Landmark-based Meta Best-First Search Algorithm: First Parallelization Attempt and Evaluation

Simon Vernhes, Guillaume Infantes and Vincent Vidal
Onera Toulouse, France
firstname.lastname@onera.fr

Abstract

In this paper, we revisit the idea of splitting a planning problem into subproblems hopefully easier to solve with the help of landmark analysis. While this technique initially proposed in the first approaches related to landmarks has been outperformed by landmark-based heuristics, we believe that it is still a promising research direction. To this end, we propose a new method for problem splitting based on landmarks which has two advantages over the original technique: it is complete (if a solution exists, the algorithm finds it), and it uses the precedence relation over the landmarks in a more flexible way. We lay in this paper the foundations of a meta best-first search algorithm, which explores the landmark orderings to create subproblems and can use any embedded planner to solve subproblems. Furthermore, we propose and evaluate a parallel version of this algorithm. It opens up avenues for future research: among them are new heuristics for guiding the meta search towards the most promising orderings, different policies for generating subproblems, influence of the embedded subplanner and other parallelization strategies of the meta search.

1 Introduction

Automated Planning in Artificial Intelligence (Ghallab, Nau, and Traverso 2004) is a general problem solving framework which aims at finding solutions to combinatorial problems formulated with concepts such as actions, states of the world, and goals. Landmark-based analysis is actually among the most popular tools to build efficient planning systems, either optimal or suboptimal. Landmarks are facts that must be true at some point during the execution of any solution plan, and some of them can be found, as well as an ordering, in polynomial time (Hoffmann, Porteous, and Sebastia 2004; Keyder, Richter, and Helmert 2010). Landmarks have been used in two main ways. The most successful one is the design of heuristic functions to guide search algorithms, such as the landmark-counting heuristic used in the LAMA suboptimal planner (Richter, Helmert, and Westphal 2008) or the LM-Cut heuristic for optimal cost-based planning (Helmert and Domshlak 2009). An anterior method proposed in (Hoffmann, Porteous, and Sebastia 2004) is to divide a planning problem into successive subproblems whose goals are disjunctions of landmarks to be achieved in turn by any embedded planner. This method is not as efficient as using landmark-based heuristics: among the most prominent problems are its incompleteness and its lack of flexibility with respect to an initial ordering of the landmarks. STeLLa (Sebastia, Onaindia, and Marzal 2006) is another problem-splitting method which creates a set of subproblems using conjunctions of landmarks.

Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

We aim in this paper to revisit these last methods, with the objective of devising a complete algorithm for subproblem splitting based on landmarks. Roughly speaking, our method consists in performing a best-first search algorithm in the space of landmark orderings, in which node expansion implies the search of a subproblem by an embedded planner, these orderings being explored in parallel. This search algorithm is performed at a meta level, the low level being the search made by the embedded planner that can itself use a best-first search algorithm. After giving some background about classical planning and landmark computation, we define the basic components later used to describe the landmark-based meta best-first search algorithm (LMBFS), along with several heuristics to guide the meta search. We then propose a parallel version of this algorithm and experimentally evaluate both sequential and parallel versions. We finally conclude and present some perspectives for future works.

2 Background on Classical Planning

2.1 STRIPS Model of Planning

The basic STRIPS (Fikes and Nilsson 1972) model of planning can be defined as follows. A *state* of the world is represented by a set of ground atoms. A *ground action* a built from a set of atoms A is a tuple $\langle pre(a),add(a),del(a)\rangle$ where $pre(a) \subseteq A$, $add(a) \subseteq A$ and $del(a) \subseteq A$ represent the preconditions, add and delete effects of a, respectively.

A *planning problem* is defined as a tuple $\Pi = \langle A, O, I, G\rangle$, where A is a finite set of *atoms*, O is a finite set of ground actions built from A, $I \subseteq A$ represents the *initial state*, and $G \subseteq A$ represents the *goal* of the problem. The *application* of an action a to a state s is possible if and only if $pre(a) \subseteq s$ and the resulting state is $s' = (s \setminus del(a)) \cup add(a)$. A *solution plan* is a se-

quence of actions $\langle a_1, \ldots, a_n \rangle$ such that for $s_0 = I$ and for all $i \in \{1, \ldots, n\}$, the intermediate states $s_i = (s_{i-1} \setminus del(a_i)) \cup add(a_i)$ are such that $pre(a_i) \subseteq s_{i-1}$ and $G \subseteq s_n$. $S(\Pi)$ denotes the set of all solution plans of Π.

We also denote $\circ$ the concatenation of two plans, i.e. $\langle a_1, \ldots, a_i \rangle \circ \langle a_j, \ldots, a_k \rangle = \langle a_1, \ldots, a_i, a_j, \ldots, a_k \rangle$.

2.2 Landmarks

Classical landmark definitions state that *landmarks* are facts that must be true at some point during the execution of any solution plan (Hoffmann, Porteous, and Sebastia 2004; Keyder, Richter, and Helmert 2010). We use the following definition of landmarks:

Definition 1 (Causal landmark). *(Zhu and Givan 2003) Given a planning problem $\Pi = \langle A, O, I, G \rangle$, an atom l is a* causal landmark *for Π if either $l \in G$ or $\forall \rho \in S(\Pi), \exists a \in \rho : l \in pre(a)$.*

An intuitive precedence relation among landmarks and a graph based on this relation can be defined as follows:

Definition 2 (Precedence relation $<_{\mathcal{L}}$). *$<_{\mathcal{L}}$ can be defined on a set of landmarks $\mathcal{L}$: $(\forall (l, l') \in \mathcal{L}^2)$ $l <_{\mathcal{L}} l'$ if the first occurence of l is reached before the first occurence of l' by the execution of every solution plan.*

Definition 3 (Landmark graph Γ). *Given a set of landmarks $\mathcal{L}$ and a precedence relation $<_{\mathcal{L}}$, we define $\Gamma = (\mathcal{V}, \mathcal{E})$, the corresponding landmark directed graph where the set of vertices $\mathcal{V} = \mathcal{L}$ and the set of edges $\mathcal{E}$ is the transitive reduction of the graph $(\mathcal{V}, \{(l, l') \in \mathcal{L}^2 \mid l <_{\mathcal{L}} l'\})$.*

Definition 4 (Relatives of a landmark l). *Accordingly to the graph Γ, we denote $Pa_\Gamma(l)$ the set of parents of l, $Ch_\Gamma(l)$ the set of children of l, and $\mathcal{P}_\Gamma(l)$ the set of ancestors of l.*

We now introduce the following definitions that we will rely on. First, we denote *root landmarks* of a landmark graph all landmarks associated to vertices with no parents:

Definition 5 (Root landmark set). *Let $\Gamma = (\mathcal{V}, \mathcal{E})$ be a landmark graph: $roots(\Gamma) = \{l \in \mathcal{V} \mid Pa_\Gamma(l) = \emptyset\}$.*

We now define the subgraph $\Gamma \setminus \mathcal{A}$ built by removing from the landmark graph Γ the vertices associated to landmarks in $\mathcal{A}$ and their corresponding edges:

Definition 6 (Landmark subgraph). *Let $\Gamma = (\mathcal{V}, \mathcal{E})$ be a landmark graph and $\mathcal{A}$ be a set of landmarks: $\Gamma \setminus \mathcal{A} = (\mathcal{V} \setminus \mathcal{A},\ \{(v, v') \in \mathcal{E} \mid v \notin \mathcal{A} \wedge v' \notin \mathcal{A}\})$.*

Landmark Graph Generation Practicals methods proposed to produce landmark graphs (Hoffmann, Porteous, and Sebastia 2004; Zhu and Givan 2003) are based on a Relaxed Planning Graph (RPG) of Π. More complex types of landmarks might be considered (Keyder, Richter, and Helmert 2010). In this work, we choosed the method of (Zhu and Givan 2003) for its simplicity.

Related Works on Using Landmarks Previous approaches used landmarks in mainly two different ways. One approach is computing heuristics. For example, the LAMA heuristic (Richter, Helmert, and Westphal 2008) estimates a heuristic value of the states by counting unreached and required again landmarks. Another approach is to split a planning problem into subproblems. Disjunctive Search Control (DSC) (Hoffmann, Porteous, and Sebastia 2004) is a search control algorithm based on the landmark graph. It runs a subplanner on the problem Π whose goal is the disjunction of the roots of the landmark graph and G. If a valid plan is found, then the reached landmark is removed from the landmark graph and the algorithm iterates (the reached state is used as the new initial state) until the landmark graph is empty. Finally, the subplanner is called a last time with G as goal.

3 The Landmark-based Meta Best-First Search (LMBFS) Algorithm

Our approach is based on problem splitting with a flexible exploitation of the landmark graph: LMBFS performs a best-first search in a space of subproblems generated on-the-fly, based on possible landmark orderings.

More precisely, LMBFS builds a search tree where nodes represent planning problems that are subproblems of the original one. Solving subproblems along a branch of this tree leads to iteratively reach each landmark in a possible ordering, the initial state of each subproblem being the final state obtained by applying the plan found for the previous subproblem. We formally define in this section the metanodes and associated planning problems, as well as different ways of generating the next subproblems to solve from a metanode (the children of that metanode in the search tree). Both aspects heavily rely on the landmark graph and on the partial order it defines. We then give the complete algorithm, heuristics used and implementation details.

In the following, we consider a planning problem $\Pi = \langle A, O, I, G \rangle$, its corresponding set of landmarks $\mathcal{L}$, and Γ the landmark graph associated to Π.

3.1 Metanode and Associated Planning Problem

We first define metanodes and associated problems:

Definition 7 (Metanode). *A* metanode *is a tuple $m = \langle s, h, \mathcal{A}, l, \rho \rangle$ where:*

- *s is a state of the planning problem Π;*
- *h is a heuristic evaluation of the node;*
- *$\mathcal{A}$ is a set of landmarks ($\mathcal{A} \subseteq \mathcal{L}$);*
- *l is a landmark ($l \in \mathcal{L}$);*
- *ρ is a plan yielding the state s from the initial state I.*

Definition 8 (Metanode-associated planning problem). *The planning problem associated to a metanode $m = \langle s, h, \mathcal{A}, l, \rho \rangle$ is $\Pi_m = \langle A, ops_\Gamma(l, \mathcal{A}), s, \{l\} \rangle$ with $ops_\Gamma(l, \mathcal{A})$ a subset of O defined below.*

We consider the planning problem where s is the initial state, A is the set of ground atoms of Π, $\{l\}$ is the goal. In order to focus search on reaching l, we forbid the actions producing some other landmarks by defining $ops_\Gamma(l, \mathcal{A})$ as a subset of actions associated to a landmark subgraph:

Definition 9 (Landmark subgraph action restriction)**.** *Let $m = \langle s, h, \mathcal{A}, l, \rho \rangle$ be a metanode. $ops_\Gamma(l, \mathcal{A}) = \{a \in O \mid (l \in add(a)) \vee (add(a) \cap roots(\Gamma \setminus \mathcal{A}) = \emptyset)\}$.*

In other words, $ops_\Gamma(l, \mathcal{A})$ is the set of actions producing l and actions which does not produce any root landmarks of the subgraph $\Gamma \setminus \mathcal{A}$ (except if they also produce l). In our algorithm, $\mathcal{A}$ will be the set of already achieved landmarks.

3.2 Expansion of Metanodes

There are several ways to generate children of a metanode, or equivalently defining other subproblems to solve. Let us recall that a metanode $m = \langle s, h, \mathcal{A}, l, \rho \rangle$ defines a problem starting from s and focusing on achievement of landmark l by forbidding actions producing other landmarks of $roots(\Gamma \setminus \mathcal{A})$ (except if they also produce l). In the following, h' is the heuristic evaluation of the generated metanode, discussed in section 3.4.

Next Landmarks Metanode Generation This first metanode generator tries to follow the landmark graph Γ as closely as possible, exploring sequences from the roots to the leaves. The idea is the following: when the goal landmark of the metanode can be reached, generate children in order to reach other root landmarks in Γ. We thus define the *nextLM* operator as:

Definition 10 (Next landmarks metanode generation)**.** *Let $m = \langle s, h, \mathcal{A}, l, \rho \rangle$ be a metanode. If Π_m has a solution ρ', then $\mathrm{nextLM}(m) = \{\langle s', h', \mathcal{A} \cup \{l\}, l', (\rho \circ \rho') \rangle \mid l' \in roots(\Gamma \setminus (\mathcal{A} \cup \{l\}))\}$ where s' is the state obtained by applying ρ' to s. If Π_m has no solution, $\mathrm{nextLM}(m) = \emptyset$.*

In other words, at a metanode m, we try to reach the landmark l. If there is a plan, the landmark l is added to the set of already achieved landmarks $\mathcal{A}$, and the partial plan is updated accordingly. Then, new metanodes are generated by looking at root landmarks in the restricted graph $\Gamma \setminus (\mathcal{A} \cup \{l\})$.

Remark. Using nextLM, we can explore every total order created from the precedence relation $<_\mathcal{L}$, which was our objective. Indeed, consider a metanode focusing on an initial root landmark of Γ. If we generate its children using the nextLM operator, then selecting one of them and iterating the process in a depth-first way, we will eventually empty the landmark graph Γ, achieving the exploration of a total order of all landmarks.

Unfortunately, even if the landmark graph Γ is sound and complete, using only nextLM makes the algorithm incomplete, as shown in the following counterexample. Let us consider the example in Figure 1 where circles are atoms, squares are actions, arrows mean precondition of an action or production of an atom and dashed arrows mean deletion of an atom. The initial state is $\{a, f, d\}$ and the goal set is $\{c\}$. As we can see, g and c are landmarks, and g has to be reached before c. The first metanode will have the landmark g as goal. The subplanner gives the plan $\langle \alpha \rangle$. The only generated metanode added to the open list is $m = \langle \{f, g\}, h, \{g\}, c, \langle \alpha \rangle \rangle$. The associated problem Π_m

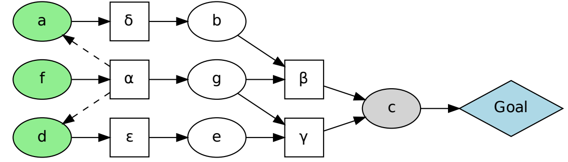


Figure 1: Planning Graph of "nextLM problem".

is unsolvable, as α deletes the preconditions of δ and ϵ which are mandatory actions that should be applied before α.

This led us to define new metanode generators, which, used in conjunction with nextLM, make the algorithm complete. These operators are based on landmark deletion from Γ, allowing for instance in the counterexample to try to reach the goal c without blindly focusing on g first.

Cut-parents Metanode Generation These operators remove the ancestors of a non-root landmark.

Definition 11 (Cut-parents metanode generations)**.** *Let $m = \langle s, h, \mathcal{A}, l, \rho \rangle$ be a metanode. If Π_m has a solution ρ', then $\mathrm{cutParent}(m) = \{\langle s', h', \mathcal{A} \cup \mathcal{P}_\Gamma(l'), l', (\rho \circ \rho') \rangle \mid l' \in Ch_\Gamma(l)\}$ where s' is the state obtained by applying ρ' to s. If Π_m has no solution, then $\mathrm{cutParent}(m) = \emptyset$.*

Definition 12 (Restart cut-parents metanode generation)**.** *$\mathrm{restartCutParent}(\langle s, h, \mathcal{A}, l, \rho \rangle) = \{\langle I, h', \mathcal{A} \cup \mathcal{P}_\Gamma(l'), l', \emptyset \rangle \mid l' \in Ch_\Gamma(l)\}$ where I is the initial state of the original planning problem.*

The idea is that a total order constructed on the partial order defined by the landmark graph may be too restrictive, as in the counterexample. Using these two operators, some landmarks may be skipped by trying to reach deeper landmarks.

Delete Landmark Metanode Generation Finally, we introduce the very generic landmark deletion operator: metanodes are generated as if the deleted landmark did not exist:

Definition 13 (Delete landmark metanode generation)**.** $\mathrm{deleteLM}(\langle s, h, \mathcal{A}, l, \rho \rangle) = \{\langle s, h', \mathcal{A} \cup \{l\}, l', \rho \rangle \mid l' \in roots(\Gamma \setminus (\mathcal{A} \cup \{l\}))\}$.

This operator discards a landmark, and causes the search to try to achieve remaining root landmarks. Applying this operator enough times on the first metanode (that has I as initial state) empties the landmark graph, eventually giving a metanode associated to the original planning problem. Also, the cut-parents operators can be seen as shortcuts for several delete landmark operators applications, guided by $\mathcal{P}_\Gamma$.

3.3 Algorithm

LMBFS (Algorithm 1) is a best-first search algorithm on metanodes of definition 7, with deferred heuristic evaluation (Richter and Helmert 2009): new nodes are inserted into the open list with the heuristic value of their parent.

Algorithm 1: LMBFS.

input : STRIPS problem $\Pi = \langle A, O, I, G\rangle$, landmark graph Γ, metanode successor function succ
output : solution plan (or $\perp$ if there is no solution)

```
1  open ← ∅; closed ← ∅;
2  ∀l ∈ roots(Γ) : add ⟨I, h, ∅, l, ∅⟩ to open;
3  while open ≠ ∅ do
4      m ← arg min_{⟨s,h,A,l,ρ⟩ ∈ open} h;
5      open ← open \ {m};
6      if m ∉ closed then
7          closed ← closed ∪ {m};
8          ρ' ← subplanner(Π_m);
9          if ρ' ≠ ⊥ then
10             s' ← result of executing ρ' in s;
11             if G ⊆ s' then
12                 return ρ ∘ ρ';
13             open ← open ∪ succ(m);
14 return ⊥
```

The algorithm is run on the problem $\Pi_g = \langle A \cup \{g\}, O \cup \{a_g\}, I, \{g\}\rangle$ where g is a dummy atom representing goal achievement, and a_g is a dummy action whose precondition is G and add effect is $\{g\}$. g is a landmark whose achievement implies that a solution to Π has been found.

First, the metanodes associated to each root landmark of Γ are added to the open list. Then, at each iteration, the best metanode m (according to a heuristic detailed in section 3.4) is extracted from the open list, and a subplanner is run on the associated problem Π_m. If the subplanner returns a plan, m is expanded by adding its *successors* to the open list. The algorithm iterates until the open list is empty or g is reached.

The function succ applied to the metanode m (Algorithm 1 line 13) computes the set obtained by an operator or the union set of several operators described in section 3.2. In our planner, we have implemented two successor functions:

- $\mathrm{succDel}(m) = \mathrm{nextLM}(m) \cup \mathrm{deleteLM}(m)$
- $\mathrm{succCut}(m) = \mathrm{nextLM}(m) \cup \mathrm{cutParent}(m) \cup \mathrm{restartCutParent}(m)$

The operator nextLM is at the heart of our algorithm in order to focus on sequences of landmarks. However to ensure completeness, we have to use a combination of the other operators: as a net effect of applying these operators at each node expansion, the metanode $m = \langle I, h', \mathcal{L} \setminus \{g\}, g, \emptyset\rangle$ corresponding to the global problem Π_g will appear.

Theorem 1. *The LMBFS algorithm using* $\mathrm{succCut}$ *or* $\mathrm{succDel}$ *as* $\mathrm{successor}$ *function is sound and complete if the subplanner is sound and complete.*

Proof. (sketch) Soundness comes from: (1) the state in the first metanode is the initial state I of the problem, (2) all successor operators build plans that can be concatenated to form a solution of the global problem or search a new plan from I, and (3) if the final landmark g appears in a metanode, then achieving it solves the global problem goal. (2) is obtained by induction: if a metanode $m = \langle s, h, \mathcal{A}, l, \rho\rangle$ is such that s is reachable by applying ρ from I, then by definition of nextLM and $\mathrm{cutParent}$, s' is the state obtained by applying ρ' to s and so s' is reachable by applying $\rho \circ \rho'$ from I (using subplanner soundness). $\mathrm{deleteLM}$ modifies neither s nor ρ, so the recursive property is ensured. $\mathrm{restartCutParent}$ produces a metanode $m = \langle I, h, \mathcal{A}, l, \emptyset\rangle$: a new search from I is started, giving a sound plan if the subplanner is sound.

Completeness comes from the fact that the operators $\mathrm{restartCutParents}$ (for the $\mathrm{succCut}$ successor function) and $\mathrm{deleteLM}$ (for the $\mathrm{succDel}$ successor function) are systematically used at each expansion of a metanode. So, the search graph contains a branch starting from the initial metanode consisting only of applications of $\mathrm{restartCutParents}$ or $\mathrm{deleteLM}$, and both will have the effect to (1) keep I as associated state, (2) put all landmarks but g in the set of landmarks $\mathcal{A}$, and (3) produce a final metanode whose associated landmark is g. From definition 9, $ops_\Gamma(g, \mathcal{L} \setminus \{g\}) = O \cup \{a_g\}$: all actions of the original problem are used for solving this final metanode, which is the global problem, and so completeness of LMBFS derives from completeness of the subplanner. ☐

Lazy Metanode Generation The delete landmark metanode generation (section 3.2) can generate a considerable amount of metanodes for some instances, thus inducing a slow-down during the insertion of these metanodes in the open list. To overcome this issue, we generate metanodes with $\mathrm{deleteLM}$ only when the open list is empty. When a metanode is inserted in the closed list, it is also pushed into a secondary open list. When the main open list is empty, we simply pop a metanode m from the secondary open list, and generate its children using $deleteLM(m)$. The heuristics for ordering metanodes in the secondary open list are the same as the ones used for the main open list.

3.4 Heuristics for Metanode Selection

In order to improve the algorithm effectiveness, the most promising metanode from the open list has to be selected. For doing so, a metanode generated by nextLM is always preferred over others, in order to focus search on reaching landmarks in sequence. Thus, the expansion of other (degraded) metanodes is delayed until we have no other choice, in the spirit that search can be focused using preferred operators (Richter and Helmert 2009).

Three heuristic functions have been implemented. The first ones evaluate G from the starting state of the metanode, with the well-known heuristics h^{add} (Bonet and Geffner 2001) and h^{ff} (Hoffmann and Nebel 2001). The last one, inspired by the landmark-counting heuristic of LAMA (Richter, Helmert, and Westphal 2008), uses the landmark graph Γ and counts the remaining landmarks to be reached. The metanode with the least number of remaining landmarks is chosen, enforcing a depth-first search in the graph. We will refer to this heuristic as $h^{\mathcal{L}_{left}}$.

Definition 14 ($h^{\mathcal{L}_{left}}$)**.** *For a metanode* $m = \langle s, h, \mathcal{A}, l, \rho\rangle$ *and an associated landmark graph* $\Gamma = (\mathcal{V}, \mathcal{E})$*, the heuristic* $h^{\mathcal{L}_{left}}$ *is defined by* $h^{\mathcal{L}_{left}}(m) = |\mathcal{V} \setminus \mathcal{A}|$.

4 Parallel LMBFS

A well known parallelization scheme is the principle used in HDA* (Kishimoto, Fukunaga, and Botea 2009): the idea is to distribute search nodes among the processing units (PUs) based on a hash key computed from planning states. Doing so, the list of nodes to be expanded (the open list) owned by each PU are disjoint: computations made on a given state (applicable actions, heuristic function, etc.) are performed only once, only by the PU the node "belongs" to.

Another important aspect is that communication between PUs can be performed asynchronously: a PU expands nodes from its open list, sends sons to the PUs they belong to, and periodically checks its incoming messages to incorporate new nodes into its open list. The principle has been initially conceived for optimal planning with successful results (Kishimoto, Fukunaga, and Botea 2009). It has also been applied successfully to suboptimal planning (Vidal, Vernhes, and Infantes 2011).

4.1 Algorithm

In our case, metanodes are distributed among the PUs. To do so, we define a key for metanode $m = \langle s, h, \mathcal{A}, l, \rho \rangle$ using (1) the starting state s, (2) the set of already achieved landmarks $\mathcal{A}$ and (3) the landmarks goal l. This key is hashed into a natural number modulo the number of PUs, thus giving the single PU the metanode belongs to. So the main modification is to distribute metanodes when they are generated instead of adding them to the local open list. Each PU runs a slightly modified version of the sequential search described in section 3, with its own open and closed lists.

We now give some details on the induced modifications:

- line 2: initial metanodes are only created by the master PU, and distributed among the PUs;
- line 3: the main loop condition is modified in order to wait if the open list is empty (rather than exiting); the loop is stopped only if a PU has found a solution (a message is broadcasted to all the PUs when a solution is found);
- line 4: before choosing the next metanode to expand, the algorithm checks if there are incoming metanodes, and if it is the case, it incorporates them in its open list;
- line 13: after calls to the metanode generators, the hash keys of generated metanodes are computed and metanodes are send asynchronously to the corresponding PUs.

4.2 Implementation details

In order to save memory, the current partial plan ρ (definition 7) is not stored inside the metanode in our implementation; instead, a metanode stores the partial plan ρ' (definitions 10 and 11) and a pointer to its parent metanode. So, in order to retrieve the solution plan, when a PU reaches the goal, it becomes a master PU and asks for its parent metanode partial plan to the PU owning it, concatenates it to its plan, and iterates until an initial metanode has been reached.

Another issue that came up during the adaption of HDA* parallelization scheme to LMBFS is that sometimes, the sequential algorithm gets stuck for a long time in a call to the subplanner searching for a solution to a (hard or unfeasible) subproblem. If such a case appears in one of the PUs, the parallel algorithm cannot finish until this PU ends its current search. To overcome this issue, we modified the underlying planner to check if a stop message has been received (this check is performed right before YAHSP opens a new node).

One improvement pointed out in the work with transposition-table driven scheduling for parallel IDA* (Romein et al. 1999), is to overcome the issue of the communication overhead by packing multiple nodes with the same destination into a single message. This packing strategy has also been experimented in HDA* (Kishimoto, Fukunaga, and Botea 2009), and is also implemented in our algorithm.

The lazy metanode generation cannot be implemented as-is in parallel because the metanodes generated by a PU will not necessarily be expanded by this PU; which means that emptiness of a PU primary open list does imply that metanodes generated by other operators than $\mathrm{deleteLM}$ have all been expanded. So we cannot rely on such condition to defer generation of metanodes by $\mathrm{deleteLM}$ operator.

5 Experiments

5.1 Experimental Setup

We conducted a set of experiments on a selection of benchmarks from the 3rd to the 7th International Planning Competition (IPC) within a 10 minutes CPU time limit. The experiments were all run on an Intel X5670 processor running at 2.93Ghz with 24GB of RAM. In the next figures, each plot represents an IPC problem. Only results with the $\mathrm{succDel}$ operator are reported here, as it yielded better results than $\mathrm{succCut}$. However, actually, nodes generated by nextLM are always preferred over nodes generated by these two operators, and we think that relevant heuristics for interleaving both kind of nodes might give a different picture.

Subplanner Embedded in LMBFS For subproblem resolution, we use YAHSP (Vidal 2004; 2011) for two reasons.

Firstly, we do not want to use a subplanner that also uses landmarks internally (especially if non-negligible preprocessing time is required), as our objective is to evaluate a new use of landmarks without benefiting from them in any other way.

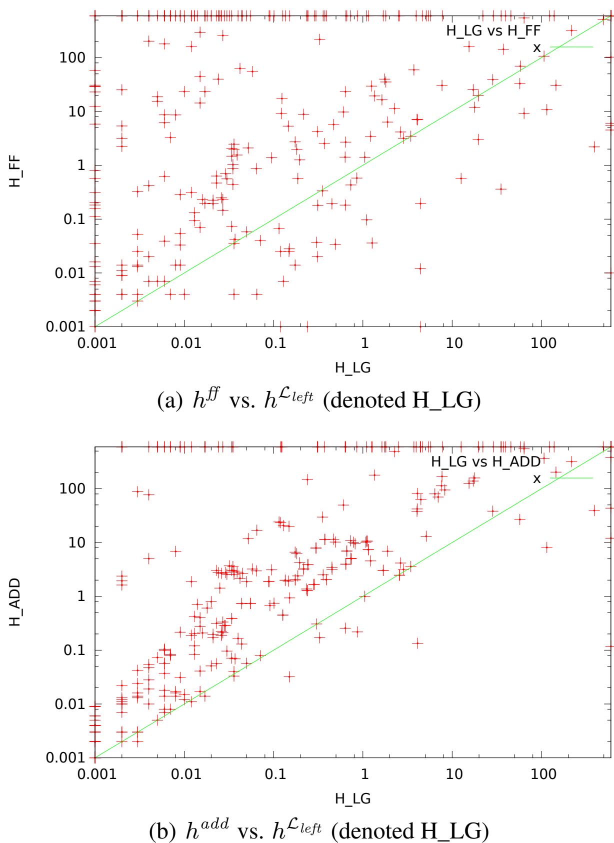
Secondly, because the successive subproblems solved during metanode expansion should be, and generally are, easy to solve with very few lookaheads computed in YAHSP. Moreover, directly embedded in the form of a C library, YAHSP does not require any preprocessing when faced with a new subproblem. It can thus generally answer very fast. It has also already been embedded with some success in another planner based on evolutionary algorithms (Bibaï et al. 2010) for solving different kinds of subproblem sequences.

Selected Domains To come up with a test suite, we ran preliminary tests in all STRIPS domains from the IPC. We selected some domains which YAHSP does not solve too easily (using few lookaheads), and also included some domains it solved easily to exhibit the possible slow-down required by the pre-computation of the landmark graph and

the splitting into smaller instances of already easy solvable problems. Thus, we selected 14 domains[1] (390 problems), which we believe could represent a classic set of domains for the IPC.

5.2 Sequential evaluation

Landmark Graph Generation For most problems, the computation time of the landmark graph is low. It takes less than $0.1s$ for 86% of the instances, and less than $1s$ for 97% of the instances (on sequential setup described below). Even if the computational time of the landmark graph on the initial state is acceptable, we consider it too long to be processed at each metanode during search. Recomputing landmarks could be more informative for search but, as LMBFS is designed for speed, we did not investigate such an option. Another reason is that adding new landmarks in the graph would break the current algorithm's completeness, which is based on emptying the initial landmark graph.



(a) h^{ff} vs. $h^{\mathcal{L}_{left}}$ (denoted H_LG)

(b) h^{add} vs. $h^{\mathcal{L}_{left}}$ (denoted H_LG)

Figure 2: Runtimes of LMBFS with several heuristics (in seconds).

Efficiency of the Different Heuristics Figure 2(a) shows a comparison of the runtime of LMBFS with $h^{\mathcal{L}_{left}}$ (x-axis) and h^{ff} (y-axis). As we can see, most of the plots are above $y = x$, meaning most instances have been solved faster using $h^{\mathcal{L}_{left}}$. Figure 2(b) shows a comparison between $h^{\mathcal{L}_{left}}$ (x-axis) and h^{add} (y-axis). The results are again in favor of $h^{\mathcal{L}_{left}}$ which outperforms h^{add} in most of the problems. Table 1 summarizes the runs performed with LMBFS using several heuristics. It shows that LMBFS with the $h^{\mathcal{L}_{left}}$ heuristic outperforms the two other (with h^{add} or h^{ff}).

Heuristic	solved $< 1s$	solved $< 10s$	solved
h^{add}	45.64%	63.85%	75.13%
h^{ff}	34.62%	45.38%	57.95%
$h^{\mathcal{L}_{left}}$	74.36%	85.90%	92.31%

Table 1: Coverage of LMBFS using different heuristics (10 minutes timeout).

Lazy Metanode Generation Table 2 compares the speed-up obtained by using the lazy metanode generation described in section 3.3. We can see that there is a strictly positive speed-up for 51.03% of the problems, and a speed-up superior to 2 for 23.74% of them. Even if there is a noticeable slow-down for 4.36% of the problems (heuristically equivalent nodes may be ordered differently in the main and secondary open lists due to implementation details), it still is a nice improvement for the overall test suite.

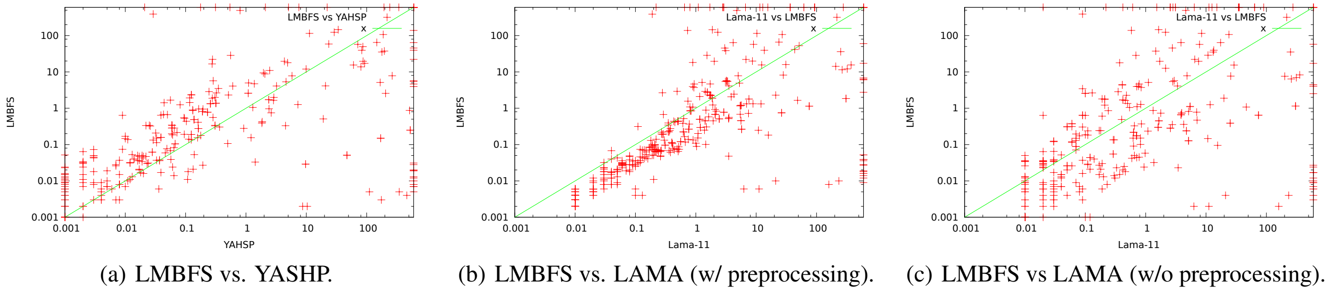
		Instances where the speed-up/slow-down is		
	Average	> 1	> 1.05	> 2
Speed-up	8.11	51.03%	36.41%	22.74%
Slow-down	2.24	14.10%	4.36%	1.81%

Table 2: Speed-up of lazy metanode generation.

LMBFS versus sub-planner (YAHSP) Table 3 summarizes the coverage of runs performed with LMBFS using the $h^{\mathcal{L}_{left}}$ heuristic in comparison with YAHSP and some state-of-the-art planners, also shown as a curve in function of the timeout in Figure 4. The comparison between LMBFS and YAHSP is also depicted in Figure 3(a). It shows that many problems are solved within 1s and most of the problems quickly solved by YAHSP (under 0.1s) are solved by LMBFS nearly as fast. On harder instances we can also see that LMBFS shows its benefits in terms of running time. Finally the total number of solved instances is slightly in favor of LMBFS.

A major drawback that has been pointed out for the DSC algorithm and the SteLLa planner is the length of computed plans which can be significantly higher. Compared to the plans computed by YAHSP, the average plan length computed by LMBFS is 8% shorter. In 43% instances LMBFS computes strictly shorter plan than YAHSP and in 19% instances the plans are strictly longer.

[1] ipc3-driverlog, ipc3-freecell, ipc3-satellite, ipc4-pipesworld_-tankage, ipc4-psr_small, ipc[56]-openstacks, ipc5-pathways, ipc[67]-transport, ipc[67]-scanalyzer, ipc7-barman, ipc7-floortile



(a) LMBFS vs. YASHP. (b) LMBFS vs. LAMA (w/ preprocessing). (c) LMBFS vs LAMA (w/o preprocessing).

Figure 3: Runtimes of LMBFS with $h^{\mathcal{L}_{left}}$ versus YASHP and LAMA-11 (in seconds).

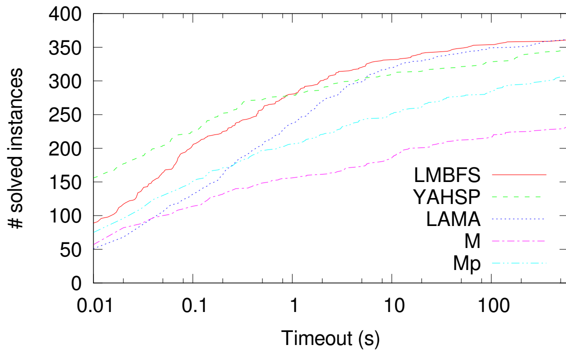


Figure 4: Coverage of LMBFS with $h^{\mathcal{L}_{left}}$ versus state-of-the-art planners as a function of the timeout (in seconds).

Planner	solved $< 1s$	solved $< 10s$	solved
with preprocessing time			
LMBFS	71.79%	84.87%	92.31%
YAHSP	71.28%	79.74%	88.97%
LAMA-11	60.51%	81.79%	92.05%
M	40.00%	47.44%	58.72%
Mp	52.82%	64.36%	78.46%
without preprocessing time			
LMBFS	74.36%	85.90%	92.31%
LAMA-11	69.74%	82.82%	92.56%

Table 3: Coverage of LMBFS with $h^{\mathcal{L}_{left}}$ versus state-of-the-art planners (10 minutes timeout), with and without preprocessing time for LMBFS and LAMA-11.

LMBFS versus State-of-the-Art Planners Table 3 and Figure 4 compare LMBFS to the LAMA-11 planner (based on landmark analysis but in a different way), as well as to the SAT-based planners M and Mp (**?**).

Figures 3(b) and 3(c) are scatter plots comparing the runtimes of LMBFS versus LAMA-11. Figure 3(b) shows runtimes including preprocessing within the 10 minutes timeout, and Figure 3(c) without taking into account the preprocessing time, as LAMA-11 can spend a lot of time during this stage.

These evaluations show that on our selection of domains, LMBFS is competitive with the state-of-the-art. It clearly outperforms M, Mp and YAHSP (for problems that require at least 1 second). It also outperforms LAMA-11, although the overall performance for a 10 minutes timeout depends on whether or not the preprocessing time is included.

LMBFS versus YAHSP (quality) A major drawback that has been pointed out for the DSC and STeLLA algorithms is the length of computed plans which can be significantly higher. Compared to the plans computed by YAHSP, the average plan length computed by LMBFS is 8% shorter. In 43% instances LMBFS computes strictly shorter plan than YAHSP and in 19% instances the plans are strictly longer.

5.3 Parallel version evaluation

To implement the parallel version, we used the Message Passing Interface (MPI) to spawn the different processes and pass messages between them. We conducted a set of experiments on a cluster of 4 servers. Each one embeds two 6-core CPUs (Intel X5670) running at 2.93Ghz with 24GB of RAM. They are connected via a gigabit network.

The parallel version of LMBFS in a HDA* fashion is denoted MPI $a \times b$ where a is the number of servers used and b is the number of processes per server.

For comparison purposes, a parallel non-cooperative version has also been developed: it consists in several sequential versions of LMBFS with no communication between the processes. The only difference in these sequential versions is that the order of the nodes which have the same heuristic value is randomized (in the open list) for both LMBFS and the underlying planner YAHSP. This random version is simply denoted Random and is run with 48 processes in parallel (with different seeds for the random number generator). For this last version, we consider only the statistics of the process which found first the solution (if any).

Number of expanded nodes We can first take a look at the growth of the number of expanded metanode per instances: Figure 5. The number of expanded nodes seems to scale well when the number of PUs increase. The random version mostly expands the same number of nodes than the classic sequential version (we consider only one process).

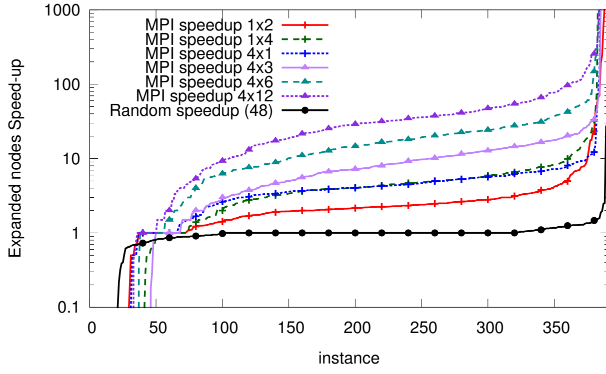
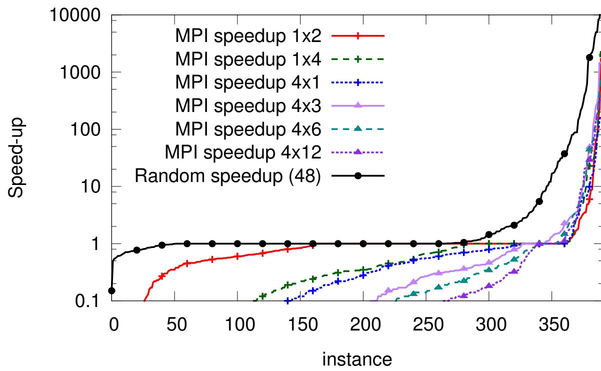
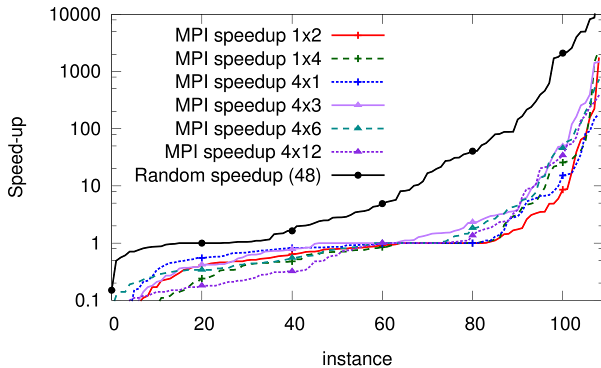


Figure 5: Expanded node speedup of parallel vs. sequential version (instances sorted by increasing speedup)



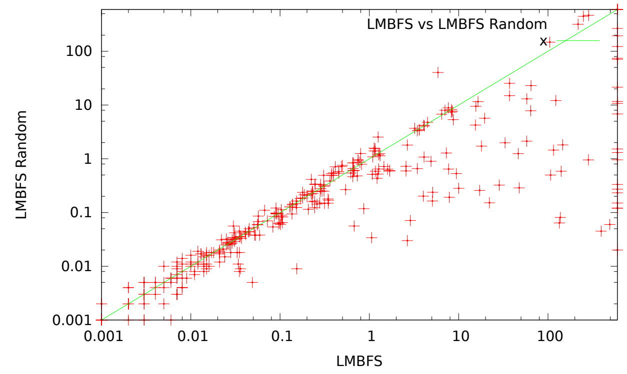
(a) Wall-clock time speedup of parallel vs sequential version



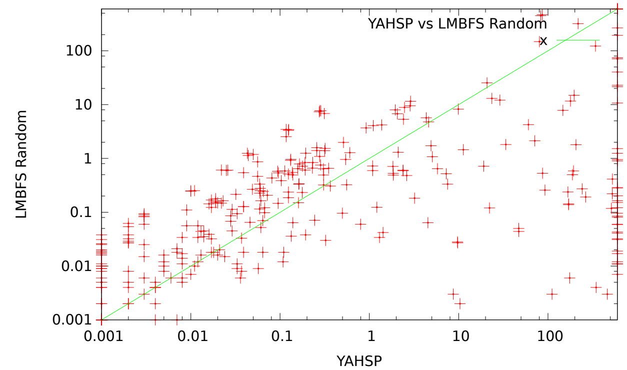
(b) Wall-clock time speedup of parallel vs sequential version (where the sequential version takes more than $1s$)

Figure 6: Solving time speedups (sorted by speedup)

Solving performance Figure 6(a) shows however, that even if the number of expanded nodes has increased a lot, the MPI algorithm does not scale up at all; it performs even worse for some instances. MPI takes some pre-processing time to boot up (around one second); but even if we just select instances that are solved in more than one second (Figure 6(b)), we can see that in most cases parallel versions take the same time as the sequential version (or even more, as we



(a) LMBFS runtimes: sequential version vs. Random (48)



(b) Runtimes of YAHSP and LMBFS Random (48)

Figure 7: Results for Random version (in seconds)

increase the number of processes). Finally, there are linear or super-linear speedups for only a small subset of instances.

The coverage is not improved either, but it increases with the number of processes: it goes from 87.69% for the MPI 1×4 version to 92.05% for the MPI 4×12 version (the only exception is the MPI 1×2 version with 90.26%). Clearly, the breadth induced by this parallelization is not efficient.

Open List Randomization On previous figures, the random version is always better than the MPI version. As shown in Figures 7(a) and 7(b), the random version is also better than both sequential LMBFS and YAHSP, especially for hard instances. It also increases the coverage to 97.69%. Clearly, a promising lead to improve LMBFS is to find a way to bend the large plateau induced by the $h^{\mathcal{L}_{left}}$ heuristic.

Discussion This parallel algorithm expands many uninteresting metanodes generated by $\mathrm{deleteLM}$. Associated problems are usually hard to solve (because this generation of subproblem is less guided by the landmark graph), and can get a subplanner stuck whereas the PU should ideally explore more interesting metanodes received in the meantime. In other words, these expansions seem to lead the algorithm into uninteresting or dead-end branches of the search graph, which asks the question of the quality of the heuristic.

This also can be seen if recalling that LMBFS is based on exploration of landmarks orders. LMBFS tries, in a depth first search way, all possible total orderings before emptying the landmark graph (this is particularly enforced by the $h^{\mathcal{L}_{left}}$ heuristic and the lazy generation of section 3.3). But in the parallel version, this is not the case, as too early exploration of metanodes expanded by $\mathrm{deleteLM}$ occurs.

6 Conclusion and Future Works

This paper presents several contributions towards a new landmark-based planning algorithm. First, we propose a sound framework for a (meta)search based on the order of landmarks, given a landmark graph. We formalize the link between so-called metanodes and subproblems of the original planning problem, including restrictions on the allowed actions themselves. We give several operators that allow to explore different orders for using landmarks as subgoals, including skipping some. We also propose a first approach for evaluating heuristic values of such metanodes, or equivalently giving priorities to subproblems. We put everything together in a (deferred) best-first search algorithm, leading to a complete algorithm. We also propose a first parallelization scheme based on asynchronous distribution of open nodes among processing units. Last but not least, we implemented it and give some promising results.

From now on, several leads will be followed.

A key point for performance is the heuristic evaluation of metanodes, linked to the operators used for generation. For instance, nodes generated with nextLM are always expanded before other metanodes, which is not necessarily the best solution. Furthermore, the comparison of one LMBFS instance against several non-cooperative randomized instances showed that there is room for heuristic improvement. A first lead to obtain a better heuristic would be to also take into account the landmark subgoal but preliminary experiments showed us that this will not be straightforward.

Another point is the operators used. While $\mathrm{deleteLM}$ is very general, $\mathrm{cutParent}$ can be seen as a special case (a shortcut for a given sequence of $\mathrm{deleteLM}$, or said differently, a lookahead in the landmark graph itself), and other special cases may be very useful.

Another next step will focus on smarter parallelization schemes. As experiments showed, while we are able to scale up the number of explored nodes, this does not lead to finding a solution in less time. It seems that the processing units are "saturated" very quickly with hard problems coming from $\mathrm{deleteLM}$ (especially because lazy generation is not implemented in parallel algorithm), and so, with this simple parallelization scheme, the whole spirit of LMBFS, which is to try to follow as closely as possible the landmark graph while being complete, is no more enforced. There are some possibilities to do a lazy-like generation for the parallel version, but it would be intrinsically different from the lazy generation of sequential version. Another interesting lead is to select a few total orders at the beginning of the search, and try to find plans following these orders. If no solution can be found, we might be able to extract some data from these searches to create other interesting total orders, and so on.

References

Bibaï, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2010. An evolutionary metaheuristic based on state decomposition for domain-independent satisficing planning. In *Proc. of ICAPS*, 18–25.

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

Fikes, R. E., and Nilsson, N. J. 1972. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3-4):189–208.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning, theory and practice*. Morgan-Kaufmann.

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

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

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *Journal of Artificial Intelligence Research* 22:215–278.

Keyder, E.; Richter, S.; and Helmert, M. 2010. Sound and complete landmarks for and/or graphs. In *Proc. of ECAI*, 335–340.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2009. Scalable, parallel best-first search for optimal sequential planning. In *Proc. of ICAPS*.

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

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proc. of the 23rd AAAI Conference on Artificial Intelligence*, 975–982.

Romein, J. W.; Plaat, A.; Bal, H. E.; and Schaeffer, J. 1999. Transposition table driven work scheduling in distributed search. In *Proc. AAAI*.

Sebastia, L.; Onaindia, E.; and Marzal, E. 2006. Decomposition of planning problems. *AI Communications* 19:49–81.

Vidal, V.; Vernhes, S.; and Infantes, G. 2011. Parallel AI planning on the SCC. In *Proc. of the 4th Symposium of the Many-core Applications Research Community (MARC)*.

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

Vidal, V. 2011. YAHSP2: Keep it simple, stupid. In *Proc. of IPC'11*.

Zhu, L., and Givan, R. 2003. Landmark extraction via planning graph propagation. In *ICAPS Doctoral Consortium*, 156–160.