Finding a Dominating State in a Haystack: Efficient Data Structures for Dominance Pruning

María Fernanda Salerno Garmendia, Daniel Fišer, Álvaro Torralba
Aalborg University, Denmark
{mfsga, dafi, alto}@cs.aau.dk

Abstract

Dominance pruning is a technique to reduce search effort by comparing states against each other and discarding dominated states. However, despite being effective at significantly reducing the number of expansions in many domains, the overhead of comparing states may lead to reduced performance. This is especially critical in hard problems, as the overhead of comparing all pairs of states grows quadratically with the number of explored states.

In this paper, we investigate how to use data structures that exploit the inner structure of dominance relations in order to perform this check efficiently. We show that these data structures can avoid the quadratic worst-case in many domains, significantly improving the performance of search with dominance pruning.

1 Introduction

In optimal heuristic search the A* algorithm (Hart, Nilsson, and Raphael 1968) is widely used. Although successful, it faces exponential search spaces in many domains, even with almost perfect heuristics (Helmert and Röger 2008). To tackle this problem, techniques based on pruning redundant or suboptimal states have been proposed. We focus on dominance pruning (Torralba and Hoffmann 2015; Torralba 2021), a technique that discards search nodes if it can be shown that another node will lead to an at-least-as-good solution. This considers dominance relations that compare two states to determine whether one is as close to the goal as the other. This information is used to discard any state s if there exists another state t reached with equal or lower cost such that s is dominated by t. In that case, exploring s cannot lead to a cheaper solution.

Although dominance pruning can significantly reduce the number of expansions, it introduces substantial computational overhead. During the search, most operations (e.g. state generation, heuristic evaluation, and duplicate detection) scale linearly with the number of generated or expanded states. In contrast, comparing each state against all previously considered states has quadratic worst-case behavior in terms of the number of expanded states. As the number of stored states grows, each new state must be compared against an increasingly large set of candidates. Thus, even relatively inexpensive dominance tests can accumulate substantial overhead and offset the benefits of pruning. Previous work has considered cheaper alternatives such as comparing each state only against their parent state (Torralba 2017), or using dominance for reformulation (Torralba and Kissmann 2015). However, they do not fully take advantage of the information available in the dominance relation.

Our goal is to get the most out of dominance pruning by improving the efficiency of dominance checks in best-first search. If the dominance relation is treated as a black-box, getting maximum pruning would require comparing against all previous states. Fortunately, the dominance relations used in current approaches, such as those automatically discovered for classical planning tasks (Hall et al. 2013; Torralba and Hoffmann 2015; Wilhelm and Torralba 2025), share a common *factored* structure: states are modeled in terms of a set of *factors*, each with finite domain. Each factor has its own local dominance relation over its set of values, and t dominates s if and only if each fact of s is dominated by the corresponding fact of t within its factor.

Previous work has addressed dominance testing in similar contexts such as branch-and-bound search (Ibaraki 1977) or scheduling problems (Sierra 2013). In multi-objective optimization (Jaszkiewicz and Lust 2018) and/or multi-objective search (Zhang et al. 2024), states are ranked according to optimization objectives. While one can establish a parallel between these objectives and the factors we consider in our representation, they rely on total orders over the objectives, unlike planning tasks, which often involve partial orders over the factors. There are also structural differences: multi-objective problems usually involve a small number of objectives with large domains, while the planning tasks we consider often have many factors with small domains. Decoupled search (Gnad and Hoffmann 2018; Gnad 2021b) provides another related framework, where states are divided into a center factor and multiple leaf factors, each associated with a price function. Dominance pruning in decoupled search (Torralba et al. 2016; Gnad 2021a) exploits this structure, since dominance can only occur between states sharing the same center-factor value.

We introduce data structures to exploit the inherent structure of factored dominance relations. Rather than comparing each state against all previous states, we organize states us-

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

ing two ideas. First, inspired by decoupled search methods, we use a hash function that clusters states such that two potentially dominating states always belong to the same cluster, filtering out states that are not dominance-comparable. Second, inspired by ND-trees (Jaszkiewicz and Lust 2018), we organize states in a tree structure that exploits dominance directionality to prune comparisons within each cluster. Both approaches exploit the same structural property of factored dominance relations but offer different trade-offs in overhead and pruning power. Our experiments show that both ideas, used combined or separately, are able to keep the number of necessary comparisons low, limiting the overhead of dominance pruning and making it viable in most domains.

2 Background

A transition system is a tuple $\Theta = \langle S, L, T, s_I, S_G \rangle$ where S is a finite set of states, L is a finite set of labels each associated with a positive cost $c(l) \in \mathbb{N}$, $T \subseteq S \times L \times S$ is a set of transitions, $s_I \in S$ is the start state, and $S_G \subseteq S$ is the set of goal states. We write $s \xrightarrow{l} t$ as a shorthand for $(s, l, t) \in T$. A path in a transition system is a sequence of transitions $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} \dots \xrightarrow{l_n} s_n$ such that $s_i \xrightarrow{l_{i+1}} s_{i+1}$ for all $0 \leq i < n$. A plan π for a state s is a path $s \xrightarrow{l_1} \dots \xrightarrow{l_n} s_n$ such that $s_n \in S_G$, the cost of π is $\sum_{i=1}^{n} c(l_i)$. We use $h^*(s)$ to denote the cost of a cheapest plan for s. A plan for s is optimal if its cost equals to $h^*(s)$.

A factored transition system is a tuple $\mathcal{T} = \langle \Theta_1, \dots, \Theta_n \rangle$ of factors, where each factor corresponds to one or more state variables. Each factor is a transition system $\Theta_i = \langle S_i, L, T_i, s_I^i, S_G^i \rangle$ and all factors share the same label set L and the same label cost function c. We consider planning tasks modeled as factored transition systems (Torralba and Hoffmann 2015). For each factor Θ_i a dominance relation $\sqsubseteq_i \subseteq S_i \times S_i$ can be computed. These local relations induce a state-dominance relation on the product state space $S = S_1 \times \dots \times S_n$ defined as $s \preceq t \iff \forall i \in \{1, \dots, n\}, s[i] \sqsubseteq_i t[i]$. This means that a state s is dominated by a state t iff s is dominated by t in each factor.

Dominance pruning methods reduce search effort by pruning nodes that cannot lead to a cheaper solution than one already encountered. It relies on a dominance relation $\preceq \subseteq S \times S$ such that if $s \preceq t$ then $h^*(t) \leq h^*(s)$; that is, t is at least as close to a goal as s. In best-first search algorithms, a search node n_s represents a path from the initial state to a state s, and is associated with a cost $g(n_s)$ equal to the cost of that path. This cost plays a key role in dominance pruning. Pruning based on dominance can be performed at two moments during the search: at generation or expansion time. At generation time, when a node n_t is generated, we check if there is another node n_u in the open or closed list such that $t \preceq u$ and $g(n_u) \leq g(n_t)$. At expansion, the same check is performed just before expanding n_t. In both cases, if such n_u is found, n_t is dominated and can be safely pruned, i.e., removed from open without being expanded or closed.

3 Data Structures for Dominance Pruning

The main computational cost in dominance pruning lies in determining, for a state s, whether there exists a previously seen state t with $g(t) \leq g(s)$ such that $s \preceq t$. This requires comparing s to all such t, yielding a quadratic number of checks. Our goal is to design data structures that exploit the structure of the dominance relation to avoid these exhaustive pairwise comparisons.

To motivate the design, we provide a running example. Consider a planning task with three factors X_1, X_2, and X_3. Each factor X_i has a set of facts, and a dominance relation $\sqsubseteq_i$ defined over them. Suppose the dominance tables for the three factors are the following:

$\sqsubseteq_1$	a	b	c
a	✓	×	×
b	×	✓	×
c	×	×	✓

$\sqsubseteq_2$	u	v	w
u	✓	×	×
v	×	✓	✓
w	×	×	✓

$\sqsubseteq_3$	p	q
p	✓	✓
q	×	✓

These tables specify, for each factor, which values dominate others. For instance, in factor X_2, the value u only dominates itself, and the value v dominates both itself and w.

A state in the search space is a tuple $s = \langle x_1, x_2, x_3 \rangle$. State dominance is defined by a composition of the local dominance relations; that is, for two states $s = \langle x_1, x_2, x_3 \rangle$ and $t = \langle y_1, y_2, y_3 \rangle$:

$$s \preceq t \iff x_1 \sqsubseteq_1 y_1 \ \wedge \ x_2 \sqsubseteq_2 y_2 \ \wedge \ x_3 \sqsubseteq_3 y_3.$$

Checking whether s is dominated by any previously encountered state t requires repeatedly performing these three independent comparisons, an operation we refer to as a *check*. If as a result of these checks we find that s is not dominated, we must store it for future comparisons, an operation we refer to as *insert*.

The remainder of this section presents two baselines: a simple data structure based on explicit lists, and the previous approach by Torralba and Hoffmann (2015). In the next sections, we introduce additional data structures that take into account the structure of the dominance relations to mitigate the cost of the check and insert operations.

3.1 Explicit Data Structure

Even though Torralba and Hoffmann (2015) did not explore the use of simple explicit lists and focused on BDDs instead (see below), we introduce the explicit data structure as a simple baseline. This is a straightforward implementation that maintains an explicit list of stored states per g-value. Thus, it maintains a mapping from cost values g to sets of stored states:

$$\text{DB} = \{(g, P_g) \mid g \in \mathbb{N}, P_g \subseteq S\}.$$

Here, P_g denotes the set of states previously encountered at cost g. This structure ensures that dominance checks for a node n_s can be restricted to the union $\bigcup_{g' \leq g(n_s)} P_{g'}$. Since the mapping is ordered by g, the algorithm can terminate early once stored costs exceed the current state's cost. For each such g'-value, we iterate over all stored states $t \in P_{g'}$ and, for each such state t, dominance is checked factor by factor: $s \preceq t \iff \forall i \in \{1, \dots, n\}, s_i \sqsubseteq_i t_i$ where $\sqsubseteq_i$ is the local dominance relation for factor i. If any such t satisfies the condition, n_s is pruned. Otherwise, s is inserted into $P_{g(n_s)}$.

3.2 Binary Decision Diagrams

Previous work by Torralba and Hoffmann (2015) used an approach based on Binary Decision Diagrams (BDDs) (Bryant 1986). BDDs provide a compact representation of sets of states (Edelkamp and Helmert 2000; Torralba et al. 2017), allowing fast membership tests. Thus, this approach optimizes the time of dominance checks, although inserting new states in it can be relatively expensive.

The idea is very similar to the explicit data structure, organizing the previously seen states into sets of states with the same g-value. But instead of using a list to represent each set of states P_g, one or more BDDs are used to represent the set of dominated states $D_g = \{s \mid t \in P_g, s \preceq t\}$.

Checking whether a newly generated node n_s is dominated then reduces to testing whether $s \in D_g$ for all $g \leq g(n_s)$. These membership tests are linear in the size of the state representation, so this approach is meant to keep these checks very efficient.

When a node n_s is expanded, the set of states that are dominated $\{t \mid t \preceq s\}$ is inserted in D_g. The BDD representation of the dominated states can always be obtained efficiently. However, inserting them in D_g may significantly increase the size of the BDD representing D_g. To address this, a disjunctive partitioning is used for D_g, using multiple BDDs to represent this set of states whenever the single BDD grows too much.

4 Hash Map Based Data Structure

To efficiently organize states for dominance checks, we exploit a key structural property of dominance relations: states can only dominate one another if in each factor they are related by the local dominance relation. This enables us to partition the state space into disjoint subsets of comparable states. For this we use a hash map whose buckets contain only states that may potentially dominate or be dominated by one another.

Definition 1 (Dominance-Preserving Hash Function)**.** *Let S be the set of all states and let $\preceq \subseteq S \times S$ be a dominance relation. A function* $\text{hash} : S \to \mathbb{N}$ *is* dominance-preserving *if for all states $s, t \in S$, $s \preceq t \implies$* $\text{hash}(s) = \text{hash}(t)$.

That is, any pair of states related by dominance must be assigned to the same hash bucket.

A dominance-preserving hash function induces a partition of the state space into disjoint buckets indexed by hash values. By definition, if two states are related by dominance, they are mapped to the same bucket. Consequently, states placed in different buckets are guaranteed to be incomparable with respect to dominance. This ensures that restricting dominance checks to states within the same bucket is sound: no potential dominance pair is ever missed.

Thus, any dominance-aware storage structure can be organized as a two-level scheme. The outer level maps hash values to buckets, while the inner level stores the states belonging to a bucket and performs the actual dominance checks. Each bucket maintains its own inner dominance data structure (e.g., explicit storage, BDD, tree). When inserting or querying a state s, the data structure first computes $\text{hash}(s)$ to identify the relevant bucket, and then performs dominance tests only within that bucket. The effectiveness of this approach depends on how well the hash function separates incomparable states, but correctness follows solely from the dominance-preserving property.

Next, we describe two automatic ways of defining dominance-preserving hash functions for any given factored dominance relation.

4.1 Hash based on Non-Dominating Factors

Take the fact-dominance relations $\sqsubseteq_i$ for each factor Θ_i. For some factors, this relation reduces to identity: $a \sqsubseteq_i b \iff a = b$. This is the case for factor X_1 in our running example, where the only dominating fact for each value is itself. We call these non-dominating factors, since they do not contribute to strict dominance but require equality to ensure comparability. For the remaining factors, which have $a \neq b$ such that $a \sqsubseteq_i b$, we refer to them as dominating factors.

This classification allows us to partition the state space into equivalence classes of states that agree on the value of non-dominating factors. Let $I = \{i_1, \dots, i_k\}$ be the indices of the non-dominating factors, i.e., factors Θ_i such that $a \sqsubseteq_i b \iff a = b$.

Definition 2 (nd-hash)**.** *Let $\sqsubseteq_1, \dots, \sqsubseteq_n$ be a factored dominance relation, and* $\text{hash-vector} : \mathbb{N}^k \mapsto \mathbb{N}$ *any hash function for a tuple. An nd-hash function is* $\text{nd-hash}(s) = \text{hash-vector}(s[i_1], \dots, s[i_k])$*, where $i_1, \dots, i_k$ are the indices of the non-dominating factors.*

Ideally, hash-vector is a perfect hash function mapping each different vector to a different value, so that states in different vectors are never compared against each other. This is possible, as the number of different values for each factor is known. However, it may not be practical, as the resulting value could exceed 2^{64} and not be representable as a 64-bit number. In practice, we compute hash values using the Boost `hash_combine` scheme for hashing vectors (Boost C++ Libraries 2024). This is still correct, as the result is dominance preserving regardless of the choice of hash-vector.

Proposition 1. *The hash function* nd-hash *is dominance-preserving.*

Proof. Let s, t be two states such that $s \preceq t$. By definition of the factored dominance relation, $s[i] \sqsubseteq_i t[i]$ for all i. For non-dominating factors $i \in I$, we have $s[i] \sqsubseteq_i t[i] \implies s[i] = t[i]$. Therefore, if $s \preceq t$, all entries of s and t in the non-dominating factors are identical. Consequently,

$$\text{nd-hash}(s) = \text{hash-vector}(s[i_1], \dots, s[i_k]) = \text{hash-vector}(t[i_1], \dots, t[i_k]) = \text{nd-hash}(t)$$

□

Take our running example. Factor X_1 is non-dominating, while factors X_2 and X_3 are dominating. The hash function is defined as $\text{nd-hash}(s) = \text{hash-vector}(s[1])$. Thus, all states are partitioned into three buckets, corresponding to the three values of factor X_1. Specifically, D_a, D_b, and D_c contain states $\langle a, x_2, x_3\rangle$, $\langle b, x_2, x_3\rangle$, and $\langle c, x_2, x_3\rangle$, respectively, for all $x_2 \in \{u, v, w\}$ and $x_3 \in \{p, q\}$.

4.2 Factorwise Hash

In contrast to non-dominating factors, some factors have more complex dominance relations where multiple values can dominate one another. In our running example, in factor X_2 v dominates w. Then, although v and w are distinct values, they are not independent with respect to dominance: a state using v in this factor may dominate a state using w. Any values connected in this way must share the same hash bucket, since dominance can happen across them. This motivates grouping factor values together if they are connected by the local dominance relation

To this end, for each factor Θ_i, we consider its local dominance relation $\sqsubseteq_i \subseteq S_i \times S_i$, which can be represented as a directed graph $G_i = (S_i, E_i)$, where $(a, b) \in E_i$ if and only if $a \sqsubseteq_i b$. We then compute the connected components of this graph with respect to $\sqsubseteq_i$, that is, an equivalence relation $\sim_i$ such that

$$a \sim_i b \iff (a \sqsubseteq_i b) \vee (b \sqsubseteq_i a).$$

Each equivalence class under $\sim_i$ corresponds to a *connected component* $C_{i,j} \subseteq S_i$, where the index j denotes the j-th such component in the partition of S_i. Intuitively, two values of factor Θ_i belong to the same component if they can reach one another through a sequence of local dominance relations. If the dominance relations are transitive, it is enough to consider direct connections.

We use these components to define a grouping of the state space. Each factor Θ_i induces a partition

$$S_i = C_{i,1} \cup C_{i,2} \cup \cdots \cup C_{i,m_i},$$

and every state $s = (s[1], \ldots, s[n])$ belongs to the product component

$$C(s) = (C_{1,j_1}, \ldots, C_{n,j_n}),$$

where $s[i] \in C_{i,j_i}$ for each i.

Definition 3 (fw-hash)**.** *Let $\sqsubseteq_1, \ldots, \sqsubseteq_n$ be a factored dominance relation, and* $\text{hash-vector} : \mathbb{N}^n \mapsto \mathbb{N}$ *any hash function for a tuple. A fw-hash function is defined as* $\text{fw-hash}(s) = \text{hash-vector}(j_1, \ldots, j_n)$*, where j_i is the index of the connected component containing $s[i]$.*

In our running example, the connected components for each factor and their corresponding facts are:

$$\sqsubseteq_1: \begin{array}{c} C_{1,1} : \{a\} \\ C_{1,2} : \{b\} \\ C_{1,3} : \{c\} \end{array} \quad \sqsubseteq_2: \begin{array}{c} C_{2,1} : \{u\} \\ C_{2,2} : \{v, w\} \end{array} \quad \sqsubseteq_3: \; C_{3,1} : \{p, q\}$$

Then, if we have for example a state $s = \langle c, v, q \rangle$, its factorwise hash is computed as $\text{fw-hash}(s) = \text{hash-vector}(3, 2, 1)$, since $c \in C_{1,3}, v \in C_{2,2}$, and $q \in C_{3,1}$. Notice this will be the same hash value for states $\langle c, v, p \rangle$, $\langle c, w, q \rangle$, and $\langle c, w, p \rangle$ since this hash function is dominance-preserving, as formalized below.

Proposition 2. *The hash function* fw-hash *is dominance-preserving.*

Proof. By definition of the global dominance relation, $s \preceq t \iff \forall i,\ s[i] \sqsubseteq_i t[i]$. If $s \preceq t$, then for each factor Θ_i, the values $s[i]$ and $t[i]$ are related by $\sqsubseteq_i$, implying that they belong to the same connected component C_{i,j_i}. Consequently,

$$\begin{aligned} \text{fw-hash}(s) = \text{hash-vector}(j_1, \ldots, j_n) = \\ \text{hash-vector}(j_1, \ldots, j_n) = \text{fw-hash}(t), \end{aligned}$$

so the hash function is dominance-preserving. □

5 Tree Structure

The hash functions introduced above exploit the fact that many pairs of states are trivially incomparable because they disagree in some factor in a way that rules out dominance. Hashing exploits this by grouping only potentially comparable states into the same bucket. But within each bucket, all remaining states are still treated uniformly. However, even though we are looking for comparable states only, we are specifically interested in finding dominating states. Take factor X_3 in our running example. Here, p dominates q, but q does not dominate p. They are comparable, but if a state has p in this factor, it can never be dominated by a state with q in this factor. This observation motivates a structure that rather than merely testing comparability, mirrors the dominance graph of each factor, separating dominating values from dominated ones. Then, when checking whether a state is dominated, we can prune entire branches of the structure that correspond to factor values that cannot dominate.

A similar idea was explored in multi-objective optimization by Jaszkiewicz and Lust (2018), who proposed a tree data structure for updating a Pareto archive composed of mutually non-dominating solutions. Their approach cannot be applied directly to our setting, since they compute approximate ideal points, which assume that all factors have totally-ordered relations. Thus, we take inspiration from this idea and adapt it to our setting, where instead of multi-objective points we have states defined by multiple factors.

Our structure organizes states into a tree according to their factor values. The tree has two types of nodes: internal nodes that partition states based on their value for a chosen factor, and leaf nodes that store actual states. To store the states inside the leaf nodes, we use the explicit data structure described earlier, which is a mapping from cost values g to sets of stored states. Then, in each leaf we have $\text{DB}_{\text{exp}} = \{(g, P_g) \mid g \in \mathbb{N}, P_g \subseteq S\}$.

Initially, the tree consists of a single root node that is a leaf. When the number of stored states in a leaf node exceeds a threshold, the node is split based on one factor Θ_i. Let S_i be the set of values for factor Θ_i. Each child node then contains only the subset of states whose Θ_i equals a specific value $v_i \in S_i$.

$$\text{DB}_{\text{tree}}(\text{DB}_{\text{exp}}) = \begin{cases} \text{Leaf}(\text{DB}_{\text{exp}}), & \text{if } \sum_{(g,P_g) \in \text{DB}_{\text{exp}}} |P_g| < \tau, \\ \text{Split}_i\Big(\{\, \text{DB}_{\text{tree}}(S_{v_i}) \mid v_i \in S_i \,\}\Big), & \text{otherwise.} \end{cases}$$

where $\text{DB}_{\text{exp}} = \{(g, P_g) \mid g \in \mathbb{N}, P_g \subseteq S\}$ is the explicit data structure storing states in the leaf node, τ is the splitting threshold, Split_i is an internal node that splits states based on their value in Θ_i, and $S_{v_i} = \{(g, P_g) \in \text{DB}_{\text{exp}} \mid \exists s \in P_g : s[i] = v_i\}$ is the subset of states with value v_i in factor Θ_i.

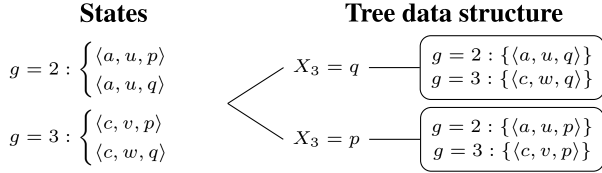

Figure 1: Tree structure organizing four states by splitting on factor X_3. Each leaf node contains an explicit data structure mapping cost values g to sets of stored states.

To illustrate this idea, Figure 1 shows how four states from our running example can be organized in a tree. We choose to split on factor X_3, whose domain is $\{p, q\}$, and $p \sqsubseteq_3 q$. With states organized in this way, when checking whether a new state s with a $g = 2$, and values $\langle x_1, x_2, p \rangle$ (for any value of x_1 and x_2) is dominated, we only need to explore the p branch, and inside the leaf node we only need to compare with states with a g value less than or equal to that of the new state, in this case 2.

To choose the splitting factor Θ_i of a given node, we consider strategies that trade off the cost of computing the split with the quality of the resulting partition. We describe the implemented strategies below.

Random Splitting. The simplest approach selects the splitting factor uniformly at random. While this requires no computation, the resulting partitioning may ignore the dominance structure and produce unbalanced trees.

Minimum Expected Checks. This strategy selects the factor that minimizes the expected number of dominance checks within the resulting subsets. For each factor Θ_i, we estimate how many pairwise comparisons would be required if the node were split by that factor. The intuition is that checks are needed only between child nodes whose states may dominate one another, so we count, for the states in the node, the number of such potential comparisons per factor.

Take the example in Figure 1. First consider splitting by factor X_3. We have two states with value p and two states with value q. States with value p must be compared only against states with value p (since they cannot be dominated by states with value q), leading to $2 \times 2 = 4$ comparisons. States with value q must be compared against states with value q and p, leading to $2 \times (2 + 2) = 8$ comparisons. We thus expect a total of 12 comparisons when splitting by X_3. We perform this calculation for each factor, and select the one minimizing the expected number of comparisons.

Formally, for a factor Θ_i with a dominance relation $\sqsubseteq_i$, let V_i be the set of values currently stored in the node for that factor, and let $c(v)$ be the number of states with value $v \in V_i$. We define the set of dominating value pairs as $\text{Pairs}_i = \{(v_1, v_2) \mid v_1, v_2 \in V_i, v_1 \sqsubseteq_i v_2\}$. The estimated number of necessary checks is $C_i = \sum_{(v_1,v_2)\in\text{Pairs}_i} c(v_1) \cdot c(v_2)$. The factor with the minimal C_i is chosen for splitting.

Minimum Variance. Instead of focusing on dominance relations, this strategy measures how evenly the states are distributed across the values of each factor. The intention is to divide the number of states as evenly as possible across the children. If one child receives almost all the states while others remain nearly empty, the tree becomes unbalanced, resulting in deeper paths, more recursive insertions, and potentially more dominance checks overall. Variance provides a direct quantitative measure of this imbalance.

For a factor Θ_i with counts $c(v)$ for $v \in V_i$, we compute the variance of these counts as $\sigma_i^2 = \frac{1}{|V_i|} \sum_{v \in V_i} (c(v) - \bar{c})^2$, where $\bar{c}$ is the mean number of states per value. The factor with the lowest variance is selected.

6 Experiments

To evaluate the effectiveness of our proposed data structures for dominance pruning, we conducted an experimental study. We implemented all data structures in the Fast Downward planning system (Helmert 2006). All variants follow the same dominance pruning procedure, checking whether each node selected for expansion is dominated by any previously expanded state. Thus, all configurations differ only in the underlying data structure used to store and query dominance between states. We used the LM-Cut heuristic (Helmert and Domshlak 2009) for all configurations.

To compute factored dominance relations, we use the label-dominance simulation method (Torralba and Hoffmann 2015). We use two configurations depending on the factored representation used to compute the dominance. The first one, atomic, uses one factor per variable. The second one, merges multiple variables into a single factor using merge-and-shrink with exact bisimulation (*m&s*) (Helmert et al. 2014). The main difference is that *m&s* has larger factors and more informed dominance relations, but also spends significantly more time during pre-computation.

We consider the following configurations: **EXP**, which uses explicit data structure; **BDD**, which uses the BDD representation; and **TREE**. On top of that, we can enable or disable the hash: **HASH-ND**, which uses the non-dominating factors hash; **HASH-FW**, which uses the factorwise hash. Hash-based configurations always use another method as the inner structure, indicated in parentheses, e.g., HASH-FW(EXP) for a factorwise hash with the explicit representation. Each configuration is evaluated with both atomic and *m&s* dominance relations, except for BDD, which is not evaluated with *m&s*. We do not have an implementation for this combination, and given that BDD already performs poorly with the atomic representation, the effort to implement it for *m&s* was deemed unnecessary.

In the tree-based configurations, we evaluated several node splitting strategies, and different thresholds for the maximum number of states per leaf node (50, 100, 150, and 200 states). Subsection 6.4 presents a detailed analysis of the different configurations for the tree-based data structure. Based on that analysis, we selected for comparison against other data structures the configuration using minimum expected comparisons (MEC) as splitting strategy, and a threshold of 50 states per leaf node.

We evaluated the approaches on classical planning benchmarks from the optimal track of the International Planning Competitions (IPC) from 1998 to 2023. All experiments

were executed using the Lab tool (Seipp et al. 2017) on a cluster with AMD EPYC 7551 CPUs and per-run limits of 4 GB memory and 30 minutes of runtime. Source code and experimental results are published in Zenodo (**?**).

6.1 Overall results

Table 1 shows coverage per domain and configuration. Domains are grouped according to whether useful dominance relations prune at least one state in (1) atomic and *m&s*, (2) *m&s*, or (3) neither. Although comparing data structures on domains without pruning reveals overhead, this can be avoided by disabling pruning; thus, Table 1 highlights only the two domains with the largest differences to the baseline, grouping the remaining domains without pruning as *Others*.

For completeness, we note that overall coverage without dominance pruning is higher, partly due to the overhead of computing the dominance relation, which is independent of our data structures. In some domains (e.g., tidybot and organic-synthesis-split), this precomputation alone exceeds the time limit. When considering search time only (i.e., assuming the dominance relation is given), pruning is more favorable. Nevertheless, even with full runtime, the proposed data structures increase the number of domains where pruning improves coverage from two to nine.

Both the hash-based and the tree data structure significantly outperform the baselines, EXP and BDD, across almost all domains. TREE achieves the best coverage for both atomic and *m&s* dominance. The use of the hash substantially improves coverage for EXP and BDD. BDD configurations consistently underperform in coverage and search time, though they follow similar trends to EXP when used as inner structures for hash configurations, indicating that the advantages of the hash can translate to different inner data structures. Combining hash and tree, however, yields no significant improvement over tree alone. We next analyze the data structures in more detail.

6.2 Performance of hash-based data structures

We first analyze the impact of the best hash variant, HASH-FW, combined with EXP and TREE. Figures 2 and 3 compare EXP and HASH-FW in terms of search time (excluding dominance precomputation, as it is identical across configurations), and the **average number of comparisons per check performed**, i.e., the average number of times two states are dominance-compared each time that the dominance check function is called. In other words, this metric indicates how many states we need to compare against every time we want to check if a state is dominated by any other state in the data structure. All plots use logarithmic scale, with values below 10^{-2} clipped to 10^{-2} for visualization. Points in the scatter plots that lie on the diagonal line indicate instances where both configurations achieved the same result. Blue and orange dots correspond to the atomic and *m&s* representations, respectively.

Looking at the search time results (Figure 2, left), the hash data structures significantly outperform their non-hash counterparts when using EXP as inner data structures. The reduction in search time becomes more pronounced as the instances become harder, as the more states are expanded, the

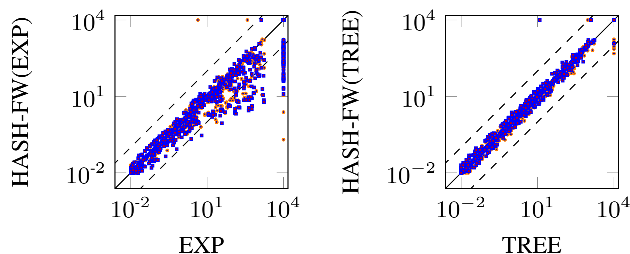

Figure 2: Search time: EXP vs HASH-FW(EXP) (right), TREE vs HASH-FW(TREE) (left). Blue and orange dots correspond to atomic and *m&s* representations, respectively.

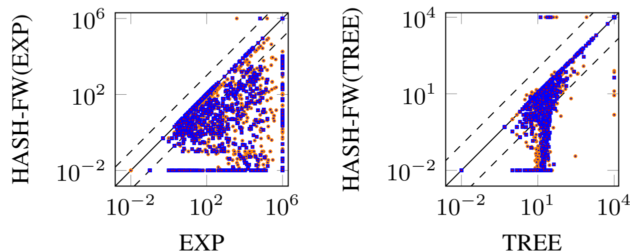

Figure 3: Average comparisons per check: EXP vs HASH-FW(EXP), TREE VS HASH-FW(TREE).

more comparisons are performed by the baseline. The number of comparisons in the left part of Figure 3 confirms that this is due to an outstanding decrease in the number of comparisons, both with atomic and *m&s* dominance relations. As can be seen in the plots, the hash-based data structures significantly reduce the average number of comparisons per check compared to their non-hash counterparts.

When using TREE as inner data structure (right part of Figures 2 and 3), we observe that the gains on search time are insignificant both with atomic and *m&s* dominance relations. It is noticeable that TREE is outperformed by its hash-based counterpart in terms of average comparisons per check. However, in terms of search time, both data structures show a similar behavior, and TREE even has a slightly better coverage in the atomic representation. This suggests that although the hash-based data structure reduces the number of comparisons per check more effectively, the overhead introduced by managing multiple inner tree data structures (one per hash bucket) may offset this advantage. It is also worth noting in the right plot of Figure 3 that TREE, although outperformed by its hash-based counterpart, usually keeps the average comparisons per check under 100, which explains its good performance in terms of search time and coverage.

6.3 Hash-ND versus Hash-FW

The coverage results from Table 1 show that, while both configurations outperform the baseline, HASH-FW gets the best performance with respect to HASH-ND, especially with the factored dominance relations computed by *m&s*.

Both hash-based data structures achieve similar coverage results, with HASH-FW slightly ahead in atomic and signifi-

		Atomic dominance relation									M&S dominance relation					
		EXP			**BDD**			**TREE**			**EXP**			**TREE**		
	hash variant →	-	ND	FW	-	ND	FW	-	ND	FW	-	ND	FW	-	ND	FW
ATOMIC AND M&S	barman	0	**4**	**4**	0	0	0	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**
	driverlog	**14**	**14**	13	13	**14**	**14**	**14**	**14**	**14**	13	**14**	**14**	**14**	**14**	13
	floortile	10	16	16	5	16	16	**17**	**17**	**17**	13	13	16	16	**17**	**17**
	grid	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	1	1	1	1	1	1
	nomystery	18	**19**	**19**	16	18	18	**19**	**19**	**19**	**19**	**19**	**19**	**19**	**19**	**19**
	parcprinter	29	29	31	29	29	31	33	33	33	35	34	35	**36**	35	35
	psr-small	48	**49**	**49**	48	**49**	**49**	**49**	**49**	**49**	48	48	**49**	**49**	**49**	**49**
	quantum-layout	11	**12**	**12**	11	11	11	**12**	**12**	**12**	7	7	8	7	7	7
	rovers	7	8	8	7	8	8	8	8	8	8	8	**9**	**9**	**9**	**9**
	satellite	7	7	7	7	7	7	8	8	8	8	8	8	**9**	**9**	**9**
	sokoban	36	44	44	21	32	32	**50**	**50**	**50**	43	43	48	49	49	48
	tidybot	8	8	8	8	8	8	**11**	8	9	0	0	0	0	0	0
	trucks	**10**	**10**	**10**	9	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**
	visitall	15	15	16	14	15	16	**18**	**18**	**18**	16	16	16	**18**	**18**	**18**
	woodworking	37	38	38	34	34	38	38	38	38	43	42	43	42	42	**44**
M&S	ged	13	**15**	**15**	7	11	13	**15**	**15**	**15**	13	13	**15**	**15**	**15**	**15**
	gripper	5	6	6	4	6	6	7	7	7	7	7	7	**8**	**8**	**8**
	hiking	8	10	10	10	9	10	10	8	10	9	**11**	**11**	**11**	10	**11**
	mprime	**21**	**21**	**21**	**21**	**21**	**21**	**21**	**21**	**21**	20	20	20	20	20	20
	pipesworld-t	9	**12**	**12**	9	**12**	**12**	**12**	**12**	**12**	11	**12**	**12**	**12**	**12**	**12**
	tpp	6	6	6	6	6	6	6	6	6	6	6	6	**7**	**7**	**7**
NONE	elevators	35	**40**	**40**	**40**	**40**	**40**	**40**	38	**40**	35	**40**	**40**	**40**	**40**	**40**
	openstacks	23	43	43	34	45	45	43	37	43	32	35	**52**	47	47	50
	Others	481	497	**498**	454	476	476	**498**	493	**498**	433	441	441	442	443	443
	Total	870	944	947	825	896	907	**964**	946	962	849	868	900	902	901	906
	Weighted Total	16.9	18.8	18.8	15.5	17.3	17.6	**19.2**	18.7	**19.2**	15.3	15.8	16.4	16.5	16.5	16.5

Table 1: Coverage results, highlighting in bold the best configurations per domain. ND and FW correspond with HASH-ND and HASH-FW, respectively. Domains are grouped according to whether useful dominance relations were found with atomic and *m&s*, only *m&s*, or none. For the latter, we only highlight the two domains with the highest difference with respect to EXP (elevators and openstacks) and group the rest under "Others". The last two rows show total coverage and weighted coverage based on the number of instances in each domain.

cantly ahead in *m&s* when using EXP as inner data structure. Figure 4 compares their search time. For the atomic representation, it can be seen that in commonly solved instances, none of the data structures clearly outperforms the other. However, HASH-FW achieves a higher coverage, especially when using the tree as its inner data structure. For *m&s*, HASH-FW shows better performance in terms of search time, and in this case, the improvement in coverage is more pronounced when using EXP as inner data structure.

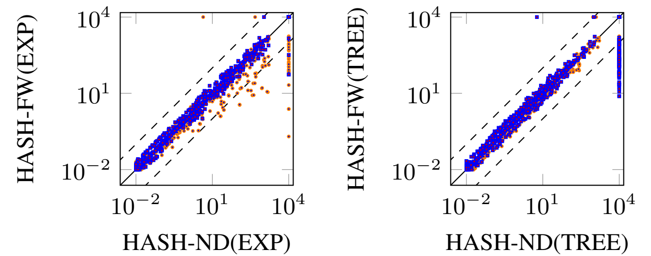

Figure 4: Search time: HASH-ND vs HASH-FW for EXP (left), and TREE (right).

We also compare the average number of comparisons per check performed by both data structures in Figure 5. As can be seen in the plots, HASH-FW tends to perform fewer comparisons per check than HASH-ND. While with the atomic representation this difference is not very pronounced, with *m&s* it is rather significant. The better results achieved by HASH-FW using *m&s* can be attributed to the fact that different factors in the dominance relation are being merged together. Therefore, it is likely that some non-dominating factors considered by HASH-ND are no longer present in the *m&s* relation. Meanwhile, HASH-FW continues to effectively partition the state space based on the new factors, since the bigger factors resulting from the merge-and-shrink process can still be partitioned in their connected components to create the hash buckets.

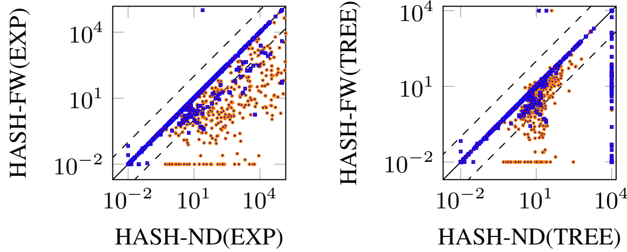

Figure 5: Average comparisons per check: HASH-ND vs HASH-FW for EXP (left), and TREE (right).

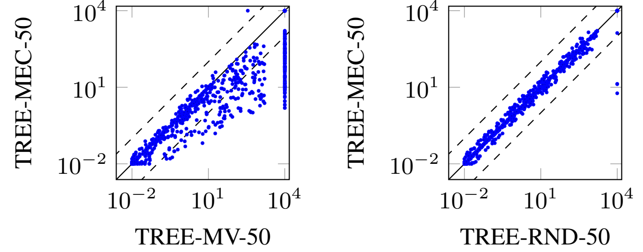

Figure 6: Search time: TREE-MEC vs TREE-MIN-VAR (left), TREE-MEC vs TREE-RND (right).

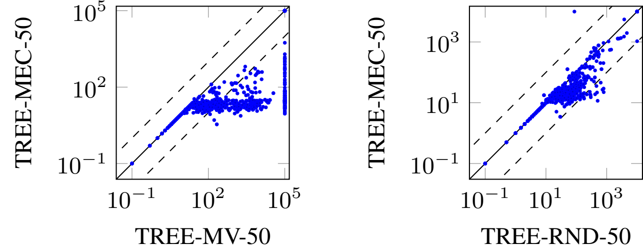

Figure 7: Average comparisons per check: TREE-MEC vs TREE-MIN-VAR (left), TREE-MEC vs TREE-RND (right).

6.4 Best configuration for TREE

We also analyzed the impact of the splitting strategy and threshold on the performance of TREE. Figures 6 and 7 compare MEC, RND and MV in terms of search time and average number of comparisons with a threshold of 50 states per leaf node. In terms of runtime, MEC clearly outperforms MV, showing that the choice of splitting strategy strongly affects TREE's performance. However, RND performs only slightly worse than MEC on average. This highlights that designing an effective splitting strategy is non-trivial: even intuitively reasonable strategies may perform poorly, while a simple random strategy can remain competitive.

Looking at the average number of comparisons provides additional insight. The differences between MEC and MV are mainly due to better tree structure under MEC rather than the overhead of computing the splitting criterion. MV constructs trees that require significantly more comparisons, whereas MEC typically keeps the number below 100. Since each leaf stores at most 50 states, this means that MEC organizes the tree in such a way that each new incoming state is only compared against states stored in a handful of leaves.

With respect to MEC and RND, although MEC clearly outperforms RND in terms of average comparisons per check, RND still keeps the number of comparisons below 1000. This means that, with these dominance relations, RND has a constant overhead in terms of the number of comparisons. As MEC has a higher overhead for calculating the splitting criterion, this makes both strategies perform similarly in terms of total time.

To determine the best threshold for the maximum number of states per leaf node, we compared the MEC strategy using thresholds of 25, 50, 100, 150, and 200 states. In terms of runtime, the differences were minor and did not indicate a clear winner. We therefore focused on the average number of comparisons per check for each threshold.

A potential concern with smaller thresholds is that they may require traversing more nodes to reach the leaves, potentially increasing the number of comparisons per check. However, the results show the opposite: smaller thresholds consistently yield fewer comparisons per check, suggesting that keeping leaf nodes small is advantageous. There is, however, an additional factor to consider when selecting the threshold: the overhead introduced by more frequent node splits when using smaller thresholds. In our experiments, there was still not a significant difference, which indicated that the overhead was manageable. However, the configuration with a threshold of 50 states per leaf node achieved a slightly better coverage than the others. Based on these observations, we selected the MEC strategy with a threshold of 50 states per leaf node as the most effective configuration for the tree-based data structure.

7 Conclusions

In this work, we addressed the computational overhead that limits the practical effectiveness of dominance pruning, and showed that the structural properties of factored dominance relations can be exploited to reduce this cost substantially. We introduced hash-based and tree-based data structures that avoid unnecessary comparisons by partitioning the set of stored states. These mechanisms restrict dominance checks to states that can dominate a given one, reducing the number of comparisons. Our empirical evaluation shows that these approaches consistently lower dominance-checking effort and yield significant runtime improvements over baseline implementations, including BDD-based methods. The best-performing data structure in our experiments was the tree-based one, which achieved the highest coverage, and the largest reductions in dominance checks. By reducing the overhead of dominance pruning, our methods expand the range of domains where dominance-aware search becomes practical, laying the foundation for more advanced dominance-guided search strategies.

Acknowledgements

This research was partly supported by the Independent Research Fund Denmark through a Sapere Aude: DFF-Starting Grant under reference number 3120-00063B.

References

Boost C++ Libraries. 2024. Boost.ContainerHash.

Bryant, R. E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8): 677–691.

Edelkamp, S.; and Helmert, M. 2000. On the Implementation of MIPS. In Traverso, P.; Veloso, M.; and Giunchiglia, F., eds., *Proceedings of the AIPS 2000 Workshop on Model-Theoretic Approaches to Planning*.

Gnad, D. 2021a. Revisiting Dominance Pruning in Decoupled Search. In Leyton-Brown, K.; and Mausam, eds., *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2021)*, 11809–11817. AAAI Press.

Gnad, D. 2021b. *Star-Topology Decoupled State-Space Search in AI Planning and Model Checking*. Ph.D. thesis, Saarland University.

Gnad, D.; and Hoffmann, J. 2018. Star-Topology Decoupled State Space Search. *Artificial Intelligence*, 257: 24–60.

Hall, D.; Cohen, A.; Burkett, D.; and Klein, D. 2013. Faster Optimal Planning with Partial-Order Pruning. In Borrajo, D.; Kambhampati, S.; Oddi, A.; and Fratini, S., eds., *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013)*, 100–108. AAAI Press.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

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

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the ACM*, 61(3): 16:1–63.

Helmert, M.; and Röger, G. 2008. How Good is Almost Perfect? In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, 944–949. AAAI Press.

Ibaraki, T. 1977. The Power of Dominance Relations in Branch-and-Bound Algorithms. *Journal of the ACM*, 24: 264–279.

Jaszkiewicz, A.; and Lust, T. 2018. ND-Tree-Based Update: A Fast Algorithm for the Dynamic Nondominance Problem. *IEEE Transactions on Evolutionary Computation*, 22(5): 778–791.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Sierra, M. R. 2013. Improving heuristic search algorithms by means of pruning by dominance. Application to scheduling problems. *AI Communications*, 26: 323–324.

Torralba, Á. 2017. From Qualitative to Quantitative Dominance Pruning for Optimal Planning. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 4426–4432. IJCAI.

Torralba, Á. 2021. On the Optimal Efficiency of A* with Dominance Pruning. In Leyton-Brown, K.; and Mausam, eds., *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2021)*, 12007–12014. AAAI Press.

Torralba, Á.; Alcázar, V.; Kissmann, P.; and Edelkamp, S. 2017. Efficient Symbolic Search for Cost-optimal Planning. *Artificial Intelligence*, 242: 52–79.

Torralba, Á.; Gnad, D.; Dubbert, P.; and Hoffmann, J. 2016. On State-Dominance Criteria in Fork-Decoupled Search. In Kambhampati, S., ed., *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, 3265–3271. AAAI Press.

Torralba, Á.; and Hoffmann, J. 2015. Simulation-Based Admissible Dominance Pruning. In Yang, Q.; and Wooldridge, M., eds., *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, 1689–1695. AAAI Press.

Torralba, Á.; and Kissmann, P. 2015. Focusing on What Really Matters: Irrelevance Pruning in Merge-and-Shrink. In Lelis, L.; and Stern, R., eds., *Proceedings of the Eighth Annual Symposium on Combinatorial Search (SoCS 2015)*, 122–130. AAAI Press.

Wilhelm, A.; and Torralba, Á. 2025. Conditional Dominance Analysis for Classical Planning. In Lynce, I.; Murano, A.; Vallati, M.; Villata, S.; Chesani, F.; Milano, M.; Omicini, A.; and Dastani, M., eds., *Proceedings of the 28th European Conference on Artificial Intelligence (ECAI 2025)*, 4905–4912. IOS Press.

Zhang, H.; Salzman, O.; Felner, A.; Kumar, T. K. S.; Ulloa, C. H.; and Koenig, S. 2024. Speeding Up Dominance Checks in Multi-Objective Search: New Techniques and Data Structures. In Felner, A.; and Li, J., eds., *Proceedings of the 17th Annual Symposium on Combinatorial Search (SoCS 2024)*, 228–232. AAAI Press.