Learning to Combine Admissible Heuristics Under Bounded Time

Carmel Domshlak and **Erez Karpas**
Faculty of Industrial Engineering & Management
Technion, Israel

Shaul Markovitch
Faculty of Computer Science
Technion, Israel

Abstract

Usually, combining admissible heuristics for optimal search (e.g. by using their maximum) requires computing numerous heuristic estimates at each state. In many cases, the cost of computing these heuristic estimates outweighs the benefit in the reduced number of expanded states. If only state expansions are considered, this is a good option. However, if time is of the essence, we can do better than that. We propose a novel method that reduces the cost of combining admissible heuristics for optimal planning, while maintaining its benefits. We first describe a simplified model for deciding which heuristic is best to compute at each state. We then formulate an active learning approach to decide which heuristic to compute at each state, online, during search. The resulting technique, which we call *selective max* is evaluated empirically, and is shown to outperform each of the individual heuristics that were used, as well as their regular maximum, in terms of number of solved instances and average solution time.

Introduction

One of the most prominent approaches to cost-optimal planning (and cost-optimal search in general) is using the A^* search algorithm with an admissible heuristic. Many admissible heuristics for domain-independent planning have been proposed (Bonet and Geffner 2001; Helmert, Haslum, and Hoffmann 2007; Haslum et al. 2007; Coles et al. 2008; Katz and Domshlak 2008; Karpas and Domshlak 2009; Helmert and Domshlak 2009; Katz and Domshlak 2009), varying from cheap to compute and not very informative (Bonet and Geffner 2001) to expensive to compute and very informative (Katz and Domshlak 2008).

Although some heuristics perform better than others on some planning problems, it is difficult to choose a clear-cut "best" heuristic for domain-independent planning in general. Sometimes it is even difficult to choose the best heuristic for a specific planning domain, as different heuristics perform differently on different problem instances in the same domain. Furthermore, it is very hard (if not impossible) to predict how well a given heuristic will perform on a new domain.

One way of producing a more robust planner is by combining several heuristics. The simplest way of doing this is by using their point-wise maximum at each state. Presumably, each heuristic is better (i.e. has a higher estimate) in different regions of the search space, and thus the maximum is more informative than each of the individual heuristics. Another way to combine several heuristics is to use additive heuristics (Katz and Domshlak 2008; Haslum, Bonet, and Geffner 2005; Felner, Korf, and Hanan 2004) or even use both addition and maximum (Coles et al. 2008; Haslum et al. 2007). The problem with both max-based and sum-based approaches is that sometimes the cost of computing numerous heuristic estimates at each state outweighs the benefit in the reduced number of states expanded.

While performing additive combination of heuristics requires computing all of their values, combining heuristics using their maximum does not. Observe that, in every state, the maximum value comes from one of the heuristics. If we had an oracle indicating the most informed heuristic at each state, then computing only that heuristic would result in the same search behavior as max-based combination. However, even if we had such an oracle, it is possible that the extra time spent on computing the more informed heuristic may not be worth the reduction in expanded states. In fact, the results from the last International Planning Competition (IPC-2008) show that it is pretty hard to beat blind search (i.e. A^* with a heuristic which returns 0 for goal states and 1 for non-goal states) – the least informative, and at the same time fastest to compute, heuristic possible.

In this paper we propose a novel method that reduces the cost of combining admissible heuristics, while maintaining its benefits, which we call *selective max*. We first describe a simplified formal model for deciding which heuristic to compute at each state in order to reduce the total search time. We then describe an online active learning scheme that uses a decision rule, derived from our formal model, to decide which heuristic to compute at every state, and can therefore be seen as a *hyper-heuristic* (Burke et al. 2003). An experimental evaluation of our approach using the h_{LA} heuristic (Karpas and Domshlak 2009) and the $h_{\text{LM-CUT}}$ heuristic (Helmert and Domshlak 2009) shows that it solves more planning problems than both of these heuristics individually, as well as their max-based combination, and does it faster on average. Our approach is also shown to exhibit better anytime behavior.

Notation

We consider planning in the $\textsc{sas}^+$ formalism (Bäckström and Nebel 1995); a $\textsc{sas}^+$ description of a planning task

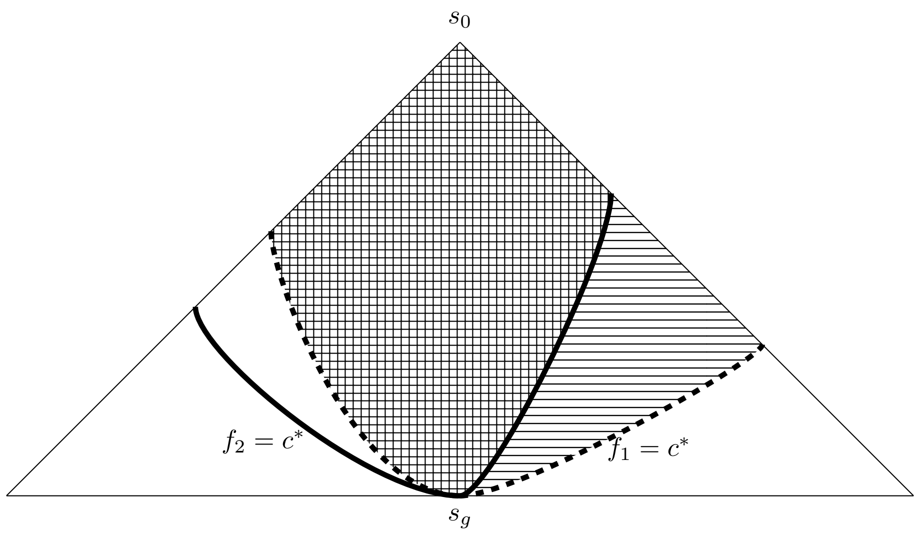

Figure 1: State Space Tree

can be automatically generated from its PDDL description (Helmert 2009). A SAS$^+$ task is given by a 4-tuple $\Pi = \langle V, A, s_0, G \rangle$. $V = \{v_1, \ldots, v_n\}$ is a set of *state variables*, each associated with a finite domain $dom(v_i)$. Each complete assignment s to V is called a *state*; s_0 is an *initial state*, and the *goal* G is a partial assignment to V. A is a finite set of *actions*, where each action a is a pair $\langle \mathsf{pre}(a), \mathsf{eff}(a) \rangle$ of partial assignments to V called *preconditions* and *effects*, respectively.

An action a is applicable in a state $s \in dom(V)$ iff $\mathsf{pre}(a) \subseteq s$, and applying such a changes the value of each state variable v to $\mathsf{eff}(a)[v]$ if $\mathsf{eff}(a)[v]$ is specified. The resulting state is denoted by $s[\![a]\!]$; by $s[\![\langle a_1, \ldots, a_k \rangle]\!]$ we denote the state obtained from sequential application of the (respectively applicable) actions $a_1, \ldots, a_k$ starting at state s. Such an action sequence is a plan if $G \subseteq s[\![\langle a_1, \ldots, a_k \rangle]\!]$.

Model for Heuristic Selection

In order to analyze when computing one admissible heuristic should be preferred to computing another, we need to make certain assumptions about the state space. We assume that the state-space is a tree with a constant branching factor b, uniform cost actions, and that there exists a single goal state (as in the classical results on heuristic-search effort (Pearl 1984)). We also assume that the heuristics are consistent, and that the time t_i required for computing h_i for a state s is independent of s. These assumption do not hold in many planning problems, but we explain how to deal with that in the next section.

A^* expands states according to $f = g + h$, where $g(s)$ is the length of the shortest path to s, and $h(s)$ is the heuristic estimate of the distance from s to the goal. Suppose we have two admissible heuristics: h_1 and h_2. Define $max_h(s) \triangleq \max(h_1(s), h_2(s))$, we then use the notation $f_1 = g + h_1$, $f_2 = g + h_2$, and $\max_f = g + \max_h$.

Assuming the goal state is at depth c^*, let us consider the states satisfying $f_1(s) = c^*$ (the dotted line in Fig. 1) and those satisfying $f_2(s) = c^*$ (the solid line in Fig. 1). The states above the $f_1 = c^*$ and $f_2 = c^*$ contours are those that are surely expanded by A^* with h_1 and h_2, respectively. The states above both these contours (the grid-marked region in Fig. 1) are those that are surely expanded by max_h. These states, which we denote SE (for "surely expanded"), are $SE = \{s \mid \max_f(s) < c^*\}$.

Observe that the optimal decision for any state $s \in SE$ is not to compute any heuristic at all, because all these states are surely expanded anyway. The optimal decision for states that are not surely expanded by max_h is a bit more complicated. Let us consider the states where $f_1(s) < c^*$ and $f_2(s) = c^*$ (that is, in Fig. 1, the states on part of the $f_2 = c^*$ contour that separates between grid-marked and lines-marked areas). Since $g(s)$ is the same in $f_1(s)$ and $f_2(s)$, we know that $h_2(s) > h_1(s)$ (i.e. h_2 is more informative in state s). If we consider only state expansions, h_2 would be the best heuristic to compute at state s. However, h_2 could be more expensive to compute than h_1, so if we consider search time, the choice is not straightforward.

If we compute $h_2(s)$, then s is no longer surely expanded (because $f_2(s) = c^*$, and therefore it may be expanded or not, according to tie-breaking). In contrast, if we compute $h_1(s)$, then s is surely expanded, because $f_1(s) < c^*$. Note that computing h_2 for one of the descendants of s is surely sub-optimal, as we pay the cost of computing h_2, yet only part of the search sub-tree rooted in s is no longer surely expanded. Therefore, our choices can be restricted only to either computing h_2 for s, or computing h_1 for all the states in the sub-tree rooted at s on the $f_1 = c^*$ line (i.e. the leaves of the sub-tree rooted at s and ending at the $f_1 = c^*$ line).

Assume we need to expand l complete levels of the state-space from s to reach the $f_1 = c^*$ line. This means we need to generate on the order of b^l states, and then calculate h_1

for all of the states on the $f_1 = c^*$ line, which would take $b^l t_1$ time. If, instead, we compute h_2, the time required to "explore" the sub-tree rooted in s would be t_2 (assuming favorable tie-breaking). Given that, the optimal decision is thus to calculate h_2 iff $t_2 < b^l t_1$, or if we rewrite this, if $l > \log_b(t_2/t_1)$. As a special case, if both heuristics take the same time to compute, this decision rule boils down to $l > 0$, that is, the optimal choice is simply the more informed (for state s) heuristic.

From Model to Practice

The above simplified formal model for deciding when it is best to evaluate which heuristic makes several assumptions, some of which appear to be problematic to meet in practice. Here we examine these assumptions more closely, suggest pragmatic compromises when possible, and then describe an algorithm, called *selective max*, for speeding up heuristic-search optimal planning according to the principle suggested by our model for exploiting an ensemble of admissible heuristics.

Dealing with Model Assumptions

First, the simplified model assumes that the search space is a tree with a single goal state and uniform cost actions, and that the ensemble heuristics are consistent. Although the first assumption does not hold in most planning problems, and the second assumption is not satisfied by some state of the art heuristics, they do not prevent us from using the decision rule suggested by the model. Considering the exponential growth of the search space "from s" as a function of the heuristic's error, there is some empirical evidence to support the conclusion from the simplified model. For instance, Helmert and Röger (2008) prove for many planning domains that heuristics with a small constant additive error lead A^* to expand an exponential number of states, and the authors also illustrate this phenomenon experimentally. From the perspective of our goals here, from the same empirical results of Helmert and Röger (2008) it is also evident that the number of expanded states increases exponentially as the (still very small and additive) error increases. The latter suggests that the connection between the number of nodes to be expanded in our model and in practice can actually be quite right.

The next step is to somehow estimate the "depth to go" l. For that, we need to make another assumption about the rate at which f_1 grows in the sub-tree rooted at s. Although there are many possibilities here, we will look at two likely options. The first option is that the heuristic estimate h_1 remains constant in the subtree rooted at s (i.e. the additive error increases by 1 for each level below s). In this case, f_1 increases by 1 for each level expanded (because h_1 remains the same, and g increases by 1), and it will take expanding $\Delta_h(s) = h_2(s) - h_1(s)$ levels to reach the $f_1 = c^*$ line. The second option we examine is when the absolute error of h_1 remains constant (i.e. h_1 increases by 1 for each level expanded, so f_1 increases by 2). In this case, we will need to expand $\Delta_h(s)/2$ levels. This can be generalized to the case where the estimate h_1 increases by any constant additive factor c, which results in $\Delta_h(s)/(c+1)$ levels being expanded. In either case, l is linear in $\Delta_h(s)$, and our decision would be to compute h_2 if $\Delta_h(s) > \alpha \log_b(t_2/t_1)$, where α is a hyper-parameter for our algorithm.

Next, the model assumes that the branching factor is constant, and that the times to compute the heuristics are the same across all states. We deal with these two assumptions by estimating and relying upon the average branching factor and average heuristic computation times. These estimates are made on the basis of a random sample of states; the precise estimation procedure is described later on. Third, the model also assumes we know if a state is surely expanded or not. Since we do not, in fact, know this, we treat every state as if it was on the decision border, and thus apply the decision rule at every state.

Learning the Selection Rule Online

Without loss of generality, assume that $t_2 > t_1$. According to the model, the correct decision rule is to use h_2 when $\Delta_h(s) = h_2(s) - h_1(s) > \alpha \log_b(t_2/t_1)$ (since we do not know c^*, we apply this decision rule at every state). In order to do this, we learn a binary classifier that predicts whether, for a given state s, $\Delta_h(s) > \alpha \log_b(t_2/t_1)$. The learning procedure we devise is an *online active learning* procedure. In this procedure, the classifier is presented with a series of examples (states). For every example, the classifier can choose whether to classify the current example using the model learnt so far, or to ask an oracle for the classification of the current example, and then learn from it.

To build a classifier, we first need to collect training examples, which should be representative of the entire search space. One option for collecting the training examples is to use the first k states of the search (where k is the desired number of training examples). However, this method has a bias towards states that are closer to the initial state, and therefore is not likely to represent well the structure of the states in the search space. Hence, we instead collect training examples by sending "probes" from the initial state. Each such "probe" simulates a stochastic hill-climbing search, until a certain depth-limit is reached. All the states generated by such a probe are used as training examples. We send several such probes, until we have collected enough training examples. A more complex sampling method was proposed in (Haslum et al. 2007), but our approach is simpler, and works well in practice.

There are several parameters for this probing procedure. The maximum depth of each probe was set to twice the heuristic estimate of the initial state, that is $2 \max_h(s_0)$. Choosing a successor state to continue the probe from was done according to the inverse of the heuristic value (that is, the probability of choosing a successor s is proportional to $1/\max_h(s)$). The "inverse heuristic" selection biases the sample towards states with a lower heuristic value, which have a higher chance of being expanded by the search.

After the training examples T are collected, they are first used to estimate b, t_1 and t_2 by averaging the respective quantities over T. Once b, t_1 and t_2 are estimated, we can compute the threshold $\alpha \log_b(t_2/t_1)$ for our decision rule. We generate a label for each training example by calculating $\Delta_h(s) = h_2(s) - h_1(s)$, and comparing it to the decision

```
**evaluate**($s$)
1. $\langle class, confidence\rangle \leftarrow$ CLASSIFY($s$, $model$)
2. **if** ($confidence > \rho$)
    (a) **return** $h_{class}(s)$
3. **else**
    (a) $\Delta_h \leftarrow h_2(s) - h_1(s)$
    (b) $label \leftarrow (\Delta_h > \alpha \log_b(t_2/t_1))$
    (c) Update $model$ with $\langle s, label\rangle$
    (d) **return** $\max(h_1(s), h_2(s))$
```

Figure 2: The state evaluation algorithm.

threshold. If $\Delta_h(s) > \alpha \log_b(t_2/t_1)$, we label s with h_2, and otherwise with h_1. If $t_1 > t_2$ we simply switch between the heuristics; our decision is always *whether to evaluate the more expensive heuristic or not* (i.e. the default is to evaluate the cheaper heuristic, unless the classifier says otherwise).

Another requirement for building a classifier is deciding what features characterize states with respect to the decision rule. We decided to start our investigation with the simplest and most accessible features possible, notably the actual state variables of the planning problem, and postpone feasibility analysis for automatic generation of more complex features for later.

Once we have a classifier (constructed from the initial training set), we start the search from the initial state. For every state we evaluate, we use the classifier to decide which heuristic to compute. If the confidence for one of the possible answers is greater than some confidence threshold ρ, then we follow that decision (i.e. compute the heuristic that was chosen, and use its value). Otherwise, we do not have sufficient information about that state to predict the best heuristic to use, and therefore want to learn from it. This is done by computing the values of both heuristics, generating a label, and learning from this new example[1]. This is described in pseudo-code in Figure 2.

Although many classifiers can be used here, for several reasons we decided to use the Naive Bayes classifier. First of all, both training and classification with Naive Bayes are very fast, which is extremely desirable in the time-bounded setting we are interested in. Another advantage is that it is an incremental classifier, which allows us to learn from a new example very quickly. Finally, when classifying an example, Naive Bayes provides us with a probability distribution over possible classifications, and these probabilities have a natural semantics in terms of the classification confidence.

The Naive Bayes classifier assumes that the features are independent. Although this is not a fully realistic assumption for planning problems, using a SAS$^+$ formulation of the problem instead of STRIPS helps, as instead of many binary variables which are highly dependent upon each other, we have a small number of variables which are less dependent upon each other. The PDDL to SAS$^+$ translator (Helmert 2009) which is used in our implementation, does not generate the "best" possible SAS$^+$ formulation, but gets pretty close to it, by detecting one kind of dependence between propositions in the PDDL representation (mutual exclusion), and creating a single SAS$^+$ state variable that represents all of them. This is most likely part of the reason Naive Bayes works well here.

As a final note, extending selective max to use more than two heuristics is rather straightforward—simply compare the heuristics in a pair-wise manner, and choose the best heuristic by a vote, which can either be a regular vote (i.e. 1 for the winner, 0 for the loser), or weighted according to the classifier's confidence. Although this requires a quadratic number of classifiers, training and classification time (at least with Naive Bayes) appear to be much lower than the heuristic computation time, and thus the overall learning and classification overhead is likely to remain relatively low.

Extending the proposed approach for use with non-uniform action costs remains as future work. Altough the decision rule proposed here can be used "as is" with non-uniform action costs, it is likely that it should be adjusted to account for the different action costs.

Experimental Evaluation

To empirically evaluate the performance of selective max we have implemented it on top of the A^* implementation of the Fast Downward planner (Helmert 2006), and conducted an empirical study on a wide range of planning domains from the International Planning Competitions 1998-2006. The search for each problem instance was limited by a time limit of 30 minutes and memory limit of 1.5 GB; all the experiments were on 3GHz Intel E8400 CPU machines. The reported times do not include the PDDL to SAS$^+$ translation as it is common to all planners, and is tangential to the issues considered in our study. The reported times do include learning and classification time for selective max. In the experiments we set the size of the training set to 100, the confidence threshold ρ to 0.6, and α to 1.

Our evaluation of selective max was based on two state-of-the-art admissible heuristics h_{LA} (Karpas and Domshlak 2009) and $h_{\text{LM-CUT}}$ (Helmert and Domshlak 2009). Both of these heuristics perform most of their computation online, in contrast to abstraction heuristics that are based on expensive offline preprocessing, and then very fast per-state computation (Helmert, Haslum, and Hoffmann 2007; Katz and Domshlak 2009). As it is later shown in Table 1, neither of our two base heuristics was better than the other across all planning domains, although $h_{\text{LM-CUT}}$ overall solves more problems. On the other hand, the empirical time complexity of computing h_{LA} is typically much lower than that of computing $h_{\text{LM-CUT}}$.

We compare our selective max approach (sel_h) to each of the two base heuristics individually, and their standard, max-based aggregation ($\max_h$). In addition, to avoid erroneous conclusions about the impact of the specific decision criterion induced by our model on the effectiveness of selective max, we also compare to a trivial version of selective max that chooses between the two base heuristics simply at random (rnd_h).

[1]We do not change the estimate for b, t_1 and t_2, so the decision threshold remains fixed.

The top portion of Table 1 shows the number of solved problem instances in each planning domain by each method in question. Selective max solved more problems overall than (A^* with) both each of the individual base heuristics, as well as their max aggregation. Also, note that selective max solved at least as many problems as $\max_h$ and rnd_h in all domains. Comparing to the individual base heuristics on a per-domain basis, selective max solved all of the problems solved by h_{LA} (and more), and did not solve only 4 problems that were solved by $h_{\text{LM-CUT}}$ (2 in AIRPORT and 1 each in PSR-SMALL and TRUCKS). Finally, note that the results with rnd_h clearly indicate that the impact of the concrete decision rule suggested by our model on the performance of selective max is spectacular.

The middle and bottom portions of Table 1 provide a more detailed view of the results, depicting the average and median number of expanded states, as well as the average solution times, per domain. The expanded states numbers for different search strategies are normalized by the respective numbers for $\max_h$. All data is averaged only for problems that were solved by all methods. The two most interesting observations from Table 1 are probably as follows.

1. Supporting our original motivation, sel_h is on average, *substantially faster* than any of the other planners, including not only the max-based aggregation of the base heuristics, but also *both* our individual base heuristics.
2. Although selective max by definition expands at least as many states as the regular $\max_h$, in our experiments sel_h typically expanded only slightly more nodes than $\max_h$. Together with the poor performance of the random-choice rnd_h, this indicates the quality of the learned prediction.

Our first observation above on the relative speed of solving problems with selective max gets additional support if the time complexity results for different methods are considered in more detail. For each examined method, Figure 3 plots the total number of solved instances as a function of timeout. The plot is self-explanatory, and it clearly indicates that selective max has much better anytime behavior than either of the heuristics individually, or their maximum.

Finally, we compared to two other version of selective max: one version which chooses the successor state in the "probes" to generate the initial training set with uniform probability, and another which always uses 0 as the threshold in the decision rule (instead of $\alpha \log_b(t_2/t_1)$). The results of this comparison are not shown here for the sake of brevity, but they indicate that the version of selective max described in this paper is slightly better than both other versions.

Domain	h_{LA}	$h_{\text{LM-CUT}}$	$\max_h$	rnd_h	sel_h
Number of Solved Instances					
airport	25	38	36	29	36
blocks	20	28	28	28	28
depots	7	7	7	7	7
driverlog	14	14	14	14	14
freecell	28	15	22	15	28
grid	2	2	2	2	2
gripper	6	6	6	6	6
logistics-2000	19	20	20	20	20
logistics-98	5	6	6	5	6
miconic	140	140	140	140	140
mprime	21	25	25	19	25
mystery	13	17	17	14	17
openstacks	7	7	7	7	7
pathways	4	5	5	4	5
psr-small	48	49	48	48	48
pw-notankage	16	17	17	17	17
pw-tankange	9	11	11	10	11
rovers	6	7	7	6	7
satellite	7	8	9	7	9
tpp	6	6	6	6	6
trucks	7	10	9	7	9
zenotravel	9	12	12	10	12
Total	419	450	454	421	460
Average (Median) Expanded States					
airport (25)	20.41 (1.48)	1 (1)	1 (1)	1.09 (1.03)	1.16 (1)
blocks (20)	10.87 (4.1)	1 (1)	1 (1)	1.25 (1.24)	1.15 (1.01)
depots (7)	17.08 (16.7)	1 (1)	1 (1)	2.59 (2.35)	1.9 (1.02)
driverlog (14)	14.95 (9.46)	1.08 (1)	1 (1)	2.2 (2.2)	2.27 (1.44)
freecell (15)	1.24 (1.11)	188.57 (22.5)	1 (1)	17.47 (2.29)	1.88 (1.37)
grid (2)	3.38 (3.38)	1.02 (1.02)	1 (1)	1.47 (1.47)	3.32 (3.32)
gripper (6)	1 (1)	1.05 (1.01)	1 (1)	1.01 (1)	1 (1)
logistics-2000 (19)	1 (1)	1 (1)	1 (1)	1 (1)	1 (1)
logistics-98 (5)	8.43 (4.78)	1 (1)	1 (1)	1.66 (1.68)	4.67 (1.8)
miconic (140)	1 (1)	1 (1)	1 (1)	1 (1)	1 (1)
mprime (19)	415.47 (6.4)	3.44 (1)	1 (1)	27.06 (2.31)	6.34 (1.13)
mystery (12)	144.33 (1.06)	1.3 (1)	1 (1)	12.08 (1.29)	3.85 (1)
openstacks (7)	1.07 (1.1)	2.32 (2.29)	1 (1)	1.18 (1.16)	1.1 (1.12)
pathways (4)	251.87 (187.08)	1 (1)	1 (1)	15.19 (10.16)	1 (1)
psr-small (48)	1.36 (1.16)	1 (1)	1 (1)	1.07 (1.03)	1.22 (1.01)
pw-notankage (16)	5.95 (3.25)	1.49 (1)	1 (1)	1.71 (1.53)	1.48 (1.11)
pw-tankange (9)	2.16 (2.03)	1.47 (1.19)	1 (1)	1.31 (1.38)	1.11 (1.02)
rovers (6)	86.2 (5.53)	1 (1)	1 (1)	5.37 (1.06)	1.42 (1.04)
satellite (7)	78.71 (30.88)	1.01 (1)	1 (1)	15.16 (2.68)	2.11 (1.05)
tpp (6)	55.39 (1)	1 (1)	1 (1)	3.42 (1)	1.35 (1)
trucks (7)	77.4 (53.6)	1.01 (1)	1 (1)	7.31 (5.11)	1.03 (1)
zenotravel (9)	24.04 (4.82)	1 (1)	1 (1)	3.61 (2.16)	2.06 (1.17)
Overall	35.41 (1.02)	8.16 (1)	1 (1)	3.97 (1)	1.61 (1)
Average (Domain)	55.61 (15.54)	9.76 (2.05)	1 (1)	5.69 (2.1)	1.97 (1.21)
Total Solution Time in Seconds					
airport (25)	125.96	35.36	73.80	54.78	68.44
blocks (20)	66.01	3.71	6.39	6.44	5.59
depots (7)	196.91	65.99	103.26	155.14	94.36
driverlog (14)	66.67	110.87	86.04	120.84	81.31
freecell (15)	6.04	249.28	23.93	44.22	9.25
grid (2)	12.05	33.78	44.27	38.3	40.26
gripper (6)	71.6	106.48	264.79	161.98	77.07
logistics-2000 (19)	73.32	152.27	255.36	153.89	79.17
logistics-98 (5)	18.84	24.11	29.55	28.69	24.43
miconic (140)	2.03	8.04	10.08	5.67	7.65
mprime (19)	17.52	17.9	15.68	111.48	8
mystery (12)	7.55	1.61	2.03	57.93	2.49
openstacks (7)	15.93	72.3	75.83	52.69	17.11
pathways (4)	5.38	0.08	0.14	1.15	0.18
psr-small (48)	3.55	4.05	7.92	5.73	4.87
pw-notankage (16)	48.8	71.34	71.49	73.92	59
pw-tankange (9)	211.43	173.61	189.89	172.99	130.98
rovers (6)	122.7	5.23	8.79	45.72	7.97
satellite (7)	46.22	3.47	4.51	21.95	3.58
tpp (6)	108.54	14.36	5.9	56.32	5.69
trucks (7)	238.85	11.69	16.48	39.64	15.56
zenotravel (9)	9.84	0.91	1.33	8.27	1.28
Average (Problem)	39.65	38.59	41.39	42.6	24.53
Average (Domain)	67.08	53.02	58.97	64.44	33.83

Table 1: Summary of Results.
The top portion of the table shows the number of solved problem instances for each domain and planner. The middle portion shows the average and median of the number of expanded states (normalized by number of expanded states for $\max_h$) for each domain and planner. The bottom portion shows the average solving time in each domain for each planner. Averages and median are over problems solved by all planners (the number of which is shown in parentheses next to each domain).

Discussion and Future Work

Learning for planning has been a very active field starting in the early days of planning (Fikes, Hart, and Nilsson 1972), and is recently receiving growing attention in the community. So far, however, relatively little work has dealt with learning for heuristic search planning, one of the most prominent approaches to planning these days. Most works in this direction have been devoted to learning macro-actions (see, e.g., Botea et al. 2005, and Coles and Smith 2007).

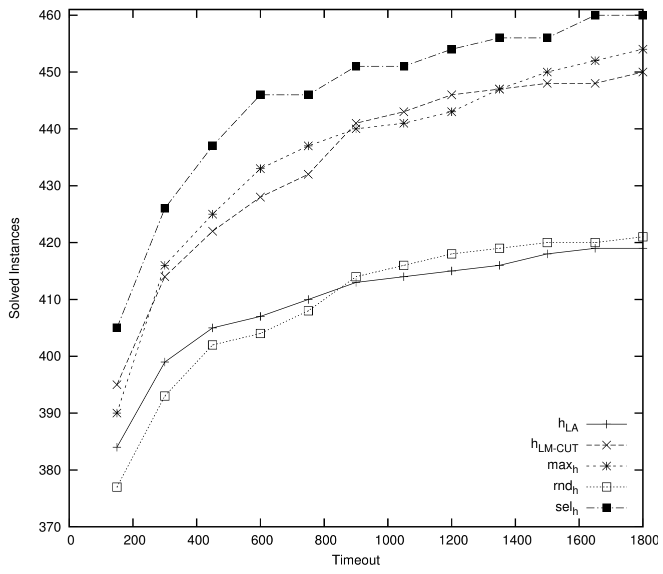

Figure 3: Total number of solved instances under different timeouts; the x-axis and y-axis capture the timeout in seconds and the number of problems solved, respectively.

Among the other works, the one most closely related to ours is probably the work by Yoon, Fern and Givan (2008) that suggest learning an (inadmissible) heuristic function based upon features extracted from relaxed plans. In contrast, our focus is on optimal planning. Overall, we are not aware of any previous work that deals with learning for optimal heuristic search.

The experimental evaluation demonstrates that selective max is a more effective method of combining arbitrary admissible heuristics than max. Another advantage for the selective max approach is that it can potentially combine two heuristics where one dominates another. For example, the h_{LA} heuristic can be used with two cost-partitioning schemes: uniform and optimal. The optimal cost-partitioning scheme dominates the uniform one, but takes much longer to compute. Selective max could be used to learn when it is worth spending the extra time to compute the optimal cost-partitioning, and when using the uniform cost-partitioning is better. Using max with these two cost-partioning schemes would simply waste the time spent on computing the uniform cost-partitioning.

These are still preliminary results. Further research is needed in order to study the effects of many of the parameters of selective max, such as training set size and confidence threshold, as well as use of different types of classifiers. Evaluating the performance of selective max with non-uniform cost actions, with different heuristics, as well as with more than two heuristics is another important direction for future work. However, we can conclude that selective max seems to be a good method of combining admissible heuristics for optimal planning.

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS$^+$ planning. *Comp. Intell.* 11(4):625–655.

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

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI planning with automatically learned macro-operators. *JAIR* 24:581–621.

Burke, E.; Kendall, G.; Newall, J.; Hart, E.; Ross, P.; and Schulenburg, S. 2003. Hyper-heuristics: an emerging direction in modern search technology. In Glover, F., and Kochenberger, G., eds., *Handbook of metaheuristics*. Kluwer Academic Publishers. chapter 16, 457–474.

Coles, A. I., and Smith, A. J. 2007. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research* 28:119–156.

Coles, A. I.; Fox, M.; Long, D.; and Smith, A. J. 2008. Additive-disjunctive heuristics for optimal planning. In *ICAPS*, 44–51.

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

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Learning and executing generalized robot plans. *AIJ* 3:251–288.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI*, 1007–1012.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *AAAI*, 1163–1168.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*. In press.

Helmert, M., and Röger, G. 2008. How good is almost perfect? In Fox, D., and Gomes, C. P., eds., *AAAI*, 944–949. AAAI Press.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, 176–183.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *AIJ*. in press.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *IJCAI*.

Katz, M., and Domshlak, C. 2008. Optimal additive composition of abstraction-based admissible heuristics. In *ICAPS*, 174–181.

Katz, M., and Domshlak, C. 2009. Structural-pattern databases. In *ICAPS*. In press.

Pearl, J. 1984. *Heuristics — Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

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