

University of Basel

Scalable Task Translation and Planning using Axiom Refactoring

Master Thesis

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

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. David Speck

Benedikt Heuser

October 07, 2026

Abstract

In domain-independent planning, tasks can be concisely formulated using PDDL. However, planners typically do not directly perform search on this representation. Instead, tasks are translated to a normalized representation, which simplifies the implementation state-space search.

The Fast Downward planner and its translation process are widely used in planning research. It parses, normalizes, and grounds tasks to bring them into a finite domain representation required by the search component. In its current version, the translator fails to efficiently process certain tasks due to an exponential blow-up resulting from an expensive normalization of condition formulas to DNF.

This thesis proposes an alternative normalization approach that avoids this bottleneck by rewriting certain conditions using axioms. We refer to our approach as *axiom refactoring*. We present two translator implementations with varying degrees of axiom refactoring, along with a hybrid approach, and we evaluate their performance and effects on translation and search for optimal and satisficing planning. We show that axiom refactoring can make previously infeasible tasks translatable, but also that it transfers some of the computational cost from translation into the subsequent search.

To mitigate these costs, we present a modified search implementation that reduces the computational and memory overhead of using axiom refactoring by partially removing derived variables from the state registry. Together with axiom refactoring, this optimization can extend the range of tasks that Fast Downward can process without becoming prohibitively expensive.

Acknowledgments

I would like to thank my supervisor Dr. David Speck for the great support and patience he showed me while writing this thesis. Furthermore, I extend my gratitude to the habitual dwellers of the “Masterarbeitsraum” and the “Zwischengeschoss”, who made the last few months much less solitary and bore the heat of an exceptional summer with me. Finally, I thank Prof. Malte Helmert for the opportunity of writing this thesis with his research group, who, in my opinion, does some of the best educational work our University has to offer.

Table of Contents

1 Introduction 6
2 Related Work 8
3 Background 10
3.1 Classical Planning Formalism 10
3.2 Planning with Axioms 11
3.3 Translation in Fast Downward 12
3.3.1 Normalization 13
3.3.2 Lifted Planning Tasks and Grounding 14
3.3.3 Invariant Generation and Variable Selection 14
4 Methodology 16
4.1 Axiom Refactoring 16
4.2 Quantifying the Blow-Up 17
4.3 Optimizing Memory During Search 18
5 Experiments 21
5.1 Experimental Setup 21
5.2 Task Structure of Axiom Refactored Tasks 22
5.3 Translation 24
5.3.1 Translator Coverage 24
5.3.2 Translator Memory Usage 26
5.3.3 Blow-Up Score and Hybrid Approach 27
5.4 Search 30
5.4.1 Expansions and Generations 30
5.4.2 Search Runtime 31
5.4.3 Memory 34
5.5 State Storage Optimization 35
5.5.1 Complete Axiom Omission 35
5.5.2 Partial Axiom Omission 37
5.5.3 Axiom Omission for Satisficing Planning 39
6 Discussion 41

6.1 Axiom Refactoring 41
6.2 Search Optimization 42
6.3 Practical Discussion of the Implementation 42
7 Future Work 44
8 Conclusion 46
A Additional Results for Hybrid Axiom Refactoring 51
B Additional Results for State Registry Optimization 56

1 Introduction

Fast Downward [1] is a widely used domain-independent planning system, which implements heuristic search with a range of admissible and inadmissible heuristics for optimal and satisficing planning. Users of the system typically provide planning tasks as input in the form of Planning Domain Definition Language (PDDL) [2] files. Upon execution, the PDDL files are parsed by Fast Downward and the task is internally translated to a custom translator output format. The newly generated output is then processed by Fast Downward's search component, which ultimately searches for a corresponding plan for the provided problem. At the time of writing this thesis, Fast Downward (version 24.06) supports PDDL 2.2 level 1 as well as action costs from PDDL 3.1[1].

Fast Downward's translator operates in four phases [3]. The first phase is a normalization of the PDDL task using logical equivalences. It comprises compiling away types, simplifying conditions and simplifying effects. The second phase is an invariant generation stage, which generates a mapping from the binary PDDL variables to a finite set of variables with finite domains. Up to this point in the translation, operators are often represented an abstract (lifted) form to reduce the size of the task representation. In the third phase, called the grounding stage, these lifted definitions are expanded (grounded). Finally, a set of state variables is chosen using the information from the previous phases to produce the final output. Formally, the custom translator output format is equivalent to a finite-domain representation (FDR) of the planning task.

The custom translator output format cannot represent disjunctive formulas directly. Consequently, a normalization of tasks' conditions into a disjunctive normal form (DNF) is performed by the translator. Because of this, the implementation sometimes experiences an exponential increase in the number of operators. We demonstrate in Section 3.3.1 how a

[1] Current information about Fast Downward's PDDL support can be found at https://www.fast-downward.org/latest/documentation/pddl-support/

single operator with a precondition consisting of a large conjunction of disjunctions is split into a large number of separate operators for each inner conjunction. Interestingly, this also results in a non-bijective mapping between plans of the original task and the transformed task. This can have effects on approaches like top-k planning [4] or plan counting [5].

The goal of this thesis is to extend the capabilities of Fast Downward to allow the solving of a greater variety of tasks. In particular, tasks that experience the operator blow-up described above are the focus of this work. We propose an alternative translator normalization process which employs axioms. We refer to this approach as *axiom refactoring*. Our approach aims to reduce the representation size of these tasks by transforming the parts of conditions that cause the explosion in the number of operators. This bottleneck also affects goal formulas, which are similarly aided by axiom refactoring.

Intuitively, axioms allow the representation of derived properties of states through the introduction of *derived variables*, which are evaluated given the state's *primary variables*. Derived variables are evaluated in a rule-based manner in every step of the search. In particular, they allow us to evaluate disjunctions in conditions by replacing the entire disjunction with a derived variable in the given condition formula. We present two variants of this idea: one that refactors only disjunctions, and a more aggressive variant that also refactors conjunctions, as well as a hybrid strategy that selectively applies refactoring based on a task-dependent blow-up estimate.

Because our approach relies on axioms, it can only be used with heuristics that support them. Whereas most satisficing heuristics do support axioms, most optimal heuristics do not. Consequently, our search experiments focus on A^* [6], [7] with the blind heuristic for optimal planning and LAMA-First [8] for satisficing planning. For these algorithms, we investigate a trade-off between translation coverage and subsequent search performance, and whether the previously described exponential blow-up can be avoided while keeping the resulting task practical for subsequent search.

Overall, this thesis makes the following contributions: (1) an axiom-based translator normalization that avoids the exponential operator increase of the current DNF-based approach, together with a more aggressive variant and a hybrid strategy that balances the two; (2) an empirical evaluation of translation coverage, memory, and representation size across a set of benchmark domains, some of which are known to trigger the blow-up; (3) an evaluation of the resulting search-time and memory overhead under both optimal and satisficing search; and (4) a state registry optimization that mitigates part of this overhead by avoiding the storage of unnecessary derived variables.

2
Related Work

Our work extends the capabilities of Fast Downward, which was originally described by Helmert [1]. With this paper, Helmert establishes many of the foundational features of Fast Downward's implementation including support for axioms and the separation of translation and search. In [3], he further provides a detailed overview of the four-phase translation process, which is still the basis of Fast Downward's translator implementation. Helmert also discusses the exponential blow-up that our work addresses and identifies axioms as a possible solution. We consider this to be the most important reference for understanding our translator-related discussion.

A central aspect of our work is the use of axioms as an alternative to explicitly expanding complex conditions. The role of axioms and derived predicates in planning was studied in detail by Thiébaux et al. [9]. They provide a formal semantics for PDDL axioms and show that, in general, axioms cannot be compiled away by expanding the formulas they represent while guaranteeing polynomial growth in the size of plans and domain descriptions. Their results therefore provide a theoretical motivation for supporting axioms directly rather than relying exclusively on compilation. This observation is closely related to our motivation for retaining some logical structure in the form of derived predicates instead of fully normalizing complex conditions into disjunctive normal form.

Axioms in planning remain an active research topic. Recent work by Grundke et al. [10] investigates how PDDL axioms provide a means of expressing structural properties of planning domains such as restrictions regarding valid input for generalized planning. Furthermore, Behnke et al. [11] have extended SAT-based planning to handle axioms and conditional effects with their new planner AxSAT. Their work demonstrates that providing support for axioms remains relevant across different planning paradigms. In the same vein, Speck et al. [12] show how to extend axiom support to symbolic search.

Our approach differs from previous work in that we introduce axioms as part of the translation process rather than requiring them as part of the original domain model. To the best of our knowledge, no previous work has provided an in-depth analysis of axiom refactoring, nor of the challenges of the resulting trade-off between translation coverage and computational cost of subsequent search.

3
Background

In this chapter we assume the reader is already familiar with the basics of propositional and predicate logic as well as the fundamentals of classical planning and state-based search. We cover the relevant parts of Fast Downward's translation approach including how tasks are normalized and how axioms are handled.

3.1 Classical Planning Formalism

A large variety of formalisms exist for specifying planning tasks. In this thesis, we use a formalism that allows us to concisely represent tasks while retaining sufficient detail to capture the relevant steps of Fast Downward's process.

Definition 3.1.1 A **planning task** is given by a tuple $\Pi = \langle \mathcal{V}, \mathcal{I}, \mathcal{O}, \mathcal{G} \rangle$ where:

- $\mathcal{V}$ is a finite set of propositional state variables v.
- $\mathcal{I}$ is an interpretation of $\mathcal{V}$ called the *initial state*.
- $\mathcal{O}$ is a finite set of *operators*. Each operator o is a tuple containing a *precondition* and an *effect* (defined later in this section) $\langle \text{pre}_o, \text{eff}_o \rangle$. The precondition is a propositional formula over $\mathcal{V}$. We say that operator o is applicable in a state s if $s \models \text{pre}_o$, where s is an interpretation over $\mathcal{V}$. For simplicity, we do not consider operator costs.
- $\mathcal{G}$ is a satisfiable propositional formula over $\mathcal{V}$ and is a conjunction of atoms and is called the *goal*.

Π induces a transition system over a set of states $\mathcal{S}$ where a *state* is an interpretation over $\mathcal{V}$. Intuitively, the goal of planning is to find a path of states from the initial state to a goal state. The sequence of operator labels of the operators applied along this path is called a *plan*. An operator's label is a (not necessarily unique) name of an operator.

In *optimal planning* we are interested in finding the shortest (or lowest cost) valid plan, whereas *satisficing planning* aims to find any valid plan for a given planning task.

Definition 3.1.2 Let $\mathcal{V}$ be a set of state variables. An **effect** over $\mathcal{V}$ is defined as follows:

- $\top$ is an (empty) effect.
- v and $\neg v$ with $v \in \mathcal{V}$ are effects.
- If e and e' are effects, then $(e \wedge e')$ is a (conjunctive) effect.
- If χ is a formula over V and e is an effect, then $(\chi \rhd e)$ is a (conditional) effect.

Intuitively, if an operator o is applied in a state s of a planning task, the effect of that operator describes the change in the values of each state variable. State variables not mentioned in an effect remain unchanged. We denote the effect of an operator o with eff_o. The exact semantics of effects are not detailed further as they are not relevant to this work. In later parts of this thesis, we sometimes refer to the collection of all operator preconditions, effect conditions and goal conditions as just the *conditions* of a planning task.

3.2 Planning with Axioms

Axioms are a feature of PDDL 2.2 that is integral to our proposed approach. They allow complex logical conditions to be represented by a single variable whose value is computed in every state based on the values of the planning task's primary variables.

Definition 3.2.3 A **planning task with axioms** is a tuple $\Pi = \langle \mathcal{V}, \mathcal{I}, \mathcal{O}, \mathcal{G}, \mathcal{D}, \mathcal{A} \rangle$. It is an extension of Definition 3.1.1 by two additional objects $\mathcal{A}$ and $\mathcal{D}$.

- $\mathcal{D}$ is a set of *derived variables*. Each derived variable d is a propositional variable with a default value of false $\bot$ (also denoted as $\neg d$).
- $\mathcal{A}$ is a set of *axioms*. An axiom has the form $head \leftarrow body$, where *head* is an assignment of true $\top$ to a derived variable (denoted as just d), and *body* is a conjunction over literals from both derived and non-derived variables $\mathcal{D} \cup \mathcal{V}$. We say an axiom is *applicable* in a state s if $s \models body$. The set of all axioms $\mathcal{A}$ is partitioned into a totally ordered set of *axiom layers* $\mathcal{A}_1 \prec ... \prec \mathcal{A}_k$. The assignment of axioms to axiom layers is chosen based on the derived variables in the axiom's head.

  For two distinct derived state variables d and d':

  1. if d appears in a body of an axiom whose head is d', then this axiom must be in the same layer or a subsequent layer of any axiom with the head d.
  2. If d appears with its default value in the body of an axiom with the head d', then this axiom must be in a subsequent layer of every axiom with the head d.
- The goal condition $\mathcal{G}$ and preconditions of operators pre_o and effect conditions may include derived variables. However, the initial state $\mathcal{I}$ as well as effects of operators are only defined over primary variables $\mathcal{V}$.

Let $s \in \mathcal{S}$ be a state of a planning task with axioms and the axiom layers $\mathcal{A}_1 \prec ... \prec \mathcal{A}_k$. Furthermore, let S_E be the set of all interpretations over $\mathcal{D} \cup \mathcal{V}$. The *extended state* $s_E \in \mathcal{S}_E$ of state s (also denoted as $\mathcal{A}(s)$) is the single state that is reached by evaluating the set of all axioms $\mathcal{A}$ according to the semantics described in the following: First, all derived variables are set to their default value. Then, for each layer $\mathcal{A}_i$, as long as there are applicable axioms in either this layer or any of the preceding layers, we apply any applicable axioms (i.e. assigning $\top$ to the derived variable mentioned in the head). Computing the extended state $\mathcal{A}(s)$ of a given state $s \in \mathcal{S}$ corresponds to the following algorithm [3]:

Algorithm 1: Evaluate Axioms

```
1:  procedure evaluate-axioms(A_1, ..., A_k, s)
2:      ▷ Initialize extended state
3:      for v ∈ V do
4:          s_E(v) ← s(v)
5:      end
6:      for d ∈ D do
7:          s_E(d) ← ⊥
8:      end
9:      for i ∈ 1, ..., k do
10:         while there exists head ← body ∈ A_i such that s_E ⊨ body and s_E ⊭ head
            do
11:             s_E(head) ← ⊤
12:         end
13:     end
14:     return s_E
15: end
```

3.3 Translation in Fast Downward

We provide a summary of Fast Downward's translation process as outlined by Helmert [3] with a focus on the normalization step, as it is the phase primarily affected by our proposed approach.

In order, translation is performed in the phases normalization, invariant generation, grounding, and variable selection. Translation is the process of mapping a given domain and problem, which together make up a planning task, to a new task in FDR. Both representations semantically encode the same planning task using different formalisms. Consequently, every plan of the translated task if it is also a plan for the original task. However, parameters of the translated task, such as the set of variables or the number of operators may differ from the provided task.

3.3.1 Normalization

In the normalization phase, condition formulas of the provided planning task are transformed to the form of single conjunctions of atoms. They are trivially in disjunctive normal form because they don't contain any disjunctions. However, before this can be done, universal quantifiers are removed by using the equivalence $\forall x\varphi \equiv \neg\exists x\neg\varphi$ and by introducing a derived variable that can be derived whenever $s \models \exists x\neg\varphi$. Following this, the universal quantifiers are replaced with $\neg d$ in conditions where they appear.

The following DNF normalization results in a blow-up in the representation size of the task. We will illustrate this using a grounded task for simplicity.

Example 3.3.1.1 Consider the planning task $\Pi_n = \langle \mathcal{V}_n, \mathcal{I}_n, \mathcal{O}_n, \mathcal{G}_n \rangle$ where:

- $\mathcal{V}_n = \{a_i, b_i \mid i \in \{1, ..., n\}\} \cup \{g\}$,
- $\mathcal{I}_n = \{a_i \mapsto \top, b_i \mapsto \top \mid i \in \{1, ..., n\}\} \cup \{g \mapsto \bot\}$,
- $\mathcal{O}_n = \{o_n\}$ where $o_n = \langle \bigwedge_{i\in\{1,...,n\}} (a_i \vee b_i), g \rangle$,
- $\mathcal{G}_n = g$.

Intuitively, this task only has a single optimal plan regardless of the value of n. The goal can be directly reached by applying operator o_n in the initial state. Fast Downward's translator, however, uses the laws of associativity, commutativity and distributivity to simplify the precondition and bring the disjunction to the root of the condition. For example, if $n = 3$, we rewrite pre_{o_3}:

$$\begin{aligned} & (a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge (a_3 \vee b_3) \\ \equiv\ & ((a_1 \wedge a_2) \vee (a_1 \wedge b_2) \vee (b_1 \wedge a_2) \vee (b_1 \wedge b_2)) \wedge (a_3 \vee b_3) \\ \equiv\ & (a_1 \wedge a_2 \wedge a_3) \vee (a_1 \wedge a_2 \wedge b_3) \vee (a_1 \wedge b_2 \wedge a_3) \vee (a_1 \wedge b_2 \wedge b_3) \\ & \vee (b_1 \wedge a_2 \wedge a_3) \vee (b_1 \wedge a_2 \wedge b_3) \vee (b_1 \wedge b_2 \wedge a_3) \vee (b_1 \wedge b_2 \wedge b_3) \end{aligned} \quad 1.$$

Notice that this final disjunction consists of $2^n = 2^3 = 8$ conjunctions. Finally, the translator splits o_n into a set of new operators with the same effect, but only the i-th conjunction as its precondition. For the purpose of reconstructing the solution, each of these sub-operators is treated as the same operator. This is a valid task transformation because o_n is applicable whenever any of the conjunctions are true under a given state.

In this example, we can see that there is an exponential blow-up in the representation size, which in turn impacts search performance. While here we considered only the operator precondition, the same normalization is also applied to effect conditions and goal formulas, which therefore also potentially suffer from the same blow-up depending on their task definition.

Finally, existential quantifiers are removed. The current implementation uses the equivalences $\exists(v, \exists(v', \varphi)) \equiv \exists(v + v', \varphi)$ and $\wedge\,(\varphi, \exists(v, \psi)) \equiv \exists(v, \wedge\,(\varphi, \psi))$ where v and v' are typed parameters to move existential quantifiers to the root of the condition and join the parameters of nested quantifiers such that there is only a single existential quantifier around the entire condition. Then, the quantifier is dropped from the condition and the parameters are added to the parameter list for the respective operator or axiom.

3.3.2 Lifted Planning Tasks and Grounding

Planning tasks are given as input to the planner using PDDL, which is a first-order language. Formally, this corresponds to a *lifted task* formalism [13], [14]. Instead of enumerating all possible operators explicitly, operators may be modeled using parameters that can later be instantiated with specified objects of the given type from the domain. This allows planning domains to be specified in a compact manner when many similar actions can be taken for similar conceptual objects. For example, the Miconic-Axioms domain uses this to represent floors and passengers.

Predicates describe relations between objects, and operators may contain parameters that appear in their preconditions and effects. A single lifted operator therefore represents a potentially large number of concrete actions obtained by instantiating its parameters with objects of the problem.

As part of translation, Fast Downward performs a *grounding* step in which these parameterized operators are expanded into a set of *ground operators* where all variables are replaced with constants. The grounding stage also replaces predicates with state variables used in the finite-domain representation employed by the search component.

Our previous discussion of the normalization phase assumed an already grounded task for simplicity. However, the computation of the disjunctive normal form can be performed analogously for lifted planning tasks.

3.3.3 Invariant Generation and Variable Selection

Aside from normalization and grounding, Fast Downward performs two further phases that are not directly affected by our proposed approach, but which we mention briefly for completeness. In the invariant generation phase, the translator analyzes the lifted operator schemas to identify sets of atoms that are mutually exclusive in every reachable state, called *mutex groups* [3]. In the variable selection phase, a subset of these mutex groups is chosen to partition the propositional atoms of the task into finite-domain variables, which reduces the number of variables tracked by the search component. Since our approach introduces derived variables only during normalization and does not otherwise modify the proposi-

tional atoms of the task, these two phases operate unchanged on the axiom refactored output of normalization, and we do not discuss them further in this thesis.

4
Methodology

4.1 Axiom Refactoring

We now propose our alternative normalization approach, which aims to avoid the blow-up we described in Section 3.3.1. The general idea is to replace disjunctions appearing in operator preconditions, effect coniditions and goal formulas with derived variables such that whenever one of the operands evaluates to true, the derived variable is also evaluated to true. Doing so allows us to prevent the exponential decomposition of condition formulas. The resulting formula is trivially in disjunctive normal form, because there are no more disjunctions.

Let us consider again the example from Section 3.3.1 for $n = 3$. Here, we add the derived variables d_1, d_2 and d_3 to represent the disjunctions $(a_1 \vee b_1)$, $(a_2 \vee b_2)$ and $(a_3 \vee b_3)$ in pre_o. Accordingly, we add the rules:

$$\begin{aligned} d_1 &\leftarrow a_1 \\ d_1 &\leftarrow b_1 \\ d_2 &\leftarrow a_2 \\ d_2 &\leftarrow b_2 \\ d_3 &\leftarrow a_3 \\ d_3 &\leftarrow b_3 \end{aligned} \tag{2.}$$

This allows us to rewrite the operator $o = \langle (a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge (a_3 \vee b_3), g \rangle$ as $o' = \langle d_1 \wedge d_2 \wedge d_3, g \rangle$. Note that the precondition $\text{pre}_{o'}$ is trivially in disjunctive normal form. Furthermore, we observe that for this example the number of axioms needed is $2n$ which is less than the 2^n operators resulting otherwise. This is a significant reduction in the

representation size of the planning task because it results in only a single operator o', as opposed to the 2^n operators with the same label, that o would otherwise be decomposed to.

In practice, our proposed approach modifies the task before it has been grounded. At this stage, condition-formulas may still have parameters. When introducing an axiom for a disjunction, we need to preserve the variables on which the replaced disjunction depends. For each disjunction, we collect its free variables and use them as the parameters of the newly introduced axioms.

Our example domain is an extreme case, because we start with a precondition in conjunction normal form (CNF). In the general case, this representation size reduction is domain-dependent. Domains that do not contain any disjunctions in conditions are not affected by the transformation, whereas tasks with disjunctions have their representation size reduced to a degree that depends on the context in which the disjunctions occur.

For comparison, we also consider a more aggressive transformation, where conjunctions are also replaced by axioms. In our running example we would add the rule:

$$d_* \leftarrow d_1 \wedge d_2 \wedge d_3 \tag{3.}$$

This results in every condition only having a single variable in its formula representation.

As a shorthand, we write the current implementation as $T^{\varnothing}$, the implementation that refactors disjunctions as $T^{\{\vee\}}$ and the implementation that also rewrites conjunctions as $T^{\{\vee,\wedge\}}$.

After the elimination of universal quantifiers in the first step of normalization conditions only consist of literals, conjunctions, disjunctions and existential quantifiers. While axiom refactoring disjunctions, the existential quantifiers are not treated specifically. Instead, existential quantifiers are eliminated in a separate step after disjunctions have been handled. After grounding, this may result in a duplication of operators, which is, however, based on the number of objects that can fill the corresponding parameters.

4.2 Quantifying the Blow-Up

In the previous section, we demonstrated with an example how refactoring disjunctions as axioms prevents a 2^n blow-up in the number of operators. However, we may not always want to introduce axioms for every disjunction, because, as we will see in later chapters, evaluating potentially thousands of axioms in every state may slow down search significantly. Additionally, many heuristics in Fast Downward do not support axioms. Therefore, if we could determine before refactoring whether a given condition will cause a significant blow-up before deciding to apply refactoring, we would be able to be more conservative with regards to which conditions should be refactored.

One way to approximate this is by calculating a score of how many conjunctions the corresponding disjunctive normal form would have. To do this we introduce the recursive Algorithm 2. It produces a high score for disjunctions nested in conjunctions and low scores for conjunctions in disjunctions. The score can be efficiently computed in $O(n)$ time with respect to the size of the condition.

Algorithm 2: Blow-Up Score

```
1:  procedure blowup-score(φ)
2:      if φ is a literal then
3:          return 1
4:
5:      else if φ is a conjunction then
6:          return ∏_{part ∈ parts(φ)} blowup-score(part)
7:
8:      else if φ is a disjunction then
9:          return ∑_{part ∈ parts(φ)} blowup-score(part)
10:
11:     end
12: end
```

The algorithm is an approximation of the number of conjunctions assuming that intermediate formulas are not simplified. For example, Algorithm 2 would yield a score of 8 for pre_{o_3} from Example 3.3.1.1. However, it results in an overestimation for formulas where the DNF transformation would result in a sub-formula that can be eliminated.

Furthermore, since the score is computed on the lifted task, it is independent of the number of objects. It captures the source of the blow-up (disjunctions nested in conjunctions), but not the combinatorial multiplication that occurs when parameters are later instantiated.

Utilizing Algorithm 2, we implement a hybrid version of $T^{\{\vee\}}$ called $T^{\{\vee\}}_{\text{hybrid}}$ which only refactors conditions whose blow-up score exceeds a certain threshold that we pick sufficiently high to avoid refactoring of disjunctions that would not cause a blow-up. An ideal threshold value would lie just high enough to include conditions for which the DNF normalization can be computed with a negligible overhead. This means that in practice, the ideal value is dependent on the domain, and can be found by examining the distribution of computed blow-up scores for the tasks we are trying to solve.

4.3 Optimizing Memory During Search

Searching for a plan requires the exploration of the state-space induced by the given planning task by iteratively expanding states. Whenever a new state is expanded Fast Downward's search component implementation currently stores an identification number for the newly generated states alongside the values of each state variable in a lookup table

called the state registry. To reduce the amount of memory consumed by the state registry, the state variables are packed into a set of integers such that each variable only uses the smallest number of bits required to represent all values in its respective finite domain. Consequently, we say that states registered in the state registry are *packed* states.

When using eager search, derived variables are initialized with their default value and are overwritten by their evaluated value on generation of the state. The current implementation of Fast Downward evaluates axioms on the *unpacked* representation of the state. This means that the packed state is expanded to a full vector of integers which only persists as long as the state is being explored by the search algorithm. In contrast, the state registry persists for the entire duration of the search and grows in size with each new generated state.

Its size can quickly become a limiting factor for large state-spaces. Additionally, despite the domain of derived variables being binary by their definition, axiom-refactoring can largely inflate the state registry's size because the additional variables' state values are stored for every single registered state.

However, one way to remedy this effect would be to exclude derived variables from the state registry, as they can always be recomputed from the set of primary variables at the cost of computation time. In this work, we refer to this approach as *complete axiom omission* for simplicity.

Refining this idea further, we consider which derived variables values are required when looking up a state in the state registry. The value of a state variable must be retrieved only when it is required for computing a heuristic or to evaluate goals, preconditions and effect conditions. We can thereafter also implement the state registry in such a way that only derived variables that appear in any of these conditions are registered and all other derived variables are excluded. This allows states to be retrieved and used directly without having to store the entire state or reevaluating axioms. We refer to this approach as *partial axiom omission*.

As part of our experiments we implement both versions and compare their performance to the current implementation. The first stores only primary variables and the second stores primary variables and necessary derived variables.

In this thesis, we generally use LAMA-First for the satisficing paradigm, which employs a (lazy) deferred evaluation algorithm. In contrast to eager search, lazy search computes the values of state variables during evaluation rather than during generation. Consequently, axioms can only be evaluated when a node is expanded. Despite this difference, axiom omission can be implemented analogously in lazy search and eager search. However, because the landmark heuristics used by LAMA-First require states to be unpacked before

evaluation with the partial axiom omission strategy, the partial axiom omission strategy loses its practical advantage over the full axiom omission implementation. We therefore consider an additional search configuration that allows us to perform satisficing search without using landmark heuristics. Specifically, we consider greedy best-first search [15] with two open lists that alternate [16]: one using the FF heuristic and the other using FF-only preferred operators [17].

5
Experiments

In the following, we present the results of our experiments exploring the effect of axiom refactoring on search and translation behaviour for a limited set of benchmark tasks. We start investigating the effect of axiom refactoring on the structure of translated tasks. We then evaluate translation independently of the subsequent search and compare different normalization implementations on their ability to successfully translate the benchmark tasks. Following this, we explore how those same translation configurations affect search. Finally, we present the results of our search optimizations using the axiom omission strategies from the previous chapter.

5.1 Experimental Setup

Our proposed approach only affects tasks containing at least one disjunction in any operator precondition, effect condition or goal formula. To construct a representative benchmark set, we collected benchmarks which meet this criterion from the standard Fast Downward benchmarks repository[2] and from the benchmarks used in [12]. Additionally, we generated tasks for the example domain from Section 3.3.1 for the range $5 \leq n \leq 20$. Together, they comprise 18 domains of varying complexity and axiom structure.

All experiments were performed on the University of Basel's cluster using Intel® Xeon® Silver 4114 CPUs with a memory limit of 4 GB and time limit of 20 minutes. The Python version used for the translator was version 3.11.5.

In this thesis, we largely limit our discussion to two search configurations. For optimal planning, we employ the A^* search algorithm with the blind heuristic. This heuristic provides a suitable baseline for comparing implementations due to its uninformed nature resulting in a similar number of expansions and generations for all implementations. For

[2] https://github.com/aibasel/downward-benchmarks#

the satisficing paradigm we use LAMA-First, which corresponds to the plan found by the first phase of LAMA [8]. LAMA-First employs deferred (lazy) evaluation with four open lists: two using the FF heuristic and two using a Landmark heuristic. For each heuristic, one open list uses the regular heuristic and the other considers only preferred operators. Both algorithms represent widely used planning configurations with support for axioms.

5.2 Task Structure of Axiom Refactored Tasks

Axioms are evaluated in a layered model. Each layer can only be evaluated when all layers before it have been evaluated. If we think of axioms as a graph of derived variables with an edge between any two variables, where one depends on another to be evaluated, we can find strongly connected components (SCCs) of derived variables. Strongly connected variables in the same SCC cannot be strictly ordered by dependency, since they each depend on each other. They are consequently assigned the same axiom layer. Since we introduce a large number of derived variables when we use axiom refactoring, we may want to know how these newly introduced axioms affect the dependency structure of derived variables in the task.

From Table 1, we can see that for most tasks in our benchmark set, the average maximum SCC size is much lower than the average number of SCCs. This implies that the dependency structure is generally fragmented into many small SCCs of size 1, unless larger SCCs already existed before axiom refactoring. This suggests that axiom refactoring adds rules in such a way that it increases the sizes of existing large components, while individual weakly connected derived variables are not strongly connected with the newly introduced variables. Intuitively, axiom refactoring induces a tree-like structure where axiom rules connect subformulas of the original condition formula through derived variables.

Furthermore, we consider the context of where derived variables appear in conditions. Figure 1 shows the distribution of derived variables in preconditions, effect conditions, and goal conditions. We count only derived variables that occur directly in a condition, excluding variables that occur only in the body of the axiom defining them or in axioms required to derive them. Moreover, these counts are computed after translation and grounding, rather than directly after normalization. Thus, the analysis is performed on the final FDR representation, i.e., exactly the representation that is parsed by the search component of Fast Downward. This information is relevant for our approach to search optimization through axiom omission in the state registry (see Section 4.3).

As shown in the figure, the average percentage of derived variables occurring in any of the three condition types is relatively low across all implementations, including $T^{\varnothing}$. This percentage decreases further with axiom refactoring. Across all domains, an average of 61%

Domain	Number of SCCs			Maximum SCC Size		
	$T^{\varnothing}$	$T^{\{\vee\}}$	$T^{\{\vee,\wedge\}}$	$T^{\varnothing}$	$T^{\{\vee\}}$	$T^{\{\vee,\wedge\}}$
acc-cc2 (8)	34.00	68.00	794.00	1.00	1.00	1.00
assembly (30)	176.43	270.57	1135.70	1.00	1.00	1.00
blocker (6)	827.17	999.50	4655.17	1.00	1.00	1.00
collab-and-comm (1)	60.00	420.00	767.00	1.00	1.00	1.00
explode (14)	0.00	11.50	12.50	0.00	1.00	1.00
ged1-ds1 (4)	25.00	47.50	5819.50	2.00	2.00	13.50
ged1-ds2nd (12)	43.67	82.83	15169.83	3.50	3.50	20.83
miconic-axioms (150)	1.00	1230.67	1283.10	31.00	31.00	1291.67
miconic-fulladl (150)	119.21	1232.79	4727.32	1.00	1.00	1.00
muddy-child (1)	20.00	112.00	185.00	1.00	1.00	1.00
muddy-children (1)	48.00	240.00	445.00	1.00	1.00	1.00
pathways (30)	0.00	20.43	1318.60	0.00	1.00	1.00
psr-large (41)	1822.80	5792.34	14801.41	68.32	221.66	494.41
psr-middle (50)	107.62	346.40	940.98	28.26	91.68	204.48
social-planning (2)	124.00	266.00	795.50	5.00	5.00	5.00
sum (1)	36.00	261.00	721.00	1.00	1.00	1.00
telegraphs (48)	1225.00	1225.00	7728.50	1.00	1.00	1.00
word-rooms (2)	275.00	1630.00	2823.00	1.00	1.00	1.00
Arithmetic Mean (551)	274.72	792.03	3562.39	-	-	-

Table 1: Average number of strongly connected components (SCCs) and size of the largest SCC for each domain and translator implementation. Only tasks where all implementations successfully terminated are included. The number in parentheses after each domain is the number of tasks for which this was the case.

of derived variables occur in at least one precondition, effect condition, or goal condition for $T^{\varnothing}$, compared to 39% for $T^{\{\vee\}}$ and 43% for $T^{\{\vee,\wedge\}}$. However, the mean is influenced by several domains with substantially higher percentages, while most tasks contain only a small fraction of the task's derived variables in these conditions.

Finally, in Section 4.1 we described how axiom rules can be reused when they correspond to exactly the same expression. However, this situation appears to be rare. In our experimental evaluation, we encountered only a single instance of axiom-rule reuse, which occurs in the Social-Planning domain. We note, however, that this may partly be attributable to how tasks are modeled in PDDL using axioms: when two derived properties are semantically equivalent, the corresponding property may already have been explicitly modeled by the user as a derived variable, thereby eliminating an opportunity for axiom-rule reuse. One

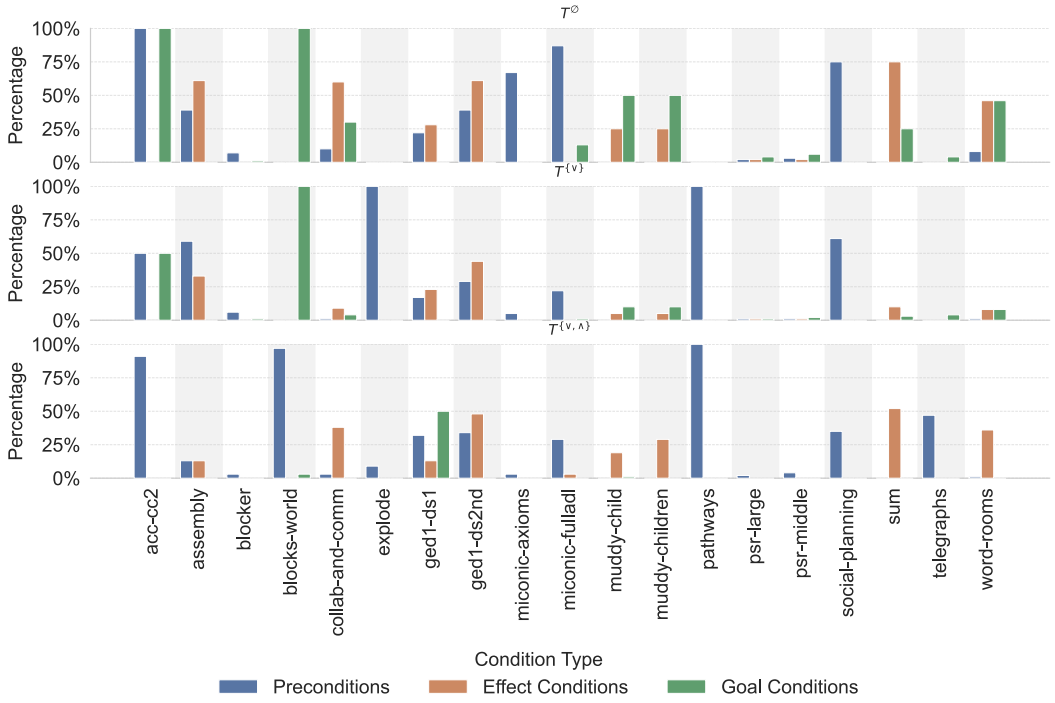

Figure 1: Shows the percentage of derived variables appearing in at least one precondition, effect condition or goal condition after translation. Every derived variable may be present in multiple types of conditions, therefore, percentages for a given domain do not sum to 100%. If a condition does not include any derived variables the percentage given is 0%.

could also consider other axiom-rule reuse strategies. However, we leave this for future work, as we believe the benefit to be limited given our benchmark set.

5.3 Translation

In this section we explore the effects axiom refactoring on the translation process in isolation. We compare the current normalization $T^{\varnothing}$ with the two uniform refactoring strategies $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ from Section 4.1, and with the hybrid variant $T^{\{\vee\}}_{\text{hybrid}}$ from Section 4.2. All configurations are run with the resource limits described in the experimental setup. We evaluate them along the following dimensions: how many tasks can be translated within the resource limits, how much memory the translator requires, and how large the resulting task representations are. Additionally, we analyze the blow-up score distribution and the threshold used for $T^{\{\vee\}}_{\text{hybrid}}$.

5.3.1 Translator Coverage

Table 2 compares the ability of each normalization algorithm to translate tasks from our benchmark domains within the previously described memory and runtime parameters. We can see that $T^{\{\vee\}}$ is able to translate a strictly greater or equal number of tasks per domain than $T^{\{\vee,\wedge\}}$. This result is unsurprising since rewriting conjunctive conditions as axioms

Domain	$T^{\varnothing}$	$T^{\{\vee\}}$	$T^{\{\vee,\wedge\}}$	$T^{\{\vee\}}_{\mathbf{hybrid}}$
acc-cc2 (8)	8	8	8	8
assembly (30)	30	30	30	30
blocker (9)	7	7	6	7
collab-and-comm (3)	1	2	2	2
explode (21)	14	21	21	21
ged1-ds1 (110)	4	4	4	4
ged1-ds2nd (156)	12	12	8	12
miconic-axioms (150)	150	150	150	150
miconic-fulladl (150)	150	150	150	150
muddy-child (7)	1	4	4	4
muddy-children (5)	1	3	2	3
pathways (30)	30	30	30	30
optical-telegraphs (48)	48	48	48	48
psr-large (50)	50	47	40	50
psr-middle (50)	50	50	50	50
social-planning (2)	2	2	2	2
sum (5)	1	3	2	3
word-rooms (5)	2	5	4	5
Total Coverage (839)	561	576	561	579
Norm. Coverage (18)	11.83	14.27	13.39	14.33

Table 2: Number of successful translations per domain for different levels of axiom refactoring within resource limits. Success is measured by the translator's exit code. The threshold value chosen for $T^{\{\vee\}}_{\text{hybrid}}$ is equal to 10. The number in parentheses after each domain name indicates the number of available tasks. The total coverage is the sum of all translated instances, while the normalized coverage is the aggregated percentage of solved instances per domain.

simply moves those conditions into an axiom body, therefore not providing any obvious benefit for translation.

Furthermore, $T^{\{\vee\}}$ is able to translate significantly larger tasks than $T^{\varnothing}$ for domains where multiple disjunctions are nested inside conjunctions. For example, this includes domains such as Muddy Children, Word Rooms and Sum from [18] and the Explode domain from Section 3.3.1. For these domains additional translated instances correspond to higher order versions of that domain. In other words, translating 4 tasks for the Muddy Children domain corresponds to successfully translating the task with up to 6 children instead of only 3 children, which is the smallest version of that domain.

[3] PSR-Large and PSR-Middle are domains from IPC-4 https://ipc04.icaps-conference.org/deterministic/

The only domain where $T^{\{\vee\}}$ translated fewer instances than $T^{\varnothing}$ is PSR-Large[3], where three tasks were not finished due to the memory limit having been reached. The cause for this increase in memory is that $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ introduce a large number of additional variables for tasks of this domain. Despite this, the effect of axiom refactoring on memory during translation can vary greatly between domains. We discuss this further in Section 5.3.2.

$T^{\{\vee\}}_{\text{hybrid}}$ achieves the highest coverage with 580 tasks. It exactly matches $T^{\{\vee\}}$ in every domain except PSR-Large, where it translates all 50 tasks, and thus never does worse than the better of $T^{\varnothing}$ and $T^{\{\vee\}}$ on any domain. This result is influenced by the choice of the threshold parameter. We discuss the motivation behind the choice of this parameter's value in Section 5.3.3 along with further discussion of the hybrid approach. However, we include the coverage results here, such that they can be compared directly with the other translator configurations, and to avoid restating their coverage results.

5.3.2 Translator Memory Usage

Table 3 shows that memory usage can be strongly influenced by axiom refactoring. For domains such as PSR-Large and PSR-Middle, the memory usage significantly increases for higher levels of axiom refactoring. On the other hand, we see decreased memory usage for domains such as the Muddy Children domain. The domains where memory usage was noticeably lower are the same domains where $T^{\{\vee\}}$ was able to translate more tasks than $T^{\varnothing}$ (see Table 2).

The reason why these latter domains benefit from axiom refactoring is precisely the exponential blow-up we aim to address with our approach. On the other hand, domains for which the memory usage increases are ones where a large number of axioms are introduced for sub-formulas that do not contribute towards a blow-up (for example PSR-Large).

$T^{\{\vee\}}_{\text{hybrid}}$ exhibits the lowest overall memory usage because it is generally on-par with the better of $T^{\{\vee\}}$ and $T^{\varnothing}$. As before, we include this result here for completeness and to avoid restating these results.

Furthermore, we now consider the relationship between the representation size of translated tasks and our translator configurations. In the context of this experiment, task representation size refers to the size of the encoding of the planning task after translation. Figure 2 shows that the $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ configurations result in a higher task representation size for tasks where the operator blow-up does not dominate the translated task size. However, we can also see a large reduction in representation size for those domains which have large CNF-like conditions. This is most clearly visible for the Explode domain. $T^{\{\vee,\wedge\}}$ only increases

Domain	$T^{\varnothing}$	$T^{\{\vee\}}$	$T^{\{\vee,\wedge\}}$	$T^{\{\vee\}}_{\text{hybrid}}$
acc-cc2 (8)	**48.540**	**48.540**	53.421	49.520
assembly (30)	**40.438**	40.813	77.934	40.442
blocker (6)	**65.813**	67.691	133.885	66.007
collab-and-comm (1)	206.772	**189.808**	191.856	180.756
explode (14)	363.920	172.253	**172.180**	172.276
ged1-ds1 (4)	**250.554**	252.078	674.088	250.812
ged1-ds2nd (8)	**156.946**	157.197	430.894	157.998
miconic-axioms (150)	**34.643**	38.325	42.936	34.659
miconic-fulladl (150)	**43.361**	47.641	68.860	42.628
muddy-child (1)	52.588	**35.180**	36.204	35.184
muddy-children (1)	59.756	**37.228**	38.252	37.230
optical-telegraphs (48)	**63.511**	63.572	98.542	63.604
pathways (30)	38.020	38.152	42.422	37.989
psr-large (40)	**152.310**	406.325	721.065	152.367
psr-middle (50)	**44.779**	70.010	104.042	44.791
social-planning (2)	**35.180**	36.204	38.252	35.184
sum (1)	145.360	**41.324**	45.420	41.328
word-rooms (2)	959.498	163.786	167.930	**163.118**
Arithmetic Mean	153.444	105.896	174.344	**89.216**

Table 3: Translator peak memory in Megabytes averaged over all runs for the pair of domain and algorithm. The number in parentheses after each domain indicates the number of tasks that all three algorithms successfully translated. Tasks for which none of the algorithms produced a value were excluded. The numbers in bold text indicate the lowest value for each domain. $T^{\{\vee\}}_{\text{hybrid}}$ uses a threshold value of 10.

representation size compared to $T^{\{\vee\}}$ because it adds axioms for conjunctions, which add encoding size without reducing the number of operators.

Overall, we conclude that, while axiom refactoring generally introduces a memory overhead for the introduction of additional variables, it can considerably benefit translation of specific domains, which are burdened by a disjunction-based blow-up in any of its preconditions, effect conditions or goal conditions.

5.3.3 Blow-Up Score and Hybrid Approach

In Section 4.2, we introduced the blow-up score as an approximation of the number of conjunctions in the DNF of a condition. This score is the primary decision mechanism of our hybrid implementation $T^{\{\vee\}}_{\text{hybrid}}$. A condition is refactored with the strategy of $T^{\{\vee\}}$ if its score exceeds a threshold τ, and is otherwise left to the standard DNF normalization. In

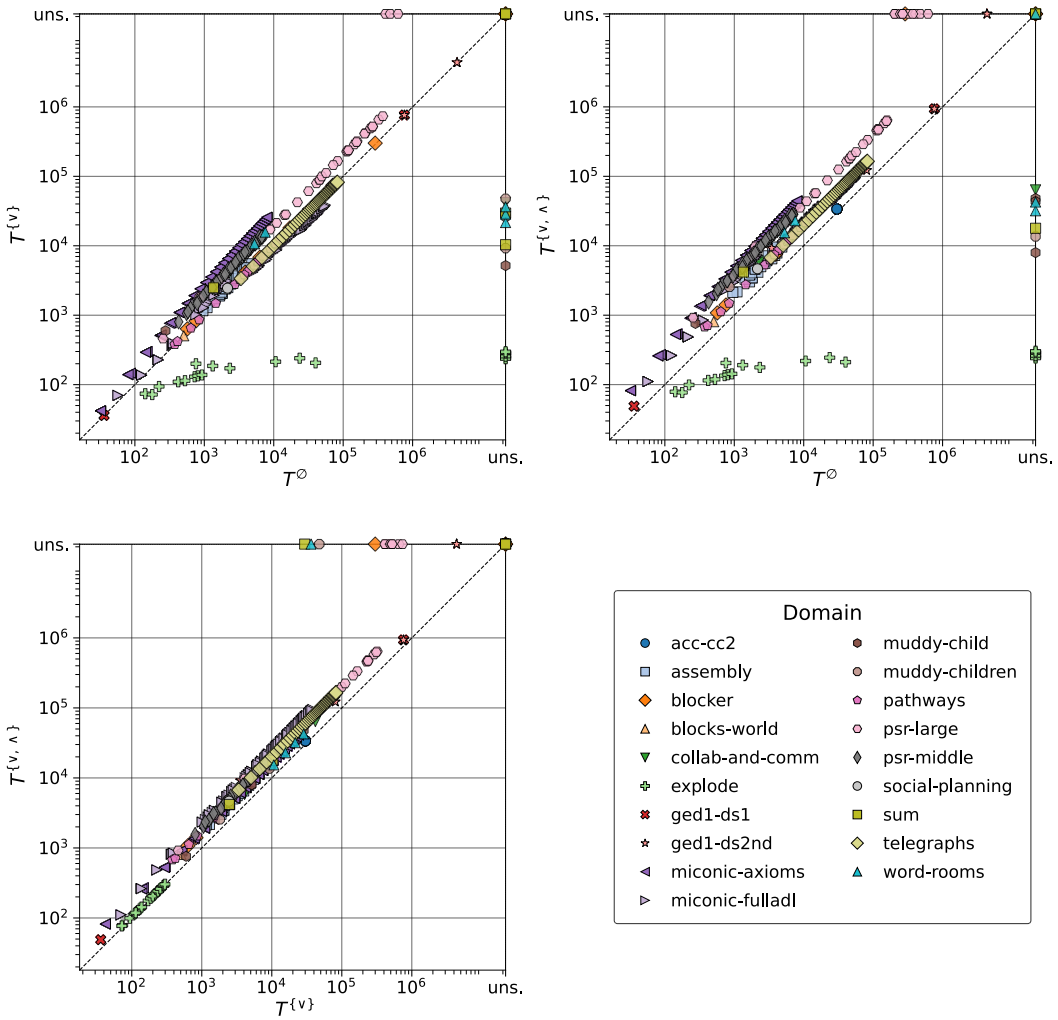

Figure 2: Shows the absolute task representation size after translation for each pair of translator configurations from $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$. Unsolved instances for one or both implementations are placed on the grid-line marked with "uns.".

this section we examine how the scores are distributed over our benchmark and motivate the threshold used in our experiments.

Figure 3 shows a separation of domains into two groups. Some domains, such as the Explode domain, have a progression in task size from one instance to another, with increasing blow-up scores with each subsequent instance. Others have the same blow-up score for every instance such as PSR-Large. Tasks of the second type differ in the objects they contain but not in their condition formulas. Note that the total blow-up score is the sum of the scores of individual preconditions, effect conditions and goal conditions. Therefore, even domains with a total blow-up score greater than τ may still only use the DNF normalization strategy.

To achieve optimal performance, we would perform a sweep of different threshold values over the benchmark domains and pick the best-performing value. Alternatively, if doing so is computationally infeasible, one could also tune the threshold parameter using a smaller

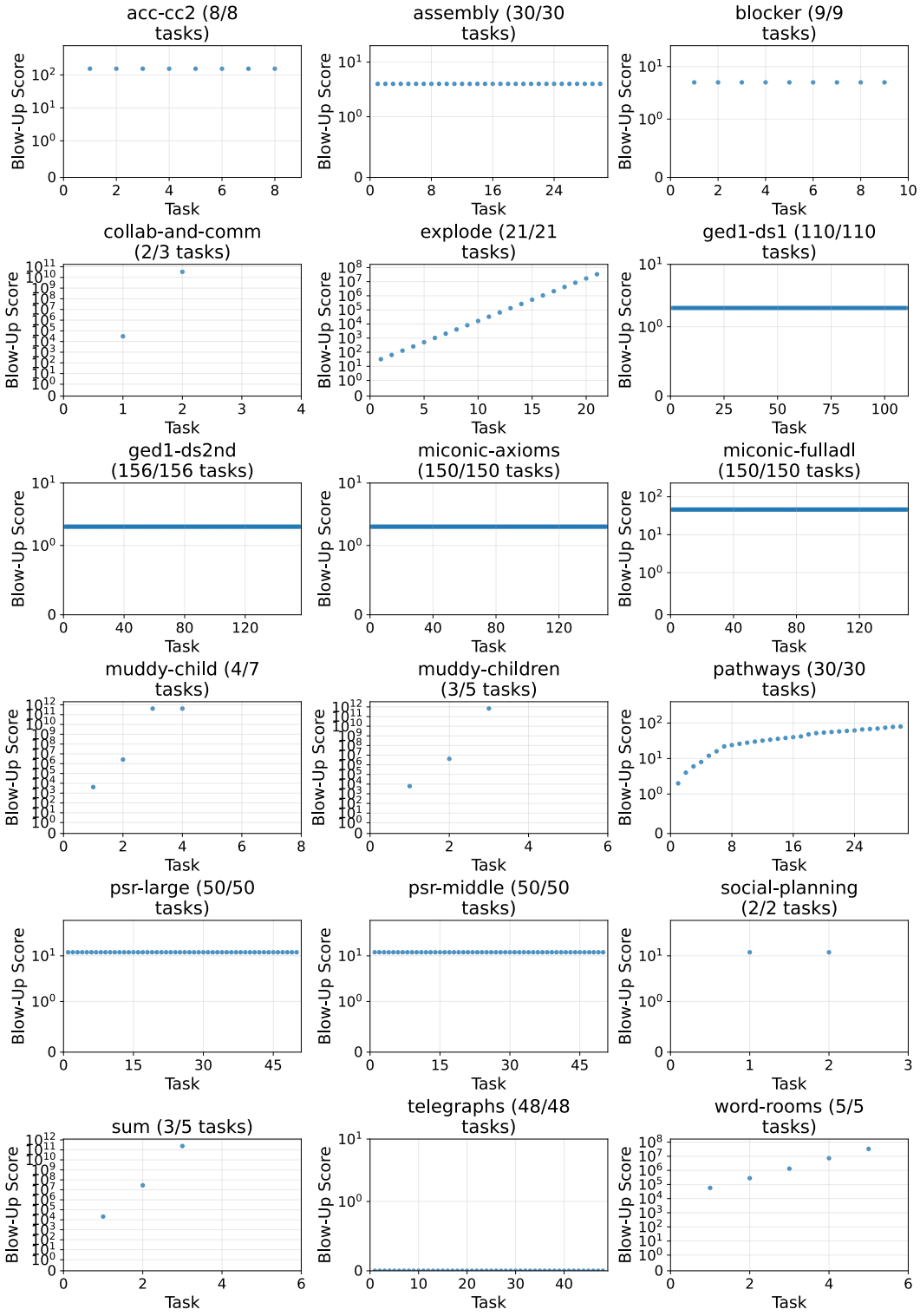

Figure 3: Visualization showing the total blow-up score distribution for all instances of the benchmark domains. The total blow-up score for a given instance is comprised of the accumulated score for all operator precondition, effect condition and goal condition formulas.

test set of instances. In our work, we forgo this because we wish to remain domain-independent.

The threshold determines a trade-off. If it is too low, the hybrid approach degenerates to $T^{\{\vee\}}$ and introduces axioms for conditions that would have been cheap to expand. If it is too high, it degenerates to $T^{\varnothing}$ and the blow-up remains. For $\tau = 10$, the running example from Example 3.3.1.1 with $n = 3$ (score $2^3 = 8$) is still expanded, whereas $n = 4$ (score 16) is refactored. We consider this to be a reasonable default value for most domains, because it catches most instances that contain a large CNF-like condition, while leaving simple disjunctions untouched. Accordingly, our results in Table 2 and Table 3 use this threshold, as a proof of concept for selective refactoring rather than as a tuned parameter.

Because the hybrid approach is practically a combination of $T^{\varnothing}$ and $T^{\{\vee\}}$, we focus our discussion on these two configurations in the following sections about search performance. This allows us to keep discussions concise and conceptually relevant, since $T^{\{\vee\}}_{\text{hybrid}}$'s results can be inferred from the results of $T^{\varnothing}$ and $T^{\{\vee\}}$. However, the interested reader may find search results for the hybrid strategy in Appendix A.

5.4 Search

Until now we have evaluated the translator implementations independently of the subsequent search. While the previous experiments show that axiom refactoring can substantially reduce the task representation size and improve translation coverage for several domains, the planner's search performance on the reformulated tasks may differ from its performance on the original task. Consequently, the goal of the following experiments is to show the impact that axiom refactoring has on the planner's search phase.

5.4.1 Expansions and Generations

Since all three implementations preserve the transition semantics of the original planning task, they induce equivalent transition systems over the primary variables and have the same optimal plan costs. Consequently, blind A^* search expands the same number of states up until the final f-layer. In the last layer the number of expansions depends on the order in which operators are expanded, which can be different for different translator implementations. This is because the resulting set of operators after translation is not identical as we have seen in Section 3.3.1. For A^* with the blind heuristic this only resulted in negligible differences in the number of nodes expanded and generated between translator implementations.

For LAMA-First, however, we observe large differences between translator implementations with respect to the number of expanded and generated nodes. Interestingly, we see

the lowest number of expansions with the $T^{\{\vee,\wedge\}}$ configuration for several domains. We attribute this partly to the effect of axiom refactoring on operator ordering. However, the complexity of LAMA-First makes it difficult to pinpoint any single source of this difference in search behaviour. Correspondingly, we also observe slightly different initial heuristic values for these implementations.

5.4.2 Search Runtime

We first consider optimal planning with A^* and the blind heuristic. Looking at runtime, we can see from Figure 4 that for all tasks solved by all three translator implementations $T^{\varnothing}$ is faster than both $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$, and $T^{\{\vee\}}$ is faster than $T^{\{\vee,\wedge\}}$. For difficult instances, this results in the slower of the two configurations in each pairing failing to solve tasks, which

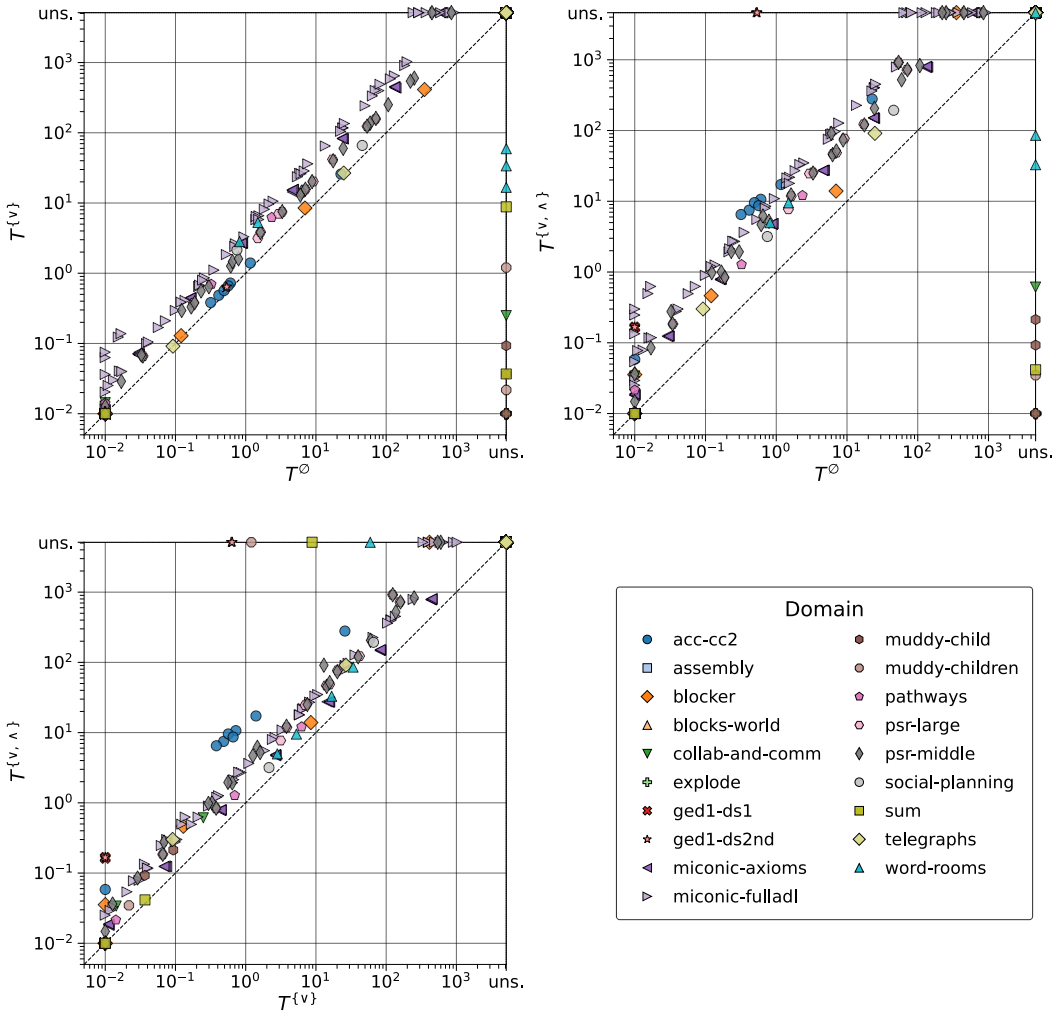

Figure 4: Pairwise search runtime comparison for A^* with the blind heuristic for the translator configurations $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ within the resource constraints detailed in Section 5.1. Data-points on the top or right edge marked with "uns." correspond to tasks that remained unsolved for the corresponding other algorithm.

the other solved. Furthermore, some instances which $T^{\varnothing}$ was not able to translate were solved by configurations with axiom refactoring. This result implies a trade-off between translation coverage and search efficiency.

We also observe that the runtime penalty introduced by axiom refactoring appears consistent with an approximately constant-factor overhead as indicated by the visual trend of the data running parallel to the diagonal. This result suggests that axiom refactoring with Fast Downward's search implementation introduces computational cost for every explored state. This effect is especially noticeable with the blind heuristic, where even a marginal runtime overhead can considerably impact overall search time due to the heuristic being uninformed with respect to the state space.

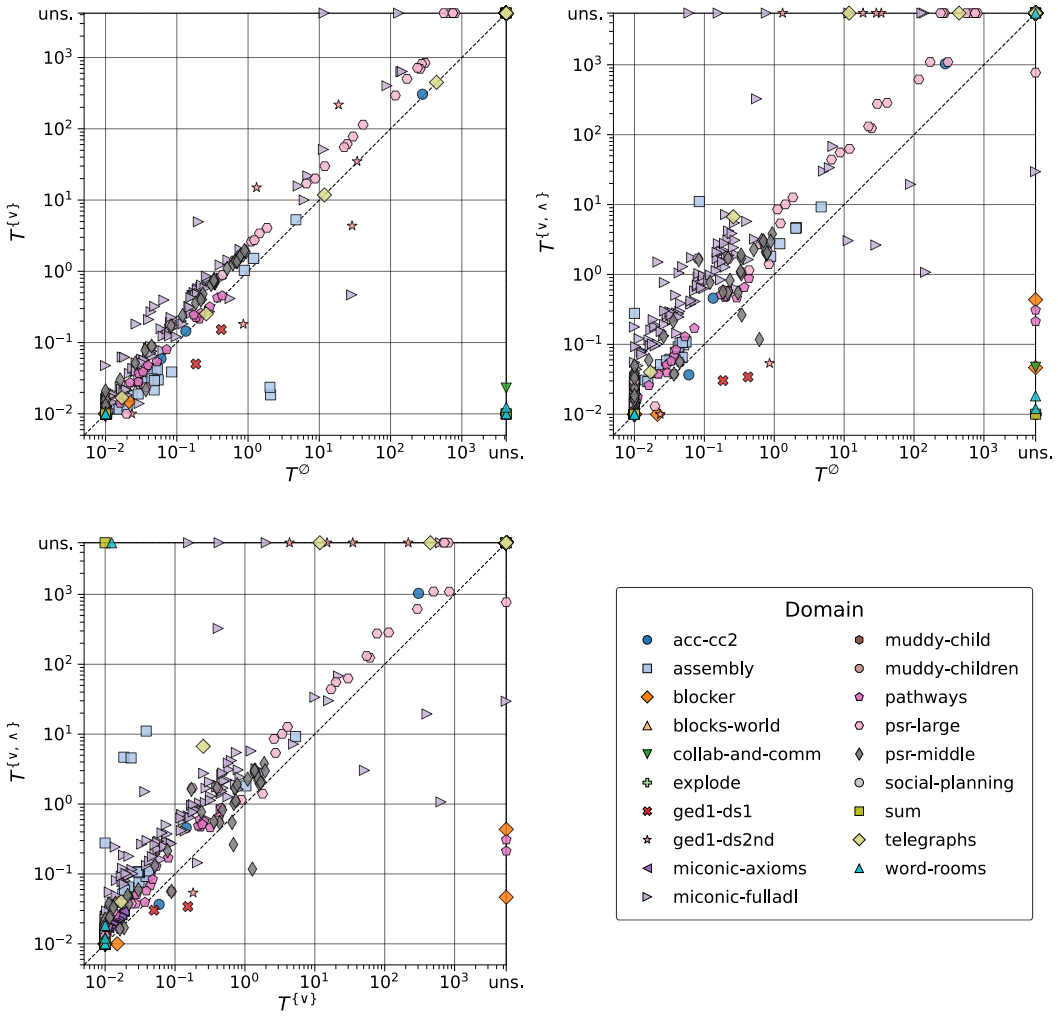

Figure 5: Pairwise search runtime comparison with LAMA-First for the translator configurations $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ within the resource constraints detailed in Section 5.1. Data-points on the top or right edge marked with "uns." correspond to tasks that remained unsolved for the corresponding other algorithm.

For satisficing planning with LAMA-First the effect on search runtime is less obvious. Figure 5 shows that for the majority of domains $T^{\varnothing}$ is the fastest, followed by $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$. However, we observe more scatter with respect to the runtime values. This is due to axiom refactoring affecting the number of expansions and generations as previously discussed (see Section 5.4.1).

In short, we conclude that higher levels of axiom refactoring slow down search with the current implementation. We believe this to be due to increased computations for evaluating axioms. Our results suggest a trade-off between translator coverage and runtime. Therefore, whether axiom refactoring provides a planning performance benefit depends on whether translation or search currently dominates the sum of translation runtime and search runtime.

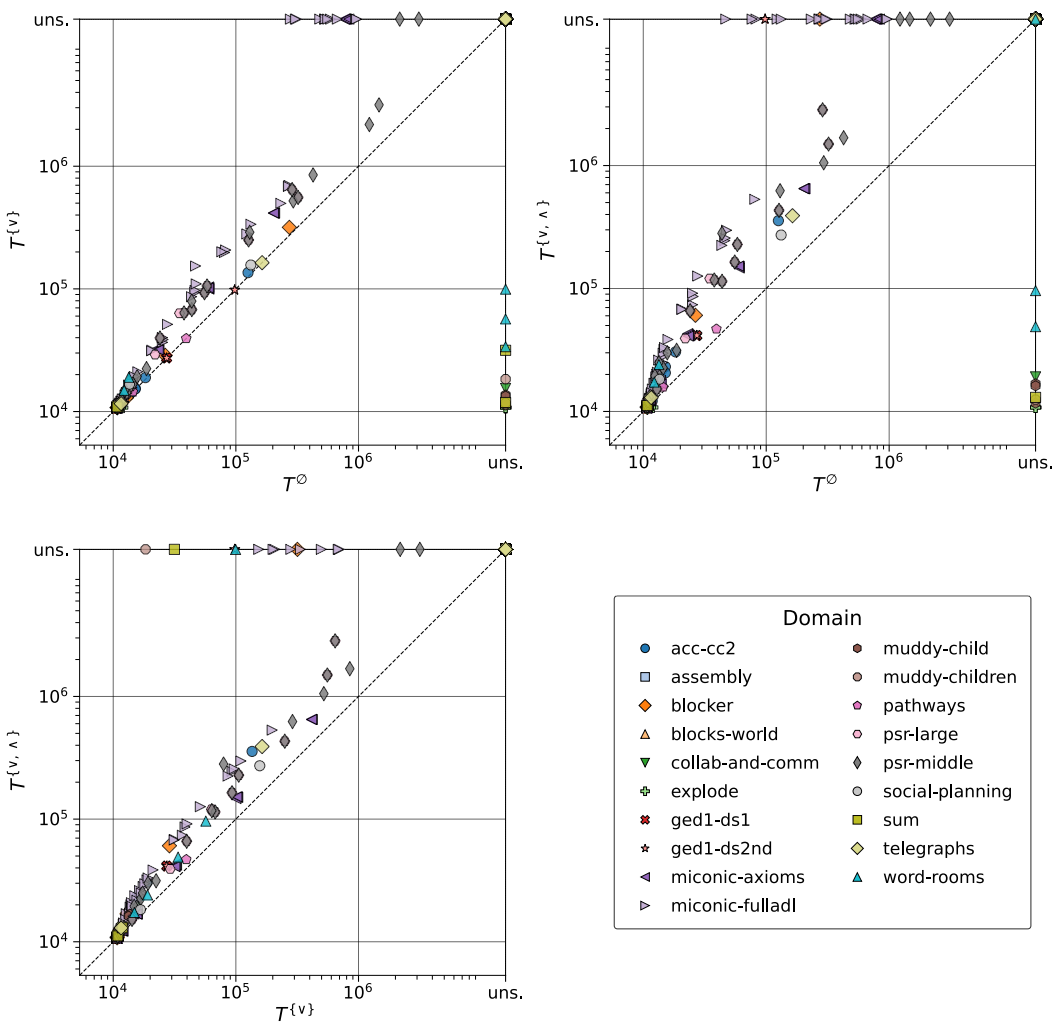

Figure 6: Per-instance comparison of total memory usage during search for the three translator configurations $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ using the A^* algorithm with the blind heuristic. Unsolved instances for one or both implementations are placed on the grid-line marked with "uns.".

5.4.3 Memory

In addition to the computational cost incurred by axiom refactoring during search, memory usage may also be affected. Figure 6 shows that, with A^*-Blind, instances with a higher number of axioms also require more memory during search. This is due to the storage of derived variables increasing the size of each state and thus making search more memory-intensive.

Furthermore, instances that could not be translated by $T^{\{\varnothing\}}$ and were successfully translated using axiom refactoring, exhibit low memory usage with $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$.

With LAMA-First, we see more variance in the memory usage due to differences in the number of generations and expansions (see Figure 7). However, the previous observations hold.

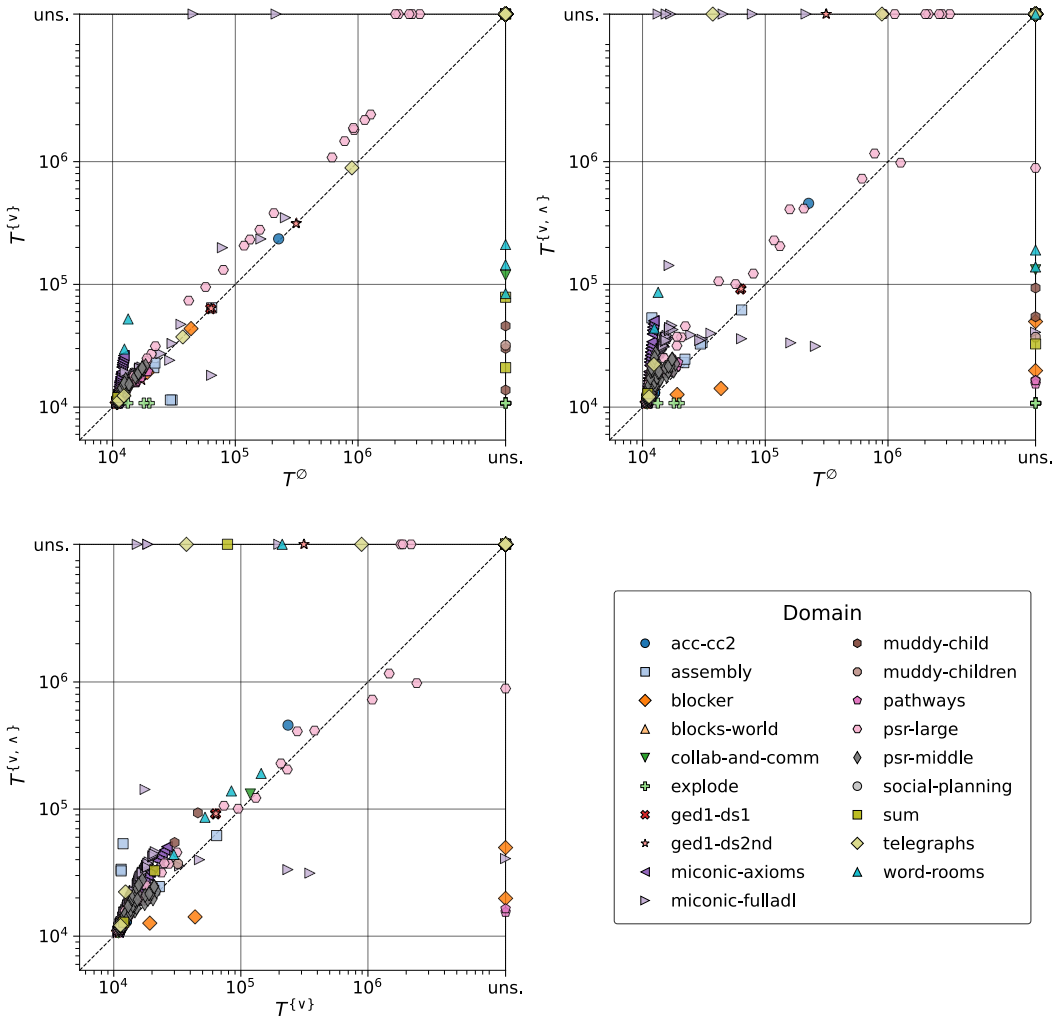

Figure 7: Per-instance comparison of total memory usage during search for the three translator configurations $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ using LAMA-First. Unsolved instances for one or both implementations are placed on the grid-line marked with "uns.".

In summary, we conclude that axiom refactoring results in a planning performance overhead with respect to memory usage and computations. Together, the memory and runtime penalty explain why search with the translator implementations $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ achieve lower overall coverage than $T^{\varnothing}$ on domains not affected by an exponential condition formula blow-up. Among these, especially domains with a large number of axioms are affected, such as the PSR-Large, PSR-Middle, and Miconic-Axioms domains. Instances affected by a severe blow-up, however, cannot be translated without axiom refactoring.

5.5 State Storage Optimization

In all experiments presented thus far, Fast Downward's search module has remained entirely unchanged from its current implementation. Furthermore, we have observed that axiom refactoring can increase the memory required during search while simultaneously resulting in elevated runtimes. In Section 4.3, we described how modifying the internal representation of state storage may help mitigate the additional memory overhead introduced by the derived variables resulting from axiom refactoring. In this section, we present the results obtained from our experiments with a modified state registry. More precisely, we present two implementations that differ in how radically they exclude derived variables from the state registry. We first evaluate a strategy that excludes derived variables from the state registry entirely, then a more conservative strategy that retains only the derived variables needed after a state has been generated.

As previously mentioned, for these experiments we consider an eager search variation of LAMA-First without landmark heuristics in addition to A^*-Blind and LAMA-First. To keep the discussion concise, visualizations for the satisficing search algorithms are not explicitly presented here. Instead, the corresponding results may be viewed in Appendix B.

5.5.1 Complete Axiom Omission

Our first implementation completely removes all derived variables from the state registry. Consequently, whenever the value of a derived variable needs to be accessed, the values of all derived variables are re-evaluated based on the values of the primary variables. An exception to this is if the evaluated unpacked values have already been cached for a given state object which has not gone out of scope.

Figure 8 shows how the complete omission strategy compares to the current implementation, when paired with different axiom refactoring configurations, with respect to memory usage for the A^*-Blind algorithm. We can see that the strategy decreases the amount of memory used regardless of the axiom refactoring configuration used. Intuitively, as the number of axioms increases and storage of derived variables begins to dominate memory

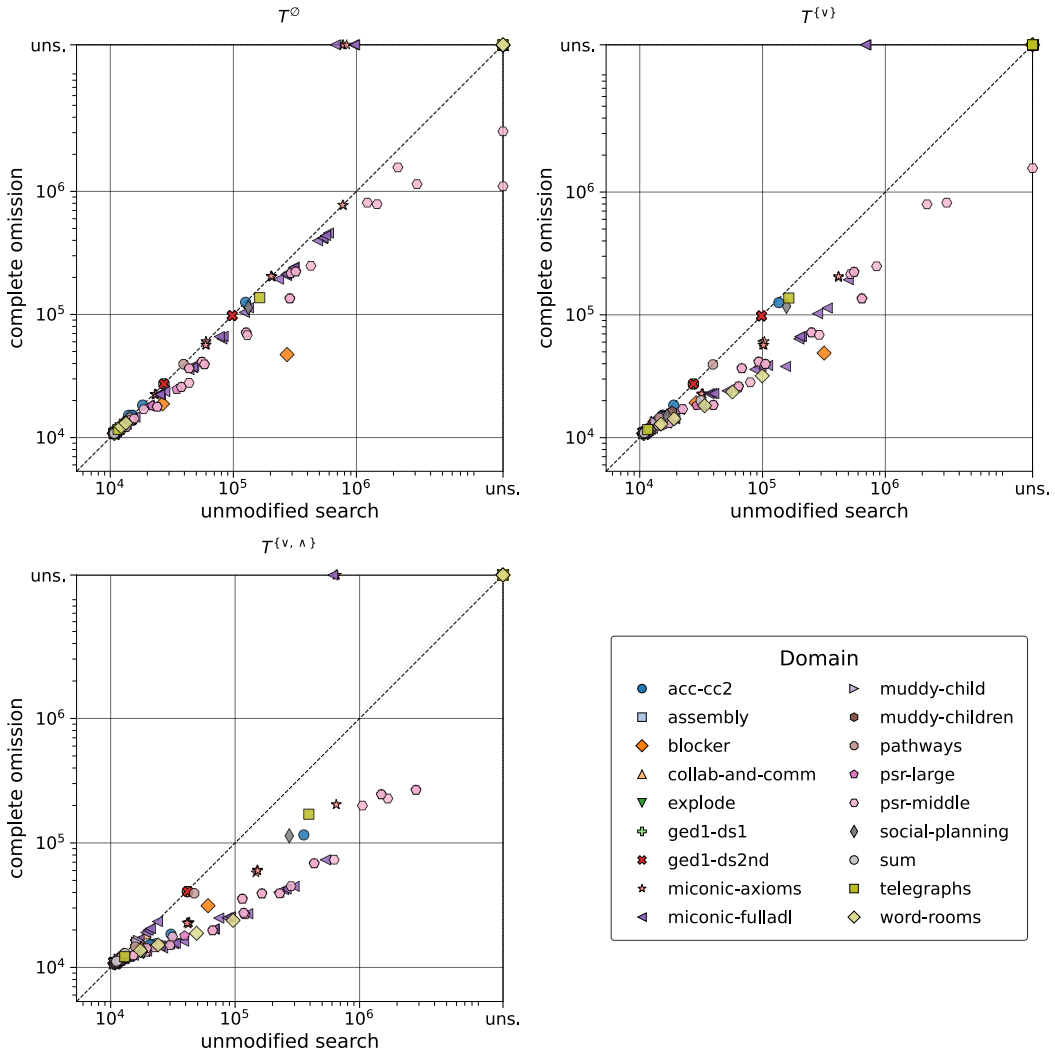

Figure 8: Per-instance comparison of total memory usage in kilobytes for the current search implementation (x-axis), where all derived and primary variables are stored in the state registry, against a modified search implementation (y-axis), where only primary variables are stored in the state registry. The search algorithm used for this comparison is A^* with the blind heuristic. Tasks that could not be solved are represented on the grid-lines labeled with the "uns." label. The $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ labels indicate the translator configuration used for both search implementations in the respective sub-plot.

usage, our optimization strategy becomes more effective. On the other hand, we can see that for all three axiom refactoring levels several tasks cannot be solved by the modified search implementation within the time limit.

Furthermore, Figure 9 shows that there is only very minor runtime differences between two algorithms. Nevertheless, the modified search algorithm is slightly slower than the current implementation, which we believe to be induced by axiom evaluation computations and by unpacking states.

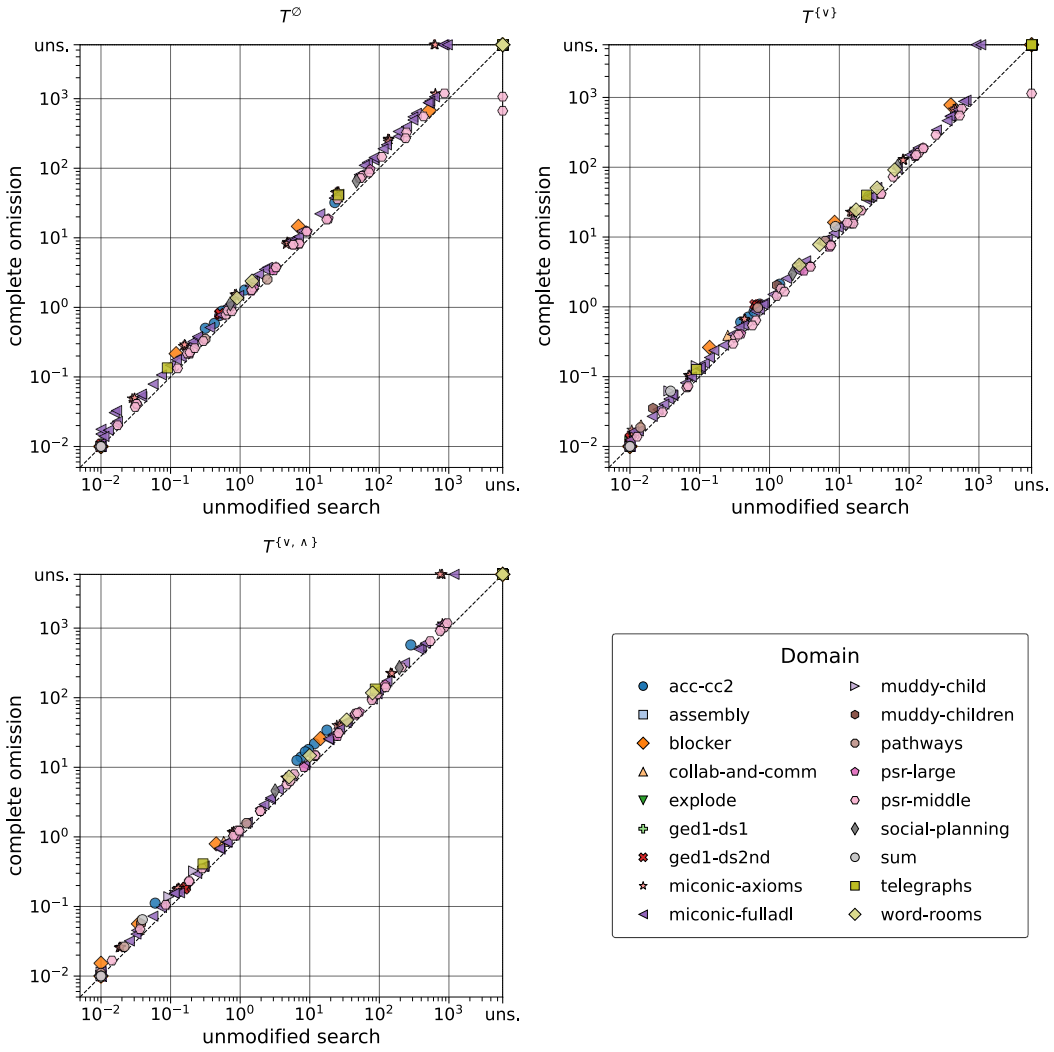

Figure 9: Per-instance comparison of search time in seconds for the current search implementation (x-axis), where all derived and primary variables are stored in the state registry, against a modified search implementation (y-axis), where only primary variables are stored in the state registry. The search algorithm used for this comparison is A^*-Blind. Tasks that could not be solved are represented on the grid-lines labeled with the "uns." label. No outliers have been excluded from this visualization. The $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ labels indicate the translator configuration used for both search implementations in the respective sub-plot.

5.5.2 Partial Axiom Omission

In our second implementation we keep derived variables in the state registry only if they need to be accessed again after generation. This includes derived variables in goal conditions, operator preconditions and effect conditions. With this approach we can ensure that once axioms have been evaluated for a state and written back to the state registry, subsequent lookups of that state do not require axioms to be re-evaluated.

Figure 10 demonstrates that, with this implementation, we achieve a lower memory usage for almost all tasks. Note that we even see an improvement for tasks with axioms prior to

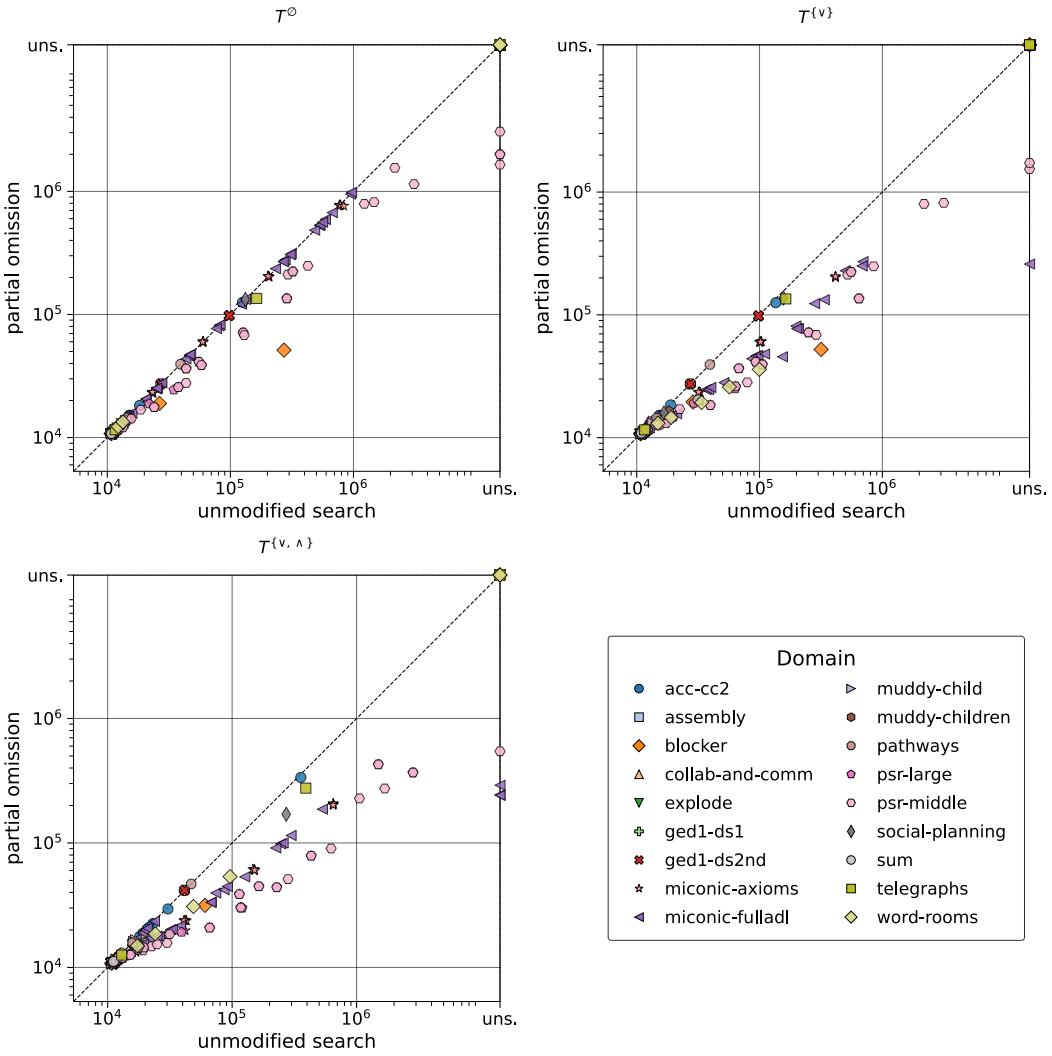

Figure 10: Per-instance comparison of total memory usage in kilobytes for the current search implementation (x-axis), where all derived and primary variables are stored in the state registry, against a modified search implementation (y-axis), where only primary variables are stored in the state registry. The search algorithm used for this comparison is A^*-Blind. Tasks that could not be solved are represented on the grid-lines labeled with the "uns." label. No outliers have been excluded from this visualization. The $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ labels indicate the translator configuration used for both search implementations in the respective sub-plot.

introducing additional axioms with axiom refactoring. Furthermore, the modified search can now solve tasks that could not be solved by the current implementation.

Finally, Figure 11 show that the search time is very similar for the current version and the modified version. This is consistent over all three translator configurations and both planning paradigms.

Overall, the strategy of partially omitting derived variables from the state registry provides the memory benefits associated with having to store fewer variables per state, as well as

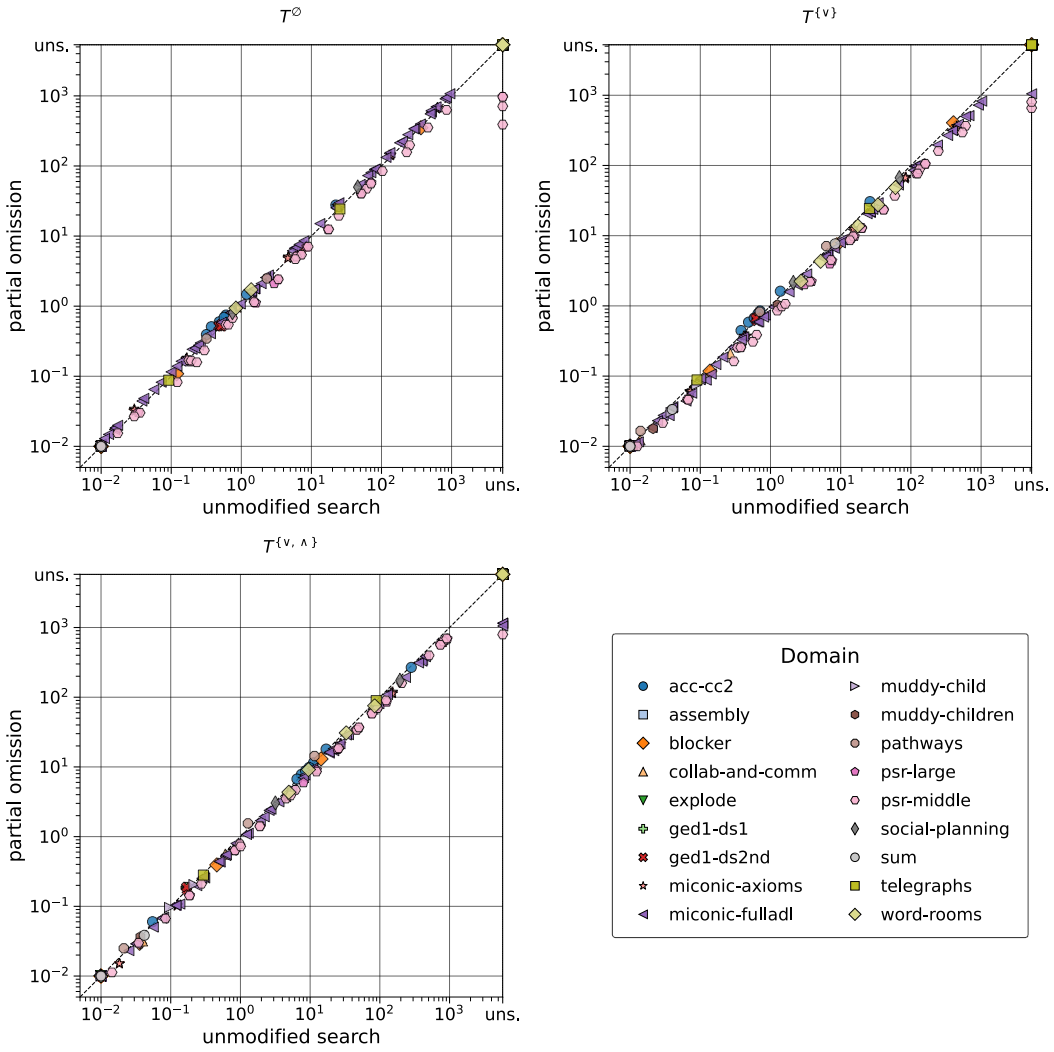

Figure 11: Per-instance comparison of total search time in seconds for the current search implementation (x-axis), where all derived and primary variables are stored in the state registry, against a modified search implementation (y-axis), where only primary variables are stored in the state registry. The search algorithm used for this comparison is A^*-Blind. Tasks that could not be solved are represented on the grid-lines labeled with the "uns." label. No outliers have been excluded from this visualization. The $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ labels indicate the translator configuration used for both search implementations in the respective sub-plot.

search times approximately matching the current implementation. We conclude that this implementation exhibits the best overall performance with respect to memory and runtime.

5.5.3 Axiom Omission for Satisficing Planning

The results for the satisficing search configurations in Appendix B show that the effects observed for A^*-Blind largely generalize to these algorithms. The memory usage for greedy-search with both partial and complete axiom omission is lower or similar to the memory usage of the current implementation. Furthermore, data points stay close to the diagonal in the runtime comparison as for A^*.

The results for LAMA-First show a less clear picture. The memory reduction is still visible, particularly for the more aggressive axiom refactoring configurations, but the runtime comparison exhibits greater variation than the corresponding results for A^*-Blind. This is especially the case for $T^{\{\wedge,\vee\}}$, where several instances substantially stray from the diagonal or cannot be solved within the resource limits.

The larger variation is consistent with the more complex behaviour of LAMA-First observed earlier in this thesis, where changes to the translated task may affect heuristic evaluation and node ordering. Additionally, the complete omission strategy requires that derived variables are recomputed when accessed. This can introduce a more noticeable computational cost for this configuration. On the other hand, the partial omission strategy avoids this cost to an extent by allowing the goal check and successor generation to read the derived variables from the state registry. And consequently exhibits a more stable runtime behaviour while retaining most of the memory savings. However, states still need to be unpacked for the heuristic computation, leading to less memory usage improvement than for A^*.

6
Discussion

The preceding experiments demonstrate that axiom refactoring can substantially impact both search and translation in planning. In the following, we discuss the implications and limitations of our work and its integration into the Fast Downward planning system.

6.1 Axiom Refactoring

We have seen that transforming disjunctive conditions using axiom refactoring allows us to normalize tasks, which were previously infeasible to normalize within resource limits. In particular, it avoids a computationally expensive DNF-conversion of condition formulas at the cost of an increase in state representation size.

Of the presented normalization procedures, a hybrid approach, which selectively refactors conditions, achieved the highest absolute translation coverage and lowest mean memory usage. However, the hybrid approach introduces complexity through the choice of a threshold parameter, which may be undesirable for application in a scientific context, as researchers must account for different normalization procedures within the same instance or between instances of the same domain.

In practice, most users would likely want to use the translator configuration where only disjunctive condition sub-formulas are refactored using axioms, as this version directly addresses the exponential blow-up problem while keeping the number of derived variables constrained to the number of disjunctions in the original planning task.

Due in part to the prevalence of Fast Downward's translator component in contexts such as the International Planning Competitions (IPC), many classical domains are manually modeled in such a way that the current version of the translator is able to ground them. Consequently, the number of domains where an exponential blow-up prevents grounding in common benchmarks is rather limited. Furthermore, domains that fail to be translated may not even be considered for study because researchers cannot study search behaviours

without getting through translation first. Given our approach, domains that were manually normalized previously, may be modeled more concisely by depending on the translator to normalize tasks instead.

A limitation of axiom refactoring is that, for tasks that don't otherwise include derived variables, many heuristics do not currently support axioms. This is particularly limiting in optimal planning because only few admissible heuristics support axioms. As progress is made towards supporting axioms for more heuristics, we expect our method to gain more wide-spread applicability.

6.2 Search Optimization

We discussed the effect of axiom refactoring on search performance with the current search implementation as well as two alternative implementations which partially or completely eliminated derived variables from the state registry. We demonstrated that when tasks are axiom refactored with the current implementation, the computation time and memory usage are increased. However, we showed that reducing the number of derived variables stored in the state registry improves search performance with respect to memory usage without significantly impacting runtime results.

The axiom omission strategies may be limited in their effectiveness for certain heuristics, even if they were to support axioms. Further study is required to determine how it impacts heuristics based on landmarks [19], [20] or pattern databases [21], [22], for example, as they may need to reason about when a derived variable becomes true.

An unexpected finding is that, when using LAMA-First, the number of expansions observed for the $T^{\{\vee,\wedge\}}$ configuration, where all conjunctions and disjunctions are refactored, differed from the other translator implementations for several tasks. This aspect is challenging to explain fully and requires further study. However, we believe that transforming the tasks may have resulted in changes in tie-breaking order among equal-cost nodes, which affects node selection. We did not isolate the cause experimentally.

6.3 Practical Discussion of the Implementation

Fast Downward's software architecture (version 24.06) posed certain challenges during the implementation of the optimized search. We include these here (without providing background) because they may be of interest to readers already familiar with Fast Downward's software implementation.

The biggest challenge is that states reference an abstraction of the planning task called an *abstract task*. Certain information, like whether a state variable should be registered in the state registry is constant for the given planning task, and therefore should not be recomputed

for every state. However, the computation relies on information only available to an abstract task instantiated as a *task proxy*. This requires us to use a work-around where the variable mapping is statically stored apart from the task, which can be conceptually confusing. This work-around is also used at the time of this thesis for accessing the causal graph.

Furthermore, since Fast Downward optimizes the number of bits required to store variables, and because derived variables and primary variables are not sorted by their axiom layer, the access patterns induced by the mapping of variable indices in the state against the indices in the state registry are not sequential. This may cause a performance overhead when compared to a more cache-aware variable order.

7

Future Work

Our thesis provides a first analysis of the methods presented. Several areas remain which could be the basis of further research.

Our approach already included some minor optimizations such as the following: When a disjunctive condition would be refactored that also appears in a different condition formula of the same planning task that has already been processed, then that condition can be represented by the same derived variable introduced for the first occurrence of that formula. Our implementation already implements a naive check for this case, which only affects a single domain in our benchmark set. However, we believe that more in-depth axiom elimination techniques may provide more significant benefit, especially for higher levels of axiom refactoring.

Future work may also explore further axiom refactoring approaches apart from the ones presented here. In this thesis we focus on $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ because they are conceptually simple, and $T^{\{\vee,\wedge\}}$ serves well as an extreme case to explore how refactoring changes search behaviour. However, our hybrid version showed that other approaches may provide even greater absolute performance. For example, it may be useful to extend our hybrid approach from Section 4.2 such that it refactors only specific sub-formulas within a condition instead of going condition-by-condition.

Our discussion of the effect of axiom refactoring on search focused on the search configurations A^*-Blind and LAMA-First, as well as the greedy search from Section 5.5. As previously mentioned, future work could expand the experiments presented here to more heuristics. Additionally, our observation that for many domains axioms form many individual strongly connected components provides motivation for exploring approaches such as selective axiom evaluation or parallel axiom evaluation. Together with axiom

refactoring these could potentially provide a performance benefit through a reduction in the computation time required by axiom evaluations.

8

Conclusion

In this thesis, we addressed an exponential blow-up that arises during Fast Downward's translation process by introducing axiom refactoring. We conducted an investigation of axiom refactoring's effects on search and translation and showed that it allows the successful translation of tasks that were previously too computationally expensive to normalize.

Our work further shows that selectively applying axiom refactoring may yield even better overall translation results. However, it comes at the cost of a less uniform translation procedure. A simpler approach that refactors disjunctive condition sub-formulas provides a more uniform alternative.

Axiom refactoring can, however, negatively affect search performance by introducing additional derived variables and increasing the cost of state representation and axiom evaluation. We therefore investigated an alternative state registry implementation that avoids storing derived variables that are not required by the search. Our results show that this optimization can partially offset the additional computational cost introduced by axiom refactoring.

Overall, our findings suggest that axiom refactoring is a practical extension to Fast Downward's capabilities. Its application is constrained by the support for axioms in the heuristics used during search. Extending axiom support to more, especially admissible, heuristics remains an open problem and would broaden where this approach can be applied.

Bibliography

[1] M. Helmert, “The Fast Downward Planning System,” *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.

[2] D. McDermott, “The 1998 AI Planning Systems Competition,” *AI Magazine*, vol. 21, no. 2, pp. 35–55, 2000.

[3] M. Helmert, “Concise Finite-Domain Representations for PDDL Planning Tasks,” *Artificial Intelligence*, vol. 173, pp. 503–535, 2009.

[4] D. Speck, R. Mattmüller, and B. Nebel, “Symbolic Top-k Planning,” in *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020)*, V. Conitzer and F. Sha, Eds., AAAI Press, 2020, pp. 9967–9974.

[5] D. Speck, M. Hecher, D. Gnad, J. K. Fichte, and A. B. Corrêa, “Counting and Reasoning with Plans,” in *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2025)*, J. Shah and Z. Kolter, Eds., AAAI Press, 2025, pp. 26688–26696.

[6] P. E. Hart, N. J. Nilsson, and B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[7] N. J. Nilsson, *Principles of Artificial Intelligence*. Morgan Kaufmann, 1980.

[8] S. Richter and M. Westphal, “The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks,” *Journal of Artificial Intelligence Research*, vol. 39, pp. 127–177, 2010.

[9] S. Thiébaux, J. Hoffmann, and B. Nebel, “In Defense of PDDL Axioms,” *Artificial Intelligence*, vol. 168, no. 1–2, pp. 38–69, 2005.

[10] C. Grundke, G. Röger, and M. Helmert, “Formal Representations of Classical Planning Domains,” in *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024)*, S. Bernardini and C. Muise, Eds., AAAI Press, 2024, pp. 239–248.

[11] G. Behnke, D. Speck, and D. Gnad, “AxSAT – Bringing Axioms to SAT Planning,” in *Logics in Artificial Intelligence*, G. Casini, B. Dundua, and T. Kutsia, Eds., Springer Nature Switzerland, 2025, pp. 77–93.

[12] D. Speck, J. Seipp, and Á. Torralba, “Symbolic Search for Cost-Optimal Planning with Expressive Model Extensions,” *Journal of Artificial Intelligence Research*, vol. 82, pp. 1349–1405, 2025.

[13] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[14] A. B. Corrêa and G. De Giacomo, “Lifted Planning: Recent Advances in Planning Using First-Order Representations,” in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, K. Larson, Ed., International Joint Conferences on Artificial Intelligence Organization, 2024, pp. 8010–8019.

[15] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[16] G. Röger and M. Helmert, “The More, the Merrier: Combining Heuristic Estimators for Satisficing Planning,” in *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, R. Brafman, H. Geffner, J. Hoffmann, and H. Kautz, Eds., AAAI Press, 2010, pp. 246–249.

[17] S. Richter and M. Helmert, “Preferred Operators and Deferred Evaluation in Satisficing Planning,” in *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, A. Gerevini, A. Howe, A. Cesta, and I. Refanidis, Eds., AAAI Press, 2009, pp. 273–280.

[18] F. Kominis and H. Geffner, “Beliefs in Multiagent Planning: From One Agent to Many,” in *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, R. Brafman, C. Domshlak, P. Haslum, and S. Zilberstein, Eds., AAAI Press, 2015, pp. 147–155.

[19] J. Hoffmann, J. Porteous, and L. Sebastia, “Ordered Landmarks in Planning,” *Journal of Artificial Intelligence Research*, vol. 22, pp. 215–278, 2004.

[20] S. Richter, M. Helmert, and M. Westphal, “Landmarks Revisited,” in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, AAAI Press, 2008, pp. 975–982.

[21] S. Edelkamp, “Planning with Pattern Databases,” in *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, A. Cesta and D. Borrajo, Eds., AAAI Press, 2001, pp. 84–90.

[22] P. Haslum, A. Botea, M. Helmert, B. Bonet, and S. Koenig, “Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning,” in *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, AAAI Press, 2007, pp. 1007–1012.

Declaration of AI Usage

Artificial intelligence tools were used in a limited and supportive capacity during the preparation of this thesis. In particular, AI assistance was used to generate and refine code for plotting and data visualisation, as well as for spelling, grammar, and language checking. AI tools were not used to generate the substantive content, analysis, or conclusions of the thesis. All final text, interpretations, and results were reviewed and verified by the author.

Models	**Uses**	**Parts affected**
ChatGPT, Claude	Spelling and grammar correction	Entire paper
Claude	Plotting code adjustments	Figures
Claude	Assistance with Typst® template layout adjustments	Entire paper

A

Additional Results for Hybrid Axiom Refactoring

This appendix provides additional results for the translation configuration $T_{\text{hybrid}}^{\{\vee\}}$. The following table provides an overview of the additional results in this appendix.

Search algorithm	Memory	Search runtime
A^*-Blind	Figure 14	Figure 12
LAMA-First	Figure 15	Figure 13

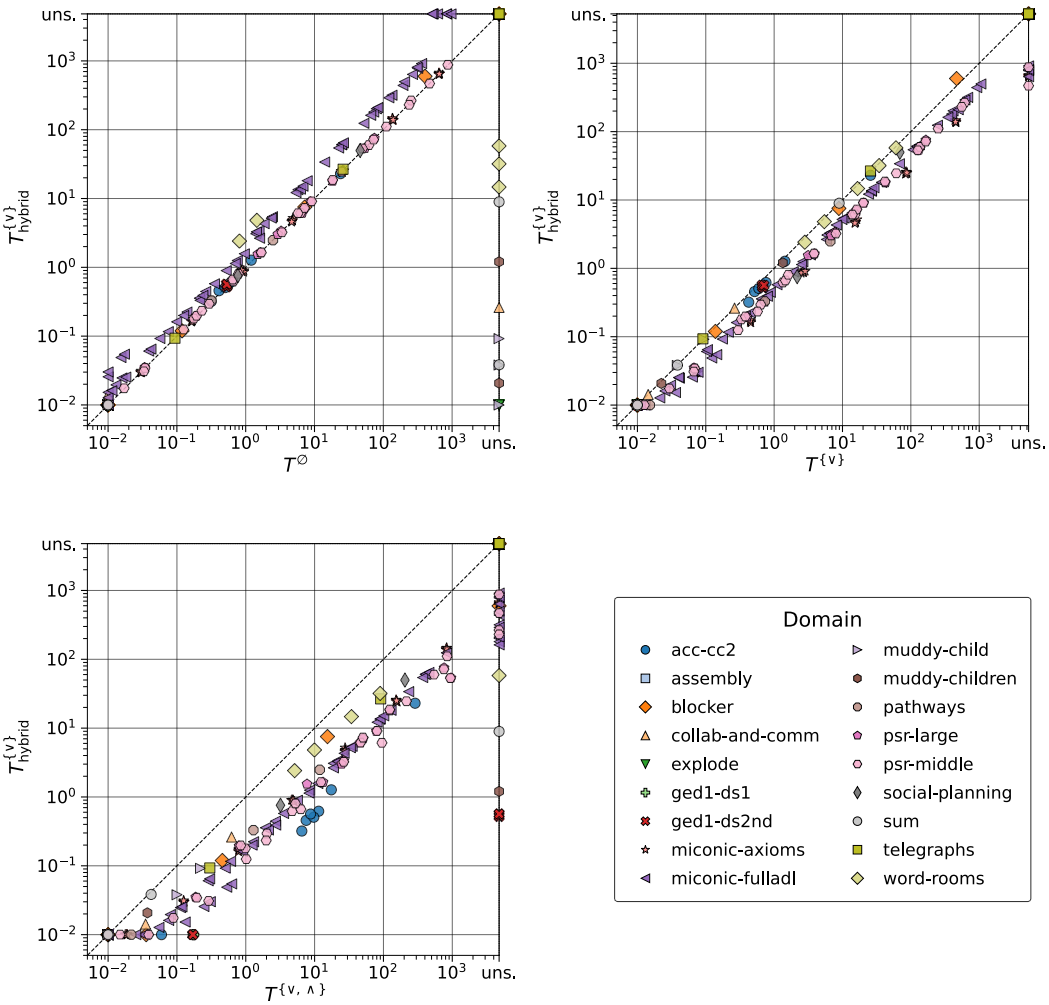

Figure 12: Visualization showing a search runtime comparison for A^* with the blind heuristic for the translator configurations $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ against $T^{\{\vee\}}_{\text{hybrid}}$. Data-points on the top or right edge marked with "uns." correspond to tasks that remained unsolved for the corresponding other algorithm.

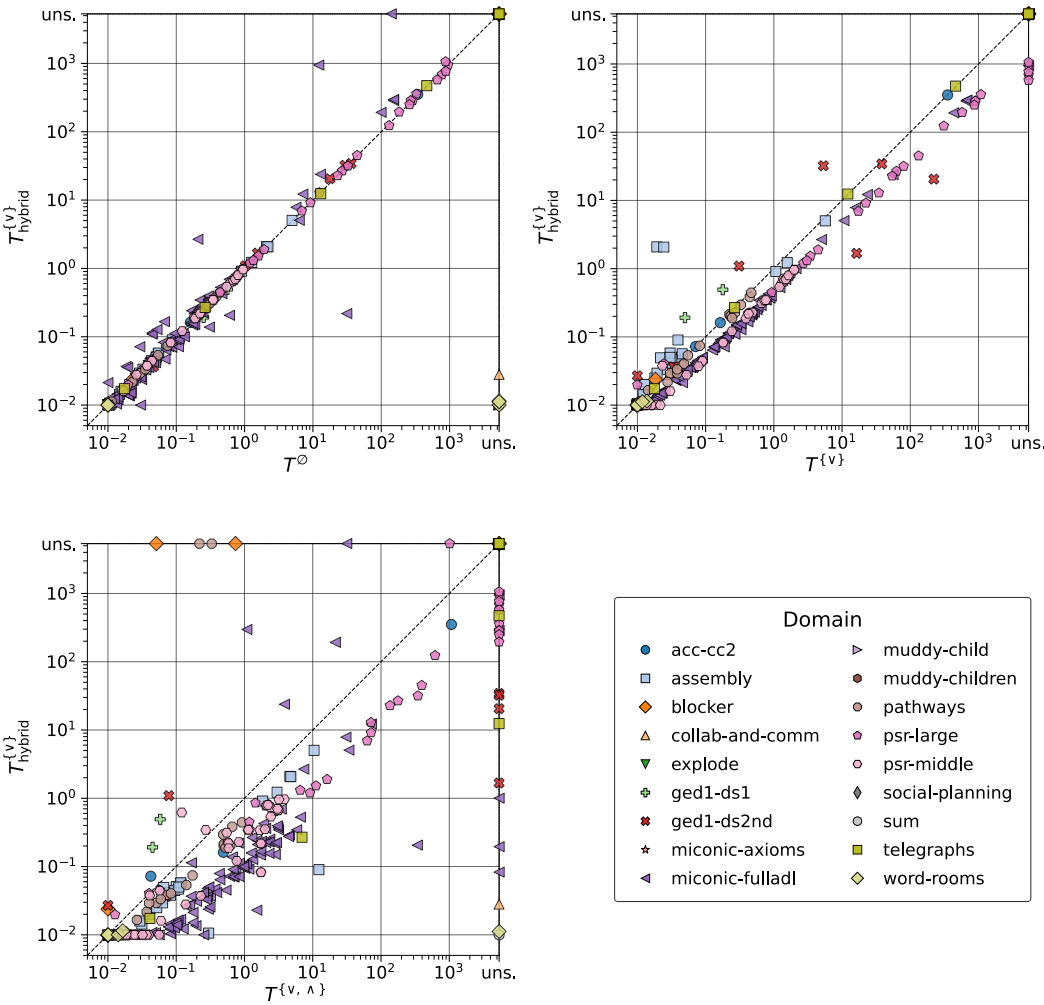

Figure 13: Visualization showing a search runtime comparison for LAMA-First for the translator configurations $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ against $T^{\{\vee\}}_{\text{hybrid}}$. Data-points on the top or right edge marked with "uns." correspond to tasks that remained unsolved for the corresponding other algorithm.

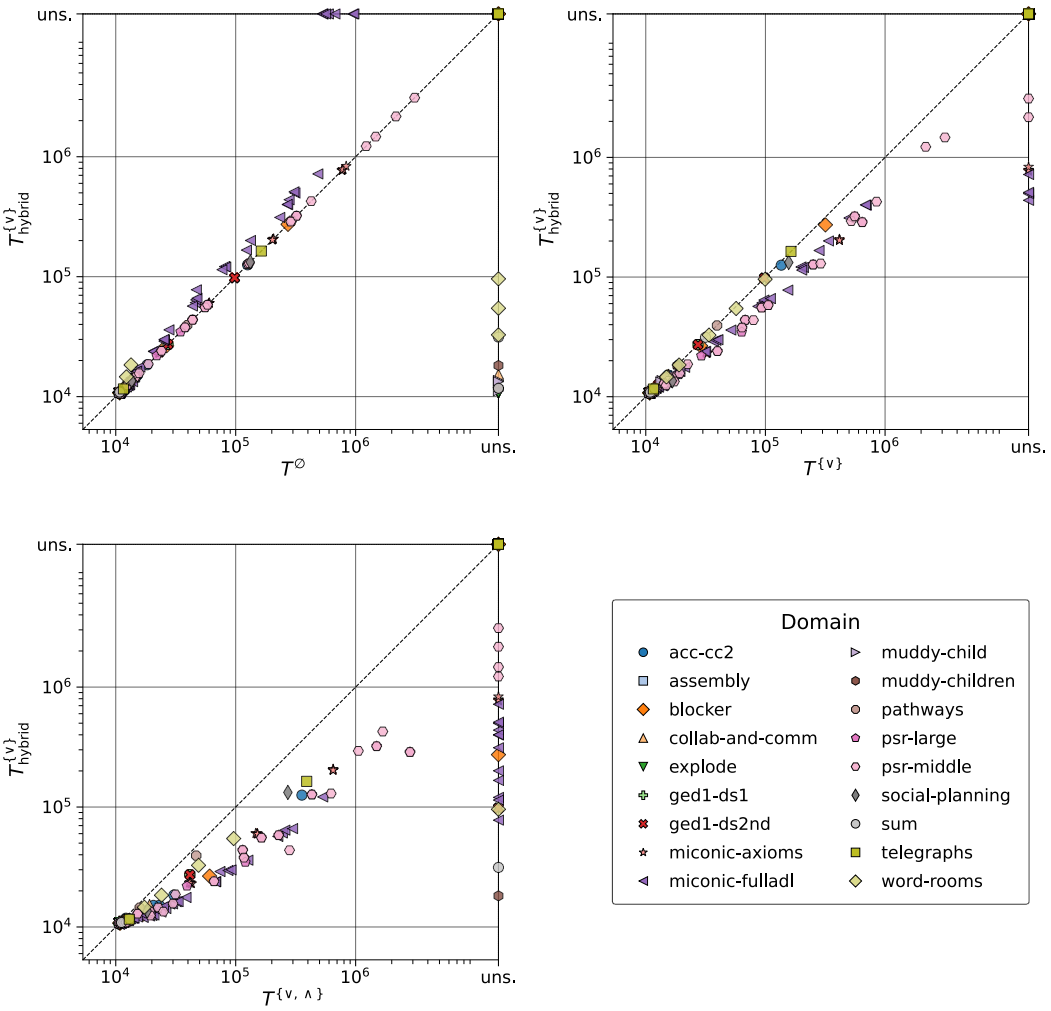

Figure 14: Visualization showing memroy usage for $A^\star$ with the blind heuristic for the translator configurations $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ against $T^{\{\vee\}}_{\text{hybrid}}$. Data-points on the top or right edge marked with "uns." correspond to tasks that remained unsolved for the corresponding other algorithm.

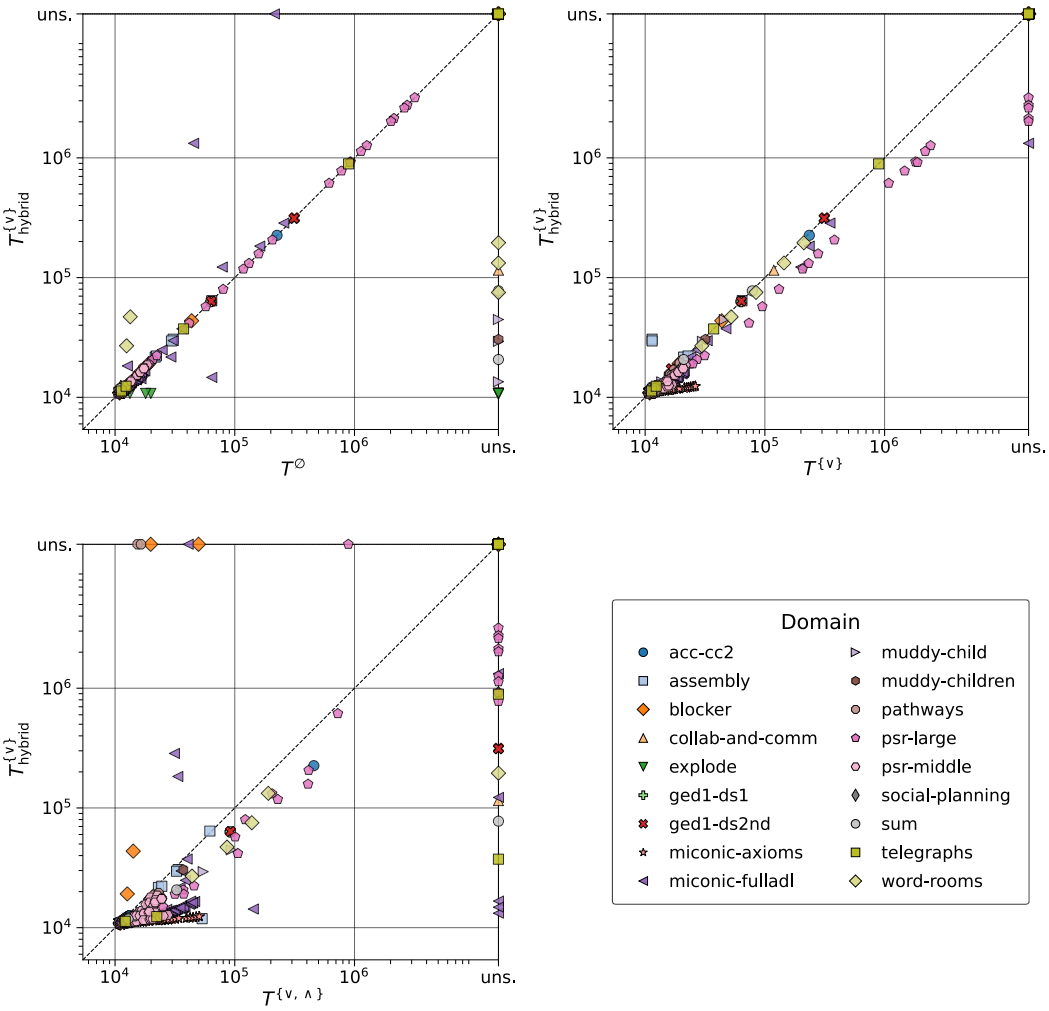

Figure 15: Visualization showing memory usage for LAMA-First for the translator configurations $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ against $T^{\{\vee\}}_{\text{hybrid}}$. Data-points on the top or right edge marked with "uns." correspond to tasks that remained unsolved for the corresponding other algorithm.

B

Additional Results for State Registry Optimization

This appendix provides additional results for the state registry optimization strategy. The following table provides an overview of the additional results in this appendix. Figures are grouped by axiom omission strategy, search algorithm, and measured resource, allowing the relevant result to be located directly.

Axiom omission	Search algorithm	Memory	Search runtime
Complete	Greedy best-first search	Figure 16	Figure 17
Complete	LAMA-First	Figure 18	Figure 19
Partial	Greedy best-first search	Figure 20	Figure 21
Partial	LAMA-First	Figure 22	Figure 23

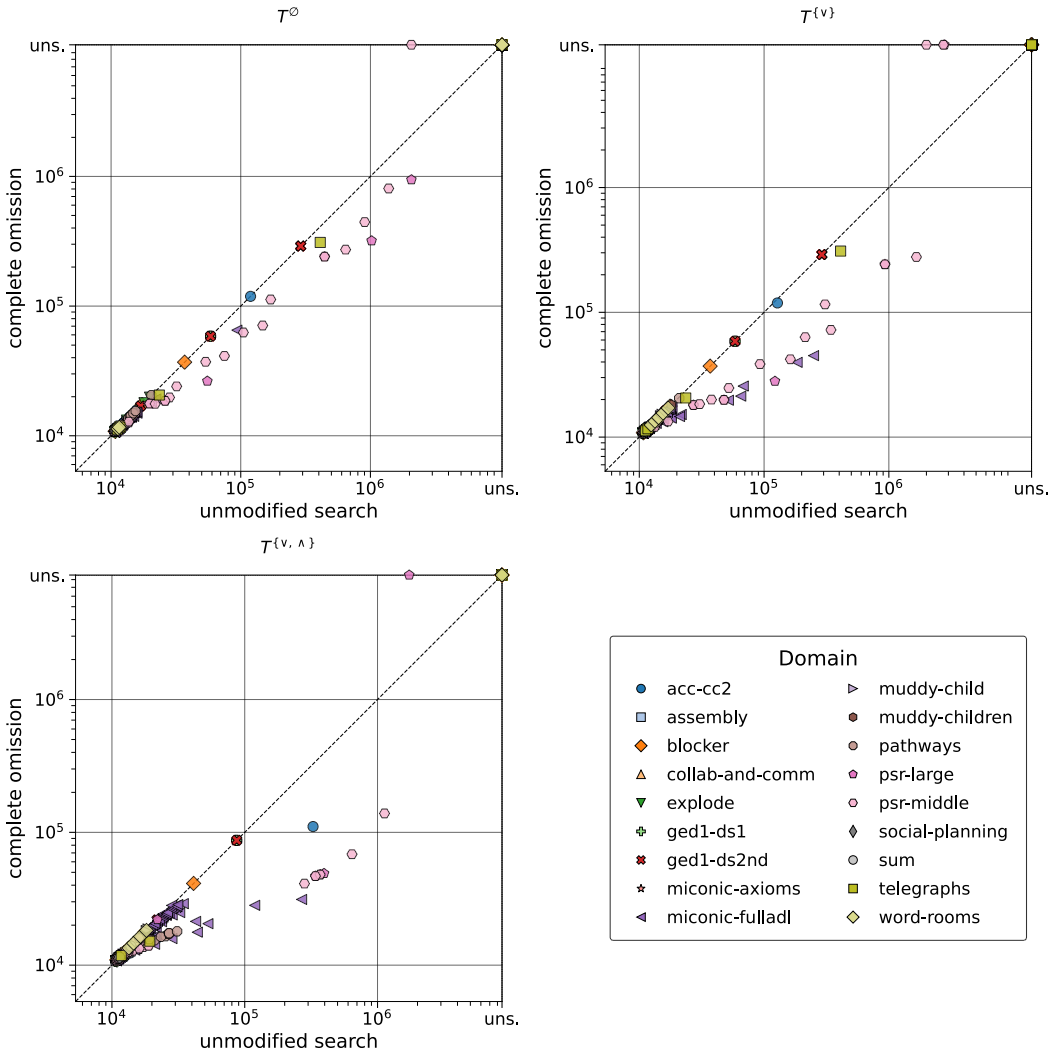

Figure 16: Per-instance comparison of total memory usage in kilobytes for the current search implementation (x-axis), where all derived and primary variables are stored in the state-registry, against a modified search implementation (y-axis), where only primary variables are stored in the state-registry. The search algorithm used for this comparison is greedy best-first search with FF and FF-only preferred operators. Tasks that could not be solved are represented on the grid-lines labeled with the "uns." label. No outliers have been excluded from this visualization. The $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ labels indicate the translator configuration used for both search implementations in the respective sub-plot.

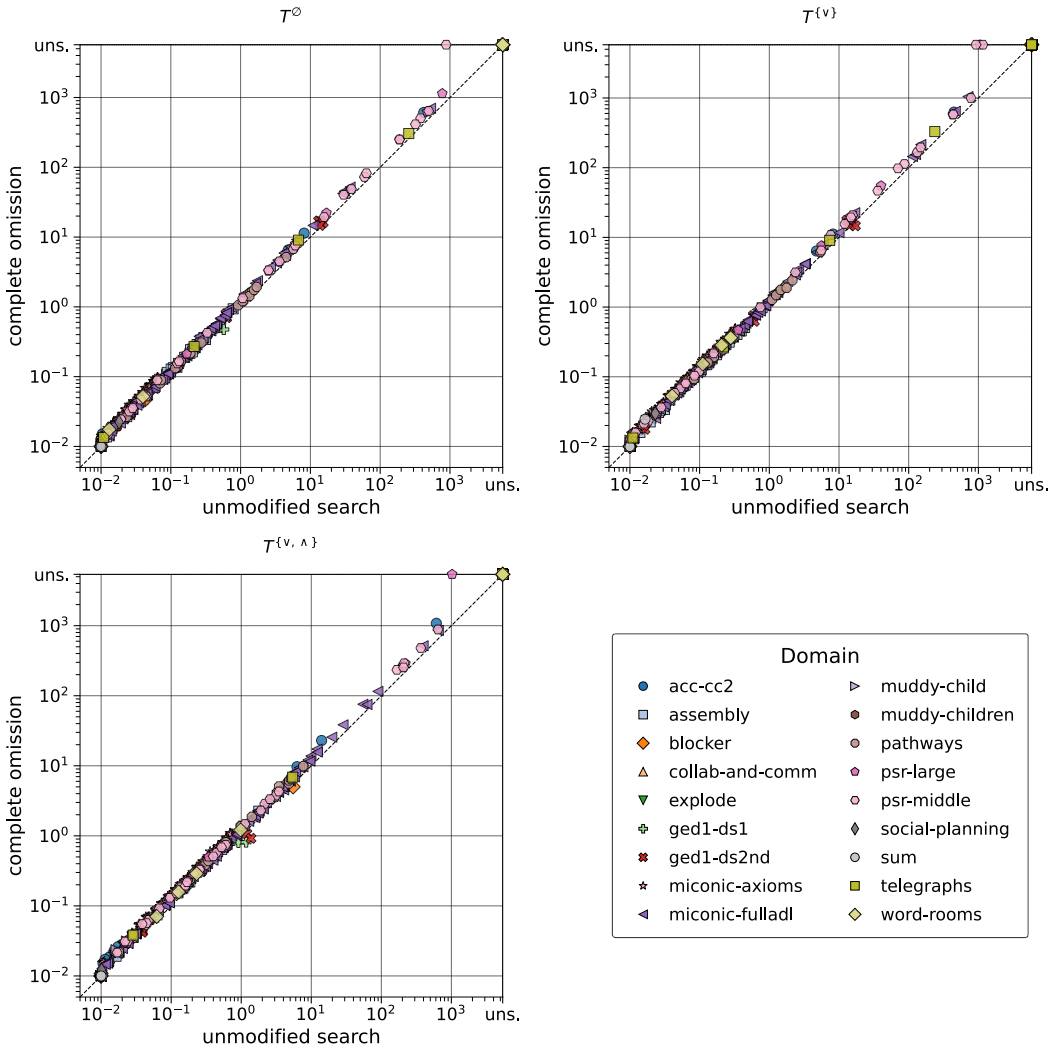

Figure 17: Per-instance comparison of search time in seconds for the current search implementation (x-axis), where all derived and primary variables are stored in the state-registry, against a modified search implementation (y-axis), where only primary variables are stored in the state-registry. The search algorithm used for this comparison is greedy best-first search with FF and FF-only preferred operators. Tasks that could not be solved are represented on the grid-lines labeled with the "uns." label. No outliers have been excluded from this visualization. The $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ labels indicate the translator configuration used for both search implementations in the respective sub-plot.

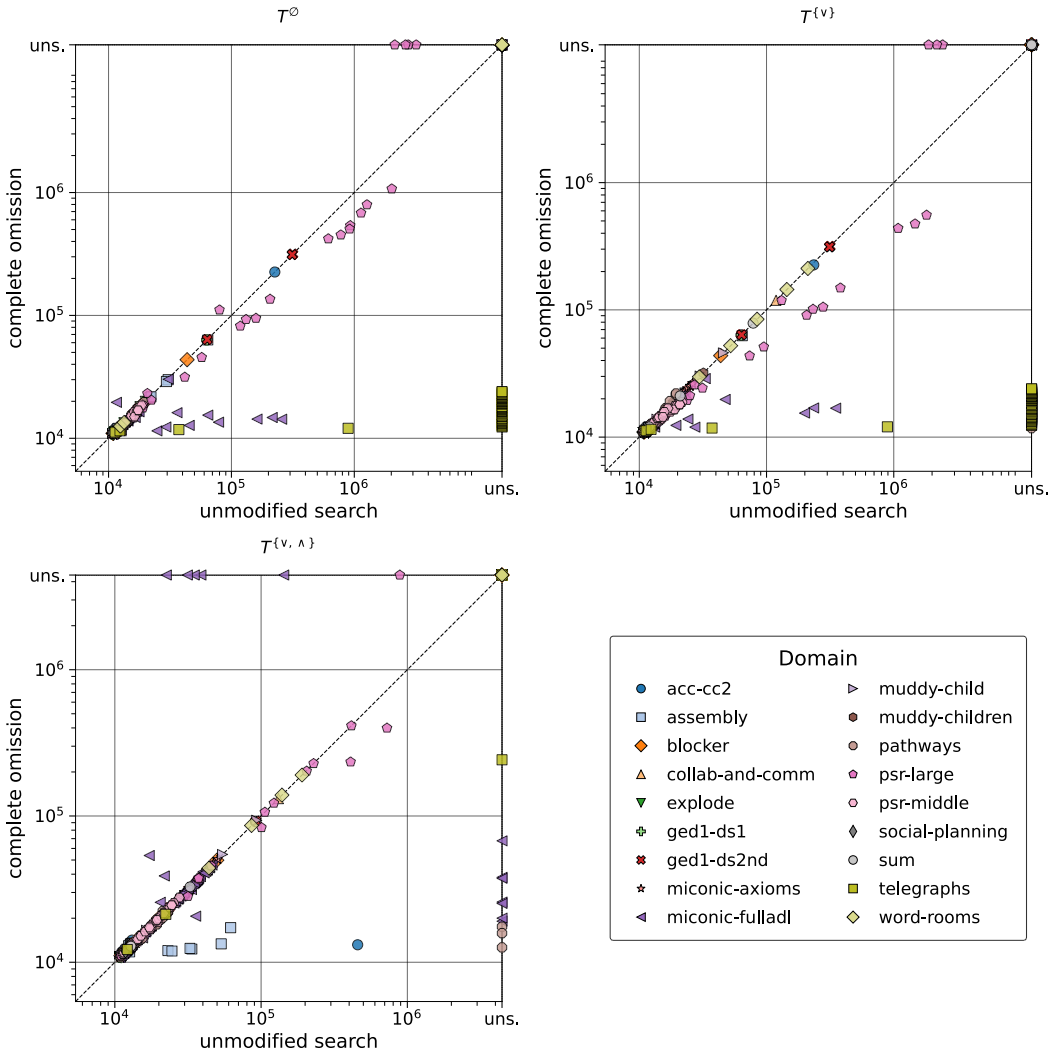

Figure 18: Per-instance comparison of total memory usage in kilobytes for the current search implementation (x-axis), where all derived and primary variables are stored in the state-registry, against a modified search implementation (y-axis), where only primary variables are stored in the state-registry. The search algorithm used for this comparison is LAMA-First. Tasks that could not be solved are represented on the grid-lines labeled with the "uns." label. No outliers have been excluded from this visualization. The $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ labels indicate the translator configuration used for both search implementations in the respective sub-plot.

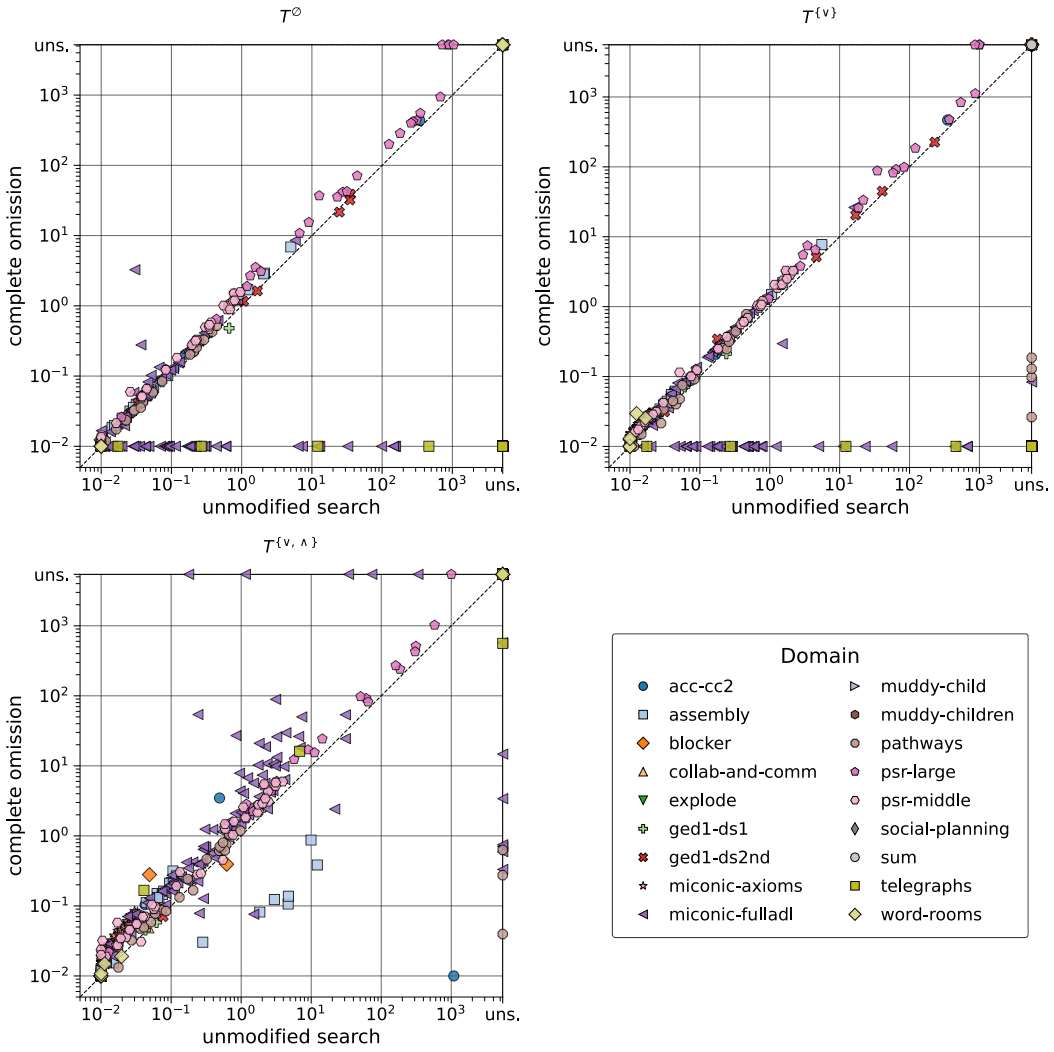

Figure 19: Per-instance comparison of search time in seconds for the current search implementation (x-axis), where all derived and primary variables are stored in the state-registry, against a modified search implementation (y-axis), where only primary variables are stored in the state-registry. The search algorithm used for this comparison is LAMA-First. Tasks that could not be solved are represented on the grid-lines labeled with the "uns." label. No outliers have been excluded from this visualization. The $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ labels indicate the translator configuration used for both search implementations in the respective sub-plot.

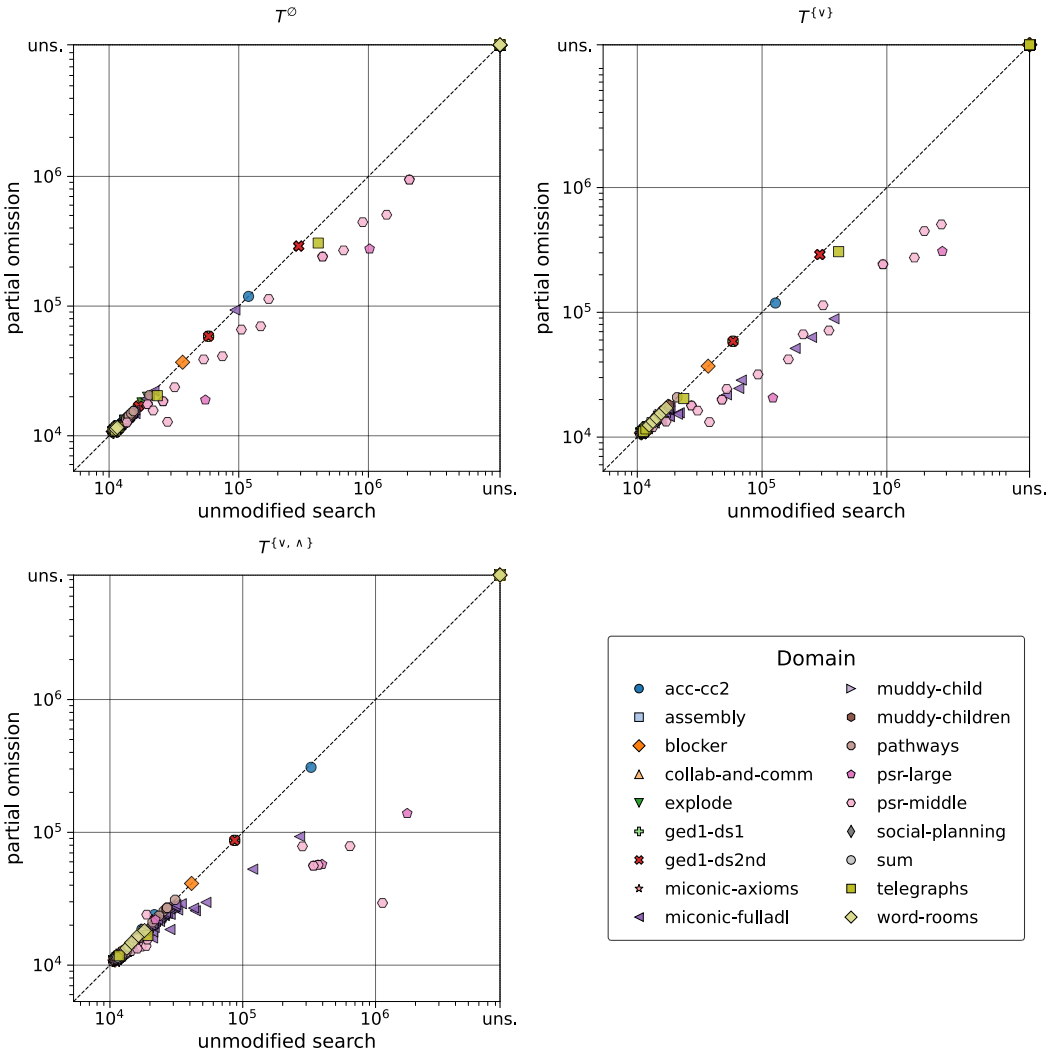

Figure 20: Per-instance comparison of total memory usage in kilobytes for the current search implementation (x-axis), where all derived and primary variables are stored in the state-registry, against a modified search implementation (y-axis), where only primary variables are stored in the state-registry. The search algorithm used for this comparison is greedy best-first search with FF and FF-only preferred operators. Tasks that could not be solved are represented on the grid-lines labeled with the "uns." label. No outliers have been excluded from this visualization. The $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ labels indicate the translator configuration used for both search implementations in the respective sub-plot.

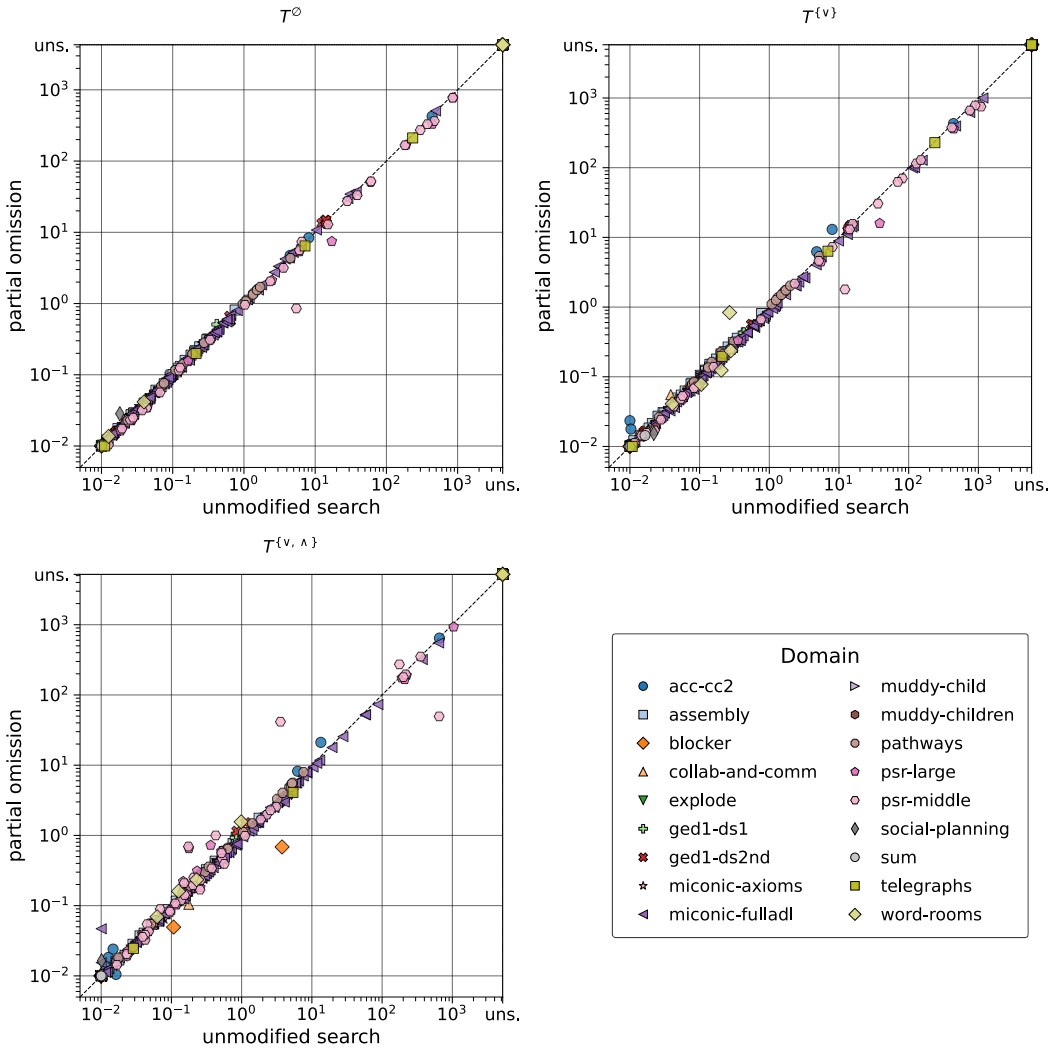

Figure 21: Per-instance comparison of total search time in seconds for the current search implementation (x-axis), where all derived and primary variables are stored in the state-registry, against a modified search implementation (y-axis), where only primary variables are stored in the state-registry. The search algorithm used for this comparison is greedy best-first search with FF and FF-only preferred operators. Tasks that could not be solved are represented on the grid-lines labeled with the "uns." label. No outliers have been excluded from this visualization. The $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ labels indicate the translator configuration used for both search implementations in the respective sub-plot.

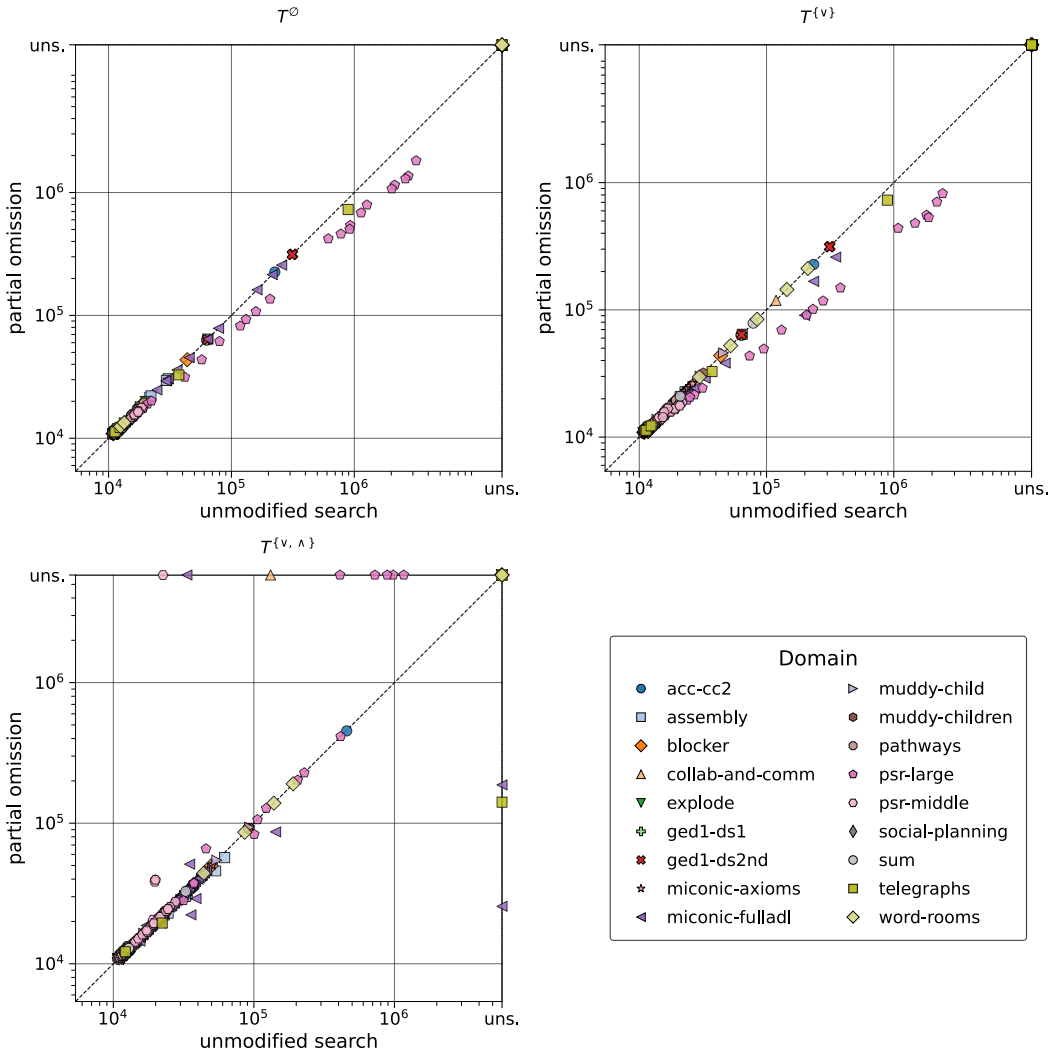

Figure 22: Per-instance comparison of total memory usage in kilobytes for the current search implementation (x-axis), where all derived and primary variables are stored in the state-registry, against a modified search implementation (y-axis), where only primary variables are stored in the state-registry. The search algorithm used for this comparison is LAMA-First. Tasks that could not be solved are represented on the grid-lines labeled with the "uns." label. No outliers have been excluded from this visualization. The $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ labels indicate the translator configuration used for both search implementations in the respective sub-plot.

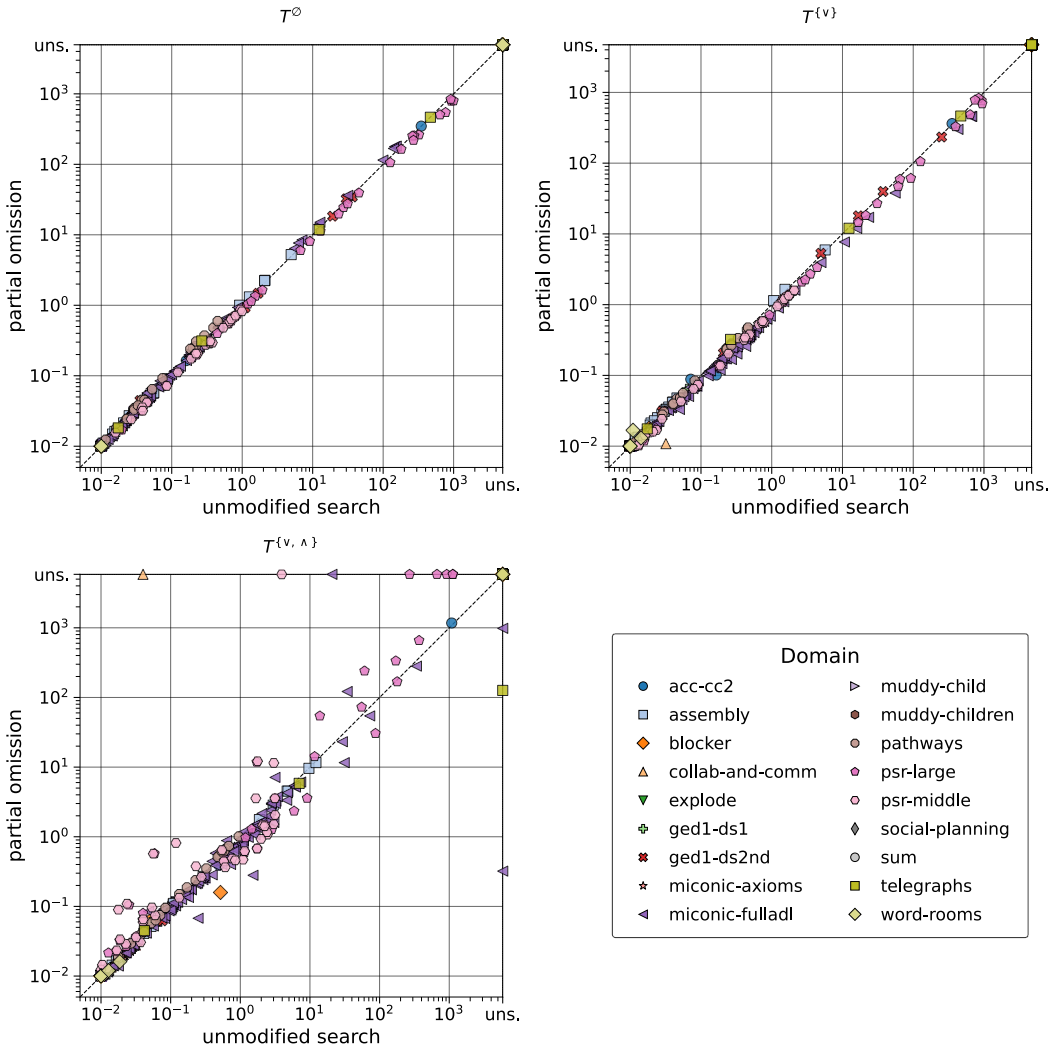

Figure 23: Per-instance comparison of total search time in seconds for the current search implementation (x-axis), where all derived and primary variables are stored in the state-registry, against a modified search implementation (y-axis), where only primary variables are stored in the state-registry. The search algorithm used for this comparison is LAMA-First. Tasks that could not be solved are represented on the grid-lines labeled with the "uns." label. No outliers have been excluded from this visualization. The $T^{\varnothing}$, $T^{\{\vee\}}$ and $T^{\{\vee,\wedge\}}$ labels indicate the translator configuration used for both search implementations in the respective sub-plot.