Noname manuscript No.
(will be inserted by the editor)

On Commutativity Based Edge Lean Search

Dragan Bošnački[1] · **Edith Elkind**[2] · **Blaise Genest** [3] · **Doron Peled**[4]

the date of receipt and acceptance should be inserted later

Abstract The problem of *state space search* is fundamental to many areas of computer science, such as, e.g., AI and formal methods. Often, the state space to be searched is huge, so optimizing the search is an important issue. In this paper, we consider the problem of visiting all states in the setting where transitions between states are generated by actions, and the (reachable) states are not known in advance. Some of the actions may commute, i.e., they result in the same state for every order in which they are taken. We show how to use commutativity to achieve full coverage of the states, while traversing a relatively small number of edges.

1 Introduction

In many application areas one has to explore a huge state space using limited resources, such as time and memory. Examples include software and hardware testing and verification [3], planning, vision, robotics, as well as problems from several other areas of computer science. In such cases, it is obviously important to optimize the search, exploring only the necessary states and transitions.

If the state space is very large, it is common to represent it implicitly, i.e., by giving an *initial state* and a *transition relation* that produces all successors of a given state. Often, transitions between states are generated by a finite number of *actions*. More specifically, for each state, there can be one or more actions available from this state, and executing an available action leads to another state. In this setting, applying an action from a given state, or even just checking whether this action is available and thus generates a new state, can be costly. On the other hand, in a given state, some actions may be redundant, i.e., applying them would lead to a state that we have already visited or that we will necessarily visit in the future. Thus, we could speed up the state

[1] Dragan Bošnački, Department of Biomedical Engineering, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, the Netherlands
[2] Edith Elkind, School of Electronics and Computer Science, University of Southampton, SO17 1BJ, UK · Nanyang Technological University, Singapore
[3] Blaise Genest, IRISA/CNRS, Campus de Beaulieu, 35042 Rennes Cedex, France
[4] Doron Peled, Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel

space search if we could predict whether a given action is redundant without actually applying it. Exploring fewer actions may also reduce the size of the search stack, and hence the memory consumption. Note, however, that such a prediction algorithm would only be useful if it involved little or no overhead in terms of time and space.

In many of the application areas described above, some of the actions may be independent, i.e., executing a after b has the same result as executing b after a. This situation can be modeled using an *independence relation* on actions: two actions are said to be independent if executing them in any order from a given state leads to the same state. Intuitively, an independence relation between actions should allow us to explore fewer transitions: if two sequences of actions lead to the same state, it suffices to apply one of them. The goal of this paper is to explore the advantages and limitations of several variants of this approach.

Our main contribution is a new algorithm for state-space search, which we call *edge-lean search*. Our algorithm selects a total order on actions, extends it to paths (i.e., sequences of actions) in a natural way, and only explores paths that cannot be made smaller with respect to this order by permuting two adjacent independent actions. The proof that combining this simple principle with depth-first search ensures visiting all states turns out to be quite non-trivial (see Section 3).

We also investigate a trace theory-inspired approach, which only considers paths corresponding to sequences in trace normal form (TNF) (see Section 4). The sequences in trace normal form are minimal with respect to the ordering described in the previous paragraph, but the converse is not true. Hence, our TNF-based algorithm provides a more powerful reduction than the edge-lean search algorithm. However, the added efficiency comes at a price: we demonstrate that if the state system may contain cycles, the TNF-based algorithm that uses depth-first search may fail to visit all states. On the other hand, it does provide complete coverage if the underlying state system does not contain cycles. As many state systems that occur in practice are naturally acyclic, there are situations where the TNF-based algorithm should be preferred over the edge-lean algorithm.

We also consider a variant of the TNF-based algorithm which explores the states in breadth-first search order, and discuss connections between the algorithms proposed in this paper and the sleep set algorithm [4,5].

In more detail, the main contributions of this paper are as follows:

- Presenting a simple depth-first search-based algorithm `EdgeLeanDfs` that visits all states, but does not explore all transitions. It has no space overhead and negligible time overhead. Furthermore, it provides significant time savings *when exploring transitions is expensive* (which is not the case in SPIN), and significant space savings in every context. Therefore, it compares favorably with the corresponding version of the sleep sets of Godefroid [4], which requires saving a set of transitions (the "sleep set") together with every reached state.
- Presenting an alternative TNF-based algorithm `TNF_Dfs` for the same problem that explores even fewer edges than `EdgeLeanDfs`, but is only guaranteed to explore all states if the underlying state system is acyclic. This algorithm is shown to coincide, under some order restrictions, with the original sleep set algorithm (see [5]), but requires less overhead per state.
- Providing an example that shows that when the state system contains cycles, `TNF_Dfs` (and hence the original sleep set algorithm) may fail to cover all states.

– Showing that a modification of the TNF-based algorithm that explores states in breadth-first order is guaranteed to explore all states.
– Providing an efficient representation and an update algorithm for a data structure used to check that traces (equivalence classes of sequences under permutation) are in normal form (i.e., minimal in their equivalence class under trace equivalence). This data structure is used by our TNF-based algorithms, but has other applications, such as generation and checking of normal form traces.

In Section 7, we provide the results of several experiments that compare our algorithms with classic depth-first search used in SPIN [8]. Our experiments show that for a number of natural problems, our methods provide a dramatic reduction in the number of transitions explored and the stack size.

Related Work

The idea of speeding up state-space search by using an independence relation between actions is well-known in the model-checking community. In particular, this approach has been explored by the family of partial order reductions [11, 4, 13]. As opposed to our setting, in general these methods, known as *ample* sets, *persistent* sets, or *stubborn* sets, respectively, do not necessarily visit all the states. However, these methods guarantee to generate a reduced state space that preserves the property that one would like to check. Our algorithms are most closely related to the sleep set approach of [4], in particular, the non-state-splitting sleep set algorithms. In fact, we show that our TNF-based algorithm generates exactly the same reduced graph as the very first version of the sleep set algorithm proposed in [5]. On the other hand, the more general algorithm of Section 3 is substantially different from all the algorithms described in the existing literature. Moreover, it has the advantage of achieving full state coverage with no memory overhead.

Our work also has obvious connections to planning. Indeed, in many planning domains there is a natural independence relation on actions, and hence our algorithms are directly applicable for forward search. However, in planning one usually assumes considerably more problem structure, such as pre- and post-condition for actions, the ability to decompose the goal, etc. This additional information is usually used by the planning algorithm in an essential way. Indeed, it has been argued that simple forward seach that explores all states in computationally infeasible in many planning scenarios due to combinatorial state explosion. Hence, it would be interesting to see if our algorithms could be combined with classic planning techniques so as to be applicable in real-life planning scenarios. Our model also bears certain similarity to *nonlinear planning*. However, in nonlinear planning the search take place in the plan space as opposed to the action space, and the independence relation is on partial plans, not on actions.

2 Preliminaries

A *system* (or *transition system*) is a tuple $\mathcal{A} = \langle S, s_0, \Sigma, T, I \rangle$ where

– S is a finite set of *states*.
– $s_0 \in S$ is an *initial state*.
– Σ is a finite *alphabet of actions*.

– $T \subseteq S \times \Sigma \times S$ is a *labeled transition relation*. We write $s \xrightarrow{a} s'$ when $(s, a, s') \in T$.
– $I \subseteq \Sigma \times \Sigma$ is a symmetric and irreflexive relation on actions, called the *independence relation*.

We say that the system has the *diamond* property if for every state s and any pair of independent actions $(a, b) \in I$ it is the case that if $s \xrightarrow{a} q \xrightarrow{b} r$ then there exists a state $q' \in S$ such that $s \xrightarrow{b} q' \xrightarrow{a} r$. All state systems considered in this paper are assumed to have the diamond property. However, we do not require the *forward diamond* property:

If $s \xrightarrow{a} q$ and $s \xrightarrow{b} q'$ then there exists a state $r \in S$ such that $s \xrightarrow{a} q \xrightarrow{b} r$.

We will sometimes identify a state system with a directed graph whose vertices correspond to states, and whose edges correspond to transitions.

An action a is *enabled* from a state $s \in S$ if there exists some state $s' \in S$ such that $s \xrightarrow{a} s'$. We say that a path $\rho = s_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} s_n$ is *loop-free* or *simple* if $s_i \neq s_j$ for all $i \neq j$. A *labeling* $\ell(\rho)$ of a path $\rho = s_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} s_n$ is given by $\ell(\rho) = a_1 \cdots a_n$. We extend the arrow notation from transitions to paths, denoting the above path by $s_0 \xrightarrow{a_1 \ldots a_n} s_n$. We will sometimes identify a path with its labeling.

Let σ, $\rho \in \Sigma^*$. We write $\sigma \overset{1}{\equiv} \rho$ if and only if there exist strings $u, v \in \Sigma^*$ and letters $(a, b) \in I$ such that $\sigma = uabv$, $\rho = ubav$. That is, $\sigma \overset{1}{\equiv} \rho$ if ρ is obtained from σ (or vice versa) by transposing adjacent independent letters. Let $\equiv$ be the transitive closure of $\overset{1}{\equiv}$. It is not hard to see that $\equiv$ is an equivalence relation. It is often called *trace equivalence* [9].

For example, for $\Sigma = \{a, b\}$ and $I = \{(a, b), (b, a)\}$ we have $abbab \overset{1}{\equiv} ababb$ and $abbab \equiv bbbaa$. Notice that if the system has the diamond property and $u \equiv v$, then $s \xrightarrow{u} r$ if and only if $s \xrightarrow{v} r$.

Let $\ll$ be a total order on the alphabet Σ. We call it the *alphabetic order*. We can extend $\ll$ to a lexicographic order on words in a standard way, i.e., by setting $v \ll vu$ and $vau \ll vbw$ for any v, u, $w \in \Sigma^*$ and any a, $b \in \Sigma$ such that $a \ll b$.

Definition 1 Let $w \in \Sigma^*$. Let $\tilde{w}$ denote the least word under the relation $\ll$ that is equivalent to w. If $w = \tilde{w}$, we say that w is in *trace normal form* (TNF) [10].

Most of the search algorithms to be presented are based on depth-first search (DFS), which provides some advantages over breadth-first search (e.g., linear time detection of loops using Tarjan's algorithm [12]). It uses a hash table to check whether a state has been previously visited.

```
proc Dfs(q);
     local variable q';
     hash(q);
     forall q --a--> q' do
            if q' is not hashed then Dfs(q');
end Dfs;
```

3 An Edge Lean Algorithm for Complete State Coverage

In this section, we show how to reduce the number of explored edges by making use of the diamond property. Clearly, any state that can be reached from the initial state

by a path labeled with some word w can also be reached by a path labeled with $\tilde{w}$. Therefore, it is tempting to limit our attention to paths labeled with words in TNF, as such paths do explore *all* reachable states. However, one has to use caution when applying this approach within the depth-first search framework. The main reason for this is that all paths explored during depth-first search are necessarily acyclic. Hence, this approach would only consider paths that are *both* acyclic *and* labeled with words in TNF. On its own, neither of these restrictions prevents us from reaching all states. However, it turns out that when the input state system has cycles, searching only paths that are labeled with words in TNF, leaves some states unexplored; we provide an example in Section 4.1. Therefore, for state systems that may have cycles, we have to settle for a less ambitious reduction. In what follows, we define a smaller relation on strings in Σ^*, and prove that it suffices to explore paths whose labeling is minimum with respect to this relation.

Definition 2 Set $ubav \rightarrowtail_1 uabv$ if and only if $a \, I \, b$ and $a \ll b$, and let $\rightarrowtail$ be the transitive closure of $\rightarrowtail_1$. We say that a word $w \in \Sigma^*$ is *irreducible* if there exists no $w' \neq w$ such that $w \rightarrowtail w'$.

Intuitively, a word is irreducible if it cannot be transformed into a smaller word with respect to $\rightarrowtail$ by a local fix (a single permutation of adjacent independent letters). We call a path ρ irreducible if its labeling $\ell(\rho)$ is an irreducible word. Observe that a prefix of an irreducible path is also irreducible. Note that if w is in TNF, then it is irreducible. However, the converse does not necessarily hold. Indeed, consider $a \ll b \ll c$, $a \, I \, b, b \, I \, c$ and $\neg(a \, I \, c)$. Then $x = cab$ is irreducible, but $\tilde{x} = bca \neq x$.

Our algorithm `EdgeLeanDfs` is based on depth-first search. However, unlike the classic depth-first search, it only explores paths whose labelings are irreducible. For this, it suffices to remember the last letter x seen along the current path, and not to extend this path with letter y whenever $x \, I \, y$ and $y \ll x$.

```
EdgeLeanDfs(s0,ϵ);

proc EdgeLeanDfs(q,prev);
     local variable q';
     hash(q);
     forall q -a-> q' such that prev= ϵ or ¬(aIprev) or prev ≪ a do
            begin
                  if q' not hashed then EdgeLeanDfs(q', a);
end EdgeLeanDfs;
```

Let $first_{el}(s)$ be the first path by which `EdgeLeanDfs(`s_0`, ϵ)` reaches the state s; if `EdgeLeanDfs(`s_0`, ϵ)` does not reach s, set $first_{el}(s)=\perp$.

Theorem 1 *For any state system $\mathcal{A}$ and any $s \in \mathcal{A}$, we have $first_{el}(s) \neq \perp$. Hence,* `EdgeLeanDfs(`s_0`, ϵ)` *explores all states of $\mathcal{A}$ that are reachable from s_0.*

Proof To prove Theorem 1, we fix a state s, and show that `EdgeLeanDfs(`s_0`, ϵ)` reaches this state. To do so, we start with an arbitrary simple irreducible path in the state graph that reaches s (we show that such a path always exists) and repeatedly apply to it a transformation T, defined below. This transformation produces another simple irreducible path that also leads to s. We show that for any ρ for which $T(\rho)$ is defined, an application of T results in a path that is smaller than ρ with respect to a certain

well-founded ordering, defined later. Therefore, after a finite number of iterations, we obtain a simple irreducible path ρ such that $T(\rho)$ is not defined. We then prove that any such ρ is a path taken by `EdgeLeanDfs(`s_0`, `ϵ`)`, i.e., s is reached by our algorithm. The details follow.

For any simple path ρ and any state t on this path, we denote by ρ_t the prefix of ρ that reaches t. In particular, ρ_s is a simple path that reaches s. We will now show that we can choose ρ_s so that it is irreducible.

Lemma 1 *For any state $s \in \mathcal{A}$ that is reachable from s_0, there exists a path ρ_s that is simple and irreducible.*

Proof We start with an arbitrary path ρ_s that leads from s_0 to s and iteratively (1) delete loops from the current path; and (2) replace the current path with an an equivalent irreducible path. Each application of (1) strictly decreases the length of the path, while (2) does not change its length. By diamond property, (2) results in a path that also leads to s. We obtain a simple irreducible path that leads to s after a finite number of iterations. ∎

Given a simple path ρ that reaches s, all states on ρ can be classified into three categories with respect to ρ. We say that a state t is *red* if $\mathit{first}_{el}(t) = \rho_t$, *blue* if $\mathit{first}_{el}(t) \neq \perp$, but $\mathit{first}_{el}(t) \neq \rho_t$, and *white* if $\mathit{first}_{el}(t) = \perp$. This classification depends on the path ρ: a state can be red with respect to one path but blue with respect to a different path. It turns out that for a simple irreducible path, not all sequences of state colors are possible.

Lemma 2 *Suppose that ρ_s is loop-free and irreducible. Then if t is the last red state along ρ_s, all states that precede t on ρ_s are also red. Moreover, either t is the last state on ρ_s, i.e., $t = s$, or the state t' that follows t on ρ_s is blue.*

Proof The first statement of the lemma follows from the definition of a red state and from the use of depth-first search. To prove the second statement, assume for the sake of contradiction that t' is white (t' cannot be red as t is the last red state on ρ_s). The path $\rho_{t'}$ is a prefix of ρ_s, so it is simple and irreducible. Hence, `EdgeLeanDfs(`s_0`, `ϵ`)` must explore the transition that leads from t to t'. Therefore, t' cannot be white. ∎

We define a transformation T that can be applied to any simple irreducible path $\rho = \rho_s$ that contains a blue state; its output is another simple irreducible path that reaches the same state s. Recall that $\ell(\pi)$ denotes the labeling of a path π. The transformation T consists of the following steps (w and v appear only for a later reference in the proof):

(1) Let ρ_t be the shortest prefix of ρ such that t is blue. Decompose ρ as $\rho = \rho_t\,\sigma$. Modify ρ by replacing ρ_t with $\mathit{first}_{el}(t)$, i.e., set $\rho{=}\mathit{first}_{el}(t)\sigma$. Set $w = y = \ell(\mathit{first}_{el}(t))$, $v = z = \ell(\sigma)$ and $x = yz = \ell(\rho)$.
(2) Eliminate all loops from ρ. Update x, y, and z by deleting the substrings that correspond to these loops.
(3) Replace ρ with an equivalent irreducible path as follows:
  (3a) Replace z with an equivalent irreducible word.
  (3b) Let a be the last letter of y, and let b be the first letter of z. If $a \gg b$ and $a\,I\,b$, move a from y to z and push it as far to the right as possible within z.
  (3c) Repeat Step (3b) until the last letter a of y cannot be moved to z, i.e., $a \ll b$ or a and b are not independent.

(3d) Set $x = yz$, and let ρ be a path reaching s with $\ell(\rho) = x$.
(4) Repeat (2) and (3) until ρ is simple and irreducible.

By the argument in the proof of Lemma 1, we only need to repeat steps (2) and (3) a finite number of times, so the computation of T terminates. Observe that if s is red with respect to ρ_s, then $T(\rho_s)$ is not defined. On the other hand, consider a simple irreducible path ρ_s such that s is not red with respect to ρ_s. By Lemma 2, we can apply T to ρ_s. The output of $T(\rho_s)$ is loop-free and irreducible, so if s is not red with respect to $T(\rho_s)$, we can apply T to $T(\rho_s)$. We will now show that after a finite number of iterations n, we obtain a path $T^n(\rho_s)$, which consists of red states only.

We will first define a new ordering $<_\sharp$ on words, and state a useful result about the properties of this ordering (Proposition 1), which follows immediately from Higman's lemma [7].

Definition 3 For a word $v \in \Sigma^*$, let $\sharp_a(v)$ be the number of occurrences of the letter a in v. We write $v <_\sharp w$ if there exists a letter a such that for all $b \ll a$, $\sharp_b(v) \leq \sharp_b(w)$ and $\sharp_a(v) < \sharp_a(w)$.

Proposition 1 *The relation $<_\sharp$ is a well-founded (partial) order, i.e., there does not exist an infinite sequence $u_1, u_2, \ldots \in \Sigma^*$ such that $u_1 >_\sharp u_2 >_\sharp \ldots$.*

Consider a simple irreducible path $\rho = \rho_s$. Suppose that both ρ and $T(\rho)$ contain blue states.

Lemma 3 *Let $\rho = \rho_t\sigma$, where t is the first blue state on ρ, and let $T(\rho) = \rho'_{t'}\sigma'$, where t' is the first blue state on $T(\rho)$. Let $v = \ell(\sigma)$, $v' = \ell(\sigma')$. Then $v >_\sharp v'$.*

Before we prove Lemma 3, let us show that it implies Theorem 1. Indeed, by Proposition 1, there does not exist an infinite decreasing sequence of words with respect to $<_\sharp$. The strings v, v' satisfy $v' <_\sharp v$, and are well-defined as long as both ρ and $T(\rho)$ contain blue states. Hence, for some finite value of n, $T^n(\rho)$ contains no blue states, it is simple and irreducible. Therefore, by Lemma 2 we obtain a path of our algorithm that reaches s. We now prove Lemma 3.

Proof We use the notation introduced in the description of T: we have $w = \ell(\mathit{first}_{el}(t))$, $v = \ell(\sigma)$, and $x = wv$ is the labeling of ρ after ρ_t was replaced by $\mathit{first}_{el}(t)$.

In the rest of the proof, we use the word "letter" to refer both to an element of Σ and an occurrence of this element in a word. The specific meaning will be clear from the context. In particular, in the coloring described in the next paragraph, we will assign colors to occurrences of the elements of Σ rather than the elements itself, whereas when we write $a \ll b$, we refer to the respective elements of Σ.

We will now color all the letters in the word wv so that all letters in w are yellow and all letters in v are green, and investigate what happens to this coloring during an application of transformation T. Note that by construction, at any point in time all letters in y will be yellow, and therefore all letters pushed into z during Step (3) will be yellow. We construct a directed acyclic graph (DAG) whose set of nodes includes all yellow letters in z as well as some of the green letters. Namely, if a yellow letter a gets pushed into z when the first letter of z is b, there is an edge from this occurrence of a to this occurrence of b. Also, if a (yellow or green) letter a that is currently the first letter of z gets transposed with its right-hand side neighbor b (by (3a)), there is an edge from this occurrence of a to this occurrence of b. Observe that in both cases if

there is an edge from an occurrence of a to an occurrence of b, then we have $b \ll a$, so our graph contains no directed cycles. We do not delete a node from this graph even if the respective occurrence is deleted from x by (2).

Lemma 4 *Each yellow letter pushed into z has an outgoing edge. Moreover, if a letter has incoming edges, but no outgoing edges, either it has been eliminated from x, or it is the first letter of z after the execution of T is completed.*

Proof Each yellow letter acquires an outgoing edge as it is moved into z. Now, consider a letter that has incoming edges. It acquired them either when it was the first letter of z and yellow letters were pushed past it, or when it was transposed with its left-hand side neighbor and became the first letter of z. In both cases, it was the first letter of z at some point in time. If it remains in that position till the end of the execution of T, we are done. Now, suppose that it stopped being the first letter of z. Then either it was deleted during the loop elimination phase, in which case we are done, or it was transposed with its right-hand side neighbor, in which case it acquired an outgoing edge. ∎[Lemma 4]

Let G be the set of nodes of our DAG that have incoming edges, but no outgoing edges. By Lemma 4 none of the letters in G is yellow, so all of them are green. Moreover, each letter in G either has been eliminated from x or is the first letter of z after the end of the execution of T.

Consider the string $x = yz$ obtained after the end of the execution of T. This string corresponds to $\rho' = T(\rho)$. Recall that w corresponds to $\mathit{first}_{el}(t)$, which consists of red states only, and y is a prefix of w. Hence, the prefix of ρ' that corresponds to y reaches a red state. Therefore, to reach a blue state along ρ', we need to progress over at least one letter of z, or, equivalently, v' is a strict suffix of z. That is, v' does not include the first letter of z. Using Lemma 4, we conclude that v' does not contain any of the letters in G.

Let a be the minimal letter of G with respect to $\ll$. It is contained in v, but not in v'. On the other hand, each letter c that is contained in v' but not in v, is a yellow letter that appears in the DAG. That is, there is a path in the DAG leading from c to some $b \in G$. By construction of the DAG, the existence of a path from c to b implies $c \gg b$, and hence $c \gg a$. Hence, for any b in v' such that $b \ll a$ or $b = a$ it holds that b is green, and therefore $\sharp_b(v') \leq \sharp_b(v)$. Also, we have argued that $\sharp_a(v') < \sharp_a(v)$. We conclude that $v' <_\sharp v$. ∎[Lemma 3,Theorem 1]

We now give an example that illustrates the performance of our algorithm. Consider two processes p and p' with a counter from 1 to n on each process. These counters can be incremented through actions a and a', respectively, and decremented through actions b and b', respectively. Clearly, the independence relation between this actions is the symmetric closure of aIa', aIb', $a'Ib$, bIb'. Let $a \ll b \ll a' \ll b'$. The left part of Figure 1 shows in solid edges the paths explored by regular depth-dirst search (`Dfs`), and in dotted edges the transitions which lead to a state already explored. On the right, we indicate in the same way the path followed by `EdgeLeanDfs`, though we have deleted the transitions not considered by `EdgeLeanDfs`.

It is easy to see that `Dfs` considers almost twice as many transitions as `EdgeLeanDfs` (i.e., $4n(n-1)$ versus $(2n+2)(n-1)$). Moreover, the stack size in `DFS` is $n^2 - 1$ (path labeled by $(a^n a' b^n a')^{(n-1)/2}$), while for `EdgeLeanDfs` it is only $2n$. Generalizing this example from 2 to n processes, we obtain an example in which `EdgeLeanDfs` has an exponentially smaller maximum stack size than `Dfs`.

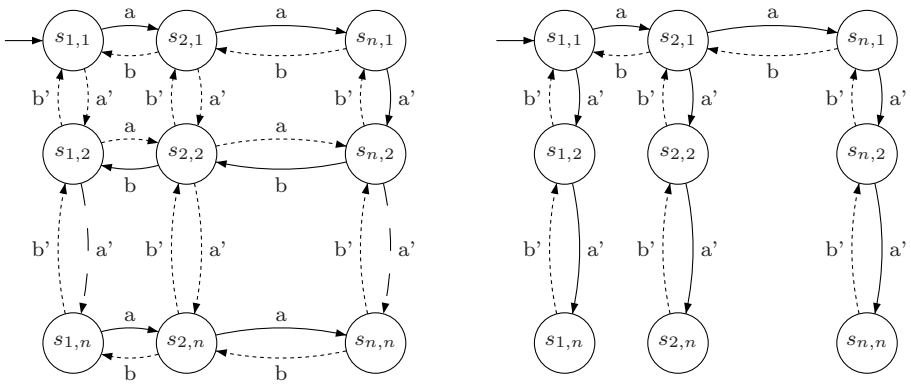

Fig. 1 Paths explored by `Dfs` (left) and by `EdgeLeanDfs` (right).

4 TNF-based Algorithms

In the beginning of Section 3, we have demonstrated that there are irreducible words that are not in trace normal form. Thus, given two paths from the same trace equivalence class that are both labeled with irreducible words, `EdgeLeanDfs` will explore both of them, even though they are guaranteed to lead to the same state. Therefore, in some sense, `EdgeLeanDfs` behaves suboptimally.

In this section, we focus on algorithms that only explore paths labeled with words in TNF. First, we demonstrate that under depth-first search order, the TNF-based algorithm is guaranteed to explore all states as long as the state space is cycle-free. We then provide an example showing that the latter condition is necessary. Finally, we show that under breadth-first search order, the TNF-based algorithm explores all states even if the state space may contain cycles.

4.1 TNF-based DFS Algorithm for Acyclic State Spaces

We will now describe an algorithm `TNF_Dfs`, which is based on depth-first search and only explores paths labeled with words in trace normal form. This algorithm often provides a significant reduction in the size of stack needed, compared to both regular depth-first search and `EdgeLeanDfs`. The algorithm `TNF_Dfs` uses *summaries* (also called last appearance records in other contexts), which are data structures that represent the order in which the last appearance of each letter occurred in the string. Using summaries provides a significant computational advantage, as they are much cheaper to store and analyze than the strings themselves.

Definition 4 Given a string σ, let $\alpha(\sigma)$ denote the set of letters occurring in σ. A *summary* of σ is the total order $\prec_\sigma$ on the letters from $\alpha(\sigma)$ such that $a \prec_\sigma b$ if and only if the last occurrence of a in σ precedes the last occurrence of b in σ. That is, $a \prec_\sigma b$ if σ can be represented as $vaubw$, where $v \in \Sigma^*$, $u \in (\Sigma \setminus \{a\})^*$, $w \in (\Sigma \setminus \{a, b\})^*$.

We will also represent a summary $\prec_\sigma$ as a word over Σ in which each letter occurs at most once, and $a \in \Sigma$ appears before $b \in \Sigma$ if and only $a \prec_\sigma b$. For instance, in this notation, the summary of the word $w = bcbabza$ is $cbza$. Clearly, the length of a summary is always at most $|\Sigma|$.

We will now formulate a criterion that allows us to check whether adding a letter to a string in TNF results in a string that is also in TNF.

Lemma 5 *Let $\sigma \in \Sigma^*$ be in TNF and $a \in \Sigma$. Then σa is* not *in TNF if and only if $\sigma = vu$ for some v, u such that (i) $vau \equiv vua$ and (ii) $vau \ll vu$.*

Proof If conditions (i) and (ii) hold, then clearly vua is not in TNF, since it is not lexicographically minimal among the sequences that are equivalent to it.

Conversely, let ρ be the lexicographically minimal string such that $\rho \equiv \sigma a$. Denote by $fst(v)$ the first letter of a nonempty string v. Let v be the maximal common prefix of ρ and σ (and thus also of ρ and σa). Let u, w be the respective suffixes of σ and ρ, i.e., $\sigma = vu$ and $\rho = vw$. We will now prove that the decomposition $\sigma = vu$ satisfies conditions (i) and (ii). Consider the following cases:

1. w starts with a.
   (a) u does not contain a. Then $au \equiv ua$, satisfying (i) and (ii).
   (b) u contains a. Write $u = u_1 a u_2$, where u_1 contains no occurrences of a. Then $u = u_1 a u_2 \equiv a u_1 u_2$. Since $\rho = vw \ll vua$, we have $a = fst(w) \ll fst(u_1) = fst(u)$. Thus, $vau_1u_2 \ll vu_1au_2 = vu$, a contradiction to the fact that σ is in TNF.
2. w does not start with a.
   Write $w = w_1 a w_2$, where w_2 contains no occurrences of a. Then, $w = w_1 a w_2 \equiv w_1 w_2 a \equiv ua$ and thus $w_1 w_2 \equiv u$. Since $vw \ll vu$, we have $fst(w_1) = fst(w) \ll fst(u)$. Thus, $vw_1w_2 \ll vu = \sigma$ and $vw_1w_2 \equiv vu$. This contradicts the fact that σ is in TNF. ∎

Intuitively, Lemma 5 means that σa is not in TNF iff we can move a backwards past an appropriate suffix of σ to obtain a string that is lexicographically smaller than σ. The following lemma shows that we can apply this criterion using a summary of σ rather than σ itself.

Lemma 6 *Consider a string $\sigma \in \Sigma^*$ in TNF and a letter $a \in \Sigma$. The string σa is not in TNF if and only if there is a letter $b \in \alpha(\sigma)$ such that $a \ll b$ and for each c such that $b \preceq_\sigma c$ we have aIc.*

Proof Suppose that σ is in TNF and σa is not. Let u be the shortest suffix of σ that satisfies the conditions of Lemma 5, i.e., $\sigma = vu$ and $vau \equiv vua$. Let b be the first letter of u. Then $a \ll b$. Furthermore, consider any c such that $b \preceq_\sigma c$. Clearly, $c \in \alpha(u)$. As aIc' for each $c' \in \alpha(u)$, we have aIc.

Conversely, let $b \in \alpha(\sigma)$ be a letter satisfying the conditions of the lemma. Let u be the shortest suffix of σ that begins with b. It follows that all letters $c \in \alpha(u)$ satisfy $b \preceq_\sigma c$, and aIc. This means that conditions (i) and (ii) of Lemma 5 hold. ∎

For instance, consider the string $w = bcbabza$ with summary $cbza$. Suppose that b commutes with a, but not with z. Then the string wb is in TNF if and only if $a \ll b$.

Our algorithm `TNF_Dfs` is desribed below. It performs a reduced depth-first search (DFS) by only considering strings in TNF. It considers all transitions enabled at the current state. For each of them, it checks if adding this transition to the labeling of the current path will result in a strng in TNF, using function `normal`; the transition is only explored if this is indeed the case. This check is performed using summaries, as suggested by Lemma 6. The summary is stored in a global array `summary[1..n]`, where

$n = |\Sigma|$. The variable `size` stores the number of different letters in the current string σ. We update the summary as we progress with the DFS, and recover the previous value when backtracking. That is, the value of the summary is calculated on the fly through the use of functions `update_sumr` and `recover_sumr`, defined later. This means that there is no need to save the value of the summary with the state information.

```
size:=0;
TNF_Dfs(s_0)

proc TNF_Dfs(q)
     local variables q', i;
     hash(q);
     forall q -a-> q' in increasing order do
            if normal(a) and q' not hashed then
               i:=ord(a);
               update_sumr(i,a);
               TNF_Dfs(q');
               recover_sumr(i,a);
end TNF_Dfs;
```

We will now describe the procedures used in the pseudocode for `TNF_Dfs`.

The function `ord` is used to find the position of a letter a in the summary.

```
func ord(a);
     for i:=size backto 1 do
         if summary[i]=a then return(i);
     return(0);
end ord;
```

The function `normal` checks whether the current string augmented with a given letter is in TNF. It makes use of the summary as described in Lemma 6.

```
func normal(a);
     for j:=size backto 1 do
         b:=summary[j];
         if ¬ (a I b) then return(true);
         if a≪b then return(false);
     return(true);
end normal;
```

The summary is updated using the procedure `update_sumr`. This procedure is given the last transition a that was executed, and its old location i (0 if it was not introduced yet) in the summary. It removes a from the ith position, shifts all elements in positions $i+1, \ldots,$ `size` one position to the left, and puts a at the end of the summary. If a did not occur in the summary, then there is no need to shift other letters, but in this case the size of the summary increases.

```
proc update_sumr(i, a);
     if i=0 then
        size:=size+1
     else
```

```
        for j:=i+1 to size do
            summary[j-1]:=summary[j];
     summary[size]:=a
end update_sumr;
```

Recovering the summary upon backtracking is done with the help of the procedure `recover_sumr`. It reverses the effect of `update_sumr` by shifting the array elements indexed i (the original position of a) and higher to the right, and putting a in the ith place. If i is zero, then there is no need for shifting, but the size of the summary needs to be decremented.

```
proc recover_sumr(i, a);
     if i=0 then
        summary[size]:=blank;
        size:=size-1
     else
        for j:=size-1 downto i do
            summary[j+1]:=summary[j]
        summary[i]:=a;
end recover_sumr;
```

Theorem 2 *Given an acyclic state space $\mathcal{A}$, the algorithm* `TNF_Dfs`(s_0) *visits all states of $\mathcal{A}$ that are reachable from s_0.*

Proof Fix a state s and consider the set of all paths that reach s. Among all such paths, pick one whose labeling is minimal with respect to $\ll$. Denote this path by $min_{tnf}(s)$. Clearly, $\ell(min_{tnf}(s))$ is in TNF. We will now show that s is reached by $min_{tnf}(s)$.

Indeed, suppose that there exists a state s that is not reached by $min_{tnf}(s)$. Among all such states, pick the one with the lexicographically minimal $\ell(min_{tnf}(s))$. Let t be the state that precedes s on $min_{tnf}(s)$, and suppose that $\ell(min_{tnf}(s)) = ua$. We have $u \ll \ell(min_{tnf}(s))$, and hence $u = \ell(min_{tnf}(t))$. Consider the moment when our algorithm has reached the state t and is about to decide whether to explore a. Since ua is in TNF and acyclic, the action a will be taken, and hence the state s will be reached by $min_{tnf}(s)$. Furthermore, since $min_{tnf}(s)$ is lexicographically minimal among all paths that reach s, no other path that reaches s has been considered before. ∎

We will now analyze the running time of our algorithm. Each call to `update_sumr`, `ord`, `recover_sumr`, and `normal` takes time $O(|\Sigma|)$. Moreover, the summary data structure can be implemented as a list, in which case `ord` and `normal` need $O(|\Sigma|)$ reads, while `update_sumr` and `recover_sumr` need O(1) writes to memory only. Let S be the total number of states. As the total number of transitions is bounded by $S|\Sigma|$, it follows that the running time of our algorithm is $O(S|\Sigma|^2)$. In Section 6, we will further reduce the running time of our algorithm in the important special case where actions belong to communicating processes. Concerning memory, compared with the usual DFS algorithm, one needs to remember $\log_2 |\Sigma|$ additional bits per letter in the stack (the index of this letter in the previous summary) to recover the summary, plus the current summary itself ($|\Sigma| \log_2 |\Sigma|$ bits, which is negligible with respect to the size of the stack). This compares favorably to the sleep set method (see Section 5), which needs to store the entire sleep set, i.e., up to $|\Sigma| \log_2 |\Sigma|$ bits, for each letter in the stack.

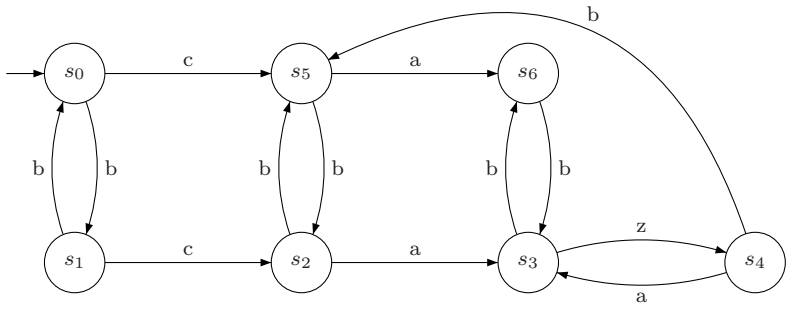

Fig. 2 A state space for which `TNF_Dfs` does not explore every state.

4.2 State Space Not Fully Explored by `TNF_Dfs`

In the previous section, we have argued that for acyclic state spaces, `TNF_Dfs` is guaranteed to explore all states. However, for general state spaces this is no longer the case.

Figure 2 provides an example of a state space that satisfies the diamond property, but is not fully explored by `TNF_Dfs`. The nodes, except s_6, are numbered in the order in which they are discovered. The node s_6 is not discovered. The alphabet is $\{a, b, c, z\}$, with the ordering $a \ll b \ll c \ll z$. The independence relation is the symmetric closure of $b\,I\,a$, $b\,I\,c$. Consequently, z depends on every other letter a, b, c, and a, c are dependent. The state s_6 can only be visited through s_3 and s_5, with $min_{tnf}(s_3) = bca$ and $min_{tnf}(s_5) = bcazb$. Neither of the strings $bcab$ ($\equiv bbca$) and $bcazba$ ($\equiv bcaaab$) is in TNF (but note that $bcab$ is irreducible, as required by `EdgeLeanDfs`), hence s_6 is not visited. On the other hand, s_6 is visited by `EdgeLeanDfs`.

4.3 TNF-Based Breadth-First Search

We will now prove that, when using breadth-first search (BFS), we can ignore paths labeled by words not in TNF and still reach all states, even on graphs with cycles. Hence, the results of Section 5, which relate the sleep set method to TNF-based search, imply that using BFS with sleep sets does not require a loop proviso. This is in contrast with the setting where the sleep set method is used in conjunction with DFS, in which case a loop proviso appears necessary. We will start by describing the classic breadth-first search. The algorithm below is initialized with a queue containing the initial state only.

```
hash(s0); Bfs(s0);

proc Bfs(queue);
     local variables q, q', queue';
     queue'=ε;
     forall q in queue do
        forall q -a-> q' do
```

```
          if q' not hashed then hash(q');
                                append(queue',q');
     Bfs(queue');
end Bfs;
```

We now modify this search to consider paths in TNF only. For this version of the algorithm, we need to keep track of the summaries. Therefore, the elements of the queue are pairs of the form (state, summary). Consequently, unlike in the DFS version of the algorithm, we do not have a global variable `summary`. The implementation below uses functions `normal(a, summary)`, `ord(a)` and `update_summary(i, a, summary)`, which are similar to procedures `normal(a)` and `update_summary(i, a)` defined in Section 4.1, but take into account the fact that `summary` is not a global variable. In particular, this means that `update_sumr` now produces an output `summary'`, which is an updated version of its input `summary`, while `summary` itself remains intact (and hence there is no need to recover it).

```
hash(s_0); TNF_Bfs(s_0);

proc TNF_Bfs(queue);
     local variables q, q', summary', queue';
     queue'=ϵ;
     forall pair (q, summary) in queue do
            forall q -a-> q' in ≪ order do
                 if normal(a, summary) and q' not hashed then
                    hash(q');
                    i:=ord(a, summary);
                    summary' = update_sumr(i, a, summary);
                    append(queue', (q', summary'));
     TNF_Bfs(queue');
end TNF_Bfs;
```

We will now prove that this algorithm explores all states.

Theorem 3 *For any state system $\mathcal{A}$, the algorithm* `TNF_Bfs`*(s_0) explores all states that are reachable from s_0.*

Proof For each node $s \in \mathcal{A}$ reached by `TNF_Bfs` from s_0, let $first_{bfs}(s)$ be the path from s_0 to s according to which s is discovered. We need to prove that $first_{bfs}(s)$ is defined for each $s \in \mathcal{A}$.

For every $s \in \mathcal{A}$, let $Min(s)$ be the set of paths of minimum length from s_0 to s in $\mathcal{A}$. That is, every path in $Min(s)$ has the same length, and there are no shorter paths reaching s. We denote by $min_{bfs}(s)$ the path of $Min(s)$ labeled by the lexicographically smallest word among the labelings of paths in $Min(s)$. Since $\mathcal{A}$ has the diamond property, the string $min_{bfs}(s)$ is lexicographically minimal among trace equivalent words, and hence necessarily in TNF.

Lemma 7 *Let $min_{bfs}(s) = \sigma a$, and let t be the state that is reached from s_0 via σ, i.e., $s_0 \xrightarrow{\sigma} t$. Then $min_{bfs}(t) = \sigma$.*

Proof First, it follows from the definition of trace normal form that any prefix of a word in TNF must also be in TNF, hence σ is in TNF. Suppose for the sake of contradiction

that $min_{bfs}(t) = \rho$, where $\rho \neq \sigma$. Then we have $\rho \ll \sigma$ and thus $\rho a \ll \sigma a$. Then $s_0 \xrightarrow{\rho a} s$. Let μ be the lexicographically minimal word that is equivalent to ρa; clearly, μ is in TNF. Then either $\mu = \rho a$ or $\mu \ll \rho a$. It follows that $\mu \ll \sigma a$, a contradiction. [Lemma 7]

Since we explore the actions available from each node in alphabetic order, it follows by induction on the order of discovering nodes by `TNF_Bfs` that if s and t are on the queue at the same time (that is, they are at the same distance from the initial node s_0), and s appears before t in the queue, then $first_{bfs}(s) \ll first_{bfs}(t)$.

Now we will prove the following claim, which implies the correctness of the algorithm: each node s in $\mathcal{A}$ is reached during the search, and is paired up with the summary that corresponds to the string $min_{bfs}(s) = first_{bfs}(s)$. The proof is by contradiction. Let S be the set of states for which this claim does not hold. Let $n = \min\{|min_{bfs}(s')| \mid s' \in S\}$. Let S_n be the states s' of S with $|min_{bfs}(s')| = n$. Among all states in S_n, pick the one with path $min_{bfs}(s)$ labeled by the lexicographically smallest word, and denote it by s. That is, for all $s' \in S$, either $|min_{bfs}(s)| < |min_{bfs}(s')|$, or $|min_{bfs}(s)| = |min_{bfs}(s')|$ and $min_{bfs}(s)$ is lexicographically before $min_{bfs}(s')$. Suppose that $min_{bfs}(s) = \sigma a$, and let t be the state that precedes s on $min_{bfs}(s)$. Then by Lemma 7 we have $min_{bfs}(t) = \sigma$. Since $|\sigma| < |\sigma a|$, the state t is discovered and put into the queue paired up with a summary that corresponds to the word $first_{bfs}(t) = min_{bfs}(t) = \sigma$. Now, when this pair is removed from the queue, checking a against the summary that is paired up with it results in the conclusion that σa is in TNF. Thus, the case where s is not explored cannot happen. It means that s was already explored with some path $\rho \neq min_{bfs}(s)$ before considering $min_{bfs}(s)$. By choice of s, we know that every state s' reached before s is reached by $min_{bfs}(s') = first_{bfs}(s')$. According to the algorithm, it implies that paths are considered according to their size, ties being broken using lexicographic order. That is ρ reaches s, and either $|\rho| < |min_{bfs}(s)|$, or $|\rho| = |min_{bfs}(s)|$ and $\rho \ll min_{bfs}(s)$, which contradicts the definition of $min_{bfs}(s)$. ∎

5 Connections with the Sleep Set Algorithm

In [4], Godefroid describes a state space search algorithm that is based on the concept of *sleep sets*. Intuitively, the sleep set of a state consists of actions that need not be explored from that state. It is constructed from the sleep set of the predecessor of that state in depth-first search. Unfortunately, as follows from Proposition 2 below and the example in Figure 2, the original sleep set algorithm [5] may fail to visit some of the states. One way of fixing this is to split states during search [11, 4], i.e., explore a state more than once depending on the sleep set that it inherits from its predecessor. In [6] a different approach is proposed, which is based on eliminating some actions from the sleep set. To compare our algorithms with the sleep set approach, we describe a generic sleep set-based algorithm that generalizes the algorithms of [5] and [6]. We then show how to represent our algorithm `EdgeLeanDfs` within this framework. Moreover, we show that `TNF_Dfs` is, in fact, equivalent to the algorithm of [5]. This equivalence is quite surprising since the rationale behind both algorithms is quite different.

Similarly to the algorithms of Sections 3 and 4.1, the generic sleep set algorithm that we are about to describe is based on depth-first search. For any node q, we store an associated set of actions sleep(q), which we call the *sleep set*. These are the actions we are going to ignore: if the label of an edge starting in q is in sleep(q), this edge is

not explored. In the beginning, we set sleep(s_0) = $\emptyset$; the sleep sets of all other nodes are constructed when we first discover these nodes. The sleep sets are updated during the execution of the algorithm. Whenever we backtrack to q from exploring an edge labeled with a, we add a to sleep(q). A newly discovered state inherits the sleep set of its parent, with some modifications. Namely, suppose that a state q' is discovered from a state q by exploring an edge labeled with a. Let dep(a) = $\{b \mid \neg bIa\}$, i.e., dep(a) is the set of actions dependent on a. Then sleep(q') is a subset of sleep(q) $\setminus$ dep(a). That is, to construct the sleep set for q', we take the sleep set for q and delete all actions that are dependent on a, as well as some other actions.

In the following description, the function remove(q, a) determines which actions should not be inherited by the state that is discovered from q by exploring an edge labeled with a. In the remainder of this section, we will compare algorithms that result from different implementations of this function.

```
proc SleepSetsDfs(q,sleep);
     local variables q', current;
     current:=sleep;
     hash(q);
     forall a ∉ sleep, q--a-->q' do
            begin
                  if q' not hashed then
                     rem = remove(q, a);
                     SleepSetDfs(q',(current \ rem) \ dep(a));
                  current := current ∪ {a};
            end;
end SleepSetsDfs;
```

This description leaves us with two degrees of freedom: the choice of function remove(q, a) and the order in which we explore the edges from a given vertex.

The algorithm of [5] can be seen as the most straightforward implementation of this approach. Namely, it sets remove(q, a) = $\emptyset$ for all q, a. We will now show that the algorithm `TNF_Dfs` described in Section 4.1 is equivalent to the algorithm of [5] as long as both consider actions in alphabetic order. More precisely, we prove that `TNF_Dfs` and `SleepSetDfs` with remove(q, a) $\equiv \emptyset$ ignore the same actions and explore exactly the same set of states.

Proposition 2 *Suppose that* `TND_Dfs` *and* `SleepSetDfs` *with* remove(q, a) $\equiv \emptyset$ *use the same alphabetic priority order* $\ll$*. Then during the search, from any given state* q *these two algorithms explore exactly the same successors.*

Proof Consider an action a that is in the sleep set of a state q. Suppose that q is reachable from the initial state via a path labeled with σ. Then σ can be decomposed as $\sigma = vu$ so that there is a state t reached from s_0 via v, a has been taken from t, and all the letters in u are independent of a. Also, we have $a \ll u$. Thus, if a is in the sleep set of q, Lemma 5 implies that σa cannot be in TNF.

Conversely, assume that `TNF_Dfs` does not take action a from a state q, where the path on the stack is labeled with σ. Then it has to be the case that σa is not in normal form. Then, according to Lemma 5, the string σ can be decomposed as $\sigma = vu$, where $vau \equiv vua$ and $vau \ll vu$. Let u be the longest such suffix of σ, and let t be the state

reached after v. Then a is enabled from t. Now, consider `SleepSetDfs`. If it takes action a from the state t, it will do so before taking the action that corresponds to the first letter of u, since $a \ll u$. Then a must be in the sleep set of q, and thus is not taken from it. On the other hand, since a is enabled at t, if `SleepSetDfs` does not take a from t, it must be because a is in the sleep set when `SleepSetDfs` reaches t. But this means that there is a longer suffix u' of σ such that a is independent of u', and $a \ll u'$, a contradiction with our choice of u. ∎

As shown by the example in Figure 2, `TNF_Dfs` may fail to discover some of the states. Hence, the same is true for this version of `SleepSetsDfs`. Indeed, one can run this algorithm on our example to verify that the state s_6 is not discovered. To follow the search, notice that the sleep sets are given by $Sleep(s_0) = Sleep(s_1) = Sleep(s_4) = \emptyset$, $Sleep(s_2) = Sleep(s_3) = \{b\}$ and $Sleep(s_5) = \{a\}$.

Another existing sleep set algorithm that fits into this framework is that of [6]. In this version of the algorithm, remove(q, a) consists of all actions that start in q and lead to a state that is currently on the search stack (that is, the set `rem` is independent of a and hence can be computed outside of the main loop). Note that this approach forces one to explore *each* action, as we have to check whether it leads to a state on the stack.

Finally, using an argument similar to that of Lemma 2, it is easy to see that `EdgeLeanDfs` can also be seen as an implementation of our generic algorithm. Namely, we set bigger$(a) = \{b \mid a \ll b\}$, and let remove$(q, a) =$ bigger(a). This version requires slightly more space and time than the other two, as the function remove(q, a) has to be called for each action.

Observe that both of our algorithms do not "split" states. Moreover, when we compare states to see if a newly generated state has been discovered before, we only look at the original state values. The additional data structures used by our algorithm are computed on the fly. As argued above, this provides significant memory savings compared to state-splitting algorithms.

6 Efficient Representation of Summaries

In many state systems, the actions belong to different processes, and the independence relation arises from the fact that the actions of any two processes are independent unless they are communication actions between these processes.

We will now show that in this setting the summaries can be represented with a better time and space efficiency than in the general case. Our representation applies when pairs of processes can communicate with each other synchronously. We chose to focus on this model because it is perhaps the simplest to represent and describe among concurrent executions models. Similar principles can be applied to other models, e.g., concurrency with shared variables or with asynchronous communication.

Assume that there are p processes. We partition the set of actions Σ according to the process that is involved. Local actions of process $1 \leq i \leq p$ are put in the set Σ_i. Communication actions between process i and process j (irrespectively of which process is the sender and which is the receiver) are put in the set Σ_{ij} for $1 \leq i \leq j \leq p$. Also, we identify Σ_i with Σ_{ii}. We order the sets Σ_{ij} lexicographically, with the right index being most significant, i.e., we set $\Sigma_{ij} \ll \Sigma_{kl}$ if either (1) $j < l$ or (2) $j = l$ and

$i < k$. We obtain

$$\Sigma_{11} \ll \Sigma_{12} \ll \Sigma_{22} \ll \Sigma_{13} \ll \Sigma_{23} \ll \Sigma_{33} \ll \ldots \ll \Sigma_{1p} \ll \Sigma_{2p} \ll \ldots \ll \Sigma_{pp}.$$

This ordering induces an ordering on actions in a natural way: $a \ll b$ if and only if $a \in \Sigma_{ij}$, $b \in \Sigma_{kl}$ and $\Sigma_{ij} \ll \Sigma_{kl}$.

In this model, actions of Σ_{ij} are independent of actions of Σ_{kl} (including the case where $i = j$ or $k = l$) exactly when $\{i, j\} \cap \{k, l\} = \emptyset$, i.e., when they involve disjoint processes.

Recall that the algorithm for checking the summary proceeds from right to left. The scan stops whenever it finds an action that is either dependent on the current one (and then the word is in TNF), or independent but bigger in the alphabetic order than the current action (and then the word is not in TNF). In the remaining case, namely, when it sees an action that is independent of the current action but smaller in the alphabetic order, the scan continues. The implementation of the summary used in Section 4 requires space for all the different actions, hence the size of the summary is at least linear in the number of actions $|\Sigma|$.

We first observe that, rather than storing actions in the summary, it suffices to only keep the index of the subset Σ_{ij} to which this action belongs, i.e., a singleton or a pair of numbers between 1 and p. Indeed, the algorithms that use the summary data structure treat all internal actions of a given process, as well as all communication actions between a given pair of processes, in the same way. This brings the size of the summary down to the size of the communication graph of the system (i.e., an undirected graph where nods represent processes, and there is an edge between two nodes if there is a potential communication between them), which is at most $O(p^2)$.

We will now show how to further reduce the size of the summary to p. We will make use of two indices $i, \hat{i}$ per process $i \in \mathcal{P}$, which will never occur simultaneously in the summary. We redefine the summary of a word σ inductively, as follows.

Definition 5 The summary of ϵ is ϵ. Let u be the summary of a word σ, and let $a \in \Sigma_{ij}$. Let $\mathrm{del}_{ij}(u)$ be an operation that deletes all of the indices i, $\hat{i}$, j, $\hat{j}$ from u. Then the summary of σa is $\mathrm{del}_{ij}(u)\hat{j}i$ if $i \neq j$, and $\mathrm{del}_{ij}(u)\hat{j}$ if $i = j$.

Note that when a summary is updated, the indices are inserted in the reversed order at the right end of the summary. For example, consider the words a and ab with $a \in \Sigma_{ij}$ and $b \in \Sigma_{ik}$. Under the old summary definition, the entries of the summary are pairs of indices, and therefore summary for a is $\langle i, j\rangle$ and the summary for ab is $\langle i, j\rangle\langle i, k\rangle$. On the other hand, under the definition proposed above, the summary for a is $\hat{j}i$ and the summary for ab is $\hat{j}\hat{k}i$.

Now, in order to check whether a word σa is in TNF, with σ in TNF and $a \in \Sigma_{ij}$, we perform a scan of the summary of σ from right to left, stopping whenever we find an index $k \in \{i, \hat{i}, j, \hat{j}\}$ (and then the word is in TNF), or when we find an index $\hat{l}$ with $l > j$ (and then the word is not in TNF). Notice that $j \geq i$, hence $l > j$ implies that $l \notin \{i, j\}$.

The intuition behind this algorithm is as follows. Updating the summary by putting the indices of the new action in reversed order enables us to give priority to checking dependence over checking alphabetic order. Accordingly, until subsequent changes of summary further shift one of the indices of an action to the right, its left index is seen before its right index during the scan. Hence, for any letter b added to σ before a, we have the following possibilities. If one of the indices of b is equal to one of the

indices of a, then a and b are dependent. Otherwise, a and b are independent, and their right indices (with a hat) are different. In this case, we can check the alphabetic order between a and b solely by looking at their right indices. (by the alphabetic ordering we defined, the left index is irrelevant for the comparison when the right index is different).

We now formally prove the correctness of the algorithm.

Theorem 4 *Let σ be a word in TNF and let $a \in \Sigma_{ij}$. The algorithm described above correctly decides whether σa is in TNF.*

Proof First, we consider the case where some index $\hat{l}$ with $l > j$ was found during the summary scan. By construction $j \geq i$, so $l \notin \{i, j\}$. In this case, the algorithm decides that σa is not in TNF. Consider the moment when $\hat{l}$ was last inserted into the summary. This was due to an occurrence of an action $b \in \Sigma_{kl}$. Decompose σ into $\rho b \mu$, where μ does not contain any occurrences of b. We have $a \ll b$, since the right index l of b is strictly bigger than right index j of a. We claim that all actions in $b\mu$ are independent of a. Assume for the sake of contradiction that this is not the case. Then the last occurrence of a letter c in $b\mu$ that is dependent of a would have caused an index $t \in \{i, \hat{i}, j, \hat{j}\}$ to occur to the right of the index l in the summary. Note that this includes the case $c = b$: in this case, since l is the right index of b and $l \notin \{i, j\}$, t would have to be the left index of b, which by construction is inserted into the summary after $\hat{l}$. Such an index t would have been detected and stopped the search before l was reached, a contradiction. Now, since all actions in $b\mu$ are independent from a, and $a \ll b$, it follows from Lemma 6 that σa is not in TNF, as correctly concluded by the algorithm.

Now suppose that an index $m \in \{i, \hat{i}, j, \hat{j}\}$ is found on the right-to-left summary search. In this case, the algorithm concludes that σa is in TNF. Again, the last update of m in the summary is due to the occurrence of some action b, and we decompose σ into $\rho b \mu$, where μ does not contain any occurrences of b. Clearly, a and b are dependent since they share the index m. Now, by the same argument as in the previous case, all actions in μ are independent of a. In order to use Lemma 6 to prove that the conclusion of the algorithm is correct, it suffices to show that μ does not contain an action c such that $a \ll c$.

Suppose that this is not the case, and consider an arbitrary action c in μ that satisfies $a \ll c$. As all actions in μ are independent from a, we have $c \in \Sigma_{kl}$ for some $l > j$. Let t be the largest value of l such that for some $k = 1, \ldots, p$ the string μ contains an action $c \in \Sigma_{kl}$. As c appears in μ, we have $t \geq l > j$. Now, among all actions in $\cup_{k=1}^{t} \Sigma_{kt}$, let c' be the one that occurs last in μ. Suppose that $c' \in \Sigma_{kt}$.

In the summary, we put $\hat{t}k$ on the far right of the summary when this occurrence of c' is seen. Since no action involving t occurs after c', $\hat{t}$ is not deleted (by putting t instead of $\hat{t}$ in the summary). But then the search would have found this $\hat{t}$ with $t > j$ before m, and consequently would have stopped and concluded that σa is not in TNF, a contradiction.

Thus, in both cases the algorithm behaves correctly, and the proof is complete. ∎

Our algorithm can easily be extended to the case where more than two processes can communicate at the same time. In this case, we partition the actions into to sets with the same communication participants (e.g., Σ_{245} will involve processes 2, 4 and 5). The order between subsets is lexicographic with more significant indices further to the right. We make the indexing equilength when needed by repeating the last index. When updating the summary, only the rightmost index is marked with a hat (thus, 2, 4 and $\hat{5}$ in the above example), and indices are put on the extreme right in reverse

order ($\hat{5}42$ in our example). Again, we can compare the order of actions in different sets by checking the rightmost (most significant) index only, since if that value is the same, the actions are interdependent anyway.

7 Experiments

We implemented the algorithms `EdgeLeanDfs` and `TNF_Dfs` from Sections 3 and 4.1 in the tool Spin [8]. We tested state space generation for several examples from the literature. The results are shown in Table 1: The rows correspond to regular depth-first search, the `EdgeLeanDfs`, and `TNF_Dfs`, respectively. For each example we give the number of states and edges explored as well as the maximal size of the stack, in units, thousands (K), and millions (M). We also list the memory used in Megabytes and runtime in seconds. It should be mentioned that we use the plain version of SPIN, without partial order reduction. The reason for this is that our method is not intended to replace the existing POR in SPIN, but rather to complement it. Notice that more memory can be *allocated* than the memory *used* we report. First Spin allocates statically memory for the stack (the actual number is usually obtained by trial and error). Second, on a 64bits machine, Spin uses 40 bytes (5 words) per stack level. When there is no rendez-vous, we implemented `EdgeleanDfs` without additional memory and `TNF_Dfs` with 1 additional byte. Rendez vous are implemented in Spin in such a way that we use an additional byte (for both `EdgeleanDfs` and `TNF_Dfs`) in case the model future rendez vous actions. That additional byte would be avoidable in other tools, and may be in Spin itself. Anyway, when 1,2 (up to 8) bytes are added, the new prototype allocates 48 bytes per stack level (instead of 40). We report the used bytes in our comparison, as the others (allocated but unused) bytes are left free for implementing new features. We also give the allocated memory in parenthesis as stated by Spin, for some arbitrary but reasonable fixed maximal value of memory for the stack.

The first two examples, DMSnoCC and DMSwithCC, are models of system-on-chip designs of a distributed memory system on message-passing network without and with cache coherency, respectively. For more details, we refer the reader to [1]. The examples RW1, RW4, and RW6 are models of various instances of the so-called Replicated Workers problem described in [2]. The rest of the models are from the test suite that comes with the standard distribution of Spin. For these models we use the default parameters (e.g., the number of processes in petersonN and leader is set to 3 and 5, respectively). For the presented examples our method improve the runtime in Spin without POR (up to 4 times), but not dramatically (sometimes being barely slower). However, one can expect huge improvement in the contexts where computing a transition has a higher cost. The number of transitions explored gives a good indication of the algorithm's runtime in other scenarios.

The memory consumption depends on whether one uses the automaton minimization technique of Spin. If this technique is used (DMS,RW6), then the size of stack is the prevailing factor in the memory consumption, and for DMSwithCC, the memory requirement is reduced by a factor of 30 for `EdgeLeanDfs`, and a factor of 150 for `TNF_DFS`. In the case where the stack size is not prevalent, then the memory consumption does not change. Anyway, the size of the stack gives a good indication of how the algorithm performs memory-wise in different scenarios.

Although `TNF_Dfs` may fail to explore the entire state space (each of our examples contains cycles), there is only one case (RW6) where we observed a difference

Table 1 Experimental Results.

DMSnoCC

	states	edges	stack [nr. of states]	memory [MB]	time [s]
Spin with regular DFS	229M	1009M	26M	1060 (1126.5)	12623
Spin with `EdgeLeanDfs`	229M	296M	264.5K	40.9 (41.1)	12620
Spin with `TNF_Dfs`	229M	265M	32.2K	31.4 (33.8)	11638

DMSwithCC

	states	edges	stack [nr. of states]	memory [MB]	time [s]
Spin with regular DFS	132M	541M	18.9M	760 (860)	8469
Spin with `EdgeLeanDfs`	132M	174M	384K	19.8 (30.9)	8523
Spin with `TNF_Dfs`	132M	151M	29.2K	5.2 (5.6)	8158

RW1

	states	edges	stack [nr. of states]	memory [MB]	time [s]
Spin with regular DFS	181K	852K	2219	18.8 (19.7)	4.32
Spin with `EdgeLeanDfs`	181K	409K	1224	18.7 (19.1)	3.18
Spin with `TNF_Dfs`	181K	339K	360	18.7 (19.1)	2.93

RW4

	states	edges	stack [nr. of states]	memory [MB]	time [s]
Spin with regular DFS	263K	1.1M	2253	26.1 (27.4)	5.34
Spin with `EdgeLeanDfs`	263K	558K	1247	26.0 (26.5)	4.22
Spin with `TNF_Dfs`	263K	462K	625	26.0 (26.5)	3.87

RW6

	states	edges	stack [nr. of states]	memory [MB]	time [s]
Spin with regular DFS	11.5M	65.6M	827K	42.9 (43.5)	560
Spin with `EdgeLeanDfs`	11.5M	59.6M	784K	42.0 (46.1)	556
Spin with `TNF_Dfs`	9.9M	41.3M	148K	12.3 (15.3)	432

petersonN

	states	edges	stack [nr. of states]	memory [MB]	time [s]
Spin with regular DFS	25362	69787	5837	3.0 (3.1)	0.06
Spin with `EdgeLeanDfs`	25362	28855	1035	2.8 (3.1)	0.05
Spin with `TNF_Dfs`	25362	28328	632	2.8 (3.1)	0.04

pftp

	states	edges	stack [nr. of states]	memory [MB]	time [s]
Spin with regular DFS	207K	604K	3077	27.3 (27.5)	1.6
Spin with `EdgeLeanDfs`	207K	480K	2578	27.2 (27.5)	1.3
Spin with `TNF_Dfs`	207K	473K	2824	27.3 (27.5)	1.4

snoopy

	states	edges	stack [nr. of states]	memory [MB]	time [s]
Spin with regular DFS	62179	213K	6877	8.5 (8.5)	0.46
Spin with `EdgeLeanDfs`	62179	193K	5670	8.4 (8.6)	0.45
Spin with `TNF_Dfs`	62179	192K	5546	8.4 (8.6)	0.44

leader

	states	edges	stack [nr. of states]	memory [MB]	time [s]
Spin with regular DFS	38863	158K	113	8.8 (9.0)	0.45
Spin with `EdgeLeanDfs`	38863	51565	112	8.8 (9.1)	0.21
Spin with `TNF_Dfs`	38863	51565	113	8.8 (9.1)	0.23

sort

	states	edges	stack [nr. of states]	memory [MB]	time [s]
Spin with regular DFS	374238	1.53M	177	62.1 (62.4)	5.6
Spin with `EdgeLeanDfs`	374238	413K	176	62.1 (62.4)	1.5
Spin with `TNF_Dfs`	374238	413K	177	62.1 (62.4)	1.7

between the number of states generated by `EdgeLeanDfs` and `TNF_Dfs`. In most of the experiments, both of our algorithms explore considerably fewer transitions than regular depth-first search. On the other hand, the difference between `EdgeLeanDfs` and `TNF_Dfs` regarding the number of transitions is not very significant. With respect to the stack size (and thus memory consumption), our algorithms are up to 1000 times better than regular DFS (see, e.g., DMS examples). Also, on many examples `TNF_Dfs` uses much less space than `EdgeLeanDfs` (see DMSwithCC, RW and the petersonN examples), while `EdgeLeanDfs` has a slight advantage for pftp, leader and sort. It is quite surprising that both algorithms explore roughly the same number of transitions, but `TNF_Dfs` needs much smaller stack than `EdgeLeanDfs` (examples DMS, RW and petersonN).

8 Conclusions and Future Work

In this paper we presented some novel algorithms for state space search optimization. The algorithms exploit the commutativity of the actions that generate the transitions of the state space. We showed that full coverage of the state space can be achieved while reducing the number of transitions. Proving this was quite-non trivial, in particular in the case of the `EdgeLean` algorithm. We implemented our DFS based algorithms `EdgeLean` and `TNF_Dfs` on top of the model checker Spin. The experiments with the prototype implementation showed that impressive savings in memory can be achieved without significant time overhead. The reduction in memory consumption is due to smaller stacks that are generated by `EdgeLean` and `TNF_Dfs`. We also discussed the relation of `TNF_Dfs` with the sleep set algorithm. In addition, we showed that, although `TNF_Dfs` does not provide a complete coverage for the state space, when cycles are present, a `BFS` version of this algorithm does provide complete coverage.

An interesting topic for future work would be to combine our algorithms with other techniques that exploit action commutativity, like various partial order techniques based on ample sets or persistent sets.

9 Acknowledgements

The work of the second author has been partially supported by EPSRC under grant GR/T07343/02 and by ESRC under grant ES/f035845/1. The work of the third author has been partially supported by ANR SETI-06 DOTS. Part of this work has been done when the second and the fourth author were with the University of Warwick.

References

1. T. Basten, D. Bošnački, M. Geilen, Cluster-based Partial Order Reduction, *Automated Software Engineering*, 11(4), pp. 365–402, Kluwer, 2004.
2. C. S. Păsăreanu, M. B. Dwyer, M. Huth, Assume-Guarantee Model Checking of Software: A Comparative Case Study, in Theoretical and Practical Aspects of SPIN Model Checking, LNCS 1680, Springer, 1999.
3. E. Clarke, O. Grumberg, D. Peled, *Model Checking*, MIT Press, 2000.
4. P. Godefroid, Partial-Order Methods for the Verification of Concurrent Systems—An Approach to the State-Explosion Problem, PhD thesis, University of Liege, Computer Science Department, November 1994.
5. P. Godefroid, P. Wolper, Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties, in *CAV 1991*, pp. 176–185, 1991.

6. P. Godefroid, G. Holzmann, D. Pirottin, State-Space Caching Revisited, *Formal Methods in System Design 7:3*, pp. 227–242, 1995.
7. G. Higman, Ordering by Divisibility in Abstract Algebra, *Proc. London Mathematical Society 2*, pp. 326–336, 1952.
8. G. Holzmann, The SPIN Model Checking, Addison Wesley, 2003.
9. A. Mazurkiewicz, Trace semantics, in *Advances in Petri Nets 1986*, LNCS 255, pp. 279–324, 1986.
10. E. Ochmanski, Languages and Automata, in *The Book of Traces*, V. Diekert, G. Rozenberg (eds.), pp. 167–204, 1995.
11. D. Peled, Combining Partial Order Reductions with On-the-fly Model-Checking, in *CAV 1994*, LNCS 818, pp. 377–390, 1994.
12. R. E. Tarjan, Depth-First Search and Linear Graph Algorithms, in *FOCS 1971*, pp. 114–121, 1971.
13. A. Valmari: A Stubborn Attack on State Explosion, Formal Methods in System Design 1(4), pp. 297–322, 1992.