EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

On commutativity based edge lean search

Citation for published version (APA):
Bosnacki, D., Elkind, E., Genest, B., & Peled, D. (2007). On commutativity based edge lean search. In L. Arge, C. Cachin, T. Jurdzinski, & A. Tarlecki (Eds.), *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP 2007) 9-13 July 2007, Wroclaw, Poland* (pp. 158-170). (Lecture Notes in Computer Science; Vol. 4596). Springer. https://doi.org/10.1007/978-3-540-73420-8_16

DOI:
10.1007/978-3-540-73420-8_16

Document status and date:
Published: 01/01/2007

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

On Commutativity Based Edge Lean Search

Dragan Bošnački[1], Edith Elkind[2], Blaise Genest[3], and Doron Peled[4]

[1] Department of Biomedical Engineering, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands
[2] Department of Computer Science, University of Southampton, U.K
[3] IRISA/CNRS, Campus de Beaulieu, 35042 Rennes Cedex, France
[4] Department of Computer Science, Bar Ilan University, Israel

Abstract. Exploring a graph through search is one of the most basic building blocks of various applications. In a setting with a huge state space, such as in testing and verification, optimizing the search may be crucial. We consider the problem of visiting all states in a graph where edges 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 (this is the case when the actions are performed independently by different processes). We show how to use commutativity to achieve full coverage of the states while traversing considerably fewer edges.

1 Introduction

In many applications one has to explore a huge state space using limited resources (such as time and memory). Such applications include software and hardware testing and verification [4], multiagent systems, games (e.g., for the purpose of analyzing economic systems), etc. In such cases, it is obviously important to optimize the search, traversing only the necessary states and edges.

In this paper, we consider the problem of searching a large state space, where transitions between states correspond to a finite number of *actions*. We do not assume that the entire system is given to us as an input. Rather, we are given an initial state, and a method to generate states from one another. More specifically, for each state, there can be one or more actions available from this state. Executing an available action leads to another state. Also, we are given a fixed *independence* relation on actions: if two actions are independent, then executing them in any order from a given state leads to the same state. It is easy to model many of the problems in the above-mentioned application areas in this framework (we provide specific examples later in the paper). Traversing an edge and checking whether it leads to a new state has a cost. Hence, we want to predict if an edge is redundant (i.e., leads to a state that we have already visited or that we will necessarily visit in the future) without actually exploring it. Exploring fewer edges may also reduce the size of the search stack, and in particular reduce the memory consumption. Intuitively, the independence relation between actions should be useful here: if two sequences of actions lead to the same state,

L. Arge et al. (Eds.): ICALP 2007, LNCS 4596, pp. 158–170, 2007.
© Springer-Verlag Berlin Heidelberg 2007

it suffices to explore one of them. Of course, it will defy our goal to use a lot of overhead, both in terms of time and space spent computing the subset of edges to be explored.

The main contribution of this paper is an efficient state space search algorithm, which we call *commutativity-based edge-lean* search (CBEL). 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). Another approach, which is inspired by trace theory [11,12], is to only consider paths that correspond to sequences in trace normal form (TNF) (defined later in the paper). This method provides a more powerful reduction than the one described above, as it only explores one sequence in each trace. In Section 4, we investigate this idea in more detail, describing an efficient data structure (called *summary*) that supports this search technique. We prove that the TNF-based algorithm is guaranteed to visit all states as long as the underlying state system contains no directed cycles. Unfortunately, if the state system is not loop-free, this algorithm may fail to reach some of the states: we provide an example in Section 4. While this limits the scope of applicability of this algorithm, many state systems, especially the ones that arise from game-theoretic applications, are naturally acyclic. Whenever this is the case, the TNF-based algorithm should be preferred over the algorithm of Section 3.

A related approach is the family of partial order reductions [13,5,14]. As opposed to our algorithms, in general these methods — known as *ample* sets, *persistent* sets, or *stubborn* sets, respectively, — do not preserve the property of visiting all the states, but 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 [5], in particular, the non-state-splitting sleep set algorithms. In Section 5, we show that our TNF-based algorithm generates, in fact, exactly the same reduced graph as the very first version of the sleep set algorithm proposed in [6] (when edges are explored according to a fixed order between actions). The counterexample of Section 4 (see Figure 1) can therefore be seen as an explanation why in the presence of cycles, all existing sleep set algorithms have to use additional techniques to visit all states. In particular, the algorithm of [7] generates and checks back-edges, and discards them when redundant, thus paying the cost of checking some of the redundant edges. A fix suggested in [5,13] requires splitting nodes into several copies, which may increase the number of effective states. Surprisingly, our edge lean algorithm achieves a full coverage of the states without incurring these costs, and even in the presence of cycles.

An obvious application area for this technique is model checking and verification, but it can be useful in many other domains as well (see the full version of the paper [2]) To illustrate this, in Section 6 we describe an example from bioinformatics where one can use this method. In Section 7, we provide the results of

several experiments that compare our algorithms with classical depth-first search used in SPIN [9]. Our experiments show that for a number of natural problems, our methods provide a dramatic reduction in the number of edges explored and the stack size.

2 Preliminaries

A *system* is a tuple $\mathcal{A} = \langle S, s_0, \Sigma, T \rangle$ such that

- S is a finite set of *states*.
- $s_0 \in S$ is an *initial state*.
- Σ is the finite *alphabet of actions*.
- $T \subseteq S \times \Sigma \times S$ is a labeled transition relation. We write $s \xrightarrow{a} s'$ when $(s, a, s') \in T$.
- A symmetric and irreflexive relation $I \subseteq \Sigma \times \Sigma$ on letters, called the *independence relation*. We require that independent transitions $a\,I\,b$ satisfy the following *diamond* condition for every state s:

  If $s \xrightarrow{a} q \xrightarrow{b} r$ then there exists $q' \in S$ such that $s \xrightarrow{b} q' \xrightarrow{a} r$. In this case, we say that the system has the *diamond* property.

  Note that we do not require the other common diamond condition:

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

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 $i \neq j$. Its *labeling* is $\ell(\rho) = a_1 \cdots a_n$.

Definition 1. *Let $\sigma, \rho \in \Sigma^*$. Define $\sigma \overset{1}{\equiv} \rho$ iff $\sigma = uabv$ and $\rho = ubav$, where $u, v \in \Sigma^*$, and $a\,I\,b$. Let $\sigma \equiv \rho$ be the transitive closure of the relation $\overset{1}{\equiv}$. This equivalence relation is often called* trace equivalence *[11].*

That is, ρ is obtained from σ (or vice versa) by commuting the order of an adjacent pair of letters. 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 $s \xrightarrow{u} r$ iff $s \xrightarrow{v} r$.

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

All the search algorithms to be presented are based on depth-first search (DFS), which provides a space complexity advantage over breadth-first search:

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

3 An Edge Lean Algorithm for Complete State Coverage

A key idea to reduce the number of explored edges is to make use of the diamond property, defined in the previous section.

Definition 2. *Let $w \in \Sigma^*$. Denote by $\tilde{w}$ the least word under the relation $\ll$ equivalent to w. If $w = \tilde{w}$, then we say that w is in* trace normal form *(TNF) [12].*

As $\tilde{w} \equiv w$, any state that can be reached by a path labeled with 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 (see Section 4). The main reason for this is that all paths explored during depth-first search are necessarily acyclic. Hence, by using this method, we 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. Unfortunately, it turns out that combining these limitations may result in leaving some states unexplored; we provide an example in Section 4. Therefore, for general state systems 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.

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, this means that w cannot be reduced, i.e., 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. Notice that if w is in TNF, then it is irreducible. The converse does not necessarily hold.

Our algorithm `EdgeLeanDfs` explores *some* irreducible paths in depth-first manner. For this, it suffices to remember the last letter a seen along the current path, and not to extend it with letter b whenever $a\,I\,b$ and $b \ll a$.

```
EdgeLeanDfs(s0,ϵ);

proc EdgeLeanDfs(q,prev);
     local 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_cbel(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_cbel(s)=$\perp$.

Theorem 1. *For any $s \in \mathcal{A}$, we have* first_cbel*(s)$\neq \perp$, i.e., our algorithm explores all states. This implies that* `EdgeLeanDfs`*(s_0, ϵ) is correct.*

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 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.

Claim 1. *For any path ρ_s, there exists a path ρ'_s that is simple and irreducible.*

Proof. We iteratively (1) delete loops and (2) rearrange the labels to obtain an irreducible path. Each application of (1) strictly decreases the length of the path, while (2) does not change its length. The path obtained also leads to s. We obtain a simple irreducible path 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 first_cbel(t)$=\rho_t$, *blue* if first_cbel(t)$\neq \perp$, but first_cbel(t)$\neq \rho_t$, and *white* if first_cbel(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 1. *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 first_cbel(t), i.e., set ρ=first_cbel(t)σ. Set $w = y = \ell$(first_cbel(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 Claim 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 1, 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.

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 $b \ll a$, $\sharp_b(v) = \sharp_b(w)$ and $\sharp_a(v) < \sharp_a(w)$.*

Claim 2. *The relation $<_\sharp$ is a well-founded (partial) order, i.e., there does not exist an infinite sequence $u_1, u_2, \ldots, u_i \in \Sigma^*$ 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 2. *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 the lemma, let us show that it implies Theorem 1. Indeed, by Claim 2, 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 1 we obtain a path of our algorithm that reaches s. We now prove Lemma 2.

Proof. We use the notation introduced in the description of T: we have $w = \ell(\text{first_cbel}(t))$, $v = \ell(\sigma)$, and $x = wv = \ell(\rho)$ after ρ_t was replaced by $\text{first_cbel}(t)$.

In the rest of the proof, we abuse notation by using 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, 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.

Let us color all the letters in the word wv so that all letters in w are yellow and all letters in v are green. By construction at any point in time all letters in y are yellow, and therefore all letters pushed into z during Step (3) are yellow. We construct a directed acyclic graph (DAG) whose set of nodes includes all yellow occurrences of letters in z as well as some of the green occurrences of 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).

Claim 3. *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 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. ■[Claim 3]

Let G be the set of nodes of our DAG that have incoming edges, but no outgoing edges. By Claim 3 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 first_cbel(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 Claim 3, we conclude that v' does not contain any of the letters in G.

Let a be the minimal (for $\ll$) letter of G. 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 of the DAG leading from c to some $b \in G$. By construction of the graph, the existence of a path from c to b implies $c \gg b$, and hence $c \gg a$. Hence, for all b in v' with $b \ll a$ or $b = a$, b is green, hence $\sharp_b(v') \leq \sharp_b(v)$, and $a \in G$ is in v but not in v', hence $\sharp_a(v') < \sharp_a(v)$, that is $v' <_\sharp v$. ■[Lemma 2,Theorem 1]

4 An Efficient Reduction for Cycle Free State Spaces

It can be argued that the reduction of Section 3 is not optimal: let $a \ll b \ll c$, $a\,I\,b, b\,I\,c$ and $\neg(a\,I\,c)$. Let $x = cab$ and $y = bca$. Then we have $x \equiv y$, i.e., the states reached after x and y are the same. However, both x and y are irreducible, since $a \ll b$ and $\neg(a\,I\,c)$. Therefore, the algorithm of Section 3 will explore both of the paths labeled by x and y.

In this section, we describe an algorithm `TNF_Dfs`(s_0) that only explores paths labeled with words in trace normal form. This algorithm often provides a significant reduction in the size of stack needed, both compared to `DFS` and `EdgeLeanDfs`. For acyclic state spaces, `TNF_Dfs`(s_0) explores all states. However, it may not be the case in general. In the end of this section, we provide an example in which some of the states are not reached. Denote by $\alpha(\sigma)$ the set of letters occurring in σ.

Definition 4. *A* summary *of a string σ is the total order $\prec_\sigma$ on the letters from $\alpha(\sigma)$ such that $a \prec_\sigma b$ iff the last occurrence of a in σ precedes the last occurrence of b in σ. That is, $\sigma = vaubw$, where $v \in \Sigma^*$, $u \in (\Sigma \setminus \{a\})^*$, $w \in (\Sigma \setminus \{a, b\})^*$.*

Our reduction will be based on generating paths that are in TNF. The proofs of the following two lemmas can be found in the full version of the paper [2].

Lemma 3. *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) $vau \ll vu$.*

Intuitively, Lemma 3 means that we can commute the last a in vua backwards over u to obtain a string that is smaller in the alphabetic order than vu. The following lemma shows how we can use a summary to decide whether σa is in TNF. It implies that it suffices to consider the suffix of the summary that commutes with a, and look among these letters for one that comes *after* a in the alphabetic order.

Lemma 4. *Let $\sigma \in \Sigma^*$ in TNF and $a \in \Sigma$. Then σa is not in TNF if and only if there is a $b \in \alpha(\sigma)$ such that $a \ll b$ and for each c such that $b \preceq_\sigma c$, aIc.*

To perform a reduced depth-first search (DFS) that only considers strings in TNF, we store the summary 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, i.e., the value of the summary is calculated on the fly and not stored with the state information.

The reduced DFS procedure `TNF_Dfs`(s_0) considers all transitions enabled at the current state. For each of them, it checks whether the current string augmented with this transition is in TNF. This is done through a call to the function `normal`, which checks the summary, according to Lemma 4. The pseudocode description of auxiliary functions used by `TNF_Dfs` can be found in [2].

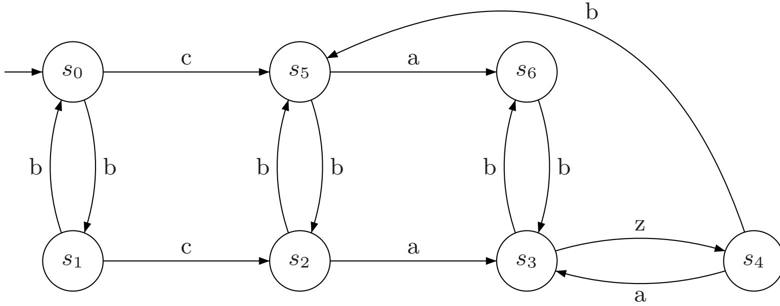

Fig. 1. A state space for which `TNF_DFS` does not explore every state

```
proc TNF_Dfs(q);
     local 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;
```

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

Proof. We show that every state s is reached by the path first(s), where first(s) stands for the path labeled by the minimal (for $\ll$) word reaching s. Clearly, first(s) is in TNF. By contradiction, if it is not the case, take the state with the smallest first(s) such that s is not explored by first(s). Then first(s) $= ua$, with u reaching t with $u \ll$ first(s), hence $u =$ first(t). When considering a, ua is in TNF and *acyclic*, hence s will be reached by $ua =$ first(s), and since first(s) is minimal for $\ll$, no other path that reaches s has been considered before. ∎

Unfortunately, for graphs that contain cycles, the conclusion of Theorem 2 is no longer true since ua and the minimal word reaching s may have loops. Figure 1 provides an example of a (diamond closed) graph that is not fully covered by the TNF algorithm (and hence, as shown in the next section, neither by the `SleepSetsDfs` version of the sleep set algorithm). The nodes, except s_6, are ordered 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 given by $b\,I\,a$, $b\,I\,c$. 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 first(s_3) $= bca$ and first(s_5) $= bcazb$. Neither $bcab$ ($\equiv bbca$) nor $bcazba$ ($\equiv bcaaab$) 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`.

5 `TNF_Dfs` and Sleep Sets

In the full version of the paper, we explore the relationship between the algorithms of Sections 3 and 4, and the sleep set approach [5]. Here, we show that the most straightforward version of the sleep set approach [6] is equivalent to `TNF_Dfs`.

```
proc SleepSetsDfs(s,sleep);
     local s', current;
     current:=sleep;
     hash(s);
     forall a ∉ sleep, s --a--> s' do
            begin
                  if s' not hashed then
                     SleepSetDfs(s',current \{b | ¬(b I a)});
                  current := current ∪ {a};
            end;
end SleepSetsDfs;
```

Lemma 5. *Assume that both* `TNF_Dfs` *and* `SleepSetsDfs` *use the same alphabetic priority order* $\ll$*. Then from any given state* s *during the search they explore exactly the same successors.*

Proof. Consider an action a that is in the sleep set of a state s. Suppose that s is reachable from the initial state via a path labeled with σ in TNF. 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. According to the priority $\ll$, we have $a \ll u$. Thus, if a is in the sleep set of s, according to Lemma 3, σa cannot be in normal form.

Conversely, assume that `TNF_Dfs` does not take a transition labeled with a after a state s, where the path on the stack is labeled with σ. This is because σa is not in normal form. As per Lemma 3, let u be the longest suffix of σ such that $\sigma = vu$, $vau \equiv vua$ and $vau \ll vu$. Let t be the state reached after v. Then a is enabled from t. Now, consider the sleep set algorithm. If a is taken from t, it will be taken before the first letter of u, according to the lexicographic order priority (since $a \ll u$). Then a must be in the sleep set of s, and thus is not taken from it. Since a is enabled at t, if a is not taken from t, it must be because a is in the sleep set when we reach 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 to the maximality of u. ∎

6 Applications

The idea of speeding up state-space search by using independence relation between actions is well-known in the model-checking community. In this section, we present an example motivated by bioinformatics, where one can also apply this technique. In [2], we provide examples from other domains such as voting theory, cellular automata theory and auction theory.

Mining Maximal Frequent Itemsets. Mining maximal itemsets is an important problem in data mining (e.g. [8]) with various applications in other areas like, for instance, bioinformatics [10]. We assume a set $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ of m distinct items and a database of n transactions $\mathcal{D} = t_1, t_2, \ldots, t_n$. Each transaction is a subset of $\mathcal{I}$. Let $X \subseteq \mathcal{I}$. We define the *support* $\sigma(X)$ of X as the number of transactions in which X occurs as a subset. A set is *frequent* iff $\sigma(X) \geq min_{sup}$, with some *minimum support value* min_{sup} . A frequent set is *maximal* if it is not a subset of any other frequent set. The algorithm for generating all maximal frequent sets [8] is a backtracking algorithm that, beginning with an empty set, builds frequent sets by adding one item at a time. An item is added to the current (frequent) set only if the new set that is obtained is frequent too. The algorithm generates a state space whose states are frequent sets. The transitions correspond to the actions of adding a new item to the set. The choice which item will be added to the set is obviously non-deterministic. Two actions are independent if they add items that are contained in all sets of the database. The maximal frequent sets correspond to "deadlock" states, i.e., states (sets) that cannot be further extended. Since added items are never removed from a set the state space graph is acyclic. A variant of this algorithm is applied in bioinformatics for finding similarities between biological networks [10]. For this purpose the original problem is transformed into a one of finding maximal subgraphs in a collection of undirected graphs. The graphs are represented as sets of edges, therefore there is a one-to-one correspondence with the original problem (edges, graphs, collection of graphs, vs. items, sets, database, respectively).

7 Experiments

We implemented the algorithms `EdgeLeanDfs` and `TNF_Dfs` from Sections 3 and 4 in the tool Spin [9]. We tested state space generation of examples from the literature. The results are shown in Table 1: The columns correspond to regular depth-first search, the edge lean algorithm, and the TNF-based algorithm, respectively. For each example we give the number of states and edges explored, and the maximal size of the stack, in unit, thousands (K) and millions (M). 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([1]). The examples RW1, RW4, and RW6 are models of various instances of the so-called Replicated Workers problem described in [3]. The rest of the models are from the test suite that comes

Table 1. Experimental Results

	Spin with regular DFS			Spin with `EdgeLeanDfs`			Spin with `TNF_Dfs`		
model	states	edges	stack	states	edges	stack	states	edges	stack
DMSnoCC	229M	1009M	26M	229M	296M	47,3K	229M	265M	32,2K
DMSwithCC	132M	541M	18,9M	132M	174M	384K	132M	151M	29,2K
RW1	181K	852K	2219	181K	409K	1224	181K	339K	360
RW4	263K	1.1M	2253	263K	558K	1247	263K	462K	625
RW6	11.5M	65.6M	827K	11.5M	59.6M	784K	9.9M	41.3M	148K
petersonN	25362	69787	5837	25362	28855	1035	25362	28328	632
pftp	207K	604K	3077	207K	480K	2578	207K	473K	2824
snoopy	62179	213K	6877	62179	193K	5670	62179	192K	5546
leader	38863	158K	113	38863	51565	112	38863	51565	113
sort	374238	1.53M	177	374238	413K	176	374238	413K	177

with the standard distribution of Spin. We observed in only one case (RW6) a difference between the number of states generated by EdgeLean and TNF.

In most of the experiments, both of our algorithms explore roughly the same number of edges, considerably fewer than regular depth-first search. With respect to the stack size (and thus memory consumption), our algorithms are up to a thousand times better than regular DFS. Also, on many examples TNF uses much less space than EdgeLean, but the converse is never the case.

References

1. Basten, T., Bošnački, D., Geilen, M.: Cluster-based Partial Order Reduction. Automated Software Engineering 11(4), 365–402 (2004)
2. Bošnački, D., Elkind, E., Genest, B., Peled, D.: On Commutativity Based Edge Lean Search (full version), `http://perso.crans.org/~genest/BEGP.ps`
3. Păsăreanu, C.S., Dwyer, M.B., Huth, M.: Assume-Guarantee Model Checking of Software: A Comparative Case Study. In: Dams, D.R., Gerth, R., Leue, S., Massink, M. (eds.) Theoretical and Practical Aspects of SPIN Model Checking. LNCS, vol. 1680, Springer, Heidelberg (1999)
4. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
5. Godefroid, P.: 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)
6. Godefroid, P., Wolper, P.: Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 176–185. Springer, Heidelberg (1991)
7. Godefroid, P., Holzmann, G., Pirottin, D.: State-Space Caching Revisited. Formal Methods in System Design 7(3), 227–242 (1995)
8. Gouda, K., Zaki, M.J.: Efficiently Mining Maximal Feqent Itemsets. In: IEEE International Conference on Data Mining (ICDM '01), pp. 163–170 (2001)
9. Holzmann, G.: The SPIN Model Checking. Addison Wesley, Reading (2003)

10. Koyuturk, M., Grama, A., Szpankowski, W.: An Efficient Algorithm for Detecting Frequent Subgraphs in Biological Networks. Bioinformatics 20, i200–i207 (2004)
11. Mazurkiewicz, A.: Trace semantics. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Advances in Petri Nets. LNCS, vol. 255, pp. 279–324. Springer, Heidelberg (1986)
12. Ochmanski, E.: Languages and Automata. In: Diekert, V., Rozenberg, G. (eds.) The Book of Traces, pp. 167–204 (1995)
13. Peled, D.: Combining Partial Order Reductions with On-the-fly Model-Checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)
14. Valmari, A.: A Stubborn Attack on State Explosion. Formal Methods in System Design 1(4), 297–322 (1992)