Elimination of Detached Regions in Dependency Graph Verification

Peter Gjøl Jensen, Kim Guldstrand Larsen, Jiří Srba, and Nikolaj Jensen Ulrik

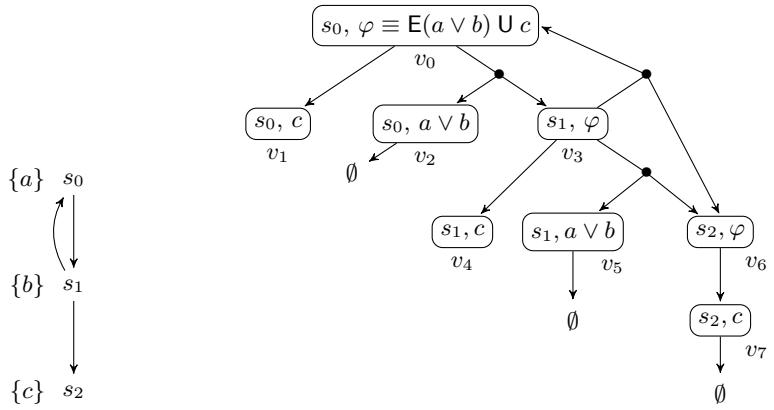
Department of Computer Science
Aalborg University, Denmark
{pgj,kgl,srba,njul}@cs.aau.dk

Abstract. The formalism of dependency graphs by Liu and Smolka is a well-established method for on-the-fly computation of fixed points over Boolean domains with numerous applications in e.g. model checking, game synthesis, bisimulation checking and others. The original Liu and Smolka on-the-fly algorithm runs in linear time, and several extensions and improvements to this algorithm have recently been studied, including the notion of negation edges, certain-zero early termination as well as extensions towards abstract dependency graphs. We present a novel improvement for computing the least fixed-point assignment on dependency graphs, with the goal of avoiding the exploration of detached subgraphs that cannot contribute to the assignment value of the root node. We also experimentally evaluate different ways of resolving nondeterminism in the algorithm and execute experiments on CTL formulae from the annual Petri net model checking contest as well as on synthesis problems for Petri games. We find out that our algorithm significantly improves the state-of-the-art.

1 Introduction

Within the fields of formal verification and model checking, the efficient computation of fixed points is of great importance for solving many problems, such as bisimulation checking [26] or model checking Computation Tree Logic (CTL) [7] or the modal μ-calculus [22]. Unfortunately, the naive approaches to computing such fixed points are prone to state space explosion as they require the entire state space to be available before verification can be done, which is often not a feasible option.

The formalism of Dependency Graphs (DGs), developed by Liu and Smolka [24], is a general formalism that encodes dependencies among the different nodes in the graph and allows for efficient, on-the-fly fixed-point computations. A *dependency graph* is a directed graph with hyperedges, i.e. edges that can have multiple targets. Figure 1b shows an example of a dependency graph that encodes the problem of checking whether the Kripke structure shown in Figure 1a satisfies the CTL formula $\mathsf{E}(a \vee b)\ \mathsf{U}\ c$ using the encoding of [10]. Hyperedges are drawn as branching from a mid-point, for example there is a hyperedge from v_0 with v_2 and v_3 as targets. An *assignment* of the dependency



(a) Kripke Structure (b) DG encoding of model checking $s_0 \models \mathsf{E}(a \vee b)\,\mathsf{U}\,c$

Iteration	$A(v_0)$	$A(v_1)$	$A(v_2)$	$A(v_3)$	$A(v_4)$	$A(v_5)$	$A(v_6)$	$A(v_7)$
0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0	1
2	0	0	1	0	0	1	1	1
3	0	0	1	1	0	1	1	1
4	1	0	1	1	0	1	1	1
5	1	0	1	1	0	1	1	1

(c) Fixed-point computation on the DG

Fig. 1: Example of dependency graph encoding a CTL model checking problem

graph is a function that assigns to each node in the graph the value 0 (false) or 1 (true). The goal of dependency graph verification is to compute the minimum fixed-point assignment A (given by Tarski's fixed point theorem [27]). This computation is shown in Figure 1c by starting from the assignment where all nodes have the value 0 and iteratively performing the following saturation procedure: if there is a hyperedge from a node v such that all the targets of the hyperedge already have the value 1, then v also gets the value 1. A hyperedge with the empty set of targets is interpreted as a vacuous truth and propagates the value 1 to its source, whereas if a node does not have any outgoing hyperedges, its value remains 0. We can see that the saturation procedure stabilizes after the fourth iteration and reaches the minimum fixed point where $A(v_0) = 1$, meaning that $s_0 \models \mathsf{E}(a \vee b)\,\mathsf{U}\,c$.

The success of dependency graphs is due to an efficient, on-the-fly and local algorithm for computing the least fixed-point assignment [24], meaning that the full state space is not necessarily needed before the fixed-point computation can start. More recently, a distributed version of the algorithm was developed [8], and the formalism of dependency graphs was extended to support negation edges [9].

Dependency graphs have shown great success in bisimulation checking [2] and model checking of recursive Hennessy-Milner logic [2] and CTL [10], and variations of dependency graphs have been applied to strategy synthesis of Timed Games [6] and model checking of weighted CTL [18], probabilistic CTL [25] and the modal μ-calculus [23]. For an overview of different applications of dependency graphs to model checking, consult [13].

In this paper, we present a novel improvement to the dependency graph verification algorithm. Our contribution is two-fold. First, we extend the dependency graph algorithm with an optimization that allows us to prune detached regions of the dependency graph (parts of the graph that were scheduled for exploration but which during the fixed-point computation became irrelevant for the fixed-point assigment of the root node) and hence to improve the running time of the computation. We experimentally demonstrate that this improvement significantly helps in solving CTL model checking and games synthesis problems. Second, we investigate and discuss in detail the impact of different choices of implementation of the general nondeterministic algorithm, in particular wrt. the search order used by the algorithm, and provide an extensive experimential evaluation.

Related Work. Dependency graphs found great success through Liu and Smolka's on-the-fly, linear-time algorithm for computing fixed-points [24]. This algorithm includes early termination when a value of 1 is computed for the designated root node of the dependency graph. Early termination was later extended with the certain-zero optimization [10], which also allows propagation of the value 0 in cases where it provably cannot be improved anymore. Other improvements and extensions include negation edges [10], distributed verification [8] of dependency graphs and most recently the development of abstract dependency graphs [15]. However, most of these additions do not address the issue that the algorithm has a risk of spending time exploring detached regions of the dependency graphs, i.e. regions where improving the fixed-point assignment will not lead to improvement of the root node value. The algorithm of [15] attempts to address the detached regions issue, however, in a manner which proved inefficient in practice (requiring a potentially expensive recursive propagation of information about detached regions). In contrast, our method of addressing the problem captures a high number of detached regions but with a minimal overhead.

Dependency graphs are used by the verification engines of several tools. The tool Caal [2] uses dependency graphs for bisimulation checking and model checking of recursive Hennessy-Milner logic on CCS [26] processes. Uppaal TiGa [3] supports controller synthesis for games on Timed Automata [1], and the tool Tapaal [11] supports strategy synthesis for Petri net games as well as CTL model checking on Petri nets, all using dependency graphs. Additionally, dependency graphs have been applied to probabilistic [25] and weighted [18] domains. The formalism of Boolean Equations Systems, which are very similar to dependency graphs, are used for various verification tasks within the μCRL2 [5] and CADP [16] tools. We believe that our technique for detached regions detection can contribute to improved verification performance of such tools.

In [14], it was observed that the basic principles behind the dependency graph algorithms used in different application areas are not very different, except that they generalize the assignments from Boolean values to more complex domains, for example to weighted domains. This prompted the theory of abstract dependency graphs, which encompasse all the aforementioned application areas. In [15] this formalism is extended to include nonmonotonicity, which generalises the notion of negation edge which was developed for CTL model checking in [10]. Our work focuses on the traditional dependency graph formalism with assignments over Boolean domain, leaving the extension towards more general domains for future work.

2 Dependency Graphs

We shall now introduce the formalism of dependency graphs together with Boolean valued assignments to its nodes and define the minimum fixed-point assignment.

Definition 1 (Dependency graph). *A* DG *is a pair $G = (V, E)$ where V is a finite, nonempty set of* configurations *(nodes) and $E \subseteq V \times 2^V$ is a set of* hyperedges.

For a hyperedge $e = (v, T) \in E$, we call v the *source* of e and $u \in T$ the *targets* of e. We let $\mathrm{sucs}(v)$ denote the set of hyperedges with v as the source configuration, i.e. $\mathrm{sucs}(v) = \{(v, T) \mid (v, T) \in E\}$.

Each configuration of a DG is associated with a Boolean function defined by a disjunction over each hyperedge, and each hyperedge is a conjunction over the set of target configurations. The semantics of a DG is given by Boolean-valued assignments such that the Boolean function on each configuration is satisfied.

Definition 2 (Assignment). *Let $G = (V, E)$ be a DG. An* assignment *on G is a function $A : V \to \{0, 1\}$, and the set of all assignments on G is denoted $\mathcal{A}^G$. For $A, A' \in \mathcal{A}^G$, let $A \preceq A'$ iff for all $v \in V$ we have $A(v) \leq A'(v)$, and let $A_\bot$ denote the assignment such that $A_\bot(v) = 0$ for all $v \in V$. An assignment A on G is a* solution *(fixed-point assignment) iff for all $v \in V$ we have*

$$A(v) = \bigvee_{(v,T)\in E} \bigwedge_{u\in T} A(u)$$

where we define the empty conjunction to be 1 *(true) and the empty disjunction to be* 0 *(false).*

Note that for any DG G, $(\mathcal{A}^G, \preceq)$ is a complete lattice with least element $A_\bot$. We can express solutions as fixed points of the function $F : \mathcal{A}^G \to \mathcal{A}^G$ defined by

$$F(A)(v) = A(v) \vee \bigvee_{(v,T)\in E} \bigwedge_{u\in T} A(u) \ . \tag{1}$$

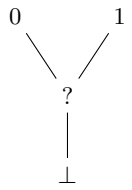


Fig. 2: Illustration of the certain-zero partial order.

Clearly F is monotonic, so by Tarski's fixed-point theorem [27] the function F has a minimum fixed point, and furthermore there exists an $i \geq 0$ such that $F^i(A_\perp) = F^{i+1}(A_\perp)$, in which case $F^i(A_\perp)$ is a minimal solution on G. We write $A_{\min}$ to refer to the minimum fixed-point solution $F^i(A_\perp)$. An example of the iterative computation of the minimum fixed point (also referred to as the global algorithm) is depicted in Figure 1c.

Remark 1. For sake of simplicity, we omit methods of dealing with negation in verified properties. The paper [10] extends DGs with negation edges and shows a correct encoding of CTL properties, assuming that negation edges are explored exactly when all configurations below have been explored, and only then. Our experimental evaluation includes a benchmark of CTL queries, so our implementation supports negation edges. As also shown in [15], this works near trivially if the DG is explored depth-first. We refer to [10] and [15] for a proper treatment of negation.

3 Local Algorithm with Detached Regions Detection

For a DG $G = (V, E)$, our goal is to compute $A_{\min}(v_0)$ for a distinguished configuration $v_0 \in V$. The original algorithm for doing this efficiently was published by Liu and Smolka [24] and later on extended with the notion of certain-zero [10], demonstrating significant improvement in verification times. The Liu and Smolka algorithm explores the dependency graph in a forward manner, starting from the root v_0, and backpropagates the value 1 along the hyperedges whenever possible. The certain-zero extension allows the algorithm to also backpropagate the value 0 when it can prove that the value of a configuration cannot be improved to 1 anymore. In effect, the certain-zero improvement extends the Boolean domain to the simple partial order of assignment values depicted in Figure 2. Here, the value $\perp$ means that a configuration has not been discovered yet, ? means that a discovered configuration waits to be processed but its value is not determined yet, and the values 1 and 0 are final configuration values in the minimum fixed-point assignment. Our new improvement to the algorithm is an additional check whether a configuration can be ignored in the situations where its backpropagation only affects configurations that are already fully resolved.

Algorithm 1 shows the procedure in pseudocode. If all lines highlighted in gray are removed, then the algorithm corresponds to the original algorithm by

Algorithm 1 Improved certain-zero algorithm

Input: DG $G = (V, E)$ and a root configuration $v_0 \in V$.
Output: Least fixed-point value $A_{\min}(v_0)$.

```
 1: W ← succs(v0)
 2: for all v ∈ V do
 3:     D(v) ← ∅
 4:     A(v) ← ⊥
 5: A(v0) ← ?
 6: while W ≠ ∅ do
 7:     e ← (v, T) ∈ W
 8:     W ← W \ {e}
 9:     if v ≠ v0 and (A(v) ∈ {0,1} or ∀(w, T') ∈ D(v) . A(w) ∈ {0,1}) then
10:         if A(v) ∉ {0,1} then A(v) ← ⊥
11:         goto line 6
12:     if ∀u ∈ T . A(u) = 1 then
13:         if v = v0 then return 1
14:         if A(v) ≠ 1 then
15:             A(v) ← 1; W ← W ∪ D(v)
16:     elseif ∃u ∈ T . A(u) = 0
17:         if A(v) ≠ 0 then
18:             E ← E \ {e}
19:             if succs(v) = ∅ then
20:                 if v = v0 then return 0
21:                 A(v) ← 0; W ← W ∪ D(v)
22:     else
23:         pick u ∈ T such that A(u) ∉ {0,1}
24:         D(u) ← D(u) ∪ {e}
25:         if A(u) = ⊥ then
26:             A(u) ← ?
27:             if succs(u) = ∅ then
28:                 if u = v0 then return 0
29:                 A(u) ← 0; W ← W ∪ D(u)
30:             else
31:                 W ← W ∪ {(u, T') | (u, T') ∈ E}
32: return 0                          ▷ v0 did not receive value, so must be 0.
```

Liu and Smolka. If the light gray lines (lines 16–21 and 27–29) are included, we obtain a single-core version of the certain zero optimization suggested in [10]. The dark gray lines 9–11 are our additional optimization to the algorithm.

The algorithm maintains a waiting list of hyperedges to be explored as well as for each $v \in V$ a set $D(v)$ of parent hyperedges that are dependent on the value of v. In the initialization step, all outgoing edges $(v_0, T) \in E$ from v_0 are pushed to the waiting-list, and $D(v)$ is initialised to the empty set for all $v \in V$. During each iteration of the **while**-loop at line 6, an edge $e = (v, T)$ is picked from the waiting list. If all the target configurations $u \in T$ have assignment 1 on line 12,

then the value is assigned to $A(v)$ and the set $D(v)$ of hyperedges dependent on v are inserted into the waiting list. If some target configuration $u \in T$ has value 0 (certain-zero) on line 16 then the edge is deleted from the DG, and if now v has no outgoing edge, a value of 0 is assigned to v and all dependents in $D(v)$ are added to the waiting-list. If the configuration that was just assigned a final value is v_0 then we can terminate early, on lines 13 and 20. As observed in [13], the checks on lines 14 and 17 are important to ensure termination of the algorithm. If neither of the above cases were true, there must be some successor node $u \in T$ such that $A(u) \in \{?, \bot\}$, among which we select one on line 23. We update $D(u)$ to also contain e, and if $A(u) = \bot$, i.e. it is not discovered yet, we set $A(u) \leftarrow ?$ and push all outgoing edges $(u, T') \in E$ to the waiting list.

The lines 9–11 are our proposed improvement. The motivation is that we want to avoid spending time exploring detached regions of the dependency graph, i.e. configurations that cannot possibly help conclude the value of v_0 at the current point in execution. The test we implement here checks whether all edges $e = (u, T) \in D(v)$ have a final value 1 or 0 assigned to their source, as this means the value of v will not contribute to any further knowledge about the fixed-point assignment on G. In this case, we simply skip the edge, remove it from the waiting list. Since there may be previously unknown edges that will need the value $A(v)$ in a later iteration, we set $A(v) = \bot$ to indicate that it should be expanded again when found in the future during the forward search.

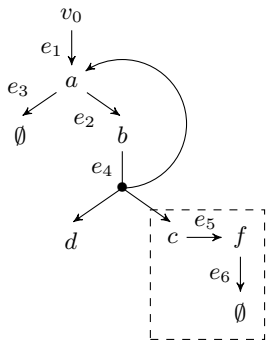
Remark 2. We can formalize the notion of detached regions as follows. Let $Dep : V \times V$ be a relation such that for $v, u \in V$, $Dep(v, u)$ holds if and only if there is $(u, T) \in D(v)$ such that $A(u) \notin \{0, 1\}$. We say that $v \in V$ is *detached* if $(v, v_0) \notin Dep^*$, where Dep^* is the transitive closure of Dep. Conversely, $v \in V$ is not detached if $(v, v_0) \in Dep^*$, indicating that there is some chain of dependencies from v to v_0 that does not contain any configurations that are assigned 0 or 1.

The test on lines 9–11 does *not* detect all detached regions. One approach to detect all detached regions can be obtained with a slight modification to Algorithm 1 of [15], calling UpdateDependents on line 11 of that algorithm. We implemented this in our tool, measuring significantly worse results than even the Certain-Zero algorithm, as we will detail in Section 5.

Example 1. Figure 3 shows different means of computing the least fixed-point value $A_{\min}(v_0)$ of the DG depicted in Figure 3a. Figure 3b shows the computation of the minimum fixed-point assignment using the global algorithm given by Equation 1. Each row corresponds to one iteration of the algorithm, terminating on iteration 4 since $F^4(A_\bot) = F^3(A_\bot)$. We notice that in the global algorithm we iterate over all nodes in the dependency graph.

Figure 3c shows the computation using the local algorithm with the certain-zero extension and Figure 3d shows the computation using our improved version of the local algorithm, each with the same search order (the choice being the right-most available configuration, and nodes v with $A(v) = ?$ have priority).

For iterations 1–4, the certain-zero algorithm simply searches through the graph in a forward fashion. Once e_3 is checked in iteration 3, we set $A(a) = 1$



(a) Dependency graph (the dashed region eventually becomes detached)

Iteration	$A(v_0)$	$A(a)$	$A(b)$	$A(c)$	$A(d)$	$A(f)$
0	0	0	0	0	0	0
1	0	1	0	0	0	1
2	1	1	0	1	0	1
3	1	1	0	1	0	1
4	1	1	0	1	0	1

(b) Execution of the global algorithm

Iteration	W	$e \in (v,T)$	$u \in T$	$A(v_0)$	$A(a)$	$A(b)$	$A(c)$	$A(d)$	$A(f)$
0	$\{e_1\}$			?	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
1	$\{e_1\}$	e_1	a	?	?	$\bot$	$\bot$	$\bot$	$\bot$
2	$\{e_2, e_3\}$	e_2	b	?	?	?	$\bot$	$\bot$	$\bot$
3	$\{e_3, e_4\}$	e_4	a	?	?	?	$\bot$	$\bot$	$\bot$
4	$\{e_3\}$	e_3	—	?	1	?	$\bot$	$\bot$	$\bot$
5	$\{e_4, e_1\}$	e_4	c	?	1	?	?	$\bot$	$\bot$
6	$\{e_1, e_5\}$	e_5	f	?	1	?	?	$\bot$	?
7	$\{e_1, e_6\}$	e_6	—	?	1	?	?	$\bot$	1
8	$\{e_1, e_5\}$	e_5	—	?	1	?	1	$\bot$	1
9	$\{e_1, e_4\}$	e_4	d	?	1	?	1	0	1
10	$\{e_1, e_4\}$	e_4	—	?	1	0	1	0	1
11	$\{e_1\}$	e_1	—	1	1	?	1	0	1

(c) Iterations of the local certain-zero algorithm

Iteration	W	$e \in (v,T)$	$u \in T$	$A(v_0)$	$A(a)$	$A(b)$	$A(c)$	$A(d)$	$A(f)$
0	$\{e_1\}$			?	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
1	$\{e_1\}$	e_1	a	?	?	$\bot$	$\bot$	$\bot$	$\bot$
2	$\{e_2, e_3\}$	e_2	b	?	?	?	$\bot$	$\bot$	$\bot$
3	$\{e_3, e_4\}$	e_4	a	?	?	?	$\bot$	$\bot$	$\bot$
4	$\{e_3\}$	e_3	—	?	1	?	$\bot$	$\bot$	$\bot$
5	$\{e_4, e_1\}$	e_4	—	?	1	?	$\bot$	$\bot$	$\bot$
6	$\{e_1\}$	e_1	—	1	1	?	$\bot$	$\bot$	$\bot$

(d) Iterations of our improved local algorithm

Fig. 3: Dependency graph and fixed-point computations. The search order is depth-first, choosing the right-most hyperedge and rightmost configuration in each step.

and re-add edges e_1 and e_4 to the waiting list in hopes that the value can be backpropagated further. The certain-zero algorithm picks the configuration c to be expanded. This enters the region consisting of the configurations c and f, which is detached since all paths back to v_0 contain some configuration u such that $A(u) \in \{0, 1\}$. From c the search continues through edges e_5 and e_6, which eventually backpropagates the value 1 to $A(c)$. Finally on iteration 9, the configuration d is expanded and receives a value of certain-zero since it has no outgoing hyperedges, and on iteration 10 this is backpropagated to b.

On the other hand, our improved algorithm detects on iteration 5 that all configurations depending on b have been assigned, so there is no need to evaluate the edge e_4. Thus it avoids exploring the detached region, saving a total of 5 iterations over the certain-zero algorithm. We also notice that while the global algorithm takes fewer iterations to complete than the local algorithms, in each iteration the assignment of each configuration is checked, which is an expensive operation if the DG is large.

4 Correctness of the Algorithm

We now proceed to show correctness of the full updated algorithm, i.e. the algorithm including all lines, starting with some technical lemmas. The proofs are adapted from the certain-zero algorithm [10], with several differences that originate from the improvements in our algorithm, in particular from the possibility to assign the value $\perp$ to nodes that already had the value ? before.

Lemma 1. *For each $v \in V$, $A(v)$ is assigned a value $x \in \{0, 1\}$ at most once.*

Proof. First we observe that for any configuration $v \in V$, whenever $A(v) \in \{0, 1\}$, neither ? nor $\perp$ are ever assigned to it, since the value ? is only assigned on line 26, which can only be reached if $A(v) = \perp$, and the value $\perp$ is only assigned on line 10, but only if $A(v) \notin \{0, 1\}$.

Any $v \in V$ is assigned a value $x \in \{0, 1\}$ on one of lines 15, 21, and 29. Consider the case of line 15. For this line to be reached, the condition on line 9 must evaluate to false, which means either $v = v_0$ or $A(v) \notin \{0, 1\}$. If $A(v) \notin \{0, 1\}$ then clearly the configuration v was not assigned before, so after assignment it has been assigned exactly once. If $v = v_0$, then the condition on line 13 is true, causing early termination and thus no further assignment of values to v. A similar argument holds for both line 21 and 29. □

Lemma 2. *Each edge $e = (v, T) \in E$ is picked on line 8 at most $\mathcal{O}(|V| + |E|)$ times.*

Proof. Let $e = (v, T) \in E$ be any edge. There are two ways for e to be added to W: either $e \in D(v')$ for some other configuration $v' \in T$ and was added on one of lines 15, 21, or 29, or v is the target of some other edge e' that is being explored and was added on line 31.

We consider each case in turn.

- If e is added at line 15, 21, or 29 then it is dependent on some $v' \in T$. By Lemma 1, this can happen at most once for each $v \in V$ and hence e can be added to W this way at most $|T|$ times.
- The edge e is added at line 31, i.e. there is a previously selected edge $e' = (v', T')$ such that $v \in T'$. Therefore, $A(v)$ was $\bot$ before this iteration of the algorithm, and is ? afterwards, so in order for e' to add e again in this way, line 10 must be reached while $A(v) = ?$. However, since $e' \in D(v)$, this can only happen if $A(v') \in \{0, 1\}$, in which case it cannot reach line 31. Hence e is added to W at line 31 at most once for each $e' = (v', T')$ such that $v \in T'$, or a total of $|E|$ times.

In conclusion, e is added to W at most $\mathcal{O}(|V|+|E|)$ times, and thus also selected from W $\mathcal{O}(|V| + |E|)$ times. □

As the **while**-loop contains no loops or recursion and DGs are finite, we get the following corollary.

Corollary 1. *Algorithm 1 terminates.*

We now state three important invariants of the main **while**-loop of the algorithm.

Lemma 3 (Loop invariant). *The following is a loop invariant for the **while**-loop of Algorithm 1.*

1. *For all $v \in V$, if $A(v) \in \{0, 1\}$ then $A(v) = A_{\min}(v)$.*
2. *For all $v \in V$ and $e = (u, T) \in E$, $e \in D(v)$ implies that $v \in T$.*
3. *For all $v \in V$, if $A(v) = ?$ then for all edges $e = (v, T) \in E$, either $e \in W$ or there exists $u \in V$ such that $e \in D(u)$ and $A(u) = ?$.*

Proof. We prove these invariants in turn.

1. Let $e = (v, T)$ be the edge picked on line 8. We focus on the places where values are assigned. If $A(v)$ is assigned on Line 15, then we immediately know that for all $u \in T$, $A(u) = 1$ so the algorithm sets $A(v) = 1$. By the invariant, we also know for all $u \in T$ that $A_{\min}(u) = 1$, so $A_{\min}(v) = 1$. If $A(v)$ is assigned on either of lines 21 or 29, then there are no successors to v that could backpropagate value 1, so $A_{\min}(v) = 0$ hence the invariant holds.
2. The only place where $D(u)$ is updated for any $u \in V$ is on line 24. On this line, $D(u)$ is updated to include edge $e = (v, T)$, but since u was selected on line 23, $u \in T$ so the invariant is maintained.
3. Let $v \in V$ be a configuration such that $A(v) = ?$, and assume that the edge selected on line 8 is $e = (v, T)$. If the algorithm assigns $A(v) = x$ where $x \in \{0, 1\}$ in this iteration then the invariant clearly holds, so we are left with the following cases.
   - If the condition on line 9 is true, then $A(v)$ is set to $\bot$, so the invariant is preserved wrt. v, and furthermore for each $e' = (u, T') \in D(v)$ we have $A(u) \in \{0, 1\}$, so the invariant also holds for each such u.

– Otherwise on line 23 we pick a configuration $u \in T$ such that $A(u) \notin \{0,1\}$ and add e to $D(u)$. If $A(v) = ?$ at this point, then either $\mathrm{sucs}(u) = \emptyset$ so $A(u)$ is assigned 0 and e is added to W, in which case the invariant holds, or $A(u)$ is set to ? and each $e' \in \mathrm{sucs}(u)$ are added to the waiting list, so the invariant holds.

□

With the help of the previous lemmas we can now prove the correctness of Algorithm 1.

Theorem 1 (Correctness). *Algorithm 1 terminates, with return value* 1 *iff* $A_{\min}(v_0) = 1$.

Proof. Termination is provided by Corollary 1, and clearly if $A(v_0)$ was assigned $x \in \{0,1\}$ while running the algorithm, then $A(v) = A_{\min}(v_0)$ by Condition 1 of Lemma 3.

We now argue that if the algorithm terminates on line 32, then $A_{\min}(v_0) = 0$. First we note that at this point $A(v_0) = ?$, as $A(v_0)$ is initialised to ? before the **while**-loop, and if $A(v_0)$ was assigned value $x \in \{0,1\}$ then we would have terminated early. To demonstrate that $A_{\min}(v_0) = 0$ we construct a assignment B defined as

$$B(v) = \begin{cases} 0 & \text{if } A(v) \in \{0, ?\} \\ 1 & \text{if otherwise.} \end{cases}$$

In words, this assignment promotes ? to 0. We wish to prove that B is a fixed-point assignment in a subgraph of G that contains v_0. The region of interest is given by a set Q, which we define as the least set such that

– $v_0 \in Q$, and
– if $v \in Q$ then for all $u \in V$ such that there exists an edge $e = (v, T) \in D(u)$ and $A(u) = ?$ we have $u \in Q$.

In words, Q is the set of configurations reachable from v_0 via only configurations with value ?. We show that B is a fixed-point assignment over all $v \in Q$, i.e. for all $v \in V$ we have

$$B(v) = \bigvee_{(v,T) \in E} \bigwedge_{u \in T} B(u) \ .$$

For the sake of contradiction, assume that there is some configuration $v \in Q$ such that $B(v) = 0$ but there exists a hyperedge $e = (v, T)$ such that $B(u) = 1$ for all $u \in T$. Due to Condition 1 of Lemma 3, it cannot be the case that $B(v) = 0$ because $A(v) = 0$, so $A(v) = ?$. Thus by Condition 3 of Lemma 3 we have either that $e \in W$ or there exists $u \in V$ such that $e \in D(u)$ and $A(u) = ?$, but since the algorithm terminated outside the **while**-loop, $W = \emptyset$, hence $e \in D(u)$ for some u where $A(u) = ?$. By Condition 2 we also have $u \in T$. But we have $B(u) = 0$ and $u \in T$, which contradicts our assumption that for all $u \in T$ we have $B(u) = 1$.

Because of this, B is a fixed-point assignment on the subgraph induced by $Q \cup \{u \in V \mid v \to u \text{ for some } v \in Q\}$. Since $A_{\min}$ is the least fixed-point assignment and $B(v_0) = 0$, $A_{\min}(v_0) = 0$, so we are done. □

5 Implementation and Experiments

We implement Algorithm 1 as an extension to the verification engine `verifypn` of the tool TAPAAL [11], a Petri net model checker written in C++. In doing so, the nondeterminism at lines 8 and 23 needs to be resolved.

DFS vs. BFS In order to resolve the nondeterminism at line 8, we need to select an $e \in W$ to process, or in other words, we should select a suitable data structure for representing the set W. We implement W as two sets, W_f containing edges inserted during the forward exploration, and W_b containing edges inserted due to the backpropagation. Edges are added to W_f on lines 1 and 31, and to W_b on lines 15, 21, and 29. When selecting an edge on line 8, we select first from W_b if possible, and from W_f otherwise. During our experiments, we experience that the underlying structure of W_b makes little difference, so we keep it as a stack. For W_f we evaluate two choices:
- W_f is a stack, which we refer to as DFS.
- W_f is a queue, which we refer to as BFS.

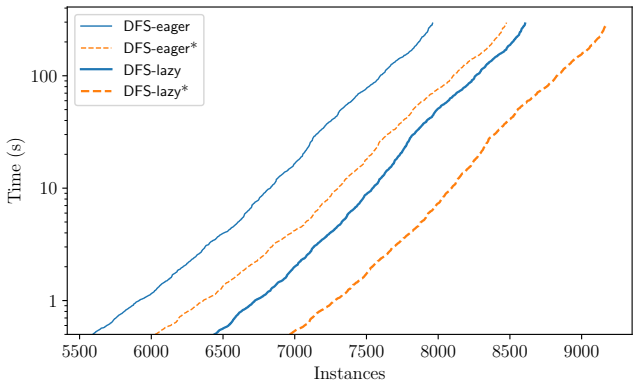
Eager vs. Lazy At line 23, we need to select an $u \in T$ that is not fully assigned. We evaluate the following two options for partially resolving this nondeterminism.
- We can prioritise configurations u such that $A(u) = ?$, i.e. configurations that were visited before but not fully assigned. We refer to this choice as lazy.
- We can prioritise configurations u such that $A(u) = \bot$, i.e. configurations that were not yet explored. We refer to this choice as eager.

In either case, if there is more than one configuration with the prioritised value, we pick an arbitrary one.

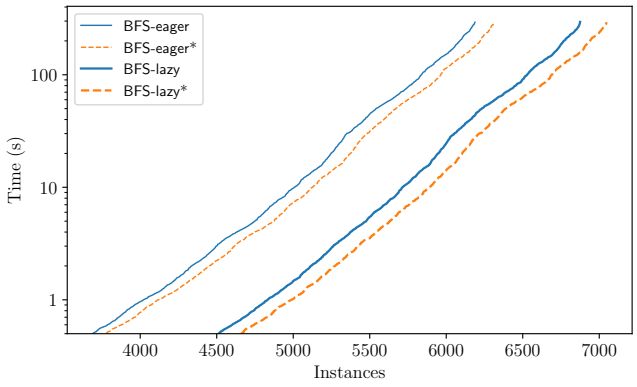
We shall now experimentally evaluate the new algorithm and the different implementation choices described above. We evaluate it against the CTL benchmark of the Model Checking Contest (MCC) 2021 [20] data set, which consists of 1181 Petri nets which are each associated with 16 CTL formulae for a total of 18 896 problem instances, as well as against benchmarks of Petri net games detailed in [4] and [12]. We name the different versions by using the naming introduced above. If our improvement starting at line 9 is enabled, we append an asterisk (*) to the configuration name, otherwise the improvement is not enabled. For example, DFS-lazy* denotes that W_f is a stack, configurations with $A(v) = ?$ are prioritised and it is our improved version of the algorithm. We do not compare against other tools, since TAPAAL significantly outperformed other competitors of the MCC'21 [20] and MCC'22 [21] even without the present improvement. A reproducibility package containing the data, raw results, binaries, and scripts used for running and analysing the experiments can be found at [17].

5.1 CTL Benchmark

For the CTL evaluation, we run the engine on each formula in the benchmark, with a time limit of 5 minutes and memory limit of 15 GiB per formula, using



(a) Performance comparison for DFS



(b) Performance comparison for BFS

Fig. 4: Experiments for CTL model checking

AMD Opteron 6376 Processors. Figures 4a and 4b show cactus plots, where for each experimental configuration, the runtime (on y-axis) for each instance is sorted in ascending order independent of the other configurations, and the instances are plotted in this order on the x-axis (we only show the most difficult instances here). Notice that the running times are plotted using a logarithmic scale. All running times refer only to time used in the verification algorithm. We observe that when using DFS, our new algorithm (shown in the plots as dashed lines) improves the average performance by about a factor 5 when using the lazy setup, and a factor 3 when using the eager setup. A more modest improvement can be seen also when using the BFS strategy and also here the eager strategy performs better. Overall, the DFS solves the largest number of CTL queries within the 5 minute timeout.

Table 1 shows the number of unique answers between each pair of experimental configurations, such that each row entry indicates the number of exclusive

Table 1: Exclusive answers, all CTL comparisons. Each row entry is the number of unique answers of the row configuration relative to the corresponding column configuration.

	DFS-eager	DFS-eager*	DFS-lazy	DFS-lazy*	BFS-eager	BFS-eager*	BFS-lazy	BFS-lazy*
DFS-eager	0	1	5	0	2102	2026	1872	1788
DFS-eager*	516	0	460	1	2570	2460	2307	2183
DFS-lazy	650	590	0	0	2737	2656	2084	1995
DFS-lazy*	**1212**	**698**	**567**	**0**	3259	3148	2596	2441
BFS-eager	329	282	319	274	0	13	25	25
BFS-eager*	380	299	365	290	140	0	133	29
BFS-lazy	786	706	353	298	712	693	0	8
BFS-lazy*	874	754	436	315	**884**	**761**	**180**	**0**

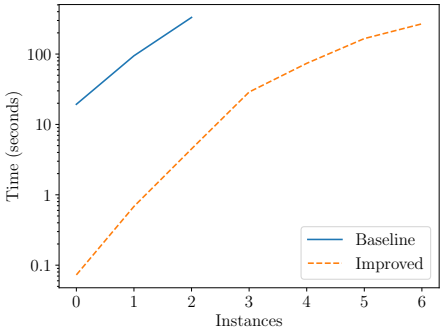
answers compared to the configuration in the corresponding column. For example, DFS-lazy gains 590 answers relative to DFS-eager* but loses 460 answers. In the top left quadrant, we observe that DFS-lazy* is clearly superior to all other three DFS strategies. In the lower right quadrant, we see a similar trend where BFS-lazy* is also the best configuration based on the BFS search strategy. Comparing the two remaining quadrants, we observe that using DFS gains between 2000–3000 unique answers over using BFS. Among these are 77 answers that were obtained only by DFS-lazy* and no other configuration. However, BFS also gains 300–800 unique answers compared to DFS, a considerable number. This indicates that both search strategies are useful and thus an ideal approach will run these strategies in parallel.

Remark 3. The full check of detached regions described in Remark 2 has 16 exclusive answers compared to DFS-lazy*, but loses 2923 answers due to the excessive overhead of rigorously keeping dependencies fully synchronised.

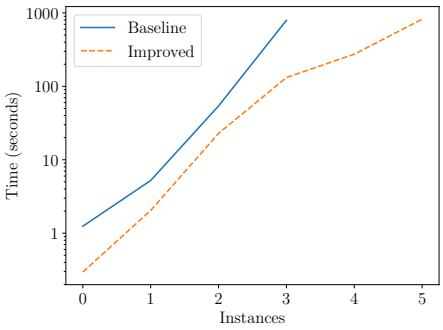
5.2 Games Synthesis Benchmark

We further implemented our improved algorithm in the Petri net games verification engine, which is also a part of the `verifypn` engine. Based on the experience from CTL experiments, we only implemented the best performing configuration using the DFS search order. We evaluate this algorithm on the game synthesis benchmarks presented in [4] and [12].

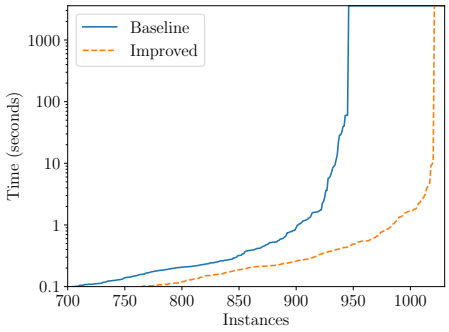
The benchmark of [4] includes 6 scalable case studies. Each model in the case study is given 1 hour and 32 GiB memory. Out of the 6 case studies, on 4 of them we observed minimal difference between the original and improved algorithm, while on the Producer/Consumer systems and on the model of the



(a) Producer/consumer experiment



(b) Lyngby station experiment



(c) Update synthesis experiments on topology zoo benchmark

Fig. 5: Petri games experiments

Lyngby Train Station we noticed significant improvements up to two orders of magnitude, as shown in Figure 5a and Figure 5b.

The topology zoo benchmark of [12] (originally described in [19]) consists of 261 real network topologies of up to 700 nodes as well as nestings and concatenations of these, for a total of 1035 problem instances. Our evaluation is based on the problem of synthesis of network updates, encoded as Petri net games. Figure 5c shows a cactus plot with the results. Our improved algorithm gains about 80 answers over the certain-zero baseline algorithm and performs better by over an order of magnitude on harder problem instances. This demonstrates that our detached regions elimination technique is applicable to a range for dependency graphs coming from both the model checking domain as well as from strategy synthesis for games.

6 Conclusion

We presented a novel improvement to the local Liu and Smolka's algorithm used for verification of dependency graphs. Our algorithm detects detached regions

in the dependency graph in order to speedup the performance of the fixed-point computation. We proved that our improved algorithm is correct and provided its efficient implementation, as part of the tool TAPAAL, both for CTL model checking and Petri games synthesis. We evaluated the performance of our algorithm on benchmarks of Petri games and CTL formulae on Petri nets, demonstrating noticeable improvements in verification speed (compared to the state-of-the-art approaches, including the most recent certain-zero algorithm) with only negligible overhead. We also observed that the depth-first search strategy is the most beneficial one, however, breadth-first search should be also considered as it can provide a large number of unique answers.

In the future work, we can consider extending the algorithm to the more general abstract dependency graphs and further improvements to the fixed-point algorithm, including e.g. the detection of loops in dependency graphs. The exact asymptotic running time of the algorithm is left as an open question.

Acknowledgements

We would like to thank the anonymous reviewers for their feedback and suggestions. This work was supported by the S40S Villum Investigator Grant (37819) from VILLUM FONDEN.

References

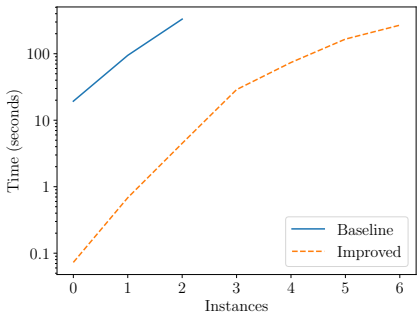
1. Alur, R., Dill, D.: Automata for modeling real-time systems. In: Paterson, M.S. (ed.) Automata, Languages and Programming. pp. 322–335. Springer Berlin Heidelberg, Berlin, Heidelberg (1990)
2. Andersen, J.R., Andersen, N., Enevoldsen, S., Hansen, M.M., Larsen, K.G., Olesen, S.R., Srba, J., Wortmann, J.K.: Caal: Concurrency workbench, aalborg edition. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) Theoretical Aspects of Computing - ICTAC 2015. pp. 573–582. Springer International Publishing, Cham (2015)
3. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-tiga: Time for playing games! In: Computer Aided Verification, pp. 121–125. Springer Berlin Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_14, https://doi.org/10.1007/978-3-540-73368-3_14
4. Bønneland, F., Jensen, P., Larsen, K., Muniz, M., Srba, J.: Partial order reduction for reachability games. In: Proceedings of the 30th International Conference on Concurrency Theory (CONCUR'19). LIPICS, vol. 140, pp. 23:1–23:15. Dagstuhl Publishing (2019). https://doi.org/10.4230/LIPIcs.CONCUR.2019.23, please note that the paper contains an error that is fixed here: https://arxiv.org/abs/1912.09875
5. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.C.: The mCRL2 toolset for analysing concurrent systems. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 21–39. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-17465-1_2, https://doi.org/10.1007/978-3-030-17465-1_2

6. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Lecture Notes in Computer Science, pp. 66–80. Springer Berlin Heidelberg (2005). https://doi.org/10.1007/11539452_9, https://doi.org/10.1007/11539452{_}9
7. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Workshop on Logic of Programs. pp. 52–71. Springer (1981)
8. Dalsgaard, A., Enevoldsen, S., Larsen, K., Srba, J.: Distributed computation of fixed points on dependency graphs. In: Proceedings of Symposium on Dependable Software Engineering: Theories, Tools and Applications (SETTA'16). LNCS, vol. 9984, pp. 197–212. Springer (2016). https://doi.org/10.1007/978-3-319-47677-3_13
9. Dalsgaard, A.E., Enevoldsen, S., Fogh, P., Jensen, L.S., Jepsen, T.S., Kaufmann, I., Larsen, K.G., Nielsen, S.M., Olesen, M.C., Pastva, S., Srba, J.: Extended dependency graphs and efficient distributed fixed-point computation. In: van der Aalst, W., Best, E. (eds.) Application and Theory of Petri Nets and Concurrency. pp. 139–158. Springer International Publishing, Cham (2017)
10. Dalsgaard, A.E., Enevoldsen, S., Fogh, P., Jensen, L.S., Jensen, P.G., Jepsen, T.S., Kaufmann, I., Larsen, K.G., Nielsen, S.M., Olesen, M.C., et al.: A distributed fixed-point algorithm for extended dependency graphs*. Fundamenta Informaticae **161**(4), 351–381 (Jul 2018). https://doi.org/10.3233/FI-2018-1707, https://doi.org/10.3233/FI-2018-1707
11. David, A., Jacobsen, L., Jacobsen, M., Jørgensen, K., Møller, M., Srba, J.: TAPAAL 2.0: integrated development environment for timed-arc Petri nets. In: Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12). LNCS, vol. 7214, p. 492–497. Springer-Verlag (2012)
12. Didriksen, M., Jensen, P., Jønler, J., Katona, A.I., Lama, S., Lottrup, F., Shajarat, S., Srba, J.: Automatic synthesis of transiently correct network updates via Petri games. In: Proceedings of the 42nd International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets'21). LNCS, vol. 12734, pp. 118–137. Springer-Verlag (2021). https://doi.org/10.1007/978-3-030-76983-3_7
13. Enevoldsen, S., Larsen, K., Mariegaard, A., Srba, J.: Dependency graphs with applications to verification. International Journal on Software Tools for Technology Transfer (STTT) **22**, 635–654 (2020). https://doi.org/10.1007/s10009-020-00578-9
14. Enevoldsen, S., Larsen, K., Srba, J.: Abstract dependency graphs and their application to model checking. In: Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19). LNCS, vol. 11427, pp. 316–333. Springer-Verlag (2019). https://doi.org/10.1007/978-3-030-17462-0_18
15. Enevoldsen, S., Larsen, K., Srba, J.: Extended abstract dependency graphs. International Journal on Software Tools for Technology Transfer (STTT) **24**, 49–65 (2022). https://doi.org/10.1007/s10009-021-00638-8
16. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: Cadp 2011: a toolbox for the construction and analysis of distributed processes. International Journal on Software Tools for Technology Transfer **15**(2), 89–107 (2013)
17. Gjøl Jensen, P., Larsen, K.G., Srba, J., Jensen Ulrik, N.: Reproducibility package: Elimination of detached regions in dependency graph verification (Mar 2023). https://doi.org/10.5281/zenodo.7712764

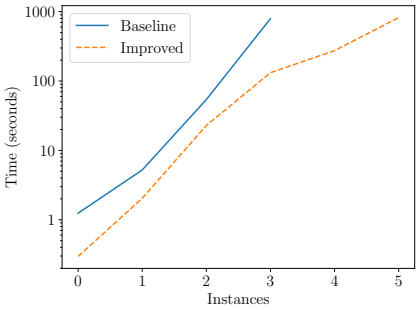
18. Jensen, J.F., Larsen, K.G., Srba, J., Oestergaard, L.K.: Efficient model-checking of weighted CTL with upper-bound constraints. International Journal on Software Tools for Technology Transfer **18**(4), 409–426 (Nov 2016). https://doi.org/10.1007/s10009-014-0359-5, https://doi.org/10.1007/s10009-014-0359-5
19. Knight, S., Nguyen, H.X., Falkner, N., Bowden, R., Roughan, M.: The internet topology zoo. IEEE Journal on Selected Areas in Communications **29**(9), 1765–1775 (2011). https://doi.org/10.1109/JSAC.2011.111002
20. Kordon, F., Bouvier, P., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Amat., N., Amparore, E., Berthomieu, B., Biswal, S., Donatelli, D., Galla, F., , Dal Zilio, S., Jensen, P., He, C., Le Botlan, D., Li, S., , Srba, J., Thierry-Mieg, ., Walner, A., Wolf, K.: Complete Results for the 2020 Edition of the Model Checking Contest. http://mcc.lip6.fr/2021/results.php (June 2021)
21. Kordon, F., Bouvier, P., Garavel, H., Hulin-Hubard, F., Amat., N., Amparore, E., Berthomieu, B., Donatelli, D., Dal Zilio, S., Jensen, P., Jezequel, L., He, C., Li, S., Paviot-Adet, E., Srba, J., Thierry-Mieg, Y.: Complete Results for the 2022 Edition of the Model Checking Contest. http://mcc.lip6.fr/2022/results.php (June 2022)
22. Kozen, D.: Results on the propositional μ-calculus. In: Automata, Languages and Programming, pp. 348–359. Springer Berlin Heidelberg (1982). https://doi.org/10.1007/bfb0012782, https://doi.org/10.1007/bfb0012782
23. Liu, X., Ramakrishnan, C., Smolka, S.A.: Fully local and efficient evaluation of alternating fixed points. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 5–19. Springer (1998)
24. Liu, X., Smolka, S.A.: Simple linear-time algorithms for minimal fixed points. In: Automata, Languages and Programming, pp. 53–66. Springer Berlin Heidelberg (1998). https://doi.org/10.1007/bfb0055040, https://doi.org/10.1007/bfb0055040
25. Mariegaard, A., Larsen, K.G.: Symbolic dependency graphs for PCTL$^{\geq}_{\leq}$ model-checking. In: Abate, A., Geeraerts, G. (eds.) Formal Modeling and Analysis of Timed Systems. pp. 153–169. Springer International Publishing, Cham (2017)
26. Milner, R.: Communication and concurrency. Prentice Hall International series in Computer Science, Prentice Hall (1989)
27. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific journal of Mathematics **5**(2), 285–309 (1955)

Appendix

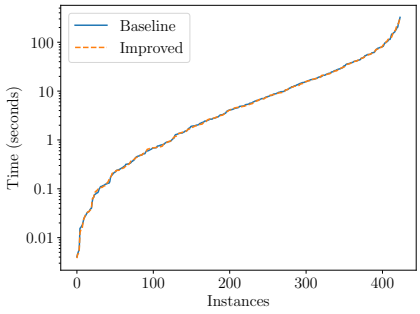
A Games results



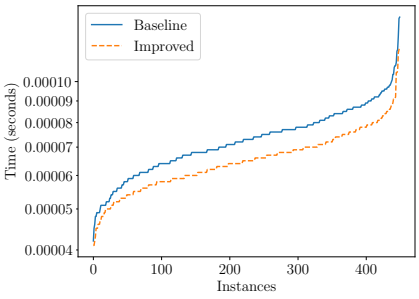
(a) Producer/Consumer System



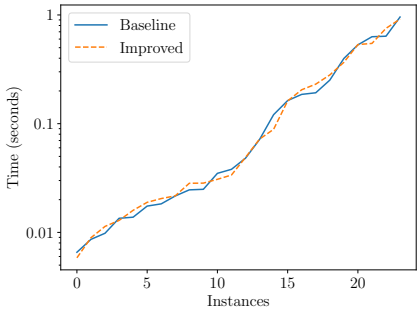
(b) Lyngby Train Station



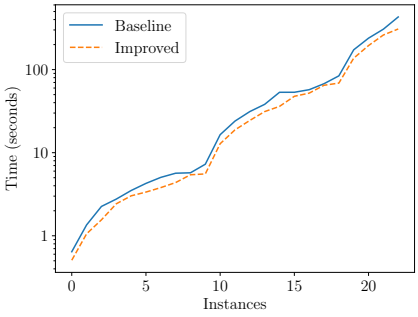
(c) Autonomous Intersection Management



(d) Nim Game



(e) Manufacturing Workflow



(f) Order Workflow

Fig. 6: Cactus plots for games experiments. Benchmarks detailed in [4].