# Optimizing State Vector Compression for Program Verification by Reordering Program Variables

Simon de Vries
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
s.h.s.devries@student.utwente.nl

## ABSTRACT

When using automated tools for explicit-state model checking, a hash table containing all reachable states is created. This table typically contains billions of states; therefore it requires much memory. However, in a state transition usually only a few variables in a program state change. This causes much redundancy in states. To exploit this redundancy, program states are stored in compressed tree structures. Previous research has shown that a program state can potentially be stored using only 8 bytes on average, yet this optimum is not always reached in practice. Ordering the elements in a state vector differently can reduce the amount of space a compressed tree needs.
Two heuristics to determine a better ordering of elements are introduced and implemented in the LTSmin tool. The heuristics are validated using real-world data. Our experiments demonstrate that on average, the heuristics do not reduce the size of compressed state vectors. Nonetheless, the heuristics provide good compression for some models and can therefore be used to restart the state space exploration using a better variable ordering when the tool has run out of memory.

## Keywords

Model checking; program state vector; tree compression.

## 1. INTRODUCTION

Model checking allows for automatic verification of whether a model of a system satisfies a given set of properties. Two types of properties are distinguished: safety and liveness properties [1]. Although reachability analysis can only check deadlocks and invariants, which are both safety properties, it is 'a key component in model checking algorithms and other verification methods' [6].

Our research focusses on explicit-state reachability analysis using *exhaustive exploration.* When this method is employed, all possible states a model can be in are searched, by iteratively exploring all successor states starting from the initial state [3]. To prevent exploring states which have already been verified, a hash table is maintained in which all states are stored. A model is often specified as a computer program, which may consist of multiple processes that can be executed concurrently. This parallelism results in many possible states the program can be in. A state contains the values of both local and global variables at the current program counter for each process of the program being checked, and the current program counters themselves. For example, consider the program below, which consists of two parallel processes:

| | **Program $P_1$** | | **Program $P_2$** |
|---|---|---|---|
| 1 | $a = 0$ | 1 | $a = 1$ |
| 2 | $b_1 = 0$ | 2 | $b_2 = 0$ |
| 3 | **while** $b_1 < 100$: | 3 | **while** $b_2 < 100$: |
| 4 | $\quad b_1 = b_1 + 1$ | 4 | $\quad b_2 = b_2 + 1$ |

Note that variable $a$ is shared between the two programs, while variables $b_1$ and $b_2$ are both local variables.

Now the state of the parallel processes $P_1$ and $P_2$ can be described using the vector below:

$$\langle b_1, \quad b_2, \quad a, \quad pc_1, \quad pc_2 \rangle \tag{1}$$

In this example, $b_1$ is a variable in $P_1$, $b_2$ is a variable in $P_2$ and $a$ is shared in both programs. The program counters of $P_1$ and $P_2$ are stored in $pc_1$ and $pc_2$, respectively. An example state is:

$$\langle 12, \quad 15, \quad 1, \quad 4, \quad 4 \rangle \tag{2}$$

Using this state, there are several possibilities for the next state. If the program were executed, it is possible that a statement in $P_1$ will be executed first, but it is also possible a statement in $P_2$ will be executed first. However, in model checking, the program is not actually executed. Instead its semantics is interpreted to explore all possible interleavings or non-deterministic execution steps. This way, all possible next states are found and explored. We call the transition of a previous state to a new state a *state transition.* An example of a new state based on the state in Equation (2) is given below:

$$\langle 13, \quad 15, \quad 1, \quad 3, \quad 4 \rangle \tag{3}$$

In this case, variable $b_1$ has been increased by 1, and the program counter of $P_1$ is moved to line 3.

In a real-world situation, many program states have to be stored. This can be in the order of millions or even billions. Each vector itself usually takes only a few hundreds bytes of space. Because so many program states are stored, memory is often the bottleneck for program verification. This problem is known as the *state space explosion* problem, and has been called "the main challenge in model checking" [4].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*21st Twente Student Conference on IT* June 23rd, 2014, Enschede, The Netherlands.
Copyright 2014, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

```
1 type Tree = TreeTable(Tree left, Tree right, Table table, int k) | Nil
2 proc Tree tree_create(k)
3     if (k = 1)
4         return Nil
5     return TreeTable(tree_create(⌈k/2⌉), tree_create(⌊k/2⌋), table_create(2), k)
```

**Algorithm 1. Tree data structure. Taken from [7].**

```
1 proc (int, bool) tree_find_or_put(TreeTable(left,right,table,k), V)
2     assert (|V| = k)
3     (R_l, _) := tree_find_or_put(left, lhalf(V))
4     (R_r, _) := tree_find_or_put(right, rhalf(V))
5     return table_find_or_put(table, ⟨R_l, R_r⟩)

7 proc (int, bool) tree_find_or_put(Nil, V)
8     assert (|V| = 1)
9     return (V[0], _)
```

**Algorithm 2. Tree data structure algorithm for the tree_find_or_put function. Taken from [7].**

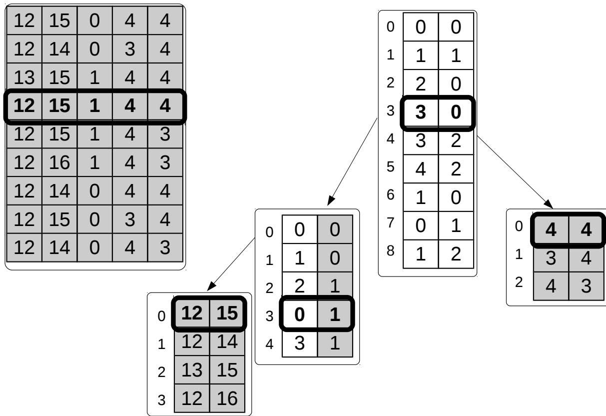


**Figure 1. Example of a state vector stored in a compressed tree.**

## 2. BACKGROUND

To reduce storage requirements for the program states, the program states can be compressed. This is possible because program states typically contain highly redundant information; often, after a state transition, only a few variables are changed. For example, after the transition from the vector in Equation 2 to the vector in Equation 3, only the first and fourth element of the vector are modified. To exploit the similarity between state vectors, the vectors are stored in compressed tree structures [7, 2]. This tree structure is defined in Algorithm 1. Every node in the tree contains a hash table, and may contain left and right children. The call to `table_create(2)` in the `tree_create` method creates a hash table in which tuples can be stored. The `tree_create` method creates a new tree by creating a left and right subtree, and a hash table. In the hash table of each node, tuples of two integers are stored. These integers may contain either the value of a variable, or pointers to elements in the table of a child tree. If a node has a left subtree, the left integer in each tuple will be a pointer to an element in the table of the left subtree (and similarly for a right subtree). Given a suitable implementation, it is possible to use tree compression even for parallel (multicore) model checking [7]. This is for example possible with the LTSmin tool.

Algorithm 2 shows how to put a new state vector $V$ in the tree. The method `tree_find_or_put` takes a compressed tree and a vector $V$, and returns an integer representing the index at which the vector is stored in the hash table of the tree and a boolean indicating whether the vector was contained by the tree already. First the left half of the vector is looked up in the left subtree. It is inserted if it is not yet contained in the left subtree. Then the right half of the vector is looked up or inserted in the right subtree. Finally, the two pointers to both halves of the vector in the subtrees are stored in the table corresponding to the current tree.

Consider the reachable state space of Program $P$, which consists of the parallel processes $P_1$ and $P_2$ (that is, $P = P_1||P_2$). For this simple example, a subset of the total state space is given below:

$$\{\langle b_1, \quad b_2, \quad a, \quad pc_1, \quad pc_2\rangle \\ |b_1, b_2 \in 1 \ldots 100 \wedge a \in \{0,1\} \\ \wedge\, pc_1, pc_2 \in \{3,4\}\} \tag{4}$$

This is not the complete state space; for clarity, the states with $1 \leq pc_1 \leq 2$ or $1 \leq pc_2 \leq 2$ are omitted.

In Figure 1, an example of a compressed tree is shown. The table on the left shows a subset of the set of vectors representing the states of program $P$, as used in the example above. The fourth vector in this table is the same as the vector in Equation 2. The vectors contained in the table are also stored in the tree on the right. In the tables on the right, indices to child hash tables are indicated by white boxes, whereas grey boxes are the leafs of the tree. The grey boxes represent the values of the vector; when read from left to right the original vector is reproduced. Each entry in the root table (the table in the center, containing only white boxes) represents a vector. By following the indices to the child tables, the original vectors can be recovered.

The main advantage of storing the vectors this way is that if variables have the same values in different states, this will automatically be shared. We can see this by calculating the storage requirements for both the table on the left and the tree on the right. We assume all boxes have size $b$ bits. In the table, 9 vectors are stored using 5 boxes. Together, this takes $9 \times 5b = 45b$. The tree takes $9 \times 2b + 5 \times 2b + 4 \times 2b + 3 \times 2b = 42b$. While this

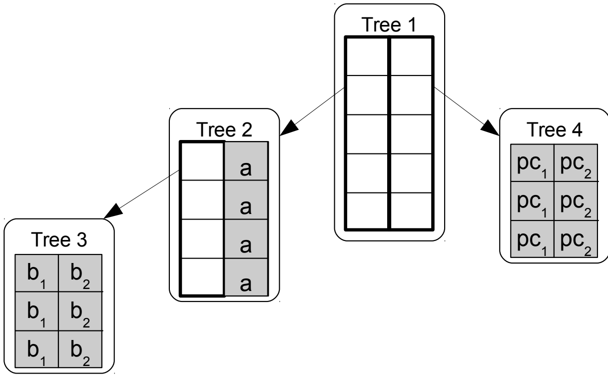


**Figure 2. Tree structure corresponding to the state vector of Equation 1.**

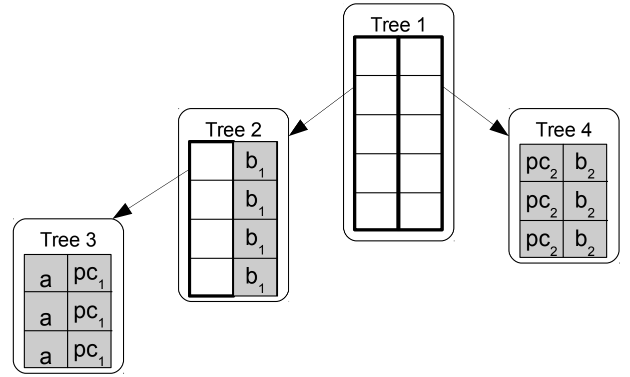


**Figure 3. Tree structure corresponding to the state vector of Equation 5.**

may seem minimal, for larger state vectors the difference is often much larger [7].

To gain some insight into how far state vectors can potentially be compressed, consider the following example. If a table $d$ contains $|d|$ tuples, and its sibling table $e$ contains $|e|$ tuples, then their parent table may contain up to $|d| \times |e|$ entries – all combinations of indices to table $d$ and $e$. The same principle can be applied to all parent and child tables and their siblings. The root table contains one entry for each vector, which costs 8 bytes. Because the root table contains combinations of elements in its child tables, the number of elements in the subtrees may be much smaller. In the best case, the average number of bytes a state vector costs in a compressed tree will be a small constant (approaching to 8 bytes), even for large vectors [7].

## 3. PROBLEM STATEMENT

Unfortunately, in practice it is highly unlikely that a parent table contains all combinations of indices to its child tables. This is because in most real-world programs, some variables (e.g. loop counters) change often, whereas other variables hardly ever change. In the example program given in section 1, the variable $a$ is changed only twice, whereas the loop counters $b_1$ and $b_2$ will change 101 times during the program's execution. This difference results in certain tables containing much more than the optimal number of entries, and some tables containing fewer elements. This in turn results in a suboptimal compression ratio.

To solve this problem, reordering the variables in the state vector has been proposed [7, 2]. By storing variables that often and independently change in different child tables, the tree structure can be used more efficiently. For example, by moving the often-changing variables $b_1$ and $b_2$ to different subtrees, the state vector in Equation (1) can also be stored as:

$$\langle a, \quad pc_1, \quad b_1, \quad pc_2, \quad b_2 \rangle \qquad (5)$$

The corresponding tree structure is shown in Figure 3. Compare this to the organisation in Figure 2, which shows the tree structure corresponding to the ordering of Equation 1.

To compare the storage requirements, the number of elements the hash table in each tree contains is shown in Table 1. The old ordering refers to the variable ordering from Equation 1, and the new ordering refers to the variable ordering from Equation 5. In this case, simply storing variables in the state vectors using a different order saves over 41%.

**Table 1. Comparing the old and new ordering of variables by the number of elements the hash tables of each tree contains.**

| Tree | Old ordering | New ordering |
|---|---|---|
| **Tree 1** | 41616 | 41616 |
| **Tree 2** | 20403 | 410 |
| **Tree 3** | 10202 | 8 |
| **Tree 4** | 16 | 204 |
| **Total** | 72237 | 42238 |

The numbers of the tree refer to the tree numbering in Figure 2 and Figure 3. For example, Tree 4 contained 16 elements using the old ordering, and 204 elements in the new ordering. From Figure 2 and 3, we can decude that this means there are 16 combinations of $(pc_1,\ pc_2)$, and 204 combinations of $(pc_2, b_2)$. This is indeed the case: the combinations of $(pc_1,\ pc_2)$ are $(1 \cdots 4, 1 \cdots 4)$, and the combinations of $(pc_2, b_2)$ are $(\{3, 4\},\ 0 \cdots 100) \cup (\{1, 2\},\ 0)$. The table also shows that Tree 1 contains 41616 elements in both the old and new ordering. This is because the total number of vectors, and therefore the number of elements in the root tree, remains the same after reordering.

The difference in memory requirements using a different order in the state vector can make, brings us to the following research questions:

1. What heuristics are possible to find a better ordering of variables?
2. Can these heuristics be validated and be shown to reduce space usage for all real-world models?

## 4. RELATED WORK

For model checking, [3] provides a good overview of different model checking techniques and their challenges. However, [3] does not discuss any state compression techniques.

A possible state vector compression method is the COLLAPSE method, as described by Holzmann [5]. The COLLAPSE method stores variables of specific processes in seperate hash tables, so-called *process tables*. The elements in the process tables are then referenced by a root hash table.

More recently, a better compression method called 'tree compression' has been developed, and has been described by Laarman *et al.* in [7]. Laarman *et al.* also mention the

specific problem of changing the order of variables in the context of model checking to achieve optimal compression for state vectors, but solving that problem is left for future research.

Blom *et al.* prove that on average, the size of a compressed state vector will stay below a certain constant threshold, regardless of the vector length [2]. Reordering the state vector and changing the tree structure are mentioned as possible ways for improving tree compression, and are explicitly mentioned as possibilities for future work.

This paper presents an approach to improve the results of [7] and [2], by proposing heuristics to determine better variable orderings and validating these heuristics.

# 5. RESEARCH METHOD

In order to automatically find an ordering that yields a better compression ratio, we develop several heuristics. We then analyse whether these heuristics fulfil the requirements outlined in Section 5.2.

## 5.1 Finding heuristics

Our task is to develop heuristics which can find an ordering of the variables such as in Equation 5, such that state vectors stored using the new ordering on average cost less memory than when stored using ordering.

A trivial heuristic for finding a better variable ordering is to simply count the number of insertions in each tree table. After a fixed exploration depth it can then be calculated how the variables could have been ordered in order to achieve a more balanced tree. Finding the best variable ordering is not trivial, given that the number of state vector permutations increases exponentially with the size of the state vector. However, it is possible to obtain a better variable ordering using some simple methods. It is for example possible to put variables that change often in different subtrees. Intuitively, this means these two variables are combined at a higher level, closer to the root tree. The fact that the pairs of these two variables are also only available close to the root tree means that the combinations do not have to 'propagate' through several other levels, saving many elements in the lower trees.

After a better variable ordering has been found, the tree tables are emptied, and filled again using the newly discovered ordering of variables. This trivial heuristic is developed first, and is be used as a starting point and reference for other heuristics.

## 5.2 Validating heuristics

After developing heuristics, we validated them using real-world data provided by the BEEM database. The BEEM database is a database containing benchmarks for model checking software [8]. The most important factor when validating heuristics is space usage: does the heuristic actually decrease space usage for all models, or do specific datasets exist that actually require more space than before?

The second validation criterium is execution time. A heuristic is most useful if a trial run of only a few iterations provides enough information for the heuristic to find a better ordering.

# 6. MATHEMATICAL DESCRIPTION

Given a set of vectors $S$ and an order $O$ such that $|O| \geq 2$, it is possible to calculate the total number of bytes required to store the set of vectors in a tree. First, we define $S_{\{O_1,O_2,\cdots O_m\}}$ as the set of all combinations of $x_{O_1}$, $x_{O_2}, \cdots x_{O_m}$. $S_O$ is the set $S$, projected on the elements in the set $O$. More precisely, given the set $S$ of all state vectors $\langle x_1, \cdots x_n \rangle$, the *subvector set* $S_{O_1,O_2,\cdots O_m}$ $(m \leq n)$ is given by the following definition:

$$S_{O_1,O_2,\cdots O_m} = \{\langle x_{O_1}, x_{O_2}, \cdots x_{k_m}\rangle | \langle x_1, x_2, \cdots x_n\rangle \in S\}$$

The total number of bytes a tree takes up in memory can now be determined by adding the size of the root tree, and its subtrees. For any given order $O$ and set of vectors $S$, the total tree size $n_O(S)$ can be calculated using the following equation:

$$n_O(S) = \begin{cases} 0 & \text{if } |O| < 2 \\ |S_{\{o_1,o_2\}}| & \text{if } O = (o_1, o_2) \\ n_{\mathrm{lhalf}(O)}(S) + |S_O| & \text{if } |O| > 3 \\ +n_{\mathrm{rhalf}(O)}(S) & \end{cases} \quad (6)$$

In this equation, lhalf($O$) and rhalf($O$) compute the left and right half of the ordering vector $O$, respectively. The function 'lhalf' also includes a middle element, if $|O|$ is odd. As an example, $\mathrm{lhalf}((1,2,3,4,5)) = (1,2,3)$ and $\mathrm{rhalf}((1,2,3,4,5)) = (4,5)$.

This is perhaps best explained using the example seen before. In the example shown in Figure 1, using the order given in Equation 1, the root tree corresponds to the set $S_{\{1,2,3,4,5\}} = S$. The left subtree contains the elements in $S_{\{1,2,3\}}$, whereas the right subtree corresponds to $S_{\{4,5\}}$. The leftmost tree contains the combinations of $b_1$ and $b_2$, which are the first two elements in each state vector. Therefore the leftmost tree contains $S_{\{1,2\}}$. The number of bytes this tree takes up in memory can be calculated by adding the number of elements in the tree and all subtrees, multiplied by the number of bytes a tuple of two variables costs. In this example, this size $n_{(1,2,3,4,5)}(S)$ amounts to:

$$\begin{aligned} n_{(1,2,3,4,5)}(S) &= 2(|S_{\{1,2,3,4,5\}}| + |S_{\{1,2,3\}}| \\ &\quad + |S_{\{4,5\}}| + |S_{\{1,2\}}|) \\ &= 2(9+5+3+4) \\ &= 42 \text{ bytes} \end{aligned}$$

Now the problem of finding the ordering that results in an optimal compression ratio is reducible to the problem of finding the ordering vector $O$ for which $n_O(S)$ is minimal.

Even when an optimal ordering has been found, it is possible that the tree does not yet have an optimal structure. For example, we define the set of state vectors $S$ as follows:

$$S = \{\langle i, 1, 1, 1\rangle | 1 \leq i \leq 1000\} \quad (7)$$

If this set $S$ is stored in a tree like in Figure 4a), this will cost $2(1000 + 1000 + 1) = 4002$ bytes. However, if we change the structure of the tree to Figure 4b), this is reduced to $2(1000 + 1 + 1) = 2004$ bytes. This is not a focus for our research, though.

## 6.1 Limitations

Although we have found a mathematical function to determine the memory requirements of a given set and ordering, this function is of limited use so far. This is for three reasons: firstly, calculating $|S_k|$ is an expensive operation. In practice, calculating $|S_k|$ means all vectors in $S$ need to be projected on $k$ and added to a new set.

Secondly, the number of orderings is too large to calculate $n_O(S)$ for all orderings $O$. If this would be possible, we can simply find which ordering $O$ results in the smallest $n_O(S)$ and use that ordering. However, for a vector of length $l$, $l!$

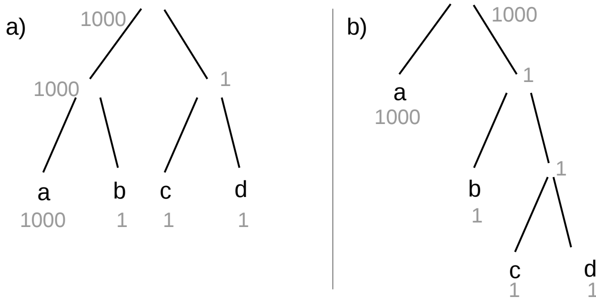


**Figure 4. Example of two different tree structures for the vector $\langle a, b, c, d\rangle$. The numbers in gray represent the number of elements stored in the hash tables at each level.**

different orderings are possible. Even for relatively small values of $l$, the number of different orderings is so large it is infeasible to try them all.

Thirdly, in practice the set $S$ is not known entirely when the new ordering is determined. A limited trial run is performed, which results in a partial vector set. This set does not contain all vectors, but is a subset of the full set $S$. Therefore, even if an optimal ordering $O$ can be found for this partial set, this ordering is not necessarily optimal for the full set of vectors $S$.

A final limitation has to do with the specific implementation of the hash tables in LTSmin. The function $n_O$ does not take *maximal sharing* within a hash table into account. Maximal sharing allows a tuple which is used in the table of multiple trees to be stored only once, in one table [7]. This reduces memory requirements, but this optimization is not taken into account by the function above. The function simply counts duplicate tuples multiple times instead of only one tuple.

# 7. HEURISTICS

In this section, first the implementation of heuristics in general is described. After that, the implementation of our two new heuristics is described in detail, as well as the limitations for both heuristics.

## 7.1 Implementation

Each heuristic requires some data to be able to find a better variable ordering. This data is produced by performing a limited trial exploration, in which the state space exploration algorithm is run for a limited number of iterations. This number of iterations as well as the exploration strategy for the trial run can be varied. When the number of iterations is reached, the exploration algorithm is stopped. Then a method is called that produces a better ordering, given the information provided by the trial exploration. Then the state space exploration algorithm is started again, now using the new ordering.

## 7.2 Counting distinct values

The first heuristic is described in Algorithm 3. The function `get_variables` returns a list of variables in the state vector stored by the tree. For each variable, the number of unique values in the tree are counted. In practice, this could be determined by creating a hash table for each variable before the exhaustive search is started. While storing a state vector in the tree database, each variable is stored in the corresponding hash table too. This allows for efficient retrieval of the number of distinct values of each variable afterwards.

After a list of variables is obtained, the list is sorted by the '`count`'-field the variables. This list is then passed as an argument for '`find_better_ordering`'. This method again returns a list of variables, but now in a potentially better ordering. This list of variables is finally converted to a list of numbers of variables, so the new order can be used by the reordering algorithms.

Variables are ordered in ascending order by their frequency of change. Next, consecutive variables are placed in different subtrees, i.e. the first is placed in the left subtree, the second in the right, and so on. This redistribution of variables over the tree is applied recursively for the subtrees. This way, variables in the state vector that change often will appear in different subtrees. The tree distance[1] between two variables that change often (or change hardly) is maximized.

It can be seen from Algorithm 1 that a tree is constructed by splitting the state vector into two (approximately) equal parts. This is why appending the optimized lists ensures that each item in the first list will appear in the left subtree, and each item in the second list will appear in the right subtree.

Consider the program $P$, as defined in Section 1. First the number of different values that are assigned for each variable is determined. For the variables $b_1$ and $b_2$ this number is 101 ($0 \cdots 100$), $a$ will be assigned 2 values (0, 1), and $pc_1$ and $pc_2$ will both be assigned 4 values ($1 \cdots 4$). Ordering them in ascending order yields:

$$a \quad pc_1 \quad pc_2 \quad b_1 \quad b_2 \tag{8}$$

Now $a$ is put leftmost in the left subtree, which is at position 0 in the state vector. Then $pc_1$ is put leftmost in the right subtree, which is at position 4. The variable $pc_2$ is placed in the right subtree of the left subtree, and $b_1$ is put rightmost in the tree. Finally, $b_2$ is put in the right subtree of the left subtree of the left subtree. The new ordering then becomes:

$$\langle a, \quad b_2, \quad pc_2, \quad pc_1, \quad b_1\rangle \tag{9}$$

In Algorithm 3, is possible to eliminate the recursive call to `find_ordering`. Algorithm 4 shows a non-recursive implementation of this heuristic. It uses the index of each variable in the sorted array to determine its position in the state vector. It takes advantage of the binary representation of each index in the sorted array of variables to calculate each variable's new path in the tree. From these paths, the position in the state vector is derived.

The major drawback of this heuristic is that it fails to take combinations of variables into account. Variables are only ordered by how much distinct values each of them have. This is only optimal if all variables are independent of each other. In practice, this is hardly ever the case. Consider for example the following set of state vectors:

$$S = \{\langle i, i, 1, 1\rangle | 1 \leq i \leq 1000\} \tag{10}$$

Now the number of distinct values will be 1000 for both the first and the second element in the vector. The first variable is therefore put in another subtree than the second variable. However, because both variables are equal in each vector, $|S_{\{1,2\}}|$ is only 1000 (instead of $1000 \times 1000$). This means it would have cost less memory if both variables had been put in the same subtree.

[1]The minimum number of nodes passed when travelling from the first to the second variable in the tree

```
 1 type Tree = TreeTable(Tree left, Tree right, Table table, int k) | Nil
 2 type Variable = (int number, int count)

 4 proc find_better_ordering(Tree tree)
 5    return (
 6       v.number for Variable v in
 7       find_variable_ordering(
 8          order_by_count(
 9             get_variables(tree) // get_variables() returns a list of variables in the state vector stored by the tree
10          )
11       )
12    )

14 Variable[] find_ordering(Variable[] variables)
15     if (|variables| ≤ 1) return variables // only one ordering possible

17     // Now we put the most changing variable in the left subtree,
18     // the variable changing second most in the right subtree,
19     // then the variable changing third most in the left subtree again,
20     // etc.
21     Variable[] left = (variables[i] for even i)
22     Variable[] right = (variables[i] for odd i)

24     return append(find_ordering(left), find_ordering(right))
```

**Algorithm 3. First heuristic to find a better ordering of a state vector. See Section 7.2 for a detailed description.**

```
 1 type Tree = TreeTable(Tree left, Tree right, Table table, int k) | Nil
 2 type Variable = (int number, int count)

 4 proc find_better_ordering(Tree tree)
 5    Variable[] ordered_variables
 6    ordered_variables = order_by_count(
 7       get_variables(tree)
 8    )

10    depth = ⌈log_2 |ordered_variables| ⌉

12    result = (0 ··· n − 1)
13    for v ∈ (0 ··· |ordered_variables| −1):
14       pos = 0 // The current position in the state vector
15       rightOffset = 2^(depth−1) // The offset which will be added to pos if the variable should be added to the right subtree
16       for d ∈ (0 ··· depth − 1):
17          // If bit_set(v, d) = 0, then this variable should be added to the left subtree
18          // Otherwise, it should be added to the right subtree
19          leftOrRight = bit_set(v, d)
20          if (leftOrRight = 0)
21             // Add variable to the left subtree
22             // Don't need to adjust pos, just descend into the left subtree
23          else
24             // Add variable to the right subtree
25             pos = pos + rightOffset
26          rightOffset = rightOffset / 2
27       result[v] = ordered_variables[pos].number

29    return result

31 // Checks if bit bit is set in number number.
32 // Bit 0 is the least significant bit.
33 // An example implementation of bit_set is given below:
34 proc bit_set(int number, int bit)
35    return (number & (1 << bit)) >> bit
```

**Algorithm 4. Non-recursive variant of Algorithm 3**

## 7.3 Swapping paths

A way to improve an existing ordering is to walk the tree, taking the largest subtree at each node. The element in the state vector that is reached this way, is the element that is 'responsible' for most combinations of all elements. Now we walk the tree again, but now taking the smallest subtree at each node. By swapping the variable found this way with the variable that is found by each time taking the largest subtree, a new tree can be created that likely has a higher compression factor. By repeating this process for all subtrees, an even better reordering can be found. This algorithm is described in Algorithm 5.

The main idea for this heuristic is to use the information about combinations of variables found by the the trial exploration. The cardinality of the following sets can be deduced from the tree yielded by the trial exploration:

- $S_{\{1,2\}}$, $|S_{\{3,4\}}|$, $|S_{\{5,6\}}|$, $\cdots$. These sets are stored in hash tables in the lowest level of the tree.
- $S_{\{1,2,3,4\}}$, $|S_{\{5,6,7,8\}}|$, $\cdots$. These sets are stored in hash tables in the second lowest level of the tree.
- $S_{\{1,2,3,4,5,6,7,8\}}$, $|S_{\{9,10,11,12,13,14,15,16\}}|$, $\cdots$. These sets are stored in hash tables in the third lowest level of the tree.
- $\vdots$
- $S$, which is stored in the highest level of the tree, in the root table.

The algorithm uses this information in the tree by finding paths in which many large nodes[2] are passed and paths in which many small nodes are passed. The variables these two paths point to are then swapped. Intuitively, this 'spreads the load' of the tree better. Nevertheless, due to the fact that the information contained by the tree is only partial, this does not necessarily result in the optimal ordering.

# 8. EXPERIMENTAL RESULTS

In order to validate the heuristics presented in Section 7.2 and 7.3, a benchmark was performed. For a number of models, the original memory requirements were measured, and compared to the memory requirements of the new heuristics. The memory requirements were measured in terms of average number of bytes per stored state vector. The results of this benchmark are shown in Figure 5. The benchmark was executed for a total of 238 models in the BEEM-database. For reasons of clarity, only a small sample of these benchmarks are shown in Figure 5. The average has been calculated over all 238 models, though.

In this figure, it is clear that the heuristics does not reduce memory requirements much for most models. On average, the heuristics make hardly any difference. Compared to the original variable ordering, the first heuristic saves on average 0.63 bytes per state, and the second heuristic saves on average 0.06 bytes per state. Still, for some models a significant improvement has been achieved. For example the `elevator.2` model required 19 bytes per state using the original ordering of the state vector, but when using the ordering found by the second heuristic this is reduced by almost 40% to 11.8 bytes per state. This is much closer to the theoretical limit of 8 bytes per state.

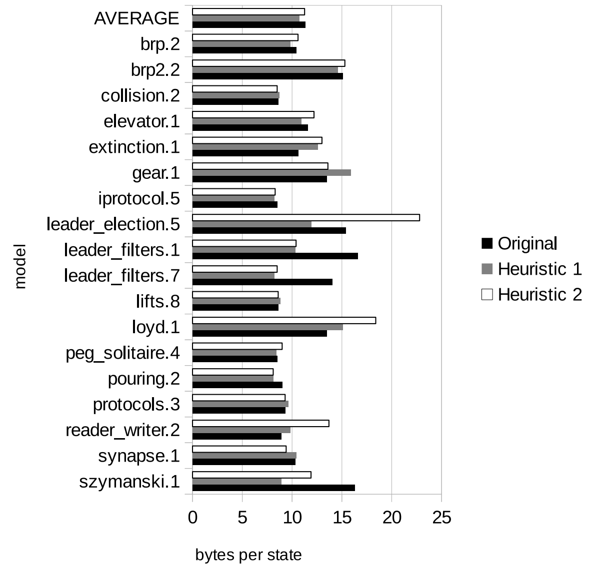


**Figure 5. Benchmarking memory requirements of both heuristics, compared to original memory requirements.**

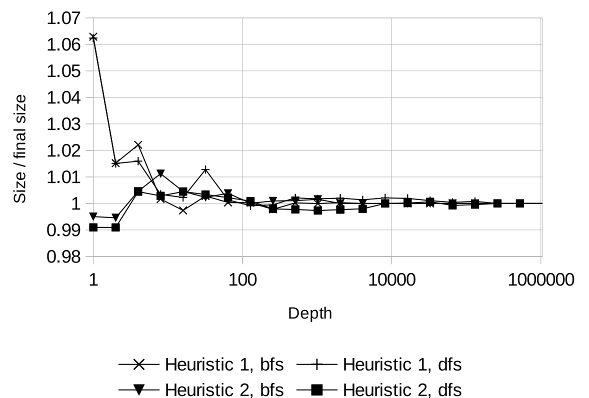


**Figure 6. Diagram showing the average size per state vector over the 'final size' on the vertical axis, and the maximum exploration depth during the trial exploration on the horizontal axis.**

```
1  type Tree = TreeTable(Tree left, Tree right, Table table, int k) | Nil
2  proc find_better_ordering(Tree tree)
3      sorted_combinations = sort_combinations(tree, 0)
4      better_ordering = (0 ··· n − 1)

6      n = |sorted_combinations|
7      for i ∈ (0 ··· ⌊n/2⌋ − 1):
8          swap(
9              better_ordering[sorted_combinations[i]],
10             better_ordering[sorted_combinations[n − i + 1]]
11         )

13     return better_ordering

15 proc sort_combinations(TreeTable(Tree left, Tree right, Table table, int k), int leftmost)
16     if (left = Nil)
17         return leftmost

19     if (right = Nil)
20         return sort_combinations(left, leftmost)

22     if (|left.table| < |right.table|)
23         return append(
24             sort_combinations(left, leftmost),
25             sort_combinations(right, leftmost + ⌈k/2⌉)
26         )
27     else
28         return append(
29             sort_combinations(right, leftmost + ⌈k/2⌉),
30             sort_combinations(left, leftmost)
31         )
```

**Algorithm 5. Second heuristic to find a better ordering of a state vector. See Section 7.3 for a detailed description.**

## 8.1 Trial exploration depth and strategy

For the benchmark above, first a complete trial exploration was performed. After all vectors were added to the tree database, the heuristics were executed and a new exploration was started using the new ordering. In practice, this undermines the goal of the heuristics to reduce memory requirements without increasing execution time too much. When a complete trial exploration is run first, memory for at least the complete old tree database is needed, and the execution time is doubled. Therefore, in a real-world situation it is better to perform a *limited* trial run. The trial run is limited in exploration depth. This raises the question what the minimal exploration depth is for which the heuristics can generate a better ordering. The ordering should not just be better for the limited trial run, but for the full run. This means that ideally, the trial exploration should yield a small set of vectors that is representative for the full set of vectors. Essentially, this is a trade-off between performance and memory usage. A more extensive takes more time, but may result in a better ordering of variables, which may result in less memory usage.

In LTSmin, multiple exploration strategies are implemented. The major strategies are breadth-first search (which is the default strategy) and depth-first search. Selecting a different exploration strategy will change the order in which the state space is explored. Therefore, experiments were performed for both the breadth-first search and the depth-first search strategies.

See Figure 6 for the results of the experiments. We show the size per state vector over the final size, as a function of the maximum exploration depth during the trial exploration. The ‘final size’ is the average size of a state vector after a full trial run, without limits on the depth. All data points are averaged over 193 models from the BEEM database. In this figure, `bfs` refers to breadth-first search, `dfs` refers to depth-first search. The heuristics ‘heuristic 1’ and ‘heuristic 2’ refer to the heuristics described in Section 7.2 and 7.3, respectively.

A good heuristic does not require many iterations in the trial run to be able to find a better ordering. Our experiments indicate that when the trial exploration depth is set to 16, on average the final memory usage deviates less than 0.5% from the memory usage when a full trial exploration would have been performed. For a trial exploration depth of 128, this is reduced to 0.1%.

# 9. CONCLUSION

Our first research question is what heuristics can be used to find a better ordering of variables. We developed two heuristics which try to find a better variable ordering, given the results of a trial run.

Our second research question is whether these heuristics can be experimentally shown to reduce memory usage for all real-world models. In general, our heuristics do not reduce memory requirements significantly. In fact, applying our heuristics often results in increased memory usage. On average, memory usage remains approximately the same when using our heuristics.

However, for some models memory requirements were reduced significantly when using our heuristics. Our heuristics can still be useful for these models. A possible way to use the heuristics is to run the state space exploration algorithm first without our heuristics, and when the machine - or machines - running the exploration algorithm runs out of memory, offer an option to the user to use our heuristic to find a better variable ordering, and try again. The algorithm can then be started again, possibly using less memory and completing the exploration process successfully.

# 10. REFERENCES


[1] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.

[2] S. Blom, B. Lisser, J. Van De Pol, and M. Weber. A database approach to distributed state-space generation. *Journal of Logic and Computation*, 21(1):45–62, 2011.

[3] E. M. Clarke. The birth of model checking. In *25 Years of Model Checking*, pages 1–26. Springer, 2008.

[4] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[5] G. J. Holzmann. State compression in spin: Recursive indexing and compression training runs. In *Proceedings of third international Spin workshop*, 1997.

[6] A. Laarman, J. van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, pages 247–256. FMCAD Inc, 2010.

[7] A. Laarman, J. Van De Pol, and M. Weber. Parallel recursive state compression for free. In *Model Checking Software*, pages 38–56. Springer, 2011.

[8] R. Pelánek. Beem: Benchmarks for explicit model checkers. In *Model Checking Software*, pages 263–267. Springer, 2007.


[2] A large node is a node of which the hash table contains many elements.