Dynamic Tree Databases in Automated Planning

Oliver Joergensen, Dominik Drexler, Jendrik Seipp

Linköping University, Sweden
oliver.harold.joergensen@liu.se, dominik.drexler@liu.se, jendrik.seipp@liu.se

Abstract

A central challenge in scaling up explicit state-space search to large tasks is compactly representing the set of generated states. Tree databases, originally developed for model checking, can achieve constant space per generated state in the best case by sharing common substructures across stored states, but require a large up-front memory allocation that limits their applicability when the size of the reachable state space is unknown. We propose a *dynamic* variant of tree databases that compresses state sets over propositional and numeric variables, and supports the operations needed by standard search algorithms. We prove that it supports insertion in amortized linear time in the sequence length, and we describe the data structures and engineering choices that make the approach practical. Our empirical evaluation on classical planning tasks shows compression ratios of up to two orders of magnitude, with the largest savings observed on tasks with many state variables. In grounded planning, tree databases do not outperform a compact hash set in general, but reach compression ratios similar to those in lifted planning once the state size is large. In numeric planning, they yield mixed coverage results but slightly improve the memory score. In lifted classical planning, where states are large and structured, the resulting memory savings improve coverage across all evaluated search configurations despite a modest runtime overhead.

1 Introduction

Explicit state-space search is a widely used technique for solving classical and numeric planning tasks (e.g., Hoffmann and Nebel 2001; Richter and Westphal 2010; Lipovetzky and Geffner 2012; Scala et al. 2016; Seipp, Keller, and Helmert 2020; Kuroiwa et al. 2022). It iteratively expands states, starting from the initial state, to generate their successors, and is often guided by a goal-directed heuristic to reach a goal state. Even with strong heuristics, the set of generated states can be huge. Compactly representing this set is a fundamental challenge, as it enables the storage and exploration of more states within limited memory.

One approach for compressing state sets is to use *tree databases* (Blom et al. 2008). They are a family of data structures that originate from model checking and support efficient state insertion and lookup operations, which are essential for explicit search. A tree database is a forest of trees, each encoding a sequence over a finite alphabet. In automated planning, the alphabet can consist of finite-domain variables, indices of propositional variables or values of numeric variables. Shared subtrees enable significant memory savings, and in the best case, the space required per state approaches the information-theoretical lower bound when successor states differ from the parent state on average in only a single variable (Laarman 2019). These properties make tree databases particularly interesting for automated planning, where each ground action typically changes only a few state variables.

Previous work implements tree databases on top of statically sized *hash ID maps* (e.g., Darragh, Cleary, and Witten 1993; Blom et al. 2008; Laarman, van de Pol, and Weber 2011) or compact hashing (Cleary 1984). A hash ID map is a hash table that maps keys to identifiers assigned by the data structure itself. The main drawback is that statically sized tree databases require manual memory management with potentially large preallocations, which are difficult to estimate in advance. This limitation reflects the state of hash-table technology at the time the original tree database was proposed (Blom et al. 2008), before later advances in dynamically resizable hash tables that retain compact addressing and high lookup throughput (Barnat et al. 2015).

In this paper, we propose and evaluate a new variant of tree databases that addresses these drawbacks by dynamically resizing during execution. We build it on top of off-the-shelf dynamic hash table libraries, which keeps the implementation simple while remaining fast and memory-efficient. It acts as a drop-in replacement for a standard state set, requiring only that each state encoding be decomposed into fixed-width words. We conduct an extensive empirical evaluation on classical and numeric planning benchmarks, covering both grounded and lifted settings. Specifically, we evaluate our method on three representative state encodings: the finite-domain representation used for grounded planning in Fast Downward (Helmert 2006), the sparse propositional representation and the dense numeric representation used for lifted planning in Tyr (Drexler, Joergensen, and Seipp 2026).

Our empirical evaluation reveals significant reductions in state memory usage for the lifted case across all evaluated search configurations, resulting in fewer out-of-memory failures and a higher number of solved tasks despite a modest runtime overhead. We observe compression ratios of up to two orders of magnitude, demonstrating that tree databases are a practical technique for scaling explicit state-space search in automated planning to larger tasks. For the grounded setting,

our approach does not perform better in aggregate. However, the compression ratio of the state set using tree databases grows near-linearly with the number of atoms, indicating that tree databases become increasingly competitive as state size grows. In the lifted numeric setting, tree databases do not improve overall performance. Our key contributions are:

1. We introduce tree databases as a state compression technique for explicit state-space search in automated planning, covering both propositional and numeric state variables.
2. We propose a *dynamically* sized tree database variant that compresses sequences z of k elements over arbitrary finite alphabets with amortized $\Theta(k)$ insertion time, removing the need for manual preallocation.
3. We empirically demonstrate the effectiveness of tree databases on classical and numeric benchmark sets.

2 Related Work

We now discuss compression techniques for state-space search. Throughout, we assume a RAM model with w-bit words and a set of n states, each consisting of k elements from a finite universe.

Optimal Compression of Combinatorial State Spaces. Laarman (2019) analyzes the optimal compression ratio of combinatorial state spaces from an information-theoretical perspective. More precisely, they seek a lower bound on the number of bits required to encode a combinatorial state space. They assume that on average only a single variable changes, chosen uniformly at random, resulting in a lower bound of a single w-bit word plus $\mathcal{O}(\log_2(k))$ bits per state. This result also applies to classical planning, where ground actions usually modify only a small number of state variables.

Finite-Domain Representation (FDR). A set of propositional state variables forms a *mutex group* if at most one of them is true in every reachable state (Helmert 2009; Fišer 2020). A mutex group g can be represented by a single *finite-domain variable* whose values correspond to the propositions in g, plus an additional value *none* for the case where none of them holds. The *packed* FDR of a state s concatenates the binary encodings of all its finite-domain variables, using $\lceil \log_2 d \rceil$ bits per variable with domain size d, rather than a full w-bit word. Its size grows linearly with the number of finite-domain variables, which can still become a limiting factor. Fast Downward uses packed FDR to represent states.

Binary Decision Diagrams (BDDs). Binary Decision Diagrams (BDDs) are canonical graph-based representations of Boolean functions (Bryant 1985) that take the form of directed acyclic graphs. In symbolic search (Burch et al. 1992; Edelkamp and Kissmann 2008; Torralba et al. 2017), BDDs symbolically represent the successor function. Applying the successor function yields sets of states that are also symbolically represented as BDDs. The size of a BDD depends on the ordering of its variables. In the best case, it can represent exponentially many states in space that is polynomial in the number of variables. Finding an optimal variable ordering that minimizes the size of a BDD is NP-complete (Bollig and Wegener 1996). In explicit search, any state needs to be uniquely indexable. BDDs represent a single state as a terminating path from the root node. As there may be multiple such paths, one cannot uniquely identify a state based on a node index, making BDDs inappropriate for explicit search.

Level-Ordered Edge Sequences (LOES). Unlike BDDs, LOES avoids the use of pointers by encoding a binary decision tree as a flat edge sequence ordered by tree depth, leading to a more compact memory layout (Schmidt and Zhou 2011). As with BDDs, its compression efficiency is sensitive to the ordering of state variables. Efficient insertion into LOES requires batching, which amortizes the cost of updating the structure. For a set of n states of size k, LOES requires at most twice the size of the concatenated packed FDR in the worst case and as few as $2n$ bits in the best case, when prefix sharing is maximal and $n > k$. However, LOES is unsuitable for algorithms that incrementally add states, such as explicit search, because of expensive and frequent resizing operations.

Decoupled Search. Decoupled search (Gnad and Hoffmann 2018) exploits the task structure by partitioning the variables into a central component and leaf components, where the leaf components depend solely on the central component. One aspect that makes decoupled search effective is its compact state representation, achieved by combining the state of the central component with a reachability function over the leaf states. These two jointly represent a set of states. The function is a mapping that indicates whether a leaf state is reachable from the initial state via the current central state, and is implemented as a set of unique *IDs* that map to leaf states. In the best case, the state size reduces to a polynomial in the number of leaf factors and the size of the central component; in the worst case, the reachable decoupled state space can be exponentially larger than the explicit one.

3 Background

In this section, we present the background of classical and numeric planning, state set compression and tree databases.

3.1 Classical and Numeric Planning

A (numeric) *planning task* is typically specified in a compact, factored representation, such as a first-order description in the Planning Domain Definition Language (PDDL) (McDermott et al. 1998; Fox and Long 2003) or a grounded SAS$^+$ encoding (Bäckström and Nebel 1995). If a task contains only propositional variables (i.e., no numeric state variables), it is called a *classical planning task*. Each planning task induces a (possibly infinite) state-space model that captures its semantics. In this work, we operate at this semantic level and do not rely on the internal syntactic structure of each action. A *state-space model* is a tuple $\mathcal{S} = \langle S, s_0, S_G, A, App, Succ \rangle$. Here, S is a set of states, where each state is a value assignment to a set of propositional and numeric variables; $s_0 \in S$ is the initial state; $S_G \subseteq S$ is the set of goal states; A is a set of actions; App maps each state s to the set of actions in A applicable in s; and $Succ$ maps each state s and applicable action $a \in App(s)$ to its deterministic successor s'. A

Bits 0..1	Bits 2..3	Bit 4
$v_0 = p_1$	$v_1 =$ *none*	$v_2 = p_5$

(a) Packed FDR of state s in which only propositions p_1 and p_5 hold. Assume that the task has mutex groups $\{p_0, p_1, p_2\}$, $\{p_3, p_4\}$ and $\{p_5\}$, which give rise to the FDR variables v_0, v_1 and v_2, with domains $\{p_0, p_1, p_2, \textit{none}\}$, $\{p_3, p_4, \textit{none}\}$ and $\{p_5, \textit{none}\}$, respectively. (The *none* value indicates that none of the propositions in the mutex group holds.) It sequentially encodes the values of v_0, v_1 and v_2 with 2, 2 and 1 bits, resulting in a total of 5 bits.

Bits 0..31	Bits 32..63	Bits 64..95
size $= 2$	p_1	p_2

(b) Sparse propositional representation of state s, in which only propositions p_1 and p_5 hold. Each field uses a word size w of 32 bits. The first field defines the number of true propositions, followed by the indices of the two propositions, resulting in a total of 96 bits.

Bits 0..63	Bits 64..127
$n_0 = 0.8$	$n_1 = 5.7$

(c) Dense numeric representation of $n_0 = 0.8$ and $n_1 = 5.7$. It sequentially encodes the values of every numeric variable, with 64 bits per variable, resulting in a total of 128 bits.

Figure 1: (Partial) state representations for a task with propositions $p_0, \ldots, p_5$ and numeric variables n_0 and n_1.

state trajectory is a sequence $s_0, s_1, \ldots, s_n$ such that for all $i = 0, \ldots, n-1$ there exists an action $a \in App(s_i)$ with $Succ(s_i, a) = s_{i+1}$. A state trajectory that starts in s_0 and ends in a goal state is a *plan*. We can find a plan by running a search, i.e., starting from the s_0, we iteratively apply actions to generate successor states until we reach $s \in S_G$.

We make the closed-world assumption, meaning that atoms that are not true in a state are implicitly assumed to be false. This leads to what we call the *sparse propositional representation*, where a state is represented as the set of true atoms.

In planning, we distinguish between two different search modes. First, in *lifted planning*, we compute the applicable actions $App(s)$ from a given lifted first-order logic representation of the actions on the fly (Corrêa et al. 2020; Ståhlberg 2023). Second, in *grounded planning*, we translate the lifted representation into a grounded representation, where each atom corresponds to a Boolean state variable. In addition, we generate mutex groups over these variables to form finite-domain variables, also known as *FDR variables*, where the value of such a variable indicates either the unique atom that is true in the state or that none of the atoms in the mutex group is true (Helmert 2009). Since numeric variables lack a natural default value, we assume a *dense numeric representation*, where each numeric variable uses 64 bits to represent its value. Figure 1 illustrates the different representations.

3.2 State Set Data Structure

An explicit search algorithm requires a *state set* data structure $\mathcal{D}$ for representing the set of currently generated states, along with an $insert(s)$ operation that inserts a newly generated state s into $\mathcal{D}$ and returns an index i and a $lookup(i)$ operation that returns the state with index i. We say an insert or lookup operation is *efficient* if it runs in amortized $\mathcal{O}(|s|)$ time, where $|s|$ denotes the number of represented variables in s.

Beyond runtime efficiency, we are also interested in the space efficiency of a state set data structure. For a given set of states $S' \subseteq S$, a state set data structure $\mathcal{D}$ maps S' via a representation function $rep_{\mathcal{D}}$ to a binary representation $\{0,1\}^*$ with size $|rep_{\mathcal{D}}(S')|$. The compression ratio for a state set S' is the quotient between the size of the uncompressed ($\mathcal{D}$) and the compressed ($\mathcal{D}'$) representation, i.e., $|rep_{\mathcal{D}}(S')|/|rep_{\mathcal{D}'}(S')|$.

Structure	Worst case	Best case
Hash table	kw	kw
Tree database	$2kw - 2w$	$2w + \epsilon w$

Table 1: Theoretical bounds by Laarman (2019) on the representation size of a single sequence of length k over w-bit integers.

3.3 Tree Databases

Actions in planning tasks often change only a few state variables, so a state and its successor typically share many atoms and numeric values. Tree databases exploit these common assignments by encoding each state as a tree whose nodes recursively subdivide the sequence of ground atoms and numeric variable assignments. All stored trees and subtrees together form a forest of unique trees, allowing different states to share subtrees and yielding significant space savings over representations that do not exploit shared structure.

A tree database over a finite alphabet Σ is a forest of binary trees. A *tree* in such a database consists of nodes of the form $v = \langle l(v), r(v) \rangle$, where $l(v)$ and $r(v)$ are either child nodes or elements of Σ. The structure of a tree is defined by a function λ, which maps each node v to the number of leaf positions in the subtree rooted at v. If $\lambda(v) = 1$, then v is a leaf node and both entries are elements of Σ. If $\lambda(v) > 1$, then the tree is *perfectly balanced* iff $\lambda(l(v)) = 2^{\lfloor \log_2(\lambda(v)-1) \rfloor}$ and $\lambda(r(v)) = \lambda(v) - \lambda(l(v))$.[1]

A tree database unambiguously encodes a sequence $z = z_0, z_1, \ldots, z_{k-1}$ consisting of k elements over Σ in a perfectly balanced tree with root node v and $\lambda(v) = \lfloor (k+1)/2 \rfloor$ leaf nodes such that the i-th leaf node is $v_i = \langle z_{2i}, z_{2i+1} \rangle$ with $i = 0, \ldots, \lambda(v) - 1$. If k is odd, z_{k-1} has no partner, and hence we store z_{k-1} directly in $r(v')$ of its parent v' and remove the leaf $v_{\lambda(v)-1}$. A tree database supports operations for inserting and looking up a sequence. Figure 2 illustrates a sequence of insert operations into an initially empty tree database with perfectly balanced trees.

Table 1 reports theoretical per-sequence space bounds by

[1] Blom et al. (2008) originally defined trees as *balanced* iff the number of leaf nodes in the left subtree satisfies $\lambda(l(v)) = \lceil \lambda(v)/2 \rceil$, which is appropriate for fixed-length sequences. The notion of *perfectly balanced* is beneficial for variable-length sequences because it restricts left subtrees to have a power-of-two number of leaf positions. This often creates structurally similar left subtrees, which is a prerequisite for effective node sharing among trees of different lengths (van der Berg 2021).

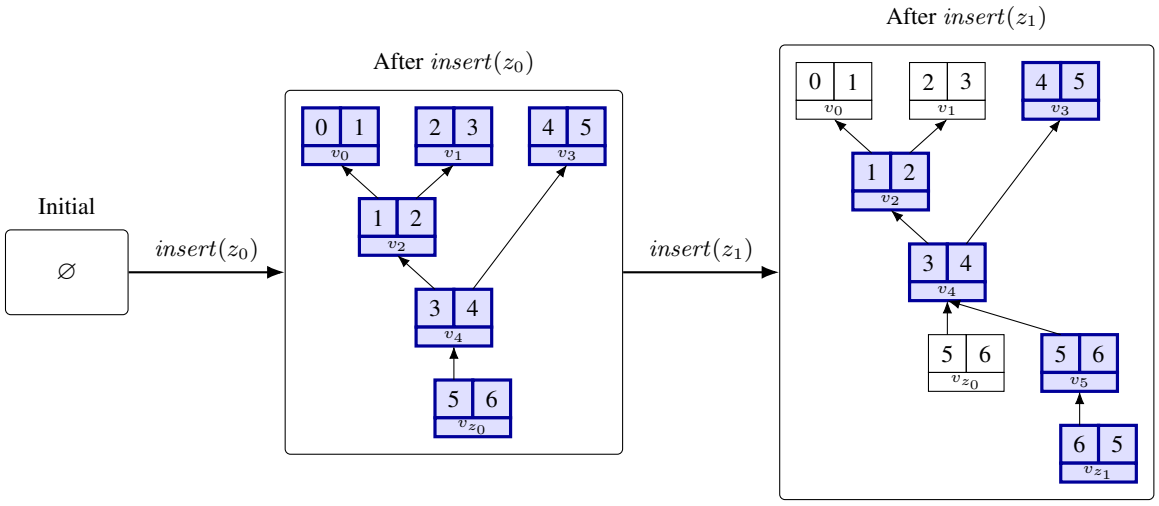

Figure 2: A tree database of perfectly balanced trees after inserting two sequences: $z_0 = \langle 0, 1, 2, 3, 4, 5 \rangle$ and $z_1 = \langle 1, 2, 4, 5, 6 \rangle$. Each node v_i shows its left and right children or values at the top and its index $i = f(l(v_i), r(v_i))$ at the bottom. Shading marks the nodes that make up the tree of the most recently inserted sequence. For z_0, the root has $\lambda(v_{z_0}) = \lfloor (6+1)/2 \rfloor = 3$ leaf positions. Its left subtree therefore has $2^{\lfloor \log_2(3-1) \rfloor} = 2$ leaf positions and its right subtree has $3 - 2 = 1$, so v_0, v_1 and v_3 store the values of z_0 in order. The tree for z_1 shares the subtree rooted at v_4 and the leaf v_3 with the tree for z_0. Because z_1 has odd length, its final element 6 is stored directly in the right entry of the parent node v_5, that is, $r(v_5) = 6$.

Laarman (2019) for tree databases and conventional hash tables, assuming sequences of length k over w-bit integers. In the worst case, the tree database requires roughly twice as much space as the hash table. In the best case, however, it stores a sequence in essentially two integers, independent of k. In particular, when actions modify only a single variable—which is common in planning—the tree database approaches the information-theoretical lower bound of Laarman (2019).

3.4 Tree Database Algorithm

Algorithms 1 and 2 present the read and insert operations of a tree database. Both algorithms operate on an *indexed hash set* f, which is a bijective function that maps keys to unique indices, supporting *getOrInsert* and index-based retrieval via *lookup*. In both algorithms, indices serve a dual role: a leaf index directly encodes an element of Σ, while an inner-node index v refers to a pair of child indices $\langle l(v), r(v) \rangle$ stored in f. The length parameter k determines which interpretation applies, removing the need to distinguish the two cases in the index space itself. This is possible as the structure of the tree is completely derivable from k, as described in Section 3.3.

The read operation (Algorithm 1) reconstructs a sequence top-down: it splits the length at the largest power of two $mid < k$, recursively reads the two subtrees referenced by f and concatenates them, bottoming out at $k = 1$, where the index directly encodes a single element. The insert operation (Algorithm 2) is symmetric but proceeds bottom-up, interning each pair $\langle l(v), r(v) \rangle$ in f to obtain its node index. Both operations run in amortized $\mathcal{O}(|z|)$ time (Blom et al. 2008).

Lookup Structure. The indexed hash set f described above stores inner-node keys $\langle l(v), r(v) \rangle$, enabling nodes to be shared across different trees. To initiate a read or insert operation, a second indexed hash set with keys $\langle v, k \rangle$, where v is a root node and k is the sequence length, serves as the entry point passed to the algorithms above. A third indexed hash set with keys from Σ deduplicates elements of the alphabet that exceed the index bit width of f, which is particularly important for the sparse representation.

Algorithm 1 Read

Require: $v \geq 0, k \geq 1$, Indexed Hash Set f
Ensure: $z = \langle z_0, \ldots, z_{k-1} \rangle$
1: **function** READ(v, k, f)
2: **if** $k = 1$ **then**
3: **return** $\langle v \rangle$
4: $\langle l(v), r(v) \rangle \leftarrow f.lookup(v)$
5: $mid \leftarrow 2^{\lfloor \log_2(k-1) \rfloor}$ ▷ *Largest* 2^m *smaller than* k
6: **return** READ($l(v)$, mid, f)∘READ($r(v)$, $k-mid$, f)

Algorithm 2 Insert

Require: $z = \langle z_0, \ldots, z_n \rangle$, Indexed Hash Set f
Ensure: $v \geq 0$
1: **function** INSERT(z, f)
2: **if** $|z| = 1$ **then**
3: **return** z_0
4: $mid \leftarrow 2^{\lfloor \log_2(|z|-1) \rfloor}$ ▷ *Largest* 2^m *smaller than* $|z|$
5: $l(v) \leftarrow$ INSERT($\langle z_0, \ldots, z_{mid-1} \rangle$, f)
6: $r(v) \leftarrow$ INSERT($\langle z_{mid}, \ldots, z_n \rangle$, f)
7: **return** $f.getOrInsert(\langle l(v), r(v) \rangle)$

4 Dynamic Tree Databases (DTDB)

We now describe our *dynamic* tree databases, which use *flat hash tables*, also known as Swiss tables, as the underlying data structure (Benzaquen et al. 2018).

4.1 Motivation

Blom et al. (2008) proposed tree databases as statically sized hash ID maps, where the position of a node implicitly becomes its identifier. When used for a state set, the static allocation is problematic because estimating a reasonable size in advance is difficult when other parts of the system also allocate memory. The result is either under-utilization or premature saturation of the state set, depending on whether too much or too little memory was allocated. We therefore propose a *dynamic* tree database that overcomes this limitation by resizing when the database reaches its maximum capacity, building on modern flat hash tables.

4.2 Flat Hash Tables

Flat hash tables are open-addressing hash tables that store keys compactly in a dynamic array of size c, using a hash function h and a maximum load factor of 7/8. In addition to the keys, flat hash tables maintain a parallel array of control bytes, one per key, which store the upper 7 bits of the hash value. Special control byte values (sentinels) are reserved to indicate empty or deleted slots. Hence, each table entry consists of a key and a control byte, resulting in an additional memory cost of one byte per key.

Another key innovation of flat hash tables is the use of control bytes to filter potential key matches. Each control byte acts as a fingerprint, allowing the table to identify likely matches without accessing the actual keys. This overapproximation has a false-positive rate of $\frac{1}{2^7}$ and enables the use of SIMD instructions to compare 16 control bytes in parallel.

When looking up a key k, the algorithm begins at index $h(k) \bmod c$ and scans forward through the control bytes. When a control byte matches, the algorithm compares the actual key, greatly reducing the number of key comparisons and mitigating cache misses. To insert a key k, the algorithm probes forward from $h(k) \bmod c$ to find the first empty control byte. It then stores k in the key array and its 7-bit fingerprint in the control byte array at the same index.

4.3 Construction

We implemented a dynamic tree database, which we aptly call DTDB because of its dynamic allocation and underlying stable indexing hash set. Recall that a tree database consists of indexed hash sets that represent a bijective function f from keys to indices. We split this bijection into two parts. For the backward mapping f^{-1}, we use a dynamic array in which position i stores the pair $\langle l(i), r(i) \rangle$ that defines the node with index i. For the forward mapping f, we use a flat hash table whose entries reference array positions rather than the pairs themselves. Inserting a new node appends its pair to the end of the array if the node does not already exist, with deduplication handled by the flat hash table. This indirection has low overhead, as the probability of the control bytes falsely predicting a matching key is less than 1%. With a maximum load factor of 1.0 for the array and 7/8 for the flat hash table, our implementation avoids excessive unused space. Each stored node requires one w-bit word and one byte of overhead in the flat hash table. When the dynamic array or the flat hash table reaches its maximum load, we grow it by a constant factor $\delta > 1$. Because dynamic arrays with a constant resize factor $\delta > 1$ admit amortized $\mathcal{O}(1)$ insertion, DTDB remains efficient.

Theorem 1. *Assuming expected $\Theta(1)$ insertion into the flat hash table at the chosen load factor, inserting a sequence z of length k into DTDB takes amortized expected $\Theta(k)$ time.*

Proof. Let $T(k)$ denote the number of internal nodes created by INSERT on a sequence of length k. The recursion satisfies $T(1) = 0$ and, for $k > 1$,

$$T(k) = T(mid) + T(k - mid) + 1,$$

where $mid = 2^{\lfloor \log_2(k-1) \rfloor}$. Since $1 \leq mid < k$, every recursive call reduces the sequence length and each internal node corresponds to one proper subsequence of the original sequence. Thus the recursion tree has k leaves and $k - 1$ internal nodes, so $T(k) \in \Theta(k)$. Appending a node to the dynamic array costs amortized $\Theta(1)$ for a constant resize factor $\delta > 1$, and interning that node in the flat hash table costs expected amortized $\Theta(1)$ under the stated hashing assumption. Therefore, inserting the full sequence takes amortized expected $\Theta(k)$ time. □

5 Experiment Setup

We use Fast Downward (Helmert 2006) to evaluate the compression of FDR for grounded planning, and we use Tyr for lifted planning (Drexler, Joergensen, and Seipp 2026) with sparse propositional and dense numeric state representations.

We incorporate FDR variables into tree databases by partitioning the FDR into w-bit words such that no finite-domain variable crosses a word boundary. For the sparse representation, each atom index occupies w bits. For numeric variables, we use the binary64 format (also known as 8-byte double) and deduplicate the resulting 64-bit values in a separate indexed hash set. Inner nodes are shared between propositional and numeric variables.

We use a resize factor $\delta = 2$ for DTDB and a word size w of 32 bits. We limit each planner run to 30 minutes and 4 GiB of memory and conduct all experiments using the Lab toolkit (Seipp et al. 2017) on a compute cluster equipped with Intel Xeon Gold 6130 CPUs.

The Fast Downward configurations are as follows:

- **Hashset-FDR** uses a hashset that stores packed FDR states explicitly.
- **DTDB-FDR** uses a DTDB and bit-packs multiple FDR variables into a 32-bit leaf node entry.

The Tyr configurations are as follows:

- **Hashset-Sparse** uses a hashset that stores the sparse propositional and dense numeric states explicitly.
- **DTDB-Sparse** uses a DTDB with perfectly balanced trees and a separate indexed hash set to represent numeric variable assignments as 8-byte doubles.

We evaluate both the grounded and lifted configurations with A*+h^{blind}, GBFS+h^{FF} and GBFS+h^{GC}. For numeric planning, we report results for A*+h^{blind} and GBFS+h^{GC}, as numeric h^{FF} is not yet implemented in Tyr.

5.1 Benchmarks

We consider 6495 classical planning tasks from 166 domains. This benchmark set contains all 1847 STRIPS (Fikes and Nilsson 1971) and all 1011 ADL (Pednault 1989) tasks from the Optimal Tracks of the International Planning Competition (IPC) 1998–2023,[2] the 900 STRIPS tasks from the Learning Track of the IPC 2023, the 992 tasks for transforming quantum circuits into CNOT-only layouts (Shaik and van de Pol 2024), the 269 tasks from the Airbus Beluga challenge in 2024,[3] the 1058 hard-to-ground tasks (some of which use conditional effects),[4] the 223 ADL tasks from the Pushworld domain, which represents reasoning tasks involving differently shaped objects in a grid world,[5] and the 195 object-centric ADL tasks from the Minecraft domain, which produce millions of ground atoms (Hill et al. 2024). Furthermore, we consider 595 numeric planning tasks from 36 domains. The numeric benchmark set contains the 400 tasks with numeric variables from the Numeric Planning Track of the IPC 2023[6] and the 195 object-centric numeric tasks from the Minecraft domain (Hill et al. 2024).

5.2 Performance Metrics

For a planner configuration, we aggregate the results across all benchmark tasks with the following metrics:

- **#C** is *coverage*, i.e., the number of solved tasks.
- **#M** is the number of tasks that run out of memory.
- **#T** is the number of tasks that run out of time.
- **MS** is the sum of *memory scores* over all benchmark tasks.

For a planner run, the *memory score* is 0 if the planner runs out of time or exceeds the 4 GB memory limit. The memory score is 1 for planner runs that use at most 2 MiB. In between those two extremes, we interpolate the memory score logarithmically: $\text{MS}(m) = 1 - {}^{\ln(m)-\ln(L)}/_{\ln(U)-\ln(L)}$, where m is the memory usage of the state set, $L = 2\,\text{MiB}$ and U is the overall memory limit (4 GB), clamped to the interval $[0, 1]$. The memory score provides more granular insight into memory usage than #M.

6 Experiment Results

Table 2 presents results across the classical configurations, while Table 3 summarizes numeric lifted configurations.

Grounded Planning. Hashset outperforms DTDB on both coverage and memory score in every grounded configuration. The margins are small for the GBFS configurations (3971 vs. 3968 for GBFS+h^{FF}; 3778 vs. 3776 for GBFS+h^{GC})

Configuration	Storage	#C	#M	#T	MS
Grounded (Fast Downward)					
GBFS+h^{GC}	Hashset	**3778**	2527	178	**1431**
	DTDB	3776	2479	228	1428
GBFS+h^{FF}	Hashset	**3971**	1564	948	**1533**
	DTDB	3968	1538	977	1530
A*+h^{blind}	Hashset	**2108**	4145	230	**783**
	DTDB	2066	4049	368	766
Lifted (Tyr)					
GBFS+h^{GC}	Hashset	3514	2461	520	2120
	DTDB	**3615**	1529	1351	**2234**
GBFS+h^{FF}	Hashset	3885	1608	1002	2416
	DTDB	**3944**	1207	1344	**2510**
A*+h^{blind}	Hashset	1828	4055	612	987
	DTDB	**1871**	3026	1598	**1050**

Table 2: Results on the 6495 classical planning tasks.

Configuration	Storage	#C	#M	#T	MS
GBFS+h^{GC}	Hashset	**282**	230	83	**163**
	DTDB	277	277	41	160
A*+h^{blind}	Hashset	**217**	319	59	127
	DTDB	216	285	94	**129**

Table 3: Results for the Tyr lifted configurations on the 595 numeric tasks.

but more pronounced for A*+h^{blind} (2108 vs. 2066). This is consistent with Table 4, which shows a median FDR state size of just 8 bytes. States this small fit into a single tree node, leaving no shared prefixes to exploit. The per-lookup tree-traversal overhead then dominates, making DTDB both slower and less memory-efficient. For A*+h^{blind}, DTDB times out more often than Hashset (368 vs. 230). The grounded time plot in Figure 3 makes this visible: the hollow markers along the top edge mark the 47 tasks that Hashset solves but DTDB does not, against only 5 in the opposite direction. DTDB runs out of memory on fewer tasks (4049 vs. 4145) and instead keeps expanding states until it reaches the time limit, while its higher per-operation cost pushes some tasks that Hashset solves just within the time limit over it. On the tasks both configurations solve, their runtimes remain comparable, suggesting the overhead is negligible.

Lifted Classical Planning. The picture reverses for lifted search on classical tasks. DTDB achieves higher coverage and a better memory score in all three configurations. For A*+h^{blind}, DTDB gains 43 tasks (1871 vs. 1828) and reduces out-of-memory failures by over 1000 (3026 vs. 4055), at the cost of more timeouts. This is a direct consequence of spending the recovered memory on additional search. With GBFS+h^{FF}, the coverage gain grows to 59 tasks (3944 vs. 3885) and the memory score improvement to 94 points (2510

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

[3] https://github.com/TUPLES-Trustworthy-AI/Beluga-AI-Challenge

[4] https://github.com/abcorrea/htg-domains

[5] https://github.com/google-deepmind/pushworld

[6] https://github.com/ipc2023-numeric/ipc2023-dataset

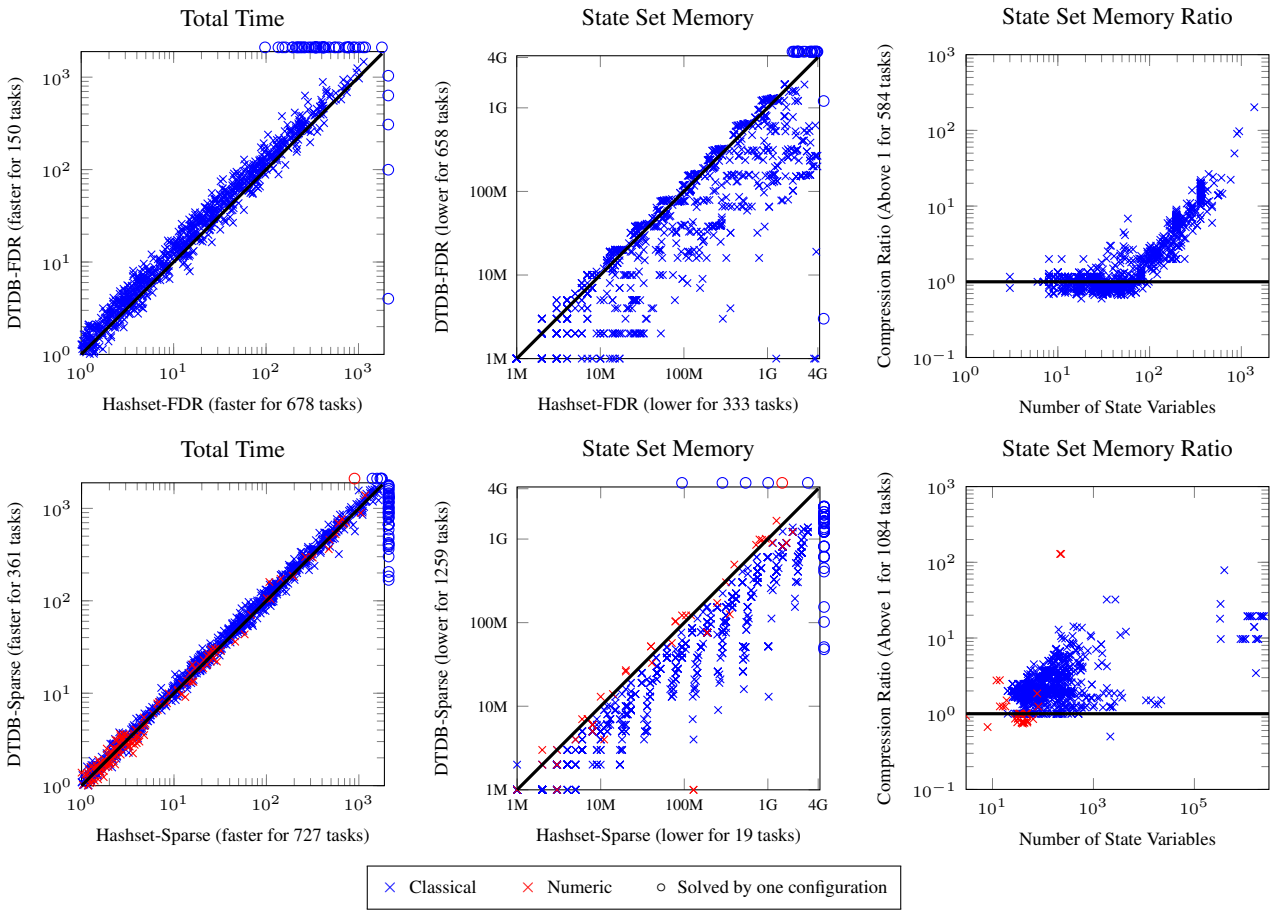

Figure 3: Comparison of Hashset and DTDB (ours) for grounded search with Fast Downward (top row) and lifted classical and numeric search with Tyr (bottom row) using A*+h^{blind}. Columns show the total search time in seconds (left), peak state-set memory in MB (center) and the ratio of Hashset to DTDB state memory against state size (right). For the cross markers in the time plots, we report only tasks that both configurations solve in at least one second, as shorter times are dominated by measurement noise. Hollow markers denote tasks solved by only one configuration; we place them just beyond the axis of the failing configuration, at its resource limit. In the time and memory plots, points above the diagonal indicate instances where DTDB uses more time or memory. In the ratio plots, points above $y{=}1$ indicate instances where Hashset uses more state memory than DTDB.

Quantiles	Min	Q1	Median	Q3	Max
Bytes	4	4	8	28	4804

Table 4: Distribution of bytes required for the FDR state representation aggregated over all groundable classical tasks.

vs. 2416). With GBFS+h^{GC}, the gains are 101 tasks (3615 vs. 3514) and 114 memory-score points (2234 vs. 2120). The reduction in out-of-memory failures across all lifted classical configurations supports the view that the memory savings from DTDB contribute to the additional solved tasks.

Numeric Lifted Planning. Table 3 shows a more mixed picture for numeric tasks. For GBFS+h^{GC}, Hashset achieves slightly higher coverage (282 vs. 277) and a marginally better memory score (163 vs. 160). DTDB times out on fewer instances (41 vs. 83) but runs out of memory more often (277 vs. 230), suggesting that the tree-database overhead outweighs its memory savings in this configuration. For A*+h^{blind}, the two approaches are nearly tied in coverage (217 vs. 216), but DTDB reduces out-of-memory failures (285 vs. 319) and attains a slightly higher memory score (129 vs. 127), at the cost of more timeouts (94 vs. 59). Taken together, the numeric results suggest that dynamic tree databases improve memory behavior in this setting only slightly, and the net effect on coverage is small. These results therefore support a weaker claim than the lifted classical case: tree databases are competitive for numeric planning, but they do not dominate Hashset across the board.

Detailed Analysis. Figure 3 examines A*+h^{blind} in detail. For grounded search (top row), DTDB achieves comparable speed while using *less* state memory on 658 of 1920 jointly

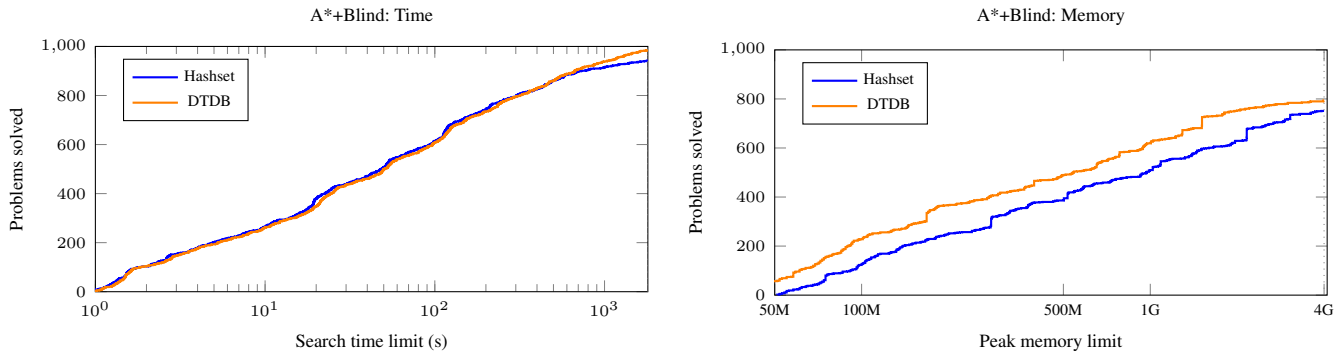

Figure 4: Cumulative coverage as a function of time limit and memory limit for lifted A*+h^{blind}. The dotted vertical line marks the actual resource limit (30 minutes / 4 GiB).

solved instances. Hashset still wins in aggregate, likely due to its faster operations. Notably, the rightmost grounded plot shows that the compression ratio of DTDB over Hashset grows with state size, so tree databases become competitive in the large-state regime, where FDR packing alone is insufficient. For lifted search (bottom row), with numeric results overlaid in red, the runtime overhead remains modest overall: Hashset is faster on 727 tasks while DTDB is faster on 361. The state-set memory advantage, however, is significant: DTDB uses less state-set memory on 1259 tasks versus only 19 for Hashset. The hollow markers along the right edge of both plots are the 48 tasks that DTDB solves but Hashset does not, against only 5 in the opposite direction. The rightmost lifted plot shows that compression ratios approach two orders of magnitude on some instances, with the vast majority of instances achieving a compression ratio above 1.

Figure 4 shows cumulative coverage for lifted A*+h^{blind} with trivially solved tasks filtered out. For the time plot, we exclude tasks that both configurations solve in under one second; for the memory plot, we exclude tasks that both configurations solve using under 50 MiB. The two approaches remain close across the full time budget, whereas DTDB stays above Hashset across nearly the entire memory range. The memory advantage of tree databases is therefore present throughout the search, not only at the resource limit.

7 Conclusions

We proposed a dynamic variant of tree databases for encoding sequences over an arbitrary finite alphabet. We applied it to classical and numeric planning benchmarks and empirically compared it against three standard state representations: FDR, sparse propositional and dense numeric. We observed compression ratios of up to two orders of magnitude in the sparse setting. In lifted classical planning, these memory savings came with a modest runtime overhead and yielded fewer out-of-memory failures together with higher coverage across all three evaluated search algorithms. In numeric planning, dynamic tree databases improved memory behavior, but the effect on coverage was mixed and configuration-dependent. For grounded planning, Hashset outperformed DTDB in aggregate because FDR states tend to be small; however, the compression ratio grew near-linearly with the number of atoms and DTDB became increasingly competitive as state size grew. We conclude that dynamic tree databases are particularly effective for lifted classical planning and less so for lifted numeric planning. For grounded planning, the aggregate results favor Hashset, but the observed memory trend suggests that DTDB would prevail on sufficiently large tasks.

Acknowledgements

We would like to thank Elliot Gestrin and the anonymous reviewers for providing valuable feedback on an earlier draft of this work. This work was supported by the Swedish Foundation for Strategic Research and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement no. 2022-06725.

References

Bäckström, C.; and Nebel, B. 1995. Complexity Results for SAS$^+$ Planning. *Computational Intelligence*, 11(4): 625–655.

Barnat, J.; Rockai, P.; Still, V.; and Weiser, J. 2015. Fast, Dynamically-Sized Concurrent Hash Table. In Fischer, B.; and Geldenhuys, J., eds., *Model Checking Software - 22nd International SPIN Symposium, Beijing, China, July 24–26, 2015, Proceedings*, volume 9263 of *LNCS*, 49–65. Springer.

Benzaquen, S.; Evlogimenos, A.; Kulukundis, M.; and Perepelitsa, R. 2018. Swiss Tables and absl::Hash. Abseil Blog.

Blom, S.; Lisser, B.; van de Pol, J.; and Weber, M. 2008. A Database Approach to Distributed State-Space Generation. *Electronic Notes in Theoretical Computer Science*, 198(1): 17–32.

Bollig, B.; and Wegener, I. 1996. Improving the Variable Ordering of OBDDs Is NP-Complete. *IEEE Transactions on Computers*, 45(9): 993–1002.

Bryant, R. E. 1985. Symbolic Manipulation of Boolean Functions Using a Graphical Representation. In *Proc. DAC 1985*, 688–694.

Burch, J. R.; Clarke, E. M.; McMillan, K. L.; Dill, D. L.; and Hwang, L. J. 1992. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2): 142–170.

Cleary, J. G. 1984. Compact Hash Tables Using Bidirectional Linear Probing. *IEEE Transactions on Computers*, C-33(9): 828–834.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation using Query Optimization Techniques. In *Proc. ICAPS 2020*, 80–89.

Darragh, J. J.; Cleary, J. G.; and Witten, I. H. 1993. Bonsai: a compact representation of trees. *Software: Practice and Experience*, 23(3): 277–291.

Drexler, D.; Joergensen, O.; and Seipp, J. 2026. Parallel Lifted Planning via Semi-Naive Datalog Evaluation. arXiv:2605.07584.

Edelkamp, S.; and Kissmann, P. 2008. Limits and Possibilities of BDDs in State Space Search. In *Proc. AAAI 2008*, 1452–1453.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *AIJ*, 2: 189–208.

Fišer, D. 2020. Lifted Fact-Alternating Mutex Groups and Pruned Grounding of Classical Planning Problems. In *Proc. AAAI 2020*, 9835–9842.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR*, 20: 61–124.

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

Helmert, M. 2006. The Fast Downward Planning System. *JAIR*, 26: 191–246.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *AIJ*, 173: 503–535.

Hill, W.; Liu, I.; Koch, A. D. M.; Harvey, D.; Kumar, N.; Konidaris, G.; and James, S. 2024. MinePlanner: A Benchmark for Long-Horizon Planning in Large Minecraft Worlds. In *ICAPS Workshop on the International Planning Competition*.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *JAIR*, 14: 253–302.

Kuroiwa, R.; Shleyfman, A.; Piacentini, C.; Castro, M. P.; and Beck, J. C. 2022. The LM-Cut Heuristic Family for Optimal Numeric Planning with Simple Conditions. *JAIR*, 75: 1477–1548.

Laarman, A. 2019. Optimal compression of combinatorial state spaces. *Innovations in Systems and Software Engineering*, 15(3): 235–251.

Laarman, A.; van de Pol, J.; and Weber, M. 2011. Parallel Recursive State Compression for Free. In *Proc. SPIN 2011*, 38–56.

Lipovetzky, N.; and Geffner, H. 2012. Width and Serialization of Classical Planning Problems. In *Proc. ECAI 2012*, 540–545.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language – Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Yale University.

Pednault, E. P. D. 1989. ADL: Exploring the Middle Ground between STRIPS and the Situation Calculus. In *Proc. KR 1989*, 324–332.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *JAIR*, 39: 127–177.

Scala, E.; Haslum, P.; Thiebaux, S.; and Ramirez, M. 2016. Interval-Based Relaxation for General Numeric Planning. In *Proc. ECAI 2016*, 655–663.

Schmidt, T.; and Zhou, R. 2011. Succinct Set-Encoding for State-Space Search. In *Proc. AAAI 2011*, 87–92.

Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated Cost Partitioning for Optimal Classical Planning. *JAIR*, 67: 129–167.

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

Shaik, I.; and van de Pol, J. 2024. Optimal Layout-Aware CNOT Circuit Synthesis with Qubit Permutation. In *Proc. ECAI 2024*, 4207–4215.

Ståhlberg, S. 2023. Lifted Successor Generation by Maximum Clique Enumeration. In *Proc. ECAI 2023*, 2194–2201.

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

van der Berg, F. I. 2021. Recursive Variable-Length State Compression for Multi-core Software Model Checking. In *Proc. NFM 2021*, 340–357.