Deeply Preferred Operators: Lazy Search Meets Lookahead

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

Abstract

Heuristics in state-space search are primarily used to estimate the distance from states to the goal. In domain-independent heuristic-search planning, using extra information derived from the heuristic computation to mark some successors as preferred, and then biasing the search towards the preferred successors, resulted in significant improvements in planning performance. Preferred operators, however, help to discriminate only between the immediate successors of the evaluated state. We propose a simple and effective technique that takes advantage of more of the information provided by the heuristic computation. This technique, called *lazy lookahead*, consists of two components: A generalization of preferred operators to deeper descendants of the evaluated states, and a suitable generalization of deferred heuristic evaluation (aka lazy search) to such "deeply preferred" descendants. Our evaluation shows that employing lazy lookahead results in better performance than using standard preferred operators.

Introduction

Heuristic state-space search is one of the most prominent approaches to domain independent planning. For satisficing planning, the most common such approach is to use greedy best-first search, guided by heuristic functions. Heuristics are used primarily to estimate the distance from search states to the goal. The search algorithm can then use these distance estimates to choose a state which is likely to be closer to the goal, and thus hopefully to find a solution faster.

One of the most important advances in satisficing planning was the introduction of *helpful actions* in the FF planner (Hoffmann and Nebel 2001), where FF's relaxed plan was used not only for estimating the distance to the goal, but also to mark a few successors of the evaluated state as "helpful", in the sense of, "more likely to lead towards the goal". This concept was later generalized under the name of *preferred operators* (Helmert 2006), and was found especially helpful when used with *lazy search* (also called *deferred evaluation*, Richter and Helmert 2009). In lazy search, a state is evaluated not when it is generated, but when it is selected for expansion and removed from the open list. Lazy search aims at reducing the number of expensive heuristic evaluations: many states in the last layer of the search do not need to be evaluated, and preferred operators help focusing the evaluation on the more promising such states.

While which operators are considered preferred varies between the specific search components, the set of preferred operators of a given state always appears to be a subset of operators applicable at that state. At least in principle, this property appears to be unnecessarily limiting. Consider, for example, a state s for which a relaxed plan ρ^+ is generated. It is possible that ρ^+ (considered in terms of the original actions) is actually a valid plan from s to a goal state, and thus it provides us with the overall solution to the problem. Nevertheless, even using preferred operators, the search would need to evaluate at least every state along ρ^+ until the goal is found, despite the fact that we already have a solution at hand. Furthermore, it is possible that when the successor of s along ρ^+ is evaluated, a completely different relaxed plan will be generated for it, leading the search off the solution path that was already discovered.

Previous work has noted this, and suggested using FF's relaxed plan ρ^+ from state s for *lookahead* from s onwards (Vidal 2004). Specifically, an action sequence π applicable at state s is constructed from ρ^+, and then the state s' reached by π from s, called a lookahead state, is added to the open list, along with the regular successors of s. The weak point of such a lookahead is that often it is only some prefix of π that takes us closer to the goal, while the remaining part of π goes off in the wrong direction. In that case, adding only the end state achieved by π from s would probably not be the best thing to do.

Here we propose a technique that takes advantage of such *heuristically-suggested paths* π, that is, "preferred sequences" of real actions induced by the heuristic computation. This technique, called *lazy lookahead*, consists of two components. First, it extends the notion of preferred operators into what we call *deeply preferred operators*, which lead not only to the immediate successors of state s, but rather to all the states along the heuristically-suggested path π from s. Second, it extends the machinery of lazy search to properly support such deeply preferred descendants. Comparing to various approaches to satisficing heuristic-search planning that are based on standard notions of preferred operators and lazy search, our technique aims at reducing the number of heuristic evaluations even further, by obviating the need to compute heuristic estimates for states at various depths, not just in the last layer of the search. Our empirical evaluation shows that searching with lazy lookahead indeed results in

significant runtime and memory improvements, and even increases the number of planning tasks being solved.

Lazy Search Meets Lookahead

We consider planning tasks $\Pi = \langle P, A, cost, s_0, G\rangle$ formulated in STRIPS with action costs, where P are propositions, A are (standard syntax and semantics) actions, $s_0 \subseteq P$ is the inital state, and $G \subseteq P$ is the goal (Fikes and Nilsson 1971). The cost $cost(\pi)$ of an action sequence $\pi = \langle a_0, a_1, \dots, a_n\rangle$ is $\sum_{i=0}^{n} cost(a_i)$. An action sequence π is an s-path if it is applicable in state s; the state resulting from applying an s-path π in s is denoted by $s[\![\pi]\!]$. An s-path is an s-plan if $G \subseteq s_0[\![\pi]\!]$. In basic satisficing planning, the objective is to find an s_0-plan as efficiently as possible.

Lazy Lookahead

As previously mentioned, heuristics can provide us with more than just an estimate of the distance from state s to the goal. For now, let us assume that an (imperfect) oracle provided us with some s-path $\pi = \langle a_1, a_2, \dots a_n\rangle$, which is likely to lead towards the goal. We call such a path π a *heuristically-suggested path*, and later we discuss usage of a distinct class of heuristics as a respective oracle. In any case, given such π, the current mechanisms for supporting preferred operators allows us to "favor" the immediate successor $s[\![\langle a_1\rangle]\!]$ of s, biasing the search towards expanding it. However, these mechanisms cannot assist us with "favoring" the indirect successors $\{s[\![\langle a_1, \dots, a_i\rangle]\!]\}_{i=2}^{n}$ of s, despite their purported attractiveness suggested via π.

Aiming at taking advantage of as much of the information provided by the heuristic computation as possible, we propose a simple new mechanism for preferring "deeper" states along the simulated execution of the heuristically-suggested paths that we refer to as *lazy lookahead*. Given a heuristically-suggested s-path $\pi = \langle a_1, \dots, a_n\rangle$, lazy lookahead adds to the open list *all* the states $\{s[\![\langle a_1, \dots, a_i\rangle]\!]\}_{i=1}^{n}$, with the lazy heuristic estimate of $s[\![\langle a_1, \dots, a_i\rangle]\!]$ being set to the true heuristic value of s, adjusted by the cost of the actions $\{a_1, \dots, a_i\}$. That is, for $1 \leq i \leq n$, $h^{\text{lazy}}(s[\![\langle a_1, \dots, a_i\rangle]\!]) = h(s) - \sum_{j=1}^{i} cost(a_j)$.

Note that the adjusted heuristic estimate differs from what is normally done in lazy search, where the successors of state s are added to the open list with a heuristic estimate of $h(s)$ (Richter and Helmert 2009). The reason for this difference is that we want the "deeper" states to be expanded earlier, as they are more likely to be closer to a goal state. In contrast, lazy search only adds states on the same level, making this argument irrelevant. We also note that it is, of course, possible to simply perform an eager lookahead by evaluating each state along the simulated execution of π at s before adding these states to the open list. However, as heuristic computation is typically much more expensive than state generation, we believe this effort is very unlikely to pay off. For example, if the last state $s[\![\pi]\!]$ along that simulated execution is already a goal state, then search with lazy lookahead finishes immediately, with no need to evaluate all the intermediate states $\{s[\![\langle a_1, \dots, a_i\rangle]\!]\}_{i=1}^{n-1}$. Even if the last state $s[\![\pi]\!]$ is not a goal state, but it has a lower true heuristic estimate than $h(s)$, the search will continue from there, again, without the need to evaluate all the intermediate states.

Of course, there is no guarantee that a heuristically-suggested path leads anywhere near the goal. Consider the following scenario, where the search reaches a large heuristic plateau: State s is evaluated, and a heuristically-suggested path leading to state s' is generated. s' is evaluated next and found to have the same heuristic estimate as state s, and a heuristically-suggested path leading to state s'' is generated, and so on. This adds numerous states to the open list, as each heuristically-suggested path adds all the states along its simulation to the open list. In this scenario, the open list fills up with "junk" states, which might go much deeper than what search without lookahead would need to expand to escape the bad region. Having this potential negative effect of the lookahead in mind, we have also evaluated a variant of the lazy lookahead that we call *conditional lookahead*. The basic idea is simply to apply lookahead only for a selection of the expanded states. A simple and intuitive selection condition we have evaluated empirically aimed at preventing sequential lookahead that does not show any improvement. Specifically, we only look ahead from state s if it meets one of the following criteria:

1. s has been generated via the "regular" search, not via a heuristically-suggested path.
2. s was generated via a heuristically-suggested path from state s', and $h(s) < h(s')$.

These conditions prevent us from performing deeper and deeper lookahead, without any sign of improvement. Of course, more involved conditions, which might also be based upon information from the heuristic h, can be found even more beneficial in practice.

Generating Heuristically-Suggested Paths

Having described the way we suggest exploiting heuristically-suggested paths in the context of best-first search, we proceed with considering means for generating such heuristically-suggested paths. A natural option that we adopt here is to extract heuristically-suggested paths from the artifacts of computation of a certain class of heuristics—those that are based on solving a simplified version of the planning task at hand. Examples of such heuristics include FF's relaxed plan heuristic (Hoffmann and Nebel 2001), and abstraction heuristics such as PDBs (Culberson and Schaeffer 1998), merge and shrink (Helmert, Haslum, and Hoffmann 2008), and implicit abstraction heuristics (Katz and Domshlak 2010). What all of these heuristics have in common is that the heuristic value for state s is based upon a solution to a simpler, but still a planning, problem. While the state-of-the-art abstraction heuristics listed above avoid storing these solutions explicitly in order to reduce their memory overhead, the relaxed-plan FF heuristic generates such a solution every time it is evaluated. Note that this solution does not have to be a linear sequence of actions, but can rather comprise a partially ordered set of actions. The procedure we describe next is general, and can be used

with any heuristic that is based upon estimating the cost of such a partially ordered set of actions.

Given a partially ordered relaxed plan ρ^+ from state s, we attempt to find an applicable action sequence π which is compatible with that partially ordered plan. Our procedure attempts to apply actions from ρ^+, while keeping track of the current heuristically-suggested path, and of the state that is reached by it. An action from ρ^+ is eligible to be tried if all of its predecessors in ρ^+ have already been added to π. The partially ordered plan ρ^+ is traversed according to some linear order compatible with it, and checks whether an eligible action is applicable at the state $s[\![\pi]\!]$ reached by the current heuristically-suggested path. If an applicable action a was found, we update the currently reached state to $s[\![\pi \cdot \langle a \rangle]\!]$, and we update π to $\pi \cdot \langle a \rangle$. Once a complete pass over ρ^+ is accomplished, we go back to the beginning of ρ^+, and perform another pass, as some previously inapplicable actions might now become applicable. The procedure terminates after a complete pass over ρ^+, in which no action was added to the heuristically-suggested path π. Note that traversing ρ^+ according to different orders might lead to different heuristically-suggested path.

It is possible to employ some sophisticated tactics for choosing the linear order in which the partially ordered plan ρ^+ shall be traversed; for instance, some of such possible tactics in the context of FF relaxed plans are discussed in a related context by Vidal (2004). In our evaluation, however, our objective was to separate between the basics and optimizations, and thus we have implemented two simple choices: either try actions according to some arbitrary, implementation-dependent order, or choose the next eligible action at random. The only optimization we do apply is using the layer structure of FF's relaxed plan so that the linear order on actions is compatible with the order of the layers, and choosing at random from actions in the same layer. As the empirical evaluation will demonstrate, even these simple choices suffice to improve over the baseline lazy search with the FF heuristic.

Empirical Evaluation

In order to evaluate lazy lookahead empirically, we implemented it on top of the Fast Downward planning system (Helmert 2006), and conducted experiments on all domains from IPC 1998–2008, except MOVIE and ASSEMBLY. All of the experiments reported here were conducted on a single core of an Intel E8400 CPU, with a time limit of 30 minutes, and a memory limit of 1.5 GB.

As the current implementations of abstraction heuristics do not support obtaining a concrete solution for the abstraction from each state, we only evaluated the effect of lazy lookahead on search using FF's relaxed plan heuristic. In all of the experiments here, we used lazy greedy best-first search (with lazy lookahead, in our configurations) using boosted dual queues (Richter and Helmert 2009) and preferred operators (deeply preferred operators in our configurations). All of the states which were generated by the lazy lookahead procedure have been distinguished as preferred. We compare the baseline relaxed plan heuristic with using both (unconditional) lazy lookahead (denoted by *LL*)

domain	rnd-LL	LL	rnd-CLL	CLL	FF
airport (50)	**38**	37	35	**38**	37
depot (22)	**20**	19	18	19	19
elevators (30)	**30**	19	25	12	11
logistics98 (35)	**35**	33	**35**	**35**	33
mystery (30)	**16**	15	**16**	**16**	**16**
openstacks (30)	6	6	6	6	6
optical-telegraphs (48)	**3**	**3**	**3**	**3**	2
parcprinter (30)	17	**27**	18	26	21
pathways (30)	22	20	22	21	**29**
philosophers (48)	**48**	**48**	40	20	42
pw-notankage (50)	**44**	43	43	43	41
pw-tankage (50)	**43**	41	41	40	40
psr-large (50)	**16**	15	**16**	15	15
psr-middle (50)	42	43	**44**	43	42
schedule (150)	**150**	**150**	**150**	**150**	149
sokoban (30)	**29**	28	28	28	28
storage (30)	19	17	19	17	**20**
transport (30)	**30**	**30**	**30**	**30**	21
trucks (30)	17	16	17	16	**18**
woodworking (30)	29	**30**	**30**	**30**	27
TOTAL (shown domains)	**654**	640	636	608	617

Table 1: Number of tasks solved, per domain and overall. Domains where all approaches solved the same number of problems are not shown.

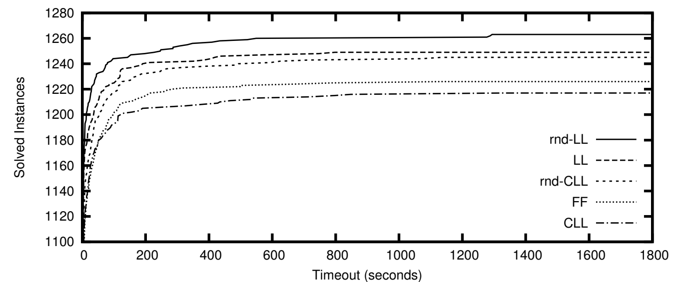

Figure 1: Anytime profile of different approaches. Each line shows the number of problems solved by each approach (y-axis), under different timeouts (x-axis).

and conditional lazy lookahead (denoted by *CLL*). For each such method, we have two variants of partial order plans linearization: random (denoted with the prefix *rnd-*) and arbitrary (no prefix).

Table 1 shows the number of problems solved using each approach, in each domain. We omitted here the domains in which all the approaches solved all the tasks in the domain. These results show that greedy best first search using the randomized variant of both conditional and unconditional lazy lookahead solves overall more problems than the baseline, and that adopting lazy lookahead was beneficial on more domains than domains in which it hurt the performance.

A more detailed examination of the results shows that randomization in the lookahead greatly helps in ELEVATORS, and greatly hurts in PARCPRINTER. One possible reason for this is that the arbitrary action ordering in PARCPRINTER is actually very good for the relaxed plan, while in ELEVATORS it is very bad. However, overall, randomization performs better than using the arbitrary order.

Another observation is that conditional lookahead does not seem to pay off, and does not seem to perform as well as (unconditional) lookahead. This is likely to do with the extra overhead associated with conditional lookahead, where we

domain	rnd-LL	LL	rnd-CLL	CLL	FF
airport	0.81	0.63	**0.82**	0.65	0.72
blocks	**0.71**	0.7	0.69	0.58	0.7
depot	0.59	0.55	**0.63**	0.55	0.36
driverlog	0.7	0.68	**0.72**	0.71	0.32
elevators	**0.93**	0.35	0.46	0.12	0.11
freecell	**0.76**	0.74	0.65	0.66	0.37
grid	**0.87**	0.59	0.62	0.62	0.44
gripper	0.76	0.88	0.93	**0.94**	0.22
logistics00	**0.82**	0.7	0.78	0.78	0.1
logistics98	**0.83**	0.51	0.8	0.58	0.06
miconic	0.83	**0.95**	0.77	0.91	0.17
mprime	**0.81**	0.72	0.58	0.62	0.29
mystery	0.66	0.71	**0.81**	0.73	0.48
openstacks	0.53	0.52	0.57	0.44	**0.76**
optical-telegraphs	0.51	**0.72**	0.17	0.04	0.07
parcprinter	0.46	**0.65**	0.5	0.59	0.49
pathways	**0.89**	0.82	0.8	0.8	0.12
pegsol	0.47	0.55	0.6	0.56	**0.71**
philosophers	0.82	**1**	0.02	0.03	0.04
pw-notankage	0.48	0.53	**0.58**	0.57	0.55
pw-tankage	**0.59**	0.56	0.48	0.45	0.39
psr-large	0.82	0.84	0.88	0.87	**0.91**
psr-middle	0.84	0.84	0.88	0.89	**0.91**
psr-small	0.66	0.67	0.73	0.73	**1**
rovers	0.84	0.81	**0.87**	0.85	0.07
satellite	0.76	0.87	0.82	**0.9**	0.12
scanalyzer	0.78	**0.85**	0.5	0.48	0.29
schedule	0.73	0.69	**0.74**	0.67	0.11
sokoban	0.72	0.62	0.72	0.72	**0.85**
storage	0.64	0.62	**0.82**	0.78	0.81
tpp	**0.93**	0.59	0.48	0.32	0.14
transport	**0.83**	0.75	0.36	0.27	0.16
trucks	0.46	0.38	**0.69**	0.46	0.52
woodworking	0.74	0.8	0.75	**0.83**	0.42
zenotravel	0.83	**0.87**	0.75	0.86	0.2
NORM. AVG	**0.73**	0.69	0.66	0.62	0.4

Table 2: Generated states: average metric scores.

domain	rnd-LL	LL	rnd-CLL	CLL	FF
airport	0.84	0.66	**0.85**	0.7	0.53
blocks	**0.75**	**0.75**	0.69	0.59	0.57
depot	0.61	0.56	**0.62**	0.54	0.32
driverlog	0.7	0.68	**0.72**	0.7	0.28
elevators	**0.93**	0.35	0.44	0.12	0.1
freecell	**0.75**	**0.75**	0.62	0.66	0.3
grid	**0.87**	0.59	0.56	0.58	0.3
gripper	0.76	0.88	**0.94**	0.92	0.11
logistics00	**0.79**	0.68	0.75	0.77	0.07
logistics98	**0.83**	0.51	0.8	0.59	0.06
miconic	0.83	**0.95**	0.77	0.91	0.15
mprime	**0.82**	0.72	0.61	0.65	0.32
mystery	0.7	0.73	**0.8**	0.73	0.47
openstacks	0.62	0.6	0.59	0.48	**0.68**
optical-telegraphs	0.51	**0.74**	0.14	0.03	0.04
parcprinter	0.41	**0.59**	0.48	0.56	0.38
pathways	**0.92**	0.83	0.76	0.79	0.15
pegsol	0.53	**0.63**	**0.63**	0.57	0.59
philosophers	**1**	**1**	0.02	0.02	0.02
pw-notankage	0.52	0.56	**0.59**	0.58	0.5
pw-tankage	**0.59**	0.58	0.47	0.46	0.37
psr-large	0.89	0.9	**0.93**	0.92	0.9
psr-middle	0.9	0.91	**0.93**	**0.93**	0.91
psr-small	0.98	0.98	**1**	0.99	0.97
rovers	0.84	0.85	0.85	**0.88**	0.06
satellite	0.77	0.87	0.82	**0.9**	0.11
scanalyzer	0.79	**0.85**	0.5	0.47	0.26
schedule	**0.74**	0.68	0.73	0.66	0.12
sokoban	**0.87**	0.79	0.79	0.8	0.7
storage	0.76	0.74	**0.86**	0.82	0.77
tpp	**0.91**	0.57	0.42	0.3	0.07
transport	**0.83**	0.75	0.34	0.25	0.13
trucks	0.42	0.35	**0.66**	0.46	0.58
woodworking	0.74	0.8	0.71	**0.82**	0.45
zenotravel	0.79	0.82	0.74	**0.85**	0.19
NORM. AVG	**0.76**	0.72	0.66	0.63	0.36

Table 3: Evaluated states: average metric scores.

need to keep track of how each state was reached (via lookahead or not), and of the heuristic value of the state where the respective lookahead started (if it was reached via lookahead).

However, the number of problems solved after 30 minutes does not tell the complete tale. Figure 1 shows the number of problems solved by each approach, under different timeouts. As the results show, deeply preferred operators help, no matter which timeout is used, with an even greater advantage over the baseline relaxed plan heuristic when the timeout is smaller.

Finally, we wish to explore the impact of using deeply preferred operators on the number of generated states, as well as on the number of heuristic evaluations performed. Tables 2 and 3 give the metric score of the number of generated and evaluated state, respectively, over the problems solved by all 5 configurations. The metric score for configuration c on some problem is v^*/v_c, where v_c is the value of configuration c (number of generated or evaluates stated), and v^* is the best value of any configuration on that problem. Thus the best value for each problem is assigned a metric score of 1, and generating twice as many states would lead to a score of 0.5. For each domain we report the average score, as well as the average across all domain averages. As these tables show, using lazy lookahead significantly reduces both the number of generated states, as well as the number of evaluated states, and this across all the four settings of lazy lookahead.

References

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *AIJ* 2:189–208.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2008. Explicit-state abstraction: A new method for generating heuristic functions. In *Proc. AAAI 2008*, 1547–1550.

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

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Katz, M., and Domshlak, C. 2010. Implicit abstraction heuristics. *JAIR* 39:51–126.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In *Proc. ICAPS 2009*, 273–280.

Vidal, V. 2004. A lookahead strategy for heuristic search planning. In *Proc. ICAPS 2004*, 150–159.