Artificial Intelligence 175 (2011) 2075–2098

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Learning heuristic functions for large state spaces

Shahab Jabbari Arfaee[a], Sandra Zilles[b,*], Robert C. Holte[a]

[a] University of Alberta, Department of Computing Science, Edmonton, Alberta, Canada T6G 2H8
[b] University of Regina, Department of Computer Science, Regina, Saskatchewan, Canada S4S 0A2

ARTICLE INFO

Article history:
Received 19 September 2010
Received in revised form 27 July 2011
Accepted 1 August 2011
Available online 5 August 2011

Keywords:
Heuristic search
Planning
Learning heuristics

ABSTRACT

We investigate the use of machine learning to create effective heuristics for search algorithms such as IDA* or heuristic-search planners such as FF. Our method aims to generate a sequence of heuristics from a given weak heuristic h_0 and a set of unsolved training instances using a bootstrapping procedure. The training 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 so weak that it cannot solve any of the given instances we use random walks backward from the goal state to create a sequence of successively more difficult training instances starting with ones that are guaranteed to be 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 this method on the 24-sliding-tile puzzle, the 35-pancake puzzle, Rubik's Cube, and the 20-blocks world. In every case our method produces a heuristic that allows IDA* to solve randomly generated problem instances quickly with solutions close to optimal.

The total time for the bootstrap process to create strong heuristics for these large state spaces is on the order of days. To make the process effective when only a single problem instance needs to be solved, we present a variation in which the bootstrap learning of new heuristics is interleaved with problem-solving using the initial heuristic and whatever heuristics have been learned so far. This substantially reduces the total time needed to solve a single instance, while the solutions obtained are still close to optimal.

1. Introduction

Modern heuristic search and planning systems require good heuristics. A popular 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 admissible estimates of distances in the original space [4,5,16,24,34,36]. One limitation of this approach is that it is often memory-intensive. This has led to the study of compression schemes [3,7,42], disk-based methods [52], and distributed methods [8]. 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. There are situations in which 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

[*] Corresponding author.
E-mail addresses: jabbaria@cs.ualberta.ca (S. Jabbari Arfaee), zilles@cs.uregina.ca (S. Zilles), rholte@ualberta.ca (R.C. Holte).

0004-3702/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2011.08.001

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.

An approach to the automatic creation of heuristics that sidesteps both of these limitations is to apply machine learning to a set of states whose distance-to-goal is known (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 (see Ernandes and Gori [9] and Samadi, Felner, and Schaeffer [41]), but could not be applied to larger spaces, *e.g.*, the 24-puzzle, because of the excessive time it would take to create a sufficiently large training set containing a sufficiently broad range of possible distances to goal. To overcome this obstacle, Samadi et al. [41] 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 combined them to get a heuristic for the 24-puzzle. This approach inherits the limitations of abstraction mentioned above and, in addition, the crucial choices of which abstractions to use and how to combine them are made manually.

Ernandes and Gori [9] 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 automatic bootstrapping practical and showing experimentally that it succeeds on state spaces that are at or beyond the limit of today's abstraction methods.

Bootstrapping is an iterative procedure that uses learning to create a series of heuristic functions. Initially, this procedure requires a heuristic function h_0 and a set of states we call the bootstrap instances. Unlike previous machine learning approaches to creating heuristics, 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 standard heuristic search algorithm (*e.g.*, IDA* [29]) 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 on previous iterations).

If the initial heuristic h_0 is too weak to solve a sufficient number of the given bootstrap instances within the given time limit we use a random walk method to automatically generate 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 by Ernandes and Gori [9] and Samadi et al. [41], 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 four problem domains that are at, or beyond, the limit of what current abstraction methods can solve optimally—the 24-sliding-tile puzzle, the 35-pancake puzzle, Rubik's Cube, and the 20-blocks world—in each case 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 quickly with solutions that are very close to optimal. On these domains our method systematically outperforms Weighted IDA* [30] and BULB [15].

The time it takes for our bootstrap method to complete its learning on these large state spaces is on the order of days. This is acceptable when the learned heuristic will be used to solve many instances, but a different approach is needed in order to solve a single instance quickly. For this we introduce a method that interleaves the bootstrapping process for creating a succession of ever stronger heuristics with a process that uses the set of heuristics that are currently available (initially just h_0) to try to solve the given instance. The total time required to solve a single instance using this method is substantially less than the learning time for the bootstrap method, and the solutions it produces are of comparable suboptimality. For example, with this method the total time to solve an instance of the 24-puzzle is just 14 minutes, on average, and the solution found is only 6.5% longer than optimal. When applied to the blocksworld instances used in the IPC2 planning competition, our interleaving method solves all the instances within the 30-minute time limit, and almost all are solved optimally.

The remainder of the paper is organized as follows. Section 2 provides a full description of the Bootstrap and RandomWalk methods, which are experimentally evaluated in Section 3. The interleaving method for quickly solving single instances is described and evaluated in Section 4. Section 5 surveys previous work related to bootstrapping and Section 6 closes the paper with a summary and conclusions.

2. The Bootstrap and RandomWalk algorithms

This section describes the algorithmic approach and implementation of our method for learning heuristics. The input to our system consists of a state space, a fixed goal state g, a heuristic function h_0, a set Ins of states to be used as bootstrap instances, and a set of state features to be used for learning. We do not assume that h_0 is sufficiently strong that any of the given bootstrap instances can be solved using it. In principle, h_0 could be completely trivial (returning 0 for all states) but in practice it is useful to include weak but non-trivial heuristics among the features used for learning. If that is done it makes sense to use their maximum as h_0 in the absence of any stronger heuristic.

In the first subsection, we focus on the bootstrap procedure, which incrementally updates the initial heuristic with the help of a set of bootstrap instances. This procedure requires h_0 to be strong enough to solve several of the given instances at least suboptimally in the given time limit. If it is not, a set of easier instances is needed to improve the initial heuristic to the point where the easiest bootstrap instances can be solved. This set of easier instances is generated by the random walk method described in the second subsection.

2.1. The Bootstrap algorithm

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 the current heuristic h_{in} (line 7). Every search is cut off after a limited period of time ($t_{\max}$). If i is solved within that time then the 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 8 and 9). 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($h_0$, $h_{in}$, Ins): $\mathrm{h}_{out}$
 2: uses global variables $t_{\max}$, $t_\infty$, $\mathrm{ins}_{\min}$, g
 3: create an empty training set $TS$
 4: NumSolved := 0
 5: while (NumSolved + size(Ins) $\geqslant \mathrm{ins}_{\min}$) && ($t_{\max} \leqslant t_\infty$) do
 6:   for each instance $i \in$ Ins do
 7:     if Heuristic Search($i, g, h_{in}, t_{\max}$) succeeds then
 8:       for each state $s$ on $i$'s solution path $P_i$ do
 9:         Add (feature vector($s$), distance($s, g, P_i$)) to $TS$
10:       end for
11:       remove $i$ from Ins
12:       NumSolved := NumSolved + 1
13:     end if
14:   end for
15:   if (NumSolved $\geqslant \mathrm{ins}_{\min}$) then
16:     $h_{learn}$ := learn a heuristic from $TS$
17:     Define $h_{in}(x)$, for any state $x$, to be $\max(h_0(x), h_{learn}(x))$
18:     clear $TS$
19:     NumSolved := 0
20:   else
21:     $t_{\max} := 2 \times t_{\max}$
22:   end if
23: end while
24: 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 17) and the training set is reset to be empty. If not "enough" bootstrap instances have been solved, the time limit is increased without changing h_{in} (line 21). Either way, as long as the current time limit ($t_{\max}$) 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 15). Variable NumSolved keeps track of the number of bootstrap instances solved in each iteration. It increases whenever a new instance is solved (line 12) and it will be set to zero for the next iteration (line 19). The procedure terminates if $t_{\max}$ exceeds t_∞ or if the remaining set of bootstrap instances is too small.

Notice (line 17) that each heuristic h_{in} created by bootstrap is the maximum of the heuristic returned by the learning algorithm using the current training set (h_{learn}) and h_0. This is not an essential requirement of the bootstrap process but it is advisable when h_0 is known to be an admissible heuristic since it can only make the heuristic more accurate. In all the experiments reported below, this method was used. It is also possible to take the maximum over all previously learned heuristics as well as h_{learn} and h_0, or to add the previously learned heuristics to the set of features used for learning. We did not do either of these because of the considerable increase in computation time they would have caused.

Table 1 shows each iteration of the bootstrap procedure on the 15-puzzle (defined in Section 3) when Ins contains 5000 randomly generated solvable instances, $\mathrm{ins}_{\min}$ is 75, and $t_{\max}$ is 1 second. The definition of the initial heuristic h_0, the

Table 1
Bootstrap iterations for the 15-puzzle.

Iteration	Number solved	Average optimal cost (solved instances)	Average nodes generated (solved instances)	Average suboptimality (test instances)
0	986	46.11	517,295	1.2%
1	3326	54.37	205,836	2.8%
2	519	58.52	353,997	5.5%
3	156	60.44	276,792	8.3%

learning method, and the features used for learning are the same as those for the 24-puzzle that are given in Section 3 below. The first row shows the result of the initial iteration. All 5000 instances in Ins were attempted but IDA* with h_0 was only able to solve 986 of them in the time limit (column "Number solved"). The average optimal solution length for the solved instances is shown in column "Average optimal cost". The average number of nodes generated in solving these instances is shown in column "Average nodes generated". The states along these 986 solution paths, together with their distances to the goal, form the training set to which a learning algorithm is applied to create a new heuristic, h_1.[1] The suboptimality of the heuristic learned on this iteration (h_1), measured on an independent test set, is shown in column "Average suboptimality".

An attempt is then made using h_1 to solve each of the 4014 instances that were not solved using h_0. The next row (iteration 1) shows that 3326 of these were solved in the time limit. All the states along all these solution paths were used to learn a new heuristic h_2, which was then used in an attempt to solve each of the 688 instances that were not solved on the first two iterations. The next row (iteration 2) shows that 519 of these were solved. The heuristic, h_3, learned from these solution paths solved 156 of the 169 instances not solved to this point, and those solution paths provide the training data to create a new heuristic, h_4. The bootstrap process ends at this point because there are fewer than $\text{ins}_{\min}$ unsolved instances, and h_4 is returned as the final heuristic. In this example, there was no need for the bootstrap process to increase the time limit $t_{\max}$ because each iteration solved $\text{ins}_{\min}$ or more instances with the initial $t_{\max}$ value.

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 simply an intuitive informal requirement for which we have no proof of necessity.

2.2. The RandomWalk algorithm

It can happen that the initial heuristic h_0 is so weak that the heuristic search algorithm is unable to solve enough instances in Ins, using h_0, 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 accomplished using random walks backward from the goal[2] of a suitably chosen length to generate instances. As described in Algorithm 2, we first test whether the initial heuristic is strong enough to solve a sufficient number (at least $\text{ins}_{\min}$ many) of the user-provided bootstrap instances (Ins) in the given time limit $t_{\max}$ (line 5). If so, the bootstrap procedure can be started immediately (line 10). 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 then run on these special instances (line 7) to create a stronger heuristic. This process is repeated with increasingly longer random walks (line 8) until it produces a heuristic that is 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_∞.

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

```
1: procedure RandomWalk ($h_0$, Ins, lengthIncrement): $h_{out}$
2: uses global variables $t_{\max}, t_\infty, \text{ins}_{\min}, g$
3: length := lengthIncrement
4: $h_{in} := h_0$
5: while ($h_{in}$ is too weak to solve $\text{ins}_{\min}$ many instances in Ins within time $t_{\max}$) && ($t_{\max} \leqslant t_\infty$) do
6:     RWIns := 200 instances, each generated by applying "length" many random moves backward from $g$
7:     $h_{in}$ := Bootstrap($h_0, h_{in}$, RWIns)
8:     length := length + lengthIncrement
9: end while
10: return Bootstrap($h_0, h_{in}$, Ins)
```

[1] As explained above, h_1, and all other heuristics created by Bootstrap, are defined, for any state x, as the maximum of $h_0(x)$ and $h_{learn}(x)$, where h_{learn} is the heuristic created by the learning algorithm in the current iteration.

[2] For spaces with uninvertible operators, this requires a predecessor function, not just the successor function provided by an opaque representation. Hence the RandomWalk part of the process will not be applicable to certain opaque domains. Moreover, the RandomWalk procedure works only for single goal states, not for sets of goal states. Neither of these restrictions applies to the Bootstrap procedure itself, since there the search progresses in the forward direction.

Table 2
RandomWalk procedure applied to the 20-blocks world.

Row	RW length	Number solved	Average optimal cost	Time limit
1	20	197	8.97	1
2	40	145	11.76	1
3	60	115	13.96	1
4	60	79	16.05	2
5	80	99	16.08	2
6	80	95	19.37	4
7	100	174	20.41	4
8	120	139	23.50	4

The choice of “lengthIncrement” 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 our system, the lengthIncrement parameter was set automatically 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 backward from the goal (always disallowing the inverse of the previous move) until a state not in S is reached. Set lengthIncrement to be the floor of the average length of these 5000 random walks.

The intuition motivating this definition of lengthIncrement is as follows. Initially, it generates problem instances that, on average, are just a little more difficult than can be solved using breadth-first search with a time limit of $t_{\max}$. These are thus expected to provide training examples that are solvable using h_0 and cause a non-trivial heuristic function to be learned. On subsequent iterations, we imagine that most of the instances created by random walks whose length is the next larger multiple of lengthIncrement will be within a short breadth-first search of the instances that were solved on the previous iteration—in other words, just slightly more difficult, on average, than the previous instances. By being slightly more difficult they are easy enough to be solved using the current heuristic but provide training instances that allow a better heuristic to be learned.

The RandomWalk approach is not guaranteed to succeed. It might fail to generate problems of a suitable level of difficulty (easy enough to be solvable using the current heuristic but hard enough to help produce a better heuristic).

Table 2 illustrates the RandomWalk procedure on the 20-blocks world when Ins contains 5000 randomly generated solvable instances, $\text{ins}_{\min}$ is 75, $t_{\max}$ is 1 second, and 200 random walk instances are generated (RWIns) for each distinct random walk length. The definition of this domain, the initial heuristic h_0, the learning method, and the features used for learning are given in Section 3 below. Random walks are necessary in this domain because h_0 is too weak to solve a sufficient number ($\text{ins}_{\min}$) of the bootstrap instances (Ins). The value of “lengthIncrement” was set automatically by our method at 20.

The first row shows the result of the initial iteration. 200 instances (RWIns) have been generated by random walks of length 20 (column “RW length”) and passed to the bootstrap procedure along with h_0. IDA* using h_0 as the heuristic was able to solve 197 of these instances (column “Number solved”) within the time limit (column “Time limit”, in seconds) so there is just one iteration of the bootstrap process, which returns a new heuristic, h_1. This heuristic is then used to attempt to solve the bootstrap instances in Ins. It is too weak to solve a sufficient number of them in the time limit so another iteration of the RandomWalk process is needed.

The random walk length is increased by 20 (the value of lengthIncrement) and a set (RWIns) of 200 instances is generated by random walks of length 40 and passed to the bootstrap procedure along with h_1. 145 of them are solved in the first bootstrap iteration and the bootstrap procedure returns a new heuristic, h_2, since fewer than $\text{ins}_{\min}$ unsolved RandomWalk instances remain. This heuristic is used to attempt to solve the bootstrap instances (Ins). It is too weak to solve a sufficient number of them in the time limit so another iteration of the RandomWalk process is needed.

The random walk length is increased by 20 and a set of 200 instances (RWIns) are generated by random walks of length 60 and passed to the bootstrap procedure along with h_2. The bootstrap process (row 3) is only able to solve 115 of these instances using h_2 in its first iteration. A new heuristic, h_3, is learned from these but is not passed back to the RandomWalk procedure because there are still more than $\text{ins}_{\min}$ unsolved RandomWalk instances (RWIns). A second iteration of the bootstrap procedure attempts to solve them with its new heuristic, h_3, but fails to solve a sufficient number ($\text{ins}_{\min}$) and therefore doubles the time limit and attempts them again with h_3. Row 4 shows that this iteration of the bootstrap procedure succeeds in solving 79 of them with the new time limit, and from these it learns a new heuristic, h_4. Since there are now fewer than $\text{ins}_{\min}$ unsolved RandomWalk instances, the bootstrap procedure returns h_4 to the RandomWalk process. This heuristic is used in an attempt to solve the bootstrap instances (Ins). It is too weak to solve a sufficient number of them in the time limit so another iteration of the RandomWalk process is needed.

As the table shows, in total 6 iterations of the loop in the RandomWalk process were executed (6 distinct values of the RandomWalk length) and for each of these iterations either one or two bootstrap iterations were required to find a heuristic

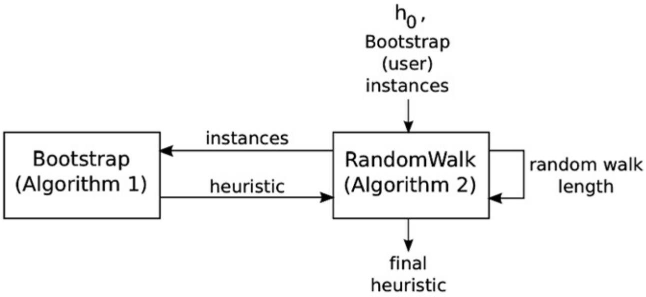

Fig. 1. System overview.

that could solve the random walk instances. The time limit had to be increased twice. The RandomWalk process ended because the heuristic, h_8, created from the RandomWalk instances solved in the final row of the table, was able to solve a sufficient number (ins_{min}) of the bootstrap instances (Ins) that the bootstrap procedure could finally be started on the set of bootstrap instances (Ins) with h_8 as its initial heuristic.

Once the bootstrap process begins to operate on the bootstrap instances, the RandomWalk process will not be invoked again. However, there is a role that it could play. If in some iteration the Bootstrap process fails to solve a sufficient number of instances, instead of doubling its current time limit (line 21 of Algorithm 1) it could instead invoke the RandomWalk process to generate instances of the appropriate level of difficulty. Preliminary experiments with this idea succeeded on artificially contrived sets of bootstrap instances for the 15-puzzle and 17-pancake puzzle, but failed on Rubik's Cube. Because of the latter we abandoned this idea and all the experiments reported in this paper are based on using RandomWalk only as an initial step, if needed, to create a heuristic strong enough to allow the bootstrap process to begin operating on the bootstrap instances.

2.3. Summary: System overview

A summary of the overall system and its operation is depicted in Fig. 1. The key inputs from the user are an initial heuristic h_0 and a set of bootstrap instances. The RandomWalk procedure tests whether h_0 is strong enough to solve a sufficiently large number of the bootstrap instances. If it is not, RandomWalk internally generates its own instances through random walks of a length that it determines automatically. These instances are passed to the Bootstrap procedure, which returns a heuristic. This procedure repeats, with instances created by random walks of increasing lengths, until the current heuristic is strong enough to solve a sufficiently large number of the bootstrap instances. At this point Bootstrap is invoked one last time, with the bootstrap instances. The final heuristic it creates on these instances is the heuristic that is output by the system.

3. Experiments with Bootstrap and RandomWalk

Except where explicitly stated otherwise, IDA* was the search algorithm used.

Domains. Because it is essential in this study to be able to determine the suboptimality of the solutions our method produces, we chose as testbeds domains in which optimal solution lengths can be computed in a reasonable amount of time, either by existing heuristic search methods or by a hand-crafted optimal solver for the domain. The following domains met this criterion.[3]

- **($n^2 - 1$)-Sliding-tile puzzle** [46] – The sliding-tile puzzle consists of $n^2 - 1$ numbered tiles that can be moved in an $n \times n$ grid. A state is a vector of length n^2 in which component k names what is located in the kth puzzle position (either a number $1, \ldots, n^2 - 1$ for a tile or a symbol representing the blank). Every operator swaps the blank with a tile adjacent to it. The left part of Fig. 2 shows the goal state that we used for the 24-puzzle while the right part shows a state created from the goal state by applying two operators, namely swapping the blank with tile 1 and then swapping it with tile 6.
  The number of states reachable from any given state is $(n^2)!/2$, cf. [1]. We report results on the 24-puzzle ($n = 5$), the largest version of the puzzle that has been solved optimally by abstraction-based heuristic search methods [32]. This domain has roughly 10^{25} reachable states.
- ***n*-Pancake puzzle** [6] – In the n-pancake puzzle, a state is a permutation of n numbered tiles and has $n - 1$ successors, with the lth successor formed by reversing the order of the first $l + 1$ positions of the permutation ($1 \leqslant l \leqslant n - 1$).

[3] Experiments on smaller versions of some of these domains (the 15-puzzle, the 17- and 24-pancake puzzles, and the 15-blocks world) can be found in a previous publication [27].

	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

1	6	2	3	4
5		7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Fig. 2. The goal state for the 24-puzzle (left) and a state two moves from the goal (right).

1	2	3	4	5	⋯	34	35

4	3	2	1	5	⋯	34	35

Fig. 3. The goal state for the 35-pancake puzzle (above) and a state one move from the goal (below).

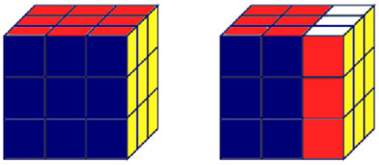

Fig. 4. The goal state for Rubik's Cube (left) and a state one move from the goal (right) (modified from Zahavi et al. [51]).

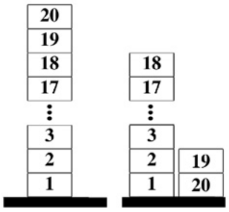

Fig. 5. The goal state for the 20-blocks world (left) and a state two moves from the goal (right).

The upper part of Fig. 3 shows the goal state that we used in our experiments, while the lower part shows the 3rd successor of the goal (the first four positions have been reversed).

All $n!$ permutations are reachable from any given state. We report results for $n = 35$ which contains more than 10^{40} reachable states. The largest version of the puzzle that has been solved optimally by general-purpose abstraction-based methods is $n = 19$ [22].

- **Rubik's Cube** [31] – Rubik's Cube is a $3 \times 3 \times 3$ cube made up of 20 moveable $1 \times 1 \times 1$ "cubies" with coloured stickers on each exposed face. Each face of the cube can be independently rotated 90 degrees clockwise or counterclockwise or 180 degrees. The left part of Fig. 4 shows the goal state for Rubik's Cube while the right part shows the state produced by rotating the right face 90 degrees counterclockwise.

  We used the standard encoding of the puzzle, including the standard operator pruning methods that reduce the branching factor from 18 to approximately 13.34847 [31]. The number of states reachable from any given state is approximately 4.3252×10^{19} [31]. Rubik's Cube is at the limit of today's general-purpose heuristic search methods for finding optimal solutions.
- ***n*-Blocks world** [45] – In the blocks world, each block can have at most one block on top of it and one block below it. A block with no block below it is said to be on the table. A block with no block above it is said to be clear. A move consists in moving a clear block to be on top of some other clear block or onto the table. We used $n = 20$ blocks in our experiments; the number of reachable states is more than 10^{20} [45]. The left side of Fig. 5 shows the goal state that we used; the right side of Fig. 5 shows the state produced from the goal state by moving block 20 to the table and then moving block 19 to the top of block 20.

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 standard backpropagation [40] and mean squared error (*MSE*).[4] Training ended after 500 epochs or when $MSE < 0.005$.

[4] We do not consider the choice of the particular learning algorithm critical; we chose this neural network setting to be the same as previous work on learning heuristics [9,41]. Using only three hidden units made sure that, for every domain we experimented with, the number of inputs was at least as large as the number of hidden units. We experimented with various error measures that penalize overestimation, but found none that yielded substantially

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 et al. [49]. In our experiments we did not carefully engineer the features used or exploit special properties of the domain. Our intention was to show that the bootstrap learning approach is effective, even in the absence of carefully chosen features and human insight into the domain. We did deliberately choose features that could be quickly calculated since their values are all needed every time a heuristic value must be computed for a state.

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. In all the domains other than Rubik's Cube, h_0 was too weak for us to evaluate it on the test instances in a reasonable amount of time. After each iteration of our method, the new heuristic was defined as the maximum of the output of the NN and the initial heuristic. No domain-specific knowledge, such as geometric symmetry or duality [51] was used to augment the heuristics in any of the experiments reported.

One advantage that h_0 has compared to the heuristics generated by bootstrapping is that it can be computed more quickly, since the maximum of a set of feature values can be evaluated faster than a neural network output with the same features as input. For example, the computation of each neural network heuristic in our experiments was between 1.25 (Rubik's Cube) and 2.0 (24-puzzle) times slower than the computation of the corresponding h_0 heuristics.

Bootstrap instances. Ins consisted of either 500 or 5000 solvable instances generated uniformly at random except for Rubik's Cube where they were generated by random walks of various lengths between 1 and 25.

Numeric parameters. In all experiments, $\text{ins}_{\text{min}} = 75$, $t_{\text{max}} = 1$ second, $t_\infty = 512$ seconds, and the size of the set RWIns was 200.

The tables below summarize the results on our test domains. All these results are based on a set of test instances generated independently of the bootstrap instances, in contrast to Table 1, where some measurements were based on the bootstrap instances solved in the respective Bootstrap iteration. In the tables with Bootstrap results, the "Iteration" column indicates which Bootstrap iteration is being described in each row. The "No. solved" and "Total unsolved" columns show, respectively, the number of bootstrap instances solved in a particular iteration and the total number of bootstrap instances that are not yet solved at the end of that iteration.

The "Avg. subopt." column gives the average suboptimality of the solutions found for the test instances by the heuristic produced at the end of an iteration, calculated as follows. We define the suboptimality for an instance as the cost of the solution found for that instance divided by its optimal solution cost. We then compute the average over all the instances of the individual suboptimalities and subtract one. For example, Avg. subopt. = 7% means that, on average, the solution found for a test instance was 7% longer than its optimal solution.

"Avg. nodes gen." is the average number of nodes generated to solve the test instances using the heuristic produced at the end of an iteration. "Avg. solving 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. "Learning time" in the row for iteration i is the time used by our method to complete all iterations up to and including i, including all the RandomWalk processing required before iteration 0 could begin. The letters "s", "m", "h", and "d" represent units of time—seconds, minutes, hours, and days, respectively.

Each row in the "Other methods" tables gives the data for a non-bootstrapping system that we tried or found in the literature. The "h (Algorithm)" column indicates the heuristic used, with the search algorithm, if different from IDA*, given in parentheses. The symbol #k indicates that the same heuristic is used in this row as in row k. The run-times taken from the literature are marked with an asterisk to indicate they may not be strictly comparable to ours. Some suboptimalities from the literature are computed differently than ours; these too are marked with an asterisk. All weighted IDA* (W-IDA*) and BULB results are for our own implementations of these algorithms, except for the BULB results on Rubik's Cube, which are from Furcy and König [15].

The last Bootstrap iteration shown in the tables represents the last successful iteration of the Bootstrap process. If there were fewer than ins_{min} unsolved bootstrap instances remaining after that iteration (24-puzzle, 35-pancake puzzle, and Rubik's Cube and the 20-blocks world using 5000 bootstrap instances), the Bootstrap process terminated as soon as that iteration was done and the "Bootstrap completion time" shown in each table, which measures the entire time required by the Bootstrap process, is equal to the "Learning time" reported for the final iteration. However, if there were ins_{min} or more unsolved bootstrap instances remaining after the last iteration shown in a table (Rubik's Cube and 20-blocks world, each with 500 bootstrap instances), another bootstrap iteration would have been attempted on those instances but Bootstrap terminated it without creating a new heuristic because t_{max} exceeded t_∞. In such a case the "Bootstrap completion time" includes the time taken by the final, unsuccessful iteration. For example, the "Learning time" for iteration 1 in Table 9 shows

better results than *MSE*. We also briefly experimented with linear regression instead of a neural network; the preliminary results were on a par with those of the neural net.

Table 3
24-Puzzle, Bootstrap (500 bootstrap instances).

Iteration	No. solved	Total unsolved	Avg. subopt.	Avg. nodes gen.	Avg. solving time	Learning time
0 (first)	141	359	5.1%	121,691,641	153.34 s	1 h 38 m
1	98	261	5.9%	156,632,352	205.68 s	2 h 16 m
2	112	149	5.6%	70,062,610	89.03 s	3 h 57 m
3 (final)	128	21	5.7%	62,559,170	81.52 s	11 h 43 m
Bootstrap completion time = 11 hours and 43 minutes						

Table 4
24-Puzzle, Bootstrap (5000 bootstrap instances).

Iteration	No. solved	Total unsolved	Avg. subopt.	Avg. nodes gen.	Avg. solving time	Learning time
0 (first)	413	4587	4.7%	1,798,903,624	2364.69 s	19 h
2	116	4060	4.9%	1,386,730,491	1776.41 s	23 h
4	84	3807	4.8%	1,051,748,928	1366.37 s	1 d 09 h
6	116	3521	5.0%	307,700,388	403.09 s	1 d 17 h
8	263	3029	5.1%	555,085,735	719.79 s	1 d 23 h
10	212	2534	5.7%	140,197,951	182.82 s	2 d 04 h
12	112	2188	6.2%	164,616,540	223.68 s	2 d 08 h
14	116	1850	6.4%	135,943,434	175.11 s	2 d 12 h
16	141	1573	6.4%	35,101,918	45.83 s	2 d 20 h
18	270	1057	7.2%	24,416,967	31.47 s	3 d 03 h
20	156	652	7.7%	18,566,788	24.60 s	3 d 09 h
22	147	393	7.3%	12,172,889	16.13 s	3 d 12 h
24	79	224	7.9%	10,493,649	13.65 s	3 d 22 h
26 (final)	137	12	**8.1**%	**7,445,335**	9.65 s	4 d 21 h
Bootstrap completion time = 4 days and 21 hours						

that it takes 2 days to learn the final heuristic for Rubik's Cube using 500 bootstrap instances, but the "Bootstrap completion time" is reported as 2 days and 19 hours. The difference (19 hours) is the time required by an iteration after iteration 1, which failed to solve ins_{min} new instances within the time limit of t_∞.

3.1. 24-Puzzle

Tables 3 and 4 show our results on the 50 standard 24-puzzle test instances first solved by Korf and Felner [32], which have an average optimal cost of 100.78. The input features for the NN were: Manhattan distance (MD), number of out-of-place tiles, position of the blank, and five heuristics, each of which is a 4-tile pattern database (PDB [5]). The total memory used to hold the PDBs was about 50 megabytes. The time to build the pattern databases and generate bootstrap instances, which we call the pre-processing time, was about 2 minutes.

The initial heuristic is sufficiently weak that nine RandomWalk iterations were necessary before bootstrapping itself could begin (ten iterations were required when there were only 500 bootstrap instances). Table 3 shows the results for all bootstrap iterations when it is given 500 bootstrap instances. Table 4 is 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 "Avg. nodes gen." and "Avg. solving time" columns) but suboptimality becomes worse. The increase in suboptimality is most likely caused by the fact that the solutions for the training instances become increasingly suboptimal on successive iterations. Suboptimal training instances bias the system to learn new heuristics that overestimate to an even greater extent which, in turn, leads to even more suboptimal solutions in subsequent iterations.

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. There is also clearly an interesting relationship between "Learning time" and "Solving time": the heuristics created later in the process solve problems faster on average. In Section 4 we present one approach to exploiting this relationship when there is only one problem instance to solve.

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 combined RandomWalk and 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 seven additional iterations), producing, in the end, faster search but worse suboptimality than when fewer bootstrap instances are used.

Fig. 6 shows the distribution of suboptimality values for iterations 0, 13, 26 of the Bootstrap process with 5000 bootstrap instances. A data point (x, y) on the plot means that for $y\%$ of the test instances the solution was at most $x\%$ suboptimal.

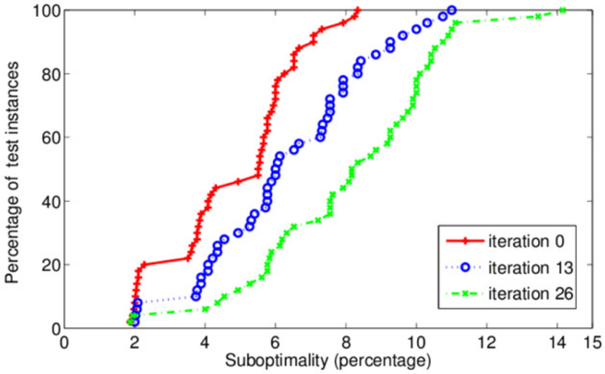

Fig. 6. 24-Puzzle, distribution of suboptimality values.

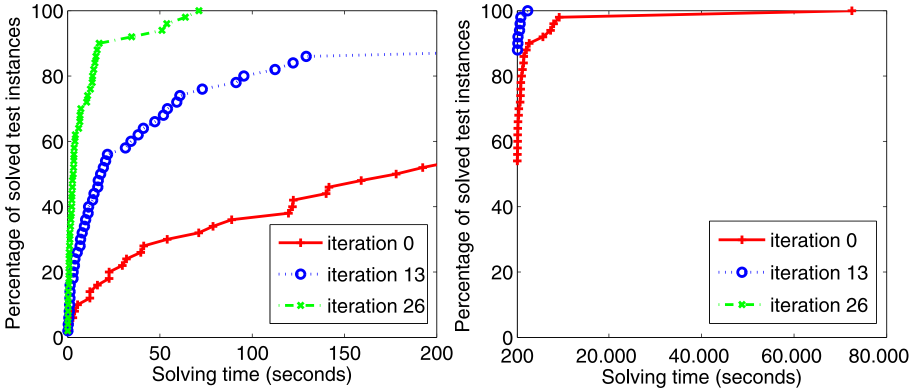

Fig. 7. 24-Puzzle, distribution of solving times.

We see that there is an across-the-board degradation of the suboptimalities from early to later iterations: the curve for iteration 26 is strictly below the curve for iteration 13 which, in turn, is strictly below the curve for iteration 0.

Fig. 7 shows the distribution of solving times in an analogous manner; the x-axis measures solving time in seconds. The left plot is for the instances that are solved in 200 seconds or less and the right plot is for the remaining instances with a different scale on the x-axis. There are a few test instances that take extremely long to solve using the heuristic learned on the first iteration, but by iteration 13 all the instances can be solved in under 2500 seconds. Using the final heuristic, all instances are solved in under 72 seconds. We see that there is an across-the-board improvement in solving times: the plot for iteration 0 is strictly below the plot for iteration 13 which, in turn, is strictly below the plot for iteration 26. The same general trends for suboptimality and solving time were seen in all other test domains unless specifically noted otherwise below.

Table 5 shows the results of other systems on the same test instances. Row 1 reports on W-IDA* using our initial heuristic (h_0) multiplied by a weight (W) chosen so that Subopt is roughly equal to the Subopt value achieved by the final bootstrap heuristic (Table 4, iteration 26). In Row 2, W is chosen so that Nodes gen. is roughly equal to the Nodes gen. value achieved by the final bootstrap heuristic (Table 4, iteration 26). The results of analogous settings for BULB's beam width (B) when h_0 is used are shown in Rows 3 and 4. Bootstrap (Table 4, iteration 26) dominates in all cases, in the sense that if W and B are set so that W-IDA* and BULB compare to Bootstrap in either one of the values (Subopt or Nodes gen.), then the heuristic obtained in the final Bootstrap iteration (Table 4, iteration 26) is superior in the other value. Note however, that W-IDA* guarantees that the solution cost achieved is always within a factor of W of the optimal one—a guarantee that our learned heuristics cannot provide. This has to be kept in mind for all subsequent comparisons of Bootstrap to W-IDA*.

Row 5 shows the results with the heuristic h_{sum}, which is defined as the sum of the heuristic values among the NN's input features (h_0 is the maximum of these values). Although h_{sum} can, in general, be much greater than the actual distance to goal, h_{sum} might be quite an accurate heuristic when a moderate number of weak heuristics are used for NN features, as in our experiments. By comparing its performance with Bootstrap's we can see the return on investment for learning how to combine the different heuristics as opposed to just giving them all equal weight as h_{sum} does. As the results show, h_{sum}, with our NN features for the 24-puzzle, performs very poorly in terms of suboptimality. It is superior to Bootstrap with 500 instances (Table 3, iteration 3) and Bootstrap with 5000 instances (Table 4, iteration 26), in terms of both nodes generated and solving time.

Table 5
24-Puzzle, other methods.

Row	h (Algorithm)	Avg. subopt.	Avg. nodes gen.	Avg. solving time
1	h_0 (W-IDA*, $W = 1.5$)	**9.0%**	39,356,250,896	28,770.8 s
2	h_0 (W-IDA*, $W = 2.6$)	80.5%	**7,579,345**	5.5 s
3	h_0 (BULB, $B = 20{,}000$)	**13.8%**	85,136,475	185.4 s
4	h_0 (BULB, $B = 10{,}000$)	122.9%	**7,624,139**	13.4 s
5	h_{sum}	181.4%	151,892	0.1 s
Results from previous papers				
6	Add 6-6-6-6	0%	360,892,479,670	47 h*
7	#6 (DIDA*)	0%	75,201,250,618	10 h*
8	#6, Add 8-8-8	0%	65,135,068,005	?
9	#6, $W = 1.4$ (RBFS)	9.4%*	1,400,431	1.0* s
10	PE-ANN, Add 11-11-2 (RBFS)	0.7%*	118,465,980	111.0* s
11	#10, $W = 1.2$ (RBFS)	3.7%*	582,466	0.7* s

Rows 6–8 show the results of state-of-the-art heuristic search methods for finding optimal solutions. Row 6 shows the results using the maximum of disjoint 6-tile PDBs and their reflections across the main diagonal as a heuristic, due to Korf and Felner [32]. Row 7 shows the results for DIDA*, obtained by Zahavi, Felner, Holte, and Schaeffer [50] using the same heuristic. In Row 8 the heuristic used is the maximum of the heuristic from Row 6 and a partially created disjoint 8-tile PDB, see Felner and Adler [10] (solving time was not reported). The very large solving times 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 9, due to Samadi et al. [41], illustrates the benefits of allowing some amount of suboptimality. Here, RBFS [30] is used with the heuristic from Row 6 multiplied by 1.4. The number of nodes generated has plummeted. Although this result is better, in terms of nodes generated and solving time, than Bootstrap (Table 4, iteration 26), it hinges upon having a very strong heuristic since we have just noted that W-IDA* with our initial heuristic is badly outperformed by Bootstrap.

Rows 10 and 11 in Table 5 show the PE-ANN results by Samadi et al. [41]. As discussed in the introduction, this is not a direct application of heuristic learning to the 24-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 10 shows that our automatic method is superior to PE-ANN used in this way by a factor of more than 20 in terms of nodes generated. The suboptimality values shown in Rows 10 and 11 are not directly comparable to those in Tables 3 and 4 because Samadi et al. defined average suboptimality differently, as the total length of the solutions found divided by the total length of the optimal solutions. The suboptimality of Bootstrap with 5000 instances, calculated in this way, happens to be the same (to one decimal place) as in Table 4 (8.1%) and is inferior to PE-ANN's. Row 11 shows that if PE-ANN's learned heuristic is suitably weighted it can outperform Bootstrap in both nodes generated and suboptimality.

To see how Bootstrap's results would change if it were given a stronger initial heuristic, we reran the experiment with h_0 being the state-of-the-art admissible heuristic, namely Korf and Felner's maximum of disjoint 6-tile PDBs and their reflections across the main diagonal [32]. We adjusted the features used by the neural network accordingly: instead of 8 features, we now used 13, namely one for each of the four disjoint PDBs, one for each of the four reflected PDBs, one for the sum of the first four PDB features, one for the sum of the four reflected PDB features, plus one each for Manhattan Distance, position of the blank, and number of tiles out of place. Note that increasing the number of features might increase Bootstrap's completion time and its solving time.

The use of a stronger h_0 decreased Bootstrap's completion time by more than 50% when 500 bootstrap instances were used but increased it by about 25% when 5000 bootstrap instances were used. The suboptimality of the solutions found using the final Bootstrap heuristic were unaffected by the use of the stronger heuristic when 500 bootstrap instances were used but increased from 8.1% to 11.2% when 5000 bootstrap instances were used. The most important consequence of using a stronger h_0 is a dramatic reduction of the number of nodes generated by the final heuristic Bootstrap produced. With 500 bootstrap instances only 5,087,295 nodes are generated on average, a 12-fold reduction compared to Table 3, and with 5000 bootstrap instances use of the stronger h_0 produces more than a 6-fold reduction in nodes generated.

We also reran W-IDA* and BULB with this strong h_0. W-IDA* is still outperformed by Bootstrap, but not as badly. $W = 1.45$ yields an average suboptimality similar to Bootstrap's with 5000 instances and the strong h_0, but generates roughly 3 times as many nodes. $W = 1.5$ generates a similar number of nodes but has a higher suboptimality (16% on average, compared to 11.2%). BULB, using the strong heuristic, is more clearly outperformed by Bootstrap. When generating a comparable number of nodes to Bootstrap with 5000 instances and the strong h_0, BULB's suboptimality is much higher than Bootstrap's (418.3% compared to 11.2%). $B = 20{,}000$ resulted in a suboptimality (13.3%) approaching Bootstrap's, but at the cost of generating 36 times more nodes on average.

3.2. 35-Pancake puzzle

For the 35-pancake puzzle the input features for the NN were seven 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. Optimal solution lengths were computed us-

Table 6
35-Pancake puzzle, Bootstrap (500 bootstrap instances).

Iteration	No. solved	Total unsolved	Avg. subopt.	Avg. nodes gen.	Avg. solving time	Learning time
0 (first)	134	366	10.4%	178,891,711	217 s	7 h
1	77	289	10.2%	181,324,430	219 s	9 h
2	81	208	11.4%	169,194,509	202 s	11 h
3	100	108	11.3%	191,333,354	228 s	16 h
4 (final)	77	31	12.3%	131,571,637	158 s	1 d 02 h

Bootstrap completion time = 1 day and 2 hours

Table 7
35-Pancake puzzle, Bootstrap (5000 bootstrap instances).

Iteration	No. solved	Total unsolved	Avg. subopt.	Avg. nodes gen.	Avg. solving time	Learning time
0 (first)	102	4898	9.2%	2,766,675,135	4168 s	1 d 17 h
2	258	4394	9.7%	1,591,749,582	1923 s	2 d 07 h
4	128	4110	10.2%	586,345,353	687 s	2 d 20 h
6	265	3630	10.8%	295,187,243	345 s	3 d 08 h
8	216	3198	11.6%	134,075,802	157 s	3 d 18 h
10	95	2999	12.2%	65,290,479	102 s	4 d 04 h
12	150	2732	12.3%	47,998,040	76 s	4 d 20 h
14	128	2456	12.3%	45,571,411	71 s	5 d 15 h
16	250	2008	12.4%	39,128,839	45 s	5 d 23 h
18	118	1766	13.2%	38,126,208	43 s	6 d 05 h
20	102	1575	13.0%	39,440,284	44 s	6 d 16 h
22	210	1177	13.5%	36,423,262	52 s	7 d 00 h
24	170	814	14.2%	25,034,580	42 s	7 d 10 h
26	105	600	14.9%	26,089,593	43 s	7 d 23 h
28	170	279	15.1%	13,156,609	21 s	8 d 07 h
30 (final)	125	36	**15.4%**	**14,506,413**	21 s	8 d 11 h

Bootstrap completion time = 8 days and 11 hours

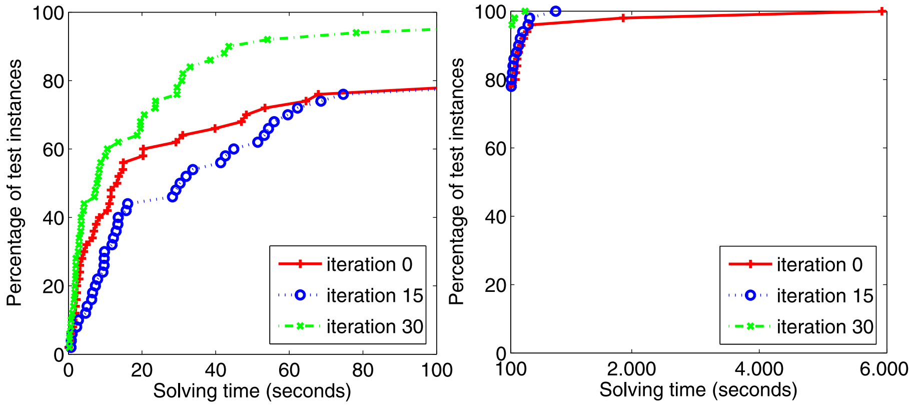

Fig. 8. 35-Pancake puzzle, distribution of solving times.

ing the highly accurate, hand-crafted “break” heuristic.[5] 50 randomly generated instances, with an average optimal solution cost of 33.6, were used for testing. The pre-processing time to build pattern databases and bootstrap instances was about 18 minutes while the memory used to hold the pattern databases was about 272 megabytes.

The initial heuristic is so weak that seven RandomWalk iterations were necessary before bootstrapping itself could begin (9 iterations were required when there were only 500 bootstrap instances). Table 6 has rows for all bootstrap iterations with 500 bootstrap instances and Table 7 has rows for selected iterations with 5000 bootstrap instances. In both cases, we see the same trends as in the 24-puzzle concerning suboptimality, solving time, and the influence of the number of bootstrap instances.

Fig. 8 shows the distribution of solving times for iterations 0, 15, and 30 of the Bootstrap process with 5000 bootstrap instances. Like the corresponding figure for the 24-puzzle (Fig. 7) we see that there are some instances that take a very long time to solve using the heuristic learned in the first iteration and that by the middle iteration there are no such problematic

[5] For details on “break”, see http://tomas.rokicki.com/pancake/ or Helmert [20].

Table 8
35-Pancake puzzle, other methods.

Row	h (Algorithm)	Avg. subopt.	Avg. nodes gen.	Avg. solving time
1	h_0 (W-IDA*, $W = 9$)	108.9%	1,092,647,373	857 s
2	h_0 (BULB, $B = 20{,}000$)	405.4%	426,578,146	1360 s
3	h_0 (BULB, $B = 500$)	2907.9%	154,193,966	483 s
4	h_{sum}	59.0%	71,081,642	69 s

instances: all instances are solved in 500 seconds or less. But unlike the 24-puzzle, here the speedup in solving the hardest instances is accompanied by a slowdown in solving the easier instances: in the left side of Fig. 8 the plot for iteration 15 is below that for iteration 0. By the final iteration, there is an across-the-board improvement in solving time compared to the two other iterations shown.

Because of its large size, no previous general-purpose search system with automatically created heuristics has been applied to this problem domain, so Table 8 includes results only for W-IDA*, BULB, and h_{sum}. None of these methods was able to achieve a "Nodes gen." value similar to Bootstrap with 5000 instances. For W-IDA* and BULB Rows 1 and 3 show 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. 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 (iteration 30 in Table 7) needed on each test instance, W-IDA* did not complete any instances at all. Row 4 shows that h_{sum}, like W-IDA* and BULB, is inferior to Bootstrap (Table 7, iteration 30) in terms of both nodes generated and suboptimality.

As we did for the 24-puzzle, to see the effect of giving Bootstrap a strong initial heuristic, we reran the experiments with h_0 being the strongest general-purpose type of admissible heuristic that is known for the 35-pancake puzzle, the additive heuristics defined by Yang et al. [48]. The particular h_0 we used was a 5-5-5-5-5-5-5 additive PDB. The features used for learning were the seven 5-pancake PDBs, their sum, and the same two non-PDB features used with the weak h_0.

The use of the stronger h_0 did not affect Completion times for either 500 or 5000 bootstrap instances. For 500 bootstrap instances, use of the stronger h_0 decreased suboptimality (from 12.3% to 5.5%) and reduced the number of nodes generated by almost a factor of 5. For 5000 bootstrap instances, the stronger h_0 decreased suboptimality even more (from 15.4% to 5.9%) but had little effect on the number of nodes generated.

3.3. Rubik's Cube

For Rubik's Cube, the input features for the NN were the three PDBs used by Korf [31], namely, one PDB based on the eight corner cubies and two PDBs each based on six edge-cubies. 333 megabytes of memory is used for the PDBs and the pre-processing took about 16 minutes.

Korf's 10 standard Rubik's Cube instances [31] were used for testing. The average optimal solution cost for these instances is 17.5. The initial heuristic was sufficient to begin the Bootstrap process directly, so no random walk iterations were necessary.

Tables 9 and 10 show the results for each bootstrap iteration when 500 and 5000 bootstrap instances are given. In either case, bootstrapping produces very substantial speedup over search using h_0. For instance, using 500 bootstrap instances produces a heuristic that reduces the number of nodes generated by a factor of 43 compared to h_0 while producing solutions that are only 4% longer than optimal. The trends across bootstrap iterations are the same as those observed in previous experiments.

The results of other systems are shown in Table 11. Rows 1 and 2 are when the initial heuristic (h_0) is used with W-IDA* on the same set of test instances. Row 3 shows the results with h_{sum}. As in the Pancake puzzle, Bootstrap (Table 10, iteration 14) outperforms h_{sum} in both suboptimality and nodes generated.

Rows 4–6 show the results of state-of-the-art heuristic search methods for finding optimal solutions. Row 4 shows the results using the initial heuristic (h_0) [31]. Row 5 shows the results by Zahavi et al. [51] when dual lookups [11] for both 6-edge PDBs were used in conjunction with the heuristic of Row 4. In Row 6 [51], the edge PDBs used in Row 5 are increased from 6-edge to 7-edge and dual lookup is used. Bootstrap outperforms all of these optimal systems in terms of nodes generated and solving time.

For BULB, we compared our results to those of Furcy and König [15], which were obtained using h_0. However, Furcy and König used a different set of test instances: they created 50 solvable instances by doing random walks of length 500 backward from the goal state. This set of instances is currently unavailable, making it impossible to do a precise comparison with our method. With that in mind, an inspection of Furcy and König's results shows that with an appropriate setting of B, BULB's performance in terms of nodes generated is similar to Bootstrap's; the average number of nodes generated on Furcy and König's 50 instances, using $B = 50{,}000$, was 189,876,775, compared to 192,012,863 for Bootstrap on Korf's 10 instances (see iteration 14 in Table 10). Because the optimal solution costs for Furcy and König's instances are not known, a comparison of suboptimalities is not possible.

Table 9
Rubik's Cube, Bootstrap (500 bootstrap instances).

Iteration	No. solved	Total unsolved	Avg. subopt.	Avg. nodes gen.	Avg. solving time	Learning time
0 (first)	256	244	2.8%	67,264,270,264	78,998 s	5 m
1 (final)	76	178	4.0%	8,243,780,391	10,348 s	2 d
Bootstrap completion time = 2 days 19 hours						

Table 10
Rubik's Cube, Bootstrap (5000 bootstrap instances).

Iteration	No. solved	Total unsolved	Avg. subopt.	Avg. nodes gen.	Avg. solving time	Learning time
0 (first)	2564	2436	3.4%	69,527,536,555	86,125 s	43 m
1	355	2081	4.6%	7,452,425,544	10,477 s	10 h
2	126	1955	5.8%	3,314,096,404	3976 s	1 d 06 h
3	82	1873	9.7%	3,722,365,147	4444 s	2 d 16 h
4	166	1707	12.6%	974,287,428	1119 s	5 d 08 h
5	149	1558	16.0%	748,608,645	848 s	7 d 05 h
6	162	1396	21.8%	599,503,676	823 s	9 d 09 h
7	166	1230	20.1%	614,676,983	842 s	11 d 07 h
8	76	1154	21.8%	465,772,443	626 s	13 d 04 h
9	256	898	22.9%	552,259,662	624 s	16 d 14 h
10	85	813	22.9%	518,980,590	577 s	19 d 10 h
11	136	677	25.3%	624,542,989	686 s	23 d 20 h
12	218	459	24.7%	422,066,562	464 s	27 d 06 h
13	206	253	27.5%	251,228,458	280 s	30 d 02 h
14 (final)	192	61	**29.3%**	**192,012,863**	208 s	31 d 15 h
Bootstrap completion time = 31 days and 15 hours						

Table 11
Rubik's Cube, other methods.

Row	h (Algorithm)	Avg. subopt.	Avg. nodes gen.	Avg. solving time
1	h_0 (W-IDA*, $W = 1.9$)	**30.4%**	5,653,954,001	6632 s
2	h_0 (W-IDA*, $W = 3.3$)	76.4%	**217,463,103**	245 s
3	h_{sum}	54.5%	246,235,226	256 s
Results from previous papers				
4	h_0	0%	360,892,479,670	102,362 s*
5	#4 with dual lookup	0%	253,863,153,493	91,295 s*
6	max{8, 7, 7} with dual lookup	0%	54,979,821,557	44,201 s*

3.4. 20-Blocks world

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 [45]. We used 50 random test instances in which the goal state has all the blocks in one stack. The average optimal solution length of the test instances is 30.92. The total amount of memory required for this experiment was less than 3 kilobytes while the pre-processing took a few seconds.

The initial heuristic is so weak that six RandomWalk iterations were necessary before bootstrapping could begin (eight iterations for 500 bootstrap instances). Tables 12 and 13 show the bootstrap iterations for the 20-blocks world. The heuristics used in the feature vector were so weak that solving the test instances using the early heuristics produced by Bootstrap was infeasible; therefore, iteration 0 is not shown in Table 12 and iterations 0 through 2 are not shown in Table 13. The completion time of Bootstrap using 500 bootstrap instances is much longer than the total time to learn the final heuristic (iteration 3 in Table 12) because “enough” instances were not solved in the last iteration of the Bootstrap and the process terminated due to $t_{\max}$ exceeding t_∞. The trends in these results are the same as for the domains discussed previously.

The results of BULB on the same set of test instances are shown in Table 14. For suboptimality, BULB could not compete with Bootstrap; we tried 15 values for B between 2 and 20,000. The best suboptimality achieved by BULB is shown in Row 1. It shows that even with much greater suboptimality, BULB is inferior to Bootstrap in terms of nodes generated and solving time. BULB's results when B is set so that BULB is approximately equal to Bootstrap (Table 13, iteration 13) in terms of nodes generated is shown in Row 2. Again Bootstrap dominates.

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. An attempt to compare our results to h$_{sum}$ failed because the heuristics used in the feature

Table 12
20-Blocks world, Bootstrap (500 bootstrap instances).

Iteration	No. solved	Total unsolved	Avg. subopt.	Avg. nodes gen.	Avg. solving time	Learning time
1	75	338	1.8%	13,456,726,519	55,213 s	11 h
2	95	243	2.4%	8886,906,652	35,692 s	1 d 02 h
3 (final)	90	167	**3.8%**	**615,908,785**	2763 s	1 d 10 h
Bootstrap completion time = 2 days						

Table 13
20-Blocks world, Bootstrap (5000 bootstrap instances).

Iteration	No. solved	Total unsolved	Avg. subopt.	Avg. nodes gen.	Avg. solving time	Learning time
3	275	2781	2.2%	12,771,331,089	52,430 s	3 d 06 h
4	290	2491	3.2%	8,885,364,397	35,636 s	4 d 04 h
5	450	2041	3.5%	941,847,444	3828 s	5 d 21 h
6	556	1485	4.0%	660,532,208	2734 s	7 d 03 h
7	162	1323	4.1%	789,515,580	3240 s	8 d 05 h
8	117	1206	5.4%	191,696,476	791 s	9 d 05 h
9	508	698	6.5%	22,413,312	93 s	9 d 22 h
10	377	321	8.0%	11,347,282	47 s	10 d 18 h
11	98	223	8.7%	17,443,378	72 s	10 d 10 h
12	83	140	8.9%	7,530,329	31 s	10 d 20 h
13 (final)	89	51	**9.6%**	**5,523,983**	23 s	11 d 01 h
Bootstrap completion time =11 days and 1 hour						

Table 14
20-Blocks world, other methods.

Row	h (Algorithm)	Avg. subopt.	Avg. nodes gen.	Avg. solving time
1	h_0 (BULB, $B = 20{,}000$)	28.8%	278,209,980	2482 s
2	h_0 (BULB, $B = 2400$)	58.8%	**5,809,791**	32 s

vector were so weak that even the sum of these values is still a weak heuristic for this domain. h_{sum} failed to solve any instance given a time limit of one day per instance.

4. Solving single instances quickly

The preceding experiments demonstrate that bootstrap learning can help to speed up search dramatically with relatively little degradation in solution quality. An inherent and non-negligible expense is the time invested in learning the heuristic function. The Bootstrap completion times reported are on the order of days. Such a lengthy process would be warranted if the final heuristic was going to be used to solve numerous problem instances that were distributed in the same way as the bootstrap instances, since one would expect most of the new instances would be solved as quickly with the final heuristic as the bootstrap instances were, *i.e.*, within the time limit used in the last iteration of the bootstrap process.

However, many planning problems require just a single instance to be solved—a task for which our bootstrapping approach may seem ill-suited because of the large total time required. In this section we investigate whether a variation of our bootstrapping method can quickly solve a single instance of a given problem domain. Instead of minimizing solving time at the expense of requiring very large learning times, as in the previous sections, we are now looking for a balance in the learning and solving times so that the sum of the learning and solving times is made as small as can be. With this goal in mind, we present a method that involves interleaving the learning and solving processes. The method is fully automatic once the ratio of solving time to learning time is specified. We present experimental results on the 24-puzzle, the 35-pancake puzzle, Rubik's Cube, the 20-blocks world and the IPC2 blocksworld instances. In all domains other than Rubik's Cube, interleaving bootstrap learning and problem-solving proves very effective at solving single instances.

4.1. Method and implementation

An important factor influencing the total time for the bootstrap process in the previous experiments is the number of bootstrap instances. For instance, Tables 3 and 4 show that increasing this number from 500 to 5000 increases the total time from 12 hours to 5 days for the 24-puzzle. In fact, a very large portion of the training time is spent on trying to solve bootstrap instances that are still too difficult for the current heuristic. This suggests that considerable time could be saved if we ran our system without any initial bootstrap instances given at the outset, just using random walks to create training instances at successive levels of difficulty until a heuristic was created with which the one and only target instance ins* could be quickly solved. Following Algorithm 2, this procedure would basically work as follows, starting with h being the initial heuristic.

> If h is too weak to solve ins* within a time limit of $t_{\max}$, generate a set RWIns of instances by random walks backward from the goal. Improve h by applying the bootstrap procedure (Algorithm 1) to (h_0, h, RWIns), where $h_0 = h$. Repeat this process, increasing the length of the random walks in each iteration, until ins* can be solved using the current heuristic h within a time limit of $t_{\max}$.

The total time required by this procedure, including training time and solving time, would be the measure for evaluation.

The obvious problem with this approach is the use of the parameter $t_{\max}$, because the total time will strongly depend on the value of this parameter. If $t_{\max}$ is too low, we might need many iterations. If $t_{\max}$ is too high, we force the solver, when using too weak a heuristic, to spend the full amount of $t_{\max}$ in vain while it would be advantageous to invest more in learning. Automatic adjustment of $t_{\max}$ involves the time-consuming process of attempting to solve a non-negligible number of instances created by RandomWalk, and hence the naive method just described is expected to be infeasible.

Avoiding $t_{\max}$ completely by fixing a training time and then trying to solve ins* with the heuristic learned after the fixed amount of training is not any more promising. Here the training time is the critical parameter that cannot be set without prior knowledge.

Our approach to an automated process that does not hinge critically on such parameters is to interleave learning and solving as follows. We alternate between the execution of two threads, a learning thread and a solving thread. The learning thread runs the RandomWalk process in the manner just described to produce a sequence of stronger and stronger heuristics. The solving thread uses the heuristics generated by the RandomWalk process to solve ins*. Initially this thread uses the initial heuristic. When a new heuristic is produced, the solving thread is updated to take into account the existence of a new, probably stronger, heuristic.

There are many possible ways of updating the solving thread when a new heuristic becomes available; here we examine just three.

1. The simplest approach to updating the solving thread is to abort the current search and start a new search from scratch using the new heuristic. We call this approach “Immediate Restart”.
2. The second approach is to finish the current IDA* iteration but using the new heuristic instead of the previous one. If that iteration ends without solving ins* it will have computed the IDA* bound, DB, to use on the next iteration. The next iteration uses the new heuristic, h, and IDA* bound $\max(DB, h(\text{ins}^*))$. We call this approach “Heuristic Replacement”.
3. The third approach is to subdivide the solving thread into a set of solving sub-threads, one for each heuristic that is known. As soon as a new heuristic is learned in the learning thread, this approach starts an additional solving sub-thread, which uses the new heuristic to try to solve ins*. In this approach no thread is ever stopped completely until ins* is solved in one of the solving sub-threads. We call this approach “Independent Solvers”.[6]

Regardless of the approach, the total time by which we evaluate the interleaved learning and solving process is the sum of the times used by both threads (including all the sub-threads) up to the point when ins* is solved in one sub-thread.

Pseudocode for the interleaved learning and solving processes for the Immediate Restart and Heuristic Replacement approaches is shown in Algorithm 3. We use a fixed ratio, $t_s : t_l$, of the time allocated to solving (t_s) and the time allocated to learning (t_l).[7] Line 3 calls “continue” with the Solver thread and a time limit t_s. This executes the solver until it has solved ins* or until t_s seconds have elapsed. If ins* is not yet solved the loop (lines 4–10) is executed until it is. Line 5 calls “continue” with the RandomWalk procedure described above and a time limit t_l. This resumes execution of the RandomWalk process at the point where it was previously suspended, runs it for time t_l, suspends it, and returns whatever its current heuristic is at the time of suspension. If the heuristic returned is new, the solving thread is updated, as described above, to take into account the new heuristic (line 7). Line 9 resumes the (updated) solving thread. The entire process stops when a solution for ins* is found.

Algorithm 3

```
 1: procedure Interleaving(ins$^*$, $h_{in}$, $t_s$, $t_l$): solution
 2: Create a solving thread, Solver, using $h_{in}$.
 3: solved := continue(Solver, $t_s$)
 4: while (!solved) do
 5:     $h$ := continue(RandomWalk, $t_l$)
 6:     if ($h$ is a new heuristic) then
 7:         UPDATE(Solver, $h$)
 8:     end if
 9:     solved := continue(Solver, $t_s$)
10: end while
11: return the solution from the solving thread
```

[6] As opposed to the other two approaches, Independent Solvers has the advantage that it can easily be parallelized.

[7] Technically, not only is the ratio of solving time to learning time given, but also the actual time units. Using a “ratio” of 100:200 will in practice yield different results than a “ratio” of 1:2. For simplicity, and since we always set $t_s = 1$ second in our experiments, we still use the term “ratio” to refer to the setting of t_s and t_l.

Determining the best ratio $t_s : t_l$ for each domain is beyond the scope of this paper; in the current system the ratio has to be set manually. Generally, the weaker the initial heuristic the more the ratio should allocate time to the learning thread. We ran experiments with ratios of 1:1 to 1:10 and, if necessary, for larger values of t_l; see Section 4.2 for details. Since the initial heuristics in our experiments are always rather weak, we did not use ratios favouring the solving thread.

Pseudocode for the interleaved learning and solving processes for the Independent Solvers approach is shown in Algorithm 4. It follows exactly the same general pattern as Algorithm 3, but there is a growing list of Solvers instead of just one Solver. When a new heuristic is learned a new solving sub-thread using this heuristic is added at the beginning of the list. The procedure IndependentSolvers divides the available time for solving, t_s, among the set of available solving threads—exactly how this is done is described in the next paragraph. No thread is terminated until ins* is solved in one of the solving sub-threads.

Algorithm 4

```
procedure Interleaving(ins*, $h_{in}$, $t_s$, $t_l$): solution
Create a list, Solvers, containing just one solving sub-thread using $h_{in}$.
solved := IndependentSolvers(Solvers, $t_s$)
while (!solved) do
    $h$ := continue(RandomWalk, $t_l$)
    if ($h$ is a new heuristic) then
        add a solving sub-thread using $h$ to the beginning of Solvers
    end if
    solved := IndependentSolvers(Solvers, $t_s$)
end while
return the solution from the independent solving sub-threads
```

For the allocation of solving time among the various solving sub-threads, many strategies are possible. The one we report here we call "Exponential". When a new heuristic is learned, this strategy halves the time allocated to the solving sub-threads using previous heuristics and allocates $\frac{t_s}{2}$ seconds to the sub-thread using the new heuristic. Thus the solving sub-thread for the new heuristic gets half the total time available for solving on each round until another heuristic is created. The motivation for this strategy is that heuristics created later in the learning process are expected to be stronger than those created at early stages, so the more recently created heuristics may be more likely to quickly solve the target instance. It therefore seems reasonable to invest more time in solvers using the heuristics learned in later iterations. The reason not to suspend solving sub-threads with weak heuristics completely is that there is still a chance that they are closer to finding a solution than the solving sub-thread using the most recently created heuristic. This may be (i) because more time has already been invested in the sub-threads using weaker heuristics or (ii) because a weaker heuristic may occasionally still behave better on one particular target instance than an overall stronger heuristic.

Other strategies, such as Röger and Helmert's alternation technique [39], are certainly possible.[8]

Algorithm 5

```
procedure IndependentSolvers (exponential) (Solvers, time): status
$t$ := time
for $i = 1$ to |Solvers| do
    $S$ := $i$th sub-thread in Solvers
    if $i \neq$ |Solvers| then
        $t := t/2$
    end if
    if continue($S$, $t$) succeeds then
        return true
    end if
end for
return false
```

Pseudocode for the Exponential time allocation strategy is shown in Algorithm 5. The time invested in the solving sub-thread using the best available heuristic (the first in the Solvers list) is twice as large as that invested in the sub-thread using the second best heuristic, which again is a factor of two larger than the time for the next "weaker" sub-thread, and so on. The weakest two sub-threads will always be allocated the same amount of time, so that the total time spent on the sub-threads sums up to the time allocated to the solving thread overall.

This strategy for allocating the total solving time into time budgets for the currently available heuristic solvers borrows from the hyperbolic dove-tailing approach to interleaved search introduced by Kirkpatrick [28]. Kirkpatrick proved his approach to be average-case optimal and worst-case optimal for a certain variation of the so-called cow path problem, which was first studied by Baeza-Yates, Culberson, and Rawlins [2]. However, this variation of the cow path problem does not

[8] Not reported here are the results of using a uniform strategy, which allocates the same amount of time to all solving sub-threads. We found its performance inferior to that of the "Exponential" strategy. See Jabbari Arfaee's thesis [26] for details.

Table 15
Solving a single instance of the 24-puzzle.

Row	Ratio (t_s : t_l)	min	max	mean	med	std	Subopt.
Immediate restart							
1	1:1	5 m 30 s	62 m 05 s	18 m 49 s	17 m 25 s	8 m 45 s	6.3%
2	1:2	4 m 24 s	47 m 04 s	15 m 54 s	14 m 18 s	6 m 54 s	6.4%
3	1:5 (best)	4 m 04 s	51 m 00 s	15 m 24 s	15 m 18 s	7 m 58 s	6.5%
Heuristic replacement							
4	1:1	5 m 30 s	61 m 54 s	18 m 19 s	17 m 03 s	8 m 29 s	6.2%
5	1:2	4 m 16 s	36 m 30 s	15 m 06 s	13 m 52 s	5 m 48 s	6.3%
6	1:5	4 m 04 s	37 m 32 s	14 m 28 s	14 m 08 s	5 m 57 s	6.3%
7	1:6 (best)	3 m 58 s	36 m 34 s	14 m 05 s	13 m 56 s	5 m 48 s	6.5%
Independent solvers (exponential)							
8	1:1	20 m 48 s	44 m 54 s	23 m 36 s	21 m 54 s	4 m 07 s	6.4%
9	1:2	15 m 36 s	43 m 36 s	18 m 03 s	16 m 30 s	4 m 15 s	6.7%
10	1:5	12 m 30 s	42 m 42 s	15 m 50 s	14 m 30 s	5 m 53 s	6.9%
11	1:10 (best)	11 m 31 s	53 m 46 s	15 m 48 s	14 m 14 s	6 m 55 s	7.0%

exactly model the search problem we are facing. Hence we do not have any formal guarantees on the efficiency of our method.

4.2. Experiments

We ran experiments comparing the three versions of our interleaving approach, with different $t_s : t_l$ ratios, on the same domains used in Section 3. The experimental settings for each domain, *i.e.*, the features, and the neural network settings were the same as those described in Section 3, and we used the same computer. The test instances used for each domain in Section 3 are here used as individual target instances for testing. We also report results on the IPC2 blocksworld instances, along with comparisons to state-of-the-art planners on those instances. In all our experiments, the parameter t_s used in Algorithm 4 was set to 1 second, while t_l was varied from 1 to 10 seconds in steps of 1. Whenever the ratio 1:10 resulted in a lower mean total time than the ratios 1:1 to 1:9, we also tested the ratios 1:11, 1:12, *etc.* until the mean total time started increasing again. The ratio resulting in the lowest total time is marked in the tables below as the “best” ratio. The tables only show the results for ratios 1:1, 1:2, 1:5, and the best ratio.

4.2.1. 24-Puzzle

Table 15 shows the results for our three interleaving strategies for the solvers. The “min”, “max”, “mean”, “med” and “std” columns, respectively, show the minimum, maximum, mean, median, and standard deviation, of the total times on the 50 instances of the 24-puzzle that were used for this experiment. The “subopt.” column shows the average suboptimality of the solutions found, calculated in the same manner as in Section 3. The trends apparent in these results are:

- The average suboptimality increases as the $t_s : t_l$ ratio increases in favour of the learning thread. This can be explained by the trends observed in Section 3. There we have seen that more bootstrap iterations result in larger suboptimality. Since more bootstrap iterations also result in stronger heuristics, the target instance is more likely to be solved first by one of the strongest heuristics created in the interleaving process. A solver using this stronger heuristic, though solving the target instance faster, provides a solution that has a higher cost than the solutions that solvers using the weaker heuristics would have eventually provided.
- The mean and median values initially decrease with growing t_l, *i.e.*, when the $t_s : t_l$ ratio favours the learning thread. It turns out that, on average, the heuristic that solves the target instance requires only a few seconds of solving time. Therefore, most of the solving time is spent on unsuccessful trials using other heuristics. Increasing the learning time makes the system produce stronger heuristics faster. This in turn decreases the total solving time for most instances (mean and median decreases). However, the mean and median values eventually start to increase at some point. This happens for the following reason. As just noted, the heuristic that solves the target instance requires a few seconds of solving time. Since t_s is one second, this means that the solving thread must be suspended and resumed a few times in order for this heuristic to completely solve an instance. As t_l increases this heuristic gets created sooner but the delays between suspending and resuming the solving thread also get longer (they are length t_l), and for a sufficiently large t_l the increase in the delays between solving episodes outweigh the advantage of creating the heuristic sooner.
- The mean total times for the best ratio of all three strategies is similar (less than 10% difference).

The mean total time spent on a target instance (including the learning time)—under 16 minutes (960 seconds)—is substantially lower than the total time spent by our bootstrap system using a large set of bootstrap instances but no interleaving. According to Tables 3 and 4, the latter requires more than 11 hours when using 500 bootstrap instances and almost 5 days when using 5000 bootstrap instances. Alternatively, to minimize learning time one could consider using the heuristics

Table 16
Solving a single instance of the 35-pancake puzzle.

Row	Ratio (t_s : t_l)	min	max	mean	med	std	Subopt.
Immediate restart							
1	1:1	2 h 36 m	6 h 48 m	3 h 41 m	3 h 24 m	46 m	7.5%
2	1:2	2 h 07 m	5 h 18 m	2 h 54 m	2 h 39 m	40 m	8.0%
3	1:5	1 h 45 m	5 h 57 m	2 h 38 m	2 h 26 m	48 m	8.2%
4	1:6 (best)	1 h 37 m	4 h 29 m	2 h 07 m	2 h 07 m	33 m	8.3%
Heuristic replacement							
5	1:1	2 h 34 m	6 h 45 m	3 h 36 m	3 h 24 m	42 m	7.6%
6	1:2	2 h 06 m	5 h 15 m	2 h 50 m	2 h 39 m	38 m	8.0%
7	1:5	1 h 45 m	5 h 00 m	2 h 34 m	2 h 19 m	42 m	8.2%
8	1:8 (best)	1 h 39 m	4 h 54 m	2 h 29 m	2 h 14 m	42 m	8.1%
Independent solvers (exponential)							
9	1:1	7 h 13 m	9 h 48 m	7 h 36 m	7 h 28 m	30 m	7.8%
10	1:2	5 h 24 m	7 h 28 m	5 h 45 m	5 h 32 m	24 m	8.0%
11	1:5	4 h 19 m	6 h 42 m	4 h 45 m	4 h 39 m	28 m	8.3%
12	1:9 (best)	1 h 50 m	7 h 01 m	3 h 23 m	3 h 28 m	50 m	8.9%

created by the first bootstrap iteration. With 5000 bootstrap instances, this heuristic solves instances considerably slower, on average, than the interleaving methods (2364.9 seconds, or about 39 minutes—see iteration 0 in Table 4—compared to under 16 minutes). The heuristic created on the first iteration of bootstrapping with 500 bootstrap instances solves instances faster than interleaving (153.34 seconds, or about 2.5 minutes—see iteration 0 in Table 3), but it takes 98 minutes to learn this heuristic (see the “Learning time” column for iteration 0 in Table 3). Therefore, our method to solve a target instance has substantial speedup over the normal bootstrap method. Our method also fares well in comparison to the systems reported in Table 5. It dominates W-IDA* ($W = 1.5$) and has a suboptimality superior to BULB ($B = 20{,}000$) and far superior to h_{sum}. Its total time (under 960 seconds) is less than that of any of the optimal methods. Its time is inferior to that of the weighted RBFS system reported in line 9 of Table 5 but its suboptimality is superior. Comparisons with the PE-ANN system in lines 10 and 11 of Table 5 are not possible because the training times for that system are unknown.

If Bootstrap is given a strong initial heuristic h_0 (the maximum of disjoint 6-6-6-6 PDBs and their reflections), the total times are similar to those reported in Table 15, but the suboptimality reduces to roughly 4% for all the interleaving strategies.

4.2.2. 35-Pancake puzzle

Table 16 provides detailed results for the 35-pancake puzzle. The trends observed in this experiment are similar to those observed for the 24-puzzle except here the Independent Solvers strategy has mean and median times that are considerably higher than those of the other two strategies.

The suboptimality of the heuristics produced by any of the interleaving strategies is superior to any of the suboptimalities reported for basic bootstrapping in Tables 6 and 7, and the mean total solving time for the interleaving strategies are less than half the time required to finish the first bootstrapping iteration with 500 bootstrap instances (7 hours—see Table 6). In Table 8 we see that instances are solved much more quickly using h_{sum}, W-IDA* ($W = 9$), or BULB ($B \leqslant 20{,}000$), than using any of our interleaving methods, but with much greater suboptimality.

If Bootstrap is given a strong initial heuristic h_0 (a 5-5-5-5-5-5-5 additive PDB), the total times are slightly smaller than in Table 16 and suboptimality decreases to around 4.5% for all the interleaving strategies.

4.2.3. Rubik’s Cube

Table 17 shows the experimental results for Rubik’s Cube. As for the 35-pancake puzzle, the Independent Solvers strategy has mean and median times that are considerably higher than the other two strategies.

The reason the suboptimality is the same for all variations tested is that in all cases, almost all the instances are solved using the third heuristic created by bootstrapping. This happens because h_0 and the first two learned heuristics are very weak, and too much time (25 hours) is needed to learn the fourth heuristic. This may also explain why the best ratio here is smaller than for the other domains.

Although interleaving is very much superior to the basic bootstrapping process when there is only a single Rubik’s Cube instance to solve, the best mean total time in Table 17 (10 hours and 54 minutes, or 39,240 seconds) is only 11% better than the time required to solve an average instance optimally using the best known heuristic for Rubik’s Cube (44,201 seconds, see Row 6 of Table 11). However, all the interleaving strategies shown in Table 17 outperform simply using our initial heuristic to solve each instance, which requires 102,362 seconds (28 hours and 26 minutes) on average (see Row 4 of Table 11). Heuristic Replacement with a 1:5 ratio dominates W-IDA* which, with $W = 1.4$, requires more time to solve an instance (12 hours and 36 minutes on average) and produces a greater suboptimality (13.3% compared to 6.4%).

Table 17
Solving a single instance of Rubik's Cube.

Row	Ratio (t_s : t_l)	min	max	mean	med	std	Subopt.
Immediate restart							
1	1:1	0 h 41 m	19 h 27 m	13 h 29 m	13 h 50 m	4 h 54 m	6.4%
2	1:2 (best)	1 h 01 m	19 h 48 m	11 h 42 m	11 h 18 m	4 h 45 m	6.4%
3	1:5	2 h 01 m	28 h 01 m	13 h 08 m	11 h 10 m	7 h 02 m	6.4%
Heuristic replacement							
4	1:1	0 h 41 m	18 h 43 m	13 h 21 m	13 h 28 m	4 h 59 m	6.4%
5	1:2	1 h 01 m	17 h 54 m	11 h 18 m	10 h 42 m	4 h 45 m	6.4%
6	1:5 (best)	2 h 01 m	19 h 41 m	10 h 54 m	9 h 16 m	5 h 24 m	6.4%
Independent solvers (exponential)							
7	1:1	1 h 22 m	26 h 16 m	15 h 36 m	15 h 02 m	6 h 34 m	6.4%
8	1:2 (best)	2 h 02 m	29 h 54 m	14 h 51 m	13 h 04 m	4 h 45 m	6.4%
9	1:5	4 h 03 m	29 h 29 m	17 h 23 m	14 h 45 m	4 h 30 m	6.4%

Table 18
Solving a single instance of the 20-blocks world.

Row	Ratio (t_s : t_l)	min	max	mean	med	std	Subopt.
Immediate restart							
1	1:1	17 m	25 h 38 m	5 h 18 m	1 h 22 m	7 h 28 m	1.3%
2	1:2	17 m	19 h 24 m	4 h 28 m	1 h 28 m	5 h 53 m	1.3%
3	1:5	10 m	15 h 58 m	4 h 15 m	1 h 24 m	5 h 15 m	1.3%
4	1:9 (best)	26 m	15 h 48 m	4 h 01 m	2 h 00 m	4 h 40 m	1.3%
Heuristic replacement							
5	1:1	17 m	25 h 35 m	5 h 00 m	1 h 23 m	7 h 17 m	1.3%
6	1:2	17 m	19 h 22 m	4 h 16 m	1 h 16 m	5 h 46 m	1.3%
7	1:5 (best)	10 m	15 h 55 m	4 h 06 m	1 h 22 m	5 h 12 m	1.3%
Independent solvers (exponential)							
8	1:1	17 m	25 h 52 m	6 h 04 m	2 h 03 m	7 h 38 m	1.3%
9	1:2	17 m	19 h 44 m	4 h 52 m	1 h 52 m	5 h 58 m	1.3%
10	1:5 (best)	10 m	15 h 48 m	3 h 49 m	1 h 24 m	4 h 45 m	1.3%

4.2.4. 20-Blocks world

Table 18 shows the experimental results for the 20-blocks world. In each case the solutions were only 1.3% longer than optimal, on average, and at least 37 of the 50 instances were solved optimally. Unlike the previous domains, here the Independent Solvers strategy slightly outperforms the others in terms of mean total time.

In this experiment, our initial heuristic is so weak that it takes a few iterations of RandomWalk until the heuristic becomes sufficiently strong that the solver using it can solve the instance in a reasonable amount of time. After this point, for a few iterations, the learned heuristics enable the instances to be solved more quickly without changing the solution quality. For this reason, we observe a constant suboptimality of 1.3% for all different strategies.

The speedup compared to the initial bootstrap method (which needed 2 days when using 500 bootstrap instances and 11 days when using 5000 bootstrap instances) is again remarkable. In addition, the solution lengths are much closer to optimal than before (*cf.* Tables 12 and 13 for Bootstrap results on the 20-blocks world).

BULB with $B \leqslant 20{,}000$ would solve a single instance faster than our method (see Table 14), but even for $B = 20{,}000$ the suboptimality would be more than 20 times higher than that of Bootstrap. For W-IDA* we were unable to find a value of W that can solve all the test instances in a reasonable amount of time with a suboptimality close to our interleaving method so we base our comparison on the instances solved using a 2-hour time limit per instance. Heuristic Replacement with a ratio of 1:5 solved 27 of the 50 instances with this time limit and its solutions were just 2.4% suboptimal. With this time limit W-IDA* ($W = 4$) also solved 27 instances but its suboptimality was 150%.

4.2.5. IPC2 blocks world instances

We further tested our interleaving technique on the 35 instances of blocks world domains of varying size used in Track 1 of the IPC2 planning competition.[9] This version of the blocks world has a "hand" that is used to pick up and put down blocks, as opposed to the "handless" version we have used in the 20-blocks world experiments elsewhere in this paper. Despite this difference we used the same features and initial heuristic here as in the previous experiments with the "handless" 20-blocks world.

[9] See http://www.cs.toronto.edu/aips2000/ for more details.

Table 19
IPC2 blocks world instances, results for interleaving using the exponential allocation strategy.

Instance	Optimal	Ratio (1:1)		Ratio (1:2)		Ratio (1:5)	
		Time	Subopt.	Time	Subopt.	Time	Subopt.
9-0	30						
9-1	28						
9-2	26						
10-0	34	22		33		65	
10-1	32					2	
10-2	34	13		19		38	
11-0	32	73		58		47	
11-1	30	58		42		46	
11-2	32	2		3		6	
12-0	34	12		18		37	
12-1	34	3		4		9	
13-0	42	1451	4.8%	1102	4.8%	914	4.8%
13-1	44	170		1024		861	
14-0	38	23		51		67	
14-1	36	62		73		176	
15-0	40	313	5%	310	5%	242	5%
15-1	52	627		475		393	
16-1	54	1347		1271		1105	
16-2	52	1001		751		603	
17-0	48	331		258		230	

Table 19 shows the results on the hardest 20 instances of the IPC2 set. The first column names the instances, where x–y refers to the yth instance that consists of x blocks. The other columns show the total time (in seconds) and suboptimality achieved by our interleaving method using the exponential allocation strategy. In this table, empty "Time" entries indicate that the total time was below 0.1 seconds and empty "Subopt." entries indicate the instance was solved optimally. The 15 instances with the fewest blocks (between 4 and 8) are not shown; all were solved optimally by our system in less than 0.1 seconds. Table 19 shows that our interleaving method is capable of solving all the instances in less than 30 minutes (the time limit for solving an instance in the IPC2 competition) while the solutions are almost always optimal, and those that are not optimal are very close to optimal.

The Fast Downward planner [19], with the setting[10] that uses multi-heuristic best-first search[11] and preferred operators, also solved all 35 of these blocks world instances. It took Fast Downward, on average, less than a second to solve each instance, and its solutions are 200% suboptimal.[12] Our interleaving approach required much more time (about 138 seconds, on average) but found solutions that were only 0.3% suboptimal.

The FF planner [23] solved 29 of the 35 instances in the time limit of 30 minutes. The solutions generated for the solved instances were 2.3% suboptimal and it took less than 6 seconds, on average, for FF to solve one of these 29 instances. These 29 instances were all solved optimally by our interleaving approach but in a greater amount of time (25 seconds) on average. Of course, our method was also able to solve the 6 problems in 30 minutes that FF could not.

The best performing optimal planners from IPC5 such as Gamer and HSP^*_F[13] solved 30 of the 35 instances within a time limit of 30 minutes [21].[14] Furthermore, the landmark cut heuristic [21], which is competitive with the state-of-the-art optimal planners in overall performance, solved 28 of the 35 instances within the same time limit. Its average solving time on these 28 instances was 76 seconds. Our interleaving approach also solved these 28 instances optimally, and did so in less than 19 second on average.

Yoon, Fern, and Givan [49] report two sets of results on these blocks world instances. One set is for a method they present for learning a heuristic function to guide planning, the other is for a method they present for learning a decision list policy to guide planning. In both cases they learned from the solutions found by solving the easiest 15 instances (the ones not shown in Table 19) with FF's heuristic and then solved the remaining 20 instances with the learned heuristic/policy in conjunction with FF's heuristic. Their methods are therefore "one step" methods, they are not methods aimed at solving single instances quickly. The learning phase for the method that learned a heuristic took 600 seconds. They then took 12.94 seconds, on average, to solve each of the 20 test instances (all 20 were solved within the 30-minute time limit). The solutions found had an average suboptimality of 120%.[15] Our interleaving method solved the same instances in 242 seconds,

[10] This setting is referred to as "$\mathbf{M}+\mathbf{P}$" in Helmert's paper [19].

[11] This search algorithm is a best-first search algorithm that alternates between expanding nodes from different open lists that are sorted based on different heuristics [39]. Here, the casual graph heuristic [19] and FF's relaxed plan heuristic [23] are used.

[12] All results for the planning systems discussed here are taken from the papers cited.

[13] See http://ipc.informatik.uni-freiburg.de/ for more details about the competition and the planners.

[14] Neither the solving time nor the instances solved is reported for these two planners.

[15] Fern et al. computed average suboptimality differently than we have defined it in this paper. They defined average suboptimality as the total length of the solutions found divided by the total length of the optimal solutions. In this paragraph, we use their method to compute the suboptimality of our systems to allow a comparison to be made.

on average, with almost optimal solutions (the average suboptimality was 0.5%). Our single-instance method could therefore solve approximately 3 problems from scratch in the same time that their method could perform learning once and solve 3 problems. Their method for learning a decision list policy took less time to learn and solve a single instance than our method (100.05 seconds, on average) but produced longer solutions (their average suboptimality was 17%).

5. Related work

Bootstrap learning to iteratively improve an initially weak evaluation function for single-agent search is an idea due to Rendell [37,38], who used it to enable unidirectional search to solve random instances of the 15-puzzle for the first time. Our method differs from Rendell's in several key details, the most important being that Rendell assumed the user would provide a set of bootstrap instances for each iteration, at least one of which was required to be solvable using the current evaluation function. We, on the other hand, assume that the entire set of bootstrap instances is given at the outset, and if the initial system cannot solve any of them it generates its own instances.

The only other study of bootstrap learning of heuristics is due to Humphrey, Bramanti-Gregor, and Davis [25]. Their SACH system learns a heuristic to solve a single instance, and the bootstrapping is done over successive failed attempts to solve the instance. Impressive results were obtained on the fifteen most difficult standard 100 instances of the 15-puzzle. On average these instances were solved by A* with only 724,032 nodes generated in total over all of SACH's iterations, and the solutions found were only 2% suboptimal.

Hauptman et al. [17,18] use genetic programming [33] to iteratively improve an initial population of heuristic functions. The key difference between this and bootstrapping is that it creates new heuristics by mutating and recombining heuristics in its current population rather than learning a new heuristic from solved instances. Training instances in their system (the analog of our bootstrap instances) are used only for evaluating the fitness of the newly created heuristics. The main application to date has been to the standard 6 × 6 Rush Hour puzzle, which is sufficiently small (3.6×10^{10}) that most instances can be solved quickly even without a heuristic, hence guaranteeing that the evaluation of fitness will succeed in distinguishing better heuristics from worse ones. The heuristic learned by their system reduced the number of nodes generated by an IDA* variant by a factor of 2.5 compared to search with no heuristic. The time required for learning and the suboptimality of the solutions generated were not reported. They have also used FreeCell as testbed [17], but in that application a policy was evolved to guide search, not a heuristic function.

The online learning of heuristics as studied by Fink [13] is also related to bootstrapping. 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.

Thayer, Dionne, and Ruml [47] used online learning to update an initial heuristic function during a greedy best-first search that aims at solving a specific instance of a search problem. They computed the error of the heuristic after each node is expanded by the search algorithm. The error is defined as the difference between the heuristic value of the state and the sum of the heuristic value of the child with the largest heuristic estimate and the cost of the action that generates the child. This error estimate is then used to update the heuristic function during search. Their experimental results showed that when such an update is used with the greedy best-first search, it can improve the performance of the initial heuristic in terms of both solution quality and solving time. For example, the heuristic created by this system improves over Manhattan Distance in the 15-puzzle by a factor of about 3 in terms of solution cost and by factor of about 2 in terms of time needed to solve each problem instance. In their experiments, Thayer et al. included a system, "ANN-offline, which learns a heuristic in a one-step manner resembling the system of Ernandes and Gori [9]. This produced a heuristic that was much more accurate than the heuristic learned by their online method but, interestingly, much poorer at guiding greedy best-first search. As they observe, this highlights the different requirements for heuristics that are used for pruning unpromising paths, which is the focus of our paper, compared to heuristics that are used to determine the order in which paths are to be explored.

Other systems for learning heuristics limit themselves to just one step of what could be a bootstrapping process [9,35,41,43,44,49]. Such systems typically assume the initial heuristic (h_0) is sufficiently strong that arbitrary instances can be solved with it, and use learning to create a better heuristic, *i.e.*, one that allows instances to be solved more quickly than with h_0 although perhaps with greater suboptimality. If our bootstrap method is given an initial heuristic as strong as these systems require, it performs the same as they do, *i.e.*, it performs only one iteration and produces an improved heuristic without introducing much suboptimality. For example, on the 15-puzzle Samadi et al.'s one-step system [41] creates a heuristic that allows solutions to random solvable instances to be found by RBFS after generating only 2241 nodes, on average, and the solutions found are only 3.3% longer than optimal. Our system, if supplied with an initial heuristic comparable in strength to Samadi et al.'s initial heuristic, terminates after one iteration with a heuristic that allows solutions to the same instances to be found by RBFS after generating only 9402 nodes, and the solutions found are only 0.5% longer than optimal [27]. Of course, our bootstrapping method has the advantage over these systems that it does not require a strong initial heuristic; it will succeed even if given an initial heuristic so weak that it cannot solve any of the bootstrap instances in a reasonable amount of time.

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, Yoon, and Givan [12] used random walks in learning policies to control a Markov Decision Process, and Finkelstein and Markovitch [14] 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.

6. Conclusions

This paper gives experimental evidence that machine learning can be used to create strong heuristics from very weak ones through an automatic, incremental bootstrapping process augmented by a random walk method for generating successively more difficult problem instances. Our system was tested on four problem domains that are at or beyond the limit of current abstraction methods and in each case it successfully created heuristics that enable IDA* to solve randomly generated test instances quickly and almost optimally. The total time needed for this 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.

The total time for the bootstrap process to create strong heuristics for these large state spaces is on the order of days. This is acceptable when the learning time can be amortized over a large number of test instances. To make heuristic learning effective when only a single problem instance needs to be solved, we presented a variation in which the bootstrap learning of new heuristics is interleaved with problem-solving using the initial heuristic and whatever heuristics have been learned so far. When tested on the same four domains, this method was shown to substantially reduce the total time needed to solve a single instance while still producing solutions that are very close to optimal. When applied to the blocksworld instances used in the IPC2 planning competition, our interleaving method solved all the instances within the 30-minute time limit, and almost all were solved optimally.

Acknowledgements

Thanks to Neil Burch, Richard Valenzano, Mehdi Samadi, Fan Yang, Uzi Zahavi, and Ariel Felner for sharing their code, Jonathan Schaeffer for suggesting the ideas of heuristic replacement and immediate restart, reviewers for their insightful comments, the Alberta Ingenuity Centre for Machine Learning, and NSERC.

References

[1] Aaron F. Archer, A modern treatment of the 15-puzzle, American Mathematical Monthly 106 (1999) 793–799.

[2] Ricardo A. Baeza-Yates, Joseph C. Culberson, Gregory J.E. Rawlins, Searching in the plane, Information and Computation 106 (2) (1993) 234–252.

[3] Marcel Ball, Robert C. Holte, The compression power of symbolic pattern databases, in: Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS 2008), 2008, pp. 2–11.

[4] Blai Bonet, Héctor Geffner, Planning as heuristic search, Artificial Intelligence 129 (2001) 5–33.

[5] Joseph C. Culberson, Jonathan Schaeffer, Searching with pattern databases, in: Proceedings of the Canadian Conference on Artificial Intelligence, in: LNAI, vol. 1081, Springer, 1996, pp. 402–416.

[6] Harry Dweighter, Problem E2569, American Mathematical Monthly 82 (1975) 1010.

[7] Stefan Edelkamp, Symbolic pattern databases in heuristic search planning, in: Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS 2002), 2002, pp. 274–283.

[8] Stefan Edelkamp, Shahid Jabbar, Peter Kissmann, Scaling search with pattern databases, in: Proceedings of the 5th International Workshop on Model Checking and Artificial Intelligence (MoChArt), in: LNCS, vol. 5348, Springer, 2009, pp. 49–65.

[9] Marco Ernandes, Marco Gori, Likely-admissible and sub-symbolic heuristics, in: Proceedings of the 16th European Conference on Artificial Intelligence (ECAI 2004), 2004, pp. 613–617.

[10] Ariel Felner, Amir Adler, Solving the 24 puzzle with instance dependent pattern databases, in: Proceedings of the 6th International Symposium on Abstraction, Reformulation and Approximation (SARA 2005), in: LNCS, vol. 3607, Springer, 2005, pp. 248–260.

[11] Ariel Felner, Uzi Zahavi, Jonathan Schaeffer, Robert C. Holte, Dual lookups in pattern databases, in: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI, 2005), pp. 103–108.

[12] Alan Fern, Sungwook Yoon, and Robert Givan, Learning domain-specific control knowledge from random walks, in: Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS 2004), 2004, pp. 191–199.

[13] Michael Fink, Online learning of search heuristics, in: Proceedings of the 11th International Conference on Artificial Intelligence and Statistics (AISTATS 2007), 2007, pp. 114–122.

[14] Lev Finkelstein, Shaul Markovitch, A selective macro-learning algorithm and its application to the $N \times N$ sliding-tile puzzle, Journal of Artificial Intelligence Research 8 (1998) 223–263.

[15] David Furcy, Sven König, Limited discrepancy beam search, in: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), 2005, pp. 125–131.

[16] Patrik Haslum, Héctor Geffner, Admissible heuristics for optimal planning, in: Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS 2000), 2000, pp. 140–149.

[17] Ami Hauptman, Achiya Elyasaf, Moshe Sipper, Evolving hyper heuristic-based solvers for Rush Hour and FreeCell, in: Proceedings of the 3rd Annual Symposium on Combinatorial Search (SoCS 2010), 2010, pp. 149–150.

[18] Ami Hauptman, Achiya Elyasaf, Moshe Sipper, Assaf Karmon, GP-rush: using genetic programming to evolve solvers for the Rush Hour puzzle, in: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO 2009), ACM, New York, NY, USA, 2009, pp. 955–962.

[19] Malte Helmert, The Fast Downward planning system, Journal of Artificial Intelligence Research 26 (2006) 191–246.

[20] Malte Helmert, Landmark heuristics for the pancake problem, in: Proceedings of the 3rd Annual Symposium on Combinatorial Search (SoCS 2010), 2010, pp. 109–110.

[21] Malte Helmert, Carmel Domshlak, Landmarks, critical paths and abstractions: What's the difference anyway? in: Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009), 2009, pp. 162–169.

[22] Malte Helmert, Gabriele Röger, Relative-order abstractions for the pancake problem, in: Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010), 2010, pp. 745–750.

[23] Jörg Hoffmann, Bernhard Nebel, The FF planning system: Fast plan generation through heuristic search, Journal of Artificial Intelligence Research 14 (2001) 253–302.

[24] Robert C. Holte, Jeffery Grajkowski, Brian Tanner, Hierarchical heuristic search revisited, in: Proceedings of the 6th International Symposium on Abstraction, Reformulation and Approximation (SARA 2005), in: LNAI, vol. 3607, Springer, 2005, pp. 121–133.

[25] Timothy Humphrey, Anna Bramanti-Gregor, Henry W. Davis, Learning while solving problems in single agent search: Preliminary results, in: Proceedings of the 4th Congress of the Italian Association for Artificial Intelligence (AI*IA 1995), in: LNCS, vol. 992, Springer, 1995, pp. 56–66.

[26] Shahab Jabbari Arfaee, Bootstrap learning of heuristic functions, Master's thesis, Computing Science Department, University of Alberta, 2010.

[27] Shahab Jabbari Arfaee, Sandra Zilles, Robert C. Holte, Bootstrap learning of heuristic functions, in: Proceedings of the 3rd Annual Symposium on Combinatorial Search (SoCS 2010), 2010, pp. 52–60.

[28] David G. Kirkpatrick, Hyperbolic dovetailing, in: Proceedings of the 17th Annual European Symposium on Algorithms (ESA 2009), in: LNCS, vol. 5757, Springer, 2009, pp. 516–527.

[29] Richard E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, Artificial Intelligence 27 (1) (1985) 97–109.

[30] Richard E. Korf, Linear-space best-first search: Summary of results, in: Proceedings of the 10th AAAI Conference on Artificial Intelligence (AAAI 1992), 1992, pp. 533–538.

[31] Richard E. Korf, Finding optimal solutions to Rubik's Cube using pattern databases, in: Proceedings of the 14th AAAI Conference on Artificial Intelligence (AAAI 1997), 1997, pp. 700–705.

[32] Richard E. Korf, Ariel Felner, Disjoint pattern database heuristics, Artificial Intelligence 134 (2002) 9–22.

[33] John R. Koza, Genetic Programming II: Automatic Discovery of Reusable Programs, MIT Press, Cambridge MA, May 1994.

[34] Judea Pearl, Heuristics, Addison–Wesley, 1984.

[35] Marek Petrik, Shlomo Zilberstein, Learning heuristic functions through approximate linear programming, in: Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS 2008), 2008, pp. 248–255.

[36] Armand Prieditis, Machine discovery of effective admissible heuristics, Machine Learning 12 (1993) 117–141.

[37] Larry A. Rendell, Details of an automatic evaluation function generator for state-space problems, Technical Report CS-78-38, Department of Computer Science, University of Waterloo, 1978.

[38] Larry A. Rendell, A new basis for state-space learning systems and a successful implementation, Artificial Intelligence 20 (1983) 369–392.

[39] Gabriele Röger, Malte Helmert, The more, the merrier: Combining heuristic estimators for satisficing planning, in: Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS 2010), 2010, pp. 246–249.

[40] David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams, Learning internal representations by error propagation, in: Parallel Distributed Processing: Explorations in the Microstructure of Cognition, MIT Press, Cambridge, MA, USA, 1986, pp. 318–362.

[41] Mehdi Samadi, Ariel Felner, Jonathan Schaeffer, Learning from multiple heuristics, in: Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI 2008), 2008, pp. 357–362.

[42] Mehdi Samadi, Maryam Siabani, Ariel Felner, Robert Holte, Compressing pattern databases with learning, in: Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008), 2008, pp. 495–499.

[43] Sudeshna Sarkar, Partha P. Chakrabarti, Sujoy Ghose, Learning while solving problems in best first search, IEEE Transactions on Systems, Man, and Cybernetics, Part A 28 (1998) 535–541.

[44] Sudeshna Sarkar, Sujoy Ghose, Partha P. Chakrabarti, Learning for efficient search, Sadhana: Academy Proceedings in Engineering Sciences 2 (1996) 291–315.

[45] John Slaney, Sylvie Thiébaux, Blocks world revisited, Artificial Intelligence 125 (2001) 119–153.

[46] Jerry Slocum, Dic Sonneveld, The 15 Puzzle, Slocum Puzzle Foundation, 2006.

[47] Jordan Thayer, Austin Dionne, Wheeler Ruml, Learning inadmissible heuristics during search, in: Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS 2011), 2011, pp. 250–257.

[48] Fan Yang, Joseph Culberson, Robert Holte, Uzi Zahavi, Ariel Felner, A general theory of additive state space abstractions, Journal of Artificial Intelligence Research 32 (2008) 631–662.

[49] Alan Fern Sungwook Yoon, Robert Givan, Learning control knowledge for forward search planning, Journal of Machine Learning Research 9 (2008) 683–718.

[50] Uzi Zahavi, Ariel Felner, Robert Holte, Jonathan Schaeffer, Dual search in permutation state spaces, in: Proceedings of the 21st AAAI Conference on Artificial Intelligence (AAAI 2006), 2006, pp. 1076–1081.

[51] Uzi Zahavi, Ariel Felner, Robert C. Holte, Jonathan Schaeffer, Duality in permutation state spaces and the dual search algorithm, Artificial Intelligence 172 (4–5) (2008) 514–540.

[52] Rong Zhou, Eric A. Hansen, External-memory pattern databases using structured duplicate detection, in: Proceedings of the 20th AAAI Conference on Artificial Intelligence (AAAI 2005), 2005, pp. 1398–1405.