Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling

Incremental LM-Cut

Florian Pommerening and **Malte Helmert**
Universität Basel
Fachbereich Informatik
Bernoullistrasse 16
4056 Basel, Switzerland
firstname.lastname@unibas.ch

Abstract

In heuristic search and especially in optimal classical planning the computation of accurate heuristic values can take up the majority of runtime. In many cases, the heuristic computations for a search node and its successors are very similar, leading to significant duplication of effort. For example most landmarks of a node that are computed by the LM-cut algorithm are also landmarks for the node's successors. We propose to reuse these landmarks and incrementally compute new ones to speed up the LM-cut calculation. The speed advantage obtained by incremental computation is offset by higher memory usage. We investigate different search algorithms that reduce memory usage without sacrificing the faster computation, leading to a substantial increase in coverage for benchmark domains from the International Planning Competitions.

Introduction

The performance of heuristic search for optimal classical planning depends on accurate heuristic values to a large degree. Computing good heuristic values is hard and can take up a large amount of the search algorithm's runtime. However, some hard to compute heuristic functions only change slightly from state to state. In this case, it is possible to store the result of the heuristic computation and use it when generating successor nodes. Instead of computing their heuristic value from scratch, the stored information of the parent node is adapted to accommodate the changes in the successor state. A famous example for this is the *Manhattan distance* (Korf 1985). Consider a sliding-tile puzzle where the heuristic estimate is the sum of the Manhattan distances of all tiles to their goal position. When one tile is moved it is sufficient to update the heuristic value by the difference in the heuristic value of this tile. All other tiles do not move and their heuristic values stay the same. Burns et al. (2012) report that this feature doubled the performance of their sliding-tile solver.

We investigate the incremental computation of the LM-cut heuristic (Helmert and Domshlak 2009), an accurate estimate of the delete-relaxation heuristic h^+. LM-cut seems promising for incremental computation, as it is comparatively expensive to compute but yields high-quality heuristic estimates. We show that after applying an operator, most of the computational effort for heuristic calculation can be reused from the parent node's calculation. To this end, the resulting landmarks are stored after each heuristic computation. As in the sliding-tiles example, most of this stored information remains valid in a successor node and updating the heuristic value is cheaper than computing it from scratch.

With an initial version we achieve speedups of over an order of magnitude but at the cost of an often prohibitively high memory consumption. We thus explore different techniques to save memory while still being able to benefit from the performance of incremental computation.

We start by discussing related work and introducing relevant notation, followed by a discussion of the theoretical background. We then introduce three possible ways to use incremental computation in A* search and evaluate them. To reduce the amount of used memory we also tried using IDA* search instead of A* search, which we discuss in the last section.

Related Work

We previously reported on incrementally computing the LM-cut heuristic (Pommerening and Helmert 2012). Our work there is restricted to a delete-free setting, however, and uses a search space that is not complete for general STRIPS planning. The search space takes advantage of the fact that in the delete-free setting no operator has to be used more than once. In this setting a tree search over the subsets of operators is possible, so that no duplicates occur. This tree search can be performed with depth-first methods in a memory-efficient way. We now generalize this approach to arbitrary STRIPS planning tasks, where this is not valid. Instead we use A* and IDA* search with the classical search space that has to deal with duplicates in the search graph.

Liu, Koenig, and Furcy (2002) use incremental heuristic computation in an incremental value iteration (*IVI*) approach to calculate the additive heuristic h^{add} (Bonet and Geffner 2001). In their case the calculation of h^{add} uses a value iteration (*VI*) method that updates h^{add} values until they no longer change. Instead of resetting the values for each variable before a computation, IVI maintains the values from the last calculation in memory and starts from there. This is most beneficial for siblings in the search tree, since they often have similar values for most variables. The

algorithm is correct no matter which node's information is used for the incremental computation. For IVI it is thus sufficient to store the information obtained from one heuristic calculation, which in this case can be done in-place without memory overhead. For the incremental computation of $h^{\text{LM-cut}}$, on the other hand, it is crucial that the landmarks of the parent node are known to allow an incremental computation. Memory restrictions are thus a major limitation for incremental LM-cut, although they are no issue for IVI. Liu, Koenig, and Furcy (2002) also introduce a second incremental algorithm called PINCH (Prioritized INCremental Heuristic calculation) that combines IVI with a variable ordering within the state. It inherits the property of IVI that incremental computation can be done for any two states, so memory restrictions are no issue for PINCH either.

Zhang and Bacchus (2012) use LM-cut to seed a MAXSAT encoding of planning problems. While not incremental, the algorithm they propose shares some important aspects with ours. In a technique called *lazy heuristic evaluation* the LM-cut value of a search node is calculated and this node is added to the open list. When the node is popped from the open list and still has its LM-cut value as its stored f-value, a more expensive heuristic based on the landmarks found by LM-cut is computed. The article does not mention whether the computed landmarks are stored with each node or recomputed in a second LM-cut computation. In the former case the algorithm is bound to show problems with limited memory and might benefit from the techniques discussed in the following. In the latter case, the technique shares one key idea with one of our algorithms, where incremental computation is only done locally: computing a cheaper heuristic on node generation and a more expensive heuristic later in the search, when the node is expanded, saves search time. The reason for this is that not all nodes that are generated are also expanded, which is discussed in more detail later.

Notation

The planning tasks we try to solve are STRIPS tasks with action costs:

Definition 1 (Planning task). A *planning task* is a tuple $\Pi = \langle V, I, G, O \rangle$ with a finite set of propositional *variables* V, an *initial state* $I \subseteq V$, a set of *goals* $G \subseteq V$ and a finite set of *operators* O, where each $o \in O$ consists of a set of *preconditions* $\text{pre}(o) \subseteq V$, a set of *add effects* $\text{add}(o) \subseteq V$, a set of *delete effects* $\text{del}(o) \subseteq V$, and a *cost* $\text{cost}(o) \in \mathbb{R}_0^+$.

An operator o is *applicable* in state $s \subseteq V$ if $\text{pre}(o) \subseteq s$. Applying o in s results in the state $s[o] = (s \setminus \text{del}(o)) \cup \text{add}(o)$. An operator sequence $\pi = o_1 \dots o_n$ is applicable in s if there are states $s_0, \dots, s_n$ such that $s_0 = s$ and for all $1 \leq i \leq n$, o_i is applicable in s_{i-1} and $s_{i-1}[o_i] = s_i$. Applying π in s results in $s[\pi] = s_n$. If $G \subseteq s[\pi]$, the sequence π is called a *plan* for s. The cost of an operator sequence is $\text{cost}(o_1 \dots o_n) = \sum_{i=1}^{n} \text{cost}(o_i)$. *Optimal planning* is the PSPACE-complete problem of finding minimal-cost plans for I (Bylander 1994).

Heuristic functions h map states s to *heuristic values* $h(s) \in \mathbb{R}_0^+ \cup \{\infty\}$. The *perfect heuristic* h^* maps each state s to the cost of a cheapest plan for s. Heuristics h that never overestimate this cost, i.e., where $h(s) \leq h^*(s)$ for all s, are called *admissible*. Using an admissible heuristic, the A* (Hart, Nilsson, and Raphael 1968) and IDA* (Korf 1985) algorithms are guaranteed to find optimal solutions.

Theory

LM-cut Heuristic. Experiments by Helmert and Domshlak (2009) have shown that the LM-cut heuristic offers high quality admissible estimates of the delete-relaxation heuristic h^+ for many planning tasks. The delete-relaxation heuristic solves an easier version of the planning task where all delete effects are ignored. It is a good, but intractable (Betz and Helmert 2009; Bylander 1994) lower bound for the perfect heuristic.

The values of LM-cut are defined by *disjunctive action landmarks* (called *landmarks* in the following). A landmark is an operator set L such that every solution must contain at least one operator from L. Its cost (written as $\text{cost}(L)$) is the minimum over the cost of the contained operators, reflecting the fact that at least the cost for the cheapest operator has to be spent. The LM-cut computation is performed in *rounds* where each round proceeds as follows:

1. Compute the $h^{\max}$ values (Bonet and Geffner 2001) of all variables. If $h^{\max}(g) = 0$ for all goal variables g, stop and return the computed heuristic value. If $h^{\max}(g) = \infty$ for some goal variable g, stop and return ∞.
2. Use the $h^{\max}$ values to compute a landmark L with nonzero cost c and add c to the heuristic value (which starts as 0). For our purposes the details of how L is discovered do not matter.
3. Reduce the operator costs of all operators in L by c.
4. Discard L.

Before returning, all operator costs are reset to their original value. For a more detailed description of this process and the discovery of landmarks in step 2., we refer to the literature (Helmert and Domshlak 2009; Bonet and Helmert 2010).

Incremental Computation. The central idea presented here is to store the landmarks computed in step 2. of the LM-cut computation and use them during the heuristic computation of successor nodes.

Consider a search node σ associated with the state s (written as $s = \text{state}(\sigma)$ in the following). When we compute the heuristic value of s we store the discovered landmark set $\mathcal{L}_s$. When generating the successor node σ_o that results from applying operator o to s, most landmarks in $\mathcal{L}_s$ will also be landmarks for $s' = \text{state}(\sigma_o)$. The landmarks in $\mathcal{L}_s$ are operator sets where at least one operator has to be used when starting from s. Applying the operator o does not change this for landmarks that do not mention o.

We can thus start the LM-cut algorithm for s' with the landmark set $\{L \in \mathcal{L}_s \mid o \notin L\}$ already computed. If the LM-cut algorithm would have discovered these landmarks in its first rounds, it would also have adjusted the operator costs for each discovered landmark, so this has to be done as well.

The remaining operator costs at the end of each LM-cut calculation can either be stored alongside $\mathcal{L}_s$ or recreated from the stored landmarks. We chose to do the latter because it frees up memory that can be used to save additional landmark sets. In this case, the remaining operator costs are computed for each operator o as its original cost reduced by the sum of all stored costs of landmarks containing o. If the same set of landmarks is discovered in two different orders, the resulting cost can be different but remains an admissible estimate.

If the LM-cut algorithm would have discovered the same landmarks in the first rounds, the final calculated heuristic value would be the same with incremental computation. However, this is not necessarily the case. If the first rounds would have discovered different landmarks, the $h^{\text{LM-cut}}$ value could be higher or lower than the incrementally computed value. The estimate remains admissible, though, because the operator costs are decreased for each landmark. This introduces an additive cost partitioning with one partition per landmark that assigns the landmark's cost to all its operators and zero to all other operators. Experiments with expansion-limited searches have shown that incremental computation does not affect the overall predictive quality of LM-cut (Pommerening 2011).

For a full incremental computation, the landmarks calculated for all generated search nodes have to be saved. If this is not feasible because of memory constraints, stored landmarks can be discarded for any node at any time. The successors of such nodes where landmarks were discarded can be generated with the regular (non-incremental) LM-cut heuristic leading to a new set of stored landmarks. In the following section we discuss and evaluate different strategies when and for which nodes the stored landmarks should be discarded.

Practice

The introduction and discussion of strategies is interleaved with their empirical evaluation in the following sections because results for one strategy influence design choices for the next. We evaluated our algorithms on 1396 tasks in 44 domains from IPC 1998–2011. These are all domains from the respective optimal tracks where Fast Downward (Helmert 2006) generates a grounded task without conditional effects or axioms. All experiments were run on a cluster of Intel Xeon X5550 quad-core CPUs (2.67GHz) with one task per core. Memory was limited at 2 GB and time at 30 minutes per run.

We report the number of problems solved (*coverage*) and the chance to solve a randomly selected task from a randomly selected domain (*coverage score*). The coverage score gives the same weight to each domain and is thus better suited to compare performance than simple averages if domains have different sizes.

We also report *time scores* which represent the expected time to solve a random task logarithmically scaled between 1 second and the time limit. For details on this scoring system we refer to the literature (Richter and Helmert 2009). The data is right-censored, i.e., the time scores for runs that are stopped because of exhausted memory cannot be measured. Hence, the time score is a pure measure of computation speed independent of memory usage only when it is calculated for a timeout where no task stopped because of exhausted memory (55 seconds in our case).

We thus report three values: an uncensored time score computed for a time limit of 55 seconds and two estimates for the uncensored time score with a limit of 1800 seconds. The assumption for the first is that every problem exhausting memory would not have been solved within the time limit. For the second we assume that the task was solved at the time the program was stopped.[1]

Application to A* Search

We implemented the incremental computation of LM-cut in the Fast Downward planner. In this basic algorithm (called $h^{\text{iLM-cut}}$) all landmarks are stored and all computations except the first are incremental.

Note that $h^{\text{iLM-cut}}$, unlike the original non-incremental LM-cut heuristic, is consistent: if σ' is the successor node of node σ via operator o, then $h^{\text{iLM-cut}}(\sigma') \geq h^{\text{iLM-cut}}(\sigma) - \text{cost}(o)$. To see this, observe that when going from σ to σ', the heuristic value can only decrease by the sum of costs of landmarks containing the applied operator o (only landmarks containing o are discarded by the incremental computation). Because the landmarks found by the LM-cut procedure induce a cost partitioning, the costs of the landmarks containing o must sum to at most $\text{cost}(o)$. Note that we also introduce variants of $h^{\text{iLM-cut}}$ that do not store all landmarks and are thus not consistent.

Using A* with $h^{\text{iLM-cut}}$ instead of $h^{\text{LM-cut}}$ already improves the coverage and computation time (see Table 1). The improvement in computation speed comes at the cost of a larger memory consumption. With $h^{\text{LM-cut}}$ only 31 instances exhausted the memory limit, while the A* search with $h^{\text{iLM-cut}}$ exhausted its memory 526 times. The time score on the other hand increases by at least four percentage points. This increase in computation speed leads to an increase in coverage by 5. We will now look into how the memory requirements of $h^{\text{iLM-cut}}$ can be reduced without sacrificing too much of the increased time score.

Saving Memory by Forgetting Landmarks

As explained earlier, stored landmarks can be discarded at any time and can be recomputed on demand. This can be used to trade computation speed for memory. The decisions which landmarks are discarded and when this is done are referred to as a strategy. To reduce the memory consumption, we used reference-counting pointers instead of storing the landmarks explicitly for each node. When a successor node is generated only pointers need to be copied from its parent. When all landmark sets containing a specific landmark are discarded, the landmark is also discarded automatically.

[1] Note that the uncensored time score for a limit of 1800 seconds must fall between these two values, though not necessarily in the middle of that interval. Since large, complex tasks tend to be the ones with memory and time problems, the value is more likely closer to the first value.

Coverage per domain	$A^*/h^{\text{LM-cut}}$	$A^*/h^{\text{iLM-cut}}$	$A^*/h^{\text{iLM-cut}}_{\text{frontier}}$	$A^*/h^{\text{iLM-cut}}_{\text{local}}$	$A^*/h^{\text{iLM-cut}}_{50\,\text{MB}}$	$A^*/h^{\text{iLM-cut}}_{250\,\text{MB}}$	$A^*/h^{\text{iLM-cut}}_{500\,\text{MB}}$	$A^*/h^{\text{iLM-cut}}_{\text{dynamic}}$	$\text{IDA}^*/h^{\text{LM-cut}}$	$\text{IDA}^*/h^{\text{iLM-cut}}_{\text{frontier}}$	$\text{IDA}^*_{\text{L}}/h^{\text{iLM-cut}}_{\text{frontier}}$
airport (50)	28	**30**	**30**	29	**30**	**30**	**30**	***30***	**30**	**30**	**30**
barman (20)	**4**	**4**	**4**	**4**	**4**	**4**	**4**	***4***	0	**4**	**4**
blocks (35)	28	28	28	28	28	**29**	**29**	***29***	28	**29**	28
depot (22)	7	7	7	7	7	7	7	*7*	7	7	7
driverlog (20)	13	13	13	**14**	13	13	13	*13*	13	13	**14**
elevators (2008) (30)	**22**	21	21	**22**	**22**	**22**	**22**	***22***	18	21	**22**
elevators (2011) (20)	**18**	17	17	**18**	**18**	**18**	**18**	***18***	15	17	**18**
floortile (20)	**7**	6	6	**7**	**7**	**7**	6	***7***	6	**7**	**7**
freecell (80)	15	15	15	15	15	15	15	*15*	15	15	15
grid (5)	2	2	2	2	2	2	2	*2*	2	2	2
gripper (20)	**7**	6	**7**	**7**	**7**	**7**	**7**	***7***	6	**7**	**7**
logistics00 (28)	20	20	20	20	20	20	20	*20*	20	20	20
logistics98 (35)	6	6	6	6	6	6	6	*6*	6	6	6
miconic (150)	141	141	141	**142**	**142**	**142**	**142**	***142***	141	141	141
mprime (35)	22	22	22	**24**	23	23	23	*23*	22	23	**24**
mystery (30)	17	**18**	**18**	17	17	**18**	**18**	***22***	16	17	17
nomystery (2011) (20)	14	**18**	**18**	16	17	**18**	**18**	***18***	14	15	15
openstacks (2006) (30)	7	7	7	7	7	7	7	*7*	7	7	7
openstacks (2008) (30)	18	18	18	18	18	18	18	*18*	**26**	**26**	**26**
openstacks (2011) (20)	13	13	13	13	13	13	13	*13*	**19**	**19**	**19**
parcprinter (2008) (30)	**18**	**18**	**18**	**18**	**18**	**18**	**18**	***18***	17	17	**18**
parcprinter (2011) (20)	**13**	**13**	**13**	**13**	**13**	**13**	**13**	***13***	12	12	**13**
parking (20)	2	2	2	**5**	3	2	2	*3*	1	3	3
pathways (30)	5	5	5	5	5	5	5	*5*	5	5	5
pegsol (2008) (30)	27	27	27	28	27	28	28	*28*	27	**29**	28
pegsol (2011) (20)	17	17	17	18	17	18	18	*18*	17	**19**	18
pipesworld-notankage (50)	17	16	16	**18**	**18**	17	17	***18***	16	**18**	**18**
pipesworld-tankage (50)	**12**	11	11	**12**	11	11	11	***12***	10	**12**	**12**
psr-small (50)	**49**	48	**49**	**49**	**49**	**49**	**49**	***49***	48	**49**	**49**
rovers (40)	7	7	7	7	7	7	7	*7*	7	7	7
satellite (36)	7	7	7	**9**	7	7	7	*7*	7	7	7
scanalyzer (2008) (30)	15	15	15	**16**	15	15	**16**	***16***	15	15	**16**
scanalyzer (2011) (20)	12	12	12	**13**	12	12	**13**	***13***	12	12	**13**
sokoban (2008) (30)	**30**	29	**30**	**30**	**30**	**30**	**30**	***30***	27	**30**	**30**
sokoban (2011) (20)	**20**	**20**	**20**	**20**	**20**	**20**	**20**	***20***	19	**20**	**20**
tidybot (20)	14	**16**	**16**	14	14	15	**16**	***16***	13	14	14
tpp (30)	6	6	6	6	6	6	6	*6*	6	**7**	6
transport (2008) (30)	**11**	**11**	**11**	**11**	**11**	**11**	**11**	***11***	10	**11**	**11**
transport (2011) (20)	6	**7**	**7**	**7**	6	**7**	**7**	***7***	5	6	6
trucks (30)	10	10	10	10	10	10	10	*10*	10	10	10
visitall (20)	10	11	**12**	11	11	11	**12**	***12***	10	**12**	11
woodworking (2008) (30)	16	18	18	**20**	19	19	19	*19*	17	**20**	**20**
woodworking (2011) (20)	11	12	12	**14**	13	13	13	*13*	11	**14**	**14**
zenotravel (20)	**13**	12	12	**13**	12	12	12	*12*	11	12	12
Coverage (1396)	757	762	766	783	770	775	778	*786*	744	787	**790**
Coverage score (in %)	52.5	53.1	53.4	54.8	53.5	54.0	54.4	*55.0*	51.2	55.1	**55.4**
Time score (min in %)	40.0	44.8	**45.0**	42.3	44.2	44.8	44.9	*45.2*	37.5	42.0	43.2
Time score (max in %)	40.3	**52.7**	52.2	42.8	45.0	45.9	46.8	*45.2*	37.5	42.6	44.0
Time score (55 **s in %)**	32.8	38.7	**38.8**	35.3	38.4	38.6	38.6	*38.9*	30.5	34.8	36.3

Table 1: Coverage and time score results. Results for $h^{\text{iLM-cut}}_{\text{dynamic}}$ are interpolated and not regarded for the maximum.

Discard Information After Expansion. Stored landmarks of a node are only needed to compute the heuristic value of its successor nodes. Once the node has been expanded and the heuristic value of its successors is known, the only situation where its landmarks could be needed again is if it would be reopened, which rarely happens. In a first try to reduce the memory requirements we discard the stored landmarks of a node as soon as all its children are generated. This idea is somewhat related to frontier search (Korf 1999) as it only stores landmarks for nodes on the search frontier and will thus be called $h^{\text{iLM-cut}}_{\text{frontier}}$. However, since only landmarks and no nodes are discarded, the search algorithm remains a classical A* search.

Comparing A* with $h^{\text{iLM-cut}}$ and $h^{\text{iLM-cut}}_{\text{frontier}}$ shows that this change can reduce the required memory by factors of up to 3.5 with a geometric mean of 1.05. The time score remains more or less unaffected[2] (cf. Table 1). This modest but steady improvement in available memory leads to one additional solved problem in four domains. As this technique did not seem to negatively influence the time score, it was also included in the following strategies for A* search.

Figure 1 shows a scatter plot of the search times needed for A* with $h^{\text{iLM-cut}}_{\text{frontier}}$ and $h^{\text{LM-cut}}$. The incremental computation improves the search time with few exceptions by up to one order of magnitude. When ignoring easy problems (solved with $h^{\text{LM-cut}}$ in under 1 s) the runtime was reduced by 77% in the geometric mean.

The domains NoMystery and Miconic seem to benefit particularly well from the incremental computation with runtime reductions of 98% and 93% respectively (geometric mean). In Miconic the number of expansions is the same for the incremental and non-incremental computation, so the decrease in runtime purely results from the faster heuristic computation. For NoMystery the number of expansions is also decreased by incremental computation.

Fixed Memory Bounds. Failing a planner run because there are too many stored landmarks should not be necessary, because memory can be freed up by removing some of them. This potentially slows down the search because these landmarks need to be recomputed, but if the alternative is running out of memory, discarding landmark information should always be preferred. The amount of memory available to store landmarks can thus be fixed and some of the information can be discarded whenever this bound is reached. To distinguish the reserved space from the overall memory limit we call it *bound* in the following.

There are different possibilities for picking landmarks that should be discarded once the bound is reached. We decided to sort all nodes with attached landmarks according to their f-value. Landmarks of a node with highest f-value are then repeatedly discarded until the memory consumption of all stored landmarks is below half the desired bound. The reason for picking nodes with a high f-value is that nodes σ with $f(\sigma) > h^*$ are never expanded in A*. Removing half of the stored landmarks can be beneficial to amortize the time used to sort the nodes according to their f-values. For a

[2]The minimum score is actually increased slightly, but this can be due to noise.

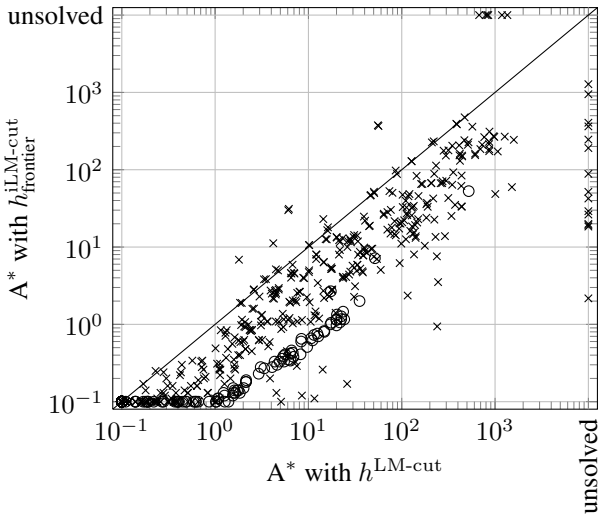

Figure 1: Search times in seconds for incremental and non-incremental LM-cut. Miconic domain printed as circles.

bound m the resulting heuristic is called $h^{\text{iLM-cut}}_m$. Note that $h^{\text{iLM-cut}}_{\text{frontier}}$ is the same as $h^{\text{iLM-cut}}_\infty$. This technique could be further improved by only marking elements as available and deleting them on demand, as in lazily discarding batch replacement strategies for transposition tables (Akagi, Kishimoto, and Fukunaga 2010). In our case the data structures used to store landmarks made it hard to implement this technique efficiently. Although the performance actually decreased with batch replacement, it could still be a possible improvement with different data structures. We leave this as future work and in the following discuss results achieved without batch replacement.

Note that the amount of memory freed differs from node to node for two reasons. Firstly, two nodes might have different amounts of (and differently sized) landmarks. Secondly, landmarks are stored as reference-counting pointers and thus are only discarded once the last pointer to them is discarded. Other ways to maintain the fixed bound are certainly possible but were not evaluated. One could for example discard nodes randomly.

A planner run can fail for two reasons: running out of time and running out of memory. Figure 2 shows the number and distribution of failure conditions for different bounds on the available memory for landmarks. Not surprisingly the number of failures caused by exhausting memory increases with an increasing bound. The number of timeouts is also expected to decrease as large instances that result in a timeout with a small bound can run out of memory if more space is reserved.

However, the results also show that the total number of failures decreases with an increasing bound up to an optimal bound of 500 MB. This means that some instances that could not be solved when only few landmarks are stored, can be solved when more landmarks are stored and the heuristic can be computed incrementally more often. For the best bound of 500 MB the coverage is 778, which corresponds

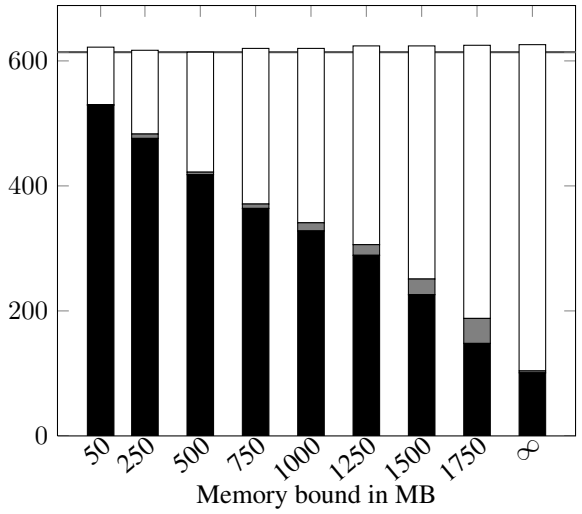

Figure 2: Distribution of failure conditions for different bounds. Bars show the number of instances that were stopped because they reached their time (black) or memory (white) limit. Instances were over 99% of both resources where used when the task was stopped are shaded in gray.

to 12 additional tasks solved compared to $h^{\text{iLM-cut}}_{\text{frontier}}$ and 21 additional tasks compared to $h^{\text{LM-cut}}$.

Dynamic Memory Bound. As the optimal bound strongly depends on the problem, the search algorithm should ideally not use a fixed memory bound but a dynamic one ($h^{\text{iLM-cut}}_{\text{dynamic}}$). In that case all available memory could be used to store landmarks until it is needed by the search algorithm to store search nodes. This is difficult to implement for several reasons. Measuring the memory pressure of a complex program accurately is not trivial, for one. Predicting and limiting memory requirements is also hard, because data structures use overallocation to ensure their amortized runtime properties. This change would also have required changing large parts of the planner's implementation, so it was not evaluated.

We conjecture that an instance which can be solved with at least one fixed bound can also be solved with a dynamic bound. For such a task the dynamic bound will be at least as high as the fixed bound and more landmarks can be stored. The search should not result in a timeout because storing more landmarks typically decreases the search time as long as there is enough available memory. It should also not run out of memory as the search with a fixed bound fits into memory and all additionally used memory can be freed on demand. If the results are interpolated over the results for fixed bounds the estimated coverage is 786 or another 8 additionally solved tasks compared to the best bound (see Table 1).

Local Incremental Computation. Another approach to save memory and still benefit from incremental computation is to use it only locally. This relies on the observation that A* search typically expands fewer nodes than the number of nodes that are generated. The geometric mean over the ratio of generated nodes to expanded nodes ranges from 2.47 (Visitall) to 39.00 (Mprime). The geometric mean over all domains for this ratio is 8.05. This result shows that additional work during the expansion of a node can be amortized if it saves time during the evaluation of its successors.

Incremental computation can thus be used in the following way (called $h^{\text{iLM-cut}}_{\text{local}}$). During the expansion of a node its heuristic is (re-)calculated using the non-incremental LM-cut heuristic before its children are generated. The resulting landmarks are stored and used for the incremental computation of its child nodes' heuristic values. After the generation of all child nodes, the stored landmarks can be discarded. This means that there are two heuristic calculations for every expanded node (with the exception of the initial node): (i) a (fast) incremental computation when it is generated from its parent, and (ii) a (slow) non-incremental computation when it is expanded.

This introduces an overhead for each node expansion but decreases the time needed to generate a child node. The slow, non-incremental computation is only used for nodes that are actually expanded. If the incremental computation is sufficiently faster than the non-incremental one, the additional heuristic computation can be compensated.

The coverage for A* with $h^{\text{iLM-cut}}_{\text{local}}$ is 783 which outperforms $h^{\text{LM-cut}}$ by 26 and $h^{\text{iLM-cut}}$ by 17 instances (see Table 1). The time score is between that of $h^{\text{LM-cut}}$ and the score of $h^{\text{iLM-cut}}$.

Figure 3 shows a scatter plot of the search times for A* with $h^{\text{iLM-cut}}_{\text{local}}$ and our baseline algorithm. The runtime of non-trivial problems was reduced by 49% geometric mean. As with $h^{\text{iLM-cut}}$, the domain Miconic seems to benefit from the incremental computation particularly well with runtime reductions of 89% (geometric mean). Openstacks (2008 and 2011) is the only domain where instances show a negative tendency in the geometric mean (40% increase). This might be due to the structure of Openstacks tasks which is discussed later.

Application to IDA* Search

When memory restrictions are too tight for an A* search, the IDA* algorithm is an often used alternative. It traverses the nodes in depth-first order and only needs to keep one branch of the search tree in memory at all times. Transposition tables with limited memory are usually used to store some of the closed nodes and detect duplicates. A larger transposition table requires more space but allows to detect more duplicates. A* search with the non-incremental LM-cut heuristic rarely ran out of space (30 out of 1396 cases), i.e., all nodes encountered during A* search typically fit into memory with this heuristic. This allows us to use IDA* with an unlimited-size transposition table that stores all encountered states. Such an IDA* search expands and reopens the same nodes as A* but is repeated for increasing limits of f until a solution is found. This is normally not done, as it has the high space requirements of A* and slower computation time of IDA*. In this case, however, the full transposition table fits into memory and the depth-first expansion order

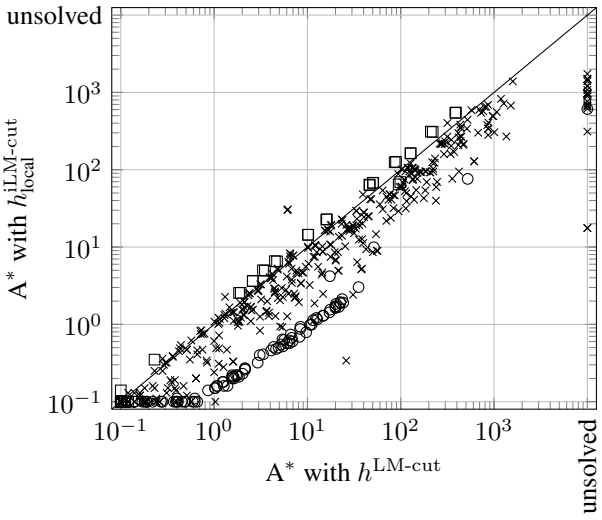

Figure 3: Search times in seconds for non-incremental LM-cut and incremental LM-cut used only for siblings. Miconic domain printed as circles and Openstacks printed as squares.

of IDA* can be used to significantly reduce the number of stored landmarks. Although all states have to be stored in the transposition table for duplicate detection, it is sufficient to only store the discovered landmarks along the current branch of the search tree for incremental computation.

With (non-incremental) $h^{\text{LM-cut}}$, IDA* with full duplicate detection has no advantage over A* and the coverage drops by 13 tasks to 744 in this case (see Table 1). These results are skewed by a surprisingly good performance of IDA* in the domain Openstacks. A total of 14 additional Openstacks tasks (8 in the 2008 variant and 6 for 2011) could be solved with IDA*. We believe this to be a result of better tie-breaking. In this domain, the large number of 0-cost operators leads to an LM-cut value of 1 in all states except the goal state. Both A* and IDA* search thus have no information regarding which successor to prefer. In the case of Openstacks it is always useful to apply applicable 0-cost operators. A depth-first search such as IDA* implicitly breaks ties in favor of a longer plan, i.e., more applied actions, and so has an advantage in this domain.

The effect of incrementally computing the LM-cut heuristic in this scenario are clearly positive: a total of 787 tasks could be solved this way. This means that IDA* with $h^{\text{iLM-cut}}_{\text{frontier}}$ could solve 43 more tasks than IDA* with $h^{\text{LM-cut}}$ and 30 additional tasks compared to A* with $h^{\text{LM-cut}}$ (see Table 1). Even if we do not count the 14 additional tasks in Openstacks, this is still a significant improvement.

Speeding Up Known Parts of the IDA* Search Tree

Every iteration of IDA* repeats the search effort done in the previous iteration. The assumption is that the number of newly generated nodes in an iteration (i.e., the size of the *IDA* layer*) increases exponentially, so the last iteration takes up the bulk of the computation time. When IDA* regenerates nodes in the search tree from the previous iteration, heuristic values for such nodes are recomputed. In a classical IDA* search this makes sense, as there is no reliable way to recognize if a node is part of the already explored part of the search tree. In the case of an unlimited transposition table, however, this can be determined by a simple hash table lookup. The heuristic value can be saved together with a state in the transposition table and instead of recomputing it, it can be looked up. If a heuristic computation takes significantly longer than a lookup, IDA* search will then be able to traverse the existing tree much faster. This results in a new problem in the case of incremental computation. The heuristic values for the first nodes outside the already explored part cannot be computed incrementally, because the landmarks for their parents are not stored. The value of such a node could be computed non-incrementally. Alternatively, its parent's heuristic value could be computed non-incrementally, allowing an incremental computation for the node itself. We argue that the latter makes more sense in this case. Nodes that are not yet saved in the transposition table are successors of nodes that were pruned in the last iteration of IDA*. The current iteration generates such a node, because IDA*'s limit on the f-value just passed its parent's f-value. This means that all of its siblings will also be generated in this IDA* iteration. By using non-incremental computation for its parent instead of itself, the parent's landmark information will become available. The heuristic values of all the node's siblings can then be incrementally computed. We call the modified IDA* search that looks up known f values in its transposition table and uses non-incremental computation for parent nodes IDA$^*_{\text{L}}$.

This idea is similar to that of $h^{\text{iLM-cut}}_{\text{local}}$ in A* as described earlier. If the heuristic function is inaccurate, the number of IDA* iteration increases and the IDA* layers contain fewer nodes. Thus the length of operator sequences that can be applied without increasing the f-value decreases. If all such sequences have length 1, the search calculates the heuristic twice for every node. The situation is different for more accurate heuristics, when the number of nodes per IDA* layer is larger and thus such chains are longer. In this case, the heuristic can be computed incrementally for a larger subtree and the cost for the non-incremental computation can be amortized. This technique can be particular effective for domains with a large branching factor, as this also increases the size of the subtree that can be evaluated incrementally. Adding this technique to our IDA* search increased its coverage by another 3 tasks to 790 (see Table 1). Figure 4 shows a scatter plot comparing IDA$^*_{\text{L}}$ with $h^{\text{iLM-cut}}_{\text{frontier}}$ to our baseline. Runtimes are decreased by 48% in the geometric mean over all domains. This technique has the largest impact in the domains Airport and Miconic, reducing the runtime by 96% and 92% respectively.

Conclusion

We have introduced a way to incrementally compute landmarks with the LM-cut heuristic to speed up the heuristic calculation during the search. The incremental computation is up to an order of magnitude faster than the non-incremental computation but uses more memory. We introduced several ways that trade off the gained speed against

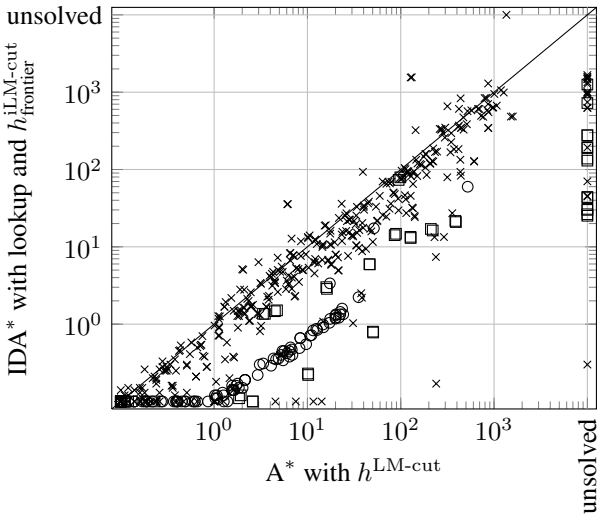

Figure 4: Search times in seconds for A* with non-incremental LM-cut and IDA* with incremental LM-cut and reevaluations for parent nodes. Miconic domain printed as circles and Openstacks printed as squares.

Domain - Problem	A*/$h^{\text{LM-cut}}$	A*/$h^{\text{iLM-cut}}$	A*/$h^{\text{iLM-cut}}_{\text{frontier}}$	A*/$h^{\text{iLM-cut}}_{100\text{MB}}$	A*/$h^{\text{iLM-cut}}_{500\text{MB}}$	A*/$h^{\text{iLM-cut}}_{\text{local}}$	IDA*/$h^{\text{LM-cut}}$	IDA*$_{\text{L}}$/$h^{\text{iLM-cut}}_{\text{frontier}}$
airport #38	290	**13**	14	14	15	159	293	**13**
airport #25	T	191	191	**177**	191	707	1160	193
elevators (2008) #17	982	271	**266**	908	392	493	T	635
floortile #04-7	1186	M	M	938	M	**827**	T	1060
miconic #17-4	522	**50**	53	58	55	76	516	60
mprime #19	T	M	M	844	1177	969	1782	**744**
mystery #06	1006	53	**49**	50	50	305	T	949
mystery #14	T	442	**363**	**363**	491	T	T	T
pipesworld-tankage #13	1270	285	**268**	521	290	718	T	990
scanalyzer (2011) #06	866	182	**174**	492	269	339	1270	343
sokoban (2008) #28	1589	M	244	**226**	262	1390	T	485
tidybot #12	1068	**172**	**172**	190	177	558	T	670
tidybot #20	T	1423	1278	T	**1266**	T	T	T
transport #07	T	250	**245**	T	553	1734	T	T
woodworking (2008) #15	T	M	M	1303	663	655	T	626
woodworking (2011) #20	T	19	19	21	20	**18**	T	45
zenotravel #13	1355	M	M	T	T	**269**	T	T

Table 2: Runtimes for hard problems in seconds. If no runtime is given the task exhausted time (T) or memory (M).

the lost memory. With A* search the best performing heuristic used the incremental computation only locally to generate the successor nodes. With IDA* search and an unbounded transposition table an even larger improvement in coverage was achieved (33 additional tasks compared to the baseline). This heuristic also used incremental computation locally by looking up f values in the known part of the search tree and recomputing heuristic values for parents of the first nodes in the unknown part.

Table 2 shows explicit runtime results of tasks that are the hardest in their domain for either A*/$h^{\text{LM-cut}}$ or A*/$h^{\text{iLM-cut}}$. The fast incremental methods frequently save 90% or more of the runtime compared to non-incremental LM-cut. If the full incremental heuristic $h^{\text{iLM-cut}}$ fails, the reason is often exhausted memory. The heuristics $h^{\text{iLM-cut}}_{\text{frontier}}$, $h^{\text{iLM-cut}}_{\text{m}}$ and $h^{\text{iLM-cut}}_{\text{local}}$ are increasingly more memory friendly and can solve tasks that do not fit in memory with $h^{\text{iLM-cut}}$. With an IDA*$_{\text{L}}$ search memory requirements can also be reduced, leading to a performance comparable to that of A* with $h^{\text{iLM-cut}}_{\text{local}}$.

Comparing the two best-performing heuristics to an A* search with $h^{\text{iLM-cut}}$ suggests that there is still room for improvement for the search times (see Table 2). This full incremental computation most often fails by hitting the memory limit. Thus, one way to improve its performance would be a more memory-efficient implementation. For example, compressed bitsets could be used to store the landmarks and reduce their memory footprint.

Another way to make the best use of all available memory is to use a dynamic memory bound ($h^{\text{iLM-cut}}_{\text{dynamic}}$). This would require keeping track of all allocated memory in the planning system to correctly assess the remaining free memory. Alternatively the memory bounded version ($h^{\text{iLM-cut}}_{k}$) could be combined with $h^{\text{iLM-cut}}_{\text{local}}$. Landmark information could then be stored as long as there is memory available. Heuristic calculations for nodes where the parent did not store landmarks could be performed by computing the landmarks for the parent, storing them temporarily and then computing the incremental heuristic for its successor states.

The techniques presented in this paper can also be applied to other heuristics that benefit from incremental computation but require too much memory to store all information. For example, the optimal cost partitioning for landmark heuristics (Karpas and Domshlak 2009) is based on solving a linear program. This LP is similar for a node and its successor and thus could be computed incrementally. This would also require storing a large amount of data for each search node and could thus benefit from the techniques discussed here. As a second example, consider the improved LM-cut heuristic by Bonet and Helmert (2010). Its value is based on a hitting set computation, which could also reuse information from the parent node. The information contained in the hitting set problem is the same that is contained in the landmarks, so memory restrictions will also be an issue here. Finally, as mentioned earlier, the MAXSAT based heuristic by Zhang and Bacchus (2012) could store the landmarks discovered by LM-cut to discover the landmarks for successor nodes faster. It could also store parts of the MAXSAT computation and perform this computation incrementally.

Acknowledgments

This work was supported by DFG grant HE 5919/2-1.

References

Akagi, Y.; Kishimoto, A.; and Fukunaga, A. 2010. On transposition tables for single-agent search and planning: Summary of results. In *Proceedings of the 3rd Annual Symposium on Combinatorial Search (SoCS 2010)*, 2–9.

Betz, C., and Helmert, M. 2009. Planning with h^+ in theory and practice. In *Proceedings of the 32nd Annual German Conference on Artificial Intelligence (KI 2009)*, volume 5803 of *Lecture Notes in Artificial Intelligence*, 9–16.

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

Bonet, B., and Helmert, M. 2010. Strengthening landmark heuristics via hitting sets. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, 329–334.

Burns, E.; Hatem, M.; Leighton, M. J.; and Ruml, W. 2012. Implementing fast heuristic search code. In *Proceedings of the 5th Annual Symposium on Combinatorial Search (SoCS 2012)*, 25–32.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69:165–204.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 1728–1733.

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

Korf, R. E. 1999. Divide-and-conquer bidirectional search: First results. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999)*, 1184–1189.

Liu, Y.; Koenig, S.; and Furcy, D. 2002. Speeding up the calculation of heuristics for heuristic search-based planning. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI 2002)*, 484–491.

Pommerening, F., and Helmert, M. 2012. Optimal planning for delete-free tasks with incremental LM-cut. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS 2012)*, 363–367.

Pommerening, F. 2011. Optimal planning for delete-free tasks with incremental LM-cut. Master's thesis, Albert-Ludwigs-Universität Freiburg.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 273–280.

Zhang, L., and Bacchus, F. 2012. MAXSAT heuristics for cost optimal planning. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI 2012)*, 1846–1852.