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 Liverpool Liverpool L69 3BX, United Kingdom
[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 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 [3], 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, 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 spent calculating the subset of edges that need to be explored, and in term of additional space required for supporting the search per each state.

We provide a simple solution for *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 [12, 13], 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 multi-agent and bioinformatics applications are naturally acyclic (for examples, see Section 6). 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 [14, 6, 15]. As opposed to our edge-lean algorithm, in general the 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 [6], 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 [7] (when edges are explored according to a fixed order between actions). Our counterexample in Section 4, shows that in the presence of cycles, not all the states are explored. This counterexample can then 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 one in [8] generates and checks back-edges, discards them when redundant, thus pays the cost of checking some of the redundant edges. A fix suggested in [6, 14] allows spliting 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 any such proviso, and even in the presence of cycles.

An obvious application area for this technique is model checking and verification. However, our methods can also be applied in other domains. To illustrate this, in Section 6 we describe examples from areas as diverse as voting theory, cellular automata theory, data mining and bioinformatics where one can use these techniques. We hope that our search methods will find applications in other fields as well. In Section 7, we provide the results of several experiments that compare our algorithms with classical depth-first search used in SPIN [10]. 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 $u_i \neq u_j$ 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 relation is often called* trace equivalence *[12].*

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:

```
Dfs(s_0);

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) [13].*

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 reach *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 them 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 weaker 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 all 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 $b \ll a$. This algorithm is given below:

```
EdgeLeanDfs(s_0,ε);

proc EdgeLeanDfs(q,prev);
     local q';
     hash(q);
     forall q --a--> q' such that prev = ε or ¬(a I prev) 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 with t blue and σ with $\rho = \rho_t\,\sigma$. Modify ρ by replacing ρ_t with first_cbel(t), i.e., set ρ=first_cbel(t).σ. Set $w = y = \ell(\text{first\_cbel}(t))$, $v = z = \ell(\sigma)$ and $x = y.z = \ell(\rho)$.
(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 after a finite number of steps. 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.

To continue, we need the following definition.

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. To complete the proof of Theorem 1, it suffices to prove the following lemma.

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. Our algorithm provides a significant reduction in the size of stack needed, both compared to `DFS` and `EdgeLeanDfs` (see Section 7), while keeping time and space overhead small. For acyclic state spaces, `TNF_Dfs`(s_0) explores all states. However, as hinted in Section 3, 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. It is possible to overcome this limitation by revisiting a state along a loop (as with some sleep set algorithms [6]), but then most of the gains would be lost.

The algorithm `TNF_Dfs`(s_0) is based on exploring only paths that correspond to some normal form of Mazurkiewicz traces [12]. 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.

Lemma 3. *Let $\sigma \in \Sigma^*$ be in TNF, and $a \in \Sigma$. Then σa is* not *in TNF exactly when we can decompose $\sigma = vu$, such that (i) $vau \equiv vua$ and (ii) $vau \ll vu$.*

Proof. If the two conditions (i) and (ii) hold, then obviously vua cannot be in TNF since it is not minimal under the alphabetic order among sequences equivalent to it.

Conversely, let ρ be the minimal string such that $\rho \equiv \sigma a$. Denote by $\mathit{first}(v)$ the first letter of a nonempty string v. Let v be the maximal common prefix of ρ and σ (and thus also of σa). Let u, v 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 an a.
   (a) u does not contain an a. Then $au \equiv ua$, satisfying (i).
   (b) u contains a. Write $u = u_1 a u_2$, where u_1 contains no a's. Then $u = u_1 a u_2 \equiv a u_1 u_2$. Since $\rho = vw \ll vua$, we have that $a = \mathit{first}(w) \ll \mathit{first}(u_1) = \mathit{first}(u)$. 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 an a.
   Write $w = w_1 a w_2$, where w_2 does not contain an 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 that $\mathit{first}(w_1) = \mathit{first}(w) \ll \mathit{first}(u)$. Thus, $vw_1w_2 \ll vu = \sigma$ and $vw_1w_2 \equiv vu$. This contradicts the fact that σ is in TNF. ∎

Intuitively, lemma 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. Since $|\sigma|$ is usually quite larger than the size of the summary (essentially $|\Sigma|$), this makes the generation of normal forms much more efficient.

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

Proof. Suppose that σ is in TNF and σa is not. Let u be the shortest suffix of σ according to the conditions of the lemma 3, i.e., $\sigma = vu$ and $vau \equiv vua$. Let b be the first letter of u. Then $a \ll b$. Let $C = \alpha(u)$. We have aIc for each $c \in C$, hence at least for each $b \preceq_\sigma c$.

Conversely, let $b \in \alpha(\sigma)$ a letter satisfying the conditions of the Lemma. Let u be the shortest suffix of σ that begins with b. Since $\prec_\sigma$ is the summary of σ, it follows that all the letters $c \in \alpha(u)$ satisfy $b \preceq_\sigma c$, and $a\,I\,c$. This means that (a) and (b) from the previous lemma hold. ∎

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. There is no need to save the value of the summary with the state information; it is calculated on-the-fly and has the appropriate value for the current state.

```
size:=0;
TNF_Dfs(s0);

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;
```

In order to perform the update, we need to keep the last transition a that was executed, and its old location i (0 if it was not introduced yet) in the summary. The updating is performed using the procedure `update_sumr`. It pushes all the elements from the ith location 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 in the shift, but in this case the size of the summary is increased.

```
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;
```

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

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

Recovering the summary upon backtracking is done using the procedure `recover_sumr`. It reverses the effect of `update_sumr` by shifting the vector 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;
```

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.

```
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;
```

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. ∎

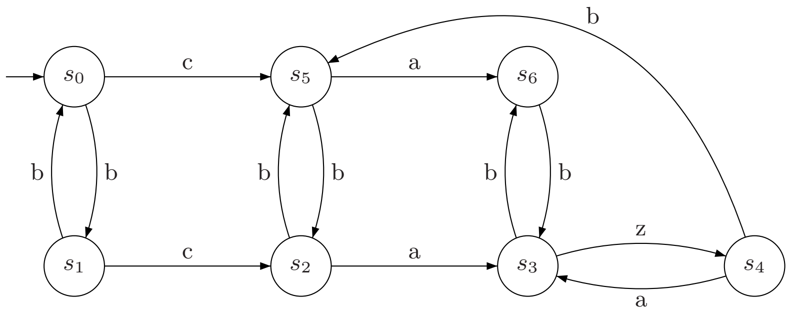

Fig. 1. A state space for which `TNF_DFS` does not explore every state.

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 $\mathrm{first}(s_3) = bca$ and $\mathrm{first}(s_5) = bcazb$. Now, neither $bcab$ nor $bcazba$, correspondingly, are 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 Connections with sleep sets

In [6], 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 transitions 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 Lemma 5, the original sleep set algorithm [7] may fail to visit some of the states. One way of fixing this is to split states during search [14, 6], i.e., explore a state more than once depending on the sleep set that it inherits from its predecessor. Paper [8] proposes a different approach that is based on eliminating some transitions 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 [7] and [8]. We then show how to represent our edge-lean algorithm within this framework. Moreover, we show that our TNF-based algorithm is, in fact, equivalent to that of [7].

Similarly to the algorithms of Sections 3 and 4, our generic sleep set algorithm is based on depth-first search. For any node s, we store an associated set of actions $\mathrm{sleep}(s)$, which we call the *sleep set*. These are the actions we are going to ignore: if the label of an edge starting in s is in $\mathrm{sleep}(s)$, this edge is not explored. In the beginning, we set $\mathrm{sleep}(s_0) = \emptyset$; 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 s from exploring an edge labeled a, we add a to $\mathrm{sleep}(s)$. A newly discovered state inherits the sleep set of its parent, with some modifications. Namely, suppose that a state s' is discovered from a state s by propagating over an edge labeled with a. Let $\mathrm{dep}(a) = \{b \mid \neg bIa\}$, i.e., $\mathrm{dep}(a)$ is the set of actions dependent of a. Then $\mathrm{sleep}(s')$ is a subset of $\mathrm{sleep}(s) \setminus \mathrm{dep}(a)$. That is, to construct the sleep set for s', we take the sleep set for s and delete all actions that are dependent on a, as well as some other actions.

In the following description, the function $\mathrm{remove}(s, a)$ determines which transitions should not be inherited by the state that is discovered from s by exploring transition a. In the remainder of this section, we will compare algorithms that result from different implementations of this function.

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

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

Clearly, the algorithm of [7] can be seen as the most straightforward implementation of this approach: namely, it sets $\mathrm{remove}(s, a) = \emptyset$ for all s, a. As we prove in Lemma 5, the algorithm of Section 4 is equivalent to this algorithm as long as it considers actions in alphabetic order. More precisely, we prove that the TNF algorithm described in Section 4 and the `SleepSetDfs` with $\mathrm{remove}(s, a) \equiv \emptyset$ ignore the same transitions and explore exactly the same set of states.

Lemma 5. *Assume the same alphabetic priority order $\ll$ used by both the reduced DFS based on TNF, and the sleep set search algorithm with* $\mathrm{remove}(s, a) \equiv \emptyset$ *Then from any given state s during the search we 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. 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 according to the TNF reduction algorithm we do 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. ∎

As shown by the example in Section 4, our TNF-based algorithm 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 as follows: $Sleep(s_0) = Sleep(s_1) = Sleep(s_4) = \emptyset$, $Sleep(s_2) = Sleep(s_3) = \{b\}$ and $Sleep(s_5) = \{a\}$.

Another existing sleep set algorithm that fits into this framework is that of [8]. It this version of the algorithm, $\text{remove}(s, a)$ consists of all transitions that start in s 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 us to look at each transition, as we have to check whether it leads to a state on the stack.

Finally, using an argument similar to that of Lemma 5, it is easy to see that the edge lean algorithm presented in Section 3 can also be seen as an implementation of our generic algorithm. Namely, we denote $\text{bigger}(a) = \{b \mid a \ll b\}$, i.e.,the set of actions bigger than a in the alphabetic order, and set $\text{remove}(s, a) = \text{bigger}(a)$.

This version requires slightly more space and time than the other two, as the function $\text{remove}(s, a)$ has to be called for each transition.

Observe that both of our algorithms do not "split" states. Moreover, when we compare states (upon generation, for backtracking), we only compare the original state values. The additional data structures used by our algorithm are computed on-the-fly. As argued above, this provides significant time and memory savings compared to state-splitting algorithms.

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 several examples from areas as various as bioinformatics, auction theory, voting theory and game theory, where one can also apply the techniques we developped in this paper.

Mining Maximal Frequent Itemsets Mining maximal itemsets is an important problem in datamining (e.g. [9]) with various applications in other areas like, for instance, bioinformatics [11]. 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 [9] 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 [11]. 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).

Consecutive auctions with budgets Suppose than we have m items for sale and there are n agents, each of which is interested in some or all of the items. More precisely, each agent has a non-negative valuation for each item; if an agent is not interested in an item at all, he values it at 0. Moreover, each agent has a *budget*, which is the total amount of money available to him. The items are auctioned off consecutively. For each item, we run a second-price auction [5], i.e., a sealed-bid auction in which all agents submit their bids, the agent who submitted the highest bid wins and pays the second highest bid. When there is only one object for sale, this auction format is *truthful*, i.e., it is a dominant strategy for each bidder to bid his true value for the object. In our scenario, this is not necessarily the case. However, it is a fairly common approach [4] to assume that the bidders are *myopic*, i.e., they bid truthfully while trying to remain within their budget. More formally, if item i is auctioned off in the jth auction, the agent k bids $\min\{v_{ik}, b_k^{(j-1)}\}$, where v_{ik} is his valuation for this item and $b_k^{(j-1)}$ is what remains of his budget after the first $j-1$ auctions.

Clearly, the overall outcome depends on the order in which the objects are being sold. We may want to know if for given valuations and budgets, there is an order that achieves certain allocation, or a certain total profit. To reduce the search space, we observe that if for two objects the sets of bidders interested in these objects (i.e., the bidders who have strictly positive valuations for them) are disjoint, the order in which these objects are auctioned off is irrelevant. Moreover, the respective state space is clearly acyclic, as after each round the number of unallocated objects decreases.

Voting We are given n voters and m candidates. Each voter has a preference ordering over some of the candidates, which is given by a total transitive irreflexive relation on this subset of candidates. We assume that all candidates not included in this preference ordering are considered to be strictly worse than the ones that are included; moreover, all these candidates are equally bad from this voter's point of view. The goal is to select a committee that consist of r members, $r < m$. To do this, we use the classical Single Transferable Vote (STV) method. Namely, each voter submits a ballot in which he lists the candidates from his preference ordering, best to worst. We count the number of first-place votes received by each candidate. We then select a candidate with the smallest number of first-place votes and cross him out from all ballots. Now all ballots that ranked this candidate first are transferred to the candidates who are listed second on these ballots, i.e., some candidates gain a few first-place votes. We repeat this process till only r candidates remain.

As described, this process is non-deterministic: there may be several candidates with the smallest number of first-place votes, and we have to choose which one of them to eliminate. Different choices may lead to different election outcomes. Generally speaking, these choices are not independent: if there are two candidates c_1 and c_2 with the smallest number of first-place votes, after we eliminate c_1, candidate c_2 may gain some votes and it will no longer be among the candidates with the smallest number of first-place votes. However, if no voter includes both c_1 and c_2 in his ballot, eliminating c_1 adds no votes to c_2 and vice versa. This may happen, for example, if all voters and candidates belong to some parties, and each voter only ranks the candidates from his party. In this case, eliminating c_1 and eliminating c_2 are two independent actions.

We may want to know whether for any choice of candidates to be eliminated at each step, a certain candidate or a group of candidates is (or is not) elected. Again, the system is acyclic, since after each round the number of surviving candidates decreases.

Multi-item auctions Suppose than we have m items $I = \{i_1, \dots, i_m\}$ for sale and there are n agents $a_1, \dots, a_n$, each of which is interested in a subset $I_i \subset I$ of these items. An agent gains a value v_i if he acquires all items in his desired bundle and 0 otherwise.

To sell the objects, we run m simultaneous ascending (English) auctions [5]. In any such auction, the bidding starts at 0, and any of the participants is allowed to bid an increment δ above the current price to become the provisional winner. The bidding stops when noone wants to bid above the current price. The participant who is the provisional winner at the end of the auction receives the object and pays his bid.

The bidding takes place in *rounds*. During each round, a single agent is allowed to bid on a single object. We restrict the agents' behavior as follows: an agent is only allowed to bid on the objects in his desired bundle, and his total bid (i.e., the total price of the items for which he is the provisional winner) should not exceed his value. Also, if the agent is the provisional winner for all object in his bundle, he is allowed not to bid; otherwise, he has to bid on one of the items in his bundle, or, if he cannot afford to, drop out of the game.

We may want to check that all trajectories of this system satisfy certain properties. For example, given all agents bundles and values, we may want to verify that it is never the case that agent 1 gets 3 items, while agent 2 gets no items whatsoever.

Note that if the bundles of two agents do not intersect, their actions are independent. Moreover, each transition increases the sum of all bids, so the state system (to be described in more detail) is acyclic.

Game of Life John Conway's Game of Life is the best-known example of a cellular automaton. The game takes place on a two-dimensional grid. In the beginning of the game, a subset of cells is *alive*, while other cells are *dead*. The game proceeds in stages. During each stage, each cell may change is state from dead to alive or vice versa depending on the states of its eight neighbors.

We consider a modified version of this game, in which during each stage exactly one cell changes its state according to the rules of the original game. Also, we assume that the game takes place on a finite board; we extend the rules for the cells on the boundary in some reasonable way. Clearly, if two cells are not neighbors, their actions are independent. Given an initial configuration, we may want to know if a certain configuration ever appears on the board. Unlike in the previous three cases, this state system is not necessarily acyclic, hence the TNF_DFS algorithm may not explore every state.

7 Experiments

We implemented the algorithms `EdgeLeanDfs` and `TNF_Dfs` from Sections 3 and 4 in the tool Spin [10]. 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. 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.

Although the algorithm `TNF_Dfs` might not explore the whole state spaces (every of our examples contains cycles), 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 considerably fewer transitions than regular depth-first search. On the other hand, the difference between `EdgeLeanDfs` and `TNF_Dfs` regarding the number of transistions is not very significant. With respect to the stack size (and thus memory consumption), our algorithms are up to a thousand times better than regular DFS (DMS examples). Also, on many examples TNF_DFS uses much less space than EdgeLean

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

(see DMSwithCC, RW and the petersonN examples), while EdgeLean barely comes on top in pftp, leader and sort. It is quite surprising that both algorithms explore roughly the same number of transitions, but TNF_DFS needs a stack much smaller than the one needed by EdgeLean_DFS (examples DMS, RW and petersonN), though it can be explained.

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. Cramton, Y. Shoham, R. Steinberg, *Combinatorial Auctions*, MIT Press, 2006
5. D. Fudenberg and J. Tirole, *Game Theory*, MIT Press, 1991.
6. 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.
7. P. Godefroid, P. Wolper, Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties, in *CAV 1991*, pp. 176–185, LNCS 575, 1991.
8. P. Godefroid, G. Holzmann, D. Pirottin, State-Space Caching Revisited, *Formal Methods in System Design 7:3*, pp. 227–242, 1995
9. K. Gouda, M.J Zaki, Efficiently Mining Maximal Feqent Itemsets, *IEEE International Conference on Data Mining (ICDM '01)*, pp. 163–170, 2001.
10. G. Holzmann, The SPIN Model Checking, Addison Wesley, 2003.
11. M. Koyuturk, A. Grama, W. Szpankowski, An Efficient Algorithm for Detecting Frequent Subgraphs in Biological Networks *Bioinformatics 20, suppl. 1*, pp.i200–i207, Oxford University Press, 2004.
12. A. Mazurkiewicz, Trace semantics, in *Advances in Petri Nets 1986*, LNCS 255, pp. 279–324, 1986.
13. E. Ochmanski, Languages and Automata, in *The Book of Traces*, V. Diekert, G. Rozenberg (eds.), 167–204, 1995.

14. D. Peled, Combining Partial Order Reductions with On-the-fly Model-Checking, in *CAV 1994*, LNCS 818, pp. 377-390, 1994.
15. A. Valmari: A Stubborn Attack on State Explosion, Formal Methods in System Design 1(4): 297-322, 1992.