Dynamic Tree Databases in Automated Planning

Compressing the set of generated states in explicit search

Oliver Joergensen Dominik Drexler Jendrik Seipp

HSDIP 2026

Linköping University, Sweden

The memory cost of explicit search

- Explicit search requires storing visited states for deduplication
- The state space grows exponentially with the variable count
- Thus on larger instances memory becomes a real bound
  - Largest instance we tested has a single-state size of 10.5 MiB
    - MinePlanner [3], domain `Scaled_Move_to_Location`, instance `Scaled_Move_to_Location_74`

Types of State Encoding

Packed FDR (grounded)

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

Sparse propositional (lifted)

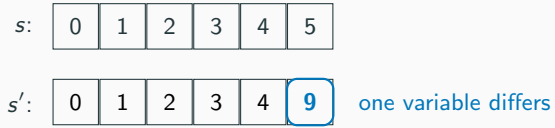
Bits 0..31	Bits 32..63	Bits 64..95
$\text{size} = 2$	p_1	p_2

Dense numeric

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

Key observation from Laarman et al. [5]

When only a few variables change between successors, most of the stored successor is redudant information



A state and its successor often share most variable assignments.

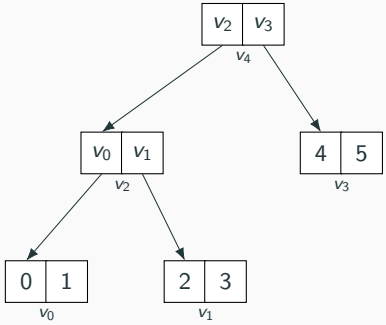
Meaning most explicitly stored states repeatedly write seemingly redudant information.

State Compression via Tree Databases

A tree database encodes each state as a balanced binary tree:

- leaves hold pairs of values,
- inner nodes hold pairs of child indices,
- every distinct node is stored once in a hash set.

Equal subtrees are shared across states $\Rightarrow$ memory savings.



Tree for $z_0 = \langle 0, 1, 2, 3, 4, 5 \rangle$

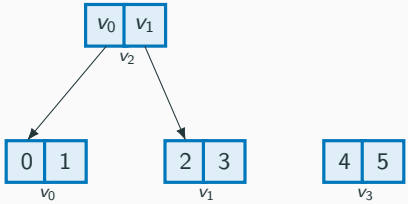
Inserting a state, step by step (1/3)

Insert $z_0 = \langle 0, 1, 2, 3, 4, 5 \rangle$ bottom-up, interning each node.

Pair adjacent values into leaves v_0, v_1, v_3 and intern them.

Inserting a state, step by step (2/3)

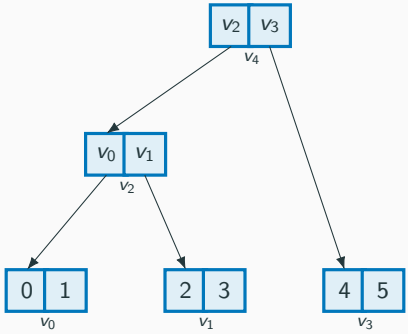
Insert $z_0 = \langle 0, 1, 2, 3, 4, 5 \rangle$ bottom-up, interning each node.



Intern the pair of child indices $\langle v_0, v_1 \rangle$ as a new inner node v_2.

Inserting a state, step by step (3/3)

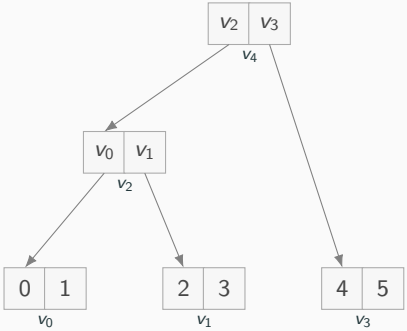
Insert $z_0 = \langle 0, 1, 2, 3, 4, 5 \rangle$ bottom-up, interning each node.



Intern the root $v_4 = \langle v_2, v_3 \rangle$. The forest now stores 5 nodes.

Now insert a similar state

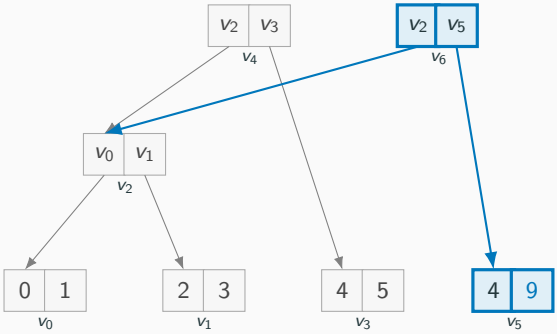
Insert $z_1 = \langle 0, 1, 2, 3, 4, 9 \rangle$ (only the last variable changed).



The left subtree v_2 (and v_0, v_1) already exists in the forest.

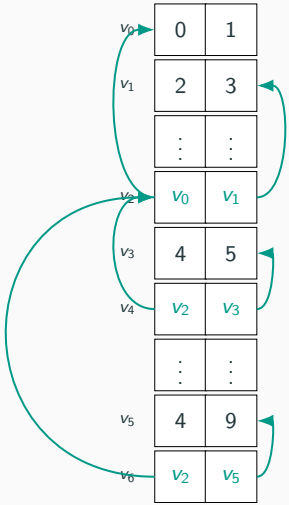
Now insert a similar state

Insert $z_1 = \langle 0, 1, 2, 3, 4, 9 \rangle$ (only the last variable changed).



Only $v_5 = \langle 4, 9 \rangle$ and the root v_6 are new: a whole new state costs 2 nodes.

Data structure



All nodes are stored in the same indexed hashset.

Inner/leaf nodes can be differenciated as the tree structure is known at insertion/read.

The sequence length k is enough to infer the structure, through a deterministic strategy.

Theoretical gain

	worst case	best case
Hash set	kw	kw
Tree database	$2kw - 2w$	$2w + \epsilon$

Space per state, sequence length k, word size w [4].

In the best case, successors differing in one variable, the sequence length stops being a scaling factor.

This is the information-theoretic lower bound for combinatorial state spaces.

Existing Techniques

Existing work implements a custom Tree Database structure which is:

- Statically-sized (Almost impossible size beforehand)
- Complex to implement (Custom/specialized lookup logic)
- Difficult to optimize (Lightning fast response is necessary)

We propose a variant that is instead:

- Dynamically-sized
- Easy to implement (Uses off-the-shelf libraries)
- Still retains an amortized $\Theta(k)$ for insertion (sequence length).

Experiment setup

Planners & encodings

- Fast Downward [2] w. packed FDR (grounded)
- Tyr [1] w. sparse propositional & dense numeric (lifted)

Searches

- A*+h^{blind}, GBFS+h^{FF}, GBFS+h^{GC}

Baseline: compact Hashset
Limits: 30 min / 4 GiB

Benchmarks

6495 classical tasks (166 domains)
595 numeric tasks (36 domains)

IPC, hard-to-ground, Minecraft [3], Beluga, Pushworld, CNOT ...

Question

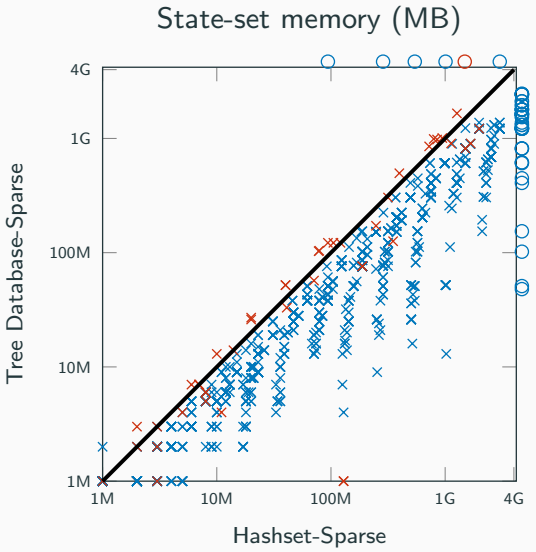
Does sharing pay off, despite tree-traversal overhead?

Coverage

Setting	Search	Hashset #C	Tree Database #C
Lifted classical	GBFS+h^{GC}	3514	**3615**
	GBFS+h^{FF}	3885	**3944**
	A*+h^{blind}	1828	**1871**
Grounded	GBFS+h^{GC}	**3778**	3776
	GBFS+h^{FF}	**3971**	3968
	A*+h^{blind}	**2108**	2066
Numeric (lifted)	GBFS+h^{GC}	**282**	277
	A*+h^{blind}	**217**	216

Coverage (#C).

Lifted memory usage



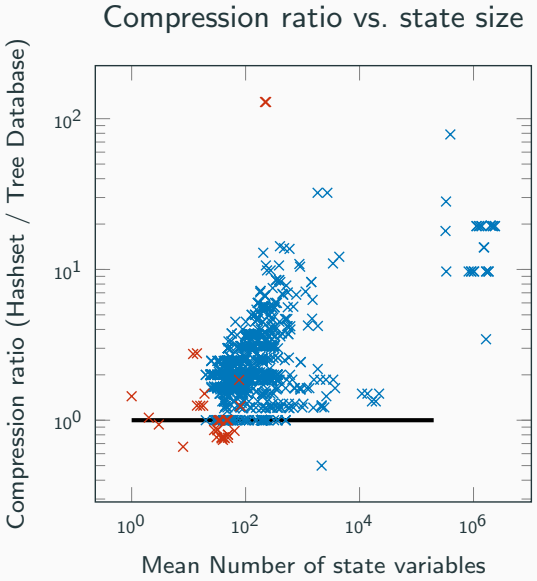
Tree Database uses less state memory on 1259 tasks, Hashset on only 19.

- Fewer OOM failures
- Negliable lookup overhead

Below the diagonal = Tree Database wins.

× classical × numeric

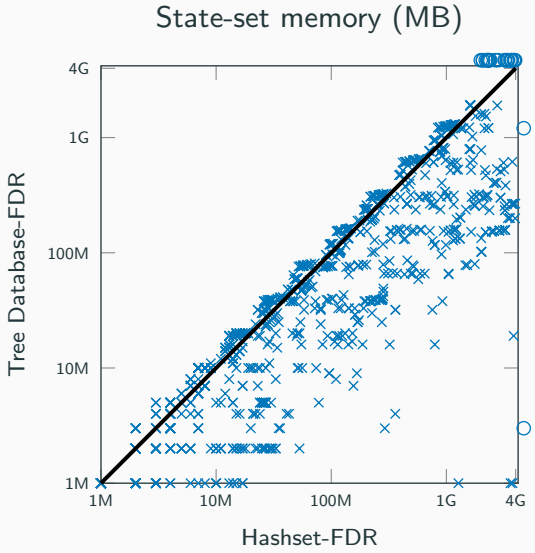
Lifted compression ratio



Lifted A^*+h^{blind}: compression ratio of Hashset over Tree Database against state size.

Above 1 for 1084 tasks; ratios approach two orders of magnitude.

Grounded memory usage



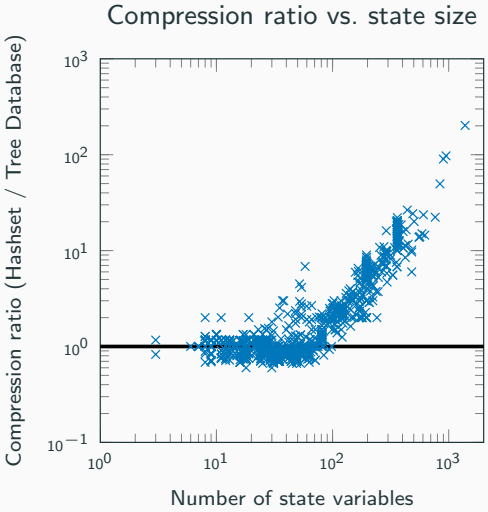
Tree Database uses less state memory on 658 tasks, Hashset on 333; the two tie on the many tasks whose state fits in a single node.

The thin packed-FDR states leave little to share, so the gap is smaller than in the lifted case.

Below the diagonal = Tree Database wins.

× grounded classical

Grounded compression ratio



Median FDR state is just 8 bytes, so it fits in a single node. Effectively, there is nothing to share.

But the compression ratio grows with the number of variables. Tree databases becomes more efficient as state grows, validating the theoretical results.

Takeaways

- Dynamic tree database is fast and memory-efficient on particularly lifted planning.
- Amortized $\Theta(k)$ insertion; up to two orders of magnitude compression.
- Lifted classical planning: higher coverage everywhere.
- Grounded: competitive when states are large.

That's all Folks!

References

[1] Dominik Drexler, Oliver Joergensen, and Jendrik Seipp. Parallel lifted planning via semi-naive Datalog evaluation, 2026. URL `https://arxiv.org/abs/2605.07584`.

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

[3] William Hill, Ireton Liu, Anita De Mello Koch, Damion Harvey, Nishanth Kumar, George Konidaris, and Steven James. MinePlanner: A benchmark for long-horizon planning in large Minecraft worlds. In *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024)*. AAAI Press, 2024.

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

[5] Alfons Laarman, Jaco van de Pol, and Michael Weber. Parallel recursive state compression for free. In Alex Groce and Madanlal Musuvathi, editors, *Proceedings of the 18th International SPIN Workshop (SPIN 2011)*, volume 6823 of *Lecture Notes in Computer Science*, pages 38–56. Springer, 2011.

Backup: Tree Structure Inference

Both operate on an *indexed hash set* f (keys $\leftrightarrow$ indices). The tree shape is fully determined by the length k:

$$mid = 2^{\lfloor \log_2(k-1) \rfloor} \quad \text{(largest power of two} < k\text{)}.$$

- **Read** (top-down): split at *mid*, recurse on both halves, concatenate; bottom out at $k = 1$.
- **Insert** (bottom-up): recurse, then *getOrInsert* the pair $\langle l(v), r(v) \rangle$ to obtain its index.

Odd-length sequences store the final element directly in the parent's right entry.