outperforms Weighted IDA* and BULB (Furcy & König 2005).

Method and implementation

The input to our system consists of a state space, a fixed goal state g, a heuristic function h_{in}, and a set Ins of states to be used as bootstrap instances. We do not assume that h_{in} is sufficiently strong that any of the given bootstrap instances can be solved using it. Our bootstrap procedure, Algorithm 1, proceeds in two stages.

In the first stage, for every instance i in Ins, a heuristic search algorithm is run with start state i and h_{in} as its heuristic (line 6). Every search is cut off after a limited period of time ($\mathrm{t}_{\max}$). If i is solved within that time then some user-defined features of i, together with its solution length, are added to the training set. In addition, features and solution lengths for all the states on the solution path for i are added to the training set (lines 7 and 8). This increases the size of the training set at no additional cost and balances the training set to contain instances with long and short solutions.

Algorithm 1

```
 1: procedure Bootstrap(h_in, Ins): h_out
 2: uses global variables t_max, t_∞, ins_min, g
 3: create an empty training set TS
 4: while (size(Ins) ≥ ins_min) && (t_max ≤ t_∞) do
 5:    for each instance i ∈ Ins do
 6:       if Heuristic Search(i, g, h_in, t_max) succeeds then
 7:          for each state s on i's solution path do
 8:             Add (feature vector(s), distance(s,g)) to TS
 9:          end for
10:       end if
11:    end for
12:    if (#(new bootstrap instances solved) > ins_min) then
13:       h_in := learn a heuristic from TS
14:       remove the solved instances from Ins
15:       clear TS
16:    else
17:       t_max := 2 × t_max
18:    end if
19: end while
20: return h_in
```

The second stage examines the collected training data. If "enough" bootstrap instances have been solved then the heuristic h_{in} is updated by a learning algorithm (line 13). Otherwise the time limit is increased without changing h_{in} (line 17). Either way, as long as the current time limit ($\mathrm{t}_{\max}$) does not exceed a fixed upper bound (t_{∞}), the bootstrap procedure is repeated on the remaining bootstrap instances with the current heuristic h_{in}. "Enough" bootstrap instances here means a number of instances above a fixed threshold $\mathrm{ins}_{\min}$ (line 12). The procedure terminates if $\mathrm{t}_{\max}$ exceeds t_{∞} or if the remaining set of bootstrap instances is too small.

There are no strong requirements on the set Ins of bootstrap instances – it may be any set representative of the instances of interest to the user. However, for the bootstrap process to incrementally span the gap between the easiest and hardest of these instances, Ins must contain instances at intermediate levels of difficulty. At present this is an informal requirement for which we have no proof of necessity.

It can happen that the initial heuristic h_{in} is so weak that the heuristic search algorithm is unable to solve enough instances in Ins, using h_{in}, to get a sufficiently large set of training data. For this case, we need a procedure that generates bootstrap instances that are (i) easier to solve than the instances the user provided but (ii) harder to solve than instances solvable by simple breadth-first search in acceptable time (to guarantee a high enough quality of training data).

This is done using random walks backwards from the goal[1] of a suitably chosen length to generate instances. As

[1]Given uninvertible operators, this needs a predecessor func-

Algorithm 2 In a random walk we always disallow the inverse of the previous move.

1: **procedure RandomWalk** (h_{in}, Ins, $length_{++}$): h_{out}
2: **uses global variables** $t_{\max}$, t_∞, $ins_{\min}$, g
3: length := $length_{++}$
4: **while** (h_{in} is too weak to start the Bootstrap process on Ins) && ($t_{\max} \leq t_\infty$) **do**
5: RWIns := generate instances by applying "length" many random moves backward from the goal
6: h_{in} := **Bootstrap**(h_{in}, RWIns)
7: length := length + $length_{++}$
8: **end while**
9: **return Bootstrap**(h_{in}, Ins)

described in Algorithm 2, we first test whether the initial heuristic is strong enough to solve a sufficient number of the user-provided instances (line 4). If so, the bootstrap procedure can be started immediately (line 9). Otherwise, we perform random walks backward from the goal, up to depth "length", and collect the final states as special bootstrap instances (RWIns). The bootstrap procedure is run on these special instances (line 6) to create a stronger heuristic. This process is repeated with increasingly longer random walks (line 7) until it produces a heuristic strong enough for bootstrapping to begin on the user-given instances or fails to produce a heuristic with which sufficiently many instances in RWIns can be solved within time limit t_∞.

The choice of "$length_{++}$" is an important consideration. If it is too large, the instances generated may be too difficult for the current heuristic to solve and the process will fail. If it is too small, a considerable amount of time will be wasted applying the bootstrap process to instances that do not substantially improve the current heuristic. In the next section we describe a method for automatically choosing an appropriate value for "$length_{++}$".

Experiments

Our experiments ran on a 2.6 GHz computer with 32GB of RAM. Except where explicitly stated otherwise, IDA* was the search algorithm used. We never attempted fine-tuning of any of the settings used in our experiments.

Domains. Although our motivation for creating the bootstrap method was to have a technique that could be applied when existing techniques could not—either because of memory limits, having only an opaque representation, or there being no feasible way to create a training set good enough for the one-step methods to succeed—the domains we have chosen for our experiments do not have these properties. There are two reasons for this. First, it is very important in this study to be able to determine the suboptimality of the solutions our method produces. This requires choosing domains in which optimal solution lengths can be computed in a reasonable amount of time, which typically requires that the domain is solvable by existing heuristic search methods or that there exists a hand-crafted optimal solver for the domain. The second reason for our choice of domains is the desire to compare the solutions produced by bootstrapping, with a very weak initial heuristic, with those produced by existing methods when they are given much stronger heuristics. This necessarily entails giving bootstrapping an artificially weak starting point. We will consider bootstrapping successful if the heuristics it produces, given its much weaker starting point, are almost as good as those produced by existing systems.

Based on these considerations, we chose the following domains for our experiments. (i) *sliding-tile puzzle*. We used the 15- and 24-puzzles. (ii) *n-pancake puzzle* (Dweighter 1975) – In the n-pancake puzzle, a state is a permutation of n numbered tiles and has $n-1$ successors, with the l^{th} successor formed by reversing the order of the first $l+1$ positions of the permutation. $n = 17$ and $n = 24$ were used in our experiments. (iii) *n-blocks world* (Slaney & Thiébaux 2001) – We used the 3-action handless blocks world with $n = 15$ and $n = 20$ blocks.

Learning algorithm and features. The learning algorithm used in all experiments was a neural network (NN) with one output neuron representing distance-to-goal and three hidden units trained using backpropagation and mean squared error (MSE). Training ended after 500 epochs or when $MSE < 0.005$. It is well known that the success of any machine learning application depends on having "good" features. The issue of automatically creating good features for learning search control knowledge has been studied, in the context of planning by (Yoon, Fern, & Givan 2008). In our experiments we did not carefully engineer the features used, exploit special properties of the domain, or alter our features on the basis of early experimental outcomes or as we scaled up the problem. The input features for the NN are described separately for each domain below; most of them are values of weak heuristics for the respective problems.

Initial heuristics. The initial heuristic h_0 for each domain was defined as the maximum of the heuristics used as features for the NN. After each iteration of our method, the new heuristic was defined as the maximum over the output of the NN and the initial heuristic.

Bootstrap instances. Ins always consisted of either 500 or 5000 solvable instances generated uniformly at random.

Numeric parameters. In all experiments, $ins_{\min} = 75$, $t_{\max} = 1$sec, $t_\infty = 512$sec, and the size of the set RWIns was 200. For RandomWalk, $length_{++}$ was set as follows.

1. Run a breadth-first search backward from the goal state with a time limit given by the initial value of $t_{\max}$. Let S be the set of states thus visited.
2. Repeat 5000 times: Do a random walk backwards from the goal until a state not in S is reached. Set $length_{++}$ to be the floor of the average length of these random walks.

The tables below summarize the results on our test domains, which are based on a set of test instances generated independently of the bootstrap instances. In the tables, the column "h (Algorithm)" denotes the heuristic used; the search algorithm, if different from IDA*, is given in parentheses. The symbol #k indicates that the same heuristic is used in this row as in row k. "Subopt" indicates the subop-

tion, not just the successor function from an opaque representation.

timality of the solutions found (Subopt=7% means the solutions generated were 7% longer than optimal on average). "Nodes" is the average number of nodes generated to solve the test instances. "Time" is the average search time (in seconds) to solve the test instances. Unless specifically stated, no time limit was imposed when systems were solving the test instances. Each row gives the data for a system that we tried or found in the literature. The run-times taken from the literature are marked with an asterisk to indicate they may not be strictly comparable to ours. All weighted IDA* (W-IDA*, W is the weight by which h is multiplied) and BULB results are for our own implementations. Hyphenated row numbers "row-y" indicate Bootstrap results after iteration y. The last Bootstrap iteration shown represents the termination of the Bootstrap process, either due to having fewer than $\text{ins}_{\min}$ many instances left (15-puzzle, 17- and 24-pancake puzzle, 15-blocks world and the 20-blocks world using 5000 bootstrap instances) or due to $\text{t}_{\max}$ exceeding t_∞ (24-puzzle, 20-blocks world with 500 bootstrap instances).

15-puzzle

For the 15-puzzle the input features for the NN were Manhattan distance (MD), number of out-of-place tiles, position of the blank, number of tiles not in the correct row, number of tiles not in the correct column, and five heuristics, each of which is the maximum of two 4-tile pattern databases (PDBs, (Culberson & Schaeffer 1996)). Except for the last row in Table 2, all the results in Tables 1 and 2 are averages over the standard 1000 15-puzzle test instances (Korf & Felner 2002), which have an average optimal cost of 52.52.

Table 1 shows the results for bootstrapping on the 15-puzzle. The initial heuristic (h_0) was sufficient to begin the Bootstrap process directly, so no random walk iterations were necessary. Row 1 shows the results when h_0 is used by itself as the final heuristic. It is included to emphasize the speedup produced by Bootstrapping. The next three rows (2-0 to 2-2) show the results for the heuristic created on each iteration of the Bootstrap method when it is given 500 bootstrap instances. The next four rows (3-0 to 3-3) are analogous, but when 5000 bootstrap instances are given. In both cases, there is a very clear trend: search becomes faster in each successive iteration (see the Nodes and Time columns) but suboptimality becomes worse. In either case, bootstrapping produces very substantial speedup over search using h_0. For instance, using 500 bootstrap instances produces a heuristic in 11 minutes that makes search more than 4000 times faster than with h_0 while producing solutions that are only 4.5% (2.4 moves) longer than optimal.

There are two key differences between using 500 and 5000 bootstrap instances. The most obvious, and in some settings by far the most important, is the total time required for the bootstrap process. Because after every iteration an attempt is made to solve every bootstrap instance, having 10 times as many bootstrap instances makes the process roughly 10 times slower. The second difference is more subtle. The larger bootstrap set contains a larger number of more difficult problems, and those drive the bootstrap process through additional iterations (in this case one additional iteration), producing, in the end, faster search but worse suboptimality than when fewer bootstrap instances are used. There is clearly a rich set of time-suboptimality tradeoffs inherent in the bootstrap approach. In this paper we do not address the issue of how to choose among these options, we assume that a certain number of bootstrap instances are given and that the heuristic produced by the final bootstrap iteration is the system's final output.

row	iteration	Subopt	Nodes	Time
1	h_0	0%	132,712,521	116.478
500 bootstrap instances Total Time to create the final heuristic = 11m				
2-0	0 (first)	1.1%	422,554	0.424
2-1	1	2.7%	76,928	0.075
2-2	2 (final)	4.5%	32,425	0.041
5000 bootstrap instances Total Time to create the final heuristic = 1h 52m				
3-0	0 (first)	1.2%	388,728	0.379
3-1	1	2.6%	88,235	0.087
3-2	2	5.4%	21,800	0.022
3-3	3 (final)	**7.7%**	**10,104**	0.010

Table 1: 15-puzzle, Bootstrap.

Table 2 shows the results of other systems applied to the same test instances (except for the last row). Rows 1 through 4 are when our initial heuristic (h_0) is used with W-IDA* and BULB. Both algorithms are dominated by Bootstrap, *i.e.*, if W and B (BULB's beam width) are set so that W-IDA* and BULB compare to Bootstrap in either one of the values Subopt or Nodes, then the heuristic obtained in the final Bootstrap iteration (Table 1, Row 3-3) is superior in the other value.

row	h (Algorithm)	Subopt	Nodes	Time
1	h_0 (W-IDA*,W=1.6)	**8.9%**	310,104	0.194
2	h_0 (W-IDA*,W=2.3)	41.9%	**10,734**	0.007
3	h_0 (BULB,B=250)	**7.7%**	26,013	0.017
4	h_0 (BULB,B=90)	12.7%	**10,168**	0.006
Results from previous papers				
5	Add 7-8	0%	136,289	0.063*
6	#5 + reflected lookup	0%	36,710	0.027*
7	#6 + dual lookup	0%	18,601	0.022*
8	NN using tile positions +#6+MD (RBFS)	3.3%	2,241	0.001*
9	PE-ANN version of #8 (RBFS)	0.2%	16,654	0.014*
10	NN "A"	3.5%	24,711	7.380*

Table 2: 15-puzzle, other methods.

The next three rows in Table 2 show state-of-the-art results for optimally solving the 15-puzzle. Rows 5 and 6 refer to (Korf & Felner 2002), where the maximum of two disjoint 7-8 PDBs, and their reflections across the main diagonal are the heuristics. Row 7 is from (Felner *et al.* 2005), where the heuristic is as in Row 6, augmented with dual lookup (in both the regular and the reflected PDB). Bootstrap with 5000 bootstrap instances (Table 1, Row 3-3) outperforms all of these systems in terms of Nodes and Time.

The last three rows of Table 2 show state-of-the-art results

for the one-step heuristic learning systems described in the introduction. Rows 8 and 9 are taken from (Samadi, Felner, & Schaeffer 2008). Row 8 uses the tile positions as features for the NN along with the heuristic from Row 6 and Manhattan Distance. Row 9 is the same as Row 8 but using a modified error function during the neural net training to penalize overestimation. This drives suboptimality almost to 0, at the cost of substantially increasing the search time. Row 10 shows the results for the NN in (Ernandes & Gori 2004) that generated the fewest nodes ("A"). These are averages over 700 random instances with an average optimal solution length of 52.62, not over the 1,000 test instances used by all other results in this section. The NN input features are the positions of the tiles and the initial heuristic.

Bootstrap with 5000 bootstrap instances (Table 1, Row 3-3) outperforms all of these systems in terms of Nodes and Time except for Row 8, which also outperforms Bootstrap in terms of suboptimality. To prove that its superior performance is simply the result of having vastly stronger heuristics as input features to the NN, we reran our Bootstrap system with 5000 bootstrap instances, exactly as described above, but with the following features replacing the weak PDB features used above: the value of the 7- and 8-tile additive PDBs individually, for both the given state and its reflection, and the maximum of the sum of 7- and 8-tile PDB values for the state and its reflection. With these input features, and the corresponding h_0, Bootstrap solved all 5000 bootstrap instances on its first iteration, and the heuristic it produced, when used with RBFS, solved the 1000 test instances with an average suboptimality of 0.5% while generating only 9,402 nodes on average. We thus see that Bootstrap, when given a heuristic that is strong enough that it can solve all bootstrap instances in the given time limit is equivalent to the one-step systems previously reported. But, as was seen in Table 1, its iterative RandomWalk and Bootstrap processes take it beyond the capabilities of those systems by enabling it to perform very well even when the initial heuristic is not strong enough to solve the bootstrap instances.

24-puzzle

Table 3 shows our results on the 50 standard 24-puzzle test instances (Korf & Felner 2002), which have an average optimal cost of 100.78. The input features for the NN are the same as for the 15-puzzle. Note that here 4-tile PDBs, though requiring more memory, are much weaker than for the 15-puzzle. The initial heuristic is sufficiently weak that eight RandomWalk iterations were necessary before bootstrapping itself could begin (ten iterations were required when there were only 500 bootstrap instances). The trends in these results are those observed for the 15-puzzle: (a) search becomes faster in each successive iteration but suboptimality becomes worse; and (b) having more bootstrap instances is slower and results in extra bootstrap iterations.

Table 4 shows the results of other systems on the same test instances. Row 1 reports on W-IDA* for a weight with which a number of nodes comparable to that using the final bootstrap heuristic is achieved. Even if allowed 10 times more time per instance than used by Bootstrap in the final iteration, W-IDA* could not compete in terms of suboptimality. Rows 2 and 3 are when our initial heuristic (h_0) is used with BULB. Bootstrap (Table 3, Row 2-10) dominates in all cases.

row	iteration	Subopt	Nodes	Time
500 bootstrap instances Total Time to create the final heuristic = 2 days				
1-0	0 (first)	5.1%	2,195,190,123	4,987.10
1-1	1	5.7%	954,325,546	2,134.78
1-3	3 (final)	6.1%	164,589,698	273.51
5000 bootstrap instances Total Time to create the final heuristic = 18 days				
2-0	0 (first)	5.4%	1,316,197,887	2,439.72
2-4	4	6.9%	115,721,236	214.59
2-7	7	7.9%	29,956,637	55.85
2-10	10 (final)	**9.6%**	**5,221,203**	9.83

Table 3: 24-puzzle, Bootstrap.

row	h (Algorithm)	Subopt	Nodes	Time
1	h_0 (W-IDA*,W=2.6)	80.3%	**5,849,910**	4.3
2	h_0 (BULB,B=18000)	**10.5%**	6,896,038	22.3
3	h_0 (BULB,B=16000)	12.0%	**5,286,874**	18.2
Results from previous papers				
4	Add 6-6-6-6	0%	360,892,479,670	47 hours*
5	#4 (DIDA*)	0%	75,201,250,618	10 hours*
6	#4, Add 8-8-8	0%	65,135,068,005	?
7	#4, W=1.4 (RBFS)	9.4%	1,400,431	1.0*
8	PE-ANN, Add 11-11-2 (RBFS)	0.7%	118,465,980	111.0*
9	#8, W=1.2 (RBFS)	3.7%	582,466	0.7*

Table 4: 24-puzzle, other methods.

Rows 4 to 6 show the results of state-of-the-art heuristic search methods for finding optimal solutions. Row 4 shows the results using the maximum of disjoint 6-tile PDBs and their reflections across the main diagonal as a heuristic (Korf & Felner 2002). Row 5 shows the results for DIDA* (Zahavi *et al.* 2006) using the same heuristic. (Felner & Adler 2005) compute the maximum of this heuristic and a disjoint 8-tile PDB (partially created), see Row 6 (Time was not reported). The very large time required by these systems shows that the 24-puzzle represents the limit for finding optimal solutions with today's abstraction methods and memory sizes. Row 7 (Samadi, Felner, & Schaeffer 2008) illustrates the benefits of allowing some amount of suboptimality, for RBFS with the heuristic from Row 4 multiplied by 1.4. Although these results are better, in terms of Nodes and Time, than Bootstrap (Table 3, Row 2-10), they hinge upon having a very strong heuristic since we have just noted that W-IDA* with our initial heuristic is badly outperformed by Bootstrap.

Rows 8 and 9 show the PE-ANN results (Samadi, Felner, & Schaeffer 2008). As discussed in the introduction, this is not the same system as was applied to the 15-puzzle because it was infeasible to generate an adequate training set for a one-step method. Critical choices for abstracting the 24-puzzle were made manually to obtain these results. Row 8 shows that our fully automatic method (exactly the same as was applied to the 15-puzzle) is superior to this PE-ANN by a factor of more than 20 in terms of nodes generated, al-

though it is inferior in terms of suboptimality. Row 9 shows that if PE-ANN's learned heuristic is suitably weighted it can outperform Bootstrap in both Nodes and Subopt.

17-pancake puzzle

For the 17-pancake puzzle the input features for the NN were six 5-token PDBs, a binary value indicating whether the middle token is out of place, and the number of the largest out-of-place token. All the results in Tables 5 and 6 are averages over the 1000 test instances used in (Yang *et al.* 2008), which have an average optimal solution length of 15.77. Optimal solution lengths were computed using the highly accurate, hand-crafted "break" heuristic.[2]

Table 5 shows the results for bootstrapping on the 17-pancake puzzle. The initial heuristic (h_0) was too weak for us to evaluate it on the test instances in a reasonable amount of time. The first two rows (1-0 and 1-1) show the results for the heuristic created on each iteration of the Bootstrap method using 500 bootstrap instances. The next three rows (2-0 to 2-2) are analogous, but when 5000 bootstrap instances are given. In both cases, we see the same trends as in the 15-puzzle: (a) search becomes faster in each successive iteration but suboptimality becomes worse; and (b) having more bootstrap instances is slower and results in extra bootstrap iterations. Note that a suboptimality of 7% here means the solutions generated are only 1.1 moves longer than optimal. When there were 5000 bootstrap instances h_0 was able to solve enough bootstrap instances to begin the Bootstrap process directly, but when there were only 500 two iterations of the RandomWalk process were needed before bootstrapping on the bootstrap instances could begin.

row	iteration	Subopt	Nodes	Time
500 bootstrap instances Total Time to create the final heuristic = 26m				
1-0	0 (first)	3.1%	253,964	0.112
1-1	1 (last)	4.6%	75,093	0.034
5000 bootstrap instances Total Time to create the final heuristic = 2h 23m				
2-0	0 (first)	2.4%	15,232,606	6.700
2-1	1	4.8%	76,346	0.034
2-2	2 (final)	**7.1%**	**30,341**	0.014
5000 bootstrap instances + duality Total Time to create the final heuristic = 2h 4m				
3-0	0 (first)	2.1%	14,631,867	30.790
3-1	1	4.2%	57,119	0.111
3-2	2 (final)	**7.0%**	**7,555**	0.009

Table 5: 17-pancake puzzle, Bootstrap.

The final three rows in Table 5 examine the effect of infusing into the bootstrapping process the domain-specific knowledge that for every pancake puzzle state s there exists another state s^d, called the dual of s, that is the same distance from the goal as s (Zahavi *et al.* 2006). To exploit this knowledge, when training the NN for every training example (s,$cost$) we added an additional training example (s^d,$cost$), and when calculating a heuristic value for s we took the maximum of the values produced by the NN when it was applied to s and to s^d. The initial heuristic here was given by the maximum over the heuristic values occurring in the feature vectors for s and s^d. A comparison of Rows 3-2 and 2-2 shows that the additional knowledge substantially reduced search time without affecting suboptimality.

row	h (Algorithm)	Subopt	Nodes	Time
1	h_0 (W-IDA*,W=2)	**7.1%**	20,949,730	9.800
2	h_0 (W-IDA*,W=8)	201.5%	**8,650**	0.004
3	h_0 (BULB,B=5000)	**7.5%**	955,015	0.715
4	h_0 (BULB,B=10)	155.5%	**11,005**	0.008
Results from previous papers				
5	Add 3-7-7	0%	1,061,383	0.383*
6	#5 + dual lookup	0%	52,237	0.036*
7	#6 (DIDA*)	0%	37,155	0.026*

Table 6: 17-pancake puzzle, other methods.

Table 6 shows the results of other systems. Rows 1 to 4 are when our initial heuristic (h_0) and duality is used with W-IDA* and BULB. As was the case for the 15-puzzle, both algorithms are dominated by Bootstrap (Table 5, Row 3-2). Rows 5 to 7 are for the state-of-the-art optimal heuristic search methods that have been applied to the 17-pancake puzzle. Rows 5 and 6 (Yang *et al.* 2008) use a set of three additive PDBs, without and with dual lookups. Row 7 uses dual lookups and dual search (Zahavi *et al.* 2006). In terms of Nodes and Time, Bootstrap outperforms these systems even without exploiting duality (Table 5, Row 2-2).

24-pancake puzzle

The experimental setup for the 24-pancake puzzle is identical to that for the 17-pancake puzzle, but note that a 5-token PDB is a much weaker heuristic when there are 24 pancakes. 1000 randomly generated instances, with an average optimal solution cost of 22.75, were used for testing. The initial heuristic is so weak that four RandomWalk iterations were necessary before bootstrapping itself could begin. Table 7 is analogous to Table 5, with rows for selected iterations with 500 bootstrap instances, 5000 bootstrap instances, and 5000 bootstrap instances with duality exploited. All the trends seen in previous domains are evident here.

No previous system has been applied to this problem domain, so Table 8 includes results only for W-IDA* and BULB. The results shown are when those algorithms use duality, so the appropriate comparison is with Row 3-6 of Table 7. Neither algorithm was able to achieve a "Nodes" value similar to Bootstrap, so the table just shows the minimum number of nodes these two algorithms generated (we tried 15 values for W between 1.1 and 10, and 15 values for B between 2 and 20,000). As can be seen, W-IDA* and BULB produce a very high degree of suboptimality when generating the fewest nodes and are therefore not competitive with Bootstrap. Looking for settings for which W-IDA* or BULB can compete with Bootstrap in terms of suboptimality was not successful. Allowing 10 times more time than IDA* with Bootstrap's final heuristic needed on each test instance, W-IDA* did not complete any instances at all and BULB completed too few to allow for a comparison.

[2]For details on "break" see http://tomas.rokicki.com/pancake/

row	iteration	Subopt	Nodes	Time
500 bootstrap instances Total Time to create the final heuristic = 1h 19m				
1-0	0 (first)	8.1%	9,502,753	9.70
1-2	2	9.3%	3,189,610	3.20
1-4	4 (final)	10.3%	1,856,645	1.89
5000 bootstrap instances Total Time to create the final heuristic = 15h				
2-0	0 (first)	6.9%	24,488,908	32.68
2-4	4	8.3%	4,389,271	5.65
2-8	8	10.2%	1,547,765	2.19
2-12	12	11.2%	1,285,021	1.82
2-16	16 (final)	12.1%	770,999	0.80
5000 bootstrap instances + dual lookup Total Time to create the final heuristic = 9h				
3-0	0 (first)	8.2%	3,345,657	7.08
3-2	2	10.9%	445,420	0.93
3-4	4	12.1%	226,475	0.37
3-6	6 (final)	**14.1%**	**92,098**	0.16

Table 7: 24-pancake puzzle, Bootstrap.

row	h (Algorithm)	Subopt	Nodes	Time
1	h_0 (W-IDA*,W=8)	130.1%	2,128,702	1.88
2	h_0 (BULB,B=10)	2726.7%	267,017	1.10

Table 8: 24-pancake puzzle, other methods.

15-blocks world

For the 15-blocks world we used 200 random test instances in which the goal state has all the blocks in one stack. Their average optimal solution length is 22.73. We used 9 input features for the NN: seven 2-block PDBs, the number of out of place blocks, and the number of stacks of blocks. Optimal solutions were computed using the hand-crafted blocks world solver PERFECT (Slaney & Thiébaux 2001).

Table 9 shows the bootstrap results. The initial heuristic is so weak that three RandomWalk iterations were needed before bootstrapping. The trends are, again, (a) search is sped up in each iteration but suboptimality increases; and (b) having more bootstrap instances is slower and requires more bootstrap iterations. A suboptimality of 7% here means the solutions generated are about 1.6 moves longer than optimal.

row	iteration	Subopt	Nodes	Time
500 bootstrap instances Total Time to create the final heuristic = 1h 46m				
1-0	0 (first)	1.5%	1,157,510,765	2,656.99
1-1	1	2.8%	554,160,659	2,149.07
1-2	2	4.7%	21,289,247	69.37
1-3	3 (final)	**6.9%**	**3,651,438**	15.87
5000 bootstrap instances Total Time to create the final heuristic = 8h				
2-0	0 (first)	1.6%	2,253,260,711	5,081.57
2-3	3	3.9%	44,616,679	101.19
2-6	6	6.7%	3,468,436	7.65
2-9	9 (final)	**7.3%**	**155,813**	0.35

Table 9: 15-blocks world (1-stack goal), Bootstrap.

Table 10 shows the results of BULB using our initial heuristic, which is again dominated by Bootstrap (Table 9, Row 2-9). An attempt to compare with W-IDA* on the 1-stack goal failed due to the poor performance of W-IDA* with time limits 10 times larger than the solving time using Bootstrap's final heuristic for each test instance. Varying the weights between 1.2 and 10, W-IDA* never solved more than about 70% of the instances (W=8 was best).

row	h (Algorithm)	Subopt	Nodes	Time
1	h_0 (BULB,B=4000)	**7.3%**	972,380	2.09
2	h_0 (BULB,B=500)	24.4%	**177,187**	0.46

Table 10: 15-blocks world, other methods.

We compare our solution quality (Table 9, Row 2-9) to three hand-crafted suboptimal solvers for the blocks world, US, GN1, and GN2 (Gupta & Nau 1992). With an average solution length of 24.4, Bootstrap performed almost as well as GN1 (23.88) and GN2 (22.83), and slightly better than US (25.33), even though US by construction should work well on 1-stack goal problems.

20-blocks world

The experimental setup for 20 blocks was identical to that for 15 blocks, but here a 2-block PDB is a much weaker heuristic than for 15 blocks. We used 50 random test instances with an average optimal solution length of 30.92. The initial heuristic is so weak that six RandomWalk iterations were necessary before bootstrapping (eight iterations for 500 bootstrap instances). The trends across bootstrap iterations are those observed in all previous experiments.

row	iteration	Subopt	Nodes	Time
500 bootstrap instances Total Time to create the final heuristic = 2 days				
1-1	1	1.5%	13,456,726,519	55,213
1-3	3 (final)	**3.6%**	**615,908,785**	2,763
5000 bootstrap instances Total Time to create the final heuristic = 11 days				
2-3	3	2.1%	12,771,331,089	52,430
2-5	5	3.3%	941,847,444	3,828
2-7	7	3.9%	789,515,580	3,240
2-10	10	7.6%	11,347,282	47
2-13	13 (final)	**9.2%**	**5,523,983**	23

Table 11: 20-blocks world (1-stack goal), Bootstrap.

Bootstrap with an average solution length of 33.78 (Table 11, Row 2-13) is again somewhat inferior to GN1 (32.54) and GN2 (30.94), and slightly better than US (34.58).

Similar to the 15-blocks world, W-IDA* with time limits 10 times larger than the solving time using Bootstrap's final heuristic for each test instance failed to solve more than half the test instances (W was varied between 1.2 and 10). In the best case (W=9) W-IDA* solved 24 of the test instances.

For suboptimality, BULB could not compete with Bootstrap (Table 11, Row 2-13); we tried 15 values for B between 2 and 20,000. For nodes, BULB (h_0, B=2,400) achieved values of 109.6% (Subopt), **5,809,791** (Nodes), and 32 (Time).

Related work

Bootstrap learning to iteratively improve an initially weak evaluation function for single-agent search is an idea due to Rendell (1983). Our method differs from his in some key details. Most importantly, Rendell required the user to provide at least one bootstrap instance per iteration that was solvable with the current evaluation function. We assume all bootstrap instances are given at the outset, and if the system cannot solve any of them it generates its own.

The only other study of bootstrap learning of heuristics is due to Humphrey, Bramanti-Gregor, and Davis (1995). Their system SACH learns a heuristic to solve a single instance, while bootstrapping over successive failed attempts.

A related approach is online learning of heuristics as studied by Fink (2007). Fink proves his learning algorithm has certain desirable properties, but it has the practical shortcoming that it requires optimal solution lengths to be known for all states that are generated during all of the searches.

Two previous systems have used random walks to generate successively more difficult instances to bootstrap the learning of search control knowledge in a form other than a heuristic function. Fern et al. (2004) used random walks in learning policies to control a Markov Decision Process, and Finkelstein and Markovitch (1998) used them in the context of learning macro-operators to augment a heuristic-guided hill-climbing search. In both cases the initial random walk length and the increment were user-specified.

Conclusions

This paper gives experimental evidence that machine learning can be used to create strong heuristics from very weak ones through a fully automatic, incremental bootstrapping process augmented by a random walk method for generating successively more difficult problem instances. Our system was tested on small and large versions of three different problem domains and successfully created heuristics that enable IDA* to solve randomly generated test instances very quickly and almost optimally. The total time needed for our system to create these heuristics strongly depends on the number of bootstrap instances it is given. Using 500 bootstrap instances, heuristics are produced approximately 10 times faster than using 5000 bootstrap instances. Search is slower with the heuristics produced using fewer bootstrap instances, but the solutions found are closer to optimal. This work significantly extends previous, one-step methods that fail unless they are given a very strong heuristic to start with.

Acknowledgements

Thanks to M. Samadi and R. Valenzano for sharing their code, reviewers for their insightful comments, the Alberta Ingenuity Centre for Machine Learning, and NSERC.

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artif. Intell.* 129:5–33.

Culberson, J. C., and Schaeffer, J. 1996. Searching with pattern databases. *Advances in Artificial Intelligence (LNAI 1081)* 402–416.

Dweighter, H. 1975. Problem E2569. *American Mathematical Monthly* 82:1010.

Edelkamp, S.; Jabbar, S.; and Kissmann, P. 2009. Scaling search with pattern databases. In *MOCHART*, volume 5348 of *LNCS*, 49–64. Springer.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *AIPS*, 274–283.

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

Felner, A., and Adler, A. 2005. Solving the 24 puzzle with instance dependent pattern databases. In *SARA*, volume 3607 of *LNCS*, 248–260. Springer.

Felner, A.; Zahavi, U.; Schaeffer, J.; and Holte, R. C. 2005. Dual lookups in pattern databases. In *IJCAI*, 103–108.

Fern, A.; Yoon, S.; and Givan, R. 2004. Learning domain-specific control knowledge from random walks. In *ICAPS*, 191–199. AAAI Press.

Fink, M. 2007. Online learning of search heuristics. *AISTATS* 114–122.

Finkelstein, L., and Markovitch, S. 1998. A selective macro-learning algorithm and its application to the nxn sliding-tile puzzle. *J. Artif. Intell. Res.* 8:223–263.

Furcy, D., and König, S. 2005. Limited discrepancy beam search. In *IJCAI*, 125–131.

Gupta, N., and Nau, D. S. 1992. On the complexity of blocks-world planning. *Artif. Intell.* 56:223–254.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *AIPS*, 140–149.

Humphrey, T.; Bramanti-Gregor, A.; and Davis, H. W. 1995. Learning while solving problems in single agent search: Preliminary results. In *AI*IA*, 56–66. Springer.

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

Rendell, L. A. 1983. A new basis for state-space learning systems and a successful implementation. *Artif. Intell.* 20:369–392.

Samadi, M.; Siabani, M.; Felner, A.; and Holte, R. 2008. Compressing pattern databases with learning. In *ECAI*, 495–499.

Samadi, M.; Felner, A.; and Schaeffer, J. 2008. Learning from multiple heuristics. *AAAI* 357–362.

Slaney, J., and Thiébaux, S. 2001. Blocks world revisited. *Artif. Intell.* 125:119–153.

Yang, F.; Culberson, J.; Holte, R.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstractions. *Artif. Intell.* 32:631–662.

Yoon, S.; Fern, A.; and Givan, R. 2008. Learning control knowledge for forward search planning. *J. Mach. Learn. Res.* 9:683–718.

Zahavi, U.; Felner, A.; Holte, R.; and Schaeffer, J. 2006. Dual search in permutation state spaces. *AAAI* 1076–1081.

Zhou, R., and Hansen, E. 2005. External-memory pattern databases using structured duplicate detection. In *AAAI*, 1398–1405.