Using Learned Policies in Heuristic-Search Planning

SungWook Yoon
Computer Science & Engineering
Arizona State University
Tempe, AZ 85281
Sungwook.Yoon@asu.edu

Alan Fern
Computer Science Department
Oregon State University
Corvallis, OR 97331
afern@cs.orst.edu

Robert Givan
Electrical & Computer Engineering
Purdue University
West Lafayette, IN 47907
givan@purdue.edu

Abstract

Many current state-of-the-art planners rely on forward heuristic search. The success of such search typically depends on heuristic distance-to-the-goal estimates derived from the plangraph. Such estimates are effective in guiding search for many domains, but there remain many other domains where current heuristics are inadequate to guide forward search effectively. In some of these domains, it is possible to learn reactive policies from example plans that solve many problems. However, due to the inductive nature of these learning techniques, the policies are often faulty, and fail to achieve high success rates. In this work, we consider how to effectively integrate imperfect learned policies with imperfect heuristics in order to improve over each alone. We propose a simple approach that uses the policy to augment the states expanded during each search step. In particular, during each search node expansion, we add not only its neighbors, but all the nodes along the trajectory followed by the policy from the node until some horizon. Empirical results show that our proposed approach benefits both of the leveraged automated techniques, learning and heuristic search, outperforming the state-of-the-art in most benchmark planning domains.

Introduction

Heuristic search has been the most successful and dominant approach for suboptimal AI Planning (Hoffmann & Nebel 2001; Alfonso Gerevini & Serina 2003; Vidal 2004). The success of this approach is largely due to the development of intelligent automated heuristic calculation techniques based on relaxed plans (RPs), plans that ignore the action effect delete lists. RP-based heuristics provide an effective gradient for search in many AI planning domains. However, there are some domains where this gradient provides inadequate guidance, and heuristic search planners fail to scale up in such domains.

For some of those domains, e.g. Blocksworld, there are automated techniques (Martin & Geffner 2000; Fern, Yoon, & Givan 2004) that can find good policies through machine learning, enabling planners to scale up well. However, induced policies lack deductive guarantees and are in practice prone to be faulty—even more so when resource constraints limit the size of the training data, as occurs in planning competitions. Nevertheless such policies still capture useful, though imperfect, constraints on good courses of action. The main goal of this paper is to develop and evaluate an approach for combining such imperfect policies and heuristics in order to improve over the performance of either alone.

There are techniques that can improve on imperfect policies. Policy rollout (Bertsekas & Tsitsiklis 1996) and limited discrepancy search (LDS) (Harvey & Ginsberg 1995) are representative of such techniques. Policy rollout uses online simulation to determine for each encountered state, as it is encountered, which action performs best if we take it and then follow the learned base policy to some horizon. Policy rollout performs very poorly if the base policy is too flawed to find any reward, as all actions look equally attractive. This occurs frequently in goal-based domains, where policy rollout cannot improve on a zero-success-ratio policy at all.

Discrepancy search determines a variable search horizon by counting the number of discrepancies from the base policy along the searched path, so that paths that agree with the policy are searched more deeply. The search cost is exponential in the number of discrepancies tolerated, and as a result the search is prohibitively expensive unless the base policy makes mostly acceptable/effective choices.

Due to limitations on the quantity of training data and the lack of any guarantee of an appropriate hypothesis space, machine learning might produce very low quality policies. Such policies are often impossible to improve effectively with either policy rollout or LDS. Here, we suggest using such learned policies during node expansions in heuristic search. Specifically, we propose adding not only the neighbors of the node being expanded, but also all nodes that occur along the trajectory given by the learned policy from the current node. Until the policy makes a bad decision, the nodes added to the search are useful nodes.

In contrast to discrepancy search, this approach leverages the heuristic function heavily. But in contrast to ordinary heuristic search, this approach can ignore the heuristic for long trajectories suggested by the learned policy. This can help the planner critically in escaping severe local minima and large plateaus in the heuristic function. Policy evaluation is typically cheaper than heuristic function calculation and node expansion, so even where the heuristic is working

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

well, this approach can be faster.

We tested our proposed technique over all of the STRIPS/ADL domains of International Planning Competitions (IPC) 3 and 4, except for those domains where the automated heuristic calculation is so effective as to produce essentially the real distance. We used some competition problems for learning policies and then used the rest of the problems for testing our approach. Note that this approach is domain-independent. Empirical results show that our approach performed better than using policies alone and in most domains performed better than state-of-the-art planners.

Planning

A deterministic planning domain $\mathcal{D}$ defines a set of possible actions $\mathcal{A}$ and a set of states $\mathcal{S}$ in terms of a set of predicate symbols P, action types Y, and objects C. Each symbol σ in P or Y has a defined number of arguments it expects, denoted by $\text{arity}(\sigma)$. A state s is a set of state facts, where a state fact is an application of a predicate symbol p to $\text{arity}(p)$ objects from C. An action a is an application of an action type y to $\text{arity}(y)$ objects from C.

Each action $a \in \mathcal{A}$ is associated with three sets of state facts, $\text{Pre}(a)$, $\text{Add}(a)$, and $\text{Del}(a)$ representing the precondition, add, and delete effects respectively. As usual, an action a is applicable to a state s iff $\text{Pre}(a) \subseteq s$, and the application of an (applicable) action a to s, results in the new state $a(s) = (s \setminus \text{Del}(a)) \cup \text{Add}(a)$.

Given a planning domain, a *planning problem* is a tuple (s, A, g), where $A \subseteq \mathcal{A}$ is a set of actions, $s \in \mathcal{S}$ is the initial state, and g is a set of state facts representing the goal. A solution plan for a planning problem is a sequence of actions $(a_1, \ldots, a_l)$ from A, where the sequential application of the sequence starting in state s leads to a goal state s' where $g \subseteq s'$.

Learning Policies

A reactive policy π for a planning domain $\mathcal{D}$ is a function that maps each state s to an action a that is applicable to the state s. We desire a policy π for which iterative application of π to the initial state s of a problem p in $\mathcal{D}$ will find a goal, so that $g \subseteq \tau_\pi(\tau_\pi(\ \ldots \tau_\pi(s)))$, writing $\tau_\pi(s)$ for the next state under π, i.e., $(\pi(s))\,(s)$. Good reactive decision-list policies have been represented and learned for many AI planning domains (Khardon 1999; Martin & Geffner 2000; Yoon, Fern, & Givan 2002; 2005).

Taxonomic Decision List Policies

We represent a reactive policy as an ordered list of rules (Rivest 1987), each of which specifies constraints on the arguments of an action:

$$DL = \{\text{rule}_1, \ldots, \text{rule}_n\}$$
$$\text{rule}_i = \langle y : x_1 \in C_{i1}, \ldots, x_m \in C_{im} \rangle$$

Here, m is $\text{arity}(y)$ for action type y and each C_{ij} is a concept expression specifying a set of objects, as described below. A rule can fire on any tuple of objects $o_1, \ldots, o_m$ such that each o_i is in set specified by the corresponding C_{ij}—upon firing the rule suggests action $y(o_1, \ldots, o_m)$. The earliest rule in the list that can be fired in the current state will be fired. If more than one action is suggested by that rule, the tie is broken lexicographically based on an arbitrary unchanging ordering of the domain objects.

The syntax for the concepts is the following.

$$C \;=\; \textbf{any-object} \mid C \cap C \mid \neg C \mid$$
$$(p\ \ C_1 \ldots C_{i-1}\ \ *\ \ C_{i+1} \ldots C_{\text{arity}(p)})$$

Here, p is a predicate symbol, from P or as specified below. The predicate symbol p is applied to smaller class expressions, but with one argument omitted, indicated by a "*". Such applications denote the class of objects that make the predicate true when filled in for "*", given that the other arguments of p (if any) are provided to match the class expressions given. The semantics of the other constructs as classes of objects is as expected—please refer to (Yoon, Fern, & Givan 2002; 2006; McAllester & Givan 1993) for the detailed specification of the syntax and semantics. Note that this syntax automatically derives from predicate symbols of the target planning domain.

The evaluation of the semantics of each class C is relative to the relational database D constructed from the current state, the goal information and other information that is deduced from the current state and goal information. In our case this includes the relaxed plan from the current state to a goal state, a form of reasoning to construct features (Geffner 2004) and the reflexive transitive closure $p*$ for each binary predicate p. Predicate symbols other than those in P are allowed in order to represent goal information and features of the relaxed plan, as described in detail in (Yoon, Fern, & Givan 2002; 2006).

As an example decision list policy, consider a simple Blocksworld problem where the goal is clearing off a block A. The following decision-list policy is an optimal policy. $\{\langle \text{putdown}(x_1) : x_1 \in \text{holding}(*)\rangle, \langle \text{unstack}(x_1) : x_1 \in \text{clear}(*) \cap (\text{on}^*\ \ *\ \ A)\rangle\}$. The first rule says "putdown any block that is being held" and the second rule says "unstack any block that is above the block A and clear".

Learning Reactive Policies

For our evaluation, we learn reactive policies from solved small problems. For this purpose we deploy a learner similar to that presented in (Yoon, Fern, & Givan 2002). Given a set of training problems along with solutions we create a training set to learn a classifier that will be taken to be the learned policy. The classifier training set contains states labeled by positive and negative actions. The positive actions are those that are contained in the solution plan for each state, and all other actions in a state are taken to be negative. The learning algorithm conducts a beam search through the candidate class expressions, greedily seeking constraints on each argument of each action type to match the positive actions and not match the negative actions. Note that the training set is noisy in the sense that not all actions labeled as negative are actually bad. That is, there can be many optimal/good actions from a single state, though only one is included in

the training solutions. This noise is one reason that learned policies can be imperfect.

Using Non-Optimal Policies

Typically learned policies will not be perfect due to the bias of the policy language and variance of the learning procedure. In some domains, the imperfections are not catastrophic and the policies still obtain high success rates. In other domains, the flaws lead to extremely poor success rates. Nevertheless, in many states, even learned policies with poor success rates suggest good actions and we would like methods that can exploit this information effectively. First, we describe two existing techniques, rollout and discrepancy search that utilize search to improve upon an imperfect policy. Our experiments will show that these techniques do not work well in many planning domains, typically where the quality of the learned policy is low. These failures led us to propose a new approach to improving policies with search, which is discussed at the end of this section.

Policy Rollout

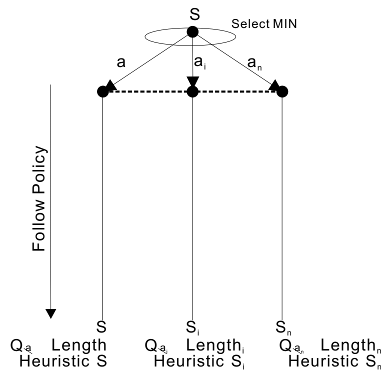

Figure 1: Policy Rollout. At a state s, for each action a, rollout the input policy π from state $a(s)$ until some horizon. For deterministic domains, the length of the rollout trajectory plus the heuristic at the end can be used as the Q-value for that action. Select the best action.

Policy rollout (Bertsekas & Tsitsiklis 1996) is a technique for improving the performance of a non-optimal "base" policy using simulation. Policy rollout sequentially selects the action that is the best according to the one-step lookahead policy evaluations of the base policy. Figure 1 shows the one-step lookahead action selection. For each action, this procedure simulates the action from the current state and then simulates the execution of the base policy from the resulting state for some horizon or until the goal is found. Each action is assigned a cost equal to the length of the trajectory following it plus the value of a heuristic applied at the final state (which is zero for goal states). For stochastic domains the rollout should be tried several times and the average of the rollout trials is used as a Q-value estimate. Policy rollout then executes the action that achieved the smallest cost from the current state and repeats the entire process from the resulting state. While policy rollout can improve a policy that is mostly optimal, it can perform poorly when the policy commits many errors, leading to inaccurate action costs. Multi-level policy rollout, e.g. as used in (Xiang Yan & Van Roy 2004), is one way to improve over the above procedure by recursively applying rollout, which takes time exponential in the number of recursive applications. This approach can work well when the policy errors are typically restricted to the initial steps of trajectories. However, for policies learned in planning domains the distribution of errors does not typically have this form; thus, even multi-step rollout is often ineffective at improving weak policies.

Discrepancy Search

A discrepancy in a search is a search step that is not selected by the input policy or heuristic function. Limited discrepancy search (LDS) (Harvey & Ginsberg 1995) bounds the search depth to some given number of discrepancies. Discrepancy search limited to n disrepancies will consider every search path that deviates from the policy at most n times.

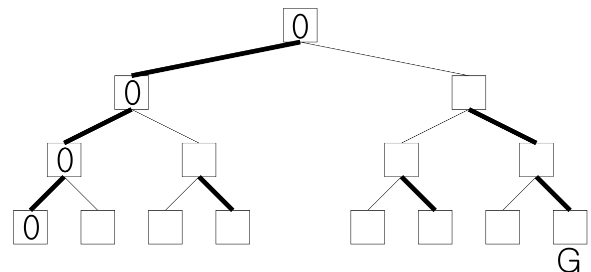

Figure 2: An Example of Discrepancy Search. Thin edges in the figure represent discrepancies. Nodes are labeled with their discrepancy depth.

Figure 2 shows an example of discrepancy search. The thick lines are choices favored by the heuristic function and the thin lines are discrepancies. Each node is shown as a rectangle labeled by the number of discrepancies needed to reach that node. Consider the depth of the goal in the search tree and the number of discrepancies needed to reach the goal node from the root node. In figure 2, the goal node is at depth three and discrepancy depth one from the root node. Plain DFS or BFS search will search more nodes than limit one disrepancy search. DFS and BFS need to visit 14 nodes before reaching the goal, while discrepancy search with limit one will find the goal after a 9-node search. Greedy heuristic search on this tree needs to backtrack many times before it reaches the goal node. The search space of LDS has size exponential in the discrepancy bound, and so, like policy rollout, LDS also is only effective with policies or heuristic functions with high quality. Heuristics can be incorporated by applying the heuristic at the leaves of the discrepancy search tree and then selecting the action that lead to the best heuristic value.

Incorporating Policies into Heuristic Search

In this work, we consider an alternative approach for incorporating imperfect policies into search. We attempt to take advantage of both approaches, automated heuristic calculation and automated policy learning. The main idea is to use policies during node expansions in best-first heuristic search, as described by the node expansion function shown in Figure 3. At each node expansion of the best-first search, we add to the search queue the successors of the current best node as usual, but also add the states encountered by following the policy from the current best node for some horizon. Our approach is similar to Marvin (Coles & Smith 2004), MacroFF (Botea *et al.* 2004), and YAHSP (Vidal 2004). However, unlike these approaches, we do not just add the final state encountered by the policy or macro, but rather add all states along the trajectory.

Embedding the policy into node expansion during heuristic search yields two primary advantages over pure heuristic search. First, when the input policy correlates well with the heuristic values, the embedding can reduce the search time. Typically, the direct policy calculation is much faster than greedy heuristic search because greedy heuristic search needs to compute the heuristic value of every neighbor, while direct policy execution considers only the current state in selecting actions. Second, like Blocksworld, where heuristic calculation frequently underestimates the true distance, our node expansion can lead the heuristic search out of local minima.

Node-Expansion-with-Policy (s, π, H)
// problem s, a policy π, a horizon H

$N \leftarrow$ Neighbors(s)
$s' \leftarrow s$
for $i = 0$ until $i == H$
    $s' \leftarrow \pi(s')$
    $N \leftarrow N \cup \{s'\}$
Return N

Figure 3: Node expansion for a heuristic search with a policy. Add nodes that occur along the trajectory of the input policy as well as the neighbors

Experiments

We evaluated the above approaches on the STRIPS domains from the recent international planning competitions IPC3 and IPC4, and on Blocksworld. We show the performance of each planning system in each row of the following figures. To test the effectiveness of our approach, we tested our base system, LEN, as well as our proposed technique, PH. The LEN system uses best-first search with the relaxed-plan–length (RPL) heuristic. We also compare against the planner FF, and, for each IPC4 domain, against the best planner in the competition on the particular domain. Finally, we compare against policy rollout (PR) and limited discrepancy search (D) techniques. For each system and domain, we report the number of solved problems, the average solution time for solved problems and the average length of the solved problems in separate columns of the results.

We used the first 15 problems as training data in each domain and the remaining problems for testing. We considered a problem to be unsolved if it was not solved within 30 minutes. For PR, D and PH systems, we used a horizon of 1000. All the experiments were run on Linux machines with a 2.8 Xeon Processor and 2GB of RAM.

Blocksworld

Figure 4 shows the results on Blocksworld from Track 1 of IPC 2. In this domain, LEN, PH, D solved all the problems but FF and PR failed to solve some problems. We used 100 cpu secs in learning the policy. The learned policy, $\pi_{\text{Blocksworld}}$ solved 13 problems. Thus, all of the policy improvement approaches, PR, D, and PH improved the input policy. The PH system solved all of the problems and improved the solution time over LEN, PR and D, showing the benefit of our approach for speeding-up planning, though PH produced slightly longer plans than FF and D.

Blocksworld (IPC2)			
Systems	Solved (20)↑	Time ↓	Length ↓
FF	16	0.64	38.1
LEN	20	11.74	116.3
PR	19	7.89	37.8
D	20	2.09	38
PH	20	0.06	44

Figure 4: Blocksworld Results

IPC3

Figure 5 summarizes the results on the IPC3 domains. The domains are significantly more difficult to learn policies for and we used 8h, 6h and 6h of cpu time for learning the policies in Depots, Driverlog, and FreeCell respectively. Although the learning times are substantial, this is a one-time cost: such learned policies can be resused for any problem instance from the domain involved. In these domains, the learned policies cannot solve any of the tested problems by themselves.

We see that all of PR, D and PH are able to improve on the input policies. PH solved more problems overall than FF, though FF has a slight advantage in solution length in the Depots domain. The Freecell domain has deadlock states and our policy performs poorly, probably by leading too often to deadlock states. Still the experiments show that our approach attained better solution times, demonstrating again that using policies in search can reduce the search time by quickly finding low-heuristic states.

PH outperforms LEN decisively in Depots and Driverlog, showing a clear benefit from using imperfect policies in heuristic search for these domains.

IPC4

Figure 6 shows the results for the IPC4 domains. We used 18, 35, 1, 3 and 4 hours of CPU time for learning poli-

IPC3				
Domain	Systems	Solved(5) ↑	Time ↓	Length ↓
Depots	FF	5	4.32	54.2
	LEN	1	0.28	29.0
	PR	2	70.86	32
	D	3	34.23	36.5
	PH	5	3.73	63.2
Driverlog	FF	1	1105	167.0
	LEN	1	1623	167.0
	PR	0	-	-
	D	0	-	-
	PH	4	75.91	177.3
Freecell	FF	5	574.84	108.4
	LEN	5	442.94	109.0
	PR	3	545.11	111.3
	D	3	323.23	85.6
	PH	4	217.49	108.3

Figure 5: IPC3 results

cies for Pipesworld, Pipesworld-Tankage, PSR, Philosopher and Optical Telegraph respectively. Here we evaluated one more system, B, which in each domain denotes the best performer from that domain from IPC4. The numbers for B were downloaded from the IPC web site and cannot be directly compared to our systems results. Still, the numbers give some idea of the performance comparison. Note that the learned policies alone can solve all the problems from the Philosopher and Optical Telegraph domains.

PH generally outperforms FF in solved problems, rendering solution length measures incomparable, except in PSR where the two approaches essentially tie.

For Pipesworld and Pipesworld-with-Tankage domains, the best performers of those domains in the competition performed much better than our system PH. However, PH still performed better than LEN, FF, PR and D, showing the benefits of our approach. Marvin, Macroff and YAHSP participated the competition. Each of these macro action based planners solved 60 or so problems. The best planners for each domain combined solved 105 problems. PH solved 133 problems, showing a clear advantage for our technique.

Overall, the results show that typically our system is able to effectively combine the learned imperfect policies with the relaxed-plan heuristic in order to improve over each alone. This is a novel result. Neither policy rollout nor discrepancy search are effective in this regard and we are not aware of any prior work that has successfully integrated imperfect policies into heuristic search planners.

Heuristic Value Trace in Search

In order to get a view of how incorporating policies can improve the heuristic search, we plotted a trace of the heuristic value for each node expansion during the search for problem 20 of Depots. Figure 7 shows a long plateau of heuristic values by the LEN system, while Figure 8 shows large jumps in heuristic value for the PH system, using far fewer node expansions, indicating that the policies are effectively helping to escape plateaus in the search space.

IPC4				
Domain	Systems	Solved (35) ↑	Time ↓	Length ↓
Pipesworld	FF	21	71.20	48.2
	LEN	15	71.38	48.2
	B	35	4.94	74.6
	PR	18	472.79	59.5
	D	14	215.51	49.3
	PH	29	129.21	76.7
Pipesworld Tankage	FF	4	532.20	62.0
	LEN	4	333.34	62.0
	B	28	221.96	165.6
	PR	7	366.11	49.7
	D	5	428.71	66.4
	PH	21	124.07	86.3
PSR (middle complied)	FF	17	638.42	104.2
	LEN	22	646.59	107.2
	B	18	124.93	106.8
	PR	6	691.04	72.0
	D	4	225.56	55.5
	PH	17	659.37	107.0
Philosophers	FF	0	-	-
	LEN	0	-	-
	B	14	0.20	258.5
	PR	33	3.89	363.0
	D	33	2.59	363.0
	PH	33	2.59	363.0
Optical Telegraph	FF	0	-	-
	LEN	0	-	-
	B	10	721.13	387.0
	PR	33	23.77	594.0
	D	33	19.67	594.0
	PH	33	19.67	594.0

Figure 6: IPC4 results

In Figures 10 and 9, we show the heuristic traces for LEN and PH on a Freecell problem where PH failed. Here PH is unable to escape from the plateau, making only small jumps in heuristic value over many node expansions. Interestingly, the trace of LEN system shows a plateau at a higher heuristic value than that for PH followed by a rapid decrease in heuristic value upon which the problem is solved. We are currently investigating the reasons for this behavior. At this point, we speculate that the learned policy manages to lead the planner away from a useful exit.

Conclusion

We embedded learned policies in heuristic search by following the policy during node expansion to generate a trajectory of new child nodes. Our empirical study indicates advantages to this technique, which we conjecture to have three sources. When the policy correlates well with the heuristic function, the embedding can speed up the search. Secondly, the embedding can help escape local minima during the search. Finally, the heuristic search can repair faulty action choices in the input policy. Our approach is easy to implement and effective, and to our knowledge represents the first demonstration of effective incorporation of imperfect policies into heuristic search planning.

References

Alfonso Gerevini, A. S., and Serina, I. 2003. Planning through stochastic local search and temporal action graphs in lpg. *Journal of Artificial Intelligence Research* 20:239–290.

Bertsekas, D. P., and Tsitsiklis, J. N. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

Botea, A.; Enzenberger, M.; Muller, M.; and Schaeffer, J. 2004. Macro-ff. In *4th International Planning Competition*.

Coles, A. I., and Smith, A. J. 2004. Marvin: Macro-actions from reduced versions of the instance. IPC4 Booklet, ICAPS 2004. Extended Abstract.

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

Geffner, H. 2004. Planning graphs and knowledge compilation. In *International Conference on Principles of Knowledge Representation and Reasoning*.

Harvey, W. D., and Ginsberg, M. L. 1995. Limited discrepancy search. In Mellish, C. S., ed., *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95); Vol. 1*, 607–615. Montréal, Québec, Canada: Morgan Kaufmann, 1995.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:263–302.

Khardon, R. 1999. Learning action strategies for planning domains. *AIJ* 113(1-2):125–148.

Martin, M., and Geffner, H. 2000. Learning generalized policies in planning domains using concept languages. In *KRR*.

McAllester, D., and Givan, R. 1993. Taxonomic syntax for first-order inference. *Journal of the ACM* 40:246–283.

Rivest, R. 1987. Learning decision lists. *MLJ* 2(3):229–246.

Vidal, V. 2004. A lookahead strategy for heuristic search planning. In *International Conference on Automated Planning and Scheduling*.

Xiang Yan, Persi Diaconis, P. R., and Van Roy, B. 2004. Solitaire: Man versus machine. In *NIPS*.

Yoon, S.; Fern, A.; and Givan, R. 2002. Inductive policy selection for first-order MDPs. In *UAI*.

Yoon, S.; Fern, A.; and Givan, R. 2005. Learning measures of progress for planning domains. In *AAAI*.

Yoon, S.; Fern, A.; and Givan, R. 2006. Learning heuristic functions from relaxed plans. In *ICAPS*.

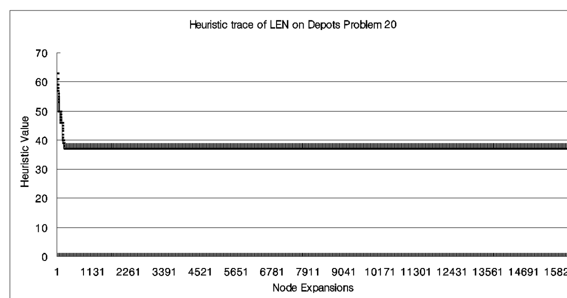

Figure 7: heuristic trace of LEN on Depots problem 20

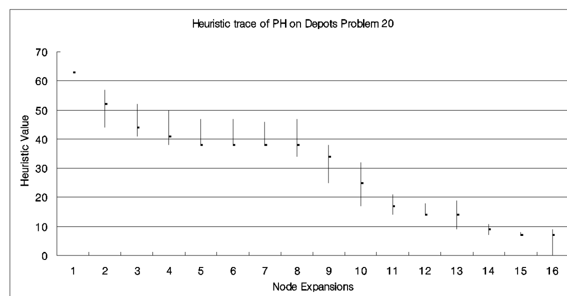

Figure 8: heuristic trace of PH on Depots problem 20

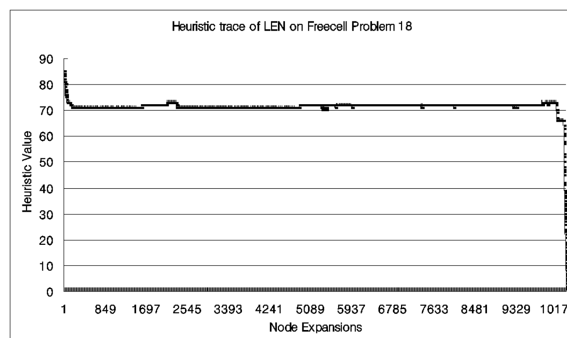

Figure 9: heuristic trace of LEN on FreeCell problem 18

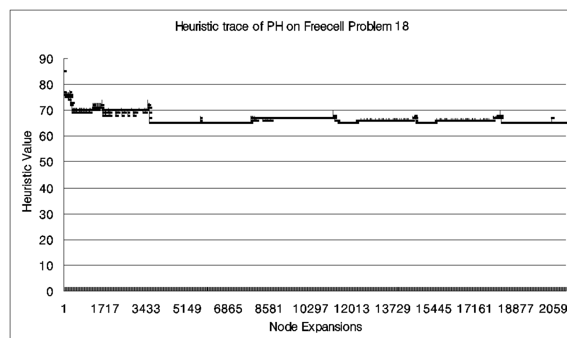

Figure 10: heuristic trace of PH on FreeCell problem 18