Compositional Upper-Bounding of Diameters of Factored Digraphs

MOHAMMAD ABDULAZIZ, Technical Unviersity of Munich, Germany
CHARLES GRETTON, Australian National University, Australia
MICHAEL NORRISH, Data61, Australia

In AI planning and model checking a problem is described using a succinct representation of the underlying transition system. That system corresponds to a directed graph (*digraph*), where vertices and edges model states and transitions, respectively. Of interest is the default situation, where that digraph is so large it cannot be represented explicitly. A number of solution methods in this setting require that we first obtain or bound a topological property of the succinctly represented graph. Examples of properties of interest here are: (i) the diameter (the length of the longest *shortest* path), and (ii) the recurrence diameter (the length of the longest *simple* path). We develop and study *compositional* approaches to compute upper bounds on such topological properties. Compositional approaches proceed by treating small abstractions of the concrete system. Properties are calculated exactly for very small abstractions of the given system, and then composed to obtain a bound on the value of the property for the concrete system.

We present new compositional algorithms and related abstraction techniques. These are implemented, and those implementations form the basis of a detailed empirical study. We also develop theoretical results that capture exactly the limitations and tightness of different compositional bounding approaches.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; **Planning and scheduling**; • **Theory of computation** → *Verification by model checking*; Dynamic graph algorithms; • **Mathematics of computing** → Graph theory.

Additional Key Words and Phrases: AI planning, Succinct Graphs, Bounded Model Checking, Algorithms

ACM Reference Format:
Mohammad Abdulaziz, Charles Gretton, and Michael Norrish. 2019. Compositional Upper-Bounding of Diameters of Factored Digraphs. 1, 1 (May 2019), 42 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Due to the appeal of their compactness, succinct representations of directed graphs (*digraphs*) are studied in both practical [33] and theoretical [35] contexts. Their compactness is of great utility in applications like Artificial Intelligence (AI) planning and model checking, which are the motivating applications for this work. In those applications, state spaces are modelled as digraphs where vertices and edges represent states and transitions, respectively. One natural way of handling the enormous size of realistic state spaces in such applications is to represent them succinctly in a factored way, in languages such as STRIPS [33] and SMV [51]. In a factored representation, every state (i.e. vertex) is defined as an assignment to a set of Boolean state variables, and multiple transitions (i.e. edges) are factored into actions that identify which states are connected to which other states. Also, like other succinct representations, factored representations can be exponentially smaller than explicitly representing the digraph. An example transition system is a 3-bit *binary counter system*. A factored description of this counter system would be specified in terms of state variables b_0, b_1, b_2 and would have the actions π_0, π_1, π_2, π_{reset} defined as follows:

- π_0: set b_0 to 1, if $b_0 = 0$,
- π_1: set b_1 to 1, if $b_0 = 1$ and $b_1 = 0$,
- π_2: set b_2 to 1, if $b_0 = 1$, $b_1 = 1$, and $b_2 = 0$, and
- π_{reset}: set b_2 to 0, set b_1 to 0, and set b_0 to 0.

Our factored presentation has just 4 actions, whereas an explicit description of the 3-bit counting system would consist of $2^3 - 1$ transitions.

Fundamental properties of digraphs, and thus state spaces, are the *diameter* and the *recurrence diameter*. The diameter is the length of the longest path in the set of shortest paths between pairs of vertices in a digraph. For example, the diameter of the state space of the counting system is $2^3 - 1$. The recurrence diameter, conceived by Biere et al. [15], is defined in terms of *simple* paths—i.e. paths that do not encounter the same state twice. The recurrence

Authors' addresses: Mohammad Abdulaziz, Technical Unviersity of Munich, Munich, Germany; Charles Gretton, Australian National University, Canberra, Australia; Michael Norrish, Data61, Canberra, Australia.

XXXX-XXXX/2019/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

diameter of a digraph is the length of the longest simple path in the digraph, and it is an upper-bound on diameter. For the counting system, the recurrence diameter of its state space is $2^3 - 1$. However, the recurrence diameter can be exponentially larger than the diameter. For instance, if every two configurations of b_1, b_2, and b_3 were reachable from each other by one action, i.e. the state space was a clique, the diameter would be one, while the recurrence diameter would still be $2^3 - 1$.

Knowing an upper bound on the diameter, or the recurrence diameter of the state space of a factored system comes up as a necessary ingredient for many practical applications and algorithms in AI planning and model checking. Prominent examples are satisfiability (SAT) based planning [40] and the bounded model checking (BMC) algorithm [15]. For completeness, those algorithms need a *completeness threshold*, which is an upper bound on the length of action sequences those algorithms need to consider. Completeness thresholds are specified in terms of the diameter or the recurrence diameter of the state space. Accordingly, upper-bounding state space diameters and recurrence diameters was studied earlier [15, 21, 27, 45, 46]. The practical incompleteness of bounded model checking and of planning as satisfiability checking, due to the absence of tight completeness thresholds, was noted by Clarke et al. [27] as a significant and challenging problem to overcome. That is what we address here.

A challenge with succinct representations of digraphs (including factored transition systems) is the exponentiation of the complexity of solving problems on succinct digraphs relative to solving the same problems on explicitly represented digraphs [54]. A prominent occurrence of this complexity magnification is the s-t reachability problem, which is PSPACE-complete for succinctly represented digraphs [22, 32, 48], while it is in P for explicit digraphs. More relevant to this work, for a succinct digraph, computing diameter is Π_2^P-hard [38] and computing recurrence diameter is NEXP-hard [54] for succinctly represented digraphs, versus being in P [8, 24, 34, 64] and NP-hard [55], respectively, for explicit representations.

Practically, to compute exact values for topological properties like the diameter or recurrence diameter of a factored transition system, complexity exponentiation means that one must first construct the explicit digraph modelling the state space of the system, and then compute the property on the explicit representation. But this is impossible in practice: factored representations of state spaces with 2^{1000} states are not uncommon, an issue referred to as the *state space explosion* problem [26]. Furthermore, even if digraph construction were possible, state-of-the-art algorithms to compute or reasonably approximate the diameter [1, 7, 25, 58, 58] or the recurrence diameter [17] have infeasible worst-case run-times.

The *compositional* approach mitigates state space explosion: a solution is approximated for a system via processing multiple (significantly) smaller derived abstractions of the system [13, 23, 44]. This way, the compositional approach avoids explicitly constructing and processing an explicit state space of the concrete system. Instead, only abstract state spaces, which are usually (exponentially) smaller than the concrete state space, need to be constructed. This makes the compositional approach a feasible approach in many practical cases.

A well-studied method to compute abstractions is to *project* the given system on a set of state variables *vs*, where state variables other than those in *vs* are "removed" from the concrete system. For example, consider a system specified in terms of state variables $b_0, b_1, b_2, b_3, b_4, b_5, b_6$ and whose dynamics are modelled by the actions π_0, π_1, π_2, and π_{reset} from above and the actions π_3, π_4, π_5, and π_6 described below:

- π_3: set b_3 to 1, if $b_3 = 0$, $b_0 = 1$, $b_1 = 1$, and $b_2 = 1$,
- π_4: set b_4 to 1, if $b_3 = 1$, $b_4 = 0$, $b_0 = 1$, $b_1 = 1$, and $b_2 = 1$,
- π_5: set b_5 to 1, if $b_5 = 0$, $b_0 = 0$, $b_1 = 1$, and $b_2 = 1$, and
- π_6: set b_6 to 1, if $b_5 = 1$, $b_6 = 0$, $b_0 = 0$, $b_1 = 1$, and $b_2 = 1$.

To reason about this system compositionally we can decompose it into three *projections*. The first is a projection of the system on variables b_0, b_1, and b_2, producing a counter system with actions π_0, π_1, π_2, and π_{reset}, and state variables b_0, b_1, and b_2. The second is a projection on variables b_3 and b_4, producing a counter with actions π_3 and π_4, and state variables b_3 and b_4. The third is a projection on variables b_5 and b_6, producing a counter with actions π_5 and π_6, and state variables b_5 and b_6. Projection is equivalent to repetitive vertex contraction operations in the state space: the size of the state space is halved for every state variable removed. Thus, performing computations on projections can be exponentially easier than on the concrete system.

One might hope that a bound on the entire composite counter system diameter could be obtained by computing the diameter of each of its three projections. However, a problem with most abstraction techniques, including projection, is that they cause information loss. Thus most techniques will, at least for some systems, produce abstractions that do not suit compositional algorithms. Nonetheless, classes of systems for which a certain abstraction technique produces abstractions suitable for compositional reasoning can be characterised in terms of system structure.

A prominent system structure is that of subsystem dependencies, where a system is composed of dependent subsystems, like the composite counter system above: it is composed of a 3-bit counter and two independent 2-bit counters that both depend on the 3-bit counter. Subsystem dependency can be analysed using *state-variable dependencies*. Analysing the structure of state-variable dependencies has been frequently used to characterise systems

whose projections are suitable for compositional reasoning [4, 10, 12, 20, 37, 41, 42, 44, 57, 61, 63]. In the composite counting system, for instance, variables b_3, b_4, b_5, and b_6 depend on variables b_0, b_1, and b_2, but not the other way around. Dependencies between variables are described by a *dependency graph* and dependencies between sets of variables are described by a *lifted dependency graph*. Analysis of the structure of the lifted dependency graph is very successful in devising projection-based compositional algorithms.

In the context of diameter bounding, Baumgartner et al. [12], and independently later Rintanen and Gretton [57], introduced a method to compose projections' recurrence diameters to bound a concrete system diameter. That method is defined recursively bottom-up on the structure of acyclic lifted dependency graphs. Applied to the composite counter system, their method would firstly compute the recurrence diameters of each of the three projections: the 3-bit counter, and the two 2-bit counters, which would be $2^3 - 1$, $2^2 - 1$ and $2^2 - 1$, respectively. Their method then derives an upper bound on the composite system diameter of $2(2^2 - 1)(2^3 - 1) + 2(2^3 - 1)$.

1.1 Summary of Results

In this work, we provide the first comprehensive study of *compositional* techniques for computing upper bounds on *completeness thresholds*. We have the following contributions. We note that some of those results have appeared in preliminary form [3, 5, 6].

1.1.1 The Top-Down Algorithm. We identify a new algorithm to compose the projections' recurrence diameters into an upper bound on the concrete system diameter. That algorithm is similar to that of Baumgartner et al. in that it is recursively defined on the structure of an acyclic lifted dependency graph, except that the recursion in our algorithm is done top-down. We experimentally show that our top-down algorithm computes substantially smaller upper bounds compared to the bottom-up algorithm of Baumgartner et al. For instance, for the composite counting system from above, the top-down algorithm would compute the bound $2(2^2 - 1)(2^3 - 1) + 2^3 - 1$.

1.1.2 The Sublist Diameter. To prove the soundness of the top-down algorithm, we devised a new topological property which we call the *sublist diameter*. It is a generalisation of the recurrence diameter that takes factoring into consideration and is bound below by the diameter and above by the recurrence diameter. Interestingly, unlike the diameter or the recurrence diameter, the sublist diameter is bound above by projections' sublist diameters composed using our top-down method.

1.1.3 Tightness of the Top-Down Algorithm. It is of course informative to understand when bounding is possible. However, another important and related issue to consider, is whether or not the bound being provided is *tight*. Compositional algorithms use particular system features to infer a bound. In the example of the composite counter above, for instance, the bottom-up algorithm by Baumgartner et al. uses the projections' recurrence diameters and the dependencies between those projections as features to compute the bound $2(2^2 - 1)(2^3 - 1) + 2(2^3 - 1)$.

Characterising a bounding algorithm in terms of the features it uses to infer a bound, we call an algorithm tight iff there is no alternative algorithm that infers a tighter bound using only those same features. In particular we show that the top-down algorithm produces tight bounds compared to any algorithm that only uses projection recurrence diameters and the dependencies between projections as input features.

Consider again the composite counting system. The top-down algorithm applied to the three projections and their dependencies computes $2(2^2-1)(2^3-1)+2^3-1$ as an upper bound on the composite system diameter. An implication of the tightness of the top-down algorithm is that this computed bound is tight if we limit our features to (i) the projections' recurrence diameters, and (ii) the dependencies between projections. Note, however, that the diameter of the composite counter system is $2^4 - 3$, which is significantly less than the bound computed by the top-down algorithm. Nonetheless, we require extra information to make this inference. No general algorithm can provide a bound smaller than $2(2^2 - 1)(2^3 - 1) + 2^3 - 1$, in this case, without access to more than the contemplated features.

Although this notion of tightness may seem unnatural, we note that previously existing compositional approaches have not produced tight bounds in this sense. We therefore find it important and informative to know when the bounds produced by a compositional algorithm are tight, and to present algorithms that are tight. Our formulation of the tightness statements is a new way of stating optimality for bounding algorithms. The fact that those statements parameterise optimality on a certain input can be very helpful in clarifying what aspects of the algorithm do not need further improvements.

1.1.4 Limitations on Compositional Bounding of the Diameter and the Recurrence Diameter Using Dependency-Based Projections. We then address the question of whether it is at all possible to bound a system diameter using projection diameters. This question was instigated by the fact that all previous compositional methods bound a system diameter using projection recurrence diameters. We provide a negative answer to that question: for every method that composes projection diameters, there is not a dependency structure between the projections which can, in and of itself, guarantee that the composed projection diameters are an upper bound on the system diameter. In particular, an algorithm whose input features are limited to (i) the projections' diameters, and (ii) the dependencies between projections,

cannot in general compute a sound upper bound on a concrete system diameter. We also show a similar result for the recurrence diameter.

1.1.5 Exploiting Dependency-Based Projections with Acyclic State Spaces. Dependency structures alone are not the only means to obtain useful bounds for systems with factored representations. In practice we encounter systems, or system projections, in which there are no paths which encounter the same state two or more times, i.e. systems with acyclic state spaces. We call these *acyclic* systems. For example, the 3-bit counter from above is acyclic if we remove the action π_{reset}.

Now, consider a system that has two projections, where one of them is acyclic and where dependencies run in one direction: from the acyclic projection to the other projection. In that case, the diameter of the concrete system is bound above by the sum of the recurrence diameters of the two projections instead of being bound by their product, as would be the case if the two projection state spaces were not acyclic. Replacing products by additions can potentially lead to much tighter bounds. To exploit that, we devise a modified version of the top-down algorithm that, in addition to the projection recurrence diameters and their dependencies, takes one bit of information per projection, indicating whether the projection state space is acyclic. For instance, this modified top-down algorithm computes $2(2^2 - 1) + 2^3 - 1$ as a bound on the composite counter system's diameter, if the action π_{reset} is removed.

We show that this modified top-down algorithm computes sound upper bounds on the diameter, and we also show that it is tight w.r.t. the features it takes as input. We also provide experimental evidence that the modified top-down algorithm computes better bounds than those computed by the original top-down algorithm in domains where acyclic projections are encountered.

1.1.6 Exploiting General Projections with Acyclic State Spaces. Experimentally, however, we observe that the modified top-down algorithm computes bounds smaller than the original top-down algorithm in only a limited set of benchmarks. This indicates that, although most benchmarks have acyclic projections, it is uncommon to encounter acyclic projections in the desired one-way dependency configuration just discussed. Furthermore, the modified top-down algorithm does not improve on the original top-down algorithm in terms of the ability to decompose a system, measured by how small the abstractions are. This is because both the modified and original top-down algorithms use the same projections as abstractions.

To further the exploitation of state space acyclicity for better bounds and smaller abstractions, we exploit acyclic projections identified by standard prepossessing workstreams, which do not use dependency analysis. With those acyclic projections we can bound compositionally as follows. For each state of a given acyclic projection, we calculate a version of the concrete system constrained so that the projection is fixed statically according to that state. That constrained system is an abstraction of the concrete system and we call it a *snapshot*. In the example of the composite counter system, if action π_{reset} is removed, an acyclic projection of the system will be a 3-bit counter. One state in that acyclic projection is the state where $b_0 = 0$, $b_1 = 0$, and $b_2 = 0$. The snapshot of the composite counter on that state only has the action π_0, because the preconditions or effects of other actions violate the given assignments of b_0, b_1, and b_2.

The concrete system's (recurrence) diameter cannot exceed the sum of the (recurrence) diameters of all system snapshots. More generally, we devise a compositional algorithm that bounds the (recurrence) diameter by composing the snapshots' (recurrence) diameters. We show that, for the recurrence diameter, this bounding algorithm is tight in case the only features available are: (i) the system snapshots, and (ii) a digraph representing the state space of the acyclic projection from which the snapshots are derived.

Experimentally, that algorithm successfully decomposes systems into abstractions smaller than those produced by the top-down algorithm. This comes at the cost of potentially exponentially worse run-times than those encountered with either one of the top-down algorithms.

1.1.7 Practical Combination of State Space Ayclicity and Dependency Acyclicity. Our final contribution is a novel algorithm which pragmatically uses ideas from compositional bounding with dependencies and acyclic projections together. Our motivation for that algorithm is that a hybrid approach can improve bound computation by: (i) being able to use smaller abstractions than what is possible by compositional reasoning solely based on dependency analysis or state space acyclicity, and (ii) avoiding the intractable run-time incurred by state space acyclicity reasoning. We provide a highly detailed empirical evaluation of the hybrid algorithm showing that it significantly outperforms existing approaches in terms of the tightness of the bounds obtained on standard AI planning benchmarks, and the size of abstractions it reasons about. We also demonstrate its utility in accelerating SAT-based planning.

2 RELATED WORK

Computing the diameter exactly can be done via solving the all pairs shortest path (APSP) problem for the (di)graph at hand. For an explicitly represented digraph, APSP cannot be solved in better than quadratic time (in the number of vertices of the (di)graph), and existing exact solutions have a run-time close to cubic [8, 24, 34, 64]. Furthermore,

there is strong evidence that the diameter cannot be computed in time better than quadratic [58]. This run-time can be very limiting in digraphs arising in practical applications, like AI planning and model checking, due to their sizes.

In the algorithms community, a lot of work has been done on developing methods to approximate the diameter. In a seminal paper, Aingworth et al. [7] devised an algorithm that computes a $\frac{3}{2}$-approximation of the diameter for digraphs in $O(m\sqrt{n \log n} + n^2 \log n)$ time, where n is the number of vertices and m is the number of edges. Examples of other work studying approximation algorithms for digraph diameters include [1, 25, 58].

Computing the recurrence diameter is an NP-hard problem [55] for explicitly represented digraphs, with the only known exact solutions taking exponential time. Accordingly it is much harder to compute than the diameter, especially for practical digraphs. Furthermore, the hardness of computing the recurrence diameter in digraphs was reaffirmed by Björklund et al. [17], who showed that, under plausible assumptions, it is impossible in general to get a polynomial approximation of the length of the longest path in polynomial time. Existing polynomial time approximation algorithms for the longest path can only find paths of lengths logarithmic in the length of the longest path [9, 16].

In the graph theory and combinatorics communities, diameters of undirected graphs have been extensively studied since 1965. Much work was done to compute upper bounds for different classes of undirected graphs [31, 43, 52]. However, for digraphs, work on upper-bounding diameters started more recently. In 1992, Soares [62] provided a tight upper bound on the diameter of biregular digraphs. Later, Dankelmann et al. treated different structures of digraphs. [28–30].

The completeness of satisfiability (SAT) based planning [40] and bounded model checking [14, 15] depends on having a completeness threshold. Depending on the problem at hand, the completeness threshold is required to be an upper bound on either the diameter (for planning and model checking safety properties) or the recurrence diameter (for model checking liveness properties), and the tighter that bound, the more quickly the algorithm will likely terminate. Because of that, studying the diameter and the recurrence diameter is an active research topic in the verification community [14, 15, 21, 27, 45, 46, 60]. The dominant approach for computing the diameter or the recurrence diameter in verification applications is via encodings in SAT [15, 46]. Most notably, Biere *et al.* [15] conjecture that for the question of "whether for a certain digraph, a number N is its diameter", there is not a SAT encoding of size polynomial in N. However, if the question is "whether N is the recurrence diameter", they provide a SAT encoding of size $O(N^2)$, which was improved by Kröning [46] to $O(N \log N)$.

Applying the compositional approach for diameter upper-bounding was pioneered by Baumgartner et al. [12] in the context of bounded model checking. That work showed that the diameter of a system can be upper-bounded using a polynomial expression in the recurrence diameters of abstract subsystems.Recently Rintanen and Gretton applied a similar approach to upper-bound the diameters for AI planning [57] problems.More recently, the first author described a new topological property, the *traversal diameter* [2]. The traversal diameter is one less than the largest number of distinct states that can be encountered by a single execution. This is an upper bound on the recurrence diameter, and a lower bound on the number of states in the state-space. A key practical advantage of the traversal diameter compared to the concepts of diameter and recurrence diameter, is that it is linear time to compute in the size of the graph. Also, the traversal diameter is susceptible to compositional bounding in the case of a concrete system composed of co-dependant subsystems. It has been combined with some techniques we present here to further tighten the bounds one can obtain in number of benchmark problems.

3 CONCEPTS AND NOTATIONS

Definition 1 (Digraph). *An α-graph, G_α, is a digraph whose vertices are labelled by values of type α. That labelled graph G_α is given by a tuple $\langle V, E, \mathcal{L}\rangle$, where: V is a set of vertices which we denote as $V(G_\alpha)$, E is a set of edges, that are distinct ordered pairs of* vertices *from $V(G_\alpha)$, which we denote as $E(G_\alpha)$, and $\mathcal{L} : V(G_\alpha) \Rightarrow \alpha$ is a labelling function, where a vertex $u \in V(G_\alpha)$ has the label $\mathcal{L}(u)$ of type α. We find it useful to treat graphs, say G_α and G_β, that are only different in terms of the vertex labels. Here, when discussing the labelling of a vertex u in G_α and G_β, we write $G_\alpha(u)$ and $G_\beta(u)$, respectively. When we illustrate a digraph in a figure, a vertex u is docketed with the label $\mathcal{L}(u)$ and not the vertex name u, and we also omit self loops. For $u_1 \in V(G_\alpha)$, the set of children of u_1 is $\mathsf{children}_{G_\alpha}(u_1) = \{u_2 \mid (u_1, u_2) \in E(G_\alpha)\}$. If $\mathcal{L}$ is injective, we use $l \in G_\alpha$ to denote that l is a label of some $u \in V(G_\alpha)$, i.e. $l = \mathcal{L}(u)$, and we use $\mathsf{children}_{G_\alpha}(l)$ to refer to the labels of the children of u, i.e. $\{G_\alpha(u_2) \mid u_2 \in \mathsf{children}_{G_\alpha}(u)\}$. For $G_\alpha \equiv \langle V, E, \mathcal{L}\rangle$, the image of a function $g : \alpha \Rightarrow \beta$ on G_α is the β-graph $g(\!|G_\alpha|\!) \equiv \langle V, E, g \circ \mathcal{L}\rangle$, where $g \circ \mathcal{L}$ is the composition of $\mathcal{L}$ and g. Effectively, the image operation on a digraph is a relabelling operation.*

With the above definition in hand, we can proceed to the topic of factored descriptions of digraphs that model transition systems. The edges of such a graph are described compactly by a set of *actions*, and the vertices correspond to *states*, each of which maps to a unique truth assignment over the set of state characterising propositions. In this formalism, edges and paths between vertices correspond to executing actions and action sequences, respectively,

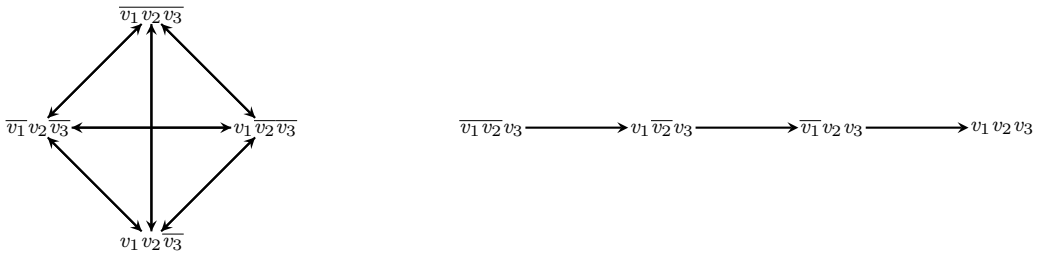


Fig. 1. The state space of δ in Example 1.

between states. We depart slightly from conventional expositions in the related literature, as we do not treat *initial conditions*, i.e. the state a system begins/starts in, nor do we treat *goal/safety* criteria, i.e. intuitively the description of the "system states we are searching for". The bounds and the different topological properties that we study here are independent of those features, and so these aspects are peripheral to our work. Except for this small departure, the system representation we give here is equivalent to representations commonly used in the model checking and AI planning communities, such as STRIPS [33] and SMV [51].

Definition 2 (States and Actions). *A maplet, $v \mapsto b$, maps a variable v—i.e. a state-characterising proposition—to a Boolean value b. A state, x, is a finite set of maplets. We write $\mathcal{D}(x)$ to denote $\{v \mid (v \mapsto b) \in x\}$, the domain of x. For states x_1 and x_2, the union, $x_1 \uplus x_2$, is defined as $\{v \mapsto b \mid v \in \mathcal{D}(x_1) \cup \mathcal{D}(x_2) \wedge \mathsf{if}\ v \in \mathcal{D}(x_1)\ \mathsf{then}\ b = x_1(v)\ \mathsf{else}\ b = x_2(v)\}$. Note that the state x_1 takes precedence. An action is a pair of states, (p, e), where p represents the* preconditions *and e represents the* effects. *For action $\pi = (p, e)$, the domain of that action is $\mathcal{D}(\pi) \equiv \mathcal{D}(p) \cup \mathcal{D}(e)$.*

Definition 3 (Execution). *When an action π $(= (p, e))$ is executed at state x, it produces a successor state $\pi(x)$, formally defined as $\pi(x) = \mathsf{if}\ p \subseteq x\ \mathsf{then}\ e \uplus x\ \mathsf{else}\ x$. We lift execution to lists of actions $\overrightarrow{\pi}$, so $\overrightarrow{\pi}(x)$ denotes the state resulting from successively applying each action from $\overrightarrow{\pi}$ in turn, starting at x, which corresponds to a path in the state space.*

We give examples of states and actions using sets of literals. For example, $\{v_1, \overline{v_2}\}$ is a state where state variables v_1 is (maps to) true, and v_2 is false and its domain is $\{v_1, v_2\}$. Then $(\{v_1, \overline{v_2}\}, \{v_3\})$ is an action that if executed in a state where v_1 and $\overline{v_2}$ hold, it sets v_3 to true. The domain of this action, $\mathcal{D}((\{v_1, \overline{v_2}\}, \{v_3\}))$ is equal to $\{v_1, v_2, v_3\}$.

Definition 4 (Factored System). *For a set of actions δ we write $\mathcal{D}(\delta)$ for the domain of δ, which is the union of the domains of all the actions in δ. The set of valid states, written $\mathbb{U}(\delta)$, is $\{x \mid \mathcal{D}(x) = \mathcal{D}(\delta)\}$. The set of valid action sequences is the Kleene closure of δ, i.e. $\delta^* = \{\overrightarrow{\pi} \mid \mathsf{set}(\overrightarrow{\pi}) \subseteq \delta\}$, where $\mathsf{set}(l)$ is the set of members in list l.*

The set of actions δ is a factored representation of the digraph $G(\delta) \equiv \langle V, E, \mathcal{L}\rangle$, where $V = \mathbb{U}(\delta)$, the labelling function $\mathcal{L}$ is the identity function, the set of edges in the graph is $E \equiv \{(x, \pi(x)) \mid x \in \mathbb{U}(\delta), \pi \in \delta\}$. We refer to $G(\delta)$ as the state space of δ. For states x and x', $x \rightsquigarrow x'$ denotes that there is some $\overrightarrow{\pi} \in \delta^*$ such that $\overrightarrow{\pi}(x) = x'$.

Example 1. Consider the following factored representation, δ.

$$\left\{ \begin{array}{l} p_1 = (\{\overline{v_1}, \overline{v_2}, v_3\}, \{v_1\}), p_2 = (\{v_1, \overline{v_2}, v_3\}, \{\overline{v_1}, v_2\}), p_3 = (\{\overline{v_1}, v_2, v_3\}, \{v_1\}), \\ k_1 = (\{\overline{v_3}\}, \{v_1, v_2\}), k_2 = (\{\overline{v_3}\}, \{\overline{v_1}, v_2\}), k_3 = (\{\overline{v_3}\}, \{v_1, \overline{v_2}\}), k_4 = (\{\overline{v_3}\}, \{\overline{v_1}, \overline{v_2}\}) \end{array} \right\}$$

The digraph in Figure 1 represents the state space of δ, where different states defined on the variables $\mathcal{D}(\delta) = \{v_1, v_2\, v_3\}$ are shown, along with their connectivity. This system has two "modes" of operation, where the assignment to the toggle variable v_3 determines the mode of operation. Each of those modes is represented by one connected component of the digraph in Figure 1. Actions k_1, k_2, k_3 and k_4, which execute successfully only if v_3 maps to false, represent edges in the clique component. For instance, action k_1 represents all incoming edges to the vertex that corresponds to state $\{v_1, v_2\, \overline{v_3}\}$. Actions p_1, p_2 and p_3, which execute successfully only if v_3 maps to true, each represent an edge in the simple path component.

Treating system dynamics only, factored transition systems are equivalent to the STRIPS representation [33] of AI planning problems with the possibility of having negations in the preconditions, and to the SMV formalism [51] of model checking problems. To put them into context w.r.t. other succinct graph representations, factored transition systems effectively represent digraphs as propositional formulae in disjunctive normal form (DNF). Accordingly, factored systems can be exponentially smaller than equivalent OBDDs [32] and exponentially larger than equivalent Boolean circuits [35]. For example, a factored system comprised of the two actions $(\{\overline{c}\}, \{\overline{a}\})$ and $(\{a, \overline{b}\}, \{c\})$ is equivalent to the DNF formula $(\overline{c_1} \wedge \overline{a_2}) \vee (a_1 \wedge \overline{b_1} \wedge c_2)$ that takes as input variables $\{a_1, b_1, c_1\}$ and $\{a_2, b_2, c_2\}$ representing two vertices. Note: action preconditions and effects are restricted to conjunctions of literals so that

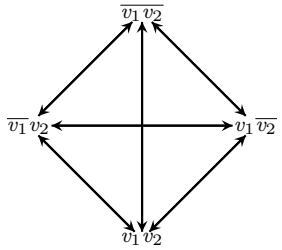


Fig. 2. Taking δ from Example 1, we depict the contraction of the state space of $\delta\downarrow_{\{v_1,v_2\}}$, the projection of δ on $\{v_1, v_2\}$.

representing planning problems as factored systems solves the *frame problem* [47, 49]. An additional advantage of factored systems is that they enable computing state-variable dependencies in linear time, which is critical for the efficiency of the majority of compositional algorithms.

3.1 Topological Properties: Bounding Concepts

An optimal path from a starting state to an ending state is a minimum-length action sequence that causes the system to transition from that starting state to that ending state. The diameter of a factored system is the length of the longest optimal path between any two valid states.

Definition 5 (Diameter). *Let $|l|$ denote the length of a list l. Formally, for a system δ, the diameter $d(\delta)$ is:*

$$d(\delta) = \max_{x \in \mathbb{U}(\delta)} \max_{\vec{\pi} \in \delta^*} \min\{|\vec{\pi}'| \mid \vec{\pi}' \in \delta^* \wedge \vec{\pi}(x) = \vec{\pi}'(x)\}$$

Whereas standard expositions define the diameter by quantifying over pairs of states, we choose to quantify over states and executions from those states. Our approach avoids complexities arising from situations where states are not connected.

The recurrence diameter of a factored system is the length of the longest simple path between any two valid states.

Definition 6 (Recurrence Diameter). *Let $\mathsf{distinct}(x, \vec{\pi})$ be a predicate that is true iff the states encountered executing $\vec{\pi}$ from x, inclusive of x, are all different. Formally, for a system δ, the recurrence diameter $rd(\delta)$ is:*

$$rd(\delta) = \max_{x \in \mathbb{U}(\delta)} \max_{\vec{\pi} \in \delta^*} \max\{|\vec{\pi}| \mid \mathsf{distinct}(x, \vec{\pi})\}$$

It should be clear that if there is a valid action sequence between any two states, then there is a valid action sequence between them that is not longer than the diameter. Also, it should be clear that the recurrence diameter is an upper-bound on the diameter.

3.2 Projection and State-Variable Dependency

A key concept for compositional reasoning about factored representations of digraphs is *projection*. Projection of a transition system is equivalent to a sequence of vertex contractions in the digraph modelling the state space.

Definition 7 (Projection). *Projecting an object (a state x, an action π, a sequence of actions $\vec{\pi}$ or a factored representation δ) on a set of variables vs restricts the domain of the object or the components of composite objects to vs. Projection is denoted as $x\downarrow_{vs}$, $\pi\downarrow_{vs}$, $\vec{\pi}\downarrow_{vs}$ and $\delta\downarrow_{vs}$ for a state, action, action sequence and factored representation, respectively. In the case of action sequences, an action with no effects after projection is dropped entirely.*

The projection of a system to a subset of its variables is an abstraction of the concrete system at hand. In and of itself, the concept of an abstraction obtained using projection is not all that useful for bounding. However, we shall soon see that by analysing projections that are characterised by system dependency structures, we can achieve useful bounds compositionally.

Also, we note that the digraph corresponding to the projection's state space is a contraction of the digraph corresponding to concrete system's state space. We do not use this property explicitly in the sequel, however, we believe it is something worth bringing to the attention of the reader.

Example 2. Letting $vs = \{v_1, v_2\}$, below is the projection $\delta\downarrow_{vs}$, for δ from Example 1. Figure 2 shows $G(\delta\downarrow_{vs})$.

$$\left\{ \begin{array}{l} p_1\downarrow_{vs} = (\{\overline{v_1}, \overline{v_2}\}, \{v_1\}), p_2\downarrow_{vs} = (\{v_1, \overline{v_2}\}, \{\overline{v_1}, v_2\}), p_3\downarrow_{vs} = (\{\overline{v_1}, v_2\}, \{v_1\}), \\ k_1\downarrow_{vs} = (\emptyset, \{v_1, v_2\}), k_2\downarrow_{vs} = (\emptyset, \{\overline{v_1}, v_2\}), k_3\downarrow_{vs} = (\emptyset, \{v_1, \overline{v_2}\}), k_4\downarrow_{vs} = (\emptyset, \{\overline{v_1}, \overline{v_2}\}) \end{array} \right\}$$

State-variable dependency analysis captures many structures present in factored systems. Informally, a state variable v_1 depends on v_2 iff the assignment of v_2 at some state can possibly affect assignment of v_1 in a current or a future state.

Fig. 3. (a) the dependency graph of δ from Example 1, and (b) a lifted dependency graph of δ. Actions induce edges in (a): for example, v_1 and v_2 co-occur in action effects, while v_3 only happens to be in the precondition.

This relation lifts to sets of state variables, where a set vs_2 depends on vs_1 iff vs_2 has a variable that depends on some variable in vs_1. Dependencies between variables are described by a *dependency graph* and dependencies between sets of variables are described by a *lifted dependency graph*. In many cases, the soundness of compositional algorithms relies on restrictions on the structure of the dependency graph. Previous authors identified several practically abundant lifted dependency graph structures which guarantee the soundness of powerful compositional methods that use projections. One notable example is projections on sets of variables that are closed under mutual dependency (i.e. projections computed from acyclic lifted dependency graphs): they allow for compositional solution of reachability and upper-bounding, and are useful in solving problems in many practical systems. Compositional methods that compute projections via dependency analysis are described in a number of earlier papers [4, 10, 12, 20, 37, 41, 42, 44, 57, 61, 63].

Definition 8 (Dependency). *A variable v_2 is dependent on v_1 in δ (written $v_1 \rightarrow v_2$) iff one of the following statements holds:* [1] *(i) v_1 is the same as v_2, (ii) there is $(p, e) \in \delta$ such that $v_1 \in \mathcal{D}(p)$ and $v_2 \in \mathcal{D}(e)$, or (iii) there is a $(p, e) \in \delta$ such that both v_1 and v_2 are in $\mathcal{D}(e)$. A set of variables vs_2 is dependent on vs_1 in δ (written $vs_1 \rightarrow vs_2$) iff: (i) vs_1 and vs_2 are disjoint, and (ii) there are $v_1 \in vs_1$ and $v_2 \in vs_2$, where $v_1 \rightarrow v_2$.*

Definition 9 (Dependency Graph). *This graph, sometimes called the* causal graph*, was described independently by Knoblock [42] and then Williams an Nayak [63]. $G_{\mathcal{D}(\delta)}$ is a dependency graph of δ iff (i) its vertices are bijectively labelled by variables from $\mathcal{D}(\delta)$, and (ii) it has an edge from vertex u_1 to u_2 iff $v_1 \rightarrow v_2$, where v_1 and v_2 are the labels of u_1 and u_2, respectively.*

Definition 10 (Lifted Dependency Graph). *G_{vs} is a lifted dependency graph of δ iff (i) its vertices are bijectively labelled by members of a partition $vs_{1..n}$ of $\mathcal{D}(\delta)$, and (ii) it has an edge from vertex u_1 to u_2 (labelled by vs_1 and vs_2, respectively) iff $vs_1 \rightarrow vs_2$, for all $vs_1, vs_2 \in vs_{1..n}$.*

Example 3. Figure 3a and Figure 3b show the dependency graph and a lifted dependency graph of δ from Example 1.

4 BOUNDING THE DIAMETER COMPOSITIONALLY USING SUBSYSTEM BOUNDS

Compositional algorithms for computing bounds proceed by generating a polynomial expression which evaluates to the desired bound. We focus on compositional algorithms that produce polynomial expressions which evaluate to upper-bounds on the concrete system diameter. The indeterminates in that polynomial expression correspond to bounds on carefully identified projections. We refer to such algorithms as *polynomial generators*. The compositional algorithm described by Baumgartner et al. [12] is a case in point, where the indeterminates in the computed polynomials are the recurrence diameters of projections of the system at hand. We shall describe the algorithm from Baumgartner et al. below as the polynomial generator denoted $\mathsf{M}_{\mathsf{sum}}$. A similar approach to that of Baumgartner et al. was developed by Rintanen and Gretton [57], where the indeterminates in the computed polynomials are the cardinalities of the state spaces of the same family of projections. The common features in these approaches are:

(i) both algorithms are defined recursively "bottom-up" on acyclic lifted dependency graphs (or equivalently, design *netlists*)
(ii) both algorithms map dependency graphs to polynomial expressions, which are determined according to structure of the lifted dependency graph.

Before laying out the definition of the bottom-up algorithm we define the concepts of *leaves* and *ancestors*.

Definition 11 (Leaves). *We define the set of leaves* $\mathsf{leaves}(G_{\mathsf{vs}})$ *to contain those vertices of G_{vs} from which there are no outgoing edges.*

Definition 12 (Ancestors). *We write* $\mathsf{ancestors}(vs)$ *to denote the set of ancestor vertices of vs in G_{vs}. It is the set $\{vs_0 \mid vs_0 \in G_{\mathsf{vs}} \wedge vs \in G_{\mathsf{vs}} \wedge vs_0 \rightarrow^+ vs\}$, where $\rightarrow^+$ is the transitive closure of $\rightarrow$.*

The bottom-up bounding algorithm is defined as follows.

[1] Our definition is equivalent to those in [42, 63] in the context of AI planning.

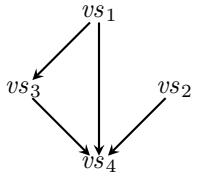


Fig. 4. A lifted dependency graph for a factored digraph which has four sets of variables closed under mutual dependency.

Definition 13 (Bottom-Up Acyclic Dependency Compositional Bound).

$$\mathsf{M}\langle b\rangle(vs,\delta,G_{\mathsf{VS}}) = b(\delta\downarrow_{vs}) + (1 + b(\delta\downarrow_{vs})) \sum_{a\in\mathsf{ancestors}(vs)} \mathsf{M}\langle b\rangle(\delta\downarrow_a)$$

Then, let $\mathsf{M}_{\mathsf{sum}}\langle b\rangle(\delta,G_{\mathsf{VS}}) = \sum_{vs\in\mathsf{leaves}(G_{\mathsf{VS}})} \mathsf{M}\langle b\rangle(vs,\delta,G_{\mathsf{VS}})$.

G_{VS} is a lifted dependency graph of δ used to identify abstract sub-problems, and δ is the system of interest. The functional parameter b is used to bound abstract sub-problems, and we refer to it as a base-case (bounding) function. Valid instantiations of b are the recurrence diameter [12] and the size of state space $2^{|\mathcal{D}(\delta)|}$ [57]. Also, $\mathsf{M}_{\mathsf{sum}}$ is recursively defined on the structure of G_{VS} and it is only well-defined if G_{VS} is a DAG. Baumgarnter et al. [12] showed that $\mathsf{M}_{\mathsf{sum}}\langle rd\rangle$ can be used to upper-bound the diameter, in case δ has an acyclic lifted dependency graph. A restatement of Theorem 1 from their paper is as follows.

Theorem 1. *For any factored representation δ with acyclic lifted dependency graph A_{VS}, $d(\delta) \leq \mathsf{M}_{\mathsf{sum}}\langle rd\rangle(\delta, A_{\mathsf{VS}})$.*

The following example illustrates the operation of $\mathsf{M}_{\mathsf{sum}}$.

Example 4. Figure 4 shows a lifted dependency DAG, A_{VS}, of some factored system δ. Since A_{VS} is a DAG, this implies that for $1 \leq i \leq 4$, the set of variables vs_i is closed under mutual dependency, and $\mathcal{D}(\delta) = \bigcup vs_i$. Given a base-case function b, and letting b_i denote $b(\delta\downarrow_{vs_i})$ and M_i denote $\mathsf{M}\langle b\rangle(vs_i,\delta,A_{\mathsf{VS}})$, we have

(i) $\mathsf{M}_1 = b_1$,
(ii) $\mathsf{M}_2 = b_2$,
(iii) $\mathsf{M}_3 = b_3 + \mathsf{M}_1 + b_3\mathsf{M}_1 = b_3 + b_1 + b_1b_3$,
(iv) $\mathsf{M}_4 = b_4 + (1 + b_4)(\mathsf{M}_1 + \mathsf{M}_2 + \mathsf{M}_3)$
$= 2b_1 + b_2 + b_3 + b_4 + b_1b_3 + 2b_1b_4 + b_2b_4 + b_3b_4 + b_1b_3b_4$.

Since vs_4 is the only leaf in the dependency graph, the polynomial computed by $\mathsf{M}_{\mathsf{sum}}$ will be M_4.

The previous example should make it clear that $\mathsf{M}_{\mathsf{sum}}$ can be viewed as a polynomial generating function *recursively* defined on a DAG. The structure of the terms (i.e. what is multiplied by what) of the polynomial returned by $\mathsf{M}_{\mathsf{sum}}$ depend *only* on the structure of A_{VS} (i.e. the number of vertices and their connectivity), regardless of δ or the values of b on different projections. However, $\mathsf{M}_{\mathsf{sum}}$ has a problem: it repeatedly adds the terms $\mathsf{M}\langle b\rangle(vs_i)$ as many times as there are children for vs_i. Except for the first $\mathsf{M}\langle b\rangle(vs_i)$ term it adds, all those terms are redundant as we show later. We also note that the function $\mathsf{M}_{\mathsf{sum}}$ is monotonic: for a bounding function b_1 that is an upper bound on another bounding function b_2, $\mathsf{M}_{\mathsf{sum}}\langle b_2\rangle(\delta,A_{\mathsf{VS}}) \leq \mathsf{M}_{\mathsf{sum}}\langle b_1\rangle(\delta,A_{\mathsf{VS}})$ holds.

4.1 Top-Down for Tight Bounds

Recall that the bottom-up algorithm computes redundant terms in the polynomials it generates, as shown in Example 4 In this section we define a new polynomial generator, with the motivation of avoiding the computation of those redundant terms computed by the bottom-up algorithm. The new polynomial generator, denoted $\mathsf{N}_{\mathsf{sum}}$, does the recursion in the opposite direction to previous approaches. In particular, $\mathsf{N}_{\mathsf{sum}}$ does the recursion "top-down" on acyclic lifted dependency graphs. Experimentally we find that $\mathsf{N}_{\mathsf{sum}}$ dominates $\mathsf{M}_{\mathsf{sum}}$ in terms of the tightness of bounds one can obtain in practice. We also explore the question of tightness theoretically, and find that the bounds obtained using $\mathsf{N}_{\mathsf{sum}}$ are the tightest one can hope for, given its input features.

Definition 14 (Top-Down Acyclic Dependency Compositional Bound).

$$\mathsf{N}\langle b\rangle(vs,\delta,G_{\mathsf{VS}}) = b(\delta\downarrow_{vs})(1 + \sum_{c\in\mathsf{children}_{G_{\mathsf{VS}}}(vs)} \mathsf{N}\langle b\rangle(c,\delta,G_{\mathsf{VS}}))$$

Then, let $\mathsf{N}_{\mathsf{sum}}\langle b\rangle(\delta,G_{\mathsf{VS}}) = \sum_{vs\in G_{\mathsf{VS}}} \mathsf{N}\langle b\rangle(vs,\delta,G_{\mathsf{VS}})$.

$\mathsf{N}_{\mathsf{sum}}$ is recursively defined on the structure of G_{VS}, and accordingly it is only well-defined if G_{VS} is a DAG. We note that $\mathsf{N}_{\mathsf{sum}}$ is monotonic: for a bounding function b_1 that is an upper bound on another bounding function b_2,

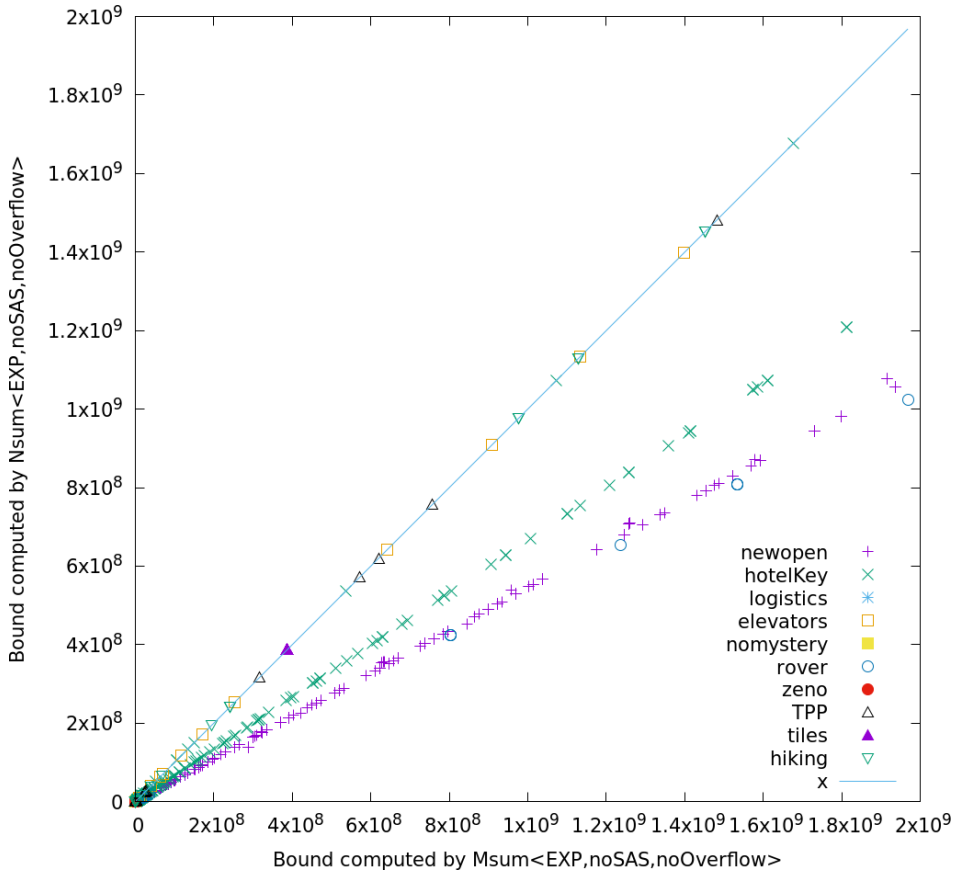


Fig. 5. The bounds computed by $\mathsf{N}_{\mathsf{sum}}\langle b\rangle$ versus $\mathsf{M}_{\mathsf{sum}}\langle b\rangle$ with $2^{|\mathcal{D}(\delta)|}-1$ as a base function.

$\mathsf{N}_{\mathsf{sum}}\langle b_2\rangle(\delta, A_{\mathsf{VS}}) \leq \mathsf{N}_{\mathsf{sum}}\langle b_1\rangle(\delta, A_{\mathsf{VS}})$ holds, for an acyclic dependency graph A_{VS}. The soundness of $\mathsf{N}_{\mathsf{sum}}$ as bound is formally stated in the following statement.

Theorem 2. *For any factored representation δ with an acyclic lifted dependency graph A_{VS}, $d(\delta) \leq \mathsf{N}_{\mathsf{sum}}\langle rd\rangle(\delta, A_{\mathsf{VS}})$.*

We postpone the proof of this theorem to the next section. However, we now compare the bounds computed by $\mathsf{N}_{\mathsf{sum}}\langle b\rangle$ with the ones computed using $\mathsf{M}_{\mathsf{sum}}\langle b\rangle$.

Example 5. Again we refer to A_{VS} from Figure 4. Given a topological property b, and letting $b(\delta\lfloor_{vs_i})$ denote b_i and N_i denote $\mathsf{N}\langle b\rangle(vs_i, \delta, A_{\mathsf{VS}})$, we have

(i) $\mathsf{N}_4 = b_4$,
(ii) $\mathsf{N}_3 = b_3 + b_3\mathsf{N}_4 = b_3 + b_3b_4$,
(iii) $\mathsf{N}_2 = b_2 + b_2\mathsf{N}_4 = b_2 + b_2b_4$,
(iv) $\mathsf{N}_1 = b_1 + b_1\mathsf{N}_3 + b_1\mathsf{N}_4 = b_1 + b_1b_3 + b_1b_3b_4 + b_1b_4$, and the polynomial returned by $\mathsf{N}_{\mathsf{sum}}$ is
(v) $\mathsf{N}_{\mathsf{sum}}\langle b\rangle(\delta, A_{\mathsf{VS}}) = b_1 + b_2 + b_3 + b_4 + b_1b_3 + b_1b_4 + b_2b_4 + b_3b_4 + b_1b_3b_4$.

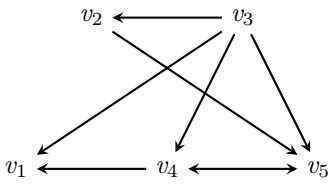
The value of $\mathsf{M}_{\mathsf{sum}}\langle b\rangle$ has an extra b_1 term and an extra b_1b_4 term, over that of $\mathsf{N}_{\mathsf{sum}}\langle b\rangle$. This is because $\mathsf{M}_{\mathsf{sum}}\langle b\rangle$ counts every ancestor vertex in the lifted dependency graph as many times as the size of its posterity. We found this phenomenon to be highly prevalent in standard benchmarks, where $\mathsf{N}_{\mathsf{sum}}$ substantially outperforms $\mathsf{M}_{\mathsf{sum}}$ in terms of the tightness of the computed bounds. Figure 5 shows the computed bounds of $\mathsf{N}_{\mathsf{sum}}$ versus $\mathsf{M}_{\mathsf{sum}}$. The base-case function that we used is defined as $b(\delta) = 2^{|\mathcal{D}(\delta)|} - 1$, for any system δ, which is an upper bound of a projection's recurrence diameter. The figure shows bounds computed for 3271 different International Planning Competition (IPC) benchmarks. It shows that $\mathsf{M}_{\mathsf{sum}}$ computes looser bounds as it repeats counting the ancestor vertices unnecessarily, while $\mathsf{N}_{\mathsf{sum}}$ avoids that repetition.

4.1.1 Soundness of $\mathsf{N}_{\mathsf{sum}}$. It remains to prove that the top-down algorithm, namely $\mathsf{N}_{\mathsf{sum}}$, is sound. Specifically, it remains to prove Theorem 2, which states that $\mathsf{N}_{\mathsf{sum}}\langle rd\rangle(\delta, A_{\mathsf{VS}})$ as an upper bound on the diameter $d(\delta)$ in case A_{VS} is an acyclic dependency graph for the system δ. We first define the *sublist diameter*.

Definition 15 (Sublist Diameter). *Recall that a list l' is a sublist of l, written $l' \preceq l$, iff all the members of l' occur in the same order in l. The sublist diameter, $\ell(\delta)$, is the length of the longest shortest sublist of any action*

$$
\begin{aligned}
[\,] \underset{vs}{\wr} \overrightarrow{\pi} &= \overrightarrow{\pi}\downarrow_{\overline{vs}} \\
\overrightarrow{\pi}_c \underset{vs}{\wr} [\,] &= [\,] \\
\pi_c :: \overrightarrow{\pi}_c \underset{vs}{\wr} \pi :: \overrightarrow{\pi} &= \begin{cases} \pi :: (\overrightarrow{\pi}_c \underset{vs}{\wr} \overrightarrow{\pi}) & \text{if } \pi\downarrow_{vs} = \pi_c \\ \pi_c :: \overrightarrow{\pi}_c \underset{vs}{\wr} \overrightarrow{\pi} & \text{o/wise} \end{cases}
\end{aligned}
$$

(a)



(b)

Fig. 6. (a)The definition of the stitching function ($\wr$), and (b) is the dependency graph of the system in Example 6.

sequence $\overrightarrow{\pi} \in \delta^*$ *which has an equivalent execution outcome to* $\overrightarrow{\pi}$ *starting at any state* $x \in \mathbb{U}(\delta)$*. Formally,*

$$\ell(\delta) = \max_{x \in \mathbb{U}(\delta)} \max_{\overrightarrow{\pi} \in \delta^*} \mathsf{min}\{|\overrightarrow{\pi}'| \mid \overrightarrow{\pi}' \preceq \overrightarrow{\pi} \wedge \overrightarrow{\pi}(x) = \overrightarrow{\pi}'(x)\}$$

It should be clear that the sublist diameter is an upper-bound on the diameter and a lower bound on the recurrence diameter. The sublist diameter is interesting because it circumvents the negative results of Theorems 5 and 6, as follows. Lemma 1 below shows that $\mathsf{N_{sum}}$ infers a sound bound on the diameter, given an acyclic lifted dependency graph, by composing the sublist diameters of the projections identified by the given graph. This inferred bound is particularly interesting as it shows that, unlike the diameter and recurrence diameter, the sublist diameter of a concrete system can be compositionally bounded using the projections' sublist diameters.

Our proof of Lemma 1 is rather technical. It is constructive and it depends on the fact that an action sequence in a system δ is, in essence, equivalent to an interleaving of different action sequences in different projections of δ. In the proof, we firstly show that for any constituent projected action sequence, redundant actions can be removed making it shorter than the sublist diameter of the respective projection. We then recombine all shortened projected action sequences with a novel proof artefact, *the stitching function* ($\wr$), and show that the recombined sequence begins and ends in the same states as the original one, and is shorter than the bound produced by $\mathsf{N_{sum}}$. This is in some sense a substantial generalisation of the theory developed earlier dealing with acyclic dependency graphs by Knoblock [42] and by Brafman and Domshlak [19], although they had different purposes from ours.

Lemma 1. *For any factored system* δ *with an acyclic lifted dependency DAG* A_{vs}*,* $\ell(\delta) \leq \mathsf{N_{sum}}\langle\ell\rangle(\delta, A_{\mathsf{vs}})$.

Proof. To prove Lemma 1 we use a construction which, given any action sequence $\overrightarrow{\pi} \in \delta^*$ violating the stated bound and a state $s \in \mathbb{U}(\delta)$, produces a sublist, $\overrightarrow{\pi}'$, of $\overrightarrow{\pi}$ satisfying that bound and $\overrightarrow{\pi}(s) = \overrightarrow{\pi}'(s)$. The proof is by induction on the structure of A_{vs}, where we repeatedly consider every set of variables $P \in A_{\mathsf{vs}}$ and each of its children $C \in \mathsf{children}_{A_{\mathsf{vs}}}(P)$, and for each parent child pair we do the following. We first consider the projected action sequence $\overrightarrow{\pi}\downarrow_C$. From the definition of ℓ, there is a sublist $\overrightarrow{\pi}'_C \preceq \overrightarrow{\pi}\downarrow_C$ satisfying $|\overrightarrow{\pi}'_C| \leq \ell(\delta\downarrow_C)$. Moreover, this guarantees that $\overrightarrow{\pi}'_C$ is equivalent, in terms of the execution outcome, to $\overrightarrow{\pi}\downarrow_C$. Hereupon, if for an action $\pi = (p, e)$, $e \subseteq vs$ is true, we call π a vs-action. The stitching function described in Figure 6a is then used to remove the C-actions in $\overrightarrow{\pi}$ whose projections on C are not in $\overrightarrow{\pi}'_C$. Thus our construction arrives at the action sequence $\overrightarrow{\pi}'' = \overrightarrow{\pi}'_C \underset{C}{\wr} \overrightarrow{\pi}$ with at most $\ell(\delta\downarrow_C)$ C-actions. Since $C \in \mathsf{children}_{A_{\mathsf{vs}}}(P)$, then $C \not\rightarrow P$ and accordingly, actions with variables from C in their effects never include P variables in their effects. Accordingly, $\overrightarrow{\pi}(s) = \overrightarrow{\pi}''(s)$, and in $\overrightarrow{\pi}''$, there will be a list, $\overrightarrow{\pi}_P$, of only P-actions between every consecutive C-actions. However, again, because of how ℓ is defined there is a $\overrightarrow{\pi}'_P \preceq \overrightarrow{\pi}_P$, whose execution outcome is the same as $\overrightarrow{\pi}_P$ and whose length is no more than $\ell(\delta\downarrow_P)$. Repeating this for all sequences of only P-actions completes our construction. □

The above construction and the usage of the stitching function are illustrated in the following example.

Example 6. Consider the following factored system, whose dependency graph is shown in Figure 6b.

$$\left\{ \begin{array}{l} \pi_1 = (\emptyset, \{v_3\}), \pi_2 = (\{v_3\}, \{v_4\}), \pi_3 = (\{v_3\}, \{\overline{v_1}\}), \pi_4 = (\{v_3\}, \{v_2\}), \\ \pi_5 = (\{v_4\}, \{v_1\}), \pi_6 = (\{v_2, v_4\}, \{v_5\}), \pi_7 = (\{\overline{v_3}\}, \{v_4, v_5\}) \end{array} \right\}$$

$\mathcal{D}(\delta) = c \cup p$, where $c = \{v_1, v_4, v_5\}$ are called the "child" variables, and $p = \{v_2, v_3\}$, and $c \not\rightarrow p$ are called the "parent" variables. In δ, the actions π_2, π_3, π_5, π_6, π_7 are c-actions, and π_1, π_4 are p-actions. An action sequence $\overrightarrow{\pi} \in \delta^*$ is $[\pi_1; \pi_1; \pi_2; \pi_3; \pi_4; \pi_4; \pi_5; \pi_6]$ that reaches the state $\{v_1, v_2, v_3, v_4, v_5\}$ from $\{v_1, \overline{v_2}, \overline{v_3}, \overline{v_4}, \overline{v_5}\}$. When $\overrightarrow{\pi}$ is projected on c it becomes $[\pi_2\downarrow_c; \pi_3\downarrow_c; \pi_5\downarrow_c; \pi_6\downarrow_c]$, which is in $({}^*\delta\downarrow_c)$. A shorter action sequence, $\overrightarrow{\pi}_c$, achieving the same result as $\overrightarrow{\pi}\downarrow_c$

is $[\pi_2\restriction_c; \pi_6\restriction_c]$. Since $\overrightarrow{\pi}_c$ is a sublist of $\overrightarrow{\pi}\restriction_c$, we can use the stitching function to obtain a shorter action sequence in δ^* that reaches the same state as $\overrightarrow{\pi}$. In this case, $\overrightarrow{\pi}_c \underset{c}{\S} \overrightarrow{\pi}$ is $[\pi_1; \pi_1; \pi_2; \pi_4; \pi_4; \pi_6]$. The second step is to contract the pure p segments which are $[\pi_1; \pi_1]$ and $[\pi_4; \pi_4]$, which are contracted to $[\pi_1]$ and $[\pi_4]$, respectively. The final constructed action sequence is $[\pi_1; \pi_2; \pi_4; \pi_6]$, which achieves the same state as $\overrightarrow{\pi}$.

Theorem 2 therefore follows, because:

(i) $\mathsf{N}_{\mathsf{sum}}$ is monotonic,
(ii) Lemma 1,
(iii) the sublist diameter is an upper bound on the diameter, and
(iv) the sublist diameter is a lower bound on the recurrence diameter.

4.1.2 $\mathsf{N}_{\mathsf{sum}}\langle rd\rangle$ is a Tight Bound. We now take our analysis a step further and prove that the top-down algorithm will provide the tightest upper bound we can hope for, however, under certain assumptions. Specifically, we prove that $\mathsf{N}_{\mathsf{sum}}$ is tight in the sense that it produces the tightest possible diameter bound compared to any bounding algorithm, given we restrict the input of the bounding algorithm to the following features: (i) the recurrence diameters of the projections identified by some acyclic lifted dependency graph, and (ii) the dependencies between the different projections—i.e. the edges of the lifted dependency graph that was use to compute the given projections. Of course, all is not lost. In subsequent sections, we shall improve on $\mathsf{N}_{\mathsf{sum}}$ by exposing more system features to the bounding procedure.

An interesting technical point concerns our formal statement of the tightness of $\mathsf{N}_{\mathsf{sum}}$. Effectively, that tightness theorem shows that, for every possible input to an algorithm that takes the two features above, $\mathsf{N}_{\mathsf{sum}}$ computes the tightest possible bound. To quantify over all possible inputs we encode the two features, namely projections' recurrence diameters and their dependencies, as a natural number labelled DAG, $A_{\mathbb{N}}$. That DAG has one vertex per projection and every one of its edges represents a dependency between two projections. Every vertex is labelled by a natural number that is the diameter of the corresponding projection. We show that $\mathsf{N}_{\mathsf{sum}}$ is tight by showing that for every possible input, there is a system whose diameter upper-bounds $\mathsf{N}_{\mathsf{sum}}\langle rd\rangle$. Together with Theorem 2—which states that $\mathsf{N}_{\mathsf{sum}}\langle rd\rangle$ upper-bounds the diameter—we therefore show that any sound algorithm that only uses the dependencies and recurrence diameters of projections cannot compute bounds that are tighter than the top-down algorithm $\mathsf{N}_{\mathsf{sum}}\langle rd\rangle$.

Theorem 3. *Let $\mathbb{N}$ be the set of natural numbers. For any $\mathbb{N}$-DAG, $A_{\mathbb{N}}$ (a DAG whose vertices are labelled with natural numbers), there is a factored system δ with a lifted dependency DAG, A_{VS}, where:*

(i) $\mathsf{N}_{\mathsf{sum}}\langle rd\rangle(\delta, A_{\mathsf{VS}}) \leq d(\delta)$, and
(ii) $A_{\mathbb{N}} = \mathfrak{R}(\!|A_{\mathsf{VS}}|\!)$, where $\mathfrak{R} : \mathsf{VS} \Rightarrow \mathbb{N}$ is a function where, for all $vs \subseteq \mathcal{D}(\delta)$, $\mathfrak{D}(vs) = rd(\delta\restriction_{vs})$. I.e. $A_{\mathbb{N}}$ is a relabelling of A_{VS}, where every vertex labelled by vs in A_{VS}, is relabelled by the recurrence diameter of the projection $\delta\restriction_{vs}$.

We defer the proof of this theorem to later. Note that because the recurrence diameter is an upper bound on the sublist diameter, and the sublist diameter is an upper bound on the diameter, and since $\mathsf{N}_{\mathsf{sum}}$ is monotonic, the following tightness theorem follows from Theorem 3 and Lemma 1.

Theorem 4. *For any $\mathbb{N}$-DAG, $A_{\mathbb{N}}$, there is a factored system δ with a lifted dependency DAG, A_{VS}, where:*

(i) $\mathsf{N}_{\mathsf{sum}}\langle \ell\rangle(\delta, A_{\mathsf{VS}}) = \ell(\delta)$, and
(ii) $A_{\mathbb{N}} = \mathfrak{L}(\!|A_{\mathsf{VS}}|\!)$, where $\mathfrak{L} : \mathsf{VS} \Rightarrow \mathbb{N}$ is a function where, for all $vs \subseteq \mathcal{D}(\delta)$, $\mathfrak{D}(vs) = \ell(\delta\restriction_{vs})$. I.e. $A_{\mathbb{N}}$ is a relabelling of A_{VS}, where every vertex labelled by vs in A_{VS}, is relabelled by the sublist diameter of the projection $\delta\restriction_{vs}$.

4.2 Why Recurrence Diameters?

Until now we have been bounding the system diameter compositionally, taking quantities other than the projections' diameters as the indeterminates in the polynomial expressions being generated by the bounding algorithm. This raises the question: could we bound the system diameter compositionally according to dependency structures, using projections' diameters? In general, the answer is no, as shown in Theorem 5 below.

More precisely, that theorem concerns the situation when the system features used by a bounding algorithm are restricted to: (i) the diameters of the projections identified by some lifted dependency graph, and (ii) the directed dependency relationship between the projections—i.e. the edges of the lifted dependency graph. Theorem 5 shows that an algorithm whose inputs are restricted to such features cannot be a sound bound on the diameter, assuming there is at least one dependency between the projections. This is done by showing that for any such algorithm and any input, there is counter-example to the validity of the output of the given algorithm as an upper bound on the diameter. In this theorem there is a quantification on algorithms and inputs. An algorithm is encoded more generally as a function mapping natural number labelled digraphs to natural numbers. An input, namely projections' diameters and their dependencies, is encoded as a natural number labelled digraph $\mathbb{N}$-graph, $G_{\mathbb{N}}$. The digraph $G_{\mathbb{N}}$ has one

vertex per projection and every edge in it represents a dependency between two projections, and every vertex in $G_{\mathbb{N}}$ is labelled by a natural number that is the diameter of the corresponding projection.

THEOREM 5. *For a function $f : \mathbb{N}$-graph $\Rightarrow \mathbb{N}$, and an $\mathbb{N}$-graph $G_{\mathbb{N}}$ with at least 1 edge, there is a factored system δ where:*

(i) $f(G_{\mathbb{N}}) < d(\delta)$, and
(ii) There is a lifted dependency graph G_{VS} for δ, such that $G_{\mathbb{N}} = \mathfrak{D}(\!|G_{\mathsf{VS}}|\!)$. I.e. $G_{\mathbb{N}}$ is a relabelling of G_{VS}: a vertex labelled by vs in G_{VS} is relabelled by the diameter of $\delta\downarrow_{vs}$.

We highlight that Theorem 5 holds irrespective of whether the lifted dependency graph exhibits a particular structure. In other words, irrespective of whether that directed graph is acyclic (which is a required condition for the soundness of $\mathsf{M}_{\mathsf{sum}}$), symmetric, etc., generally we cannot hope to compositionally bound the system diameter using only the two features stated earlier.

So far, computing the recurrence diameter is clearly motivated since we can compositionally bound the system diameter using projections' recurrence diameters. However, upper-bounding the recurrence diameter (i.e. the length of the longest simple path in a state space) for a given concrete system might be of interest in applications like model checking liveness properties, where the recurrence diameter is the smallest completeness threshold. Thus, it remains interesting to understand whether or not the recurrence diameter of a concrete system can be compositionally bounded using projections' recurrence diameters, where the projections are computed using a lifted dependency graph. Here, we examine that question finding that this is not possible, where we prove a negative result analogous to Theorem 5.

THEOREM 6. *For a function $f : \mathbb{N}$-graph $\Rightarrow \mathbb{N}$, and an $\mathbb{N}$-graph $G_{\mathbb{N}}$, there is a system δ where:*

(i) $f(G_{\mathbb{N}}) < rd(\delta)$, and
(ii) There is a lifted dependency graph G_{VS} for δ, such that $G_{\mathbb{N}} = \mathfrak{R}(\!|G_{\mathsf{VS}}|\!)$. I.e. $G_{\mathbb{N}}$ is a relabelling of G_{VS}: a vertex labelled by vs in G_{VS} is relabelled by the recurrence diameter of $\delta\downarrow_{vs}$.

4.3 Proofs and Constructions

We now present the proofs for the tightness theorem above, as well as the negative results on the inability to bound diameters using projections' diameters, and recurrence diameters using projections' recurrence diameters. However, we present the proofs in a different order from that in which we presented the theorems, an order which we find more natural considering the ideas we use in those proofs.

4.3.1 Proof of Theorem 5. The proof is made of three main steps. Firstly, for each given projection diameter m (i.e. label in $G_{\mathbb{N}}$) we construct a factored system γ_m, that is a path with diameter m. Those paths are constructed such that: i) their union is a system with a diameter that is more than $f(G_{\mathbb{N}})$, and ii) they are projections of the final construction δ, so their domains are disjoint, as the projections are supposed to be performed on a partition. Secondly, for every dependency from projection γ^{u_1} to γ^{u_2} (i.e. an edge $G_{\mathbb{N}}$), we construct an action which has preconditions from γ^{u_1} and effects from γ^{u_2}. Those actions are supposed to not change the state space of the final construction, they only add dependencies corresponding to the edges in $G_{\mathbb{N}}$. Thirdly, we show that the union of the constructed projections and the dependency inducing actions is the required witness δ, i.e. its diameter exceeds $f(G_{\mathbb{N}})$.

To facilitate our constructions, we use the following notation for states: states are annotated with a superscript, such that variables in states' domains are chosen to have different names if the superscripts of the states are different.[2] Formally, for two states x^i_a and x^j_b, $\mathcal{D}(x^i_a) = \mathcal{D}(x^j_b)$ if $i = j$, and $\mathcal{D}(x^i_a) \cap \mathcal{D}(x^j_b) = \emptyset$ otherwise. A path system is defined as the set of actions $\gamma^i_n = \{(x^i_j, x^i_{j+1}) \mid 1 \leq j \leq n\}$. We denote the sequence of actions $[(x^i_1, x^i_2); (x^i_2, x^i_3); \cdots (x^i_n, x^i_{n+1})]$ as $\overrightarrow{\gamma}^i_n$. Note that $\overrightarrow{\gamma}^i_n \in {\gamma^i_n}^*$, and that executing $\overrightarrow{\gamma}^i_n$ on x^i_1 reaches the state x^i_{n+1}, and induces a simple path with $n+1$ states in the state space of γ^i_n. Accordingly, $d(\gamma^i_n) = n$.

PROOF. Let $n = f(G_{\mathbb{N}}) + 1$. The constructed system's diameter will be at least n. Take an arbitrary edge $(u_1, u_2) \in E(G_{\mathbb{N}})$. For $u \in V(G_{\mathbb{N}})$, let γ^u denote the simple path $\gamma^{u_2}_{n+G_{\mathbb{N}}(u_2)}$ if $u = u_2$ and $\gamma^u_{G_{\mathbb{N}}(u)}$ otherwise. Firstly, we construct a system δ_{u_2} whose state space has a simple path with $G_{\mathbb{N}}(u_2)$ states and a clique with n states attached to the last state in the path. Thus, δ_{u_2} has $G_{\mathbb{N}}(u_2)$ as its diameter. Formally $\delta_{u_2} = \gamma^{u_2} \cup \{(x^{u_2}_i, x^{u_2}_j) \mid G_{\mathbb{N}}(u_2) \leq i, j \leq n + G_{\mathbb{N}}(u_2) + 1\}$.

Consider an auxiliary variable z (i.e. $z \notin \mathcal{D}(\gamma^{u_3})$, for any $u_3 \in V(G_{\mathbb{N}})$). Let $\delta'_{u_2} = \{(\{z\} \uplus p, e) \mid (p, e) \in \delta_{u_2}\} \cup \{(\{\overline{z}\} \uplus p, e) \mid (p, e) \in \gamma^{u_2}\}$, i.e. a transition system with two modes of operation determined by the value of z. If z is false then it operates as a path with a clique attached to its end, i.e. exactly like δ_{u_2}. Otherwise, it operates as a path of length $n + G_{\mathbb{N}}(u_2) + 1$, i.e. exactly like γ^{u_2}. In this way the largest component in the state space of δ'_{u_2} has a short diameter if z is removed from the actions, otherwise the diameter blows up to $d(\delta'_{u_2}) = n + G_{\mathbb{N}}(u_2)$.

[2] In our proofs we use the nodes in the dependency graph as a superscript.

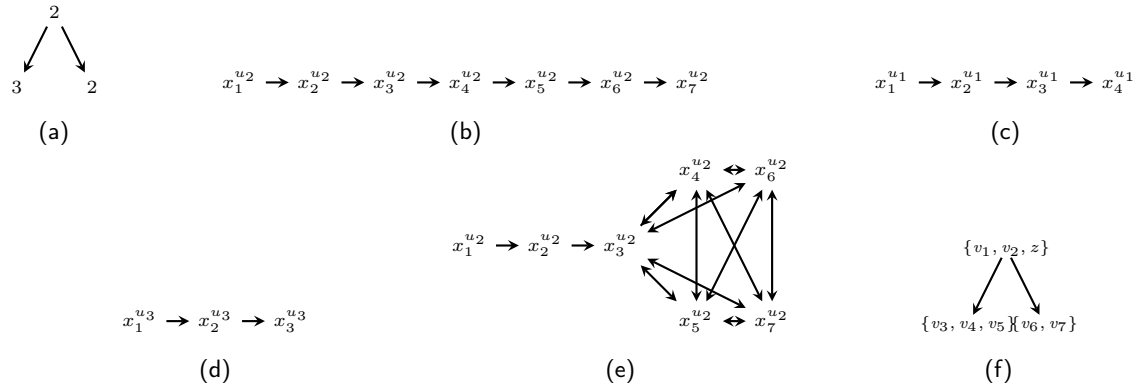


Fig. 7. Referring to Example 7: (a) is the natural number labelled graph. (b), (c), (d) and (e) are the largest connected components in $G(\gamma^{u_2})$, $G(\gamma^{u_1})$, $G(\gamma^{u_3})$, and $G(\delta_{u_2})$, respectively. (f) is a lifted dependency graph of the factored system.

Let $\delta_{u_1} = \{(\{\overline{z}\} \uplus p, \{\overline{z}\} \uplus e) \mid (p, e) \in \gamma^{u_1}\}$. This system operates exactly as the path γ^{u_1} when z is false. Since the value of z does not change, $d(\delta_{u_1}) = G_{\mathbb{N}}(u_1)$. We now show that $\delta = \{(x_1^{u_3} \uplus x_1^{u_4}, x_2^{u_4}) \mid (u_3, u_4) \in E(G_{\mathbb{N}})\} \cup \delta'_{u_2} \cup \delta_{u_1} \cup \bigcup_{u_3 \in V(G_{\mathbb{N}})} \gamma^{u_3}$ is our required witness.

To show that δ satisfies the first requirement (i), consider the states $x_1 = \{\overline{z}\} \uplus \biguplus_{u_3 \in V(G_{\mathbb{N}})} x_1^{u_3}$ and $x_2 = \{\overline{z}\} \uplus x^{u_2}_{n+G_{\mathbb{N}}(u_2)+1} \uplus \biguplus_{u_3 \in V(G_{\mathbb{N}}) \setminus u_2} x_1^{u_3}$. Let the action sequence $\overrightarrow{\pi}$ denote $\overrightarrow{\gamma}^{u_2}_{n+G_{\mathbb{N}}(u_2)}$ after adding the precondition $\overline{z}$ to the preconditions of all of its actions.[3] The sequence $\overrightarrow{\pi}$ reaches x_2 if executed at x_1, and there is not an action sequence from δ^* that can reach x_2 from x_1 that is shorter than $\overrightarrow{\pi}$. This means that the diameter of δ is at least equal to the length of $\overrightarrow{\pi}$, which is $n + G_{\mathbb{N}}(u_2)$.

To show that δ satisfies the second requirement (ii) on the witness, consider a relabelling, G_{VS}, of $G_{\mathbb{N}}$, where u_1 is relabelled by $\mathcal{D}(\delta_{u_1})$ and every other every vertex u_3 is relabelled by the domain of the path γ^{u_3}. Recall that δ had the set of actions $\{(x_1^{u_3} \uplus x_1^{u_4}, x_2^{u_4}) \mid (u_3, u_4) \in E(G_{\mathbb{N}})\}$ as a subset. These actions are constructed such that they add a dependency from $\mathcal{D}(\gamma^{u_3})$ to $\mathcal{D}(\gamma^{u_4})$ in δ iff $(u_3, u_4) \in E(G_{\mathbb{N}})$. This means that the edges of G_{VS} represent the dependencies of δ and accordingly it is a lifted dependency graph of δ. Also since, $\delta\lfloor_{\mathcal{D}(\gamma^{u_1})} = \gamma^{u_1}$, for $u_1 \in V(G_{\mathbb{N}})$, and since by construction $G_{\mathbb{N}}$ is a relabelling of G_{VS}, we have $G_{\mathbb{N}} = \mathfrak{D}(\!|G_{\mathsf{VS}}|\!)$. □

Example 7. This example shows the previous construction for a function $f : \mathbb{N}\text{-graph} \Rightarrow \mathbb{N}$, and the graph $G_{\mathbb{N}}$ shown in Figure 7a, where in this example $f(G_{\mathbb{N}}) = 4$, and accordingly $n = 5$. In $G_{\mathbb{N}}$ there are three vertices u_1 (the root), u_2, and u_3, labelled by the numbers 2, 3, and 2, respectively. In this case the first simple path system is $\gamma^{u_2} = \{(x_1^{u_2}, x_2^{u_2}), (x_2^{u_2}, x_3^{u_2}), (x_3^{u_2}, x_4^{u_2}), (x_4^{u_2}, x_5^{u_2}), (x_5^{u_2}, x_6^{u_2}), (x_6^{u_2}, x_7^{u_2})\}$, where $x_1^{u_2} = \{\overline{v_3}, \overline{v_4}, \overline{v_5}\}$, $x_2^{u_2} = \{\overline{v_3}, \overline{v_4}, v_5\}$, $x_3^{u_2} = \{\overline{v_3}, v_4, \overline{v_5}\}$, $x_4^{u_2} = \{\overline{v_3}, v_4, v_5\}$, $x_5^{u_2} = \{v_3, \overline{v_4}, \overline{v_5}\}$, $x_6^{u_2} = \{v_3, \overline{v_4}, v_5\}$, and $x_7^{u_2} = \{v_3, v_4, \overline{v_5}\}$. The second simple path system is $\gamma^{u_1} = \{(x_1^{u_1}, x_2^{u_1}), (x_2^{u_1}, x_3^{u_1}), (x_3^{u_1}, x_4^{u_1})\}$, where $x_1^{u_1} = \{\overline{v_1}, \overline{v_2}\}$, $x_2^{u_1} = \{\overline{v_1}, v_2\}$, $x_3^{u_1} = \{v_1, \overline{v_2}\}$, and $x_4^{u_1} = \{v_1, v_2\}$. The third simple path system is $\gamma^{u_3} = \{(x_1^{u_3}, x_2^{u_3}), (x_2^{u_3}, x_3^{u_3})\}$, where $x_1^{u_3} = \{\overline{v_6}, \overline{v_7}\}$, $x_2^{u_3} = \{\overline{v_6}, v_7\}$, and $x_3^{u_3} = \{v_6, \overline{v_7}\}$. The largest connected components of the state spaces of the three paths are shown in Figures 7b-7d, and their diameters are $d(\gamma^{u_2}) = n + G_{\mathbb{N}}(u_1) = 6$, $d(\gamma^{u_1}) = G_{\mathbb{N}}(u_2) = 3$, and $d(\gamma^{u_3}) = G_{\mathbb{N}}(u_3) = 2$.

The system δ_{u_2} which has a simple path of length $G_{\mathbb{N}}(u_2) = 3$ and a clique of size $n = 5$ attached to its last state, $x_3^{u_2}$, is $\delta_{u_2} = \{(x_1^{u_2}, x_2^{u_2}), (x_2^{u_2}, x_3^{u_2})\} \cup \{(x_i^{u_2}, x_j^{u_2}) \mid 3 \leq i, j \leq 7\}$. The largest component of its state space is shown in Figure 7e and $d(\delta_{u_2}) = G_{\mathbb{N}}(u_2) = 3$. To construct the system which has two modes of operation, δ'_{u_2}, we use a variable z to determine the two modes of δ'_{u_2}. We add the literal z to the preconditions of every action in δ_{u_2} to ensure that they are activated when z is true, and we add the literal $\overline{z}$ to the preconditions of every action in γ^{u_2} to ensure that they are activated when z is false. Thus, $\delta'_{u_2} = \{(\{z\} \uplus p, e) \mid (p, e) \in \delta_{u_2}\} \cup \{(\{\overline{z}\} \uplus p, e) \mid (p, e) \in \gamma^{u_2}\}$, where for instance, the action $(x_4^{u_2}, x_5^{u_2})$ from δ_{u_2} becomes $(\{z\} \uplus x_4^{u_2}, x_5^{u_2}) = (\{z, \overline{v_3}, v_4, v_5\}, \{v_3, \overline{v_4}, \overline{v_5}\})$ and the action $(x_4^{u_2}, x_5^{u_2})$ from γ^{u_2} becomes $(\{\overline{z}\} \uplus x_4^{u_2}, x_5^{u_2}) = (\{\overline{z}\} \uplus x_4^{u_2}, x_5^{u_2}) = (\{\overline{z}, \overline{v_3}, v_4, v_5\}, \{v_3, \overline{v_4}, \overline{v_5}\})$.

To construct δ_{u_1} we add $\overline{z}$ to the preconditions and effects of each action in γ^{u_1}. δ_{u_1} will have two modes of operation depending on the value of z. If z is true no actions at all can be executed, otherwise actions from γ^{u_1} can be executed, and accordingly $d(\delta_{u_1}) = d(\gamma^{u_1})$.

The required witness is $\delta = \{(x_1^{u_1} \uplus x_1^{u_2}, x_2^{u_2}), (x_1^{u_1} \uplus x_1^{u_3}, x_2^{u_3})\} \cup \delta'_{u_2} \cup \delta_{u_1} \cup \gamma^{u_3}$. The combined system has a diameter of 6 because it has a path isomorphic to the path in γ^{u_2} from $\{\overline{z}\} \uplus x_1^{u_2}$ $\{\overline{z}\} \uplus x_7^{u_2}$. In particular, this path is between the state $x_1 = \{\overline{z}\} \uplus x_1^{u_2} \uplus x_1^{u_1} \uplus x_1^{u_3} = \{\overline{z}, \overline{v_3}, \overline{v_4}, \overline{v_5}, \overline{v_1}, \overline{v_2}, \overline{v_6}, \overline{v_7}\}$ and the state $x_2 = \{\overline{z}\} \uplus x_7^{u_2} \uplus x_1^{u_1} \uplus x_1^{u_3} \{\overline{z}, v_3, v_4, \overline{v_5}, \overline{v_1}, \overline{v_2}, \overline{v_6}, \overline{v_7}\}$. The action sequence $\overrightarrow{\pi} = [(\overline{z} \uplus x_1^{u_2}, x_2^{u_2}); (\overline{z} \uplus x_2^{u_2}, x_3^{u_2}); (\overline{z} \uplus x_3^{u_2}, x_4^{u_2}); (\overline{z} \uplus x_4^{u_2}, x_5^{u_2}); (\overline{z} \uplus x_5^{u_2}, x_6^{u_2}); (\overline{z} \uplus x_6^{u_2}, x_7^{u_2})]$ reaches x_2

[3]Recall that $\overrightarrow{\gamma}^{u_2}_{n+G_{\mathbb{N}}(u_2)}$ denotes an action sequence that spans the length of the path γ^{u_2}.

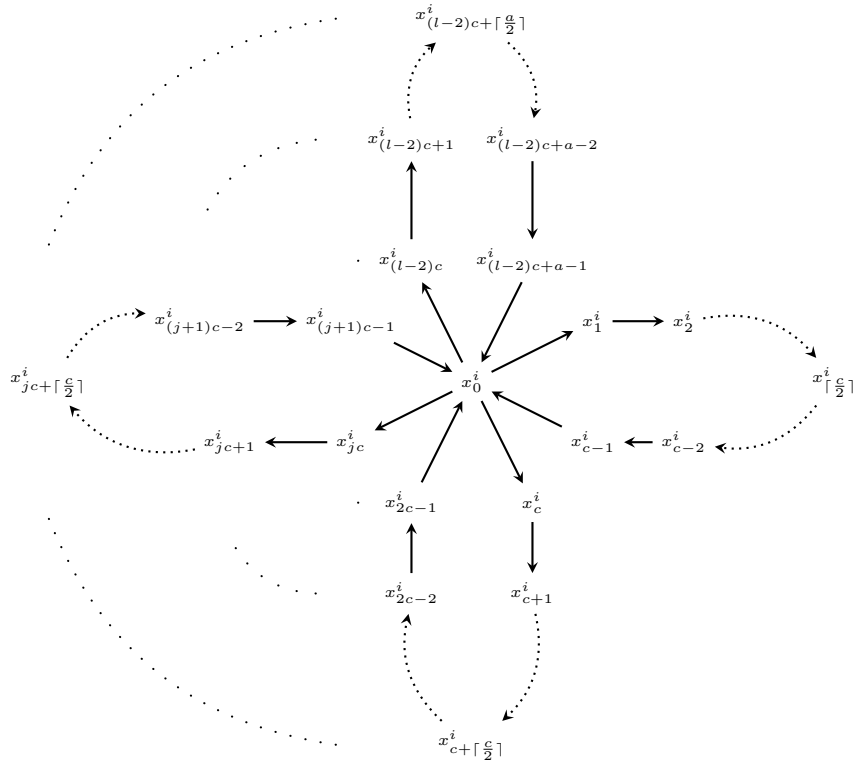


Fig. 8. The largest connected component in the state space of a flower $\mathbb{I}^i_{l,m}$ with $2 \leq m$. The vertices of this largest connected component are denoted by $\mathcal{S}(\mathbb{I}^i_{l,m})$.

if executed at x_1, and no action sequence shorter than $|\overrightarrow{\pi}|$ can reach x_2 from x_1. This means that $n+G_{\mathbb{N}}(u_1) = 6 \leq d(\delta)$. Also, Figure 7f is a lifted dependency graph of δ, as it has the actions $\{(x_1^{u_1} \uplus x_1^{u_2}, x_2^{u_2}), (x_1^{u_1} \uplus x_1^{u_3}, x_2^{u_3})\}$.

4.3.2 Proof of Theorem 6. The proof of this theorem is similar in structure to that of Theorem 5. One difference is that instead of constructing paths for projections, we construct systems whose state spaces resemble "flowers" (Figure 8). A flower can have arbitrarily many states without changing its recurrence diameter. This is needed to construct arbitrarily long paths with all distinct states in the final construction, while keeping the projections' recurrence diameters constant. To describe a flower, we start by describing systems to which we refer as petals. First let $c = \lfloor \frac{m}{2} \rfloor + 1$. A petal $\eta^i_{l,m}$ is a factored system whose state space is a loop of c states. One of those states has the index 0 and the other states have indices which depend on a given parameter l, which identifies the petal. Formally, a petal system is defined as $\eta^i_{j,m} = \{(x^i_0, x^i_{jc+1})\} \cup \{(x^i_{jc+k}, x^i_{jc+k+1}) \mid 1 \leq k \leq c-2\} \cup \{(x^i_{(j+1)c-1}, x^i_0)\}$.

A flower system, $\mathbb{I}^i_{l,m}$ is parameterised on l, which is the number of petals in it, and m, which is both, its diameter and recurrence diameter. If m is an even number, the flower $\mathbb{I}^i_{l,m}$ is defined as $\bigcup_{1 \leq j \leq l-1} \eta^i_{j,m}$. In this case, $\mathbb{I}^i_{l,m}$ is a factored system whose state space contains l cycles (the petals). All petals are of length $\lfloor \frac{m}{2} \rfloor + 1$. All petals have exactly one state in common between them (the "pistil"), which is x^i_0. In each petal j there is a state t^i_j (the "petal tip") defined as $t^i_j = x^i_{jc+\lceil \frac{c}{2} \rceil}$. For any j, t^i_j can be reached by exactly $\lceil \frac{c}{2} \rceil$ actions from x^i_0, and x^i_0 can be reached from t^i_j by exactly $\lfloor \frac{c}{2} \rfloor$ actions. Denoting by $\overrightarrow{\pi}^i_{j \mapsto k}$ the shortest action sequence that joins the petal tip t^i_j with t^i_k, $c \leq |\overrightarrow{\pi}^i_{j \mapsto k}|$ holds for $j \neq k$. Also $rd(\mathbb{I}^i_{l,m}) = m$ holds.

The above construction will not work if m is odd since the diameter of the constructed flower will be $m-1$. The constructed flower would have to be slightly modified to avoid that. Accordingly, if m is an odd number other than 1, the last petal of the flower (i.e. petal $l-1$) is constructed to have length $a = c+1$, and the other petals would still be of length c. The last petal's tip will be $t^i_j = x^i_{(l-2)c+\lceil \frac{a}{2} \rceil}$, and it can be reached by exactly $\lceil \frac{a}{2} \rceil$ actions from x^i_0, and x^i_0 can be reached from it by exactly $\lfloor \frac{a}{2} \rfloor$ actions.

If $m = 1$, $\mathbb{I}^i_{l,m}$ is constructed to be the clique $\bigcup_{1 \leq j < k \leq l-1} \{(x^i_j, x^i_k), (x^i_k, x^i_j)\}$ whose diameter is 1.

Lastly, we use $\mathcal{S}(\delta)$ to denote the largest connected component of any state in the state space of a system δ, where the connected component for a state x is $\mathcal{S}(\delta, x) = \{y \mid x \rightsquigarrow y \vee y \rightsquigarrow x\}$. Note: $\mathcal{S}(\delta)$ is always unique if δ is a flower.

Proof. Let $n = f(G_{\mathbb{N}})$. This is the number that the constructed system's recurrence diameter will exceed. For every vertex $u \in V(G_{\mathbb{N}})$, let $\mathbb{I}^u$ denote the flower $\mathbb{I}^u_{n+2,G_{\mathbb{N}}(u)}$. Let $\delta = \{(x_0^{u_1} \uplus x_0^{u_2}, x_1^{u_2}) \mid (u_1, u_2) \in E(G_{\mathbb{N}})\} \cup \bigcup_{u_1 \in V(G_{\mathbb{N}})} \mathbb{I}^{u_1}$. For each flower $\mathbb{I}^u$, denote its $n+2$ petal tips by t^u_j, for $0 \leq j \leq n+1$. Consider the two states: • $x_1 = \biguplus_{u \in V(G_{\mathbb{N}})} t^u_0$, and • $x_2 = \biguplus_{u \in V(G_{\mathbb{N}})} t^u_{n+1}$. We show that δ satisfies requirement (i) on the witness by giving a simple path between the states x_1 and x_2, whose length is more than n. Consider the action sequence $\overrightarrow{\pi}$ constructed by Algorithm 1.

Algorithm 1:

```
π⃗ =: []
for 0 ≤ i ≤ n + 1    /* Looping over petals */
    for u ∈ V(G_ℕ)    /* Looping over flowers */
        π⃗ =: π⃗ ⌢ π⃗^u_{i↦i+1}    /* Go from petal tip t^u_i to t^u_{i+1} */
```

The length of $\overrightarrow{\pi}$ is at least $(n+2)\sum_{u \in V(G_{\mathbb{N}})}(c) - 1$, since $c \leq |\overrightarrow{\pi}^u_{a \mapsto b}|$ for $a \neq b$. Since $c = \lfloor \frac{G_{\mathbb{N}}(u)}{2} \rfloor + 1$ then $n \leq |\overrightarrow{\pi}|$. Also $\overrightarrow{\pi}$ is constructed such that, if $\overrightarrow{\pi}$ is executed at x_1 it only traverses distinct states, i.e. it induces a simple path.

To show that δ satisfies requirement (ii), consider a relabelling, $G_{\mathtt{VS}}$, of $G_{\mathbb{N}}$, where every vertex u is relabelled by $\mathcal{D}(\mathbb{I}^u)$, the domain of the flower $\mathbb{I}^u$ associated with the vertex. Recall that δ had the set of actions $\{(x_0^{u_1} \uplus x_0^{u_2}, x_1^{u_2}) \mid (u_1, u_2) \in E(G_{\mathbb{N}})\}$ as a subset. These actions are constructed such that they add a dependency from $\mathcal{D}(\mathbb{I}^{u_1})$ to $\mathcal{D}(\mathbb{I}^{u_2})$ in δ iff $(u_1, u_2) \in E(G_{\mathbb{N}})$. This means that the edges of $G_{\mathtt{VS}}$ represent the dependencies of δ and accordingly it is a lifted dependency DAG of δ. Also since $\delta\restriction_{\mathcal{D}(\mathbb{I}^u)} = \mathbb{I}^u$ for any $u \in V(G_{\mathbb{N}})$, and by construction $G_{\mathbb{N}}$ is a relabelling of $G_{\mathtt{VS}}$, we have $G_{\mathbb{N}} = \mathfrak{R}(\!|G_{\mathtt{VS}}|\!)$. □

Example 8. This example shows the previous construction for a function f : $\mathbb{N}$-graph⇒ $\mathbb{N}$, and the graph $G_{\mathbb{N}}$ shown in Figure 7a, where $f(G_{\mathbb{N}}) = 2$. In this graph there are three vertices u_1 (the root), u_2, and u_3, labelled by the numbers 2, 3, and 2, respectively. We construct three flowers, one per vertex. For u_2 the constructed flower is $\mathbb{I}^{u_2}_{4,3} = \{(x_0^{u_2}, x_4^{u_2}), (x_4^{u_2}, x_5^{u_2}), (x_5^{u_2}, x_0^{u_2})\} \cup \bigcup\{\{(x_0^{u_2}, x_l^{u_2}), (x_l^{u_2}, x_0^{u_2})\} \mid 1 \leq l \leq 3\}$ where the states are defined as follows $x_0^{u_2} = \{\overline{v_4}, \overline{v_5}, \overline{v_6}\}$ (the pistil), $x_1^{u_2} = \{\overline{v_4}, \overline{v_5}, v_6\}$ (the first petal tip), $x_2^{u_2} = \{\overline{v_4}, v_5, \overline{v_6}\}$ (the second petal tip), $x_3^{u_2} = \{\overline{v_4}, v_5, v_6\}$ (the third petal tip), $x_4^{u_2} = \{v_4, \overline{v_5}, \overline{v_6}\}$, $x_5^{u_2} = \{v_4, \overline{v_5}, v_6\}$ (the fourth petal tip). Note that since for the flower $\mathbb{I}^{u_2}_{4,3}$, $m = 3$ (i.e. odd), the last petal has one more state in it (the petal at the bottom of Figure 9a). For u_3 the constructed flower is $\mathbb{I}^{u_3}_{4,2} = \bigcup\{\{(x_0^{u_3}, x_l^{u_3}), (x_l^{u_3}, x_0^{u_3})\} \mid 1 \leq l \leq 4\}$, where the states are defined as follows $x_0^{u_3} = \{\overline{v_7}, \overline{v_8}, \overline{v_9}\}$ (the pistil), $x_1^{u_3} = \{\overline{v_7}, \overline{v_8}, v_9\}$ (the first petal tip), $x_2^{u_3} = \{\overline{v_7}, v_8, \overline{v_9}\}$ (the second petal tip), $x_3^{u_3} = \{\overline{v_7}, v_8, v_9\}$ (the third petal tip), and $x_4^{u_3} = \{v_7, \overline{v_8}, \overline{v_9}\}$ (the fourth petal tip). For u_1 the constructed flower is $\mathbb{I}^{u_1}_{4,2} = \{\{(x_0^{u_1}, x_1^{u_1}), (x_l^{u_1}, x_0^{u_1})\} \mid 1 \leq 4\}$, where the states are defined as follows $x_0^{u_1} = \{\overline{v_1}, \overline{v_2}, \overline{v_3}\}$ (the pistil), $x_1^{u_1} = \{\overline{v_1}, \overline{v_2}, v_3\}$ (the first petal tip), $x_2^{u_1} = \{\overline{v_1}, v_2, \overline{v_3}\}$ (the second petal tip), $x_3^{u_1} = \{\overline{v_1}, v_2, v_3\}$ (the third petal tip), and $x_4^{u_1} = \{v_1, \overline{v_2}, \overline{v_3}\}$ (the fourth petal tip).

The required witness $\delta = \mathbb{I}^{u_1}_{4,2} \cup \mathbb{I}^{u_2}_{4,3} \cup \mathbb{I}^{u_3}_{4,2} \cup \{(x_0^{u_1} \uplus x_0^{u_2}, x_1^{u_2}), (x_0^{u_1} \uplus x_0^{u_3}, x_1^{u_3})\}$ where the actions $\{(x_0^{u_1} \uplus x_0^{u_2}, x_1^{u_2}), (x_0^{u_1} \uplus x_0^{u_3}, x_1^{u_3})\}$ add to δ dependencies equivalent to the edges of $G_{\mathbb{N}}$, i.e. the dependencies shown in Figure 9d.

Consider the states $x_1 = x_1^{u_1} \uplus x_1^{u_2} \uplus x_1^{u_3} = \{\overline{v_1}, \overline{v_2}, v_3, \overline{v_4}, \overline{v_5}, v_6, \overline{v_7}, \overline{v_8}, v_9\}$ and $x_2 = x_4^{u_1} \uplus x_4^{u_2} \uplus x_4^{u_3} = \{v_1, \overline{v_2}, \overline{v_3}, l, \overline{v_5}, v_6, v_7, \overline{v_8}, \overline{v_9}$ Following Algorithm 1, the resulting action sequence is $\overrightarrow{\pi} = \overrightarrow{\pi}^{u_1}_{1\mapsto 2} \frown \overrightarrow{\pi}^{u_2}_{1\mapsto 2} \frown \overrightarrow{\pi}^{u_3}_{1\mapsto 2} \frown \overrightarrow{\pi}^{u_1}_{2\mapsto 3} \frown \overrightarrow{\pi}^{u_2}_{2\mapsto 3} \frown \overrightarrow{\pi}^{u_3}_{2\mapsto 3} \frown \overrightarrow{\pi}^{u_1}_{3\mapsto 4} \frown \overrightarrow{\pi}^{u_2}_{3\mapsto 4} \frown \overrightarrow{\pi}^{u_3}_{3\mapsto 4}$ and its length is 18. $\overrightarrow{\pi}$ will reach x_2 if executed at x_1, while traversing all distinct states. The largest connected component of $G(\delta)$ and the path traversed by executing $\overrightarrow{\pi}$ from x_1 are shown in Figure 9e.

4.3.3 Proof of Theorem 3. We use an "inverted flower" system, which is intuitively a topological opposite of a flower system, where the pistil is replaced by a path and the petals are replaced by vertices. An inverted flower is defined as follows.

$$\mathbb{I}^i_{l,m} = \begin{cases} \bigcup_{1 \leq j < k \leq l}\{(x^i_j, x^i_k), (x^i_k, x^i_j)\} \text{ if } m = 1 \\ \bigcup_{2 \leq j \leq l+1}\{(x^i_1, x^i_j), (x^i_j, x^i_1)\} \text{ if } m = 2 \\ \gamma^i_{m-2} \cup \bigcup_{0 \leq j \leq l-1}\{(x^i_{m-1}, x^i_{m+j}), (x^i_{m+j}, x^i_1)\} \text{ o/w} \end{cases}$$

$\mathbb{I}^i_{l,m}$ is a system whose state space contains a simple path with $m-1$ states (instead of a pistil vertex) and l additional states (instead of l cycles which represent petals). Each petal has an outgoing edge to the first state in the path and an incoming edge from the last state in the path. For $\mathbb{I}^i_{l,m}$ we denote the shortest action sequence that joins x^i_a with x^i_b by $\overrightarrow{\pi}^i_{a \mapsto b}$, for $m \leq a, b \leq m + l - 1$. Since for $m \leq a, b \leq m + l - 1$ the length of $\overrightarrow{\pi}^i_{a \mapsto b}$ will always be m, we have $d(\mathbb{I}^i_{l,m}) = rd(\mathbb{I}^i_{l,m}) = m$. Figure 10 shows the state space of an inverted flower with $3 \leq m$.

[4]For two lists l_1 and l_2, $l_1 \frown l_2$ denotes their concatenation.

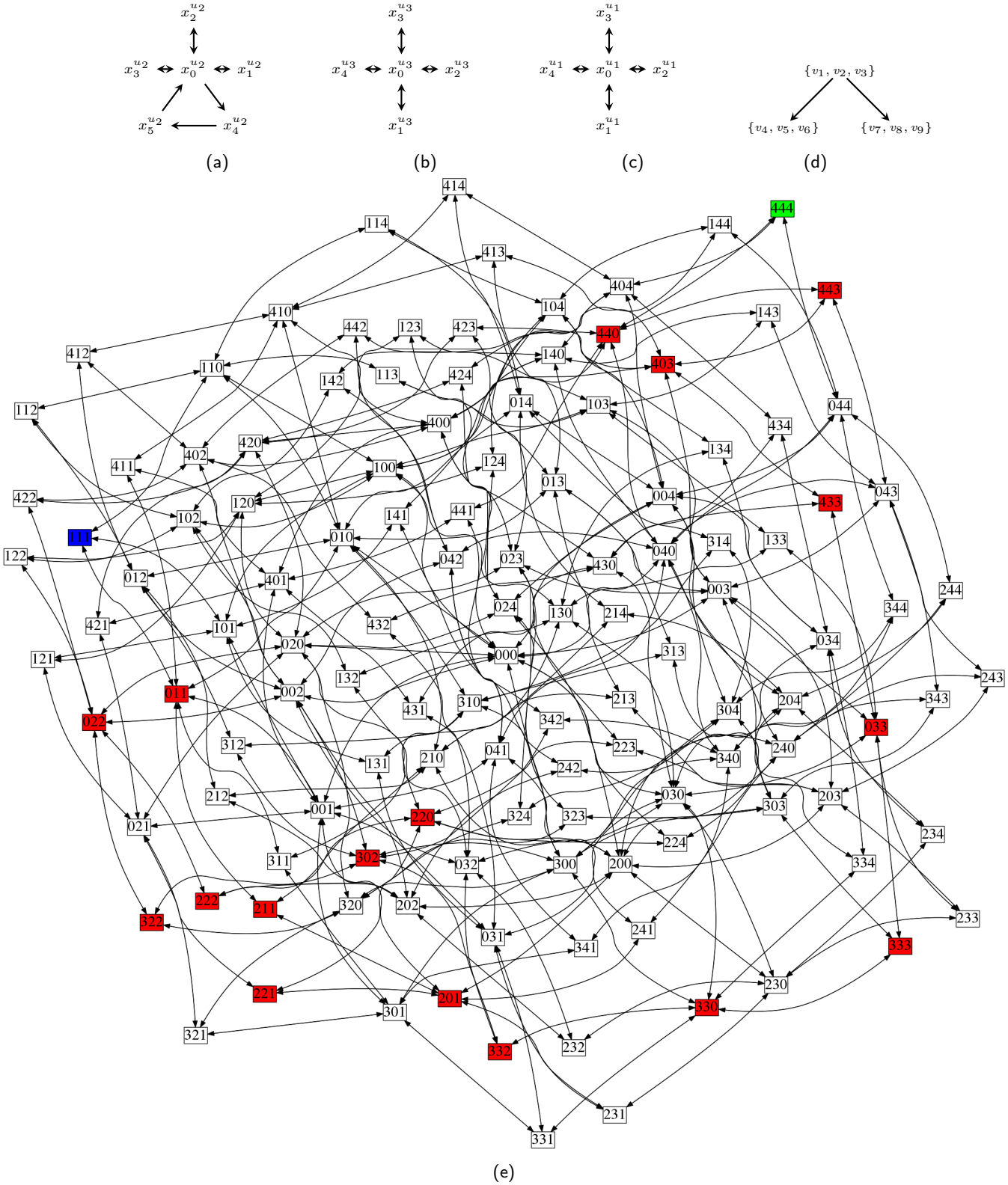


Fig. 9. Referring to Example 8. (a), (b), and (c) are the largest connected components in the state spaces of the flowers $\mathbb{I}_{4,2}^{u_1}$, $\mathbb{I}_{4,3}^{u_2}$ and $\mathbb{I}_{4,2}^{u_3}$, respectively, (d) a lifted dependency graph of the factored system δ, and (e) is the largest connected component in the state space of δ. In this graph our notation is flattened, where for instance the state $x_i^{u_1} \uplus x_j^{u_2} \uplus x_k^{u_3}$ is denoted by ijk. States x_1 and x_2 are indicated by the blue and green vertices respectively. The red vertices indicate the states traversed by $\overrightarrow{\pi}$ if executed on x_1. Note that (a), (b), and (c) are contractions of (e).

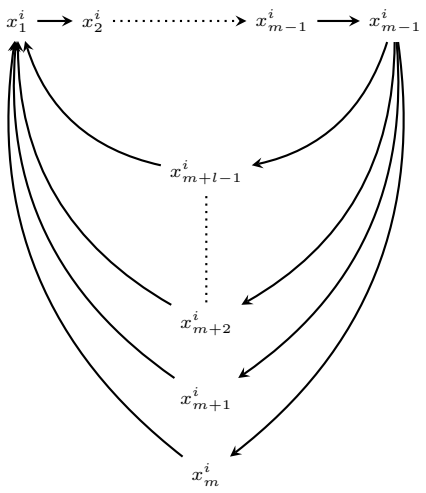


Fig. 10. An inverted flower with $3 \leq m$.

Proof. Our proof is constructive. The first step is to construct an inverted flower relabelling for every $u_1 \in V(A_{\mathbb{N}})$, so we denote by $\Downarrow^{u_1}$ the inverted flower $\Downarrow^{u_1}_{p_{u_1}, A_{\mathbb{N}}(u_1)}$. The relabelling inverted flowers are constructed with two properties in mind. Firstly, the recurrence diameter of an inverted flower label of u_1 matches $A_{\mathbb{N}}(u_1)$, the original label of u_1. Secondly, the number of petals, p_{u_1}, in the inverted flower labelling a vertex u_1 must be two more than the actions in all of inverted flowers labelling the children of u_1, i.e. $p_{u_1} = 2 + \sum_{u_2 \in \mathsf{children}_{A_{\mathbb{N}}}(u_1)} | \Downarrow^{u_2} |$. Having that many petals is crucial for the proof: we add a distinct petal as a precondition in every action in the inverted flowers of the children of a vertex u_1. This is to necessitate the execution of $A_{\mathbb{N}}(u_1)$ actions from $\Downarrow^{u_1}$ between every action from the inverted flower associated with every child of u_1. This causes the constructed system to have a diameter matching the bound computed by $\mathsf{N}_{\mathsf{sum}}$, which includes as a term the product of $A_{\mathbb{N}}(u_1)$ by the sum of all the labels of the children of u_1.

The next step in this construction is to build the required witness system δ. If we follow the same strategy as the ones in the constructions of Theorem 5 or Theorem 6, δ can be formed by the union of all the inverted flowers and then redundant actions with superfluous preconditions can be added to δ, to give the required dependencies. This would guarantee that δ satisfies condition ii. However, δ has to be constructed such that there is an action sequence $\overrightarrow{\pi} \in \delta^*$ that is $\mathsf{N}_{\mathsf{sum}}\langle rd \rangle(\delta, A_{\mathsf{VS}})$ actions long, and such that there are two states $x_1 \in \mathbb{U}(\delta)$ and $x_2 = x_1(\overrightarrow{\pi})$, with no action sequence shorter than $\overrightarrow{\pi}$ being able to reach x_2 from x_1. To guarantee this, actions added for the purpose of adding dependencies must replace the original actions, they cannot be redundant. Additionally, the preconditions of those added actions have to guarantee that in $\overrightarrow{\pi}$, between every pair of actions from an inverted flower associated with a vertex u_1, there needs to be as many action sequences as there are parents to u_1, each of which is as long as the recurrence diameter of the corresponding parent's inverted flower. To achieve that δ and $\overrightarrow{\pi}$ are constructed by Algorithm 2.

Algorithm 2:

$\overrightarrow{\pi} := []$; $\delta := \bigcup_{u_1 \in V(A_{\mathbb{N}})} (\Downarrow^{u_1})$
for $u_1 \in l_{\mathbb{N}}$ /* Loop over $V(A_{\mathbb{N}})$ in reverse topological order*/
  $i := A_{\mathbb{N}}(u_1)$; $\mathsf{children} := \bigcup_{u_2 \in \mathsf{children}_{A_{\mathbb{N}}}(u_1)} \Downarrow^{u_2}$
  for $(p, e) \in \overrightarrow{\pi}$ /*Loop over every action in $\overrightarrow{\pi}$, in execution order */
    if $\mathcal{D}(e) \cap \mathcal{D}(\mathsf{children}) \neq \emptyset$ /* check if the current action belongs to a child of u_1 */
      $\delta := \delta \setminus \{(p, e)\}$; $\delta := \delta \cup \{(x^{u_1}_{i+1} \uplus p, e)\}$ /* Add preconditions to (p, e) ensure it needs $\overrightarrow{\pi}^{u_1}_{i \mapsto i+1}$*/
      $\overrightarrow{\pi} := \mathsf{pfx}(\overrightarrow{\pi}, (p, e)) \frown \overrightarrow{\pi}^{u_1}_{i \mapsto i+1} \frown (x^{u_1}_{i+1} \uplus p, e) :: \mathsf{sfx}(\overrightarrow{\pi}, (p, e))$[5] /* Add $\overrightarrow{\pi}^{u_1}_{i \mapsto i+1}$ before (p, e)*/
    $i := i + 1$ /* Increment the petal counter*/
  $\overrightarrow{\pi} := \overrightarrow{\pi} \frown \overrightarrow{\pi}^{u_1}_{i \mapsto i+1}$ /* Add an action sequence to visit one more petal*/

We now show that δ satisfies requirement i. Consider two states, a state at which every inverted flower is at the first petal $x_1 = \biguplus_{u_1 \in V(A_{\mathbb{N}})} x^{u_1}_{A_{\mathbb{N}}(u_1)}$ and a state at which every inverted flower is at the last petal $x_2 = \biguplus_{v \in V(A_{\mathsf{VS}})} x^{u_1}_{A_{\mathbb{N}}(u_1)+p_{u_1}+1}$. By construction, $\overrightarrow{\pi}(x_1) = x_2$. Since every action in $\overrightarrow{\pi}$ had a different petal added as a

[5]For lists l, l_1, and l_3, if $l = l_1 \frown a :: l_2$, then $\mathsf{pfx}(l, a)$ denotes l_1 and $\mathsf{sfx}(l, a)$ denotes l_2.

precondition from every parent, for any $\overrightarrow{\pi}' \in \delta^*$ that is shorter than $\overrightarrow{\pi}$, $\overrightarrow{\pi}'(x_1) \neq \overrightarrow{\pi}(x_1)$. Accordingly the diameter of δ is at least $|\overrightarrow{\pi}|$. Since for an inverted flower $\Downarrow^{u_1}$, the length of $\overrightarrow{\pi}^{u_1}_{i\mapsto j}$ is $rd(\Downarrow^{u_1}) = A_{\mathbb{N}}(u_1)$, then the length of $\overrightarrow{\pi}$ is at least $\mathsf{N}_{\mathsf{sum}}\langle rd\rangle(\delta, A_{\mathsf{VS}})$, and δ satisfies conclusion i.

We now show that δ satisfies requirement ii. Consider A_{VS} which is a relabelling of $A_{\mathbb{N}}$, where every vertex $u_1 \in V(A_{\mathbb{N}})$ is relabelled by $\mathcal{D}(\Downarrow^{u_1})$, the domain of the inverted flower $\Downarrow^{u_1}$ associated with the vertex. Since for a vertex u_1, actions added from $\Downarrow^{u_1}$ to δ had preconditions added to them from the petals of inverted flowers labelling all the parents of u_1 and only the parents, edges of A_{VS} represent dependencies of δ. Accordingly A_{VS} is a lifted dependency DAG of δ. Also since $\delta\downarrow_{\mathcal{D}(\Downarrow^{u_1})} = \Downarrow^{u_1}$, and $A_{\mathbb{N}}$ is a relabelling of A_{VS}, then $A_{\mathbb{N}} = \mathfrak{R}(\!|A_{\mathsf{VS}}|\!)$. □

Example 9. For the $\mathbb{N}$-DAG shown in Figure 7a, a reverse topological ordering of the vertices in this graph is the list $[u_2; u_3; u_1]$, where leaves u_2 and u_3 come first. We first build an inverted flower system to label each of the vertices (Figures 11a-11c). As u_2 is a leaf, $p_{u_2} = 0$, so we construct an inverted flower $\Downarrow^{u_2}_{2,3}$ with two petals and a path with two states. Concretely, $\Downarrow^{u_2}_{2,3} = \{(x_1^{u_2}, x_2^{u_2}), (x_2^{u_2}, x_3^{u_2}), (x_2^{u_2}, x_4^{u_2}), (x_3^{u_2}, x_1^{u_2}), (x_4^{u_2}, x_1^{u_2})\}$. The states are $x_1^{u_2} = \{\overline{v_4}, \overline{v_5}\}$, $x_2^{u_2} = \{\overline{v_4}, v_5\}$, $x_3^{u_2} = \{v_4, \overline{v_5}\}$ (the first petal), $x_4^{u_2} = \{v_4, v_5\}$ (the second petal). For u_3 we construct $\Downarrow^{u_3}_{2,2} = \{(x_1^{u_3}, x_3^{u_3}), (x_1^{u_3}, x_2^{u_3}), (x_3^{u_3}, x_1^{u_3}), (x_2^{u_3}, x_1^{u_3})\}$. The states are $x_1^{u_3} = \{\overline{v_6}, \overline{v_7}\}$, $x_2^{u_3} = \{\overline{v_6}, v_7\}$ (the first petal), and $x_3^{u_3} = \{v_6, \overline{v_7}\}$ (the second petal). For the root vertex u_1, $p_{u_1} = 7$, as inverted flowers labelling its children have 5 actions, so $\Downarrow^{u_1}_{7,2}$ will have 7 petals. Concretely, $\Downarrow^{u_1}_{7,2} = \bigcup\{\{(x_1^{u_1}, x_j^{u_1}), (x_j^{u_1}, x_1^{u_1})\} \mid 2 \leq j \leq 8\}$. The states are $x_1^{u_1} = \{\overline{v_1}, \overline{v_2}, \overline{v_3}\}$, $x_2^{u_1} = \{\overline{v_1}, \overline{v_2}, v_3\}$, $x_3^{u_1} = \{\overline{v_1}, v_2, \overline{v_3}\}$, $x_4^{u_1} = \{\overline{v_1}, v_2, v_3\}$, $x_5^{u_1} = \{v_1, \overline{v_2}, \overline{v_3}\}$, $x_6^{u_1} = \{v_1, \overline{v_2}, v_3\}$, $x_7^{u_1} = \{v_1, v_2, \overline{v_3}\}$, and $x_8^{u_1} = \{v_1, v_2, v_3\}$.

After constructing the inverted flowers we construct the system δ and the action sequence $\overrightarrow{\pi}$. Based on Algorithm 2, we start from the leaves, and initially $\overrightarrow{\pi} = []$ and $\delta = \Downarrow^{u_2}_{2,3} \cup \Downarrow^{u_3}_{2,2} \cup \Downarrow^{u_1}_{7,2}$. u_2 has no children, i.e. $\mathsf{children} = \emptyset$, so we skip the inner for-loop and concatenate the shortest action sequence that reaches the second petal from the first one in the inverted flower associated with u_2, so $\overrightarrow{\pi} = \overrightarrow{\pi}^{u_2}_{3\mapsto 4} = [(x_3^{u_2}, x_1^{u_2}); (x_1^{u_2}, x_2^{u_2}); (x_2^{u_2}, x_4^{u_2})]$. Similarly, since u_3 has no children, we concatenate $\overrightarrow{\pi}^{u_3}_{1\mapsto 2}$ to $\overrightarrow{\pi}$, so $\overrightarrow{\pi} = \overrightarrow{\pi}^{u_2}_{3\mapsto 4} \frown \overrightarrow{\pi}^{u_3}_{2\mapsto 3} = [(x_3^{u_2}, x_1^{u_2}); (x_1^{u_2}, x_2^{u_2}); (x_2^{u_2}, x_4^{u_2}); (x_2^{u_3}, x_1^{u_3}); (x_1^{u_3}, x_3^{u_3})]$.

For u_1, $\mathsf{children} = \Downarrow^{u_2}_{2,3} \cup \Downarrow^{u_3}_{2,2}$, which means that the inner for-loop is executed, and it iterates over each action in $\overrightarrow{\pi}$. The first action in $\overrightarrow{\pi}$, $(x_3^{u_2}, x_1^{u_2})$ comes from the inverted flower labelling u_2, which is a child of u_1. Accordingly, we remove $(x_3^{u_2}, x_1^{u_2})$ from δ, add to its precondition the second petal $(x_3^{u_1})$ in the inverted flower labelling u_1, which makes the action $(x_3^{u_2} \uplus x_3^{u_1}, x_1^{u_2})$ and add the augmented action to δ. Then we add before that action, the shortest action sequence joining the first and the second petals in the inverted flower of u_1, so $\overrightarrow{\pi} = \overrightarrow{\pi}^{u_1}_{2\mapsto 3} \frown [(x_3^{u_1} \uplus x_3^{u_2}, x_1^{u_2}); (x_1^{u_2}, x_2^{u_2}); (x_2^{u_2}, x_4^{u_2}); (x_2^{u_3}, x_1^{u_3}); (x_1^{u_3}, x_3^{u_3})]$. This is repeated for the remaining actions, so $\overrightarrow{\pi} = \overrightarrow{\pi}^{u_1}_{2\mapsto 3} \frown [(x_3^{u_1} \uplus x_3^{u_2}, x_1^{u_2})] \frown \overrightarrow{\pi}^{u_1}_{3\mapsto 4} \frown [(x_4^{u_1} \uplus x_1^{u_2}, x_2^{u_2})] \frown \overrightarrow{\pi}^{u_1}_{4\mapsto 5} \frown [(x_5^{u_1} \uplus x_2^{u_2}, x_4^{u_2})] \frown \overrightarrow{\pi}^{u_1}_{5\mapsto 6} \frown [(x_6^{u_1} \uplus x_2^{u_3}, x_1^{u_3})] \frown \overrightarrow{\pi}^{u_1}_{6\mapsto 7} \frown [(x_7^{u_1} \uplus x_1^{u_3}, x_3^{u_3})]$. All the modified actions were replaced in δ with themselves, but with augmented preconditions. After the inner for loop ends, $\overrightarrow{\pi}^{u_1}_{7\mapsto 8}$ is concatenated, so $\overrightarrow{\pi} = \overrightarrow{\pi}^{u_1}_{2\mapsto 3} \frown [(x_3^{u_1} \uplus x_3^{u_2}, x_1^{u_2})] \frown \overrightarrow{\pi}^{u_1}_{3\mapsto 4} \frown [(x_4^{u_1} \uplus x_1^{u_2}, x_2^{u_2})] \frown \overrightarrow{\pi}^{u_1}_{4\mapsto 5} \frown [(x_5^{u_1} \uplus x_2^{u_2}, x_4^{u_2})] \frown \overrightarrow{\pi}^{u_1}_{5\mapsto 6} \frown [(x_6^{u_1} \uplus x_2^{u_3}, x_1^{u_3})] \frown \overrightarrow{\pi}^{u_1}_{6\mapsto 7} \frown [(x_7^{u_1} \uplus x_1^{u_3}, x_3^{u_3})] \frown \overrightarrow{\pi}^{u_1}_{7\mapsto 8}$.

Consider the states $x_1 = x_2^{u_1} \uplus x_3^{u_2} \uplus x_2^{u_3} = \{\overline{v_1}, \overline{v_2}, v_3, \overline{v_4}, v_5, \overline{v_6}, v_7\}$ and $x_2 = x_8^{u_1} \uplus x_4^{u_2} \uplus x_3^{u_3} = \{v_1, v_2, \overline{v_3}, \overline{v_4}, v_5, \overline{v_6}, v_7\}$. $\overrightarrow{\pi}(x_1) = x_2$, and the path traversed by $\overrightarrow{\pi}$ in $G(\delta)$ is shown in Figure 11e, There is not an action sequence in δ^* shorter than $\overrightarrow{\pi}$, whose length is 17, that can reach x_2 from x_1. Accordingly the diameter of δ is at least 17. Letting $\mathsf{N}(vs) = \mathsf{N}\langle rd\rangle(vs, \delta, A_{\mathsf{VS}})$, we have $\mathsf{N}(\{v_4, v_5\}) = rd(\delta\downarrow_{\{v_4,v_5\}}) = rd(\Downarrow^{u_2}_{2,3}) = 3$, $\mathsf{N}(\{v_4, v_5\}) = rd(\delta\downarrow_{\{v_6,v_7\}}) = rd(\Downarrow^{u_3}_{2,2}) = 2$, $\mathsf{N}(\{v_1, v_2, v_3\}) = rd(\delta\downarrow_{\{v_1,v_2,v_3\}})(1 + \mathsf{N}(\{v_4, v_5\}) + \mathsf{N}(\{v_6, v_7\})) = 2(1 + 3 + 2) = 12$, and $\mathsf{N}_{\mathsf{sum}}\langle rd\rangle(\delta, A_{\mathsf{VS}}) = \mathsf{N}(\{v_1, v_2, v_3\}) + \mathsf{N}(\{v_4, v_5\}) + \mathsf{N}(\{v_6, v_7\}) = 12 + 3 + 2 = 17$, so $\mathsf{N}_{\mathsf{sum}}\langle rd\rangle(\delta, A_{\mathsf{VS}}) \leq d(\delta)$.

The domain of δ is $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ and a partition of it is $\{\mathcal{D}(\Downarrow^{u_1}_{7,2}), \mathcal{D}(\Downarrow^{u_2}_{2,3}), \mathcal{D}(\Downarrow^{u_3}_{2,2})\} = \{\{v_1, v_2, v_3\}, \{v_4, v_5\}, \{v_6, v_7\}\}$. A lifted dependency graph of δ, A_{VS}, has the following labels: $A_{\mathsf{VS}}(u_1) = \{v_1, v_2, v_3\}$, $A_{\mathsf{VS}}(u_2) = \{v_4, v_5\}$, and $A_{\mathsf{VS}}(u_3) = \{v_6, v_7\}$ (shown in Figure 11d). It should be clear that $A_{\mathsf{VS}} = \mathcal{D}(\!|A_\Delta|\!)$.

4.4 Discussion

In Theorems 5 and 6, it might seem that all constructions and theorem statements are based on bad choices of projections which might be avoided. Indeed, it could very well be the case that projections could be carefully chosen to avoid the complications in the above constructions. However, the point that these results make is subtle but more profound: a compositional algorithm cannot decide whether the given projections are "sound" just by "looking" at the dependency graph. In other words, if a compositional algorithm (or a person for that matter) only analyses dependencies between variables to compute the projections, there will *always* be a rogue projection since, for every dependency configuration, we construct a system that violates the given bound.

We believe that this is interesting since the vast majority of previous algorithms that compute projections for bounding, reachability or otherwise, do so *only* via analysing dependencies between variables [4, 10, 12, 20, 37, 41, 42, 57, 61, 63]. Accordingly, existing compositional techniques that compute projections need non-trivial extensions

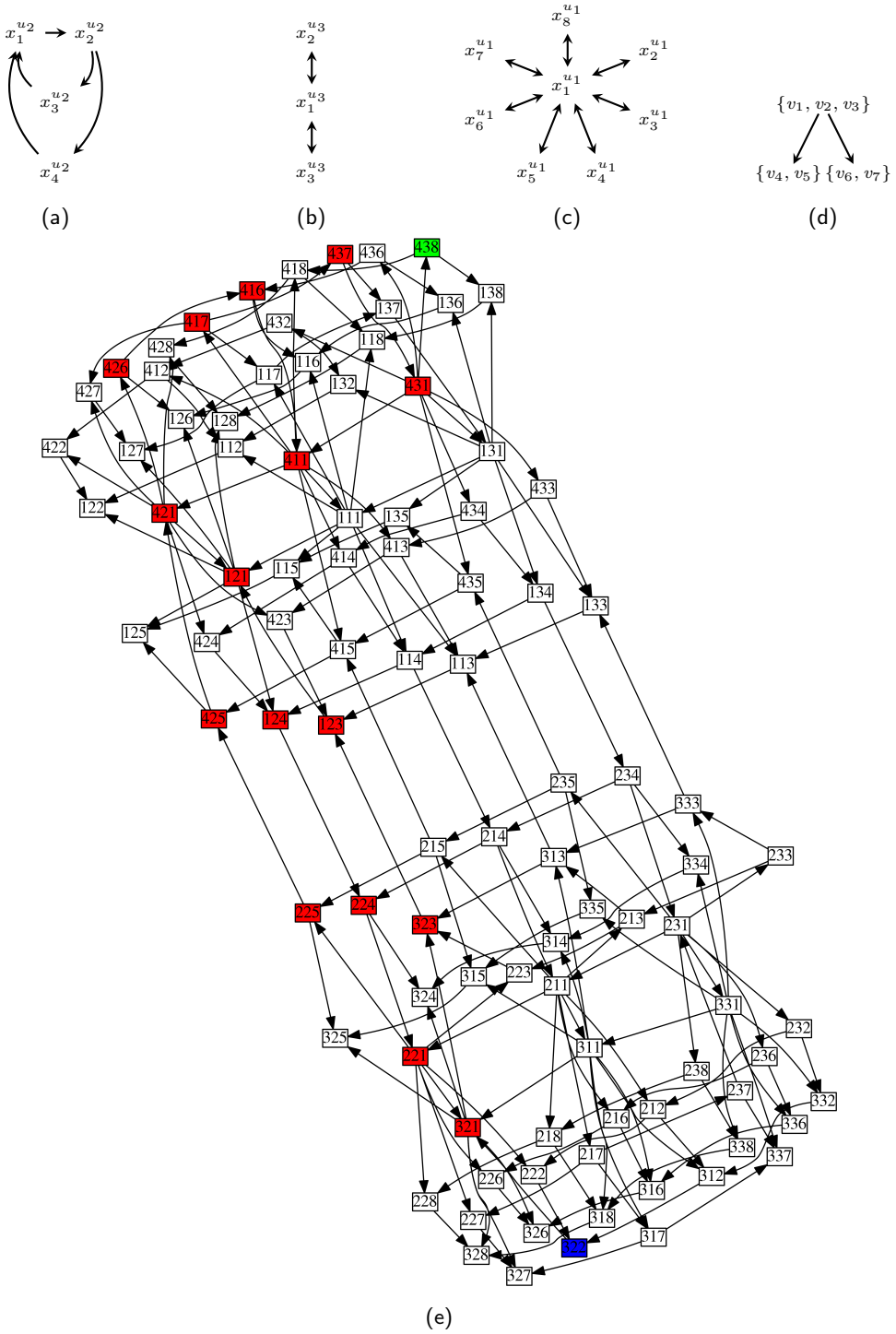


Fig. 11. Referring to Example 9, (a), (b), and (c) are the largest connected components in the state spaces of the inverted flowers $\overline{\mathfrak{l}}^{u_2}_{2,3}$, $\overline{\mathfrak{l}}^{u_3}_{2,2}$ and $\overline{\mathfrak{l}}^{u_1}_{7,2}$, respectively (i.e. $G(\overline{\mathfrak{l}}^{u_2}_{2,3}), G(\overline{\mathfrak{l}}^{u_3}_{2,2})$, and $G(\overline{\mathfrak{l}}^{u_1}_{7,2})$), (d) a lifted dependency graph of the factored system δ, and (e) is the largest connected component in the state space of δ. States x_1 and x_2 are indicated by the blue and green vertices respectively. The red vertices indicate the states traversed by $\overrightarrow{\pi}$ if executed on x_1. Note that (a), (b), and (c) are contractions of (e).

to analyse more than just dependencies (e.g. state space structure as in Section 5) to soundly bound the system's (recurrence) diameter using projections' (recurrence) diameters. This in some sense is unfortunate because analysing the dependency graph is computationally easy since its size is logarithmic in the size of the state space (i.e. linear in the number of state variables).

Furthermore, at first glance, it might seem that Theorems 5 and 6 are obvious since it is well-established that, in general, a projection over-approximates a system (i.e. a path in a projection may not correspond to a path in

the concrete system), and accordingly there can be arbitrarily short paths between states in projections that do not correspond to paths in the concrete system. However, such an impression is inaccurate, since many kinds of restrictions on the dependency graph do guarantee that paths in projections correspond to paths in the concrete system, and accordingly allow for solutions of reachability in projections to be composed to solve reachability in the concrete system, e.g. [10, 20, 37, 41, 42, 61, 63]. Thus, those two theorems are interesting as they show that *no* restrictions on the dependency graph *alone* (e.g. symmetry, acyclicity, tree decompositions, etc.) can guarantee the sound use of projections for compositional bounding of the diameter or the recurrence diameter, in a striking contrast to the problem of succinct reachability.

Theorems 5 and 6 show that projection, even when guided by dependencies, causes information loss which causes recovering any upper bound on the diameter or the recurrence diameter impossible. Interestingly, the sublist diameter seems to avoid this since it can be bounded using projections' sublist diameters as shown in Lemma 1. This might be rooted at the fact that, unlike the diameter or the recurrence diameter, the sublist diameter takes the factored system representation into consideration. However, further investigation is needed to form an intuition as to how this information loss does not affect the sublist diameter.

Similar to Theorems 5 and 6, Theorem 3 shows that for a function to compute bounds tighter than those computed by the top-down algorithm $\mathsf{N}_{\mathsf{sum}}$, the function must analyse more than just the dependencies between the projections and their recurrence diameters. This was shown by constructing a transition system whose diameter is the same as the bound computed by $\mathsf{N}_{\mathsf{sum}}$ for any combination of projections' recurrence diameters and dependencies between projection, represented as a digraph labelled by natural numbers. A challenge in formulating that tightness theorem was identifying that $\mathsf{N}_{\mathsf{sum}}$ is tight only w.r.t. projection recurrence diameters and projection dependencies. This was tricky because it implicitly meant that, although the signature of $\mathsf{N}_{\mathsf{sum}}$ appears to be taking the entire system and the dependency graph as input, we needed to pinpoint the relevant part of the input that $\mathsf{N}_{\mathsf{sum}}$ uses and w.r.t. which it is tight.

5 EXPLOITING STATE SPACE ACYCLICITY

The polynomial generator $\mathsf{N}_{\mathsf{sum}}$ is tight, and therefore if we hope to improve on the bounds we can obtain via dependency graph analysis, we must leverage additional system features that are not available to $\mathsf{N}_{\mathsf{sum}}$. The new feature we consider here is that of acyclicity in the transition system's state space. Recall, a system is acyclic iff no state can be encountered twice in an execution. We firstly motivate acyclicity by appealing to a verification case-study called the *hotel key protocol*, the details of which are given below. Then we describe different bounding procedures that exploit state space acyclicity as another system feature.

5.1 Hotel Key Protocol: A Running Example

We now consider the hotel key protocol from Jackson [39]. Reasoning about safe and unsafe versions of this protocol is challenging for state-of-the-art AI planners and model-checkers. For example, a version of the protocol was shown unsafe for an instance with 1 room, 2 guests and 4 keys using a counterexample generator [18]. The problem becomes more challenging for the safe version of the protocol, where the only feasible approach is using interactive theorem provers, as was done by Nipkow [53].

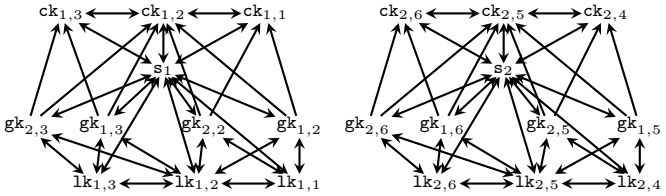
We describe the factored transition system corresponding to that protocol. The system models a hotel with R rooms, G guests, and K keys per room, which guests can use to enter rooms (Figure 14 shows an example with $R = 2$, $G = 2$ and $K = 3$). The state characterising propositions are: (i) $\mathtt{lk}_{r,k}$, reception last issued key k for room r, for $0 < r \leq R$ and $(r-1)K < k \leq rK$; (ii) $\mathtt{ck}_{r,k}$, room r can be accessed using key k, for $0 < r \leq R$ and $(r-1)k < k \leq rK$; (iii) $\mathtt{gk}_{g,k}$, guest g has key k, for $0 < g \leq G$, $0 < k \leq RK$; and (iv) $\mathtt{s}_r$, is an auxiliary variable that means that room r is "safely" delivered to some guest. The protocol actions are as follows: (i) guest g can check-in to room r, receiving key k—$(\{\mathtt{lk}_{r,k_1}\}, \{\mathtt{gk}_{g,k_2}, \mathtt{lk}_{r,k_2}, \overline{\mathtt{lk}_{r,k_1}}, \overline{\mathtt{s}_r}\})$; and (ii) where room r was previously entered using key k, guest g can enter room r using key k'—$(\{\mathtt{gk}_{g,k'}, \mathtt{lk}_{r,k}\}, \{\mathtt{ck}_{r,k'}, \overline{\mathtt{ck}_{r,k}}, \mathtt{s}_r\})$. Thus, guests can retain keys indefinitely, and there is no direct communication between rooms and reception.

For completeness, we note that this protocol was formulated in the context of checking safety properties. Safety is violated only if a guest enters a room occupied by another guest. Formally, the safety of this protocol is checked by querying if there exists a room r, guest g and keys $k \neq k'$, so that $\mathtt{lk}_{r,k'} \wedge \mathtt{ck}_{r,k} \wedge \mathtt{gk}_{g,k'} \wedge \mathtt{s}_r$. The initial state asserts that guests possess no keys, and the reception issued the first key for each room, and each room opens with its first key. Formally, this is represented by asserting $\mathtt{lk}_{r,(r-1)K} \wedge \mathtt{ck}_{r,(r-1)K}$ is true for $1 \leq r \leq R, (r-1)K < k \leq rK$, and that all other state variables are false.

We adopt some shorthand notations in order to provide examples of concepts in terms of the hotel key protocol. A variable name is written in upper case to refer to a particular assignment, where the only variable that is true is given by the indices. For example, the assignment $\{\overline{\mathtt{ck}_{1,1}}, \mathtt{ck}_{1,2}, \overline{\mathtt{ck}_{1,3}}\}$—indicating room 1 can be accessed using key 2—is

; check in to a room (at reception), receiving a new key
$(\{\mathtt{lk}_{1,1}\}, \{\mathtt{gk}_{1,2}, \mathtt{lk}_{1,2}, \overline{\mathtt{lk}_{1,1}}, \overline{\mathtt{s}_1}\}), (\{\mathtt{lk}_{1,2}\}, \{\mathtt{gk}_{1,3}, \mathtt{lk}_{1,3}, \overline{\mathtt{lk}_{1,2}}, \overline{\mathtt{s}_1}\}),$
$(\{\mathtt{lk}_{1,1}\}, \{\mathtt{gk}_{2,2}, \mathtt{lk}_{1,2}, \overline{\mathtt{lk}_{1,1}}, \overline{\mathtt{s}_1}\}), (\{\mathtt{lk}_{1,2}\}, \{\mathtt{gk}_{2,3}, \mathtt{lk}_{1,3}, \overline{\mathtt{lk}_{1,2}}, \overline{\mathtt{s}_1}\}),$
$(\{\mathtt{lk}_{2,4}\}, \{\mathtt{gk}_{1,5}, \mathtt{lk}_{1,5}, \overline{\mathtt{lk}_{1,4}}, \overline{\mathtt{s}_2}\}), (\{\mathtt{lk}_{2,5}\}, \{\mathtt{gk}_{1,6}, \mathtt{lk}_{1,6}, \overline{\mathtt{lk}_{1,5}}, \overline{\mathtt{s}_2}\}),$
$(\{\mathtt{lk}_{2,4}\}, \{\mathtt{gk}_{2,5}, \mathtt{lk}_{1,5}, \overline{\mathtt{lk}_{1,4}}, \overline{\mathtt{s}_2}\}), (\{\mathtt{lk}_{2,5}\}, \{\mathtt{gk}_{2,6}, \mathtt{lk}_{1,6}, \overline{\mathtt{lk}_{1,5}}, \overline{\mathtt{s}_2}\})$
; enter a room with new key
$(\{\mathtt{gk}_{1,2}\}, \{\mathtt{ck}_{1,2}, \overline{\mathtt{ck}_{1,1}}, \mathtt{s}_1\}), (\{\mathtt{gk}_{2,2}\}, \{\mathtt{ck}_{1,2}, \overline{\mathtt{ck}_{1,1}}, \mathtt{s}_1\}),$
$(\{\mathtt{gk}_{1,3}\}, \{\mathtt{ck}_{1,3}, \overline{\mathtt{ck}_{1,2}}, \mathtt{s}_1\}), (\{\mathtt{gk}_{2,3}\}, \{\mathtt{ck}_{1,3}, \overline{\mathtt{ck}_{1,2}}, \mathtt{s}_1\}),$
$(\{\mathtt{gk}_{1,5}\}, \{\mathtt{ck}_{2,5}, \overline{\mathtt{ck}_{2,4}}, \mathtt{s}_2\}), (\{\mathtt{gk}_{2,5}\}, \{\mathtt{ck}_{2,5}, \overline{\mathtt{ck}_{2,4}}, \mathtt{s}_2\}),$
$(\{\mathtt{gk}_{1,6}\}, \{\mathtt{ck}_{2,6}, \overline{\mathtt{ck}_{2,5}}, \mathtt{s}_2\}), (\{\mathtt{gk}_{2,6}\}, \{\mathtt{ck}_{2,6}, \overline{\mathtt{ck}_{2,5}}, \mathtt{s}_2\})$

(a)



(b)

$(\{\mathtt{lk}_{1,1}\}, \{\mathtt{gk}_{1,2}, \mathtt{lk}_{1,2}, \overline{\mathtt{lk}_{1,1}}, \overline{\mathtt{s}_1}\}), (\{\mathtt{lk}_{1,2}\}, \{\mathtt{gk}_{1,3}, \mathtt{lk}_{1,3}, \overline{\mathtt{lk}_{1,2}}, \overline{\mathtt{s}_1}\}),$
$(\{\mathtt{lk}_{1,1}\}, \{\mathtt{gk}_{2,2}, \mathtt{lk}_{1,2}, \overline{\mathtt{lk}_{1,1}}, \overline{\mathtt{s}_1}\}), (\{\mathtt{lk}_{1,2}\}, \{\mathtt{gk}_{2,3}, \mathtt{lk}_{1,3}, \overline{\mathtt{lk}_{1,2}}, \overline{\mathtt{s}_1}\}),$
$(\{\mathtt{gk}_{1,2}\}, \{\mathtt{ck}_{1,2}, \overline{\mathtt{ck}_{1,1}}, \mathtt{s}_1\}), (\{\mathtt{gk}_{2,2}\}, \{\mathtt{ck}_{1,2}, \overline{\mathtt{ck}_{1,1}}, \mathtt{s}_1\}),$
$(\{\mathtt{gk}_{1,3}\}, \{\mathtt{ck}_{1,3}, \overline{\mathtt{ck}_{1,2}}, \mathtt{s}_1\}), (\{\mathtt{gk}_{2,3}\}, \{\mathtt{ck}_{1,3}, \overline{\mathtt{ck}_{1,2}}, \mathtt{s}_1\})$

(c)

Fig. 12. (a) shows the actions of a *transition system* δ representing the hotel key protocol with 2 rooms, 2 guests and 3 keys per room; room 1 is associated with keys 1–3; room 2 with keys 4–6. (b) is the *dependency graph* for that system. (c) is the *projection* of the system on an abstraction that models only the changes related to room 1.

indicated by writing $\mathtt{CK}_{1,2}$. We refer to sets of variables by omitting an index term. For example, $\mathtt{lk}_1$ indicates the variables $\{\mathtt{lk}_{1,i} \mid 1 \leq i \leq 3\}$.

In a moment we delve into the detail of acyclicity, and expose how that property can be used to obtain a useful bound on the system diameter. It is informative to first consider the abstract sub-problems one encounters when we examine the hotel key protocol according to its state-variable dependency structure. The following examples illustrate the concepts of projection and state-variable dependency for the hotel key protocol.

Example 10. Consider the set of variables $\textsc{Room1} \equiv \mathtt{lk}_1 \cup \mathtt{ck}_1 \cup \{\mathtt{gk}_{1,2}, \mathtt{gk}_{1,3}, \mathtt{gk}_{2,2}, \mathtt{gk}_{2,3}\}$. The variables Room1 model system state relevant to the 1st hotel room. Figure 12c shows the projected system $\delta\downarrow_{\textsc{Room1}}$.

Example 11. Figure 12b shows a dependency graph associated with the system from Figure 12a. Let $\textsc{Room2} \equiv \mathtt{lk}_2 \cup \mathtt{ck}_2 \cup \{\mathtt{gk}_{1,5}, \mathtt{gk}_{1,6}, \mathtt{gk}_{2,5}, \mathtt{gk}_{2,6}\}$. Figure 12b depicts two connected components induced by the sets Room1 and Room2, respectively. One lifted dependency graph would have exactly two unconnected vertices one being a contraction of the vertices from Room1, and the other a contraction of those from Room2. Due to the disconnected structure of the dependency graph, intuitively the sum of bounds for $\delta\downarrow_{\textsc{Room1}}$ and $\delta\downarrow_{\textsc{Room2}}$ can be used to upper-bound the diameter of the concrete system.

5.2 Leveraging Acyclicity for Tighter Bounds

As we stated earlier, Theorem 3 implies that the only way $\mathsf{N}_{\mathsf{sum}}$ can be improved is by letting it analyse more structure than the dependency graph and the projections' recurrence diameters. Here, we improve it by allowing it to analyse more of the structure of the projections. In addition to using system features used by $\mathsf{N}_{\mathsf{sum}}$, we redesign $\mathsf{N}_{\mathsf{sum}}$ to use the fact of whether or not a projection is acyclic. We first formalise state space acyclicity as follows.

Definition 16 (Acyclic Transition System). *δ is acyclic iff $\forall x, x' \in \mathbb{U}(\delta).\ x \neq x'$ then $x \not\rightsquigarrow x'$ or $x' \not\rightsquigarrow x$.*

The next example illustrates that variable dependency information is not indicative of whether or not a state space is acyclic. As we develop methods to detect the property of acyclicity, we shall see that these can complement approaches that bound topological properties according to dependency between state variables. Indeed, the tightest bounds we obtain in the sequel are calculated by algorithms that use information about variable (in)dependence and acyclicity in the transition system.

Example 12. $\delta\downarrow_{\mathtt{ck}_1}$ is acyclic. For example, no state satisfying $\mathtt{CK}_{1,2}$ can be reached from a state satisfying $\mathtt{CK}_{1,3}$. Now consider $\delta\downarrow_{\text{Room1}}$ from Example 10. The dependency graph of $\delta\downarrow_{\text{Room1}}$ is comprised of one strongly connected component (SCC). Thus, acyclicity in the assignments of $\mathtt{ck}_1$ cannot be exploited in $\delta\downarrow_{\text{Room1}}$ by analysing its dependency graph.

The following modified version of the top-down algorithm exploits acyclicity in the state space. We shall soon how it can be used as a sound procedure for bounding the system diameter.

Definition 17 (Acyclic Dependency/State Space Compositional Bound).

$$\widehat{\mathsf{N}}\langle b\rangle(vs,\delta,G_{\mathtt{VS}}) \;=\; \begin{cases} b(\delta\downarrow_{vs}) & \text{if } \delta\downarrow_{vs} \text{ is an acyclic system} \\ b(\delta\downarrow_{vs})(1+\sum_{c\in\mathsf{children}_{G_{\mathtt{VS}}}(vs)}\widehat{\mathsf{N}}\langle b\rangle(c,\delta,G_{\mathtt{VS}})) & \text{o/wise} \end{cases}$$

Then, let $\widehat{\mathsf{N}}_{\mathsf{sum}}\langle b\rangle(\delta,G_{\mathtt{VS}})=\sum_{vs\in G_{\mathtt{VS}}}\widehat{\mathsf{N}}\langle b\rangle(vs,\delta,G_{\mathtt{VS}})$.

Similar to $\mathsf{N}_{\mathsf{sum}}$, $\widehat{\mathsf{N}}_{\mathsf{sum}}$ is recursively defined on the structure of $G_{\mathtt{VS}}$, and accordingly it is only well-defined if $G_{\mathtt{VS}}$ is a DAG, and it is also monotonic. However, the difference between $\mathsf{N}_{\mathsf{sum}}$ and $\widehat{\mathsf{N}}_{\mathsf{sum}}$ occurs when projections have acyclic state spaces. In such case $\widehat{\mathsf{N}}_{\mathsf{sum}}$ will only return the bound on the projection without multiplying it by the bounds on any of its children. This can lead to substantial reduction in the computed upper bound as shown in the following example.

Example 13. We again refer to $A_{\mathtt{VS}}$ from Figure 4 and use the same notation from Example 5. If $\delta\downarrow_{vs_1}$ and $\delta\downarrow_{vs_3}$ are both acyclic systems we would have

(i) $\widehat{\mathsf{N}}_4 = b_4$,
(ii) $\widehat{\mathsf{N}}_3 = b_3$,
(iii) $\widehat{\mathsf{N}}_2 = b_2 + b_2\widehat{\mathsf{N}}_4 = b_2 + b_2b_4$,
(iv) $\widehat{\mathsf{N}}_1 = b_1$, and the polynomial returned by $\widehat{\mathsf{N}}_{\mathsf{sum}}$ is
(v) $\widehat{\mathsf{N}}_{\mathsf{sum}}\langle b\rangle(\delta,A_{\mathtt{VS}}) = b_1 + b_2 + b_3 + b_4 + b_2b_4$.

In the example above, the polynomial produced by $\widehat{\mathsf{N}}_{\mathsf{sum}}\langle b\rangle$ does not have many product terms that are in the polynomial produced by $\mathsf{N}_{\mathsf{sum}}\langle b\rangle$ in Example 5. This has the potential to substantially reduce the computed bounds. Indeed, when we ran $\widehat{\mathsf{N}}_{\mathsf{sum}}$ on IPC benchmarks it yields an improvement in the computed bounds compared to $\mathsf{N}_{\mathsf{sum}}$. However, the improvement was in a limited number of domains, namely, in instances from the domains ParcPrinter, unsolvable Diagnosis, and Satellite.

5.2.1 Soundness of Using $\widehat{\mathsf{N}}_{\mathsf{sum}}$. The statement showing the soundness of $\widehat{\mathsf{N}}_{\mathsf{sum}}$ as a bound is the following.

Theorem 7. *For any factored representation* δ *with an acyclic lifted dependency graph* $A_{\mathtt{VS}}$, $d(\delta)\leq\widehat{\mathsf{N}}_{\mathsf{sum}}\langle rd\rangle(\delta,A_{\mathtt{VS}})$.

To prove the soundness of $\widehat{\mathsf{N}}_{\mathsf{sum}}$ we firstly define the following function, that intuitively returns the number of changes in the assignments of a given set of state variables during the execution of an action sequence.

Definition 18 (Subsystem Trace). *For a state* x*, action sequence* $\vec{\pi}$*, and set of variables* vs*, let* $\partial(x,\vec{\pi},vs)$ *be:*

$$\begin{aligned}\partial(x,[],vs) &= [] \\ \partial(x,\pi::\vec{\pi},vs) &= \begin{cases} x'::\partial(x',\vec{\pi},vs) & \text{if } x\downarrow_{vs}\neq x'\downarrow_{vs} \\ \partial(x',\vec{\pi},vs) & \text{otherwise}\end{cases}\end{aligned}$$

where $x'=\pi(x)$.

The following proposition follows from Definition 18 and the fact that no state can be traversed twice in an acyclic system.

Proposition 1. *For a factored system* δ*, if* $\delta\downarrow_{vs}$ *is acyclic, then for* $x\in\mathbb{U}(\delta)$ *and* $\vec{\pi}\in\delta^*$*, we have* $\partial(x,\vec{\pi},vs)\leq rd(\delta\downarrow_{vs})$.

Proof of Theorem 7. The proof of this theorem is a modified version of that of Lemma 1. Thus, we perform a similar induction on the structure of $A_{\mathtt{VS}}$, and consider every set of variables $P\in A_{\mathtt{VS}}$ and each of its children $C\in\mathsf{children}_{A_{\mathtt{VS}}}(P)$. The treatment of the C-actions is exactly as in the proof of Lemma 1, where we remove redundant actions from $\vec{\pi}\downarrow_C$, and restitch the resulting sequence back in $\vec{\pi}$. For P, if the projection $\delta\downarrow_P$ does not have an acyclic state space, we proceed exactly as in the proof of Lemma 1. However, if the projection $\delta\downarrow_P$ has an acyclic state space, from Proposition 1 we do know that at most $rd(\delta\downarrow_P)$ P-actions in $\vec{\pi}$ cause any change in the states. Accordingly, instead of considering every fragment of $\vec{\pi}$ that has only P-actions and shortening it to be $\ell(\delta\downarrow_P)$ actions as we do

in Lemma 1, we remove the P-actions from $\vec{\pi}$ that do not cause change in the state, which leaves us with at most $rd(\delta\lfloor_P)$ remaining P-actions. This completes our proof. □

5.2.2 Theoretical Guarantees on Using $\widehat{\mathsf{N}}_{\mathsf{sum}}$. We now prove a theorem of the same style of Theorem 3 for $\widehat{\mathsf{N}}_{\mathsf{sum}}$. Here, we show the tightness of the bounds computed by $\widehat{\mathsf{N}}_{\mathsf{sum}}$ w.r.t. more features than in the case of $\mathsf{N}_{\mathsf{sum}}$. In particular, we show that $\widehat{\mathsf{N}}_{\mathsf{sum}}$ produces diameter bounds that are at least as tight as those computed by any sound bounding algorithm that takes as input the features: (i) the recurrence diameters of projections computed using some acyclic lifted dependency graph, (ii) the dependencies between projections, and (iii) one bit per projection indicating whether it has an acyclic state space.

THEOREM 8. *Let $\mathbb{B} \times \mathbb{N}$ be the set of pairs where the first member of each pair is either 1 or 0 and the second member is a natural number. Let $A_{\mathbb{B}\times\mathbb{N}}$ be a $\mathbb{B} \times \mathbb{N}$-DAG (a DAG labelled with pairs of type $\mathbb{B} \times \mathbb{N}$). There is a factored system δ with a lifted dependency DAG, A_{VS}, where:*

(i) $\widehat{\mathsf{N}}_{\mathsf{sum}}\langle rd\rangle(\delta, A_{\mathsf{VS}}) \leq d(\delta)$, and

(ii) Let $\mathfrak{A} : \mathsf{VS} \Rightarrow \mathbb{B} \times \mathbb{N}$ be a function where, for every $vs \subseteq \mathcal{D}(\delta)$, $\mathfrak{A}(vs) = (1, rd(\delta\lfloor_{vs}))$ if $\delta\lfloor_{vs}$ is acyclic, and $\mathfrak{A}(vs) = (0, rd(\delta\lfloor_{vs}))$ otherwise. We have $A_{\mathbb{B}\times\mathbb{N}} = \mathfrak{A}(\!|A_{\mathsf{VS}}|\!)$. I.e. $A_{\mathbb{B}\times\mathbb{N}}$ is a relabelling of A_{VS}: a vertex labelled by vs in A_{VS} is relabelled by a pair the first member of which is 1 if $\delta\lfloor_{vs}$ has an acyclic state space and 0 otherwise, and its second member is the recurrence diameter of $\delta\lfloor_{vs}$.

PROOF. The proof for this theorem is modified version of the proof for Theorem 3. First, let for every vertex $u_1 \in V(A_{\mathbb{B}\times\mathbb{N}})$ the pair (b_{u_1}, n_{u_1}) denote the label of u_1. We amend the construction from Theorem 3 in two ways. First, in the original construction we change the natural number $A_{\mathbb{N}}(u_1)$ that labels every vertex u_1 in the given DAG to be an inverted flower factored system $\mathbb{f}^{u_1}_{p_{u_1}, A_{\mathbb{N}}(u_1)}$. Here every vertex $u_1 \in V(A_{\mathbb{B}\times\mathbb{N}})$ is labelled with a pair (b_{u_1}, n_{u_1}), so we change that label from the pair to a system δ^{u_1}, as follows: if $b = 1$ then the label δ^{u_1} is a path factored system $\gamma^{u_1}_n$[6], otherwise the label δ^{u_1} is an inverted flower $\mathbb{f}^{u_1}_{p_{u_1}, n_{u_1}}$ like the one used in the original construction.

The second modification is in the way we construct the witness action sequence. Recall that in the proof of Theorem 3, for every vertex u_1 in the given DAG, step 7 in Algorithm 2 adds an action sequence from the system labelling every parent of u_1 between every two actions from $\mathbb{f}^{u_1}$. This added action sequence has length that is equal to the recurrence diameter of the respective parent. The intuition was to make sure that the constructed action sequence would have as many action sequences from a parent as the sum of the recurrence diameters of each of its children. This was done to match the bound computed by $\mathsf{N}_{\mathsf{sum}}$ which has a product term of the parent's and the total children recurrence diameters. Here we use Algorithm 3, where the way actions from the parents are added depends on whether the parent is acyclic or not. Algorithm 3 behaves like Algorithm 2 in case the parent's state space is not acyclic. On the other hand, if the parent is acyclic Algorithm 3 adds a parent action sequence only once, and then preconditions are added to all child actions ensuring that they can only execute after the parent sequence has successfully executed. This matches the bound computed by $\widehat{\mathsf{N}}_{\mathsf{sum}}$, which only has one term without multiplications for a parent's recurrence diameter in the resulting polynomial if the parent's state space is acyclic. □

Algorithm 3:

$\vec{\pi} := []$; $\delta := \bigcup_{u_1 \in V(A_{\mathbb{N}})}(\mathbb{f}^{u_1})$
for $u_1 \in l_{\mathbb{N}}$ /* Loop over $V(A_{\mathbb{N}})$ in reverse topological order*/
  if $b_{u_1} = 1$ /* If the projection acyclic */
    $\vec{\pi} := \vec{\gamma}^{u_1} \frown \vec{\pi}$
  $i := A_{\mathbb{N}}(u_1)$; $\mathsf{children} := \bigcup_{u_2 \in \mathsf{children}_{A_{\mathbb{N}}}(u_1)} \delta^{u_2}$
  for $(p, e) \in \vec{\pi}$ /*Loop over every action in $\vec{\pi}$, in execution order */
    if $\mathcal{D}(e) \cap \mathcal{D}(\mathsf{children}) \neq \emptyset$ /* check if the current action belongs to a child of u_1 */
      $\delta := \delta \setminus \{(p, e)\}$; $\delta := \delta \cup \{(x^{u_1}_{i+1} \uplus p, e)\}$ /* Add preconditions to (p, e) ensure it needs $\vec{\pi}^{u_1}_{i\mapsto i+1}$*/
      if $b_{u_1} = 1$ /* If the vertex is relabelled by a path*/
        $\vec{\pi} := \mathsf{pfx}(\vec{\pi}, (p, e)) \frown (x^{u_1}_{i+1} \uplus p, e) :: \mathsf{sfx}(\vec{\pi}, (p, e))$ /* Add precondition to (p,e)*/
      else /* If the vertex is relabelled by an inverted flower*/
        $\vec{\pi} := \mathsf{pfx}(\vec{\pi}, (p, e)) \frown \vec{\pi}^{u_1}_{i\mapsto i+1} \frown (x^{u_1}_{i+1} \uplus p, e) :: \mathsf{sfx}(\vec{\pi}, (p, e))$ /* Add $\vec{\pi}^{u_1}_{i\mapsto i+1}$ before (p, e)*/
    $i := i + 1$ /* Increment the petal counter*/
  $\vec{\pi} := \vec{\pi} \frown \vec{\pi}^{u_1}_{i\mapsto i+1}$ /* Add an action sequence to visit one more petal*/

[6]Recall that paths were defined before the proof of Theorem 5

$$(\{\mathtt{lk}_{1,1}\},\{\mathtt{gk}_{1,2},\mathtt{lk}_{1,2},\overline{\mathtt{lk}_{1,1}},\overline{\mathtt{s}_1}\}),(\{\mathtt{lk}_{1,2}\},\{\mathtt{gk}_{1,3},\mathtt{lk}_{1,3},\overline{\mathtt{lk}_{1,2}},\overline{\mathtt{s}_1}\}),$$
$$(\{\mathtt{lk}_{1,1}\},\{\mathtt{gk}_{2,2},\mathtt{lk}_{1,2},\overline{\mathtt{lk}_{1,1}},\overline{\mathtt{s}_1}\}),(\{\mathtt{lk}_{1,2}\},\{\mathtt{gk}_{2,3},\mathtt{lk}_{1,3},\overline{\mathtt{lk}_{1,2}},\overline{\mathtt{s}_1}\}),$$
$$(\{\mathtt{gk}_{1,2}\},\{\mathtt{s}_1\}),(\{\mathtt{gk}_{2,2}\},\{\mathtt{s}_1\})$$

Fig. 13. The *snapshot* of $\delta\downarrow_{\text{ROOM1}}$ on $CK_{1,2}$, an abstraction that only analyses the changes related to room 1 when its door recognises key 2 as the current key.

5.2.3 Discussion. Although experimentally not much better than $\mathsf{N}_{\mathsf{sum}}$, $\widehat{\mathsf{N}}_{\mathsf{sum}}$ is an example of an algorithm that can produce tighter bounds than $\mathsf{N}_{\mathsf{sum}}$ by analysing more features than those analysed by $\mathsf{N}_{\mathsf{sum}}$. This shows more clearly in the statement of Theorem 8: it states that $\widehat{\mathsf{N}}_{\mathsf{sum}}$ is tight w.r.t. algorithms taking "more information" as input than that taken by $\mathsf{N}_{\mathsf{sum}}$. In particular, $\widehat{\mathsf{N}}_{\mathsf{sum}}$ uses an extra bit per dependency graph vertex, which indicates whether the projection is acyclic or not, in addition to the dependencies between projections and the projections' recurrence diameters that are used as sole input to $\mathsf{N}_{\mathsf{sum}}$. We again note that formulating the statement of the tightness theorem was a substantial challenge since it included figuring out what part of the algorithm's input is relevant. For example, $\widehat{\mathsf{N}}_{\mathsf{sum}}$ and $\mathsf{N}_{\mathsf{sum}}$ may seem to take the same input, at least when it comes to their signatures as functions. But as is shown in the tightness statements for both algorithms, $\widehat{\mathsf{N}}_{\mathsf{sum}}$ uses more information than $\mathsf{N}_{\mathsf{sum}}$ and accordingly is capable to produce tighter bounds.

5.3 Compositional Bounding with Snapshots

Although (at least theoretically) $\widehat{\mathsf{N}}_{\mathsf{sum}}$ has the potential of computing substantially tighter bounds than $\mathsf{N}_{\mathsf{sum}}$, in practice we found that the gains in terms of tightness of bounds were in a very limited set of domains. Recall that $\widehat{\mathsf{N}}_{\mathsf{sum}}$ only produces bounds better that $\mathsf{N}_{\mathsf{sum}}$ in the case there is at least one projection with an acyclic state space from the set of projections induced by the acyclic dependency graph. However, experiments seem to imply that it is not a very common scenario that the entire state space of a projection induced by the dependency graph is acyclic. Accordingly we need a more fine-grained analysis of state space acyclicity that can be applied to projections smaller than those induced by the dependency graph. In this section we introduce a different approach that is capable of exploiting state space acyclicity in any projection, regardless of whether that projection is induced by a dependency graph.

To formulate this new state space acyclicity analysis we introduce a new abstraction, to which we refer as *snapshotting*. Snapshots enable the exploitation of state space acyclicity to be independent of variable dependencies, i.e. any projection with an acyclic state space can be exploited by snapshotting. A snapshot models the system when we fix assignments to a subset of the state variables, where we remove actions whose preconditions *or* effects contradict those assignments.

Definition 19 (Snapshot). *For states x and x', let* $\mathsf{agree}(x,x')$ *denote $|\mathcal{D}(x)\cap\mathcal{D}(x')| = |x\cap x'|$, i.e. a variable that is in the domains of both x and x' has the same assignment in x and x'. For δ and a state x, the snapshot of δ at x is*

$$\delta\downarrow_{\restriction x} \equiv \{(p,e) \mid (p,e)\in\delta \wedge \mathsf{agree}(p,x)\wedge\mathsf{agree}(e,x)\}\downarrow_{\mathcal{D}(\delta)\setminus\mathcal{D}(x)}$$

Graphically, a snapshot of a digraph model of a concrete system, is a digraph that this is obtained by deleting certain edges and vertices.

Example 14. $\delta\downarrow_{\text{ROOM1}}\downarrow_{\restriction \mathtt{CK}_{1,2}}$ is shown in Figure 13.

We now investigate how snapshotting can be used for bounding. Let b be an arbitrary bounding function that satisfies $d(\delta)\leq b(\delta)$ for any δ. Consider a system δ where for some variables vs we have that $\delta\downarrow_{vs}$ is acyclic—i.e. the state space of $\delta\downarrow_{vs}$ forms a directed acyclic graph (DAG). In that case, we have that $d(\delta)\leq \mathsf{S}_{\mathsf{max}}\langle b\rangle(vs,\delta)$, where $\mathsf{S}_{\mathsf{max}}$ is a compositional bounding function defined as follows.[7]

Definition 20 (Acyclic System Compositional Bound). *Letting* $\mathsf{succ}(x,\delta)\equiv\{x' \mid \exists\pi\in\delta.\pi(x)=x'\}$, S *is*

$$\mathsf{S}\langle b\rangle(x,vs,\delta) = b(\delta\downarrow_{\restriction x}) + \max_{x'\in\mathsf{succ}(x,\delta\downarrow_{vs})}(\mathsf{S}\langle b\rangle(x',vs,\delta)+1)$$

Then, let $\mathsf{S}_{\mathsf{max}}\langle b\rangle(vs,\delta) = \max_{x\in\mathbb{U}(\delta\downarrow_{vs})}\mathsf{S}\langle b\rangle(x,vs,\delta)$.

Theorem 9. *If $\delta\downarrow_{vs}$ is acyclic and b bounds d, then $d(\delta)\leq\mathsf{S}_{\mathsf{max}}\langle b\rangle(vs,\delta)$.*

S is only well-defined if $\delta\downarrow_{vs}$ is acyclic. We only seek to consider and interpret $\mathsf{S}_{\mathsf{max}}$ in systems $\delta\downarrow_{vs}$ where no execution can visit a state more than once. In that situation $\mathsf{S}_{\mathsf{max}}$ calculates the maximal cost of a traversal through the DAG formed by the state space of $\delta\downarrow_{vs}$. Completing that intuition, take the cost of visiting a state x to be $b(\delta\downarrow_{\restriction x})$, and

[7]We postpone the proof of the validity of the function $\mathsf{S}_{\mathsf{max}}$ as a bound until the next section.

Fig. 14. The dependency graphs of snapshots of the hotel-key system that we use for illustrative purposes in the examples.

the cost of traversing an edge between states to be 1. These ideas are made concrete below, in Example 15. Also, since $\mathsf{S}_{\mathsf{max}}$ follows the scheme of an algorithm that finds the length of the longest path in a DAG, the run-time of a straightforward implementation of it is linear in the size of the state space of $\delta\downarrow_{vs}$ and the complexity of computing b.

Example 15. Since $\delta\downarrow_{\mathtt{ck}_1}$ is acyclic, and $\mathtt{CK}_{1,i} \in \mathbb{U}(\delta\downarrow_{\mathtt{ck}_1})$, then $\mathsf{S}\langle d\rangle(\mathtt{CK}_{1,i}, \mathtt{ck}_1, \delta)$ is well-defined for $i \in \{1,2,3\}$. Denoting $d(\delta\downarrow_{\mathtt{CK}_{1,i}})$ with $d_{1,i}$ and $\mathsf{S}\langle d\rangle(\mathtt{CK}_{1,i}, \mathtt{ck}_1, \delta)$ with $\mathsf{S}_{1,i}$, we have $\mathsf{S}_{1,3} = d_{1,3}$ because $\mathsf{succ}(\mathtt{CK}_{1,3}, \delta\downarrow_{\mathtt{ck}_1}) = \emptyset$. We also have $\mathsf{S}_{1,2} = d_{1,2}+1+S_{1,3} = d_{1,2}+1+d_{1,3}$ and $\mathsf{S}_{1,1} = d_{1,1}+1+S_{1,2} = d_{1,1}+1+d_{1,2}+1+d_{1,3} = d_{1,1}+d_{1,2}+d_{1,3}+2$.

It should be clear from the statement of Theorem 9 that $\mathsf{S}_{\mathsf{max}}$ enables the exploitation of state space acyclicity for projections on arbitrary sets of variables. This is in contrast to $\widehat{\mathsf{N}}_{\mathsf{sum}}$ which requires the projections to be on sets of variables coming from an acyclic dependency graph. For instance, since the dependency graph of $\delta\downarrow_{\textsc{Room1}}$ is comprised of one SCC (shown on the left in Figure 14a), then $\widehat{\mathsf{N}}_{\mathsf{sum}}$ cannot decompose it any further. Also, since the state space of $\delta\downarrow_{\textsc{Room1}}$ is not acyclic, then no state space acyclicity will be exploited by $\widehat{\mathsf{N}}_{\mathsf{sum}}$ in that example. Nonetheless, when we project $\delta\downarrow_{\textsc{Room1}}$ on $\mathtt{CK}_{1,2}$, as shown in the example above, the state space of the projection is acyclic and accordingly $\mathsf{S}_{\mathsf{max}}$ could decompose $\delta\downarrow_{\textsc{Room1}}$.

Although Theorem 9 suggests the possibility of compositional upper-bounding of the diameter given the presence of acyclicity in a transition system's state space, it does not give a concrete algorithm. For instance, it does not specify how can one find projections with acyclic state spaces. Furthermore, it does not suggest whether one should apply acyclic state space decomposition recursively, and if yes how can that be done. Thus we investigate building a practical compositional algorithm to upper-bound the diameter based on Theorem 9. One straightforward approach is given by Algorithm 4, which recursively applies state space acyclicity analysis based on $\mathsf{S}_{\mathsf{max}}$ to snapshots.

Algorithm 4: $\textsc{Pur}(\delta)$

$S = \mathsf{ch}(\{\mathsf{S}_{\mathsf{max}}\langle\textsc{Pur}\rangle(vs, \delta) \mid vs \in \Omega(\delta)\} \cup \infty)$
if $S = \infty$ **return** $\textsc{Exp}(\delta)$ **else return** S

In Pur, Ω is an oracle that returns a set of strict subsets of $\mathcal{D}(\delta)$, where $\forall vs \in \Omega(\delta).\delta\downarrow_{vs}$ is acyclic, and ch is a function that chooses a member from a given set. Pur terminates because a snapshot has fewer variables than the concrete system. In Pur the function Exp provides upper bounds for the diameters of "base-case" systems—i.e. problems that cannot be further decomposed based on state space acyclicity. Given this assumption and Theorem 9, Pur itself computes a valid upper bound on the diameter of the concrete system.

A main question for a practical implementation of Pur is the choice of Ω. The trivial choice of all strict subsets of $\mathcal{D}(\delta)$ is impractical. A pragmatic solution, is to take the situation that elements in $\mathcal{D}(\delta)$ correspond to individual state-variable assignments in the SAS^+ description of a planning problem. Here, we have adopted that pragmatic solution.

The SAS^+ formalism was described and studied by Bäckström and Nebel [11]. A SAS^+ description of a system is distinguished from equivalent propositional descriptions by the fact that state-variables are finite-domain variables. For example, for a system modelling a logistics scenario the SAS^+ description has one state variable for each vehicle. The assignment to such a variable determines the location of a vehicle in a state. Thus, whereas in a STRIPS description we would have propositions for each truck and city of the form "Truck $t1$ is in Prague", "truck $t1$ is in Budapest", etc., in the SAS^+ description we have a multi-valued state variable, "location of $t1$", whose assignment in a state is to one particular city. The SAS^+ description with multi-valued state-variables makes explicit the cliques in the *invariant* relationships between propositions. Specifically, it is clear from the SAS^+ description of a logistics scenario that a truck cannot be in two locations at once. Additionally, a SAS^+ description exposes the *domain transition graphs* (DTGs), informative problem structures that were first described by Helmert [36]. Associated with each SAS^+ state-variable is a DTG. This is a directed graph whose vertices are in one-to-one correspondence with legal assignments to the state-variable associated with the graph. The DTG has a directed edge between vertices v and u iff there is an action $\pi = (p, e)$ so that p satisfies the assignment indicated by v, and e the assignment indicated by u

For example, in a logistics scenario we could have the action "Drive $t1$ from Prague to Budapest", and therefore would have a DTG with a directed arc between the assignments "location of $t1$ Prague" and "location of $t1$ Budapest". In our work, the SAS^+ problem description is either given directly, or otherwise generated using preprocessing modules from the Fast-Downward planning system [37]. Each element in $\Omega(\delta)$ then corresponds to a set of elements from $\mathcal{D}(\delta)$ that model one SAS^+ variable whose DTG is acyclic.

Another question about the implementation of PUR is the actual implementation of the function ch. One could for example use the function min which would return the smallest member in a set of natural numbers. This would effectively mean that PUR tries all the possible orders at which the sets of variables in $\Omega(\delta)$ can be used for computing snapshots. However, this will cause the run-time of PUR to be intractable. If we assign ch to min, for a full evaluation, $\mathsf{S}_{\mathsf{max}}$ is recursively called as many as $|\Omega(\delta)|!$ times. In practice we only evaluate $\mathsf{S}_{\mathsf{max}}$ on one *arbitrarily* chosen element from $\Omega(\delta)$ by assigning ch to a function that arbitrarily chooses an element from a given set. Our experimentation never uncovered a problem where using min, where computationally feasible, produced a better bound.

A second more fundamental source of computational expense comes from the definition of $\mathsf{S}_{\mathsf{max}}$: PUR can be recursively called a number of times that is linear in the size of the state space of δ. This happens if $\Omega(\delta)$ is a partition of $\mathcal{D}(\delta)$. Although this worst case scenario is contrived, in practice $\Omega(\delta)$ can cover sufficient elements from $\mathcal{D}(\delta)$ to render PUR impractical. This is demonstrated in the following example.

Example 16. Taking $\mathcal{D}(\delta)$ associated with the hotel key protocol example, the Fast-Downward preprocessor [37] identifies partition:

$$\left\{\begin{array}{l}\mathtt{ck}_1, \mathtt{ck}_2, \mathtt{lk}_1, \mathtt{lk}_2,\\ \{\mathtt{gk}_{1,2}\}, \{\mathtt{gk}_{1,3}\}, \{\mathtt{gk}_{1,5}\}, \{\mathtt{gk}_{1,6}\},\\ \{\mathtt{gk}_{2,2}\}, \{\mathtt{gk}_{2,3}\}, \{\mathtt{gk}_{2,5}\}, \{\mathtt{gk}_{2,6}\},\\ \{\mathtt{s}_1\}, \{\mathtt{s}_2\}\end{array}\right\}$$

as SAS+ variable assignments. Let $\Omega(\delta)$ denote that set, excluding $\{\mathtt{s}_1\}$ and $\{\mathtt{s}_2\}$. Note, $\forall vs \in \Omega(\delta)$ we have that $\delta\lfloor_{vs}$ is acyclic. Consequently, we have that PUR(δ) evaluates after $\prod_{vs\in\Omega(\delta)}|vs|$,i.e. $3^4 = 81$, calls to $\mathsf{S}_{\mathsf{max}}$.

The example above shows that the worst case run-time of $\mathsf{S}_{\mathsf{max}}$ is linear in the size of the space, even if memoisation is used, i.e. exponential in the size of the given factored system. This worst case run-time would be very limiting if encountered in practical examples, especially compared to variable dependency based bounding approaches whose worst case run-times are linear in the number of state variables.

We ran PUR on standard International Planning Competition problems to compare its state space based decomposition approach with the dependency graph based decomposition of $\mathsf{N}_{\mathsf{sum}}$. First, we measure the degree of decomposition provided by either approach. We do so by comparing the size of the domain of the concrete system—i.e. $|\mathcal{D}(\delta)|$—with that of the largest abstraction computed by either approach for the concrete problem, i.e. the largest base-case.

As shown in Figure 15, our experiments show that the scenario depicted in Example 15 happens in many cases. The state space based decomposition of $\mathsf{S}_{\mathsf{max}}$ is able to decompose problems more than what is attainable using the variable dependency based decomposition of $\mathsf{N}_{\mathsf{sum}}$ or $\widehat{\mathsf{N}}_{\mathsf{sum}}$. This indicates that in some sense many of those domains have "more" acyclicity in their state spaces than their dependency graphs. Examples of those domains are TPP, NOMYSTERY, ROVER, VISITALL, HIKING, SCANALYZER, and CHESSBOARD.

On the other hand, our experiments also show that $\mathsf{S}_{\mathsf{max}}$ encounters the intractability demonstrated in Example 16 in many benchmarks. We ran the bounding algorithms on a uniform cluster with time and memory limits of 30minutes and 8GB, respectively. In those experiments we set the base-case function EXP to return the size of the domain of the SAS+ description of the given problem. Table 1 shows the results of those bounding experiments. It shows that $\mathsf{S}_{\mathsf{max}}$ was outperformed by $\mathsf{N}_{\mathsf{sum}}$ in computing bounds, even in most of the domains where it was able to compute abstractions that are substantially smaller than those computed by $\mathsf{N}_{\mathsf{sum}}$.

5.3.1 Soundness of Using $\mathsf{S}_{\mathsf{max}}$. After showing the utility of using state space acyclicity, we now prove the soundness of using $\mathsf{S}_{\mathsf{max}}$ to compute compositional bounds for different topological properties. We start by analysing its soundness as a bound on the diameter.

PROPOSITION 2. *For any δ, vs, and x, if $\delta\lfloor_{vs}$ is acyclic, $x \in \mathbb{U}(\delta\lfloor_{vs})$, and $x' \in \mathsf{succ}(x, \delta\lfloor_{vs})$, then $b(\delta{\restriction}x) + 1 + \mathsf{S}\langle b\rangle(x', vs, \delta) \leq \mathsf{S}\langle b\rangle(x, vs, \delta)$, for a base-case function b.*

PROPOSITION 3. *For any x, $\overrightarrow{\pi}$, and vs, if $\partial(x, \overrightarrow{\pi}, vs) = []$ then: (i) $x\lfloor_{vs} = \overrightarrow{\pi}(x)\lfloor_{vs}$ and (ii) there is $\overrightarrow{\pi}'$ where $\overrightarrow{\pi}(x) = \overrightarrow{\pi}'(x)$ and $|\overrightarrow{\pi}'| \leq d(\delta{\restriction}_{x\lfloor_{vs}})$.*

PROPOSITION 4. *For any x, x', $\overrightarrow{x}$, vs, and $\overrightarrow{\pi}$, if $\partial(x, \overrightarrow{\pi}, vs) = x' :: \overrightarrow{x}$, then there are $\overrightarrow{\pi}_1$, π, and $\overrightarrow{\pi}_2$ such that (i) $\overrightarrow{\pi} = \overrightarrow{\pi}_1 \frown \pi :: \overrightarrow{\pi}_2$, (ii) $\partial(x, \overrightarrow{\pi}_1, vs) = []$, (iii) $\pi(\overrightarrow{\pi}_1(x)) = x'$, and (iv) $\overrightarrow{\pi}_2(x') = \overrightarrow{\pi}(x)$.*

	Msum min/max(#instances)	Nsum min/max(#instances)	Smax min/max(#instances)	HYB min/max(#instances)
newopen (1440)	1,058e4 / 1,938e9 (1126)	5,961e3 / 1,963e9 (1176)	——— / ——— (0)	3,791e3 / 7,652e7 (1293)
hotelKey (1000)	3,100e1 / 1,812e9 (451)	3,100e1 / 1,678e9 (455)	1,500e1 / 4,194e6 (56)	7,000e0 / 1,981e3 (900)
logistics (407)	8,200e1 / 3,685e6 (406)	7,400e1 / 3,652e6 (406)	1,024e3 / 2,684e8 (51)	1,520e2 / 4,163e6 (406)
elevators (210)	7,000e0 / 1,397e9 (164)	7,000e0 / 1,397e9 (164)	8,000e0 / 6,585e8 (66)	8,000e0 / 6,585e8 (162)
rover (182)	2,240e2 / 1,968e9 (79)	1,230e2 / 1,833e9 (87)	1,540e2 / 4,014e6 (25)	1,540e2 / 3,021e8 (97)
nomystery (124)	2,500e1 / 3,290e6 (124)	1,900e1 / 3,148e6 (124)	3,600e1 / 4,991e8 (25)	1,300e1 / 1,496e5 (124)
zeno (50)	4,100e1 / 5,178e5 (50)	4,100e1 / 4,987e5 (50)	4,200e1 / 7,530e6 (17)	4,200e1 / 5,212e5 (50)
hiking (40)	1,457e3 / 1,451e9 (22)	1,457e3 / 1,451e9 (22)	1,458e3 / 2,239e8 (18)	1,458e3 / 2,239e8 (18)
TPP (120)	3,100e1 / 1,481e9 (23)	3,100e1 / 1,481e9 (23)	2,100e1 / 1,258e7 (8)	1,700e1 / 5,250e8 (21)
Transport (197)	3,750e5 / 1,547e9 (14)	3,750e5 / 1,547e9 (14)	1,501e6 / 7,581e8 (9)	4,250e5 / 7,581e8 (13)
GED (40)	7,465e6 / 6,676e8 (5)	7,465e6 / 6,676e8 (5)	7,465e6 / 6,676e8 (5)	7,465e6 / 6,676e8 (5)
woodworking (60)	1,890e2 / 1,890e2 (2)	1,890e2 / 1,890e2 (2)	2,621e5 / 2,621e5 (2)	1,920e2 / 6,169e8 (13)
visitall (90)	7,000e0 / 4,194e8 (16)	7,000e0 / 4,194e8 (16)	8,000e0 / 2,753e6 (12)	8,000e0 / 2,753e6 (12)
bottleneck (50)	5,120e6 / 4,247e7 (6)	5,120e6 / 4,247e7 (6)	5,110e2 / 1,311e5 (10)	5,110e2 / 1,311e5 (10)
openstacks (131)	2,814e5 / 6,651e8 (7)	2,799e5 / 6,651e8 (7)	——— / ——— (0)	7,776e4 / 7,776e4 (6)
3unsat (30)	1,342e8 / 1,342e8 (5)	1,342e8 / 1,342e8 (5)	1,023e3 / 6,554e4 (5)	3,100e1 / 3,277e4 (15)
tiles (45)	3,874e8 / 3,874e8 (23)	3,874e8 / 3,874e8 (23)	3,874e8 / 3,874e8 (23)	3,874e8 / 3,874e8 (23)
satellite (10)	1,032e3 / 1,228e4 (10)	5,960e2 / 6,326e3 (10)	2,048e4 / 1,153e7 (10)	7,520e2 / 7,118e3 (10)
hyp (286)	4,384e5 / 5,733e8 (8)	4,018e5 / 5,733e8 (8)	——— / ——— (0)	4,894e3 / 1,367e5 (7)
scanalyzer (60)	4,095e3 / 2,986e6 (3)	4,095e3 / 2,986e6 (3)	1,551e3 / 7,466e5 (3)	1,551e3 / 7,466e5 (3)
sliding (25)	3,874e8 / 3,874e8 (13)	3,874e8 / 3,874e8 (13)	3,874e8 / 3,874e8 (13)	3,874e8 / 3,874e8 (13)
gripper (54)	1,100e1 / 1,796e8 (7)	1,100e1 / 1,796e8 (7)	1,200e1 / 1,796e8 (7)	1,200e1 / 1,796e8 (7)
storage (30)	1,430e2 / 8,192e7 (7)	1,430e2 / 8,192e7 (7)	1,440e2 / 8,192e7 (7)	1,440e2 / 8,192e7 (7)
trucks (34)	5,360e2 / 9,175e7 (4)	4,910e2 / 9,175e7 (4)	2,490e6 / 4,260e7 (2)	9,090e2 / 3,974e8 (6)
parcprinter (60)	1,888e7 / 5,782e7 (3)	1,887e7 / 5,780e7 (3)	——— / ——— (0)	5,962e5 / 3,249e8 (2)
maintenance (5)	6,554e4 / 2,621e5 (3)	6,554e4 / 2,621e5 (3)	1,638e4 / 2,621e5 (3)	5,100e1 / 1,647e3 (5)
pipesworld (101)	——— / ——— (0)	——— / ——— (0)	5,369e8 / 5,369e8 (2)	5,369e8 / 5,369e8 (2)
pegsol (133)	1,270e2 / 1,678e7 (2)	1,270e2 / 1,678e7 (2)	1,280e2 / 1,678e7 (2)	1,280e2 / 1,678e7 (2)
chessboard (23)	1,678e7 / 1,678e7 (1)	1,678e7 / 1,678e7 (1)	1,258e7 / 1,258e7 (1)	1,573e6 / 1,573e6 (1)
blocksworld (10)	1,023e3 / 5,369e8 (5)	1,023e3 / 5,369e8 (5)	1,024e3 / 5,369e8 (5)	1,024e3 / 5,369e8 (5)

Table 1. A table showing statistics on instances for which the different bounding algorithms successfully computed bounds that are less than 10^9 within 30 minutes. Every cell in the table has the minimum bound for an IPC domain, the maximum bounds for the IPC domain, and the number of instance successfully bounded from that domain in the parentheses.

Proposition 5. *For any x, $\vec{\pi}_1$, $\vec{\pi}_2$, and vs, we have that* $\partial(x, \vec{\pi}_1 \frown \vec{\pi}_2, vs) = \partial(x, \vec{\pi}_1, vs) \frown \partial(\vec{\pi}_1(x), \vec{\pi}_2, vs)$.

Lemma 2. *For any δ and vs where $\delta\lfloor_{vs}$ is acyclic, $x \in \mathbb{U}(\delta)$, and $\vec{\pi} \in \delta^*$, there is $\vec{\pi}'$ such that $\vec{\pi}(x) = \vec{\pi}'(x)$ and $|\vec{\pi}'| \leq \mathsf{S}\langle d\rangle(x\lfloor_{vs}, vs, \delta)$.*[8]

Proof. The proof is by induction on $\partial(x, \vec{\pi})$. The base case where $\partial(x, \vec{\pi}) = []$ is trivial. In the step case we have that $\partial(x, \vec{\pi}) = x' :: \vec{x}$ and the induction hypothesis: for any $x_{\mathsf{IH}} \in \mathbb{U}(\delta)$, and $\vec{\pi}_{\mathsf{IH}} \in \delta^*$ if $\partial(x_{\mathsf{IH}}, \vec{\pi}_{\mathsf{IH}}) = \vec{x}$ then there is $\vec{\pi}_{\mathsf{IH}}{}'$ where $\vec{\pi}_{\mathsf{IH}}(x_{\mathsf{IH}}) = \vec{\pi}_{\mathsf{IH}}{}'(x_{\mathsf{IH}})$ and $|\vec{\pi}_{\mathsf{IH}}{}'| \leq \mathsf{S}\langle d\rangle(x_{\mathsf{IH}}\lfloor_{vs})$.

Since $\partial(x, \vec{\pi}) = x' :: \vec{x}$, we have $\vec{\pi}_1, \pi$ and $\vec{\pi}_2$ satisfying the conclusions of Proposition 4. Based on conclusion i, ii, and iii of Proposition 4 and Proposition 5 we have $\partial(x', \vec{\pi}_2) = \vec{x}$. Accordingly, letting x_{IH}, and $\vec{\pi}_{\mathsf{IH}}$ from the inductive hypothesis be x', and $\vec{\pi}_2$, respectively, there is $\vec{\pi}'_2$ such that $\vec{\pi}_2(x') = \vec{\pi}'_2(x)$ and $|\vec{\pi}'_2| \leq \mathsf{S}\langle d\rangle(x'\lfloor_{vs})$.†

From conclusion ii of Proposition 4 and conclusion ii of Proposition 3 there is $\vec{\pi}'_1$ where $\vec{\pi}_1(x) = \vec{\pi}'_1(x)$ and $|\vec{\pi}'_1| \leq d(\delta\restriction_{x\lfloor_{vs}})$. Letting $\vec{\pi}' = \vec{\pi}'_1 \# \pi :: \vec{\pi}'_2$, from conclusions iii and iv of Proposition 4 and † we have $\vec{\pi}(x) = \vec{\pi}'(x)$ and $|\vec{\pi}'| \leq d(\delta\restriction_{x\lfloor_{vs}}) + 1 + \mathsf{S}\langle d\rangle(x'\lfloor_{vs})$.‡

[8] In the rest of this proof we omit the vs and/or δ arguments from $\partial(,,)$ and S as they do not change.

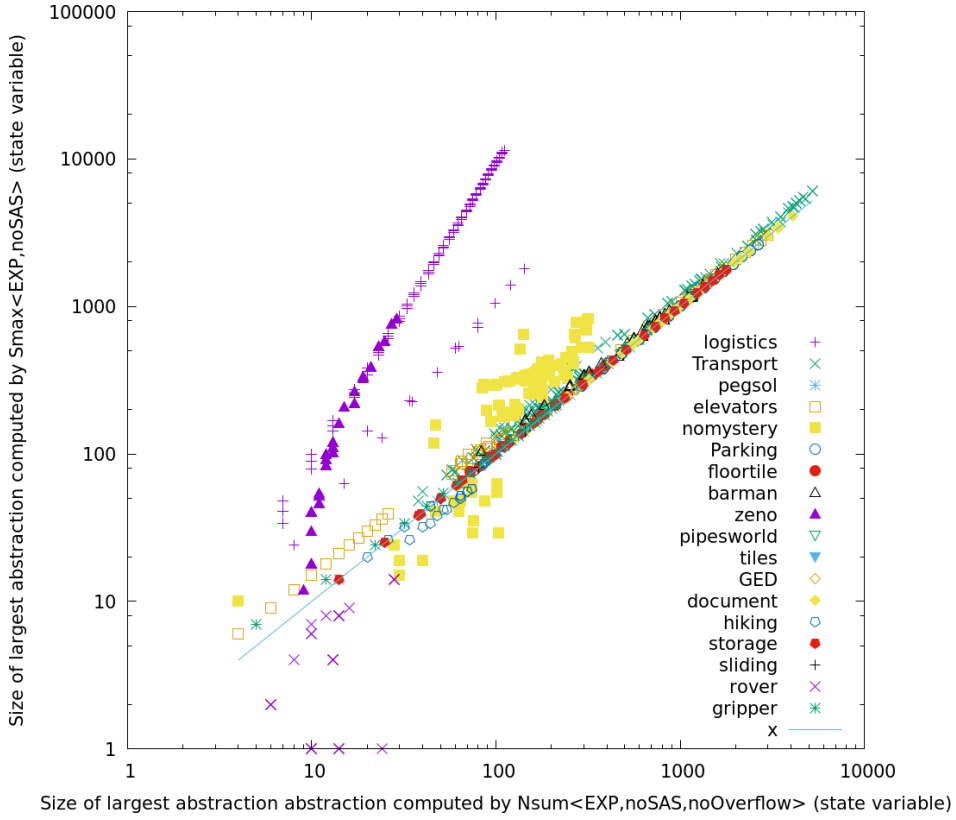


Fig. 15. Scatter plot of the size (i.e. $|\mathcal{D}(\delta)|$) of the largest base-cases computed by $\mathsf{N}_{\mathsf{sum}}$ (horizontal axis) and $\mathsf{S}_{\mathsf{max}}$ (vertical).

Lastly, from conclusion i of Proposition 3 and conclusion ii of Proposition 4 we have $x\lfloor_{vs} = \overrightarrow{\pi}_1(x)\lfloor_{vs} = \overrightarrow{\pi}'_1(x)\lfloor_{vs}$ and accordingly $\pi\lfloor_{vs}(x\lfloor_{vs}) = x'\lfloor_{vs}$. Based on that we have $x'\lfloor_{vs} \in \mathsf{succ}(x\lfloor_{vs}, \delta\lfloor_{vs})$. Then from Proposition 2 and ‡ we have $|\overrightarrow{\pi}'| \leq \mathsf{S}\langle d\rangle(x\lfloor_{vs})$. □

Theorem 9 follows from Lemma 2 and Definitions 6 and 20.

Furthermore, the following analogues of Theorem 9 for ℓ and rd can be derived.

Theorem 10. *If $\delta\lfloor_{vs}$ is acyclic and b bounds ℓ, then $\ell(\delta) \leq \mathsf{S}_{\mathsf{max}}\langle b\rangle(vs, \delta)$.*

Theorem 11. *If $\delta\lfloor_{vs}$ is acyclic and b bounds rd, then $rd(\delta) \leq \mathsf{S}_{\mathsf{max}}\langle b\rangle(vs, \delta)$.*

Both theorems above can be obtained by proofs analogous to the proof of Theorem 9, except for two modifications. Firstly, d should be replaced by ℓ in the case of Theorem 10 and rd in the case of Theorem 11. Secondly, the requirements on the constructed action sequences in Lemma 2 and Propositions 3 and 4 will be strengthened to require that they are sublists of $\overrightarrow{\pi}$ in the case of Theorem 10. In the case of Theorem 11 the constructed action sequences will be required to traverse only distinct states if executed on x.

5.3.2 $\mathsf{S}_{\mathsf{max}}$ is a Tight Bound. As is the case with $\mathsf{N}_{\mathsf{sum}}$ and $\widehat{\mathsf{N}}_{\mathsf{sum}}$, we also show a tightness result for the bounds computed by $\mathsf{S}_{\mathsf{max}}$ on the recurrence diameter, but w.r.t. different features. We show that $\mathsf{S}_{\mathsf{max}}$ produces recurrence diameter bounds that are at least as tight as those computed by any sound bounding algorithm that takes as input the features: (i) the state space of an acyclic projection of the bounded system, and (ii) the concrete system's snapshots on states of the acyclic projection. When quantifying on possible inputs to $\mathsf{S}_{\mathsf{max}}$, we encode those features as a DAG that is labelled with factored systems. The intuition is that the constructed system has the given DAG as the state space of one of its projections, and that the factored systems labelling that DAG are snapshots of the constructed system on the states of the acyclic projection. This, together with Theorem 11 shows that the $\mathsf{S}_{\mathsf{max}}$ is tight and cannot be improved for the recurrence diameter, except if it uses features more than the acyclic projection's state space and the entirety of the system's snapshots. We also show a similar tightness result for the sublist diameter.

Theorem 12. *Let Δ be the set of all finite factored systems. For any Δ-DAG, A_Δ (a DAG labelled with factored systems), there is a factored system δ and a set of variables vs where:*

(i) $\mathsf{S}_{\mathsf{max}}\langle rd\rangle(vs, \delta) \leq rd(\delta)$, and

(ii) Let $\Sigma : \mathbb{U}(\delta\restriction_{vs}) \Rightarrow \Delta$ be a function where, for any $vs \subseteq \mathcal{D}(\delta)$ and any state $x \in \mathbb{U}(\delta\restriction_{vs})$, $\Sigma(x) = \delta\downarrow_x$. We have $A_\Delta = \Sigma(\!|G(\delta\restriction_{vs})|\!)$. I.e. A_Δ is a is a relabelling of the state space of $\delta\restriction_{vs}$: a state x in the state space of $\delta\restriction_{vs}$ is relabelled by the snapshot $\delta\downarrow_x$ of the constructed system δ on x.

Proof. Our proof is constructive like our previous tightness proofs. Here, we construct a witness system δ and a set of variables vs. First, let the set of variables $vs = \{v_1, v_2, .., v_{\lceil \log_2 |V(A_\Delta)| \rceil}\}$. The reason that we have that many variables in vs is that we want to have an injection V_{vs} from $V(A_\Delta)$ to a set of states defined on vs, i.e. we want every vertex of A_Δ to be associated with a unique assignment of vs. Also vs is chosen to be disjoint from the domains of any of the systems labelling the vertices of A_Δ, i.e. for any $v \in vs$ and $u \in V(A_\Delta)$, we have $v \notin \mathcal{D}(\delta_u)$.

Intuitively, the system δ that we construct is comprised of the systems labelling the vertices in A_Δ and another system, δ_{vs}, which we construct. We construct δ_{vs} such that the largest connected component of its state space is isomorphic with A_Δ, i.e. there is a bijection E_{vs} from the edges $E(A_\Delta)$ to the actions δ_{vs}.

Next, for every vertex $u \in V(A_\Delta)$, let δ_u denote the system $\{(p \uplus V_{vs}(u), e) \mid (p, e) \in A_\Delta(u)\}$, i.e. we add to the precondition of every action in every system $A_\Delta(u)$ labelling a vertex $u \in V(A_\Delta)$ the assignments of vs corresponding to u. Now, for every vertex $u \in V(A_\Delta)$, let $\overrightarrow{\pi}^{rd}_u$ denote an action sequence from ${\delta_u}^*$ and I^{rd}_u denote a state from $\mathbb{U}(\delta_u)$ such that $\overrightarrow{\pi}^{rd}_u(I^{rd}_u)$ is a simple path (i.e. traverses only different states) whose length is equal to $rd(\delta_u)$. Note that from Definition 6 such a path and a exist for any system.

For an edge $(u_1, u_2) \in E(A_\Delta)$, the action $E_{vs}(u_1, u_2)$ is defined as $(V_{vs}(u_1), I^{rd}_{u_2} \uplus V_{vs}(u_2))$. The system δ_{vs} is thus defined as the set of actions $\{E_{vs}(u_1, u_2) \mid (u_1, u_2) \in E(A_\Delta)\}$, i.e. one action in δ_{vs} per edge from A_Δ. Lastly, the witness system that we construct δ is $\delta_{vs} \cup \{\delta_u \mid u \in V(A_\Delta)\}$, i.e. the union of the actions that label the given DAG and the actions from δ_{vs}.

To show that δ satisfies condition (i), we construct an action sequence $\overrightarrow{\pi} \in \delta^*$ and a state $x \in \mathbb{U}(\delta)$, such that executing $\overrightarrow{\pi}$ at x only traverses different states and the length of $\overrightarrow{\pi}$ is greater than $\mathsf{S}_{\mathsf{max}}\langle rd\rangle(vs, \delta)$. Consider Algorithm 5 that constructs an action sequence $\overrightarrow{\pi}_u$ for every vertex $u \in V(A_\Delta)$.

Algorithm 5:

```
1 π⃗_u := π⃗^rd_u ; children = children_{A_Δ}(u)
2 while children ≠ ∅
3     u_max = argmax_{u_2 ∈ children} S⟨rd ∘ A_Δ⟩(u_2, A_Δ)
4     π⃗_u := π⃗_u ⌢ [E_vs(u, u_max)] ⌢ π⃗^rd_{u_max}
5     children = children_{A_Δ}(u_max); u = u_max
```

Algorithm 5 constructs the path $\overrightarrow{\pi}_u$ with two properties. Firstly, $\overrightarrow{\pi}_u$ is the weightiest path in the graph A_Δ starting from u, with the vertices of A_Δ given as weights the recurrence diameters of the systems labelling them, and the edges given unit weights. In other words, it returns a path whose weight is $\mathsf{S}\langle rd\rangle(V_{vs}(u), vs, \delta)$.†

Secondly, $\overrightarrow{\pi}_u$ only traverses different states, if executed at the state $I^{rd}_u \uplus V_{vs}(u)$, i.e. $\overrightarrow{\pi}_u(I^{rd}_u \uplus V_{vs}(u))$ induces a simple path.‡ This is because, by construction, for any vertex $u \in V(A_\Delta)$, $\overrightarrow{\pi}^{rd}_u(I^{rd}_u)$ induces a simple path. Also, for any two different vertices $u_1, u_2 \in V(A_\Delta)$, if the action sequences $\overrightarrow{\pi}^{rd}_{u_1}$ and $\overrightarrow{\pi}^{rd}_{u_2}$ occur as sublists of $\overrightarrow{\pi}_u$, they will only traverse different states. This is because the action $E_{vs}(u_1, u_2)$ will which change the assignments of vs between $\overrightarrow{\pi}^{rd}_{u_1}$ and $\overrightarrow{\pi}^{rd}_{u_2}$, and because neither $\overrightarrow{\pi}^{rd}_{u_1}$ nor $\overrightarrow{\pi}^{rd}_{u_2}$ can change the assignments of vs, as vs is disjoint with the domains of systems labelling A_Δ.

Now let the required $\overrightarrow{\pi}$ be $\overrightarrow{\pi}_{u_{\mathsf{max}}}$, and the required x be $I^{rd}_{u_{\mathsf{max}}} \uplus V_{vs}(u_{\mathsf{max}}) \uplus \{v \mid v \in \mathcal{D}(\delta) \setminus (\mathcal{D}(\delta_{u_{\mathsf{max}}}) \cup \mathcal{D}(\delta_{vs}))\}$, where u_{max} denotes the vertex $u \in V(A_\Delta)$ with the largest $|\overrightarrow{\pi}_u|$. Condition (i) holds for δ because of three reasons. Firstly, executing the path $\overrightarrow{\pi}$ at the state x only traverses different states, i.e. $\overrightarrow{\pi}(x)$ induces a simple path. This holds as for any $u \in V(A_\Delta)$, from ‡, we know that $\overrightarrow{\pi}_u(I^{rd}_u \uplus V_{vs}(u))$ induces a simple path, and since by construction we have $I^{rd}_{u_{\mathsf{max}}} \uplus V_{vs}(u_{\mathsf{max}}) \subseteq x$.

Secondly, we have that $|\overrightarrow{\pi}|$ is at least $\mathsf{S}_{\mathsf{max}}\langle rd\rangle(vs, \delta)$ from †.

Thirdly, since for any $u \in V(A_\Delta)$ the action sequence $\overrightarrow{\pi}_u$ belongs to δ^*, then $\overrightarrow{\pi} \in \delta^*$ holds. Also, $x \in \mathbb{U}(\delta)$ because the by construction its domain x is equal to the $\mathcal{D}(\delta)$, and thus condition (i) holds for δ.

Condition (ii) holds for δ and vs because, firstly, the largest connected component of the state space of $\delta\restriction_{vs}$ is isomorphic to A_Δ since vs is disjoint from the domains of the effects of any of the actions in δ_u, for any $u \in V(A_\Delta)$. Note: that every vertex in $\delta\restriction_{vs}$ will be labelled by a state from the range of V_{vs} instead of a system as is the case with A_Δ.

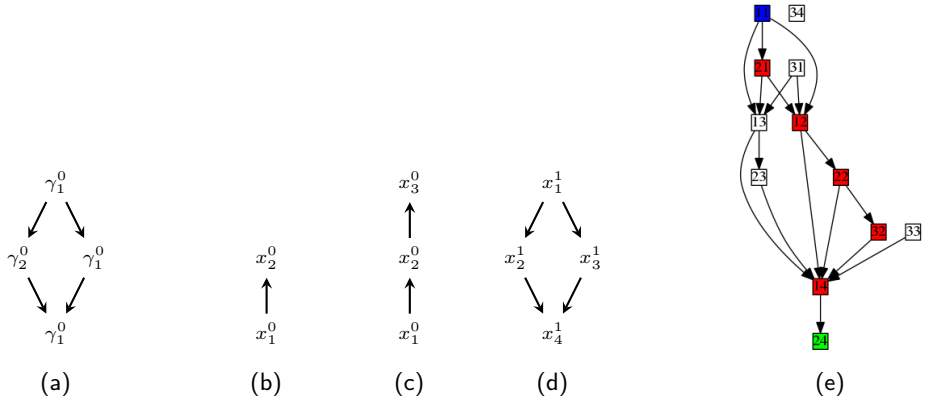


Fig. 16. Referring to Example 17: (a) is the given DAG A_Δ which is labelled by factored paths, (b) and (c) are largest connected components in the paths γ_1^0 and γ_2^0 which label A_Δ, (d) is the state space of the constructed system δ_{vs}, and (e) is the state space of the constructed system δ in which a state $x_i^0 \uplus x_j^1$ is denoted by ij. It also shows the path induced by executing the constructed action sequence $\overrightarrow{\pi}$, which starts at the constructed state $x = x_1^0 \uplus x_1^1$ and reaches $x_2^0 \uplus x_4^1$.

Secondly, for any vertex $u_1 \in V(A_\Delta)$, the snapshot $\delta{\restriction}_{V_{vs}(u_1)}$ is the system $A_\Delta(u_1)$ that labels u_1. This holds because (i) Actions from δ_{vs} will not be included in the snapshot, since every action from δ_{vs} is of the form $(V_{vs}(u_3), I_{u_4}^{rd} \uplus V_{vs}(u_4))$, where $u_3 \neq u_4$. Since V_{vs} is injective, either $V_{vs}(u_1) \neq V_{vs}(u_3)$ or $V_{vs}(u_1) \neq V_{vs}(u_4)$ holds, and thus the action $(V_{vs}(u_3), I_{u_4}^{rd} \uplus V_{vs}(u_4))$ will not be included in the snapshot $\delta{\restriction}_{V_{vs}(u_1)}$. (ii) For any $u_2 \neq u_1$, actions from δ_{u_2} will not be included in the snapshot $\delta{\restriction}_{V_{vs}(u_1)}$ because the system δ only includes modified versions of the actions in δ_{u_2}, with $V_{vs}(u_2)$ added to the precondition. □

The following example demonstrates the construction that we use in our proof.

Example 17. Consider two factored simple paths γ_1^0 and γ_2^0 (following the notation from the proof of Theorem 5) whose state spaces are shown in Figures 16b and 16c. Explicitly, those paths are $\gamma_1^0 = \{(x_1^0, x_2^0)\}$ and $\gamma_2^0 = \{(x_1^0, x_2^0), (x_2^0, x_3^0)\}$. The states on which the paths are defined are $x_1^0 = \{\overline{v_1}, \overline{v_2}\}$, $x_2^0 = \{\overline{v_1}, v_2\}$, and $x_3^0 = \{v_1, \overline{v_2}\}$. Also, consider the DAG A_Δ shown in Figure 16a. It has four vertices u_1 (top), u_2 (left), u_3 (right), and u_4 (bottom) whose labels are $A_\Delta(u_1) = \gamma_1^0$, $A_\Delta(u_2) = \gamma_2^0$, $A_\Delta(u_3) = \gamma_1^0$, and $A_\Delta(u_4) = \gamma_1^0$.

The first step in the construction from the proof of Theorem 12 is that of building the system δ and the set of variables vs. We start by partially specifying the system δ_{vs} which is constructed to have a state space isomorphic to the given DAG A_Δ. We define the mapping V_{vs} to be $\{(u_i \mapsto x_i^1) \mid 1 \leq i \leq 4\}$, where the states in its range are $x_1^1 = \{\overline{v_3}, \overline{v_4}\}$, $x_2^1 = \{\overline{v_3}, v_4\}$, $x_3^1 = \{v_3, \overline{v_4}\}$, and $x_4^1 = \{v_3, v_4\}$. Note that the variables chosen for those states x_i^1 are disjoint from the ones on which the paths γ_1^0 and γ_2^0 are defined.

The second step in the construction is building the systems $\delta_{u_i} = \{(p \uplus V_{vs}(u_i), e) \mid (p, e) \in A_\Delta(u_i)\}$, for $1 \leq i \leq 4$. Explicitly, $\delta_{u_1} = \{(x_1^0 \uplus x_1^1, x_2^0)\}$, $\delta_{u_2} = \{(x_1^0 \uplus x_2^1, x_2^0), (x_2^0 \uplus x_2^1, x_3^0)\}$, $\delta_{u_3} = \{(x_1^0 \uplus x_3^1, x_2^0)\}$, and $\delta_{u_4} = \{(x_1^0 \uplus x_4^1, x_2^0)\}$. For any δ_{u_i}, there is a state $I_{u_i}^{rd} \in \mathbb{U}(\delta_{u_i})$ and an action sequence $\overrightarrow{\pi}_{u_i}^{rd} \in {\delta_{u_i}}^*$ such that executing $\overrightarrow{\pi}_{u_i}^{rd}$ at $I_{u_i}^{rd}$ only traverses different states. Explicitly, $I_{u_1}^{rd} = x_1^0 \uplus x_1^1$, $I_{u_2}^{rd} = x_1^0 \uplus x_2^1$, $I_{u_3}^{rd} = x_1^0 \uplus x_3^1$, $I_{u_4}^{rd} = x_1^0 \uplus x_4^1$, $\overrightarrow{\pi}_{u_1}^{rd} = [(x_1^0 \uplus x_1^1, x_2^0)]$, $\overrightarrow{\pi}_{u_2}^{rd} = [(x_1^0 \uplus x_2^1, x_2^0); (x_2^0 \uplus x_2^1, x_3^0)]$, $\overrightarrow{\pi}_{u_3}^{rd} = [(x_1^0 \uplus x_3^1, x_2^0)]$, and $\overrightarrow{\pi}_{u_4}^{rd} = [(x_1^0 \uplus x_4^1, x_2^0)]$.

Thirdly we fully specify the system δ_{vs} by specifying the mapping E_{vs}, which is $\{(u_i, u_j) \mapsto (x_i^1, I_{u_j}^{rd} \uplus x_j^1) \mid 1 \leq i, j \leq 4\}$. This thus completely specifies δ, which is $\delta_{vs} \cup \{\delta_{u_i} \mid 1 \leq i \leq 4\}$, and vs which is $\{v_3, v_4\}$. The state space of δ is shown in Figure 16e.

After constructing δ we construct the required action sequence $\overrightarrow{\pi}$ and the required state x. If we literally follow our proof, we would construct $\overrightarrow{\pi}_{u_i}$ for each $1 \leq i \leq 4$ using Algorithm 5, and then set $\overrightarrow{\pi}$ to be the $\overrightarrow{\pi}_{u_i}$ with the maximum length. However, we only construct $\overrightarrow{\pi}_{u_1}$ here, since it should be clear that it is the longest one. Algorithm 5 will first set $\overrightarrow{\pi}_{u_1}$ to $\overrightarrow{\pi}_{u_1}^{rd}$. Then it will set u to the child of u_1 which is labelled with the system with the largest recurrence diameter, i.e. u_2. Next, it will set $\overrightarrow{\pi}_{u_1}$ to be $\overrightarrow{\pi}_{u_1}^{rd} \frown [E_{vs}(u_1, u_2)] \frown \overrightarrow{\pi}_{u_2}^{rd}$. Then the same steps will be repeated, but with u set to u_4, which is the only child of u_2, to finally have $\overrightarrow{\pi}_{u_1}$ set to $\overrightarrow{\pi}_{u_1}^{rd} \frown [E_{vs}(u_1, u_2)] \frown \overrightarrow{\pi}_{u_2}^{rd} \frown [E_{vs}(u_2, u_4)] \frown \overrightarrow{\pi}_{u_4}^{rd}$. Explicitly, $\overrightarrow{\pi}_{u_1} = [(x_1^0 \uplus x_1^1, x_2^0); (x_1^1, x_1^0 \uplus x_2^1); (x_1^0 \uplus x_2^1, x_2^0); (x_2^0 \uplus x_2^1, x_3^0); (x_2^1, x_1^0 \uplus x_4^1); (x_1^0 \uplus x_4^1, x_2^0)]$. Since $\mathcal{D}(\delta) \setminus (\mathcal{D}(\delta_{u_1}) \cup \mathcal{D}(\delta_{vs})) = \emptyset$ the required state x will be $I_{u_1}^{rd} \uplus V_{vs}(u_1) = x_1^0 \uplus x_1^1$. The path traversed by $\overrightarrow{\pi}_{u_1}$ starting at x in the state space of δ is shown in Figure 16e. Since it clearly does not traverse the same state twice, then the recurrence diameter of δ is at

least $|\vec{\pi}|$, which is 6. This together with the fact that evaluating $\mathsf{S}_{\mathsf{max}}\langle rd\rangle(vs,\delta)$ results in 6, shows that δ satisfies condition (i) on the required witness.

To see that the constructed system δ satisfies condition (ii) on the constructed witness, note that its snapshot on any of the states in the state space of the projection $\delta\lfloor_{vs}$ is the system that labels the corresponding vertex in the given DAG A_Δ, i.e. $A_\Delta = \Sigma(\!|G(\delta\lfloor_{vs})|\!)$. For example, the snapshot $\delta\restriction_{x_1^1}$ of δ on the state x_1^1 is $\{(x_1^0, x_2^0)\}$ which is the system δ_{u_1}. Other actions in δ will not be included in $\delta\restriction_{x_1^1}$ as they have preconditions and/or effects that contradict x_1^1. For instance, the action $(x_1^1, x_1^0 \uplus x_2^1)$ from δ_{vs} has x_2^1 in its effect which contradicts x_1^1.

Theorem 13. *For any DAG A_Δ labelled with factored systems, there is a factored system δ and a set of variables vs where:*

(i) $\mathsf{S}_{\mathsf{max}}\langle\ell\rangle(vs,\delta) \leq \ell(\delta)$, and
(ii) $A_\Delta = \Sigma(\!|G(\delta\lfloor_{vs})|\!)$, where Σ and $G(\delta\lfloor_{vs})$ are as above.

Proof. The proof for this theorem is almost exactly the same as that of Theorem 12, except that instead of showing that executing the path $\vec{\pi}$ at the state x traverses different states, we need to show that executing any proper sublist of $\vec{\pi}$ (i.e. $\vec{\pi}$ but with action(s) dropped from it) at x will not result in the same state as executing $\vec{\pi}$ from x.

To accommodate for that, all constructions should be prameterised on the sublist diameter instead of the recurrence diameter. For instance, for a vertex $u \in A_\Delta$ we use the action sequence $\vec{\pi}^{\ell}_u$ instead of $\vec{\pi}^{rd}_u$, where $\vec{\pi}^{\ell}_u$ is an action sequence whose length is equal to the sublist diameter $\ell(\delta_u)$ and from which there is not a proper sublist that achieves the same execution outcome from that state.

Now, for any vertex $u_1 \in V(A_\Delta)$, Algorithm 5 constructs an action sequence $\vec{\pi}_{u_1}$ of the form $\vec{\pi}^{\ell}_{u_1} \frown [E_{vs}(u_1,u_2)] \frown \vec{\pi}^{\ell}_{u_2} \frown [E_{vs}(u_2,$ If we drop some action(s) from any $\vec{\pi}^{\ell}_{u_i}$, executing the rest of $\vec{\pi}_{u_1}$ on the state x *will not* be the same as executing $\vec{\pi}$, which follows from the definition of $\vec{\pi}^{\ell}_{u_i}$. However, if we drop $\vec{\pi}^{\ell}_{u_i}$ entirely from $\vec{\pi}_{u_1}$, executing the rest of $\vec{\pi}_{u_1}$ on the state x *will* be the same as executing the entirety $\vec{\pi}_{u_1}$, because the actions coming after $\vec{\pi}_{u_i}$ do not have preconditions satisfied by the execution of $\vec{\pi}^{\ell}_{u_i}$. This violates the main property that we need $\vec{\pi}$ to conform to, which is that executing any proper sublist of $\vec{\pi}$ at x does not result in the same outcome as executing $\vec{\pi}$.

To avoid that, we slightly change the constructed $\vec{\pi}_{u_1}$ for every vertex u_1, by changing δ_{vs}, which will also change δ. In particular, we change the mapping of edges from A_Δ to actions from δ_{vs} to be $E_{vs}(u_1,u_2) = (\vec{\pi}^{\ell}_{u_1}(I^{\ell}_{u_1}) \uplus V_{vs}(u_1), I^{\ell}_{u_2} \uplus V_{vs}(u_2))$, for every pair of vertices $u_1, u_2 \in V(A_\Delta)$, where we add the result of executing $\vec{\pi}^{\ell}_{u_1}$ to the precondition of the action $E_{vs}(u_1,u_2)$. This way we guarantee that if any $\vec{\pi}^{\ell}_{u_i}$ is completely stripped out of $\vec{\pi}_{u_1}$, the remaining actions from $\vec{\pi}_{u_1}$ will not be able to execute, since the action $E_{vs}(u_i,u_{i+1})$, which comes just after $\vec{\pi}_{u_i}$ has the precondition $\vec{\pi}^{\ell}_{u_i}(I^{\ell}_{u_i})$ which will not be satisfied if $\vec{\pi}^{\ell}_{u_i}$ is removed.

Other than this slight modification, the construction from the proof of Theorem 12 should be exactly the same. □

The following example demonstrates how to modify the construction from Theorem 12 to make it work for Theorem 13.

Example 18. Consider the given DAG from Example 17. If we exactly follow the construction from the proof of Theorem 12 to prove Theorem 13 we would end up with the action sequence $\vec{\pi} = \vec{\pi}^{\ell}_{u_1} \frown [E_{vs}(u_1,u_2)] \frown \vec{\pi}^{\ell}_{u_2} \frown [E_{vs}(u_2,u_4)] \frown \vec{\pi}^{\ell}_{u_4} = [(x_1^0 \uplus x_1^1, x_2^0); (x_1^1, x_1^0 \uplus x_2^1); (x_1^0 \uplus x_2^1, x_2^0); (x_2^0 \uplus x_2^1, x_3^0); (x_2^1, x_1^0 \uplus x_4^1); (x_1^0 \uplus x_4^1, x_2^0)]$, which is the same action sequence constructed in Example 17. If we remove $\vec{\pi}^{\ell}_{u_2}$ from $\vec{\pi}$, the remaining action sequence $[(x_1^0 \uplus x_1^1, x_2^0); (x_1^1, x_1^0 \uplus x_2^1); (x_2^1, x_1^0 \uplus x_4^1); (x_1^0 \uplus x_4^1, x_2^0)]$ is still executable, and will lead to the same outcome as $\vec{\pi}$ if executed at the state x constructed in Example 17 (see the path in the state space of δ that it induces in Figure 17a).

If we, however, follow the new mapping E_{vs} that was introduced in the proof of Theorem 13, the constructed action sequence will be $[(x_1^0 \uplus x_1^1, x_2^0); (x_2^0 \uplus x_1^1, x_1^0 \uplus x_2^1); (x_1^0 \uplus x_2^1, x_2^0); (x_2^0 \uplus x_2^1, x_3^0); (x_3^0 \uplus x_2^1, x_1^0 \uplus x_4^1); (x_1^0 \uplus x_4^1, x_2^0)]$. This action sequence would induce the path in Figure 17b if it were executed at the constructed state x. One can check that removing any action from that new action sequence will cause its execution at x to produce a different result because some actions' preconditions will not be executable.

5.3.3 Discussion. There are two points on which we would like to comment before closing this section. Firstly, note that, in some sense, Theorems 12 and 13 are stronger statements regarding the tightness of $\mathsf{S}_{\mathsf{max}}$ compared to the statements by Theorems 3 and 8 regarding the tightness of $\mathsf{N}_{\mathsf{sum}}$ and $\widehat{\mathsf{N}}_{\mathsf{sum}}$, respectively. In particular Theorems 12 and 13 assert that a compositional algorithm that analyses the entirety of the abstractions (i.e. the snapshots) and the state space of the acyclic projection, cannot compute bounds on the recurrence diameter or the sublist diameter tighter than those computed by $\mathsf{S}_{\mathsf{max}}$. In contrast, Theorem 3 only states that no compositional algorithm can improve over $\mathsf{N}_{\mathsf{sum}}$

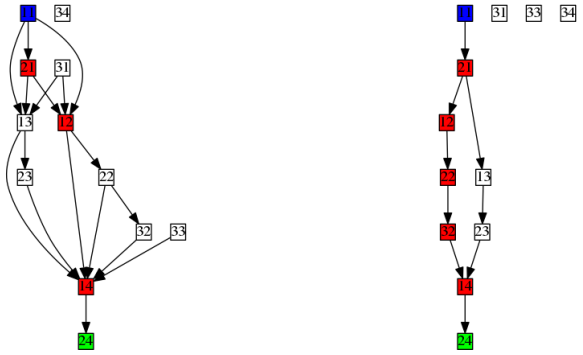


Fig. 17. Referring to Example 18: (a) is the state space of the system δ constructed in Example 17 following the proof of Theorem 12, and the path induced by executing a strict sublist of the constructed action sequence $\vec{\pi}$ at the state x. The strict sublist of $\vec{\pi}$ reaches the same state reached by $\vec{\pi}$ which is $x_2^0 \uplus x_4^1$. (b) is the state space of the system constructed in Example 18 using the new edge mapping E_{vs} from the Proof of Theorem 13 and the path induced by the action sequence constructed using that new edge mapping. It should be clear that there is not a proper sublist of the constructed actions sequence that can reach $x_2^0 \uplus x_4^1$.

if the algorithm only analyses the recurrence diameters of the abstractions (i.e. the projections), and not the entire abstraction. The tightness statement is slightly stronger for $\widehat{\mathsf{N}}_{\mathsf{sum}}$, where Theorem 8 states it is at least as tight as algorithms that analyse the abstractions' recurrence diameters, as well as abstractions' state space acyclicity. Thus, it could be the case that there are compositional algorithms that produce better bounds than $\widehat{\mathsf{N}}_{\mathsf{sum}}$, if those algorithms have access to more of the structure of the abstractions, instead of merely their sublist diameters and state space acyclicity. In essence, this suggests that there is more room for improvement in compositional bounding based on projections and dependency graphs than bounding based on snapshots and state space structure.

Secondly, $\mathsf{S}_{\mathsf{max}}$ does not produce tight bounds for the diameter because the paths traversed by $\mathsf{S}_{\mathsf{max}}$ are not necessarily the shortest paths to the furthest vertex in the given DAG. Instead $\mathsf{S}_{\mathsf{max}}$ computes the weightiest path in the DAG starting at the given vertex, which is a variation of the longest path that includes vertex weights. For instance, in Example 17 above, executing the action sequence $[(x_1^0 \uplus x_1^1, x_2^0); (x_1^1, x_1^0 \uplus x_3^1); (x_1^0 \uplus x_3^1, x_2^0); (x_3^1, x_1^0 \uplus x_4^1); (x_1^0 \uplus x_4^1, x_2^0)]$ starting at x will reach the same state as that reached by executing the constructed action sequence $\vec{\pi}$ on x. On the other hand, $\mathsf{S}_{\mathsf{max}}$ will still return 6, which is the weight of the weightiest path in δ, which is $\vec{\pi}$. One way to circumvent this is to replace $\mathsf{S}_{\mathsf{max}}$ by solving a variant of All Pairs Shortest Paths that computes the "lightest" paths after giving weights to vertices instead just shortest path. However, we do not pursue that here since, as we describe in the next section, we seek to merge $\mathsf{S}_{\mathsf{max}}$ with $\widehat{\mathsf{N}}_{\mathsf{sum}}$ to get practically useful bounds. Those practical bounds will not benefit from having a new version of $\mathsf{S}_{\mathsf{max}}$ that tightly bounds the diameter, since we showed above that $\mathsf{N}_{\mathsf{sum}}$ is only a sound bound on the diameter if the base-case function is at least the sublist diameter. Thus, we are primarily concerned about the case when the base-case function given to $\mathsf{S}_{\mathsf{max}}$ is an upper bound on ℓ.

5.4 A Practical Algorithm for Upper-Bounding: A Hybrid Algorithm

We now give a hybrid algorithm that combines the dependency based decomposition and state space based decomposition. Our main motivation for this hybrid algorithm is better run-times. We have just observed a situation where PUR can exhibit a run-time that is linear in the size of the state space. That is favourable compared to exact calculations of diameter, which in our opening remarks we noted to have worse-than-quadratic run-time. Nevertheless this is unacceptable in our factored setting, and we now seek to alleviate this computational burden by applying $\mathsf{S}_{\mathsf{max}}$ to projections computed by the top-down algorithm. Such projections can be significantly smaller than the concrete systems, thus motivating a hybrid approach that can exponentially reduce bound computation times, as shown in this example.

Example 19. Consider applying the approach outlined in Example 11 to compute PUR only on the abstractions $\delta\downarrow_{\text{ROOM1}}$ and $\delta\downarrow_{\text{ROOM2}}$. $\text{PUR}(\delta\downarrow_{\text{ROOM1}})$ can be evaluated in $\prod_{vs \in \Omega(\delta\downarrow_{\text{ROOM1}})} |vs|$ calls to $\mathsf{S}_{\mathsf{max}}$, where $\Omega(\delta\downarrow_{\text{ROOM1}}) = \{\mathtt{ck}_1, \mathtt{lk}_1, \{\mathtt{gk}_{1,2}\}, \{\mathtt{gk}_{1,3}\}, \{\mathtt{gk}_{2,2}\}, \{\mathtt{gk}_{2,3}\}\}$. The same observation can be made for the evaluation time of $\text{PUR}(\delta\downarrow_{\text{ROOM2}})$. Thus the product expression in Example 16 is split into a sum if PUR is called on projections.

Another obvious motivation of a hybrid approach is that, compared to its constituents, in some cases the hybrid approach would successfully produce abstractions when only one of its constituents is able to produce abstractions. In addition to this obvious gain, interleaving dependency based and state space based decompositions can indeed further the applicability of both types of decomposition. Decomposing a system using state space acyclicity analysis can

remove variable dependencies, thus allowing $\mathsf{N}_{\mathsf{sum}}$ to be applied to systems where it otherwise cannot. This is shown in the next example.

Example 20. As shown in Figure 12b, the dependency graph of $\delta\downarrow_{\text{ROOM1}}$ has a single SCC, and thus not susceptible to dependency analysis. Taking a snapshot of $\delta\downarrow_{\text{ROOM1}}$ at the assignment $\mathtt{CK}_{1,2}$ yields a system with one SCC in its dependency graph as well, as shown in Figure 14a. However, taking the snapshot of $\delta\downarrow_{\text{ROOM1}}\restriction_{\mathtt{CK}_{1,2}}$ at the assignment $\{\overline{\mathtt{lk}_{1,1}}, \mathtt{lk}_{1,2}, \overline{\mathtt{lk}_{1,3}}\}$, denoted by $\mathtt{LK}_{1,2}$, yields a system with an acyclic dependency graph as shown in Figure 14b.

HYB is a recursive upper-bounding algorithm, which combines exploitation of acyclic variable dependency with exploitation of acyclicity in state spaces. In HYB, $vs_{1..n}$ is a partition of $\mathcal{D}(\delta)$, and $\mathsf{ac}(vs_{1..n})$ is a member of $vs_{1..n}$ s.t. the projection $\delta\downarrow_{\mathsf{ac}(vs_{1..n})}$ has a non-trivial acyclic state space. HYB interleaves the functions $\mathsf{N}_{\mathsf{sum}}$ and $\mathsf{S}_{\mathsf{max}}$. It only calls $\mathsf{S}_{\mathsf{max}}$ if the given system's dependencies are strongly connected and δ has acyclic projections on members of $vs_{1..n}$. Note that in HYB, $\mathsf{S}_{\mathsf{max}}$ is only applied to the given transition system δ if there is no non-trivial projection (i.e.if the dependency graph, $G_{\mathcal{D}(\delta)}$, has one SCC). Also note that the dependency graph, $G_{\mathcal{D}(\delta)}$, is constructed and analysed with every recursive call to HYB since snapshotting in earlier calls can remove variable dependencies as a result of removing actions, thus breaking strongly connected dependencies. If both $\mathsf{N}_{\mathsf{sum}}$ and $\mathsf{S}_{\mathsf{max}}$ cannot be called, HYB uses EXP as a base-case function, which is to be specified later. Another point is a notational one: the hybrid algorithm is passed as the base-case function to both $\mathsf{N}_{\mathsf{sum}}$ and $\mathsf{S}_{\mathsf{max}}$, which is how recursion is achieved. It is passed with an empty first argument, denoted by $\bullet$. That argument is to be populated by projections or snapshots inside $\mathsf{N}_{\mathsf{sum}}$ or $\mathsf{S}_{\mathsf{max}}$, respectively.

We prove HYB is sound by proving it is sound for as tight a base function as possible. Then soundness for using EXP as a base function follows. As discussed above, d cannot be used, because $\mathsf{N}_{\mathsf{sum}}\langle d\rangle$ is not a valid upper bound on the diameter. However, using ℓ as a base-case function for $\mathsf{N}_{\mathsf{sum}}$ is sound. Thus, using Theorem 10, and Lemma 1, the validity of HYB as an upper bound on ℓ (and accordingly, the diameter) follows.

PROPOSITION 6. *If* EXP *bounds* ℓ*, then* $\ell(\delta) \leq \text{HYB}(\delta)$.

5.4.1 Empirical Evaluations. We first discuss the practicalities of implementing HYB. The base-case function, EXP, is taken to be the number of possible assignments of the SAS+ variables. This choice is pragmatic, taken in light of the fact that computation of alternatives, such as the recurrence diameter, is computationally hard. To optimise computing $\mathsf{N}_{\mathsf{sum}}$ and $\mathsf{S}_{\mathsf{max}}$, we use memoisation, where we compute N or S once for every projection or snapshot, respectively, and store it in a look-up table. This reduced the bound computation time by 70% on average. Our evaluation considers problems from previous International Planning Competitions (IPC), and the unsolvablity IPC, and open Qualitative Preference Rovers benchmarks from IPC2006. Below, the latter are referred to as NEWOPEN.

Quality of HYB *Bounds.* Two measurements related to a compositional upper-bounding algorithm are indicative of its quality. First, we seek an indication of the degree of decomposition provided by the algorithm. An indication is provided by comparing the size of the domain of the concrete problem—i.e. $|\mathcal{D}(\delta)|$—with that of the largest base-case or the largest abstraction computed for that concrete problem. A strong decomposition is indicated when the domain of the biggest base-case is small relative to the concrete problem. Second, we seek an approach that is able to produce bounds that grow sub-exponentially with the size of the problem, when they exist. Thus, we measure how the upper bounds scale as the size of the problem instances grow. If the bounds scale gracefully, this indicates an effective compositional approach.

We report our measurements of the performance of HYB in these terms. Our experiments were conducted on a uniform cluster with time and memory limits of 30minutes and 8GB, respectively. Figure 18 shows the domain size of the largest base-case compared to the size of the concrete problem. IPC domains with instances remarkably susceptible to decomposition by HYB are: ROVERS (both solvable and unsolvable), STORAGE, TPP (both solvable and unsolvable), LOGISTICS, NEWOPEN, NOMYSTERY (both solvable and over-subscribed), UNSOLVABLE MYSTERY, VISITALL, SATELLITES, ZENO TRAVEL, and ELEVATORS. For those problems, the size of the largest base-case is significantly smaller than the size of the concrete problem, as shown in Figure 18. One IPC domain that is particularly amenable to decomposition is the ROVERS domain, where many of its instances are decomposed to have largest base-cases modelling a single Boolean state-variable. Also, for domains susceptible to decomposition, the bounds computed by HYB grow sub-exponentially with the number of state variables, as shown in Figure 19. We also note that out of those domains, LOGISTICS, NOMYSTERY, SATELLITES, ZENO TRAVEL and ELEVATORS, have linear (or almost linear) growth of the bounds with the size of the problem.

We also ran HYB on a PDDL [50] encoding of the hotel key protocol, with the parameters G, k, and R ranging between 1 and 10 (i.e. 1000 instances of the protocol). As shown in Figure 18 (and in the examples earlier), this protocol is particularly amenable to decomposition by HYB. All instances had a largest base-case modelling a single Boolean state-variable. Additionally, as shown in Figure 19 the bounds computed by HYB for this set of benchmarks are constant in the number of guests G, grow linearly in the number of rooms R, and quadratically in the number of keys per room K.

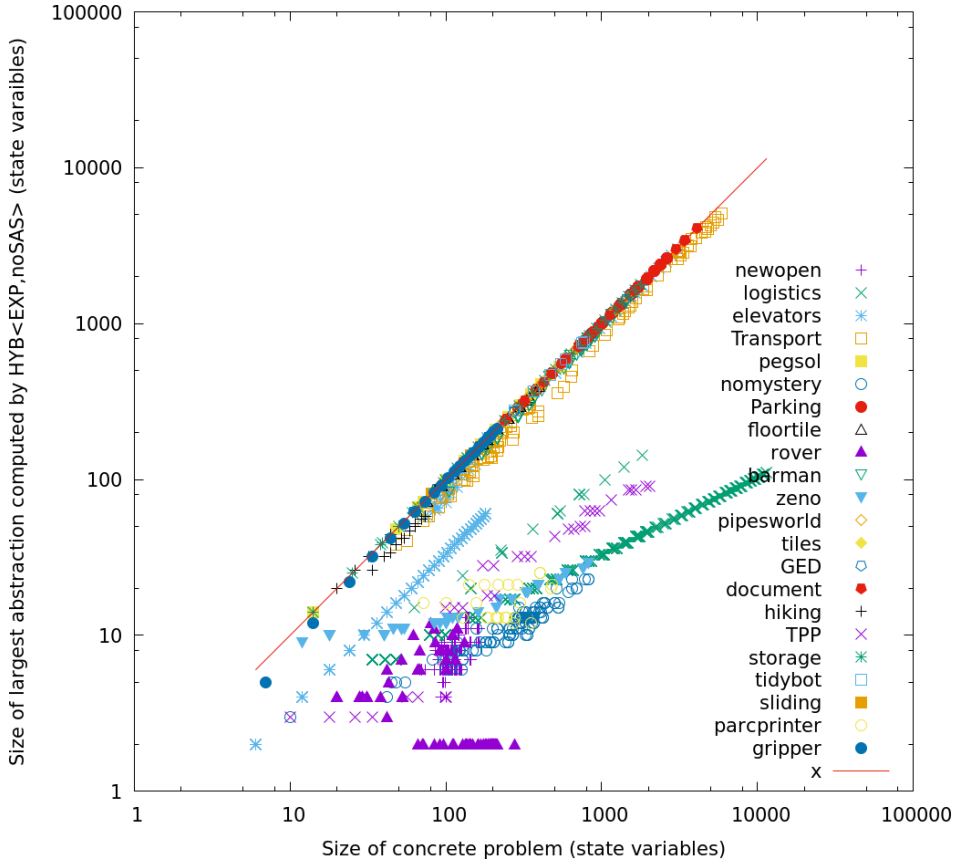


Fig. 18. Measurements related to HYB on benchmarks: scatter plot of the size of the largest base-case (horizontal), and the size of the concrete problem (vertical).

Comparison of HYB *with Other Algorithms.* We compare the performance of the hybrid compositional bounding algorithm HYB with both of its constituent algorithms, namely, $\mathsf{N}_{\mathsf{sum}}$ which we showed earlier to dominate other state-of-the-art compositional bounding algorithms, and with $\mathsf{S}_{\mathsf{max}}$. Our experimental cluster and settings are as above. Our analysis and experimentation shows that HYB significantly outperforms both of its constituents $\mathsf{N}_{\mathsf{sum}}$ and $\mathsf{S}_{\mathsf{max}}$, both in terms of decomposition quality and the tightness of computed bounds. This is shown in Table 1, which shows that HYB was able to successfully compute bounds for significantly more problems that any other algorithm. Furthermore, it shows that the bounds computed by it are substantially tighter.

The detailed analysis in Figure 20 shows that HYB outperforms $\mathsf{N}_{\mathsf{sum}}$ particularly for the domains: NEWOPEN, NOMYSTERY, ROVERS, HYP, TPP, VISITALL, and BOTTLENECK. The success of HYB in those domains compared to $\mathsf{N}_{\mathsf{sum}}$ reveals something of an abundance of acyclicity in their state space. Figure 21 indicates that HYB is more successful in decomposing problems compared to $\mathsf{N}_{\mathsf{sum}}$, where the largest base-cases for HYB are smaller than those for $\mathsf{N}_{\mathsf{sum}}$ in 71% of the IPC problems. This observation is reinforced, considering that the 1000th largest bound computed by HYB is $50,534$, while the 1000th largest bound computed by $\mathsf{N}_{\mathsf{sum}}$ is more than 10^6. In the HOTELKEY domain, the difference is even more pronounced. The bound computed by HYB is at most 990 for all the 1000 instances, while for $\mathsf{N}_{\mathsf{sum}}$ only 285 instances have bounds less than 10^6. Figure 22 shows the computational cost of this improved bounding performance. HYB typically required more computation time than $\mathsf{N}_{\mathsf{sum}}$. However, HYB terminated in 60 seconds, or less, for 93% of the benchmarks. Thus, we have not observed a significant time penalty.

For $\mathsf{S}_{\mathsf{max}}$ we have similar results. HYB outperforms $\mathsf{S}_{\mathsf{max}}$ in terms of having smaller abstraction sizes (Figure 24) and in the bound computation run-times (Figure 25). Since HYB has the advantage of applying dependency based decomposition before it applies state space based decomposition, it avoids the intractability of state space based decomposition. This most visible in domains where there is substantial acyclicity in variable dependencies like ELEVATORS, HOTEL KEY, 3-UNSAT, STORAGE, ROVERS and SATELLITE. This is further indicated in Figure 23 which shows that the bounds computed by HYB are much tighter than those by $\mathsf{S}_{\mathsf{max}}$. We note that Figures 25 and 23 are sparse because in most problems $\mathsf{S}_{\mathsf{max}}$ failed to compute a bound less than 10^9 within the 30minutes deadline.

We note that the improved decomposition of HYB over $\mathsf{N}_{\mathsf{sum}}$ and $\mathsf{S}_{\mathsf{max}}$ has further application yet to be explored. Should we take EXP to be a more expensive operator, such as the NP-hard recurrence diameter, the stronger decomposition indicates that EXP is invoked for relatively small instances when using HYB compared to $\mathsf{N}_{\mathsf{sum}}$. Thus,

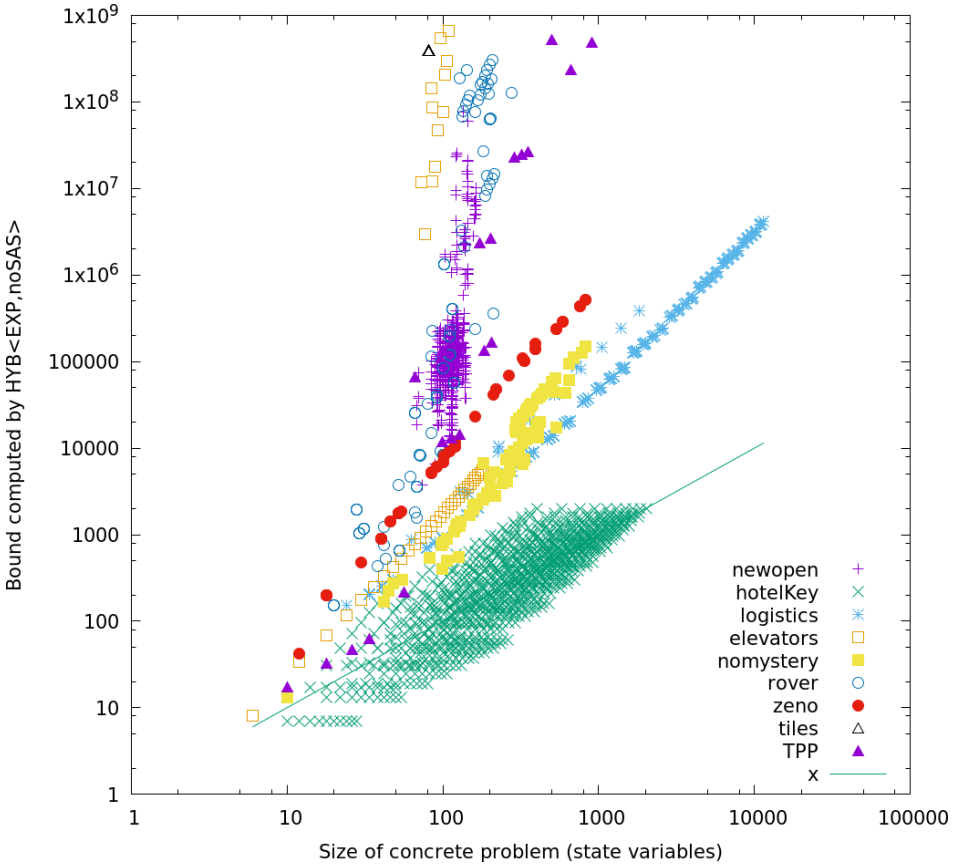


Fig. 19. Measurements related to HYB on benchmarks: scatter plot of the bound (horizontal axis) computed by HYB, and the size (i.e. $|\mathcal{D}(\delta)|$) of the concrete problem (vertical).

computing EXP can be exponentially easier for decompositions computed by HYB compared to decompositions from $\mathsf{N}_{\mathsf{sum}}$.

Planning with HYB. To evaluate the practical utility of the bounds calculated using HYB, we take them as the queried horizon using the MP version of the SAT-based planner Madagascar [56]. In our experiments we limited the time and memory for planners to 1 hour (inclusive of bound computation) and 4GB, respectively. The resulting planner proves the safety of 635 instances of the hotel key protocol, where the instance with 9 rooms, 7 guests, and 45 keys, takes the longest to prove safe—it took just under 30 minutes. This is a substantial improvement over the size of instances automatically proven safe in earlier work. We also ran AIDOS 1 [59] (unsolvability IPC winner) on the hotel key instances and it proved the safety of only 285 of them, where the instance with 2 rooms, 5 guests, and 10 keys, took the longest to prove safe—in 17 minutes. For the IPC benchmarks, our planner proved that 53 instances are unsolvable, 27 of which could not be proven unsolvable by AIDOS 1. The 27 instances are from BOTTLENECK (7 problems), 3UNSAT (4 problems), ELEVATORS (5 problems), and NEWOPEN (11 problems). We also note that compared to the system from [57], we are additionally able to close the heretofore open 7th and 8th Qualitative Preference problems from IPC2006. We also found our bounds useful in solving satisfiable benchmarks. It allowed MP to solve 162 instances that it could not with its default query strategy. Those instances are from ELEVATORS (150 problems), DIAGNOSIS (8 problems), ROVERS (1 problem) and SLIDING-TILES (3 problems).

5.4.2 Discussion. Although both $\mathsf{S}_{\mathsf{max}}$ and $\mathsf{N}_{\mathsf{sum}}$ produce optimal bounds w.r.t. the features they use, we have seen that practically the hybrid approach that combines them outperforms both of them in terms of computed bounds. Example 19 shows one possible explanation underlying improved performance of HYB compared to $\mathsf{S}_{\mathsf{max}}$. This example shows that $\mathsf{S}_{\mathsf{max}}$ can have an intractable run-time, which is avoided by HYB by performing projections before invoking $\mathsf{S}_{\mathsf{max}}$. This allows HYB to compute bounds for many more problems than $\mathsf{S}_{\mathsf{max}}$ in less than the 30 minute deadline.

Another reason behind the computation of better bounds by HYB, compared to either $\mathsf{N}_{\mathsf{sum}}$ or $\mathsf{S}_{\mathsf{max}}$, is that it decomposes concrete problems into base-cases that are much smaller than those computed by either $\mathsf{N}_{\mathsf{sum}}$ or $\mathsf{S}_{\mathsf{max}}$. The hybrid algorithm is able to use smaller bases-cases since it performs both state space based decomposition and dependency graph based decomposition, and interleaves both kinds of analysis. For instance, as shown in Example 20, performing state space based decomposition can create snapshots with acyclicity in their dependency graphs, making

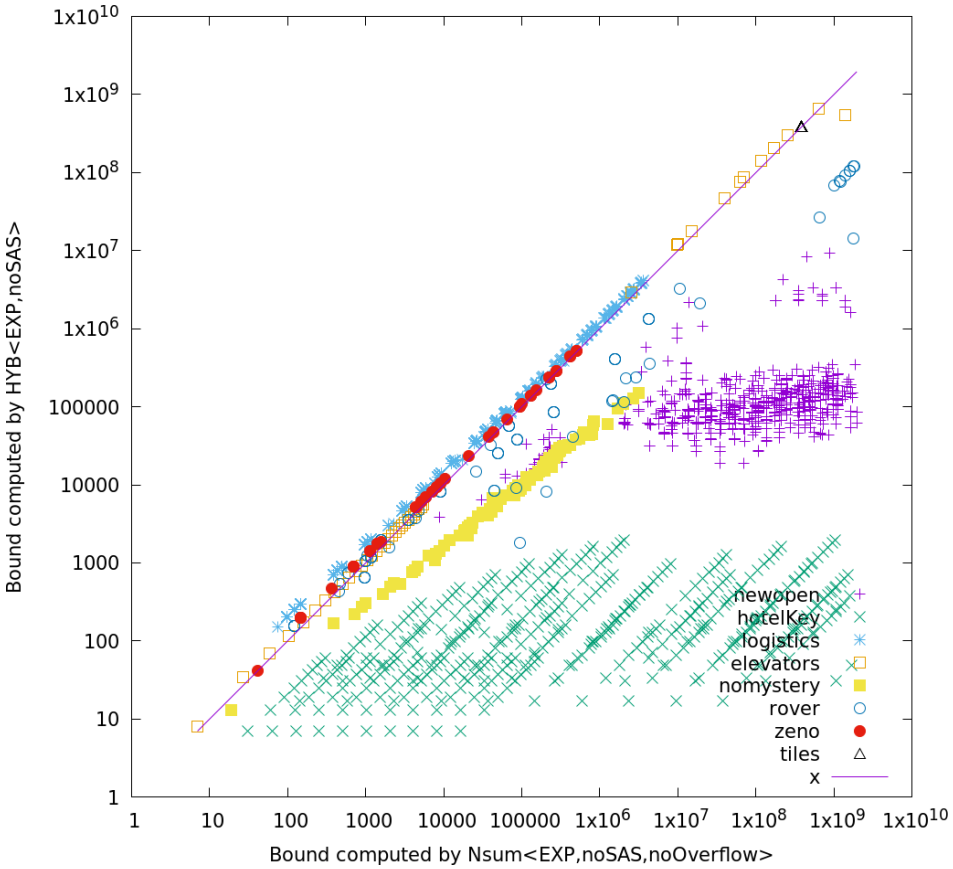


Fig. 20. Comparison of the bounding performance of N_{sum} and HYB on benchmarks: scatter plot of the bounds computed by HYB (horizontal axis) and the bounding algorithm N_{sum} (vertical).

it possible to perform dependency based analysis, when otherwise it is not possible. Having smaller base-cases in turn leads to the computation of smaller bounds because (i) in our experiments we use state space cardinality (i.e. the number of possible assignments for SAS+ state variables) as a base-case function for N_{sum}, HYB, and S_{max}, and (ii) since N_{sum}, HYB, and S_{max} do not always multiply the bounds for all base-cases (see the expression in Example 5, for instance).

The fact that HYB outperforms both N_{sum} and S_{max} begs the question of whether HYB is the best way of composing N_{sum} and S_{max}. Indeed, we need to formulate a tightness statement for HYB of the same style as the tightness statements in Theorem 3 and Theorem 12. To do that we need to first formulate what part of the input of HYB is relevant, i.e. what features of the given factored system does HYB use. Determining those features and devising the tightness statement is nontrivial as such tightness statements can be quite counter-intuitive. As stated earlier, it took quite some work to characterise the dependencies between projections and the recurrence diameters of the projections as the relevant part of the input to N_{sum} and to thus formulate the statement of Theorem 3, let alone proving it. However, we conjecture that such a tightness statement should somehow capture the interaction between the two different abstraction techniques used in HYB.

6 CONCLUSION

In this work we considered two compositional approaches to compute upper-bounds on diameters of propositionally factored transition systems. The first is based on the well-studied abstraction known as *projection*. Useful projections are identified based on a well-studied tool for analysing structure of transition systems, namely, *state-variable dependency*. The second approach is based on a new abstraction which we introduced, namely, *snapshotting*, and in that method snapshots are computed based on analysing *acyclicity in the state space*.

Our results are both negative and positive. We showed that the projection/variable dependencies based approach cannot be used to compositionally upper-bound the diameter and the recurrence diameter, which are the two most interesting topological properties, in the following sense: a concrete system diameter cannot be upper-bounded using projection diameters, and the same goes for the system recurrence diameter. However, we devised a new algorithm to compose projection recurrence diameters, producing upper bounds on concrete system diameters. That method experimentally outperforms existing one.

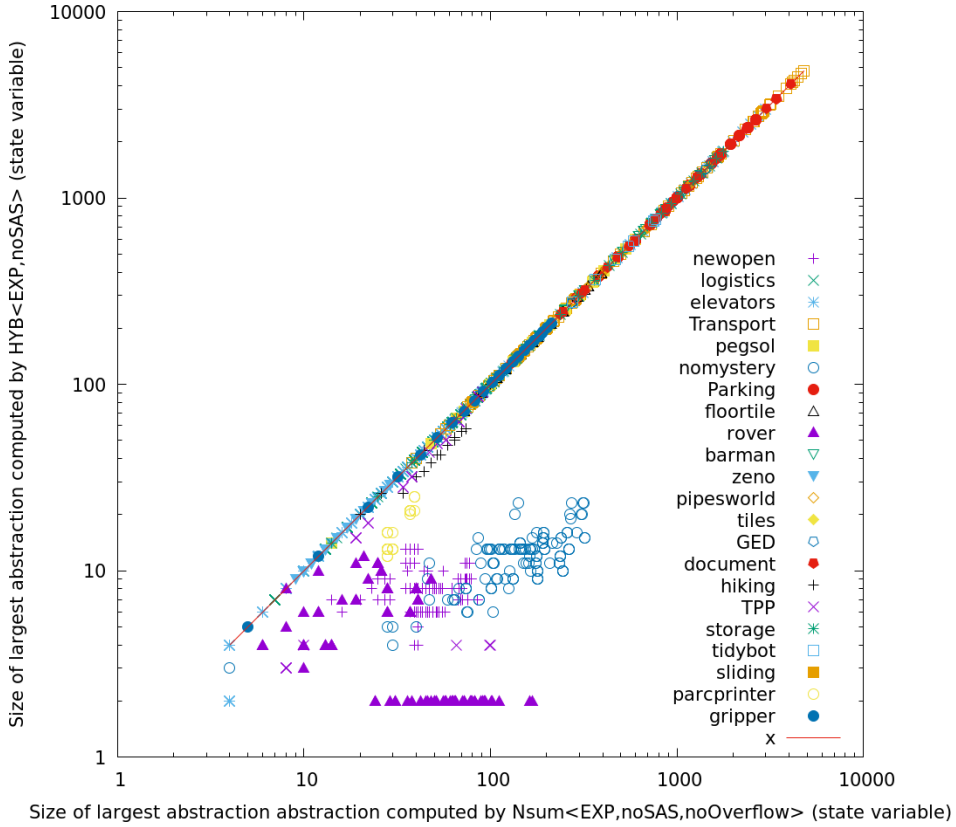


Fig. 21. Comparison of the bounding performance of N_{sum} and HYB on benchmarks: a scatter plot of the size (i.e. $|\mathcal{D}(\delta)|$) of the largest base-case using HYB (horizontal axis) and N_{sum} (vertical).

Other results include theorems which, in essence, imply that our new compositional approaches cannot be improved without analysing more system structure. For instance, the projection (resp. snapshotting) method can only be improved by analysing structures other than the variable dependencies (resp. state space structure). The fact that there is not a method better than ours for such a well-studied abstraction indicates the ongoing challenge of compositional bounding.

We circumvented those negative results by combining the two compositional bounding procedures into a hybrid procedure. That hybrid procedure produces very fine grained abstractions, and very tight bounds compared to its two constituents, which in turn outperform existing compositional bounding procedures. We did not show that our hybrid bounding algorithm is optimal, and therefore it would be interesting to investigate whether the projection and snapshotting based algorithms can be combined in a better way.

7 FUTURE WORK AND OPEN PROBLEMS

The most pressing item of future work is research into procedures that directly and exactly compute the topological properties contemplated in this work. Indeed, the reader will note that the compositional bounding procedures implemented and experimentally evaluated in this work never directly compute a recurrence or sublist diameter for abstractions. Rather, they employ loose upper bounds on abstractions' recurrence diameters as proxy values. These proxy bounds are obtained without significant computational effort, and which enable us to soundly bound the diameter. Of course, it is imperative that researchers investigate superior alternatives to these proxies moving forward, carefully trading off computation time for the tightness of the bounds obtained. Here, there are a host of avenues to explore that shall enable one to produce relatively tight bounds: (i) it remains to research and develop an efficient algorithm to compute the recurrence diameter for the compositional setting, (ii) in a similar vein, still tighter compositional bounds would be obtained using true sublist diameters rather than proxy values, and (iii) tighter bounds would be available using improved choice heuristics for decomposing problems into subproblems.

Other avenues of future study are of theoretical nature. For instance, there remains scope for further investigation into new topological properties which can be leveraged in the compositional bounding setting. The very recent work of the first author is an example in that direction [2]. Also, it remains open to understand why can the sublist diameter be bounded by projections' sublist diameters, while the diameter cannot be bounded by projections' diameters (and similarly, the recurrence diameter).

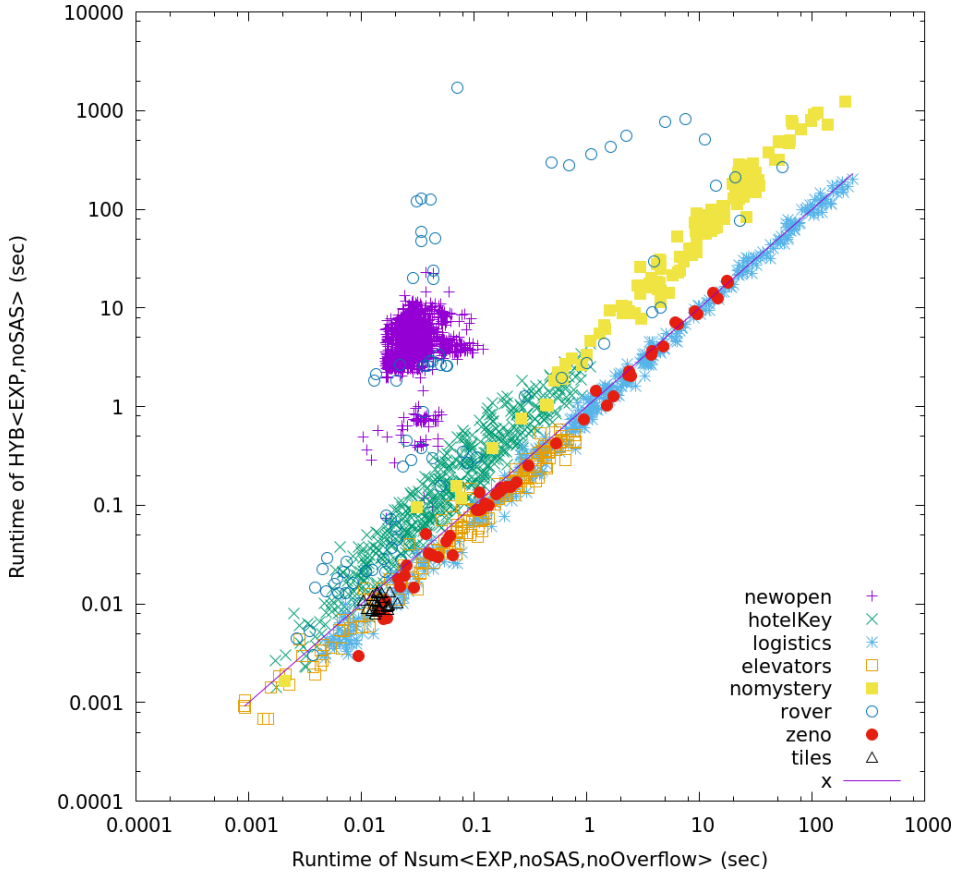


Fig. 22. Comparison of the bounding performance of N_{sum} and HYB on benchmarks: a scatter plot of the bound computation run-time by HYB (horizontal axis) and N_{sum} (vertical).

Finally, in practice the bounding problem is not quite as general as articulated here. In planning and model checking, typically a starting state is given, along with a goal condition describing what states correspond to goal states. Future work should develop solution length bounds for model checking and planning that take these constraints into consideration as was previously done by Baumgartner et al. [12] and Rintanen and Gretton [57].

REFERENCES

[1] Amir Abboud, Virginia Vassilevska Williams, and Joshua Wang. 2016. Approximation and fixed parameter subquadratic algorithms for radius and diameter in sparse graphs. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 377–391.

[2] Mohammad Abdulaziz. 2019. Plan-Length Bounds: Beyond 1-way Dependency. In *Proceedings of the Thirty-Third AAAI Conf. on Artificial Intelligence*. Association for the Advancement of Artificial Intelligence.

[3] Mohammad Abdulaziz, Charles Gretton, and Michael Norrish. 2014. Mechanising Theoretical Upper Bounds in Planning. In *Workshop on Knowledge Engineering for Planning and Scheduling*.

[4] Mohammad Abdulaziz, Charles Gretton, and Michael Norrish. 2015. Verified Over-Approximation of the Diameter of Propositionally Factored Transition Systems. In *Interactive Theorem Proving*. Springer, 1–16.

[5] Mohammad Abdulaziz, Charles Gretton, and Michael Norrish. 2017. A State Space Acyclicity Property for Exponentially Tighter Plan Length Bounds. In *International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI.

[6] Mohammad Abdulaziz, Michael Norrish, and Charles Gretton. 2015. Exploiting symmetries by planning for a descriptive quotient. In *Proc. of the 24th International Joint Conference on Artificial Intelligence, IJCAI*. 25–31.

[7] Donald Aingworth, Chandra Chekuri, Piotr Indyk, and Rajeev Motwani. 1999. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM J. Comput.* 28, 4 (1999), 1167–1181.

[8] Noga Alon, Zvi Galil, and Oded Margalit. 1997. On the exponent of the all pairs shortest path problem. *J. Comput. System Sci.* 54, 2 (1997), 255–262.

[9] Noga Alon, Raphael Yuster, and Uri Zwick. 1995. Color-coding. *Journal of the ACM (JACM)* 42, 4 (1995), 844–856.

[10] Eyal Amir and Barbara Engelhardt. 2003. Factored planning. In *IJCAI*, Vol. 3. Citeseer, 929–935.

[11] Christer Bäckström and Bernhard Nebel. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11, 4 (1995), 625–655.

[12] Jason Baumgartner, Andreas Kuehlmann, and Jacob Abraham. 2002. Property Checking Via Structural Analysis. In *Computer Aided Verification*. Springer, 151–165.

[13] Sergey Berezin, Sérgio Campos, and Edmund M Clarke. 1998. Compositional reasoning in model checking. In *Compositionality: The Significant Difference*. Springer, 81–102.

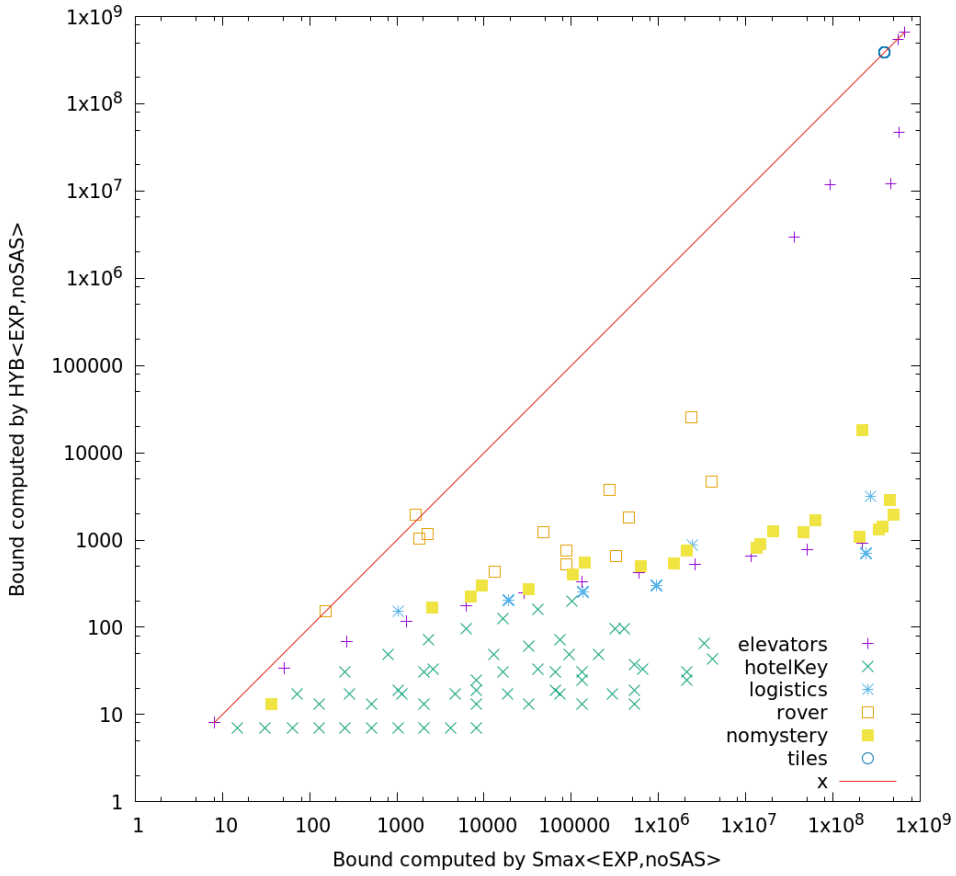


Fig. 23. Comparison of the bounding performance of S_{max} and HYB on benchmarks: scatter plot of the bounds computed by HYB (horizontal axis) and the bounding algorithm S_{max} (vertical).

[14] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded model checking. *Advances in Computers* 58 (2003), 117–148.

[15] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *TACAS*. 193–207.

[16] Andreas Björklund and Thore Husfeldt. 2003. Finding a path of superlogarithmic length. *SIAM J. Comput.* 32, 6 (2003), 1395–1402.

[17] Andreas Björklund, Thore Husfeldt, and Sanjeev Khanna. 2004. Approximating longest directed paths and cycles. In *International Colloquium on Automata, Languages, and Programming*. Springer, 222–233.

[18] Jasmin Christian Blanchette and Tobias Nipkow. 2010. Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In *Interactive Theorem Proving, First International Conference, ITP 2010*. 131–146. https://doi.org/10.1007/978-3-642-14052-5_11

[19] R. I. Brafman and C. Domshlak. 2003. Structure and Complexity in Planning with Unary Operators. *Journal of Artificial Intelligence Research* 18 (2003), 315–349.

[20] Ronen I Brafman and Carmel Domshlak. 2006. Factored planning: How, when, and when not. In *AAAI*, Vol. 6. 809–814.

[21] Daniel Bundala, Joël Ouaknine, and James Worrell. 2012. On the magnitude of completeness thresholds in bounded model checking. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*. IEEE Computer Society, 155–164.

[22] T. Bylander. 1994. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence* 69, 1-2 (1994), 165–204.

[23] Michael L. Case, Hari Mony, Jason Baumgartner, and Robert Kanzelman. 2009. Enhanced verification by temporal decomposition. In *FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*. 17–24.

[24] Timothy M Chan. 2010. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM J. Comput.* 39, 5 (2010), 2075–2089.

[25] Shiri Chechik, Daniel H Larkin, Liam Roditty, Grant Schoenebeck, Robert E Tarjan, and Virginia Vassilevska Williams. 2014. Better approximation algorithms for the graph diameter. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 1041–1052.

[26] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2001. Progress on the state explosion problem in model checking. In *Informatics*. Springer, 176–194.

[27] Edmund M Clarke, E Allen Emerson, and Joseph Sifakis. 2009. Turing lecture: model checking–algorithmic verification and debugging. *Commun. ACM* 52, 11 (2009), 74–84.

[28] Peter Dankelmann. 2005. The diameter of directed graphs. *Journal of Combinatorial Theory, Series B* 94, 1 (2005), 183–186.

[29] Peter Dankelmann and Michael Dorfling. 2016. Diameter and maximum degree in Eulerian digraphs. *Discrete Mathematics* 339, 4 (2016), 1355–1361.

[30] Peter Dankelmann and Lutz Volkmann. 2010. The diameter of almost Eulerian digraphs. *the electronic journal of combinatorics* 17, 1 (2010), R157.

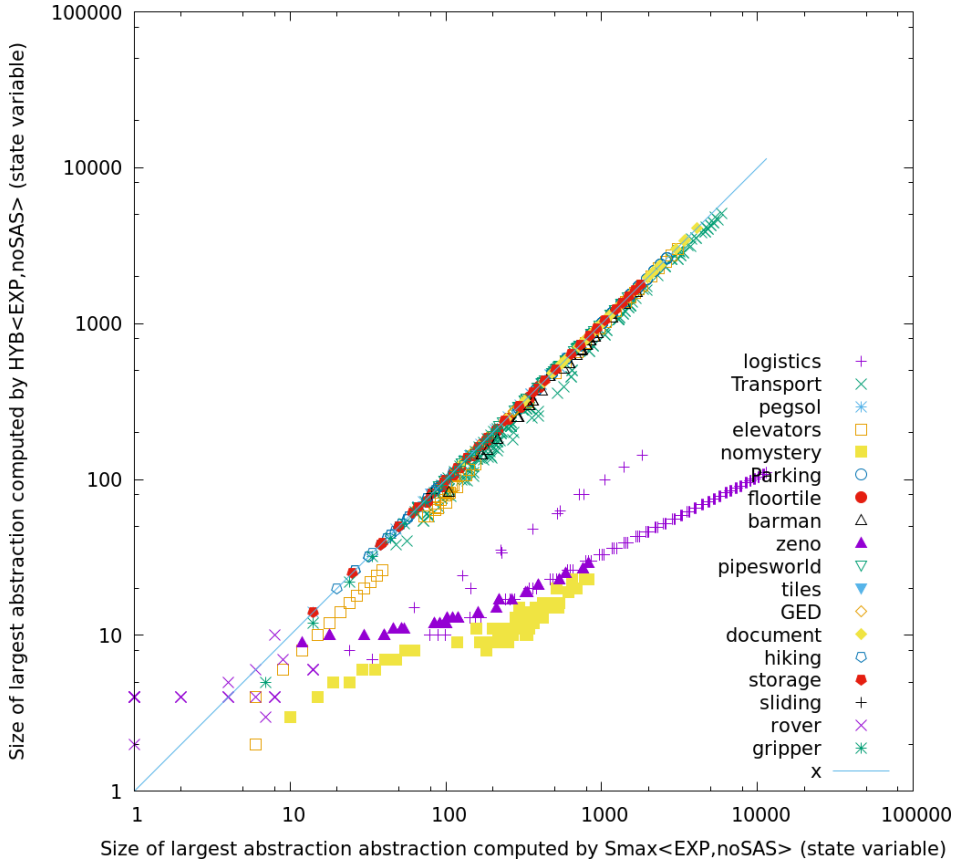


Fig. 24. Comparison of the bounding performance of S_{max} and HYB on benchmarks: a scatter plot of the size (i.e. $|\mathcal{D}(\delta)|$) of the largest base-case using HYB (horizontal axis) and S_{max} (vertical).

[31] Paul Erdős, Janos Pach, Richard Pollack, and Zsolt Tuza. 1989. Radius, diameter, and minimum degree. *Journal of Combinatorial Theory, Series B* 47, 1 (1989), 73–79.

[32] Joan Feigenbaum, Sampath Kannan, Moshe Y Vardi, and Mahesh Viswanathan. 1998. Complexity of problems on graphs represented as OBDDs. In *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 216–226.

[33] Richard E Fikes and Nils J Nilsson. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2, 3-4 (1971), 189–208.

[34] Michael L Fredman. 1976. New bounds on the complexity of the shortest path problem. *SIAM J. Comput.* 5, 1 (1976), 83–89.

[35] Hana Galperin and Avi Wigderson. 1983. Succinct representations of graphs. *Information and Control* 56, 3 (1983), 183–198.

[36] Malte Helmert. 2004. A Planning Heuristic Based on Causal Graph Analysis. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*. 161–170.

[37] Malte Helmert. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26 (2006), 191–246.

[38] Edith Hemaspaandra, Lane A Hemaspaandra, Till Tantau, and Osamu Watanabe. 2010. On the complexity of kings. *Theoretical Computer Science* 411, 4-5 (2010), 783–798.

[39] Daniel Jackson. 2006. *Software Abstractions: Logic, Language, and Analysis*. MIT Press.

[40] H. A. Kautz and B. Selman. 1992. Planning as Satisfiability. In *ECAI*. 359–363.

[41] Elena Kelareva, Olivier Buffet, Jinbo Huang, and Sylvie Thiébaux. 2007. Factored Planning Using Decomposition Trees.. In *IJCAI*. 1942–1947.

[42] C. A. Knoblock. 1994. Automatically Generating Abstractions for Planning. *Artificial Intelligence* 68, 2 (1994), 243–302.

[43] AV Knyazev. 1987. Diameters of pseudosymmetric graphs. *Mathematical Notes* 41, 6 (1987), 473–482.

[44] Daniel Kroening. 2006. Computing over-approximations with bounded model checking. *Electronic Notes in Theoretical Computer Science* 144, 1 (2006), 79–92.

[45] Daniel Kroening, Joël Ouaknine, Ofer Strichman, Thomas Wahl, and James Worrell. 2011. Linear completeness thresholds for bounded model checking. In *Computer Aided Verification*. Springer, 557–572.

[46] Daniel Kroening and Ofer Strichman. 2003. Efficient Computation of Recurrence Diameters. In *VMCAI*. 298–309.

[47] Vladimir Lifschitz. 1987. On the semantics of STRIPS. In *Reasoning About Actions & Plans*. Elsevier, 1–9.

[48] Antonio Lozano and José L Balcázar. 1989. The complexity of graph problems for succinctly represented graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 277–286.

[49] John McCarthy and Patrick J Hayes. 1981. Some philosophical problems from the standpoint of artificial intelligence. In *Readings in artificial intelligence*. Elsevier, 431–450.

[50] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. 1998. *PDDL: The Planning Domain Definition Language*. Technical Report. CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

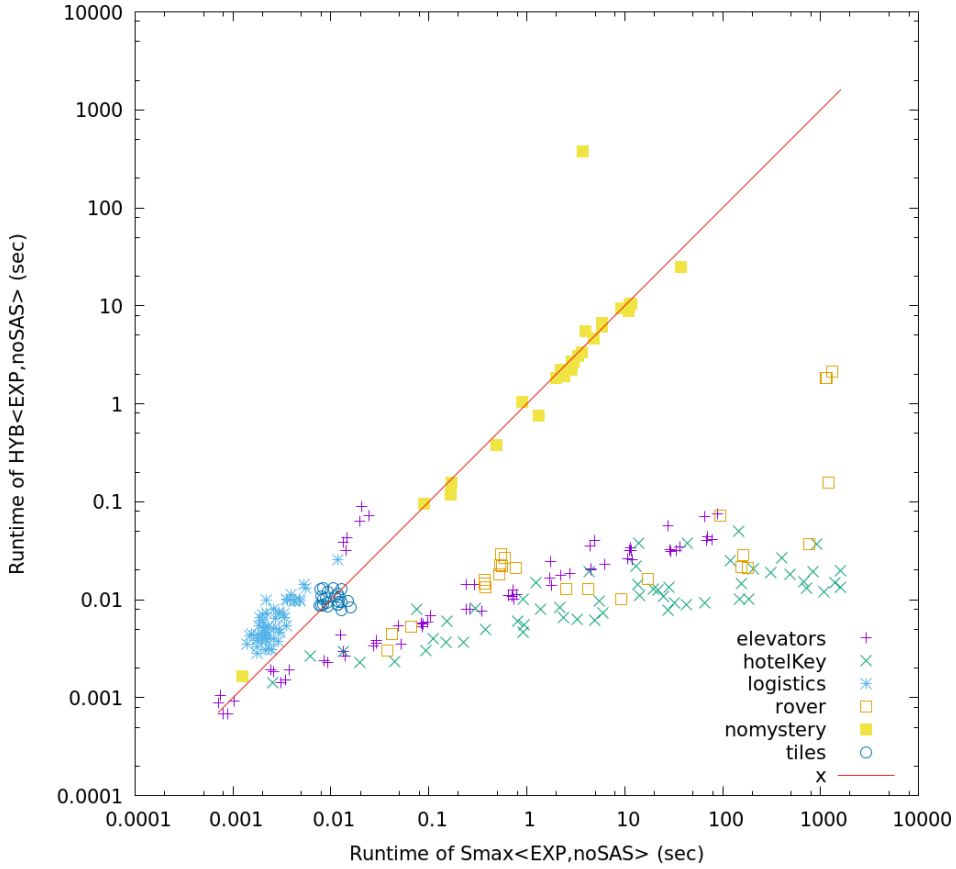


Fig. 25. Comparison of the bounding performance of S_{max} and HYB on benchmarks: a scatter plot of the bound computation run-time by HYB (horizontal axis) and S_{max} (vertical).

[51] Kenneth L McMillan. 1993. Symbolic model checking. In *Symbolic Model Checking*. Springer, 25–60.

[52] John W Moon et al. 1965. On the diameter of a graph. *The Michigan Mathematical Journal* 12, 3 (1965), 349–351.

[53] Tobias Nipkow. 2006. Verifying a Hotel Key Card System. In *Theoretical Aspects of Computing (ICTAC 2006) (Lecture Notes in Computer Science)*, K. Barkaoui, A. Cavalcanti, and A. Cerone (Eds.), Vol. 4281. Springer. Invited paper.

[54] Christos H Papadimitriou and Mihalis Yannakakis. 1986. A note on succinct representations of graphs. *Information and Control* 71, 3 (1986), 181–185.

[55] Panos M Pardalos and Athanasios Migdalas. 2004. A note on the complexity of longest path problems related to graph coloring. *Applied Mathematics Letters* 17, 1 (2004), 13–15.

[56] Jussi Rintanen. 2012. Planning as satisfiability: Heuristics. *Artificial Intelligence* 193 (2012), 45–86.

[57] Jussi Rintanen and Charles Orgill Gretton. 2013. Computing Upper Bounds on Lengths of Transition Sequences. In *International Joint Conference on Artificial Intelligence*.

[58] Liam Roditty and Virginia Vassilevska Williams. 2013. Fast approximation algorithms for the diameter and radius of sparse graphs. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*. ACM, 515–524.

[59] Jendrik Seipp, Florian Pommerening, Silvan Sievers, Martin Wehrle, Chris Fawcett, and Yusra Alkhazraji. 2014. Fast Downward Aidos. *International Planning Competition* (2014).

[60] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*. 108–125. https://doi.org/10.1007/3-540-40922-X_8

[61] Silvan Sievers, Martin Wehrle, Malte Helmert, Alexander Shleyfman, and Michael Katz. 2015. Factored Symmetries for Merge-and-Shrink Abstractions. In *Proc. 29th National Conf. on Artificial Intelligence*. AAAI Press.

[62] José Soares. 1992. Maximum diameter of regular digraphs. *Journal of Graph Theory* 16, 5 (1992), 437–450.

[63] Brian C. Williams and P. Pandurang Nayak. 1997. A Reactive Planner for a Model-based Executive. In *International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, 1178–1185.

[64] Raphael Yuster. 2010. Computing the diameter polynomially faster than APSP. *arXiv preprint arXiv:1011.6181* (2010).