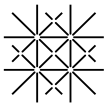

University of Basel

# Backbones for $h^1$ Landmarks

Bachelor's Thesis


Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence Research Group
https://ai.dmi.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. David Jakob Speck,
Travis Rivera Petit

Elagkian Rajendram
elagkian.rajendram@unibas.ch
20-936-969


31 July 2026

# Acknowledgments

First and foremost, I would like to thank my supervisors Dr. David Speck and Travis Rivera Petit for the weekly meetings and their guidance throughout the entire project. The meetings were a great source of insight and I learned a lot from them. I would also like to thank Prof. Dr. Malte Helmert for the opportunity to write my bachelor thesis in his research group.
Calculations were performed at sciCORE (http://scicore.unibas.ch/) scientific computing center at University of Basel.

# Abstract

CadiBack is a tool that extracts the backbone of a propositional formula, the set of literals true in every model, using few SAT solver calls. In this thesis we transfer its ideas to classical planning in order to extract landmarks efficiently. We derive the algorithm CadiLand, which computes the $h^1$ landmarks of a task, the landmarks of its delete relaxation, by testing each candidate fact with one relaxed reachability check and reusing the relaxed plans it obtains to rule out further candidates. We implement CadiLand as a Python prototype and natively inside the Fast Downward planning system and compare its modes, from naive testing through model rotation to fixed, adaptive and density adaptive chunking, on the optimal STRIPS benchmarks. Which mode is cheapest depends on the landmark density of the task, the fraction of candidates that turn out to be landmarks. Chunking helps only where landmarks are rare, below a density of about 0.3, and costs more than naive testing on dense tasks, so the chunking idea of CadiBack does not transfer to a suite on which most tasks are dense. Model rotation, in contrast, never costs more than naive testing and is the best mode on average. Adapting the chunk size to the density during the run comes close and is a good compromise when the density is not known in advance.

# Table of Contents

# 1 Introduction

Classical planning searches for a sequence of actions that leads from an initial state to a goal. The search space grows quickly with the size of the task, so planners rely on information about the problem that constrains which states are worth exploring. A landmark is a fact that holds at some point along every plan, so any solution must pass through it [8]. Knowing the landmarks of a task therefore constrains every plan at once and landmarks have become a common basis for heuristics and search guidance in modern planners. The difficulty is that landmarks are expensive to find in general, so in practice one settles for a subset that is sound but incomplete: every fact it reports really is a landmark, though some are missed.
The same shape of problem appears in propositional logic. The backbone of a satisfiable formula is the set of literals that are true in every model. The tool CadiBack [2] extracts the complete backbone efficiently, with far fewer SAT solver calls than testing each literal on its own. Landmarks are the planning analogue of the backbone: both are the facts, respectively literals, that every solution is forced to satisfy.
The goal of this thesis is to transfer the ideas of CadiBack to planning and to see whether the same techniques let us extract landmarks more efficiently. We work with the landmarks of the delete relaxation, the $h^1$ landmarks, because they are easier to compute than the landmarks of the full task. Every $h^1$ landmark is also a landmark of the original task, so anything we find is a true landmark, though some may be missed. To test the approach we implement the resulting algorithm twice, as a prototype and as an integration into the Fast Downward planning system [6], and compare several of its variants to find which extract the landmarks most efficiently.

# 2
# Background

Our work combines ideas from SAT solving and AI planning. This chapter introduces the concepts from both fields that the later chapters rely on. We assume familiarity with propositional logic and basic graph search and restrict ourselves to the concepts that are required later on. We first introduce CNF formulas and satisfiability (Section 2.1) and then backbone extraction together with the CadiBack [2] algorithm (Section 2.2). We then turn to classical planning (Section 2.3) and to landmarks (Section 2.4), the planning analogue of backbones on which this thesis centres.

## 2.1 CNF and SAT

We assume a finite set of Boolean variables $X = \{x_1, \ldots, x_n\}$. The definitions in this section are standard and follow Kleine Büning and Lettmann [9]. A *literal* is a variable $x_i \in X$ or its negation $\neg x_i$. A *clause* is a disjunction of one or more literals, for example $(x_1 \vee x_2 \vee \neg x_3)$. A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses, that is, $\varphi = \bigwedge_{i=1}^{m} C_i$ where each clause $C_i = \bigvee_j \ell_{ij}$ is a disjunction of literals.

An *assignment* maps each variable in $X$ to a Boolean value, either true or false. A clause is *satisfied* by an assignment if at least one of its literals evaluates to true, and a formula is satisfied if all of its clauses are satisfied. An assignment that satisfies $\varphi$ is called a *model* of $\varphi$. A formula is *satisfiable* if it has at least one model, and *unsatisfiable* otherwise.

The *propositional satisfiability problem* (SAT) asks whether a given CNF formula $\varphi$ is satisfiable. A *SAT solver* takes a CNF formula as input and either reports that it is satisfiable, returning a model, or reports that it is unsatisfiable. SAT was the first problem shown to be NP-complete, so no algorithm is known that decides it efficiently in the worst case, and a single solver call can be costly. Reducing how often the solver is called is therefore worthwhile, and it is the problem that CadiBack [2], which we turn to next, is built to solve.

## 2.2 Backbone Extraction and CadiBack

Given a satisfiable CNF formula $\varphi$, the *backbone* of $\varphi$ is the set of literals that are true in every model of $\varphi$. We denote it by $B$, and call a literal $\ell \in B$ a *backbone literal*. For example, the formula $(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$ has the two models $\{x_1 \mapsto T, x_2 \mapsto F\}$ and $\{x_1 \mapsto T, x_2 \mapsto T\}$, so its backbone is $B = \{x_1\}$.

The most direct way to test whether a literal $\ell$ is a backbone literal is to ask the solver whether $\varphi \wedge \neg\ell$ is satisfiable. If this query is unsatisfiable, then $\ell$ is true in every model of $\varphi$ and is therefore a backbone literal. The backbone extraction algorithm this thesis builds on is called CadiBack [2]. Testing every literal this way requires one solver call per literal, which is expensive for formulas with many variables. Reducing this number of calls is the problem that CadiBack addresses.

CadiBack extracts the complete backbone of a formula while issuing far fewer solver calls. It is built on the SAT solver CaDiCaL and uses its incremental interface, which keeps the formula $\varphi$ loaded in the solver from one query to the next instead of reloading it each time [3]. Each test is then posed as an assumption, a condition that holds for that one query and is dropped afterwards, rather than as a fresh solve of $\varphi$ from scratch. On top of this, CadiBack applies three ideas that classify literals with few or no additional calls: fixed literals, the disagreement condition, and chunking.

A unit clause is a clause consisting of a single literal. If $\ell$ occurs as a unit clause in $\varphi$, then every model must satisfy $\ell$, so $\ell$ is a backbone literal, and no solver call is needed to see this. We call such literals fixed. The solver derives further unit clauses as it runs, so more literals become fixed over the course of the extraction.

The *disagreement condition* exploits the models the solver returns. If two models $\sigma$ and $\sigma'$ assign different values to a literal $\ell$, then neither $\ell$ nor $\neg\ell$ is a backbone literal, since a backbone literal must take the same value in every model. Every returned model therefore prunes all literals on which it disagrees with a previously seen model.

A cheap special case is *model rotation* (also called flipping). A literal $\ell$ is *flippable* in a model $\sigma$ if changing only the value of $\ell$ yields another model $\sigma'$. Since $\ell$ then takes different values in $\sigma$ and $\sigma'$, neither $\ell$ nor $\neg\ell$ is a backbone literal. Whether a literal is flippable can be checked directly on the clauses without a solver call, so each returned model can rule out many candidates for free.

**Example 2.1.** *Consider $\varphi = (x_1 \vee x_2) \wedge (x_1 \vee x_3)$ with the model $\sigma = \{x_1 \mapsto T, x_2 \mapsto T, x_3 \mapsto T\}$. Flipping $x_2$ yields $\sigma' = \{x_1 \mapsto T, x_2 \mapsto F, x_3 \mapsto T\}$, which still satisfies $\varphi$. Hence $x_2$ is flippable, and neither $x_2$ nor $\neg x_2$ is a backbone literal.*

Finally, *chunking* tests several candidates with a single solver call. Given a chunk of $k$ candidate literals $\ell_1, \ldots, \ell_k$, CadiBack adds one clause $\rho = (\neg\ell_1 \vee \cdots \vee \neg\ell_k)$, which asserts that at least one candidate is false, and asks whether $\varphi \wedge \rho$ is satisfiable. If the query is unsatisfiable, none of the candidates can be made false, so all $k$ are backbone literals confirmed by a single call. If it is satisfiable, the returned model falsifies at least one candidate, which is removed by the disagreement condition, while model rotation may prune further candidates from the same model.

The chunk size is chosen adaptively. Starting from $k = 1$, CadiBack multiplies the size by a growth rate $K$ after each unsatisfiable query, which confirms a whole chunk of backbone literals at once, and resets it to 1 after each satisfiable query. This strategy is efficient whenever backbone literals occur in runs: chunk sizes grow with each confirmed chunk, so later calls confirm more literals at once, until a satisfiable query ends the run and the growth restarts. The default configuration of CadiBack does not limit the chunk size at all, which corresponds to $K = \infty$ and lets a single query cover every remaining candidate. This default is also the strongest setting reported, solving 732 instances against 702 for a growth rate of $K = 10$ and 692 for single literal chunks [2].

## 2.3 Classical Planning

We consider classical planning in the SAS$^+$ formalism [1], the representation used internally by Fast Downward. A planning task is a tuple $\Pi = \langle V, O, s_0, s_\star \rangle$, where $V$ is a finite set of state variables and $O$ a finite set of operators. Each variable $v \in V$ has a finite domain $\mathrm{dom}(v)$. A *fact* $\langle v, \vartheta \rangle$ assigns the value $\vartheta \in \mathrm{dom}(v)$ to the variable $v$. A *state* $s$ assigns a value $s[v] \in \mathrm{dom}(v)$ to every variable at once and we identify it with its set of facts. A *partial assignment* fixes values for only some of the variables. For a partial assignment $p$ and a state $s$, we write $p \subseteq s$ if $s$ agrees with $p$ on every variable that $p$ fixes, that is, if every fact of $p$ is also a fact of $s$. The initial state $s_0$ is a full state, and the goal $s_\star$ is a partial assignment.

Each operator $o \in O$ is a pair $o = \langle \mathrm{pre}_o, \mathrm{eff}_o \rangle$ of partial states, its *precondition* and its *effect*. The operator is *applicable* in a state $s$ if $\mathrm{pre}_o \subseteq s$. Applying $o$ changes the values of the variables in $\mathrm{eff}_o$ to the values given there and leaves all other variables unchanged. We write $s[o]$ for the resulting successor state, defined when $o$ is applicable in $s$. We collect the facts that $o$ establishes in $\mathrm{add}(o) = \{ \langle v, \vartheta \rangle \mid \mathrm{eff}_o[v] = \vartheta \}$; these are simply the facts of the effect, and they are what we later forbid to test whether a fact is a landmark.

**Example 2.2.** *Consider a single variable pos with domain* $\mathrm{dom}(pos) = \{A, B, C\}$*, describing the position of a truck. In a state $s$ with $s[pos] = A$, the fact $\langle pos, A \rangle$ holds. Let $o$ be the operator with precondition* $\mathrm{pre}_o = \{\langle pos, A \rangle\}$ *and effect* $\mathrm{eff}_o = \{\langle pos, B \rangle\}$*, so* $\mathrm{add}(o) = \{\langle pos, B \rangle\}$*. Since* $\mathrm{pre}_o \subseteq s$*, the operator is applicable, and applying it sets pos to $B$. In the successor state the fact $\langle pos, B \rangle$ now holds, while $\langle pos, A \rangle$ no longer does: the truck cannot be at two places at once, so assigning the new value $B$ removes the old fact $\langle pos, A \rangle$ automatically.*

A *plan* is a sequence of operators applicable from $s_0$ that leads to a state in which the goal $s_\star$ holds; such a plan *solves* $\Pi$. A *planner* is a tool that searches for a plan. The states reachable from $s_0$, together with the operators that lead from one to another, form the *state space* of the task: a directed graph whose nodes are states and whose edges are the applicable operators. A plan is then a path in this graph from $s_0$ to a state satisfying the goal.

**Delete relaxation.** Recall that in SAS$^+$ assigning a new value to a variable implicitly removes its previous value; this removal is the *delete effect* of an operator. The *delete relaxation* $\Pi^+$ of a task $\Pi$ is obtained by ignoring these delete effects, so that reassigning a variable no longer removes its old value. In the relaxed task a variable may hold several values at once, so once a fact holds, no operator can make it false again, and facts only ever accumulate. The set of *relaxed reachable* facts is the least set that contains all facts of $s_0$ and, for every operator whose precondition is relaxed reachable, also the facts in $\mathrm{add}(o)$. The goal is *relaxed reachable* if all of its facts are. Relaxed reachability can thus be decided efficiently, which makes it a suitable basis for the landmark tests developed in Chapter 3.

**Example 2.3.** *Continuing Example 2.2, take the delete relaxation of the truck task. Applying $o$ still adds $\langle pos, B\rangle$, but now the old fact $\langle pos, A\rangle$ is no longer removed. In the relaxed successor state both $\langle pos, A\rangle$ and $\langle pos, B\rangle$ hold at once: the truck is treated as being at $A$ and at $B$ simultaneously.*

## 2.4 Landmarks in AI Planning

A *landmark* [8] is a fact that holds at some point along every plan. Formally, a fact $f$ is a landmark of a task $\Pi$ if every plan of $\Pi$ passes through a state in which $f$ holds. Two kinds of landmarks are trivial: every fact of the initial state $s_0$ is a landmark, since every plan starts in $s_0$, and every fact of the goal $s_\star$ is a landmark, since every plan ends in a state in which $s_\star$ holds. Neither kind requires any computation, so the interesting landmarks are the remaining facts, those a plan must establish on its way from $s_0$ to the goal.

**Example 2.4.** *Extend the truck of Example 2.2 with the goal $s_\star = \{\langle pos, C\rangle\}$, and suppose the only movement operators lead from $A$ to $B$ and from $B$ to $C$. Starting at $A$, every plan must move the truck to $B$ before it can reach $C$, so $\langle pos, B\rangle$ holds at some point along* every *plan and is a landmark. The facts $\langle pos, A\rangle$ and $\langle pos, C\rangle$ are landmarks as well, but trivially so, being the initial and the goal fact. If we additionally allowed a direct move from $A$ to $C$, then $\langle pos, B\rangle$ would cease to be a landmark, since the plan taking the direct route never makes it true.*

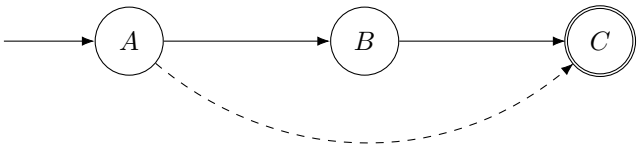


Figure 2.1: The truck task of Example 2.4. Solid arrows are the movement operators, the dashed arrow the optional direct move $A \to C$.

We focus on $h^1$ *landmarks*, the landmarks of the delete relaxation $\Pi^+$. The name refers to the $h^1$ heuristic [5], which estimates reaching costs while ignoring all interactions between facts and assigns $\infty$ to facts it cannot derive at all. We only ever use this distinction between finite and infinite: a fact is relaxed reachable if and only if $h^1$ assigns it a finite value, and the goal is relaxed reachable exactly when $h^1$ of the goal is finite. Landmarks defined over relaxed reachability are thus landmarks with respect to $h^1$.

Two properties make them well suited to our setting. First, they are cheap to extract: each landmark test reduces to a relaxed reachability check, which is polynomial, rather than to a solvability check on the unrelaxed task. Second, they are sound. Every plan of $\Pi$ is also a plan of $\Pi^+$, so any fact that holds in every relaxed plan also holds in every plan of $\Pi$; hence every landmark of $\Pi^+$ is a landmark of $\Pi$ [8]. The converse fails, since delete relaxation can miss landmarks that arise only from deleting facts. The delete relaxation landmarks therefore form a sound but incomplete subset of the task's landmarks, a lower bound on its landmark set.
This gives a direct way to test a single fact. A fact $f$ is a landmark of $\Pi^+$ if and only if the goal becomes relaxed unreachable once every operator that adds $f$ is removed: if the goal can still be reached, some relaxed plan avoids $f$, so $f$ is not a landmark; if it cannot, every relaxed plan uses $f$. Each such test is one relaxed reachability check.
Landmarks are the planning analogue of backbones: both are the facts, respectively literals, that are forced to hold in every solution. Chapter 3 turns this analogy into an algorithm by transferring CadiBack's backbone extraction techniques to landmark computation.

# 3

# The CadiLand Algorithm

CadiLand is derived from CadiBack: it computes landmarks of a planning task the way CadiBack computes the backbone of a formula, by repeatedly testing whether a single candidate can be avoided and by reusing the solutions it obtains along the way. This chapter shows how the algorithm is built, where it follows CadiBack closely and where the planning setting forces it to differ.

## 3.1 From Backbones to Landmarks

The translation between the two settings is close. A backbone literal becomes a fact landmark, and each SAT call becomes a relaxed reachability check: instead of asking whether a literal can be set to false in some model, we ask whether the goal can still be reached when a fact is forbidden, that is, when all operators adding it are removed (Section 2.3).

Both algorithms work on a set of *candidates*, the facts or literals that are still in question as possible landmarks or backbone literals. A candidate is resolved once the algorithm has shown either that it must hold in every solution, in which case it is confirmed, or that some solution avoids it, in which case it is discarded. The candidate literals of CadiBack correspond to the candidate facts of CadiLand, and resolving a candidate means the same thing in both.

In both settings the interesting information comes from the tests that fail to confirm a candidate. If the test $\varphi \wedge \neg\ell$ is satisfiable, the returned model assigns $\ell$ the value false, so $\ell$ is not a backbone literal. The model also settles other candidates: any literal it assigns differently than an earlier model is not a backbone literal either, since a backbone literal must take the same value in every model. A single call can therefore eliminate many candidates at once. CadiLand uses the returned relaxed plan for the same purpose. If the goal is still reachable without the forbidden fact, the check returns a relaxed plan $\pi'$ that reaches the goal. Every candidate that $\pi'$ does not produce is eliminated as well, since $\pi'$ reaches the goal without it. The algorithm therefore keeps only the candidates that occur in $\pi'$, replacing the open set by its intersection with facts($\pi'$). This is the planning form of model rotation.

**Example 3.1.** *Suppose the candidate pool is $\Lambda = \{A, B, C, D\}$ and the algorithm tests $A$. The goal turns out to be reachable without $A$, and the returned relaxed plan $\pi'$ produces the facts $C$ and $D$ but neither $A$ nor $B$. The single check thus rules out $A$ as intended, but also $B$, since $\pi'$ reaches the goal without producing it. The pool shrinks to $\Lambda = \{C, D\}$ at the cost of one check, whereas testing each fact on its own would have needed two.*

One step of the transfer does not carry over cleanly. CadiBack tests a chunk of $k$ candidates by adding the single disjunctive clause $\rho = (\neg\ell_1 \vee \cdots \vee \neg\ell_k)$. Since a disjunction is satisfied as soon as one of its literals is, any model of $\varphi \wedge \rho$ makes at least one of the $k$ candidates false, and such a candidate is then ruled out as a backbone literal. The query therefore asks whether any of the $k$ candidates can be refuted at all. If the answer is no, that is, if the query is unsatisfiable, then none of them can ever be false, so all $k$ are backbone literals and one call has settled the whole chunk.
CadiLand cannot reproduce this. A planning goal is a conjunction of facts, so there is no way to express the disjunctive condition "at least one of these facts is avoidable" as a single reachability query. All CadiLand can do is forbid the whole chunk at once, by removing every operator that adds any of its facts, and ask whether the goal is still reachable. If it is, no fact in the chunk is a landmark and the chunk is cleared by one call, as before. But if it is not, all we learn is that no relaxed plan avoids every fact of $\Gamma$ at once, which need not mean that one of them is a landmark on its own. We therefore retest the members individually: the one joint check on the chunk plus up to $k$ single checks, one per candidate in it, for a total of up to $1 + k$ calls where a single check per candidate would have cost $k$. The favourable and unfavourable outcomes have therefore swapped roles relative to CadiBack: here it is the *solvable* outcome that resolves a whole chunk at once, while the unsolvable outcome is the expensive one.

**Example 3.2.** *Suppose the chunk is $\Gamma = \{A, B, C\}$ and that of these only $B$ is a landmark. Forbidding all three at once makes the goal unreachable, since every relaxed plan needs $B$. The check therefore reports failure, but its answer is the same as it would have been if $A$ and $C$ were landmarks too: a single unreachable outcome cannot distinguish one landmark in the chunk from three. The algorithm learns only that no relaxed plan avoids $A$, $B$ and $C$ at once, and must test the three separately to find out that $B$ is the one, so the chunk costs four checks instead of the three that individual testing would have needed.*

Table 3.1: The mapping between CadiBack and CadiLand.

| CadiBack (SAT) | CadiLand (Planning) |
|---|---|
| backbone literal | fact landmark |
| SAT call on $\varphi$ | relaxed reachability check on $\Pi^+$ |
| candidate: literal of a model | candidate: fact of a relaxed plan |
| falsify literal $\ell$ | reach the goal without producing fact $f$ |
| model rotation (reuse model) | intersect with facts($\pi'$) |
| chunk clause $\rho = \bigvee \neg\ell$ (disjunction) | forbid all chunk facts jointly (conjunction) |

## 3.2 The Algorithm and its Modes

We now give a pseudo code for CadiLand. The modes we study differ only in the chunk size and in whether the returned plans are reused, so we present them as one algorithm and obtain each by fixing its parameters and the marked variant. Two parameters control the chunk size: $c$ is its initial value and $K$ the rate of growth. During the run the current size $k$ starts at $c$ and is either multiplied by $K$ or reset to $c$, so $c$ and $K$ stay fixed while $k$ varies. Throughout we use the planning notation of Chapter 2 and add three shorthands specific to the algorithm. For a set of facts $\Gamma$, we write

$$O_\Gamma = \{\, o \in O \mid \mathit{add}(o) \cap \Gamma \neq \emptyset \,\}$$

for the operators that establish some fact in $\Gamma$. The call $\textsc{Solvable}(\Pi^+, \Gamma)$ asks whether the goal is reachable in the delete relaxation $\Pi^+$ using only the operators $O \setminus O_\Gamma$, that is, whether the goal can be reached without producing any fact of $\Gamma$. It returns a relaxed plan reaching the goal if there is one, and $\bot$ otherwise. Finally, $\mathit{facts}(\pi) = \bigcup_{o \in \pi} \mathit{add}(o)$ denotes the facts made true by the operators of a relaxed plan $\pi$.

**Algorithm 1** Extracting the landmarks of a planning task $\Pi$.

```
// Use c = 1, K = 1 for one by one,
// c = m, K = 1 for fixed chunking of size m, and
// c = 1, K > 1 for adaptive chunking.
// The inherited rule exchanges the assignments marked *.
function CadiLand(task Π = ⟨V, O, s_0, s_⋆⟩, chunk size c = 1, chunk rate K = 1)
    π ← Solvable(Π^+, ∅)
    assert π ≠ ⊥                                              // Π^+ solvable
    Λ ← facts(π) \ (s_0 ∪ s_⋆)                                // candidates
    k ← c, B ← s_0 ∪ s_⋆                                      // trivial landmarks
    while Λ ≠ ∅ do
        Γ ← pick k' facts from Λ                              // chunk with k' = min(k, |Λ|)
        π' ← Solvable(Π^+, Γ)                                 // forbid all facts in Γ
        if π' ≠ ⊥ then                                        // goal still reachable, no landmark in Γ
            Λ ← Λ ∩ facts(π')
            k ← K · k                                         *
        else if |Γ| = 1 then                                  // the single candidate is a landmark
            B ← B ∪ Γ, Λ ← Λ \ Γ
            k ← c                                             *
        else                                                  // no plan avoids all of Γ
            for all f ∈ Γ with f ∈ Λ do                       // localise
                π' ← Solvable(Π^+, {f})
                if π' = ⊥ then                                // f is a landmark
                    B ← B ∪ {f}, Λ ← Λ \ {f}
                    k ← c                                     *
                else
                    Λ ← Λ ∩ facts(π')
                    k ← K · k                                 *
                end if
            end for
        end if
    end while
    return B
end function
```

The algorithm maintains a set $\Lambda$ of open candidates and a set $B$ of confirmed landmarks. Both are initialised in lines 8 and 9: the facts of the initial state and of the goal are landmarks by definition (Section 2.4), so they enter $B$ directly and are excluded from $\Lambda$. In each iteration the algorithm forbids a chunk of candidates and checks whether the goal is still reachable, and it shrinks $\Lambda$ with every answer it gets.
The main loop distinguishes three cases.

1. The goal is still reachable after forbidding the chunk. Then none of its facts is a landmark, and the algorithm keeps only those candidates that the returned plan produces. This discards the chunk together with every other candidate the plan avoids; without model rotation it simply discards the chunk.

2. The goal is unreachable and the chunk has a single member. Then that member is a landmark and is moved to $B$.

3. The goal is unreachable and the chunk is larger. Then no relaxed plan avoids all of its facts at once, but the check does not say which of them is responsible. The algorithm therefore tests each member on its own until all of them are resolved, skipping those that an intermediate plan has already removed from $\Lambda$.

All modes return the same set of landmarks. Model rotation only removes candidates that the returned plan reaches the goal without, and those are not landmarks. The chunk size and the reuse of plans therefore affect how many checks are needed, but not which facts end up in $B$.

Table 3.2: The studied modes of Algorithm 1.

| Mode | $c$ | $K$ | model rotation | $k$ over the run |
|---|---|---|---|---|
| naive | 1 | 1 | no | stays at $c$ |
| model rotation | 1 | 1 | yes | stays at $c$ |
| fixed chunking | $m$ | 1 | yes | stays at $c$ |
| adaptive | 1 | $K \geq 2$ | yes | grows after an unsolvable check |
| adaptive inverted | 1 | $K \geq 2$ | yes | grows after a solvable check |

*Naive* tests one candidate per check and discards nothing else, which makes it the baseline the other modes are measured against. *Model rotation* tests one candidate per check as well, but reuses every plan it receives. *Fixed chunking* keeps the size at a constant $m$ throughout the run. The two *adaptive* modes both grow the chunk geometrically by the rate $K$ and differ only in which outcome triggers the growth. Algorithm 1 shows the inverted rule, which grows after a check that leaves the goal reachable; the rule inherited from CadiBack is obtained by exchanging the two assignments at every marked line, so that the chunk grows after a confirmed landmark instead. Both rules touch only the value of $k$; the tests they perform and the sets they maintain are identical.

## 3.3 Cost Model

Since all modes agree on the result, the only question is how many checks each of them needs. A chunk test asks whether a group of candidates contains a landmark without revealing which one, and questions of this kind have been studied since Dorfman [4] proposed pooling blood samples to screen large populations for a rare disease. Instead of testing each sample on its own, one mixes several into a single pool and tests the pool. If the pool is clean, one test has cleared every sample in it; if it tests positive, the samples are then examined one by one to find the affected one.

Chunking faces the same trade-off, with the affected sample in the role of a landmark. Forbidding a whole chunk and finding the goal still reachable clears every candidate at once, just as a clean pool clears its samples. But if the goal becomes unreachable, no relaxed plan avoids all of its facts, and as with a positive pool the single test does not say which member is responsible, so the members must be checked one by one. The analogy lets us predict when chunking pays off.

Let $p$ denote the fraction of candidates that are landmarks, which we call the *landmark density*. A single candidate is thus a landmark with probability $p$ and not a landmark with probability $1-p$. A chunk of size $k$ contains no landmark exactly when each of its $k$ candidates fails to be one, and if we assume that landmarks are distributed independently over the candidates, these probabilities multiply: the chunk is free of landmarks with probability $(1-p)^k$, and contains at least one with the remaining probability $1-(1-p)^k$.

This gives the cost of a chunk. The joint check costs one in any case. With probability $(1-p)^k$ it resolves the whole chunk at once, and with probability $1-(1-p)^k$ the algorithm pays $k$ further checks to localise. The expected cost of a chunk of size $k$ is therefore

$$1 + k\left(1-(1-p)^k\right),$$

and dividing by the $k$ candidates the chunk contains gives the expected cost per candidate

$$\frac{1}{k} + \left(1-(1-p)^k\right).$$

The first term is the joint check split across the chunk, the second the probability of having to localise, which costs one check per candidate.

Landmarks are of course not distributed independently. The candidates all stem from one relaxed plan, and if a fact is needed by every plan, then the facts required to produce it usually are too. The formula is therefore an idealisation, and we use it for the qualitative behaviour it shows rather than for exact predictions.

Consider a task with few landmarks, so $p$ is small. Expanding $(1-p)^k$ gives $1-pk$ plus terms containing $p^2$ and higher powers, and for small $p$ these are negligible. Hence $1-(1-p)^k \approx pk$, which is just the expected number of landmarks in the chunk, and the cost per candidate becomes roughly

$$f(k) = \frac{1}{k} + pk.$$

The two terms pull in opposite directions: a larger chunk splits the joint check over more candidates, which lowers $1/k$, but it is also more likely to contain a landmark, which raises

$pk$. To find the size at which the two effects balance, we treat $k$ as a real variable and minimise $f$. Its derivative and second derivative are

$$f'(k) = -\frac{1}{k^2} + p, \qquad f''(k) = \frac{2}{k^3} > 0 \quad \text{for } k > 0.$$

This is a minimum and not a maximum, since the second derivative is positive for all $k > 0$. Setting $f'(k) = 0$ gives

$$\frac{1}{k^2} = p \quad \Longleftrightarrow \quad k^2 = \frac{1}{p} \quad \Longleftrightarrow \quad k^* = \frac{1}{\sqrt{p}},$$

and substituting $k^*$ back into $f$ yields the minimal cost per candidate,

$$f(k^*) = \sqrt{p} + \frac{p}{\sqrt{p}} = 2\sqrt{p}.$$

Since $f$ is convex, the best integer size is one of the two neighbours of $k^*$ and we round $1/\sqrt{p}$ to pick one of them.

**Example 3.3.** *Suppose one candidate in a hundred is a landmark, so* $p = 0.01$*. The optimal chunk size is then* $k^* = 1/\sqrt{0.01} = 10$*, and each candidate costs about* $2\sqrt{0.01} = 0.2$ *checks instead of the one the naive mode spends. If landmarks are four times rarer,* $p = 0.0025$*, the optimal size only doubles to* $k^* = 20$*, since quartering* $p$ *doubles* $1/\sqrt{p}$*, while the cost per candidate halves to* $0.1$*.*

Large chunks are thus worthwhile where landmarks are rare: most checks find nothing and clear the whole chunk in one step. The cost $2\sqrt{p}$ also tells us where this advantage disappears. The naive mode spends exactly one check per candidate, so chunking is cheaper precisely when $2\sqrt{p} < 1$, and this gives

$$2\sqrt{p} = 1 \quad \Longleftrightarrow \quad \sqrt{p} = \frac{1}{2} \quad \Longleftrightarrow \quad p = \frac{1}{4}.$$

At this density the recommended size is $k^* = 1/\sqrt{1/4} = 2$, and a chunk of two indeed costs $\frac{1}{2} + \frac{1}{4} \cdot 2 = 1$ per candidate, the same as checking each one separately. For denser tasks $k^*$ falls below 2, so the formula recommends what amounts to individual checks. This threshold inherits the approximation $1 - (1 - p)^k \approx pk$, which overestimates the chance of a positive chunk once $p$ is no longer small. Since chunking thus appears more expensive than it is, $p = 1/4$ is a lower bound on the true threshold: evaluating the exact condition $(1 - p)^k > 1/k$ instead places it higher. Dorfman's tabulated costs still show a saving at a prevalence of 30 per cent [4], so chunking helps a little beyond $p = 1/4$.

## 3.4 Estimating the Density

The alternative is to measure $p$ instead of guessing around it. Every resolved candidate is an observation: it either turned out to be a landmark or it did not, so the fraction of landmarks among the candidates resolved so far estimates the density of the task. Let $\ell$ be the number of landmarks confirmed and $r$ the number of candidates resolved at the time a chunk is formed. We estimate

$$\hat{p} = \frac{\ell + 1}{r + 2}$$

and set $k = \text{round}(1/\sqrt{\hat{p}})$, capped at the number of open candidates. The two constants implement Laplace smoothing: the estimate behaves as if one landmark and one non landmark had been observed before the run starts, so the denominator counts one pseudo observation per possible outcome. With no observations this gives $\hat{p} = 1/2$, and the influence of the constants fades as $r$ grows, so that $\hat{p}$ approaches the observed fraction $\ell/r$. Without them a run that has not yet seen a landmark would estimate $\hat{p} = 0$ and form a chunk of all remaining candidates, which is the worst possible guess if a landmark is among them.

Only the chunk size changes. The loop of Algorithm 1 is untouched, and the marked assignments no longer apply. We refer to this as the *density adaptive* mode and add it as a sixth row to Table 3.2.

# 4

# Implementation

The CadiLand algorithm was realised in two stages. A first prototype in Python served to validate the approach and to act as a reference. The algorithm was then reimplemented natively inside the Fast Downward planning system [6], with the aim of obtaining a faster implementation. This chapter describes the design of both. The empirical comparison between them is deferred to Chapter 5. The code is available on GitHub [10].

## 4.1 Fast Downward

Both implementations rely on Fast Downward [6], an open source classical planning system developed by Malte Helmert. It separates the work into two components. The *translator*, written in Python, reads a task given in PDDL, the standard input language for planning tasks, and converts it into the SAS$^+$ representation introduced in Section 2.3. The *search component*, written in C++, then reads this representation and searches for a plan.
The prototype works on the translated task, which it can read and modify on its own, whereas the native implementation is placed inside the search component itself.

## 4.2 Python Prototype

The prototype delegates the only operation CadiLand requires, deciding whether the goal is still reachable once a set of facts is forbidden, to an existing planner. It first has the translator produce the delete relaxed task as an SAS$^+$ file, and then calls Fast Downward on modified copies of that file, once per test, using `A*` with the `LM-Cut` heuristic [7]. Since the relaxed task contains no delete effects, each run either returns a plan or reports the task unsolvable.
The prototype passes two options to the translator. The task is translated in relaxed mode, so that no operator deletes anything, and invariant generation is switched off, so that atoms are not grouped into multi-valued variables. Every atom therefore ends up as its own variable with two values: the value 0 states that the atom holds, and the value 1 that it does not.
Forbidding a fact then requires no change to the planner, only to the task file. To test a candidate $f$, the prototype copies the SAS file and adds the pair $\langle v, 1\rangle$ to its goal, where

$v$ is the variable belonging to $f$. The planner is thereby asked for a plan that reaches the original goal while leaving $f$ false. Because a delete free task never retracts a fact, a fact that is false at the end was never made true at any point, so such a plan is exactly a plan that avoids $f$. If the planner finds one, $f$ is not a landmark, and the plan it returns is the one the algorithm uses for model rotation. If the planner reports the task unsolvable, no plan avoids $f$ and $f$ is a landmark. Chunks are treated the same way, by adding the negated value of every candidate in the chunk to the goal at once, which demands a plan avoiding all of them together.

One step is needed to connect the planner's output to the algorithm. Both the candidate pool and model rotation are defined over the facts a relaxed plan produces, but the planner reports a plan only as a sequence of operator names. The prototype therefore parses the operators of the SAS file together with their effects when it reads the task, and looks up each name of the returned plan in this table. Collecting the effects of all operators in the plan yields the facts the plan makes true, which is the set facts($\pi$) of Chapter 3.

The design is simple but the downside is its cost. Each single test writes a task file, starts a separate planner process and runs a full optimal search, even though the underlying question is relaxed reachability, which is polynomial and needs no search at all. The prototype therefore serves as a reference implementation for validating the algorithm rather than as a competitive method, and it motivates the native implementation described next.

## 4.3 Fast Downward Integration

The native implementation is registered as a search algorithm under the name `cadi_land`. It does not search for a plan: it receives the translated task, computes its landmarks, reports them and terminates immediately without expanding a state of the original state space. It is selected on the command line like any other search algorithm, for example

```
./fast-downward.py domain.pddl problem.pddl \
    --translate-options --relaxed \
    --search-options --search "cadi_land(
        model_rotation=true, adaptive=true)"
```

The two option groups correspond to the two components of Fast Downward. The translator is asked for the delete relaxed task, and the search options select CadiLand together with the mode it should run in. Table 4.1 lists the options and the modes of Chapter 3 they correspond to.

The task is compiled once into propositions and unary operators, each of which has a single effect and carries the preconditions of the operator it stems from. This compilation is paid once for the whole run rather than once per test.

A check, which we call a *relaxed exploration*, is a single Dijkstra style pass over this structure. Every proposition is assigned a cost, initialised to $\infty$, and a priority queue holds the propositions whose cost has been lowered, ordered by that cost. The facts of the initial state enter the queue with cost 0. Whenever a proposition is popped, the unary operators having it as a precondition are notified; once all preconditions of such an operator have been

reached, its effect is enqueued with the accumulated cost. The pass ends when the last goal proposition is popped, in which case the goal is relaxed reachable, or when the queue runs empty, in which case some goal proposition still has cost $\infty$ and the goal is unreachable. One exploration thus answers one call of SOLVABLE.

**Forbidding facts.** Blocking a candidate is not expressed in the task but inside the exploration. The implementation keeps a set of blocked propositions and, for every proposition, the set of operators that have it as an effect. Before an effect is enqueued, the operator producing it is looked up in these sets, and if it establishes any blocked proposition, the effect is not enqueued at all. A blocked operator therefore never enters the queue and cannot contribute to reaching the goal, while the task itself stays untouched. Testing a chunk means inserting its facts into the blocked set, running one exploration and removing them again; no copy of the task is created at any point.
It matters that the blocking removes whole operators and not single facts. In the compiled structure an operator with several effects appears as several unary operators, one per effect, all carrying the identifier of the operator they stem from. Suppressing only the unary operator whose effect is the blocked candidate would leave its siblings in place, so the original operator would still be applied and would still establish its remaining effects. The implementation therefore blocks by original operator: an effect is discarded as soon as the operator producing it adds any blocked proposition, which is exactly the set $O \setminus O_\Gamma$ of Section 3.2.

**Example 4.1.** *Consider the task with $s_0 = \{A\}$, $s_\star = \{G\}$ and the three operators $o_1 : A \to B \wedge C$, $o_2 : B \to G$ and $o_3 : C \to G$. Every plan has to start with $o_1$, which makes $B$ true, so $B$ is a landmark. Compiling the task yields the unary operators $A \to B$ and $A \to C$ from $o_1$, together with $B \to G$ and $C \to G$. Suppressing only $A \to B$ still allows $A \to C$ and then $C \to G$, so the goal remains reachable and $B$ is reported as no landmark. Blocking $o_1$ as a whole removes both of its effects, the goal becomes unreachable, and $B$ is confirmed.*

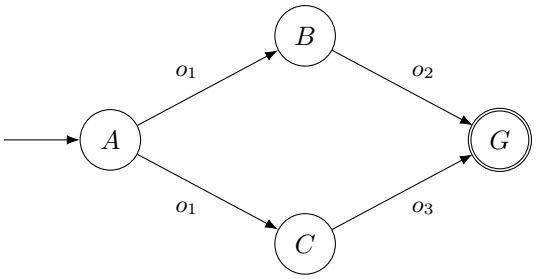


Figure 4.1: The task of Example 4.1, with $o_1$ producing both $B$ and $C$.

The Python prototype is not affected by this, since it does not suppress anything. It adds the negated candidate to the goal and thereby asks for a plan that leaves $f$ false throughout, which no plan applying an operator that adds $f$ can satisfy. The two implementations thus realise the same condition, one by constraining the goal and one by filtering the propagation.

For the same reason the candidate pool is built from all effects of the operators the relaxed plan uses, and not only from those effects the plan actually needed. Tracing the plan back from the goal touches one effect per operator, namely the one the propagation reached the goal through, but an operator with several effects establishes the others as well. In Example 4.1, $o_1$ appears in the plan because it produces $B$, yet applying it also produces $C$. The implementation collects the operators the exploration used and takes the union of their effects, so $C$ enters the pool alongside $B$ and is tested like any other candidate. Restricting the pool to the traced effects would risk missing landmarks that a task only ever establishes as a by-product.

Table 4.1: Options of the `cadi_land` plugin.

| Option | Default | Effect |
|---|---|---|
| `model_rotation` | `false` | reuse the relaxed plan of a successful check |
| `chunk_size` | 1 | candidates blocked jointly, the $c$ of Section 3.2 |
| `adaptive` | `false` | let the chunk size grow geometrically |
| `chunk_rate` | 2 | the growth rate $K$ |
| `grow_on_solvable` | `false` | grow after a check that finds no landmark |
| `density_adaptive` | `false` | set the chunk size from the estimated density |

With all options at their defaults the algorithm tests one candidate per check and discards nothing else, which is the baseline mode. The options are not fully independent: chunking requires `model_rotation`, since without it there is no plan to reduce the candidate set with, and `grow_on_solvable` as well as `density_adaptive` take effect only together with `adaptive`.

# 5
# Experimental Analysis

## 5.1 Downward Lab

All experiments were managed with Downward Lab [11], a toolkit for running and evaluating planner experiments. Each run consists of translating a task to its delete relaxation, invoking Fast Downward with one CadiLand configuration and parsing the emitted statistics; Downward Lab collects the parsed values of all runs into a single results table from which the reports and plots in this chapter are generated.

**Benchmarks.** We use the `DEFAULT_OPTIMAL_SUITE` of optimal STRIPS benchmarks from the International Planning Competitions, as distributed with Downward Lab. The domains it contains are listed in Appendix A. The suite covers a broad range of domains, so that the aggregate results are not dominated by any single one. Landmark extraction completed on 1827 tasks and these form the set the evaluation is based on. The domain `organic-synthesis-opt18-strips` is excluded: it exceeds the memory limit already during translation and yields no run for any configuration, so it contributes nothing that could be compared. Apart from this domain, every configuration is run on every task.

**Environment.** Calculations were performed at sciCORE (http://scicore.unibas.ch/) scientific computing center at University of Basel, using the Slurm workload manager. Each run was limited to 30 min of wall clock time and 3872 MiB of memory.

**Encodings.** All runs translate the task with `--relaxed`, so that the search component receives the delete relaxation the algorithm is defined on. Beyond this, the translator simplifies the task in two further ways by default, and both affect the set of facts over which landmarks are defined. Unreachable facts are filtered out, which removes variables that can only ever take a single value; under `--relaxed` these are mostly static and initial facts. Unimportant variables are filtered out as well, which removes variables the causal graph shows to be irrelevant to the goal. Each simplification can be switched off individually, with `--keep-unreachable-facts` and `--keep-unimportant-variables`.

We run the full suite four times, in a $2 \times 2$ design over the two options, listed in Table 5.1. Encoding $E_1$ is the default translation, $E_4$ the richest one, in which every fact is represented explicitly, and $E_2$ and $E_3$ isolate the effect of one option each.

Table 5.1: The four translator encodings. All of them use `--relaxed`.

| Encoding | `--keep-unreachable-facts` | `--keep-unimportant-variables` |
|---|---|---|
| $E_1$ | no | no |
| $E_2$ | yes | no |
| $E_3$ | no | yes |
| $E_4$ | yes | yes |

A translator configuration fixes the set of facts the task is built from, and every quantity we record over facts follows from it: the number of facts itself, the trivial landmarks contributed by the initial state and the goal, the candidates extracted from the first relaxed plan, and the landmarks among them. Within one encoding all modes report the same landmarks, as established in Section 3.2, and differ only in the number of explorations they need. The encoding therefore fixes what there is to find, and the mode only affects how many explorations it takes to find it, so the two effects can be discussed separately.

**Configurations.** Within each encoding we compare the seven configurations.

Table 5.2: The seven CadiLand configurations compared in the evaluation.

| Configuration | Parameters |
|---|---|
| `naive` | $c = 1$, no model rotation |
| `model-rotation` | $c = 1$ |
| `chunk-3` | $c = 3$ |
| `chunk-5` | $c = 5$ |
| `adaptive-k2` | $K = 2$, inherited rule |
| `adaptive-k2_inv` | $K = 2$, inverted rule |
| `density-adaptive` | chunk size from $\hat{p}$ |

**Recorded attributes.** Every run emits the following statistics, which Downward Lab parses and collects into the results table.

- *Cadi landmarks.* The landmarks found among the candidates. These exclude the trivial landmarks of the initial state and the goal, so they are the only ones the algorithm has to work for.
- *Candidates.* The facts extracted from the first relaxed plan, that is, the size of $\Lambda$ when the main loop starts. Goal and initial facts are excluded.
- *Coverage.* Whether the run completed, counted as one per task.
- *Explorations.* The number of relaxed reachability checks performed, one per call of Solvable.

- *Goal landmarks* and *initial landmarks.* The trivial landmarks contributed by the goal and by the initial state.
- *Landmarks.* The total reported, taken as the sum of the cadi landmarks and the goal landmarks. The initial landmarks are deliberately left out: a fact of the initial state may also be a goal fact, so adding all three counts could report the same landmark twice and overestimate the result.
- *Memory* and *time.* Peak memory, search time and total time of the run, as recorded by Downward Lab.
- *Test ratio.* The number of explorations divided by the number of candidates.
- *Total facts.* The number of facts of the translated task, counting only positive atoms. Negated atoms and the artificial values the translator introduces are excluded, since they cannot be landmarks and would only inflate the count.

The number of relaxed explorations is our primary performance metric. It counts the calls to the only non trivial operation of the algorithm and is therefore independent of the machine, free of measurement noise, and directly comparable across configurations. To make tasks of different size comparable, we normalise it to the *test ratio*, the number of explorations divided by the number of candidates. The naive mode has a test ratio of exactly 1 by definition, since it spends one exploration per candidate, and any smaller value is a saving.

## 5.2 Python Prototype versus Native Implementation

Both implementations were run on the full suite under the same limits, 30 min of wall clock time and 3872 MiB of memory. The native implementation solved every task, the prototype only a fraction, so the totals in Table 5.3 and Table 5.4 cover different task sets and are not comparable. Both tables list five modes, since the inverted adaptive rule and the density adaptive mode exist only natively.

Table 5.3: The native Fast Downward implementation on the full suite of Appendix A.

| | naive | model-rotation | chunk-3 | chunk-5 | adaptive-k2 |
|---|---|---|---|---|---|
| cadi landmarks | 48796 | 48796 | 48796 | 48796 | 48796 |
| candidates | 77652 | 77652 | 77652 | 77652 | 77652 |
| coverage | 1827 | 1827 | 1827 | 1827 | 1827 |
| explorations | 77652 | 61022 | 76568 | 70049 | 66164 |
| run time (s) | 194.13 | 183.41 | 196.53 | 191.69 | 186.99 |

Table 5.4: The Python prototype on the full suite of Appendix A.

| | naive | model-rotation | chunk-3 | chunk-5 | adaptive-k2 |
|---|---|---|---|---|---|
| cadi landmarks | 51 | 51 | 51 | 51 | 51 |
| candidates | 5073 | 5073 | 5073 | 5073 | 5073 |
| coverage | 218 | 248 | 264 | 249 | 255 |
| explorations | 163 | 84 | 88 | 79 | 93 |
| run time (s) | 5111.3 | 1778.21 | 1104.00 | 902.07 | 1821.63 |

The difference is larger than expected. The prototype answers each landmark test with a full optimal search, so a single task requires as many complete planner runs as it has candidates and each run pays for translation, preprocessing and search. The native implementation replaces all of this with one relaxed reachability pass per test. This shows most clearly in coverage. The native implementation extracts the landmarks of all 1827 tasks, whereas the prototype completes at most 264 of them, roughly one task in seven. The figure also varies with the mode: `naive` spends one search per candidate and eliminates nothing, so it is both the slowest configuration and the one that solves the fewest tasks, 218, while every mode that reuses plans reaches between 248 and 264. The gap is even clearer in the accumulated run time, where `naive` needs 5111 s against 902 s to 1822 s for the others. The prototype thus not only solves fewer tasks but is slower on the ones it does solve. The landmark counts in the two tables are summed over different task sets, 1827 against at most 264, which is why the prototype reports far fewer. On the tasks the two implementations share, however, they return the same landmarks.

Because the native implementation is far more efficient, with a geometric mean run time of about 0.03 s per task, and completes the entire suite, the remaining experiments use it exclusively. The prototype has served its purpose as a reference that validates the algorithm and it is not competitive as a method.

## 5.3 The Effect of the Encoding

Before comparing the modes, we fix a translation to compare them on. Recall from Section 3.2 that within one encoding all modes report identical landmark counts; the numbers in this section are therefore properties of the encoding alone and we quote one value per encoding. Table 5.5 gives the totals over the suite for the four encodings of Table 5.1.

Table 5.5: Totals over the full suite for the four encodings.

| | $E_1$ | $E_2$ | $E_3$ | $E_4$ |
|---|---|---|---|---|
| total facts | 505215 | 603891 | 676662 | 807029 |
| candidates | 77652 | 78289 | 96099 | 96352 |
| cadi landmarks | 48796 | 44011 | 62915 | 57854 |
| goal landmarks | 23452 | 26536 | 23452 | 26536 |
| initial landmarks | 0 | 91927 | 0 | 130002 |

With `--keep-unreachable-facts` ($E_1 \to E_2$) the number of candidates stays almost unchanged, from 77652 to 78289, while the cadi landmarks fall by 4785. By default Fast Downward removes facts it detects as constant: in the delete relaxation an initial or static fact never turns false again, so the translator optimises it away. The option keeps these constant facts in the task, and since they hold in the initial state they enter the initial landmarks, whose count jumps from 0 to 91927, rather than the candidates, which exclude initial facts by definition. The kept facts still matter, because each one can give a candidate an additional achiever, so blocking the original achievers of that candidate no longer makes the goal unreachable and the candidate loses its landmark status.

With `--keep-unimportant-variables` ($E_1 \to E_3$) the effect is the reverse. By default Fast Downward removes variables the causal graph shows to be irrelevant to the goal; the option keeps them, and the facts they carry are intermediate facts, neither initial nor goal, that lie on the relaxed plan. These facts become new candidates, so the candidate count rises from 77652 to 96099, and the cadi landmarks rise with it, from 48796 to 62915. Many of the added candidates are themselves provable landmarks, which is why both counts move together.
In summary, keeping unreachable facts lowers the number of found landmarks: the extra facts open additional ways to reach the same goal, so fewer candidates remain landmarks. Keeping unimportant variables raises it: the extra facts lie on the relaxed plan and become candidates themselves, most of which are landmarks. The two effects are close to independent but not exactly additive. Keeping unreachable facts removes about the same number of landmarks whether or not the other option is on, 4785 for $E_1 \to E_2$ against 5061 for $E_3 \to E_4$, the small difference coming from facts that the two options contribute jointly. Encoding $E_4$, with both options on, is the richest: every fact is represented explicitly. The central observation is that the landmark count is not fixed by the planning task alone but can be shifted by the encoding the translator produces. All four encodings solve the same 1827 tasks, yet each simplification changes the set of facts over which landmarks are defined, and thus how many there are.

## 5.4 Comparing the Modes

Both encodings $E_1$ and $E_4$ could be defended as the one to compare the modes on. Encoding $E_1$ is the default translation and the most optimised, since Fast Downward applies both of its simplifications. Encoding $E_4$ is the richest, representing every fact explicitly, so no fact is removed before the extraction and every candidate that could be a landmark is actually tested. We report the analysis on $E_4$, because it compares the modes on the fullest representation, independent of the translator's simplifications. The relations we find do not depend on this choice, and hold equally on $E_1$, whose figures we include afterwards without repeating the discussion.

Since all modes return the same landmarks, the comparison is entirely about cost, measured in relaxed explorations. Table 5.6 lists the explorations each mode needs over the suite, together with the test ratio, the explorations divided by the 96352 candidates. We use candidates rather than total facts as the denominator, because the modes only ever operate on the candidates: they are the facts of the first relaxed plan, so a single relaxed exploration already reduces the 807029 facts of the task to the candidates that remain in question, about 12 %. The naive mode has a ratio of exactly 1 by construction, one exploration per candidate, so any smaller ratio is a saving over testing every candidate on its own.

Table 5.6: The seven modes on encoding $E_4$, ordered by exploration ratio.

| Mode | explorations | ratio |
|---|---|---|
| `model-rotation` | 75088 | 0.78 |
| `adaptive-k2_inv` | 77346 | 0.80 |
| `density-adaptive` | 77648 | 0.81 |
| `adaptive-k2` | 80330 | 0.83 |
| `chunk-5` | 85232 | 0.88 |
| `chunk-3` | 93085 | 0.97 |
| `naive` | 96352 | 1.00 |

Every mode beats the naive baseline, so eliminating candidates from returned plans always pays. Model rotation is the cheapest, followed closely by the inverted adaptive rule and the density adaptive mode. Fixed chunking, by contrast, is only slightly better than naive. This is what the cost model predicts: chunking does not pay on tasks that are dense in landmarks. The overall density over the suite, the found landmarks divided by the candidates, is $57854/96352 \approx 0.60$, far above the threshold of about 0.3 from Section 3.3 beyond which chunking no longer pays. On the many tasks that are sparse, chunking still pays. This is why the fixed chunks beat naive at all, and it is also why the larger `chunk-5` beats the smaller `chunk-3`.

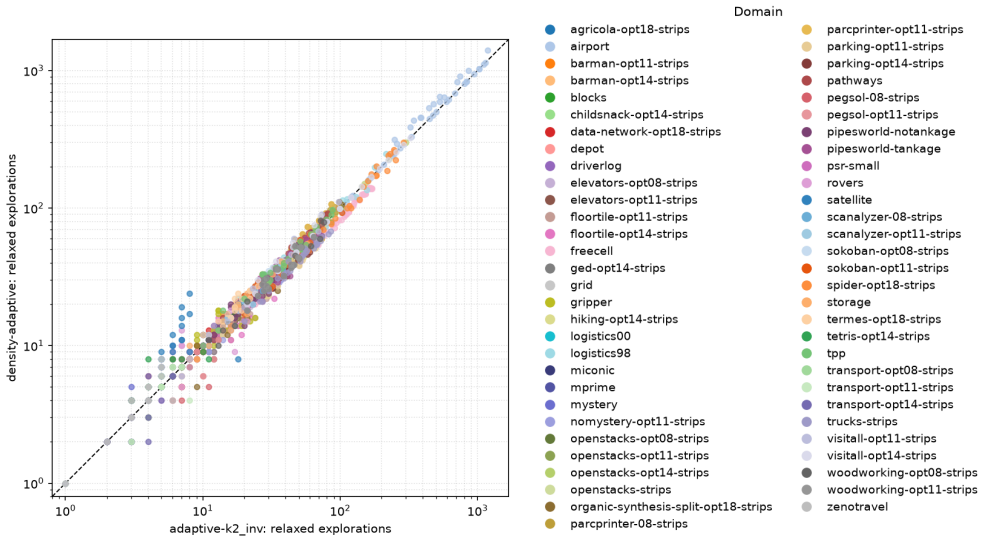


Figure 5.1: Explorations of the inverted adaptive rule ($x$) against the density adaptive mode ($y$), one point per task. On the diagonal both need the same number of explorations.

If we take a closer look at the two more interesting modes, the inverted adaptive rule and the density adaptive mode, neither of which appears in CadiBack, they differ by only about 0.01 in their overall ratio, yet they vary from task to task. Figure 5.1 plots the explorations of the two against each other, one point per task. The points fall on both sides of the diagonal: on some tasks the inverted rule needs fewer explorations, on others the density adaptive mode does. Which one wins depends on the task, and in particular on how many of its candidates turn out to be landmarks. We look into this more closely in the next section.

### 5.4.1 Landmark Density per Task

Since the modes vary from task to task, we want to find out which landmark density favours which mode. To make this visible we group the tasks by their own density, the found landmarks divided by the candidates, into a low, a medium and a high band. The lower boundary is fixed at 0.3, the threshold of Section 3.3 beyond which the model predicts chunking to stop paying; the upper boundary is the median density among the tasks above 0.3, which falls at 0.69 and splits the remainder into two nearly equal halves. Of the 1827 tasks, 135 have no candidates at all, because every fact on their first relaxed plan is already a goal or initial fact, so their density is undefined and they are left out of this grouping. The domains are listed in Appendix A. The remaining 1692 divide as shown in Table 5.7.

Table 5.7: The 1692 tasks with at least one candidate, grouped by landmark density, and the exploration ratio of each mode within each group (geometric mean).

| | low $[0, 0.3)$<br>589 tasks | medium $[0.3, 0.69)$<br>551 tasks | high $[0.69, 1]$<br>552 tasks |
|---|---|---|---|
| `naive` | 1.00 | 1.00 | 1.00 |
| `model-rotation` | 0.46 | 0.80 | 0.99 |
| `chunk-3` | 0.31 | 1.04 | 1.31 |
| `chunk-5` | 0.28 | 0.97 | 1.20 |
| `adaptive-k2` | 0.46 | 0.93 | 1.11 |
| `adaptive-k2_inv` | 0.33 | 0.94 | 1.01 |
| `density-adaptive` | 0.33 | 0.92 | 0.99 |

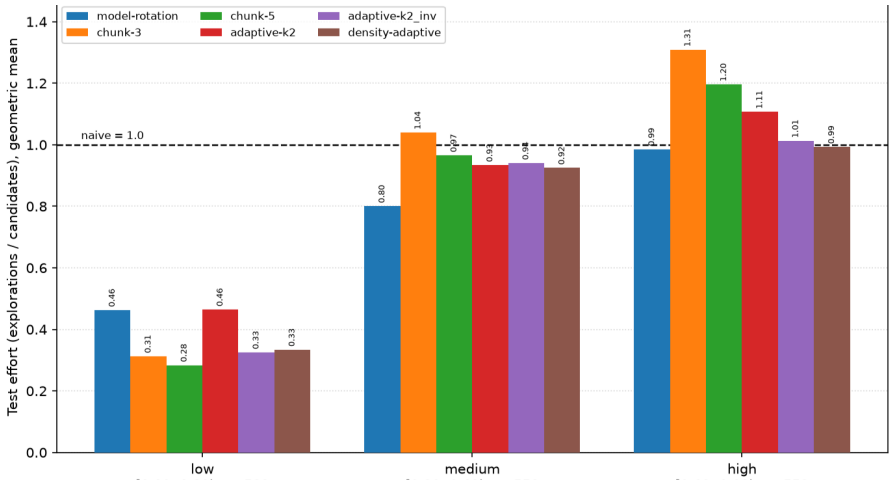


Figure 5.2: Exploration ratio of each mode by landmark density band, visualising Table 5.7.

A clear trend shows in fixed chunking. It is the best choice at low landmark density, where the fixed chunks reach 0.28 and 0.31. From a density of about 0.3 on it already does worse than the other modes on the medium band, which we expected from the cost model of Section 3.3. At high density it is not even worth it over naive, 1.31 for `chunk-3` and 1.20 for `chunk-5`. Fixed chunking thus pays off only when the task is known to have few landmarks.

`Model rotation` shows the opposite pattern. It is weaker than the chunking modes on the low density tasks, but it takes the lead once the density passes 0.3, and stays at or below naive in every band. So whenever a task has a density above 0.3, `model rotation` is the best choice.

For the tasks where the landmark density is unknown, the adaptive modes are the interesting ones. The inherited rule from CadiBack is clearly the worst of the three. It grows the chunk after a confirmed landmark, but on tasks where landmarks come one after another this makes the next chunk large exactly where it is most likely to fail, so it performs especially poorly there. This remains an assumption, since we did not measure how strongly the landmarks of a task are clustered.
The density adaptive mode and the inverted rule both perform very well and about equally. They do as well as fixed chunking on the low band but avoid its penalty as the density grows, staying at or below naive throughout. They reach this by different means: density adaptive recomputes the chunk size from a running estimate of the density, whereas the inverted rule only multiplies or resets a counter. The inverted rule is thus the simpler of the two to implement, and on this benchmark it reaches the same result without tracking any density at all.

**Excluding the Trivial Extremes** The density groups above include two kinds of task on which the best mode is fixed in advance. On a task of density 0 no candidate is a landmark, so a single chunk holding every candidate clears the whole set with one check and chunk size equal to the candidate count is optimal. On a task of density 1 every candidate is a landmark, so no returned plan ever avoids one and model rotation eliminates nothing, falling back to testing each candidate on its own; every larger chunk is then guaranteed to fail and costs strictly more, which leaves model rotation, tied with naive, as the best any mode can do. These extremes are not rare: 298 tasks (17.6 %) have density 0 and 430 (25.4 %) have density 1, together 43 % of the 1692 tasks. The domains are listed in Appendix A.
Because the optimal choice on these tasks is known without measurement, we repeat the comparison on the 964 interior tasks with $0 < d < 1$, where it is not. This shows whether the trivial cases dominate the ranking.

Table 5.8: Exploration ratio by density band on the 964 interior tasks with $0 < d < 1$ (geometric mean).

| | low $(0, 0.3)$ 291 tasks | medium $[0.3, 0.69)$ 551 tasks | high $[0.69, 1)$ 122 tasks |
|---|---|---|---|
| `naive` | 1.00 | 1.00 | 1.00 |
| `model-rotation` | 0.58 | 0.80 | 0.94 |
| `chunk-3` | 0.51 | 1.04 | 1.25 |
| `chunk-5` | 0.52 | 0.97 | 1.13 |
| `adaptive-k2` | 0.59 | 0.93 | 1.09 |
| `adaptive-k2_inv` | 0.56 | 0.94 | 1.06 |
| `density-adaptive` | 0.54 | 0.92 | 0.97 |

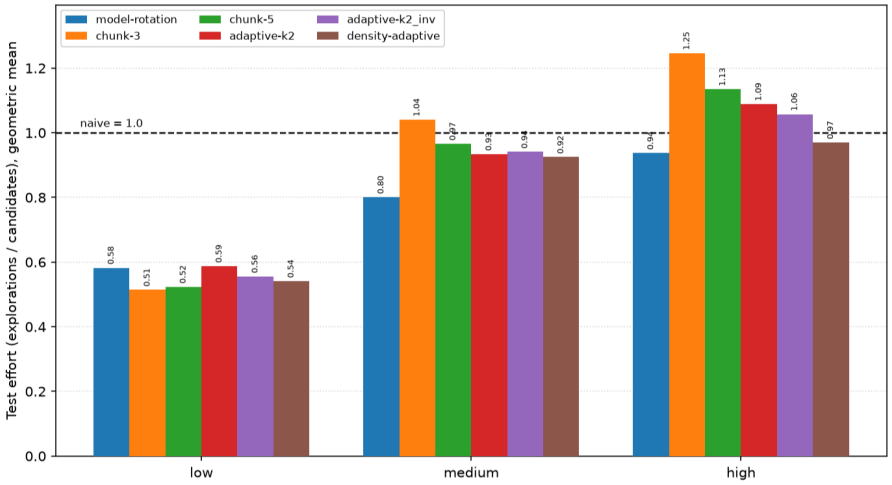


Figure 5.3: Exploration ratio of each mode by landmark density band, visualising Table 5.8.

As expected, the savings shrink on the low band. The density 0 tasks, where chunking clears whole blocks at once, are exactly the ones removed, so what remains is less favourable to large chunks. Model rotation, the only mode with chunk size 1, loses the least, from 0.46 to 0.58, since it never depended on those blocks in the first place. The fixed chunks lose more, and `chunk-5` most of all, from 0.28 to 0.52, a drop of 0.24 against 0.20 for `chunk-3`. This follows from the cost model: `chunk-5` is at its best on the sparsest tasks, therefore loses more of its advantage than `chunk-3`, and on the low band the two now come out level. On the medium band `chunk-5` still stays below naive, but this is mainly due to model rotation, not to the chunking itself.

The two adaptive modes move apart on the high band: the inverted rule gets worse, from 1.01 to 1.06, while density adaptive gets slightly better, from 0.99 to 0.97. The inverted rule looked good on the density 1 tasks only because there every check finds a landmark, so it never meets the solvable outcome it grows on and stays small by default. With those tasks gone, the interior high band still contains occasional solvable checks, on which the inverted rule grows and pays for it. Density adaptive keeps its chunks small on these tasks by reading the density directly, so it stays below naive where the inverted rule does not. Figure 5.4 makes this visible: more tasks sit below the naive line for density adaptive than for the inverted rule.

Since most of the interior tasks fall in the medium band, where chunking does not pay, the mode to pick is model rotation. Density adaptive is the second choice, and would be the better one on a benchmark with more sparse tasks, if the savings on the low band mattered more but one still wanted to stay near naive in the worst case.

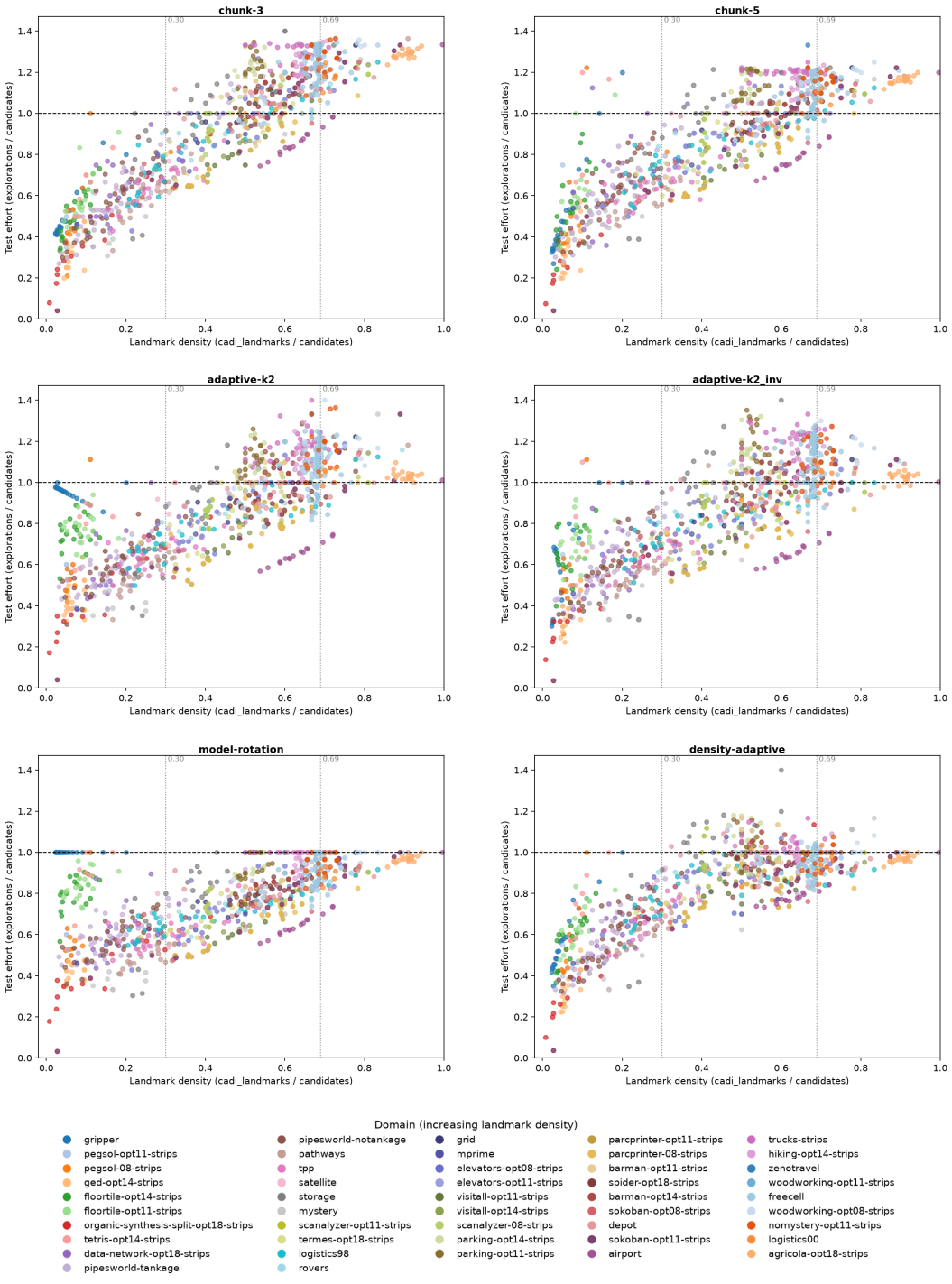


Figure 5.4: Exploration ratio against landmark density on the 964 interior tasks with $0 < d < 1$.

### Results for E1

| | low 467 | med 476 | high 709 |
|---|---|---|---|
| `naive` | 1.00 | 1.00 | 1.00 |
| `model-rotation` | 0.42 | 0.75 | 0.99 |
| `chunk-3` | 0.31 | 0.97 | 1.31 |
| `chunk-5` | 0.28 | 0.91 | 1.19 |
| `adaptive-k2` | 0.43 | 0.89 | 1.10 |
| `adaptive-k2_inv` | 0.31 | 0.88 | 1.01 |
| `density-adaptive` | 0.32 | 0.90 | 0.99 |

(a) all 1652 tasks with at least one candidate

| | low 236 | med 476 | high 190 |
|---|---|---|---|
| `naive` | 1.00 | 1.00 | 1.00 |
| `model-rotation` | 0.54 | 0.75 | 0.95 |
| `chunk-3` | 0.54 | 0.97 | 1.26 |
| `chunk-5` | 0.55 | 0.91 | 1.15 |
| `adaptive-k2` | 0.57 | 0.89 | 1.09 |
| `adaptive-k2_inv` | 0.54 | 0.88 | 1.04 |
| `density-adaptive` | 0.56 | 0.90 | 0.97 |

(b) the 902 interior tasks ($0 < d < 1$)

Table 5.9: Exploration ratio by density band low $[0, 0.3)$, medium $[0.3, 0.88)$, high $[0.88, 1]$

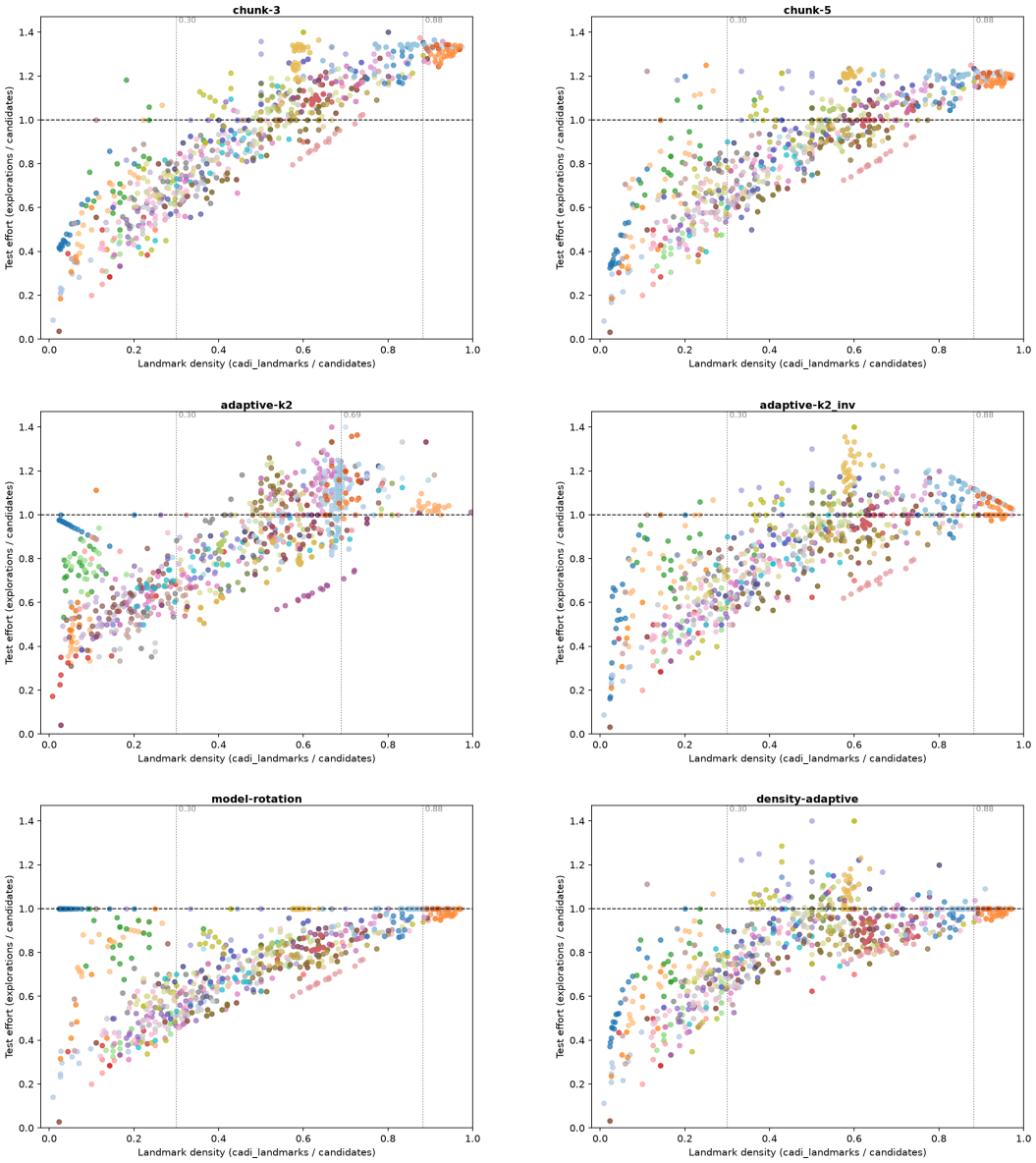


Figure 5.5: Exploration ratio against landmark density on the 902 interior tasks.

# 6
# Conclusion

In this thesis we transferred the backbone extraction of CadiBack to classical planning. We defined CadiLand, which computes the $h^1$ landmarks of a task by repeatedly asking whether the goal stays reachable in the delete relaxation when a candidate fact is forbidden. Each relaxed plan it obtains is reused to discard further candidates at no extra cost.

We realised the algorithm twice: as a Python prototype that answers each test with a full optimal planner run, and as a native search component inside Fast Downward that answers it with a single relaxed exploration. The prototype pays a full optimal search per test, so it completes only a fraction of the suite within the same limits, whereas the native implementation extracts the landmarks of all 1827 tasks. The prototype nonetheless served its purpose as a reference: on the tasks both implementations solve, the landmarks they return agree.

The evaluation led to two main findings. The first concerns the landmarks themselves: their number is not fixed by the task but changes with the translator options, since each simplification Fast Downward applies alters the set of facts over which landmarks are defined. The second concerns the cost of finding them. We compared the modes of the algorithm against each other: naive testing, model rotation, fixed and adaptive chunking, and a density adaptive variant. Which one performs best depends on the landmark density of the task.

One of the two ideas we took from CadiBack carries over. Reusing a returned solution to eliminate further candidates costs nothing and pays on every task, so model rotation performed best on average among the modes we tested and was never worse than testing every candidate on its own. Adapting the chunk size to the landmark density was competitive too, close behind model rotation overall and ahead of it on sparse tasks. Chunking itself does not carry over. The favourable and the expensive outcome swap roles in planning, so a chunk pays only when landmarks are rare, and on the benchmark suite they are mostly dense.

To summarise, chunking is the right choice when the landmark density is known to be low, below the threshold of about 0.3, and model rotation when it is known to exceed it. When the density is unknown in advance, a mode that adapts the chunk size during the run, such as the density adaptive or the inverted adaptive rule, is a reasonable compromise.

## 6.1 Future Work

Several directions remain open. We never examined how strongly the landmarks are clustered within a task, that is, whether they tend to occur in consecutive runs or lie scattered among the candidates. This matters for chunking, since a chunk clears its members in one check only when it happens to fall on a run without a landmark. Information about the clustering would open the way to algorithms that exploit it, growing chunks along landmark free stretches and shrinking them where landmarks accumulate.

The density adaptive rule could be improved with a better estimator. It never quite matched model rotation overall because its density estimate lags behind the true value early in a run, before enough candidates have resolved. An estimator that converges faster on the early candidates might close that gap on the dense tasks while keeping the advantage on the sparse ones.

Finally, CadiLand was studied purely as a landmark extractor, measured in reachability checks. Whether the landmarks it produces are useful to a planner, as heuristics or as search guidance, and whether the cost of extracting them is repaid there, is the open question that would link this work to planning in practice.

# Bibliography


[1] Christer Bäckström and Bernhard Nebel. Complexity results for $SAS^+$ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[2] Armin Biere, Nils Froleyks, and Wenxi Wang. CadiBack: Extracting backbones with CaDiCaL. In *Proceedings of the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[3] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In *Proceedings of the 36th International Conference on Computer Aided Verification (CAV 2024), Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024.

[4] Robert Dorfman. The detection of defective members of large populations. *The Annals of Mathematical Statistics*, 14(4):436–440, 1943.

[5] Patrik Haslum and Hector Geffner. Admissible heuristics for optimal planning. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, pages 140–149, 2000.

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

[7] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 162–169, 2009.

[8] Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

[9] Hans Kleine Büning and Theodor Lettmann. *Propositional Logic: Deduction and Algorithms*, volume 48 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1999.

[10] Elagkian Rajendram. fast_downward_c. https://github.com/elagkian/fast_downward_cadiland, 2026. Supervisors: David Speck, Travis Rivera Petit. Accessed: 2026-07-31.

[11] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. https://doi.org/10.5281/zenodo.790461, 2017.

# A

# Benchmark Domains

DEFAULT_OPTIMAL_SUITE without organic-synthesis-opt18-strips = [ 'agricola-opt18-strips', 'airport', 'barman-opt11-strips', 'barman-opt14-strips', 'blocks', 'childsnack-opt14-strips', 'data-network-opt18-strips', 'depot', 'driverlog', 'elevators-opt08-strips', 'elevators-opt11-strips', 'floortile-opt11-strips', 'floortile-opt14-strips', 'freecell', 'ged-opt14-strips', 'grid', 'gripper', 'hiking-opt14-strips', 'logistics00', 'logistics98', 'miconic', 'movie', 'mprime', 'mystery', 'nomystery-opt11-strips', 'openstacks-opt08-strips', 'openstacks-opt11-strips', 'openstacks-opt14-strips', 'openstacks-strips', 'organic-synthesis-split-opt18-strips', 'parcprinter-08-strips', 'parcprinter-opt11-strips', 'parking-opt11-strips', 'parking-opt14-strips', 'pathways', 'pegsol-08-strips', 'pegsol-opt11-strips', 'petri-net-alignment-opt18-strips', 'pipesworld-notankage', 'pipesworld-tankage', 'psr-small', 'quantum-layout-opt23-strips', 'rovers', 'satellite', 'scanalyzer-08-strips', 'scanalyzer-opt11-strips', 'snake-opt18-strips', 'sokoban-opt08-strips', 'sokoban-opt11-strips', 'spider-opt18-strips', 'storage', 'termes-opt18-strips', 'tetris-opt14-strips', 'tidybot-opt11-strips', 'tidybot-opt14-strips', 'tpp', 'transport-opt08-strips', 'transport-opt11-strips', 'transport-opt14-strips', 'trucks-strips', 'visitall-opt11-strips', 'visitall-opt14-strips', 'woodworking-opt08-strips', 'woodworking-opt11-strips', 'zenotravel']

Domains where every task has only initial and goal facts for E4=[movie, petri-net-alignment-opt18-strips, quantum-layout-opt23-strips, snake-opt18-strips, tidybot-opt11-strips, tidybot-opt14-strips, ged-opt14-strips (partially), mystery (partially)]

Domains where every task has density 1 for E4=[blocks, miconic, openstacks-opt08-strips, openstacks-opt11-strips, openstacks-opt14-strips, openstacks-strips, psr-small, airport (partially), logistics00 (partially), visitall-opt14-strips (partially), visitall-opt11-strips (partially), tpp (partially), scanalyzer-08-strips (partially), rovers (partially), satellite (partially), parcprinter-08-strips (partially), scanalyzer-opt11-strips (partially), storage (partially)]

Domains where every task has density 0 for E4=[childsnack-opt14-strips, driverlog, transport-opt08-strips, transport-opt11-strips, transport-opt14-strips, satellite (partially), mprime (partially), rovers (partially), zenotravel (partially), mystery (partially), pegsol-08-strips (partially), pegsol-opt11-strips (partially), tetris-opt14-strips (partially), scanalyzer-08-strips (partially), scanalyzer-opt11-strips (partially), organic-synthesis-split-opt18-strips (partially), floortile-opt11-strips (partially), pipesworld-notankage (partially)]

# Use of Generative Digital Tools

Claude (Sonnet 5) was used as a supporting tool during the preparation of this thesis, including for research orientation, suggestions for improving the structure and clarity of self written text, and assistance with debugging during implementation. Claude was not used as a scientific source and was not involved in conducting the experiments. The experimental design, implementation, analysis and final interpretation were carried out by the author. All factual claims and citations were checked against the cited sources. The author takes full responsibility for the final content of this thesis.