FI MU

Faculty of Informatics
Masaryk University Brno

Dynamic maintenance of an accepting run

by

Florent Peres
Ivana Černá

FI MU Report Series **FIMU-RS-2013-01**

 August 2013

Publications in the FI MU Report Series are in general accessible via WWW:

`http://www.fi.muni.cz/reports/`

Further information can be obtained by contacting:

**Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic**

Dynamic maintenance of an accepting run

Florent Peres
Masaryk University
florent.peres@gmail.com

Ivana Černá
Masaryk University
cerna@fi.muni.cz

August 16, 2013

Abstract

Finding an infinite sequence of transitions (a run) going through some special states (called accepting) is of importance for fields like formal verification. Whereas finding such a sequence as been extensively studied, the problem of *maintaining* this sequence upon changes in the model or the specification has received less attention. In this work, we propose a solution to the maintenance of an accepting run when the transition system representing the product of the model and of the specification is changed, using the Tarjan's algorithm as a base algorithm.

1 Introduction

A model is susceptible to change. The reason may vary: it could have been proven incorrect with regard to an expected specification, or a new feature needed to be included, or an existing one had to be modified, etc. One particularly interesting question is whether some previous computation can be reused on its new, updated version.

Consider the model checking technique which is used to verify that a model is correct to a given specification. Traditionnally, as soon as the model change, checking that the updated version posseses the expected property is done by launching the whole verification process once again. In many cases this re-computation is very costly and may greatly benefits from not recomputing everything.

Likewise, a model could be the representation of the multiples trajectories of a robot. Whenever a path has been planned (according to many good properties: shortest, going from a to b, etc.), modifying the model may obviously break the previously computed

path. Considering the previous computation to recompute a new path could be a beneficial move.

The latter example can be seen a model checking instance, which is why we will focus on model checking without (hopefully too much) loss of generality.

Checking the validity of an LTL formula $\mathtt{f}$ on a transition system $\mathtt{t}$ can be accomplished by checking the emptiness of the intersection of $\mathtt{t}$'s traces language with the language recognized by the non-deterministic Bücchi Automaton (NBA) of the formula $\neg\mathtt{f}$ ($\mathcal{B}(\neg\mathtt{f})$). Checking the emptiness of the intersection language can be done by checking whether the product of $\mathtt{t}$ by $\mathcal{B}(\neg\mathtt{f})$ has no accepting cycle. Checking that the product of $\mathtt{t}$ by $\mathcal{B}(\neg\mathtt{f})$ has no accepting cycle means that the original formula $\mathtt{f}$ is valid. Would an accepting cycle be found, it would "prove" that $\neg\mathtt{f}$ holds (i.e. $\mathtt{f}$ does not hold). The infinite trace starting from the initial state of $\mathtt{t}$ and comprising this accepting cycle is called an *accepting run* or a *counter-example* of $\mathtt{f}$.

Instead of using this traditional model-checking approach, we are more interested in finding (and maintaining) a *witness* of the property $\mathtt{f}$. This is equivalent to wanting to maintain a *counter-example* of the property $\neg\mathtt{f}$, but we prefer doing it directly by checking whether the language recognized by the product $\mathtt{t} \times \mathcal{B}(\mathtt{f})$ is empty.

Therefore, in this work we focus on the product and forget that it was obtained from a transition system and the NBA of a formula $\mathtt{f}$.

This article proposes a solution to tackle the issue of maintaining an accepting run after a modification of the product from which the run was extracted. Possible modifications include removing or adding any transition on the system.

Notice that considering any modification of the product is equivalent to consider any modification of the original model $\mathtt{t}$ *as well as* any modification of the original property $\mathtt{f}$.

1.1 Related works

An early model-checking work [1] proposes to use a model-checking technique for checking (alternation free) modal μ-calculus and adapt it to handle changes in the product. It is possible to specify the existence of an accepting run using the alternation free μ-calculus. However, modal μ-calculus being a powerful logic, the product needed to check it is bigger than what one would expect to simply check the existence of an accepting run. Finally, only the maintenance of the product is of interest in this work and no

particular attention has been given to get a witness or a counter-example, and *a fortiori* to maintain it.

Dynamic graph problems are closely related to what we try to achieve in this work. The main goal being to avoid recomputations when the graph is modified. The closest works we are aware of consider the maintenace of the reachability relation of a graph [3].

The problem usually comes with a trade-off between the *query* time and the *update* time. A query is a request of the considered problem, i.e. in [3], a query is to answer whether one vertex is reachable from one another. An update is a modification of the graph. In [3] the achieved query time is $O(n)$ in the worst case (n being the number of vertices) and the achieved amortized update time is $O(m + n \log n)$ (m is the number of edges).

Notice that the problem is at the same time more general that ours: the reachability query can be for any vertex to any vertex and also more restrictive as only finite path are considered.

Other related works are considering the maintenance of all-pairs shortest path and transitive closure ([4], [5]). Finding accepting runs with a transitive closure is easy. Unfortunately the trade-off is that computing this transitive closure takes much more time that a simple (nested) depth first search traversal performed by a model-checker.

Closer to our problem, a parity condition strategy can be updated when a Markov decision process is updated [9]. This work actually solves a more general problem, however reducing it to our framework may not lead to good results. The approach is strongly based on an offline method: the end-components have to be completely discovered, even though an accepting cycle could be detected at the beginning (our approach uses an on-the-fly algorithm). Also, to retrieve an accepting run, at least one depth first search have to be performed in the end components involved in the accepting cycle: a path must be extracted to reach each end-component (our approaches uses an on-the-fly variant of the Tarjan's algorithm that can output the accepting run when detected).

Another related work is [8], in which the authors use a SAT-based model-checker to incrementally construct an inductive proof that a property holds. When the specification or even the property is changed, that proof can be used to check which part of the system need to be updated. In this work, the authors claim that they use safety properies, which is too weak to specify an accepting run.

In [6], the authors propose an approach to model a program using control flow graphs, give a safety property using an automaton and maintain the resulting product whenever a change in the control flow graph occurs. The limitation to safety properties prevent to specify accepting cycles.

In [10], the authors study propose an incremental model-checking technique for LTL used in the "Software Product-line" engineering methodology. This methodology heavily relies on reusing *assets*, which are bits of code, requirements, test cases, etc. Assets are characterized by their *feature*, corresponding to what they can be used for. The author then restrict the incremental model-checking to *additions* of features.

2 Accepting run maintenance

In this section, the notion of accepting run is defined, along with the maintenance problem tackled in this article.

Definition 2.1. *Let $\mathcal{Q}$ be the set of all possible states.*

Definition 2.2. *An Unlabelled Büchi Automaton (UBA) is a tuple $\langle Q, \delta, s_0, A\rangle$ in which:*

- $Q \subseteq \mathcal{Q}$ *is a set of* states,
- $\delta \subseteq Q \times Q$ *is the transition relation,*
- $s_0 \in Q$ *is the* initial state,
- $A \subseteq Q$ *is the set of* accepting states.

A transition $(s,t) \in \delta$, will be written $s \xrightarrow{\delta} t$. The transitive closure of a relation R will be denoted R^+.

Definition 2.3. *A* path p *of a relation δ is a transition sequence $p_1 \dots p_{n+1}$ such that $p \in \delta^* \wedge (\forall i \in [1,n])(p_i = (s,t) \wedge p_{i+1} = (t,u))$, where δ^* is the Kleene closure of δ.*

Definition 2.4. $s \overset{\delta}{\Rightarrow} t = \{p = p_1 \dots p_n \mid p$ *is a path and* $p_1 = (s,x) \wedge p_n = (y,t)\}$. $s \overset{\delta}{\Rightarrow} t$ *is the set of paths in δ whose first transition source state is s and last transition target state is t.*

Definition 2.5. *The set of paths in δ is* $\textsc{Paths}(\delta) = \{s_0 \overset{\delta}{\Rightarrow} t | (s_0,t) \in \delta^+\}$.

Definition 2.6. *A path $p = p_1 \dots p_n$ is a* cycle *iff $p_1 = (s,x)$ and $p_n = (y,s)$.*

Definition 2.7. *A path* $p = p_1 \dots p_n$ *is a* run *iff* $(\exists i)(p_i = (s, x)$ *and* $p_n = (y, s))$. $p_1 \dots p_{i-1}$ *is called the* initial *part of the run and* $p_i \dots p_n$ *the* cyclic *part of the run.*

Definition 2.8 (Accepting run)**.** *Let* r *be a run* $r_i \dots r_n$ *be its cyclic part.* r *is* accepting *iff* $(\exists j \in [i, n])(r_j = (s, t) \wedge s \in A)$.

Definition 2.9. *A* patch *is a couple* $\langle U, T \rangle$*, in which:*

- $U \subseteq Q \times Q$ *is a set of updates.*
- $T \in \{\mathtt{Add}, \mathtt{Rem}\}$ *is the type of the updates.*

A patch whose type is $\mathtt{Add}$ is called an *additive patch*. Likewise, a patch whose type is $\mathtt{Rem}$ is called a *destructive patch*.

Definition 2.10. *Let* $b = \langle Q, \delta, s_0, A \rangle$ *be an UBA and let* $p = \langle U, T \rangle$ *be a patch. Applying the patch* p *on* B *gives the UBA* $c = \langle Q', \delta', s_0, A \rangle$*, where:*

- *If* $T = \mathtt{Add}$ *then* $\delta' = \delta \cup U$ *and* $Q' = Q \cup \{s, t \mid (s, t) \in U\}$
- *If* $T = \mathtt{Rem}$ *then* $\delta' = \delta \setminus U$ *and* $Q' = Q \setminus \{t \mid (s, t) \in U \wedge (\forall (u, v) \in \delta')(v \neq t)\}$

The patch application is written $C = [B]^T_U$ *or* $C = [B]_p$ *depending on whether we want to use the internal information of* p.

In this work, we propose a new solution for the following problem. Given an UBA $B = \langle Q, \delta, s_0, \mathcal{A} \rangle$ and a sequence of patches $P = p_1 \dots . p_n$, we want to know whether there exists an accepting run for each updated version of B (i.e. for each $\left[[B]_{p_1} \dots\right]_{p_i}$ $(\forall i \in [1, n])$), and output it if it does.

3 Toward a solution

Let us suppose that the search for an accepting run is accomplished by the algorithm α (no knowledge of α is required yet). For an UBA b, the result of $\alpha(b)$ is an accepting run (or ϵ if none was found).

All the solutions we propose in this work are based on the algorithm λ of Fig. 1 defining how patches are applied.

A different solution can be found by setting a different definition for 1) r is invalidated by x, 2) Restart the computation and 3) Manage the patch p.

Initially, given an UBA B,

$\lambda(B) = \alpha(B)$

Afterwards,

given the previous accepting run $r = \lambda(b)$,

and the patch $p_i = (u, t)$

$\lambda([[b]_{p_1} \dots]_{p_i}) =$

if $\exists x \in u$: r is invalidated by x

then Restart the computation

else Manage the patch p

Figure 1: Patch application pattern

Defining those three sentences defines how a patch is applied. The main motivation being that a patch should be applied *lazily*, i.e. the application of a patch should be avoided when it is not strictly required. Also, while we want a patch to be lazily applied, we also want the (generic) property 1 to hold: applying each patch using λ gives the same result as applying the patch to the UBA and re-computing from scratch for every patch.

Property 1. *Given a patch sequence* $p_1 \dots p_n$ *and an UBA* B*:*

$$(\forall i \in [1, n]) \left(\lambda\left(\left[[B]_{p_1} \dots\right]_{p_i}\right) = \alpha\left(\left[[B]_{p_1} \dots\right]_{p_i}\right)\right)$$

Depending on the definition of the three previous items, the laziness degree may heavily vary.

Definition 3.1. *A non-lazy patch application* λ^0 *can be defined as follows:*

- r *is invalidated by* u*: always true*
- *Restart the computation: launch* $\alpha([[b]_{p_1} \dots]_{p_i})$*, i.e. launch* α *on the UBA* b *updated with the patches* $p_1 \dots p_i$*.*
- *Manage the patch* p*: do nothing*

The λ^0 patch application has the lowest laziness degree possible, since it is not lazy at all. In fact, λ^0 is exactly equivalent to the right hand side of Prop. 1. For every patch, the

UBA is updated and the accepting run is computed from scratch. Using this solution, if the update is discovered late during the computation, then a lot of work have been potentially wasted: up to the update discovery point, the computation have been the same as for the previous iteration. The complexity of this solution is $O(n \times (|V| + |E|))$, where n is the number of patches.

An alternative could be to consider that a recomputation is only needed whenever 1) the run is broken by removing a transition or 2) if a new transition is introduced while no accepting run was previously found.

Definition 3.2. *Let* $p = (u, \mathrm{Rem})$ *and* $r = r_1 \dots r_n \in \textsc{Paths}(\delta)$*, the updates* u *in the destructive patch* p *break the run* r *(written* $u \twoheadrightarrow r$*) when* $(\exists i \in [1, n])(r_i \in u)$.

Going just one step away from λ^0 and using the previous idea, we obtain the first truly lazy patch application λ^1.

Definition 3.3. *The lazy patch application* λ^1 *is defined as follows:*

- r *is invalidated by the patch* $p = (u, t)$ *iff*

$$t = \mathrm{Rem} \wedge r \neq \epsilon \wedge u \twoheadrightarrow r$$

$$\textit{or}\ \ t = \mathrm{Add} \wedge r = \epsilon$$

- *Restart the computation: launch* $\alpha([[b]_{p_1} \dots]_{p_i})$
- *Manage the patch* p*: do nothing*

Property 2. *Property 1 is true if* $\lambda = \lambda^1$.

Proof. By induction on n:

Base Case: Initially no patch is applied to B, then $\lambda(B) = \alpha(B)$.

Induction Case: Suppose that Prop. 2 is true up to patch p_{i-1}, we need to prove that Prop. 2 is still correct after applying p_i. The computation is restarted from scratch, for the updated version of the UBA, every time the run is invalidated. We are then sure to get a correct answer iff the invalidation conditions are correct.

The conditions to invalidate a run are the following:

- $t = \mathrm{Rem}$, the patch is destructive, $r \neq \epsilon$, there is an actual accepting run r in b and $u \twoheadrightarrow r$ there is an update breaking the run. This means one of the transitions removed was used by r. Then r cannot be valid anymore.

- $t = \mathrm{Add}$, the patch is additive, $r = \epsilon$ and there is no accepting run in b. Adding a transition can create an accepting cycle, and to be safe, we require a new computation.

Otherwise:

- $t = \mathrm{Rem}$, the patch is destructive,
  - $r \neq \epsilon$, there is an actual accepting run r in b and $\neg u \twoheadrightarrow r$, none of the updates u are transitions involved in the run. This means the run is still there and valid.
  - $r = \epsilon$, there is no accepting run in b. Removing transitions can only destroy cycles. If none of them was accepting, then there cannot be new cycles after removing some transitions.
- $t = \mathrm{Add}$, the patch is additive, $r \neq \epsilon$ and there is an actual accepting run r in b. Adding transitions cannot break any cycle and a fortiori the patch cannot break the accepting run r.

□

Property 3. λ^1 *never performs more re-computations than* λ^0.

Proof. When r is invalidated (surely or potentially) by a patch, a *single* re-computation is performed. If there is m patches which are not invalidating r amongst a total number of n patches, then $(n - m) \leq n$ re-computations will be performed using λ^1, whereas n re-computations would have been required with the strict application of patches λ^0. □

This solution is still not really satisfactory. We can certainly save a lot of work by using this basic update management algorithm, but the ineffiency stated before, namely *"up to the update discovery point, the computation have been the same as for the previous iteration"*, is still there. In other words, the lazy patch application did not solve anything regarding the situation in which a single update *may* change the existence of the accepting run, i.e. the update *may* have an impact on the accepting run existence.

A computation $\alpha(b)$ can always be restarted from scratch, i.e. it can always be restarted from the initial state s_0. To show that the computation is restarted from s_0, α can be overloaded to $\alpha(b, s_0)$. We say that α is *backtrackable* to s_0.

Suppose that we already computed an accepting run for b. When patching b with p, we want to be able to identify a state i (ideally discovered the latest possible) for which

$\alpha'([b]_p, i) \Leftrightarrow \alpha([b]_p, s_0)$, where α' is a modified version of α capable of backtracking to i. In other words, we want to identify a state i such that, when applying the patch p to b, backtracking to i using α' yield the same result as restarting α from s_0.

The patch p contains an update u that caused α' to be backtracked to i. We say that i is the *impact state* of the update u, or the *oldest impact state* of the patch p.

In Section 4, we present three backtrackable algorithms that can get an accepting run in an UBA b, based on Tarjan's algorithm. And we will discuss the different heuristics we can use to manage the patches in Section 6.

4 Accepting run computation definitions

4.1 Depth first exploration

Definition 4.1. *An exploration* e *is the tuple* $\langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$*, where*

- $\langle Q_e, \delta_e, s_0, A \rangle$ *is an UBA.*
- $\lhd_e : \delta_e \times \delta_e$ *be an ordering on the transitions* δ_e*. The transitions with a common source state are totally ordered by* $\lhd_e$*, i.e.* $(\forall x \xrightarrow{\delta_e} y, x \xrightarrow{\delta_e} z)(y = z \vee x \xrightarrow{\delta_e} y \lhd_e x \xrightarrow{\delta_e} z \vee x \xrightarrow{\delta_e} z \lhd_e x \xrightarrow{\delta_e} y)$.

Definition 4.2. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be an exploration, the relation* $\prec_e$*:* $Q_e \times Q_e$ *is defined as follows:* $x \xrightarrow{\delta_e} y \wedge x \xrightarrow{\delta_e} z \wedge z \xRightarrow{\delta_e} y \neq \emptyset \implies x \xrightarrow{\delta_e} y \lhd_e x \xrightarrow{\delta_e} z$.

The triple $\langle Q_b, \prec_b, s_0 \rangle$ is a rooted tree called *depth first exploration tree*. When $s \xrightarrow{\prec_b^+} t$ (i.e. (s, t) belongs to the transitive closure of $\prec_b$), we say that s is a dfs-ancestor of t.

Definition 4.3. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be an exploration, the* depth first ordering $<_e$ *is defined as follows:* $x <_e y \iff$

$$\exists\, s_0 \xrightarrow{\prec_e^+} z \xrightarrow{\prec_e} a \xrightarrow{\prec_e^+} x \quad \wedge \quad \exists\, s_0 \xrightarrow{\prec_e^+} z \xrightarrow{\prec_e} b \xrightarrow{\prec_e^+} x \quad \wedge \quad z \xrightarrow{\prec_e} a \lhd_e z \xrightarrow{\prec_e} b.$$

The relation $<_e$ defines the depth first (total) ordering amongst the state Q_e according to the transition ordering $\lhd_e$. The expression $x <_e y \vee x = y$ is denoted $x \leq_e y$.

Definition 4.4. *Let* r *be a total ordering on some set* X*. The minimum and maximum w.r.t.* r *of a set* Y *are written respectively* $\mathrm{Min}^r(Y)$ *and* $\mathrm{Max}^r(Y)$*. If* $Y \cap X = \emptyset$ *then* $\mathrm{Min}^r(Y) = \mathrm{Max}^r(Y) = \bot$.

Definition 4.5. *The exploration* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *is a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$ *iff:*

- $Q_e \subseteq Q_f$ *and* $\delta_e \subseteq \delta_f$
- $Q_e = \{s \in Q_f \mid s \leq Max^{<_e}(Q_e)\}$
- $\delta_e = \{(s,t) \in \delta_f \mid (s,t) \leq Max^{\lhd_e}(\delta_e)\}$
- $(\forall x, y \in \delta_e)(x \lhd_e y \implies x \lhd_f y)$

Notice that the definitions of Q_b and δ_b are recursive. The sets Q_e and δ_e are defined by their maximum element: all the lower elements w.r.t $<_e$ (resp. $\lhd_e$) in Q_f (resp. δ_f) are also in Q_e (resp. δ_e). Notice that $(\forall x, y \in Q_e)(x <_e y \implies x <_f y)$ is a consequence of the fourth item in definition 4.5.

Definition 4.6. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be an exploration,* $\langle \{s_0\}, \emptyset, s_0, A, \emptyset \rangle$ *is called the* initial partial exploration *of* e.

Definition 4.7. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *Let* (s,t) *be the next transition to explore in* e. (s,t) *is given by* $Min^{\lhd_f}(\delta_f \setminus \delta_e)$. *Exploring* (s,t) *from* e *yields the partial exploration* $g = \langle Q_g, \delta_g, s_0, A, \lhd_g \rangle$ *of* f *such that:* $Q_g = Q_e \cup \{t\}$, $\delta_g = \delta_e \cup \{(s,t)\}$ *and* $Max^{<_g}(Q_g) = Max^{<_f}(Max^{<_e}(Q_e), t)$, $Max^{\lhd_g}(\delta_g) = (s,t)$ *and finally,* e *is a partial exploration of* g.

Definition 4.8. *Let* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$ *be an exploration.* BACKTRACK(e,s) *is a partial exploration* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *of* f, *in which* $Max^{<_e}(Q_e) = s$ *and* $Max^{\lhd_e}(\delta_e) = Max^{\lhd_f}\{x \xrightarrow{\delta_f} s\}$.

The partial exploration BACKTRACK(f,s) gives the configuration of the depth first exploration at which point s has just been visited for the first time.

4.2 Patching a partial exploration

Definition 4.9. *Let* $p = (u, \mathtt{Rem})$ *be a patch and* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *The set of* impacting updates *in* p *is* $I_u^{DF}(p) = \{(s,t) \in u \mid s \prec_e t\}$.

An update has an impact on the exploration state order iff it breaks one edge of the depth first exploration tree.

Definition 4.10. *Let* $p = (u, \mathtt{Add})$ *be a patch and* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *The set of* impacting updates *in* p *is:*

$$I_u^{DF}(p) = \begin{pmatrix} \{(x,y) \in u \mid x, y \in Q_e \wedge x <_e y \wedge x \not<_b^+ y\} & \cup \\ \{(x,y) \in u \mid x \in Q_e \wedge y \notin Q_e\} & \end{pmatrix}.$$

Definition 4.11. *Let* p *be a patch and* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *The set of* impacting states *of* p *is* $I_s^{DF}(p) = \{s \mid (s,t) \in I_u^{DF}(p)\}$.

Definition 4.12. *Let* p *be a patch and* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *The* impact state *of* p *is* $I^{DF}(p) = \mathrm{Min}^{<_e}(I_s^{DF}(p))$.

Definition 4.13. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *and* $g = \langle Q_g, \delta_g, s_0, A, \lhd_g \rangle$ *be two partial explorations of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$ *and let* $p = (u, t)$ *be a patch. The exploration* g *is a* patching *of* e, *written* $\textsc{patching}(e, p) = g$ *iff*

- $Q_g = Q_e$
- $\delta_g = \delta_e \setminus u$, *if* $t = \mathtt{Rem}$
- $\delta_g = \delta_e \cup \{(s,t) \in u \mid s, t \in Q_e \wedge s <_e I^{DF}(p)\}$, *if* $t = \mathtt{Add}$

Definition 4.14. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *The exploration* $h = \langle Q_h, \delta_h, s_0, A, \lhd_h \rangle$ *is a* patching backtrack *of* e *to the state* $s \in Q_e$ *using the patch* p, *written* $h = \textsc{patchback}_e^p(s)$, *iff* $h = \textsc{patching}(\textsc{backtrack}(e, s), p)$.

Property 4. *Let* $p = (u, \mathtt{Rem})$ *be a patch,* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$ *be an exploration and* $h = \langle Q_h, \delta_h, s_0, A, \lhd_h \rangle = \textsc{patchback}_e^p(I^{DF}(p))$. *The exploration* h *is a partial exploration of* $[f]_p$.

Proof. Let $g = \langle Q_g, \delta_g, s_0, A, \lhd_g \rangle$ be a partial exploration of $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$ such that $g = \textsc{backtrack}(f, I^{DF}(p))$. All the impacting updates (those in $I_u^{DF}(p)$) do not belong to δ_g, i.e. $(\delta_g \cap u) \cap I_u^{DF}(p) = \emptyset$, by definition of BACKTRACK.

We will now see that applying the (non-impacting) updates of u cannot change the ordering and just transforms g so that its updated form $h = [g]_p$ is a partial exploration of $[f]_p$.

Every transition (s,t) in $u \setminus I_u^{DF}(p)$ is such that $s \not\prec_g t$. Then removing $s \xrightarrow{\delta_g} t$ in g have no influence on the depth first state ordering, since $s \not\prec_g t$ means there exists a path in $s \overset{\delta_g}{\Rightarrow} t$ discovered before the exploration of $s \xrightarrow{\delta_g} t$.

Then the ordering of the states in Q_h is equivalent to the ordering in Q_g: $(\forall s, t \in Q_h)(s <_h t \iff s <_g t)$. As g is a partial exploration of f and because only non order-changing updates were removed from g then h is a partial exploration of $[f]_p$. □

Property 5. *Let* $p = (u, \mathtt{Add})$ *be an additive patch,* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$ *be an exploration and* $h = \textsc{patchback}_f^p(I^{DF}(p))$. *The exploration* h *is a partial exploration of* $[f]_p$.

Proof. A transition is in $I_u^{DF}(p)$ when adding it to f would change its depth first ordering. There are two possible cases (represented by the two terms in the $I_u^{DF}(p)$ union of Definition 4.10): either both source and target state (x and y) of an update are already explored in f and x is older than y but x is not an ancestor of y. Adding (x, y) makes x an ancestor of y, thus changing the state order.

Or, y is not known in e. If the exploration of x and all its descendance has been completed then there is no need to do anything: y will be explored when its turn comes. But, knowing this requires more data (or more computation) than what a simple depth first exploration has access to. This implies that to correctly insert y (and all its successors) in the state ordering, the exploration must restart from x.

Like for Property 4, adding to $\text{BACKTRACK}(f, I^{DF}(p))$ all the updates of p that have no effect and that are younger than $I^{DF}(p)$, guarantees that h is a partial exploration of $[f]_p$. □

In other words, Property 4 and 5 state that exploring $[f]_p$ from its initial partial exploration leads to a partial exploration h that can also be obtained from the exploration f by backtracking to g and patching g to h. This means that we can achieve the same result without restarting the computation from scratch.

4.3 Tarjan's algorithm

Definition 4.15. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *A state* s *is* fully explored *in* e *(written* $\text{FEX}_e(s)$*) iff* $\forall (x, y) \in (\delta_f \setminus \delta_e)(s \not\prec_f^+ x \wedge s \not\prec_f^+ y)$.

A state s is fully explored in e iff no transition left to be explored (those in $\delta_f \setminus \delta_e$) can add new descendent state to s. Or following the formal definition, s cannot be an ancestor of any source or destination state of any transition left to be explored.

Definition 4.16. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *The lowlink of a state* s *in* Q_e *is:*
$\text{LOW}(s) = \text{Min}(\{s\} \cup \{\text{LOW}(t) \mid s \xrightarrow{\delta_e} t \wedge s <_e t\} \cup \{t \mid s \xrightarrow{\delta_e} t \wedge t <_e s \wedge t \textit{ is free}\})$

Even though definition 4.16 is short, it might be hard to understand. Definition 4.17 provides an (equivalent) alternative longer version.

Definition 4.17. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *The lowlink of a state* s *in* Q_e *is:* $\text{LOW}(s) = \text{Min}^{<_e}(D(s) \cup (1\text{-STEP}(D(s)) \cap \text{FREE}))$, *where:*

- $D(s) = \{s, t \in Q_e \mid s \prec_e^+ t\}$ *is the descendance of* s *which is younger than* s,
- $1\text{-}\textsc{Step}(X) = \{t \mid s \xrightarrow{\delta_e} t \wedge s \in X\}$, *are the states reachable from* X *by one transition in* δ_e,
- $\textsc{Free} = \{s \in Q_e \mid s \textit{ is free}\}$

The definitions of a lowlink and of the freedom of a state are two dependent definitions, see Definition 4.19 below for that of $\textsc{free}(s)$. Informally, a state is *free* in e iff the strongly connected component it belongs to has not been fully explored. In terms of Tarjan's algorithm, we would say that a state is free iff it has not been assigned to a strongly connected component yet.

From the Definition 4.16, we can see that the lowlink relation is included in δ_e^+, which means that $\textsc{low}$ states the existence of a path between a state s and its lowlink $\textsc{low}(s)$. Let's call D all the states in the subtree of $\langle Q_e, \prec_e, s_0\rangle$, whose root is s. The lowlink $\textsc{low}(s)$ of a state s is the oldest *free* state that can be reached whithin one transition from any state in D. In other words, it gives the oldest *free* state that can be reached while exploring the descendence of s.

Definition 4.18. *The reachability* R_s^f *of a relation* $f : Q \times Q$ *from a state* s *is the set of state* $R_s^f = \{t \mid (s, t) \in f^+\}$

Definition 4.19. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e\rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f\rangle$. *A state* s *is free, written* $\textsc{free}(s)$ *in* e *iff* $\neg \textsc{fex}_e(\mathit{Min}^{<_e}(R_s^{\textsc{low}}))$.

The expression $\textsc{fex}(\mathit{Min}^{<_e}(R_s^{\textsc{low}}))$ means that the oldest state that s can reach through the relation $\textsc{low}$ is fully explored. If this is the case, then s is also fully explored, and more importantly, the strongly connected component to which s belongs has been completely discovered. If $\textsc{fex}(\mathit{Min}^{<_e}(R_s^{\textsc{low}}))$ then $\mathit{Min}^{<_e}(R_s^{\textsc{low}})$ is also the oldest state of its strongly connected component (s belong to the same scc as $\mathit{Min}^{<_e}(R^{\textsc{low}})$), $\mathit{Min}^{<_e}(R_s^{\textsc{low}})$ can uniquely define an $\textsc{scc}$ and is called the *root* of the $\textsc{scc}$.

Definition 4.20. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e\rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f\rangle$. *The source of the lowlink of state* s *in* e, *written* $\textsc{src}(s)$ *is defined as follows:*

$$\textsc{low}(s) \in \{t \in Q_e \mid s \xrightarrow{\delta_e} t \wedge t <_e s \wedge \textsc{free}(t)\} \implies \textsc{src}(s) = \textsc{low}(s)$$

$$\textit{or } (\exists t \in Q_e)(s \xrightarrow{\delta_e} t \wedge s <_e t \wedge \textsc{low}(s) = \textsc{low}(t) \implies \textsc{src}(s) = t)$$

The lowlink of a state s can come from two situations: either $\textsc{low}(s)$ can be directly reached from s, in which case $\textsc{low}(s) = \textsc{src}(s)$, or $\textsc{low}(s)$ comes from a descendant

t (thus t is younger than s), and then $\text{SRC}(s) = t$. In both cases, $\text{SRC}(s)$ indicates the successor of s to follow in order to reach $\text{LOW}(s)$, in other words, $s \xrightarrow{\delta_e} \text{SRC}(s)$ is the first step of a path $p \in s \overset{\delta_e}{\Rightarrow} \text{LOW}(s)$.

In what follows, the notation s^i is used to extract the i^{th} letter of the word s, i.e. $s = s^1 \dots s^i \dots s^{|s|}$, where $|s|$ denotes the length of the word s.

Definition 4.21. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be an exploration. The* common dfs-path *of two states* $a, b \in Q_e$ *is the sequence* $s^1 \dots s^n$ *of states in which* $s^n \prec_e^+ a$, $s^n \prec_e^+ b$, $(\forall i \in [0, n[)(s^i \prec_e s^{i+1})$ *and* $(\forall x)(s^n < x \Rightarrow \neg(x \prec_e^+ a \wedge x \prec_e^+ b))$. *The state* s^n *is called the* Youngest State of the Common Path, *written* $\text{YSCP}(a, b) = s^n$.

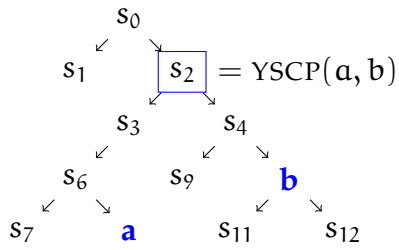

Figure 2: s_2 is the Youngest State of the Common Path of a and b

Example 4.22. *The figure 2 depicts a depth first exploration tree whose root is* s_0. *The path to reach* a *is* $s_0.s_2.s_3.s_6.a$ *and the path to reach* b *is* $s_0.s_2.s_4.b$. *The states* s_0 *and* s_2 *belong to both of them, therefore the state* s_2 *is the youngest state of the common path of* a *and* b.

Definition 4.23. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *Algorithm* T_0 *(Tarjan's algorithm) associates some data* d *with the partial exploration* e. *These are called* T_0*-data and are defined as the tuple* $d = \langle \text{sorder}, \text{index}, S, D, \text{Acc}, \text{low}, \text{src}, \text{scc} \rangle$, *where:*

- $(\forall s, s' \in Q_e)(s <_e s' \wedge \neg(\exists s'')(s <_e s'' <_e s') \iff \text{sorder}(s') = \text{sorder}(s) + 1)$
- $\text{index} = \text{sorder}(\text{Max}^{<_e}(Q_e)) + 1$
- $S \in (s_0.{Q_e}^*)$ *and* $\{s^i \mid i \in [1, |S|]\} = \{s \in Q_e \mid \text{FREE}(s)\}$ *and* $(\forall i \in [1, |S| - 1])(S^i < S^{i+1})$
- $D \in (s_0.{Q_e}^*)$ *and* $D^{|D|} = \text{Max}(Q^e)$ *and* $(\forall i \in [1, |D| - 1])(D^i \prec D^{i+1})$
- $\text{Acc} \in {Q_e}^*$: $\text{Acc} = \text{Proj}^A(D)$ *where:*
  - $\text{Proj}^A(w.x) = \text{Proj}(w).x$, *if* $x \in A$

- $\mathrm{Proj}^A(w.x) = \mathrm{Proj}(w)$, *if* $x \notin A$

- LOW *and* SRC *are defined as previously*

- $\mathrm{SCC}(s) = \mathrm{Min}(R_s^{\mathrm{LOW}})$ *if* s *is not free, or* $\bot$ *otherwise*

The previous algorithm is actually an extension of Tarjan's algorithm used to detect and retrieve accepting run on-the-fly, as proposed by [7]. SRC is the only data-structure that was not used in the original Tarjan's algorithm. A small modification from [7] is that we restored the original LOW management as given by Tarjan, as it revealed more practical to backtrack the algorithm than the variant proposed by [7].

The $\mathtt{sorder}$ relation is an implementation of the depth first exploration order. Two states s and s' are distant by one unit in $\mathtt{sorder}$ iff there is no possible state s'' that can be strictly in between them in $<_e$.

The $\mathtt{index}$ variable gives the order number that will be associated to the next not-yet-visited state encountered. $\mathrm{Max}^{<_e}(Q_e)$ is the last visited state (it is on top of both D *and* S).

S, D and Acc are three stacks (given here as sequences of states). D gives the path in $\prec_e$ to reach $\mathrm{Max}^{<_e}(Q_e)$ from s_0. S holds an ordered sequences of *free* states (from the oldest, s_0, to the youngest $\mathrm{Max}^{<_e}(Q_e)$. Acc is an ordered sequence of *accepting states*. It is a projection of D on the accepting state set A.

SCC is a relation associating a state s to the root of its Scc. If the Scc of s has not been yet fully discovered (s is still *free*), then SCC is undefined, written $\mathrm{SCC}(s) = \bot$.

Definition 4.24. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$ *and* d *be a* T_0*-data of* e*. Let* $p = (u, \mathtt{type})$ *be a patch.*

The T_0*-impact state of* p *is* $I^{T_0}(p) = \mathrm{YSCP}(I_s, \mathrm{Max}^{<_e}(Q_e))$*, where*

$$I_s = \begin{cases} \mathrm{Min}^{<_e}\left(\begin{array}{l}\{s \in Q_e \mid (s,t) \in (u \cap \delta_e) \wedge s <_e t\} \quad \cup \\ \{s \in Q_e \mid (s,t) \in u \wedge t = \mathrm{SRC}_d(s)\}\end{array}\right) & \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{if } \mathtt{type} = \mathtt{Rem} & \\ \mathrm{Min}^{<_e}\left(\begin{array}{ll}\{s \in Q_e \mid (s,t) \in u \wedge t \notin Q_e\} & \cup \\ \{s \in Q_e \mid (s,t) \in u \wedge s <_e t\} & \cup \\ \{s \in Q_e \mid (s,t) \in u \wedge s = t \wedge s \in A\} & \cup \\ \{s \in Q_e \mid (s,t) \in u \wedge t <_e \mathrm{LOW}_d(s) \wedge \mathrm{SCC}_d(t) = \bot\} & \end{array}\right) & \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{if } \mathtt{type} = \mathtt{Add} & \end{cases}$$

Remark. Notice that $s \in Q_e$ is equivalent to $\mathtt{sorder}(s) \neq \bot$.

Property 6. $\textsc{yscp}(x, Max(Q)) = y \iff y = Max\{D^i \mid D^i \leq x\}$

Proof. ($\Leftarrow$) Suppose the opposite: $y = Max\{D^i \mid D^i \leq x\}$ but $y \neq \textsc{yscp}(x, Max^{<_e}(Q))$. By definition of YSCP either $y \not\prec_e^+ x$, or $y \prec_e^+ x$ and there is a younger state z s.t. $z \prec_e^+ Max^{<_e}(Q_e) \wedge z \prec_e^+ x$. If $y \not\prec_e^+ x$ then either $x < y$ or $x > Max^{<_e}(Q_e)$ and both cases contradict the hypothesis (x is supposed to be in Q_e). If $(\exists z)(y < z \wedge z \prec_e^+ Max^{<_e}(Q_e) \wedge z \prec_e^+ x)$ then $y \neq Max^{<_e}\{D^i \mid D^i \leq x\}$.

($\Rightarrow$) Suppose the opposite: $y = \textsc{yscp}(x, Max^{<_e}(Q_e))$ and $y \neq Max^{<_e}\{D^i \mid D^i \leq x\}$. By definition of YSCP, $y \prec_e^+ Max^{<_e}(Q_e)$ and then $(\exists j)(y = D^j)$. Also, there must exists z such that $z = D^k$ and $k > j$ (i.e. $z > y$) such that $z \leq x$, which implies that $z \prec_e^+ x$. Finally, $z = \textsc{yscp}(x, Max^{<_e}(Q))$ which contradicts the hypothesis. □

Property 6 is important to show how to compute $\textsc{yscp}(x, y)$ when x is an arbitrary node and y the last node of D. Notice that this would also be true for any state in D.

Property 7. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *Let* $p = (u, t)$ *be a patch. The exploration* $h = \textsc{patchback}_e^p(I^{T_0}(p))$ *is a partial exploration of* $[f]_p$.

Proof. We need to prove that the backtrack followed by the patch application is done at a point of the exploration for which all the source states of all the updates in p that can change the state ordering of e have not yet been visited.

For a destructive patch p, I_s, that can be viewed as an "ideal" impact state w.r.t. to available data, comes from two cases: either the update breaks the state ordering, or it breaks the LOW relation. In the first case, one would want to check $\prec_e$, but this information is unavailable as a T_0-data. Therefore, $s <_e t$ replaces $s \prec_e t$, as $s <_e t$ can be checked using $\mathtt{sorder}(s) < \mathtt{sorder}(t)$. In the second case, when $t = \textsc{src}(s)$, then the path to lowlink is corrupted and $\textsc{low}(s)$ may not be correct (Notice that (s, t) is necessarily in δ_e). Comparing to $I^{DF}(p)$, since $s <_e t$ is stronger than $s \prec_e t$ and because $t = src(s)$ is an additional constraint, then $I_s \leq I^{DF}(p)$. As $\textsc{yscp}(I_s, Max^{<_e}(Q_e)) \leq I_s$, then $I^{T_0}(p) \leq I^{DF}(p)$. This means that e is a partial exploration of $\textsc{patchback}_e^p(I^{DF}(p))$. Because $\textsc{patchback}_e^p(I^{DF}(p))$ is also a partial exploration of $[f]_p$ then e is a partial exploration of $[f]_p$.

When p is an additive patch, many things changed compared to I^{DF}. Checking $s \prec_d^+ (t)$ is of theoretical possibility, but is unefficient in DF as well as in T_0, as the

only way to check this expression is to recompute the descendance of s, i.e. do a depth first exploration from s. This is why the constraint has been removed in T_0.

New constraints have been introduced. $t <_e \text{LOW}_d(s)$ ensures that $\text{LOW}_d(s)$ is correct and request a recomputation no further than s in that case. Moreover, this recomputation is only required when t is free (otherwise $\text{LOW}(s)$ stays the same). State t freedom is efficiently checked using $\text{SCC}(t)$.

The other new constraint is $s = t \wedge s \in A$. This case means $s \xrightarrow{\delta_e} s$ is an accepting cycle. If no accepting run was previously found, then one has just been found. Or the newly discovered accepting run n is "younger" than the previous one p (i.e. should the computation be restarted from the initial state, n would be found before p).

This means $I_s \leq_e I^{DF}(p)$ and by following the same logic as previously, e is a partial exploration of $[f]_p$. □

Property 8. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$ *and* c *be the* T_0*-data of* e*. Let* $p = (u, \text{type})$ *be a patch. Finally, let* $h = \text{PATCHBACK}_e^p(I^{T_0}(p))$*. If the* T_0*-data* d *is defined as follows:*

let $s = \text{Max}^{<_h}(Q_h)$*,*

- $\text{sorder}_d = \text{sorder}_c \setminus \{(a, b) \in \text{sorder}_c \mid s <_e a\}$
- $(\forall X \in \{\text{LOW}, \text{SRC}\})(X_d = X_c \setminus \{(a, b) \in X_c \mid s \leq_e a\} \wedge X_d(s) = s)$
- $index_d = \text{sorder}_c(s) + 1$
- S_d *(resp.* D_d*)* $= s_0 \dots s$ *where* $s_0 \dots s$ *is a prefix of* S_c *(resp.* D_c*)*
- $\text{Acc}_d = \text{Proj}^A(D_d)$ *(and* Acc_d *is a prefix of* Acc_c*)*

Then d *is a* T_0*-data of* h*.*

Proof. The main idea behind $I^{T_0}(p)$ is that the stack D is the most important because it is the most limiting factor of all the data when its comes to backtracking. From the definition of a T_0-data, D_d must be a path in the relation $\prec_e$. But $\prec_e$ is not available in d. The only path of $\prec_e$ that can be used is D_c itself. Therefore, D_d must be a prefix of D_c. Using Property 6, we see that $\text{Max}(Q_h) = \text{Max}^{<_e}\{D_c^i \mid D_c^i \leq I_s\}$: the state from which the exploration will start after the backtrack is the youngest state in D_c that is older than (or equal to) the state I_s.

Because $h = \text{PATCHBACK}_e^p(I^{T_0}(p))$ then all the states that were free in Q_e are still free in Q_h, therefore all the states in S_d were in S_c and likewise all the states in D_d were in D_c.

As $h = \text{PATCHBACK}_e^p(I^{T_0}(p))$ then h is a partial exploration of e, hence $(\forall x, y \in Q_h)(x <_h y \implies x <_e y)$ and $Q_e \subseteq Q_f$. Then $\mathtt{sorder}_c$ is correct for any state older than or equal to $s = \text{Max}^{<_h}(Q_h)$ (actually, it is true for any state older than or equal to I_s, but for simplicity reason and because $s \leq I_s$, we chose to reset $\mathtt{sorder}_c$ up to s).

We know that $s = \text{Max}^{<_h}(Q_h) = I^{T_0}(p) = \text{YSCP}(I_s, \text{Max}^{<_e}(Q_e))$. Thus, by definition of YSCP, $s \in Q_e$. All the states in any stack D is free, then s is free and by consequence $\text{SCC}(s) = \bot$. Any state t older than s such that $\text{SCC}(t) \neq \bot$ implies that $\text{SCC}(t)$ is not free and thus cannot be on D, i.e. it cannot be an ancestor of s. This means that when backtracking s, $\text{SCC}(s)$ cannot change. Any state t younger than s is necessary a descendent of s, or else s would not be on D. Hence, when $\text{SCC}(t) \neq \bot$, $\text{SCC}(t)$ will be updated when revisiting t from s (since sorder *is* reset, t will be considered as not visited before any attempt to read its SCC). In conclusion, SCC does not need to be reset at all.

As for LOW_d and SRC_d, those two data are computed by exploring the descendence of a state. As the update modifies the descendance of I_s. Any LOW or SRC information concerning a state strictly younger than I_s have to be reset. But resetting up to I_s is not quite correct since some older state might bear an incorrect LOW/SRC information due to the fact that I_s might have been previously fully explored (and then may have propagated its now possibly incorrect lowlink to its father node). This problem is avoided by resetting everything up to s, because $s \in D_d$ which means s has not been fully explored and cannot have propagated its lowlink to an older state.

□

Notice that a depth first exploration is expressed here using partial explorations, which concerns Q and δ only. Therefore the backtrackings performed by the depth first exploration are invisible. A backtracking is accomplished when a state is fully explored: the exploration of its parent state is then resumed. For example, when $s \prec t \prec u$, and $s \xrightarrow{\delta} t$ and $t \xrightarrow{\delta} u$ are the last transitions to explore from s and t respectively, then the backtracking from u to s is done in two *invisible* steps. This abstraction means that *the top of the stack* D *is always* $\text{Max}(Q)$. Describing the exploration into more details would require a pointer to the top of (at least) D.

Also, another information have been abstracted away in this presentation: when backtracking from a state s to a state t, the algorithm must know which transition to explore next. When removing a transition, this data must be updated to take into account the fact that this transition does not exist anymore and thus cannot be the next transition of any state in D.

The backtracking of T_0 is quite brutal: it may require erasing some data for all the states, which is an important overhead just to restart from the initial state. In the next section, we introduce a mechanism for a more efficient backtracking.

4.4 T_1 = T_0 extended with *ranks*

In the following, we will use the operator $+$ as a tuple merging operation.

Definition 4.25. *Let* $t_1 = \langle x_1, \ldots, x_n \rangle \in Q_1 \times \cdots \times Q_n$ *and* $t_2 = \langle y_1, \ldots y_n \rangle \in R_1 \times \cdots \times R_m$ *then* $t_1 + t_2 = \langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle \in Q_1 \times \cdots \times Q_n \times R_1 \times \cdots \times R_m$

Definition 4.26. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *Algorithm* T_1 *extends* T_0 *with* $\langle \mathtt{ranks} \rangle$. *Like previously, this data, called* T_1*-data, is a tuple* $d = c + \langle \mathtt{ranks}_d \rangle$ *associated with the partial exploration* e *where*

1. c *is a* T_0*-data of* e,
2. $(\forall s)(\mathtt{sorder}_d(s) \neq \bot \wedge \mathtt{sorder}_d(s) < \mathtt{index}_d \iff s \in Q_e)$
3. $(\forall s \in Q_e)(\mathtt{ranks}_d(\mathtt{sorder}_d(s)) = s)$
4. $(\forall s)(\mathtt{ranks}_d(\mathtt{sorder}_d(s)) = s \vee \mathtt{ranks}_d(\mathtt{sorder}_d(s)) = \bot)$

The major difference with T_0 is that the element which are also T_0-data are not patched in the same way. T_1 introduces a *validity* zone for all the states: a state s is valid iff it is in Q_e iff $\mathtt{sorder}_d(s) < \mathtt{index}_d$. Previously, a state s was being visited iff $\mathtt{sorder}_d(s) \neq \bot$. This is no longer the case, as $\mathtt{sorder}_d(s) \neq \bot$ can now mean that the state is invalid.

This being said, there is an essential behavior for ranks that must be defined.

Definition 4.27. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *and* $g = \langle Q_g, \delta_g, s_0, A, \lhd_g \rangle$ *be two partial explorations of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$ *such that* $\delta_g = \delta_e \cup \{\mathrm{Max}^{\lhd_f}(\delta_f \setminus \delta_e)\}$ *(i.e. from* e*, the next transition is explored, resulting in the exploration* g*). If* c *and* d *are respectively* T_1*-data of* e *and* g, *then* $\mathtt{sorder}_d(\mathtt{ranks}_c(\mathtt{index}_c)) = \bot$.

Intuitively, this says that the previouly invalid state $\mathtt{ranks}_c(\mathtt{index}_c)$ that is about to be updated to a newly found state must reset its ordering.

Property 9. *Definition 4.27 and Definition 4.26 are consistent.*

Proof. When exploring a new transition (s, t) from an exploration $a = \langle Q_a, \delta_a, s_0, A, \lhd_a \rangle$ resulting in the exploration $b = \langle Q_b, \delta_b, s_0, A, \lhd_b \rangle$, if t is newly discovered, then $\mathrm{Max}^{\lhd_b}(Q_b) = t$. To respect the second invariant, concerning $\mathtt{sorder}$, we have $\mathtt{sorder}_b(t) = \mathtt{index}_a - 1 = \mathtt{sorder}_a(\mathrm{Max}^{\lhd_a}(Q_a)) + 1 = \mathtt{index}_a - 1 + 1$.

Let us suppose that because of some backtracking happening before a, we have $\mathtt{ranks}_a(\mathtt{index}_a) \neq \bot$. In a, $\mathtt{ranks}_a(\mathtt{index}_a)$ is an invalid data, but because $\mathtt{index}_a$ will become a valid ordering number in b, $\mathtt{ranks}_b(\mathtt{index}_a)$ will be indeed valid.

Because it is invalid, $\mathtt{ranks}_a(\mathtt{index}_a)$ can be any state (valid or not), for example it can happen that $\mathtt{sorder}_a(\mathtt{ranks}_a(\mathtt{index}_a)) = \mathtt{index}_a$. If we do not change $\mathtt{sorder}_a$ in that case (i.e. $\mathtt{sorder}_b(\mathtt{ranks}_a(\mathtt{index}_a)) = \mathtt{sorder}_a(\mathtt{ranks}_a(\mathtt{index}_a))$) then we would have two states with the same index: $\mathtt{index}_a = \mathtt{sorder}_b(t) = \mathtt{sorder}_b(\mathtt{ranks}_a(\mathtt{index}_a))$. But this is a major problem, because $\mathtt{sorder}$ serves to establish a total order between states, and then $\mathtt{sorder}$ must be bijective. Also, it could lead to violate the definition of a T_1-data. Since (by definition of T1-data) $\mathtt{sorder}_a(x) = \mathtt{index}_a \implies x \notin Q_a$ and $\mathtt{ranks}_a(\mathtt{sorder}_a(x)) = x$, then $x = \mathtt{ranks}_a(\mathtt{index}_a)$ does not belong to Q_a. When $x \neq t$, then x does not belong to Q_b either, and consequently $\mathtt{sorder}_b(\mathtt{ranks}_a(\mathtt{index}_a)) = \bot$ can be chosen to respect invariant 2 of Definition 4.26 (an alternative would be to respect $\mathtt{sorder}_b(\mathtt{ranks}_a(\mathtt{index}_a)) \geq \mathtt{index}_a$, which is not as straightforward). □

Property 10. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *Applying a patch* $p = (u, t)$ *on the* T_1*-data* c *of* e*, the partial exploration* e *is patched and backtracked to a partial exploration* $h = \mathrm{PATCHBACK}_e^p(I^{T_0}(p))$. *The* T_1*-data* d*, defined as follows, is a* T_1*-data for* h.

- $(\forall x \in \{\mathtt{sorder}, \mathrm{LOW}, \mathrm{SRC}, \mathrm{SCC}, \mathtt{ranks}\})(x_d = x_c)$
- $\mathtt{index}$ *and the three stacks* S, D, Acc *are updated like for* T_0

Proof. The exploration is backtracked to the same state as for T_0, the only difference being *how* the backtracking is performed. Instead of deleting every invalid data, we defined a state validity bound that is modified when new states are discovered.

Property 9 states that the dynamic behavior of $\mathtt{ranks}$ and $\mathtt{sorder}$ respects T_1-data invariants. Also, $s \in Q$ iff $\mathtt{sorder}(s) \neq \bot \wedge \mathtt{sorder}(s) < \mathtt{index}$. Other than that, the proof of Property 8 remains valid here. □

Although the backtracking of a T_1-data is more efficient than that of a T_0-data, having to backtrack to a state in D is a strong limitation. In the next section, we change how the stacks are built and managed, bringing a noticeable improvement on the age of the impact state.

4.5 T2 = generalization of T1 stacks with *prev*

Definition 4.28. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *Algorithm* T_2 *generalizes the stacks of* T_1 *with* $\langle \mathtt{prev} \rangle$. *The* T_2*-data of* e *is a tuple*
$d = \langle \mathtt{sorder}, \mathtt{index}, \mathtt{low}, \mathtt{src}, \mathtt{scc}, \mathtt{ranks}, \mathtt{prev} \rangle$, *where:*

- $\langle \mathtt{sorder}, \mathtt{index}, \mathtt{low}, \mathtt{src}, \mathtt{scc}, \mathtt{ranks} \rangle$ *is a* T_1*-data from which the statcks* S, D *and* Acc *have been removed,*
- $\mathtt{prev} : \{S, D, Acc\} \times Q_e \to Q_e$ *and* $\mathtt{top} : \{S, D, Acc\} \to Q_e$, *where* $\{S, D, Acc\}$ *is an arbitrary enumeration.*
- $$\forall X \in \{S, Acc\}: \ \{w \in Q_e^* \mid w^1 = s_0 \ \wedge \ w^i = \mathtt{prev}(X, w^{i+1})\} = \bigcup_{e \in P} \{w \mid w = \textit{stack } X \textit{ of partial exploration } e\}$$
  where P *is the set of all partial explorations of* e *and stack* X *follows the definition of stacks* S *and* Acc *of Definition 4.23.*
- $(\forall s \in Q_e)(\mathtt{prev}(D, s) = t \iff t \prec_e s)$

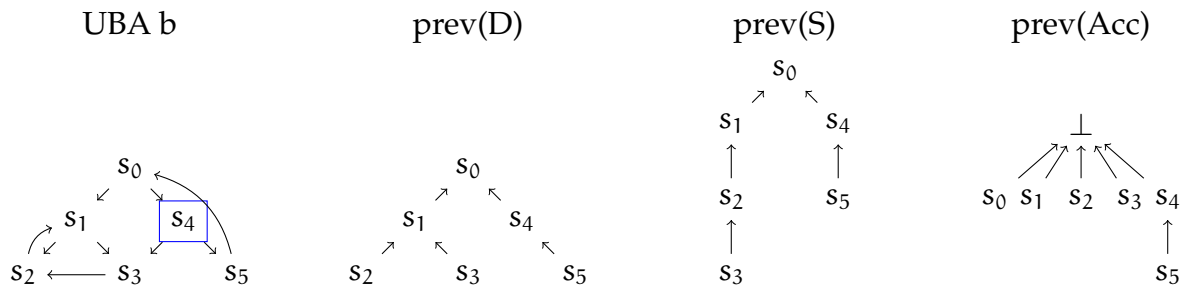

Figure 3: An UBA whose accepting state is s_4 and the value of prev(X), $X \in \{D, S, Acc\}$

Example 4.29. *Figure 3 gives an example of what prev(D) looks like. The UBA* $\mathfrak{b}$ *is explored from left to right: i.e.* $s_0 \rightharpoonup s_1$ *is explored before* $s_0 \rightharpoonup s_4$*. The relation prev(X) are represented as trees: an edge* $s_1 \rightharpoonup s_0$ *in prev(X) means* $(s_1, s_0) \in \mathrm{prev}(X)$*.*

Definition 4.30. *Let* $p = (u, \mathrm{type})$ *be a patch. Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$ *and let* d *be a* T_2*-data of* e*.*

The impact state $\mathrm{I}^{\mathsf{T}_2}(p)$ *of the patch* p *is defined as follows:*

$$\mathrm{I}^{\mathsf{T}_2}(p) = \begin{cases} \mathrm{Min}^{<_e} \left(\begin{array}{l} \{s \mid (s,t) \in u \wedge s \prec_e t\} \cup \\ \{\mathrm{LOW}(s) \mid (s,t) \in u \wedge t = \mathrm{SRC}_d(s)\} \end{array} \right) & \text{if } \mathrm{type} = \mathrm{Rem} \\ \mathrm{Min}^{<_e} \left(\begin{array}{ll} \{s \in Q_e \mid (s,t) \in u \wedge t \notin Q_e\} & \cup \\ \{s \in Q_e \mid (s,t) \in u \wedge s <_e t \wedge s \not\rightarrow_e^+ t\} & \cup \\ \{s \in Q_e \mid (s,t) \in u \wedge s = t \wedge s \in A\} & \cup \\ \{s \in Q_e \mid (s,t) \in u \wedge t <_e \mathrm{LOW}_d(s) \wedge \mathrm{SCC}_d(t) = \bot\} & \end{array} \right) & \text{if } \mathrm{type} = \mathrm{Add} \end{cases}$$

Property 11. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$*. Let* $p = (u, t)$ *be a patch. The exploration* $h = \mathrm{PATCHBACK}_e^p(\mathrm{I}^{\mathsf{T}_2}(p))$ *is a partial exploration of* $[f]_p$*.*

Proof. One difference with $\mathrm{I}^{\mathsf{T}_0}$ is that $\{s \mid (s,t) \in (u \cap \delta_e) \wedge s <_e t\}$ has been replaced $\{s \mid (s,t) \in u \wedge s \prec_e t\}$. The latter is precisely $\mathrm{I}_u^{\mathrm{DF}}(p)$.

Another difference is that s has been replaced by $\mathrm{LOW}(s)$ in $\{\mathrm{LOW}(s) \mid (s,t0 \in u \wedge t = \mathrm{src}(s)\}$, which have no influence $\lhd_e$ or $<_e$.

Finally, $s \not\rightarrow_e^+ (t)$ has been introduced since it is now possible to check it without performing a depth first exploration from s (it is now a matter of following $\mathrm{prev}(D)$ from t). But this operation is still quite expessive and may be advantageously dropped when the cost of computing a state is cheap.

In the end, the exploration $h = \mathrm{PATCHBACK}_e^p(\mathrm{I}^{\mathsf{T}_2}(p))$ is a partial exploration of $[f]_p$. □

Property 12. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$*. Applying a patch* $p = (u, t)$ *on the* T_1*-data* c *of* e*, the partial exploration* e *is patched and backtracked to a partial exploration* $h = \mathrm{PATCHBACK}_e^p(\mathrm{I}^{\mathsf{T}_2}(p))$*. The* T_2*-data* d*, defined as follows, is a* T_2*-data for* h*.*

Let $s = \mathrm{I}^{\mathsf{T}_2}(p)$*,*

- $(\forall x \in \{\mathrm{sorder}, \mathrm{LOW}, \mathrm{SRC}, \mathrm{ranks}, \mathrm{prev}\})(x_d = x_c)$

- $(\forall x \leq_e s)(\text{SCC}_c(x) \in \{\text{SCC}_c(y) \mid y \prec_e^+ s \wedge \neg\text{FREE}(y)\} \implies \text{SCC}_d(x) = \bot)$
- $\text{index}_d = \text{sorder}_c(s) + 1$

Proof. The impact states $I^{T_0}(p)$ and $I^{T_1}(p)$ belongs to D because of the operation $\text{YSCP}(I_s, \text{Max}^{\prec_e}(Q_e))$. Backtracking to I_s would have been better, but no data allowed to do that. With $prev$, things have changed, and I_s can now be used, with the differences exhibited in the proof of Property 11.

The backtracking is a bit different than previously in that s has been replaced by $\text{LOW}(s)$ in $\{\text{LOW}(s) \mid (s, t0 \in u \wedge t = \text{src}(s)\}$. Previously, the impact state s was in D, meaning that s was not fully explored. But this is not true anymore. As a consequence, the lowlink of s can have been propagated up to $\text{LOW}(s)$ while backtracking the depth first exploration. Now that (s, t) has been removed, this propagation may not be correct anymore. To get a correct lowlink for the state younger than $\text{LOW}(s)$, it is then required to re-start the exploration from $\text{LOW}(s)$.

Another consequence of $I^{T_2}(p)$ being potentially fully explored in e is that SCC have to be modified in a different way than previously. Let us suppose that $I^{T_2}(p)$ is fully explored. Then potentially every ancestor of $I^{T_2}(p)$ is also fully explored. This means that all the SCCalong the path $s_0 \xrightarrow{\prec_e^+} I^{T_2}(p)$ must be reset as "not fully discovered". The set $X = \{\text{SCC}_c(y) \mid y \prec_e^+ s \wedge \neg\text{FREE}(y)\}$ regroups all the SCCs of the non free states that are ancestors of $I^{T_2}(p)$. Any state x whose SCCbelongs to X must be un-assigned from that SCC, i.e. $\text{SCC}_d(x) = \bot$.

The words representing a path in $prev(X)$ in the exploration h are those representing X for each partial exploration of h. Because h is partial exploration of e, then $(\forall s, t \in Q_h)(s \prec_h t \implies s \prec_e t)$ and then $(\forall s, t \in Q_h)(s = prev_d(D, t) \implies s = prev_c(D, t))$. Because c is a T_2-data, then $s = prev_c(D, t) \iff s \prec_e t$ and finally $(\forall s, t \in Q_h)(s = prev_d(D, t) \iff s \prec_h t)$. Then $prev_d(D)$ respects the T_2-data definition.

The words representing Acc in each partial exploration of h are all projections of the corresponding stack D on the set of accepting states. Because h is partial exploration of e, then any path of $\prec_e$ is preserved by $\prec_h$ as long as its source and destination are in Q_h. Therefore, all the projections on the set of accepting states are preserved under the same condition. Then $prev_d(Acc)$ respects the T_2-data definition.

As for $prev(S)$, because h is partial exploration of e and c is a T_2-data of e, then all the stack S of all the partial explorations of h are also stacks S for the partial explorations

of e up to h (these are the same explorations). Then for all state $s \leq_h I^{T_2}(p)$, $prev_h(S) = prev_c(S)$ and by consequent $prev_d(S)$ conforms to the definition of a T_2-data. □

The main difference with T_1, is that prev keeps a lot more information than the stacks previously did since *prev* is the tree of the previous stack configurations. Therefore *prev* allows for much more powerful backtracking of the exploration (to the price of more memory consumption). But still, SCC must be updated in order to have a correct backtracking and be able to resume the exploration from $I^{T_2}(p)$. This updating of SCC is avoided in the next section.

4.6 T3 = T2 + DF post-order (rorder + rankr + rindex + maxr)

Definition 4.31. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be an exploration. The depth first post-order* $\preccurlyeq_e$ *is defined as follows:* $s_1 \preccurlyeq_e s_2 \iff (s_1 <_e s_2$ *and* $s_1 \not\prec_e^+ s_2) \vee (s_2 <_e s_1$ *and* $s_2 \prec_e^+ s_1)$

Definition 4.32. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$. *Algorithm* T_3 *extends* T_2 *with* $\langle rorder, rindex, rankr, maxr \rangle$. *The* T_3*-data of exploration* e*, is a tuple* $d = c + \langle rorder, rindex, ranks \rangle$ *associated with the partial exploration* e*, where:*

- c *is a* T_2*-data of* e*,*
- $\Big(\forall s, s' \in Q_e\Big)\Big((FEX(s) \wedge FEX(s') \wedge s \preccurlyeq_e s' \wedge \neg(\exists s'' \in Q_e)(s \preccurlyeq_e s'' \preccurlyeq_e s') \iff rorder(s') = rorder(s) + 1) \Big)$
- $\Big(\forall s\Big)\Big((rorder(s) \neq \bot \wedge rorder(s) < rindex) \iff FEX(s)\Big)$
- $rindex = Max\{rorder(s) \mid s \in Q_e\} + 1$
- $(\forall s \in Q_e)(rankr(rorder(s)) = s)$
- *Let* g *be a partial explorations of* e*:*

$$\Big(\delta_e \setminus \delta_g = \{(s,t)\} \wedge Q_e \setminus Q_g = \{t\}\Big) \implies maxr(t) = rindex_g$$

The relation $rorder$, that maps a state to a natural number (its ordering number), implements the order $\preccurlyeq$. The natural number $rindex$ provides a validity limit for $rorder$ and $rankr$ maps an ordering number to a state. Those data are to $\preccurlyeq$ what $sorder$, $ranks$ and $index$ are for $<$.

The relation $maxr(t)$ saves the value of $rindex$ when t was visited for the first time.

Definition 4.33. *Let* $p = (u, \mathrm{type})$ *be a patch. Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$ *and* d *be a* T_3*-data of* e*.*

The impact state $I^{T_3}(p)$ *of the patch* p *is* $I^{T_3}(p) = I^{T_2}(p)$*, where* $\mathrm{SCC}_d(t) = \bot$ *is replaced with* $\mathtt{rorder}_d(\mathrm{SCC}_d(s)) < \mathtt{rindex}_d$*.*

Property 13. *Let* $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ *be a partial exploration of* $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$*. Applying a patch* $p = (u, t)$ *on* e*, whose* T_3*-data is* c *results in* $h = \mathrm{PATCHBACK}_e^p(I^{T_3}(p))$*. The tuple* d*, defined as follows, is a* T_3*-data for* h*.*

Let $s = I^{T_3}(p)$*, in*

- $(\forall \in \{\mathtt{sorder}, \mathrm{LOW}, \mathrm{SRC}, \mathrm{SCC}, \mathtt{ranks}, \mathtt{prev}, \mathtt{rorder}, \mathtt{rankr}, \mathtt{maxr}\})(x_d = x_c)$
- $\mathtt{index}_d = \mathtt{sorder}_c(s) + 1$
- $\mathtt{rindex}_d = \mathtt{maxr}_c(s)$

Proof. First, we can see that there is need to reset T_3, because now it is possible to see whether a state is fully explored: $\mathrm{FEX}(s) \iff \mathtt{rorder}(s) \neq \wedge\mathtt{rorder}(s) < \mathtt{rindex})$. $\mathrm{SCC}(s)$ is then a valid information iff $\mathtt{rorder}(\mathrm{SCC}(s)) < \mathtt{rindex}$.

From the definition of a T_3-data, $\mathtt{maxr}_c(s) = \mathtt{rindex}_g$ where g is a partial exploration of e for which s was to be discovered for the first time by the next transition. $\mathtt{rindex}_g = \mathrm{Max}\{\mathtt{rorder}_g(s) \mid s \in Q_g\} + 1$. In fact, $g = \mathrm{BACKTRACK}^{T_3}(e, p)$ and then $\mathtt{rindex}_d = \mathrm{Max}\{\mathtt{rorder}_g(s) \mid s \in Q_g\}$ as intended.

Because the backtracking did not change since T_2, i.e. $I^{T_3}(p) = I^{T_2}(p)$, then $\Big(\forall s, t \in Q_h\Big)\Big((s <_h t \implies s <_e t) \wedge (s \prec_h t \implies s \prec_e t)\Big)$. We also have $\Big(\forall s, t \in Q_h\Big)\Big((\mathrm{FEX}(s) \wedge \mathrm{FEX}(t) \wedge s \lessdot_h t) \implies s \lessdot_e t\Big)$. This means $\mathtt{rorder}_h$ is the same as $\mathtt{rorder}_e$ for all states older than or equal to s. As a consequence, this is also true for $\mathtt{rankr}_d$.

In the end, the new data respects the T3-data invariants along with the old ones, which means that d is a T_3-data. □

5 Example

In Figure 4, s_2 is the only accepting state and four patches are applied. Each update is labeled by its patch number and a superscript $+$ or $-$ respectively indicating that the transition is removed or added. Additionally, blue transitions indicates added transitions, while red transitions indicates removed transitions. The four tree depicted below U gives the depth first exploration tree (given by the relation $\prec$). The green boxes

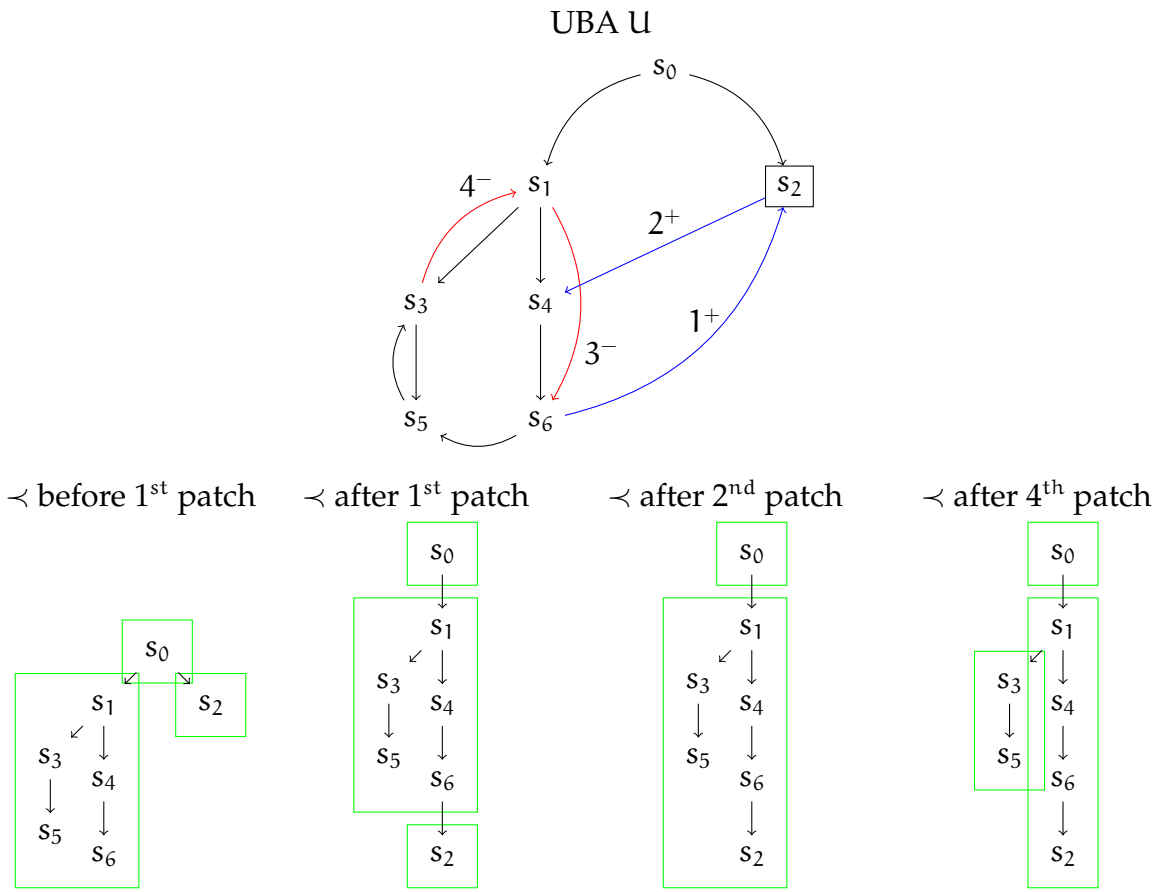

Figure 4: UBA U patched four times

around the states represent the SCCs of $\mathcal{U}$ after the corresponding patches have been applied.

At the end of the first exploration, $\mathcal{U}$ has been completely explored and no accepting run has been found.

Patch number 1 adds a transition from state s_6 to state s_2. s_6 is older than s_2 and $s_6 \not\prec_d^+ s_2$ (s_6 is not an ancestor of s_2). Because $\mathcal{U}$ is completely explored, the stack D is empty, thus T_0 and T_1 restart the exploration from s_0.

Using T_2, because s_6 is fully explored, all the SCCs along the $\prec$ path leading to s_6 have to be reset. s_6 is reachable by $s_0 \xrightarrow{\prec} s_1 \xrightarrow{\prec} s_2 \xrightarrow{\prec} s_4 \xrightarrow{\prec} s_6$, which means that $\mathrm{SCC}(s_6)$, $\mathrm{SCC}(s_4)$, $\mathrm{SCC}(s_1)$ and $\mathrm{SCC}(s_0)$ are to be reset if they are not free. This means SCC is reset for every state in $\{s_0, s_1, s_3, s_4, s_5, s_6\}$ (no state can be free since $\mathcal{U}$ has been explored exhaustively).

Using T_3, the post-order $\lessdot$ is the following: $s_5 \lessdot s_3 \lessdot s_6 \lessdot s_4 \lessdot s_1 \lessdot s_2 \lessdot s_0$. Therefore $\mathtt{maxr}(s_6) = 2$ and then $\mathtt{rindex} = 3$ when backtracking to s_6. Since $\mathtt{rorder}(\mathrm{SCC}(s_6)) = \mathtt{rorder}(s_1) = 4 \geq 3$, the state $\mathrm{SCC}(s_6)$ is considered invalid (no need to reset it). But, here, that information is not actually necessary to perform the exploration of s_2 from s_6.

After the re-exploration from s_6, no accepting run has been found and $\mathcal{U}$ has been completely explored.

Patch number 2 adds a transition from state s_2 to state s_4. The lowlink of s_2 is s_2 itself and s_4 is older than s_2. When exploring s_2, s_4 was still free because $s_4 \prec^+ s_2$. This means the lowlink of s_2 have to be updated with s_4.

Becaues $\mathcal{U}$ has been completely explored, the stack D is empty and T_0 and T_1 are restarted from s_0.

As for T_2, s_2 is reachable by $s_2 \prec s_6 \prec s_4 \prec s_1 \prec s_0$, which means the SCC of every state is reset. Then exploration is restarted form s_2.

Using T_3, we have $s_5 \lessdot s_3 \lessdot s_2 \lessdot s_6 \lessdot s_4 \lessdot s_1 \lessdot s_0$, restarting from s_2 means $\mathtt{rindex} = 2$ after the backtrack and $\mathtt{rorder}(\mathrm{SCC}(s_2)) = \mathtt{rorder}(s_2) = 2$. Then $\mathrm{SCC}(s_2)$, $\mathrm{SCC}(s_1)$ and $\mathrm{SCC}(s_0)$ are known to be invalid. Like T_2, the exploration is then restarted from s_2.

From s_2, the state s_4 is encountered. This state has been visited and it is free, either because $\mathrm{SCC}(s_4) = \bot$ for the variants T_0 to T_2, or because $\mathtt{rindex} \leq \mathtt{rorder}(\mathrm{SCC}(s_4))$ in T_3 ($\mathtt{rindex} = 2$ and $\mathtt{rorder}(\mathrm{SCC}(s_4)) = \mathtt{rorder}(s_1) = 5$). s_2 is an accepting state, which means that $\mathtt{Acc}$ is not empty and because s_4 is older than the top element of $\mathtt{Acc}$, then we found an accepting run: $s_0 \xrightarrow{\prec} s_1 \xrightarrow{\prec} s_4 \xrightarrow{\prec} s_6 \xrightarrow{\prec} s_2 \xrightarrow{\mathrm{SRC}} s_4$. Note that because an accepting run is found, no SCC has actually been discovered by any variant T_0 to T_3.

Indeed, $\text{SCC}(s_1)$ will only be discovered when backtracking the depth first exploration from s_2 to s_1 and $\text{SCC}(s_0)$ when backtracking from s_1 to s_0 (because every state have been visited when the accepting run is found).

Patch number 3 is essential harmless: it removes the transition $s_1 \xrightarrow{\delta} s_6$. Since s_6 has been discovered from the exploration of s_1 *through* s_4, $s_1 \xrightarrow{\delta} s_6$ does not influence the state order ($s_1 \xrightarrow{\prec} s_4 \xrightarrow{\prec} s_6$ is still here, or this can also be seen by remarking that (s_1, s_6) is not in $\prec$ but $s_1 < s_6$). Hence, no re-exploration has to be performed.

The last patch removes $s_3 \xrightarrow{\delta} s_1$ and this transition makes the lowlink of s_3 invalid since $\text{LOW}(s_3) = \text{SRC}(s_3) = s_1$. At this point, $D = s_0.s_1.s_4.s_6.s_2$ and $\text{YSCP}(s_3, s_2) = s_1$. Then the exploration is restarted from s_1 in T_0 and T_1.

The backtracking in T_0 is quite uninteresting as every invalid data is being updated. T_1, however, does not invalid the ordering number $\mathtt{sorder}$ of every invalid state but use $\mathtt{index}$ instead. $\mathtt{sorder}(s_1) = 1$ then $\mathtt{index} = 2$ when restarting from s_1. When exploring s_3, as $\mathtt{index} \leq \mathtt{sorder}(s_3)$ then s_3 is considered invalid and its exploration continued to s_5 wich is also considered invalid, then from s_5, s_3 is found but now $\mathtt{index} = 4$ and $\mathtt{sorder}(s_3) < \mathtt{index}$, then s_3 is considered as already visited. Although we did not describe it before, this mechanism is used by T_2 and T_3. After backtracking to s_3, it is found fully explored, and because $\text{LOW}(s_3) = s_3$, then a new SCChas been found, namely $\{s_3, s_5\}$. After that, the exploration continues from s_1 until it finds the same accepting run as previously.

As for T_2, SCC is reset for the state s_3, s_1 and s_0 (the other state of $\text{SCC}(s_1)$ are younger than s_3). Then exploration is restarted from s_3.

For T_3, the exploration is also restarted from s_3. $\mathtt{rindex} = \mathtt{maxr}(s_3) = 0$, which means no state is fully explored, and then no SCC can be different from $\perp$ (and if it is this means it is not valid).

6 Heuristics

In the previous section, all that was considered were the effects an update can have on the data-structures. Essentially, when breaking the state ordering, or the lowlink, or the source of a lowlink, an update was considered harmful and required a backtracking. But, in the end, what is important is that the output of the algorithm, the accepting run, is "correct" is the sense that it is *a* valid one, and not even necessarily the first to be found in the depth first exploration meaning.

The last patch of the example in section 5 is a good demonstration: it tries to remove a transition that cannot break the previously found accepting run. Restarting the exploration as a response to this update is then unecessary.

In the following, we present different heuristics exploiting this observation which may yield another speed increase.

But, before continuing on, recall that in section 3, we presented a pattern of lazy patch application. The heuristics will be given following this pattern.

To subsume what we presented in Section 4, the following shows how to fit them in the lazy patch application pattern.

Definition 6.1. *Let* $x \in [0,3]$. *Let* e *be an exploration.*

The lazy patch application $\lambda_2^{T_x}$ *is defined as follows:*

- r *is invalidated by the patch* $p = (u,t)$ *iff* $I^{T_x}(p) \neq \bot$
- *Restart the computation:* $h = \text{PATCHBACK}_e^p(I^{T_x}(p))$ *and then restart* T_x *from* h
- *Manage the patch* p*:* $e \leftarrow \text{PATCHING}(e,p)$

6.1 Prudent

The *prudent* heuristics that is presented here can hardly be called as such, since it is better to use it than λ_2 in a vast majority of the cases.

The prudent heuristics stores the oldest impact state of any *impacting* update that cannot invalid the accepting run or introduce a new one if none was found.

An impacting *destructive* update can change this oldest impact state when it is not part of the accepting run, or when no accepting run was previously found.

An impacting *additive* update can change this oldest impact state when any accepting run was previously found.

As soon as a run changing update appears (either by adding a transition when there exists no accepting run, or by removing a transition used by the accepting run), the computation is backtracked to the saved oldest impact state.

Definition 6.2. *Let* $x \in [0,3]$. *Let* e *be an exploration.*

The lazy patch application $\lambda_3^{T_x}$ *(prudent heuristics) is defined as follows:*

- *Initially,* $\texttt{old} = \bot$
- r *is invalidated by the patch* $p = (u,t)$ *iff*

$$t = \mathtt{Rem} \wedge r \neq \epsilon \wedge u \twoheadrightarrow r$$

or $t = \mathtt{Add} \wedge r = \epsilon$

- *Restart the computation:*
  - $h = \text{PATCHBACK}_e^p(\text{Min}^{<_e}(\mathtt{old}, I^{T_x}(p)))$
  - $\mathtt{old} \leftarrow \bot$
  - *and then restart* T_x *from* h
- *Manage the patch* p*: if* $I^{T_x}(p) \neq \bot$ *then* $\mathtt{old} \leftarrow \text{Min}^{<_e}(\mathtt{old}, I^{T_x}(p))$*. In any case,* $e \leftarrow \text{PATCHING}(e, p)$*.*

For simplicity reason, we suppose that $x <_e \bot$ is defined and is *true* for any state $x \in Q_e$. Also, when restarting the computation $I_s^{T_x}(p) \neq$ because r is invalidated by the patch p: the invalidation conditions are included in those used to get the impact state of each T_x. Therefore, $\text{Min}^{<_e}(\mathtt{old}, I_s^{T_x}(p))$ will always be defined.

In the worst case case, the heuristics does not help at all, and may even introduces some overhead when the computation of $u \twoheadrightarrow r$ becomes significant (which is quite unusual).

The good "prudent" feature of this heuristics is that $\lambda_3^{T_x}$ cannot introduce more backtracking and re-exploration than $\lambda_2^{T_x}$.

6.2 Imprudent

In some cases, restarting from the oldest impacting state of the updates which did not change the accepting run, as it is done in the prudent heuristics, can produces excessive backtracking.

Here, we introduce another heuristics that is a two-sided blade: although it can improve on the previous heuristics, it may also require more backtracking.

The idea of this heuristics is to bet on the locality of the impacts: if the current patch has an impact, this impact should not imply the previously non run changing impacting updates. This can be sometimes very wrong and will require, in the worst case, one backtracking *per update* (instead of *per patch* for the prudent heuristics).

Definition 6.3. *Let* $x \in [0, 3]$*, let* e *be an exploration and let* $I_u^{T_x}(p) = \{x \in u \mid I^{T_x}((\{x\}, t)) \neq \bot\}$ *be the set of impacting updates of the patch* p*. Let* $T(\mathtt{Add}) = A$ *and* $T(\mathtt{Rem}) = R$ *and finally,*

$\bar{T}(\mathtt{Add}) = R$ *and* $\bar{T}(\mathtt{Rem}) = A$. *The lazy patch application* $\lambda_3^{T_x}$ *(prudent heuristics) is defined as follows:*

- *Initially,* $A, R = \bot, \bot$
- *The run is invalidated by the patch* $p = (u, t)$ *in the same way as* λ_2
- *Restart the computation:*

  $T(t) \leftarrow T(t) \cup I_u^{T_x}(p)$ *and* $e \leftarrow \text{PATCHING}(e, p)$

  do
  $e \leftarrow \text{BACKTRACK}\left(e, I^{T_x}(T(t), t)\right)$
  $(\forall X \in \{A, R\})(X \leftarrow X \setminus \{(s, t) \in X \mid I^{T_x}(T(t), t) \leq s\})$
  Restart T_x *from exploration* e

  while r *is invalidated by* $(A, \mathtt{Add})$ *or* $(R, \mathtt{Rem})$

- *Manage the patch* p*:* $T(t) \leftarrow T(t) \cup I_u^{T_x}(p)$ *and* $e \leftarrow \text{PATCHING}(e, p)$.

The set $I_u^{T_x}(p)$ yields the updates of p having an impact on T_x: for each update $x \in u$, if the patch $(\{x\}, t)$ holding the single update x has an impact then $x \in I_u^{T_x}(p)$.

When the computation needs to be restarted, the first step is to update $T(t)$: $T(t) \leftarrow T(t) \cup I_u^{T_x}(p)$, meaning that the set of impacting updates of the patch p are added to the set A or R according to the type t of the patch p. The patch is then incorporated into e.

Then e is backtracked to the oldest impact state of the patch $(T(t), t)$. The sets A and R are then updated: each update (s, t) whose source state s is younger than or equal to the impact state of $(T(t), t)$ is removed from the set.

Finally, the exploration is restarted from e. The loop ends when the previous accepting run is not invalidated by any update in A or R (depending on whether an accepting run has been found by the previous exploration).

One the one hand, this heuristics will only trigger necessary re-explorations, which may bring a big improvement over the prudend heuristics, but on the other hand, some required backtracking may be missed at a given step and only be detected at a subsequent one. Therefore, using this heuristics may require more re-explorations than the prudent heuristics. Also, the list of pending update may become big and parsing it may become a problem.

Notice that any pending update whose source state is younger than the impact state is removed from the list of pending updates, as it will get covered when the exploration reaches it (indeed, remember that every update is inconditionally inserted into the graph).

As for the worst case of the heuristics: consider a sequence of n patches for which all the updates are impacting but only the last one invalidates the accepting run. This last impacting patch has m updates and the only invalidating update u has the youngest source state among all the other updates of the patch and of both A and R. This means all the updates of all the patches are in A or R, except of cause for u. Now u triggers a re-computation that switches the status of the accepting run: if the accepting run was previously found then the re-computation cannot find any accepting run anymore or if none was previously found then the re-computation find a new accepting run. When testing this new run for a possible invalidation from A or R, only one update u' in the corresponding set is found to be invalidating and, as for u, it has the youngest source state of A and R. Again re-computating from u' changes the status of the run, and the process is repeated until both A and R are empty. In this case, which can definitely happen, a backtrack and a re-exploration is required for every *update*.

6.3 Mixed

A combination of the two previous heuristics is possible by computing both the oldest (non run changing) impacting update $\mathtt{old}$ and the pending updates A, R and use one or the other heuristics based on whether a given threshold has been reached.

One such threshold can be the number of patches. When a whole patch does not invalidate the previously found accepting run (it may be empty if none was found), no re-computation is required. Therefore a re-exploration has been effectively avoided for this patch.

Definition 6.4. *Let* $x \in [0, 3]$*, let* e *be an exploration and let* $I_u^{T_x}(p) = \{x \in u \mid I^{T_x}((\{x\}, t)) \neq \bot\}$ *be the set of impacting updates of the patch* p*. Let* $T(\mathtt{Add}) = A$ *and* $T(\mathtt{Rem}) = R$ *and finally,* $\bar{T}(\mathtt{Add}) = R$ *and* $\bar{T}(\mathtt{Rem}) = A$*. The lazy patch application* $\lambda_4^{T_x}$ *(mixed heuristics) is defined as follows:*

- *Initially,* $\mathtt{old}, A, R = \bot, \bot, \bot$ *and* $\mathtt{unavoided}, \mathtt{nb\_patch} = 0$
- *The run is invalidated by the patch* $p = (u, t)$ *in the same way as* λ_2

- *Restart the computation:*

  if $\mathtt{unavoided} < \mathtt{nb\_patch}$ *then* **//uses Imprudent heuristics ...**

  $T(t) \leftarrow T(t) \cup I_u^{T_x}(p)$ *and* $e \leftarrow \text{PATCHING}(e, p)$

  increment $\mathtt{nb\_patch}$

  do

  if $\mathtt{avoided} < \mathtt{nb\_patch}$ **//... and continue to use it ...**

  $e \leftarrow \text{BACKTRACK}\left(e, I^{T_x}(T(t), t)\right)$

  $(\forall X \in \{A, R\})(X \leftarrow X \setminus \{(s, t) \in X \mid I^{T_x}(T(t), t) \leq s\})$

  Restart T_x *from exploration* e

  increment $\mathtt{unavoided}$

  else **//... until the nb_patch threshold is reached**

  $e \leftarrow \text{BACKTRACK}(e, \text{Min}^{<_e}(\mathtt{old}, I_s^{T_x}(p)))$

  restart T_x *from* e

  increment $\mathtt{unavoided}$

  while r *is invalidated by* $(A, \mathtt{Add})$ *or* $(R, \mathtt{Rem})$

  else **//uses Prudent heuristics**

  $e \leftarrow \text{PATCHBACK}_e^p(\text{Min}^{<_e}(\mathtt{old}, I_s^{T_x}(p)))$

  restart T_x *from* e

  increment $\mathtt{unavoided}$ *and* $\mathtt{nb\_patch}$ *and* $\mathtt{old} \leftarrow \bot$

- *Manage the patch* p*:*

  if $I^{T_x}(p) \neq \bot$ *then* $\mathtt{old} \leftarrow \text{Min}^{<_e}(\mathtt{old}, I^{T_x}(p))$.

  $T(t) \leftarrow T(t) \cup I_u^{T_x}(p)$

  $e \leftarrow \text{PATCHING}(e, p)$.

  increment $\mathtt{nb\_patch}$

The worst case of this heuristics is one re-exploration by patch. In some cases there may be more re-exploration than using the prudent heuristics, and in some cases there may be less, but this re-exploration number is necessarily bounded by the number of patches because when $\mathtt{avoided}$ reaches $\mathtt{nb\_patch}$ the prudent heuristics is used and we already saw that the worst case for the prudent heuristics is the number of patches.

A way to ensure that they cannot be more re-exploration than using λ_2 is to add $I^{T_x}(p) \neq \bot$ as a condition for the whole "Manage the patch p" block. Indeed, when

$I^{T_x}(p)$, λ_2 necessarily require a re-exploration, which is here avoided because the impacts of p does not invalidate the accepting run.

The avoidance threshold could be extended to re-computations that are considered as avoided by setting another threshold on the percentage of states that have avoided a recomputation w.r.t. the total number of states computed during the previous iteration. For example, when a backtracking avoids the recomputation of 80 percent of the previously computed state, the re-computation could be considered as *avoided*, even through this is not strictly correct.

7 Updating the accepting states

In the previous sections, we did not take into account the fact that the set of accepting state may change from a patch to another.

The first thing to acknowledge is that removing or adding accepting states does not alter any state ordering nor any lowlink. Instead it only impacts the stack Acc, which is only used to detect the existence of an accepting run.

Let $e = \langle Q_e, \delta_e, s_0, A, \lhd_e \rangle$ be a partial exploration of $f = \langle Q_f, \delta_f, s_0, A, \lhd_f \rangle$ and r be an accepting run of e or $r = \epsilon$ if no accepting run was found in e.

- $r = \epsilon$ and s is removed from A. In that case, notice that $e = f$.
  - in T_0 or T_1, because the exploration has explored and discovered all the SCCs, then D is necessarily empty and thus so is Acc. Since $r = \epsilon$, removing s cannot introduce a new accepting run, therefore there is nothing to update in that case.
  - in T_2 or T_3, $prev(Acc)$ will become corrupted:
    - either update $prev(Acc)$ into $prev'(Acc)$: $(\forall x \in Q_e)(prev(Acc, x) = s \implies prev'(Acc, x) = prev(Acc, s))$
    - or restart from s

    Using an heuristics like the prudent heuristics, the computation can be avoided and s can be considered a non-run-changing impact state.
- $r = \epsilon$ and s is added to A
  - s is necessarily visited

- in T_0 or T_1, because s may introduce an accepting run, but because D is empty in e, the exploration have to be backtracked to s_0.
- in T_2 or T_3, s introduces an accepting run *iff* s is part of a non-trivial SCC. This can be checked using $prev(D, sorder[ranks[s] + 1]) = s$ and $scc[ranks[s] + 1] = scc[s]$. In other words, the succesor t in the ordering is a direct successor of s ($s \xrightarrow{\delta} t$) and both s and t belong to the same SCC. If this is the case, the exploration have to be backtracked to s.

- $r \neq \epsilon$ and s is removed from A.
  - in T_0 or T_1, the stack Acc must be updated and then the exploration must be backtracked to $\text{YSCP}(s, top(D))$.
  - in T_2 or T_3, the stack Acc must be updated and the exploration must be backtracked to s.

  Removing s from A can only change the accepting run iff s is on top of Acc. Indeed, the accepting run has been detected using this value and as long as it does not change, the accepting run cannot change. Therefore, using the prudent heuristics, removing s from A only triggers a re-computation when s is on top of Acc. Otherwise, s is considered as a non-run-changing impacting state.

- $r \neq \epsilon$ and s is added to A. In this case s can introduce a "younger" run, i.e. a run discovered before r.
  - in T_0 or T_1, the stack Acc must be updated and then the exploration must be backtracked to $\text{YSCP}(s, top(D))$.
  - in T_2 or T_3, the stack Acc must be updated and the exploration must be backtracked to s.

  Adding s to A cannot break the accepting run, it may only introduce "younger" ones. Therefore, using the prudent heuristics, all the state s in that case are considered as non-run-changing impact states.

8 Experiments

In this section, we present some experiments conducted using a prototype tool that can construct random UBAs. Except for the mixed heuristics everything has been implemented in the tool.

The time consumption of the algorithm T_x ($x \in [0, 3]$) is separated from the patching and backtracking process. Patching and backtracking means modifying the underlying graph of an UBA and performing the requested backtracking (according to the algorithm and the heuristics). The time spent in T_x represents the time necessary to find an accepting run (or to explore everything when no accepting run can be found) from the impact state of the previous backtracking.

As a global performance indicator, the total amount of state as computed by a reference computation (for which the computation always restarted the computation from the initial state) is compared to the total amount of state computed by the given algorithm/heuristics combination. This gives the overall computation avoidance of the algorithm/heuristics compared to the reference computation.

Finally, when backtracking, the maximum amount of states whose computation can be avoided is exactly the amount of visited states in the previous exploration (given by $\mathtt{index}$). Then $\mathtt{ranks(impact)/index}$ describles the rate of re-computation avoidance achieved by the experiment, where $\mathtt{impact}$ is the state to which the exploration is backtracked to.

These measures will be depicted according to the key of Figure 5. If colors are not

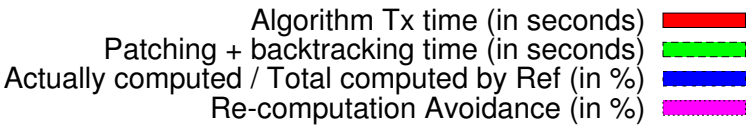

Figure 5: Measures used for the experiments

available, the measures taken from top to bottom will always be displayed from left to right.

First experiment. This experiment tests the effect of having one million patches each of which is consituted by a single update. Moreover, for each case, the breadth of the UBA is set to 2, 10 and 100 while the number of states is set to 50 000, 10 000 and 1 000, so that the maximal number of transition still remains equal to 100 000 (it is initially equal

to 100 000 and will vary as the updates happen, but it will never exceed this amount). Finally, there is only one accepting state.

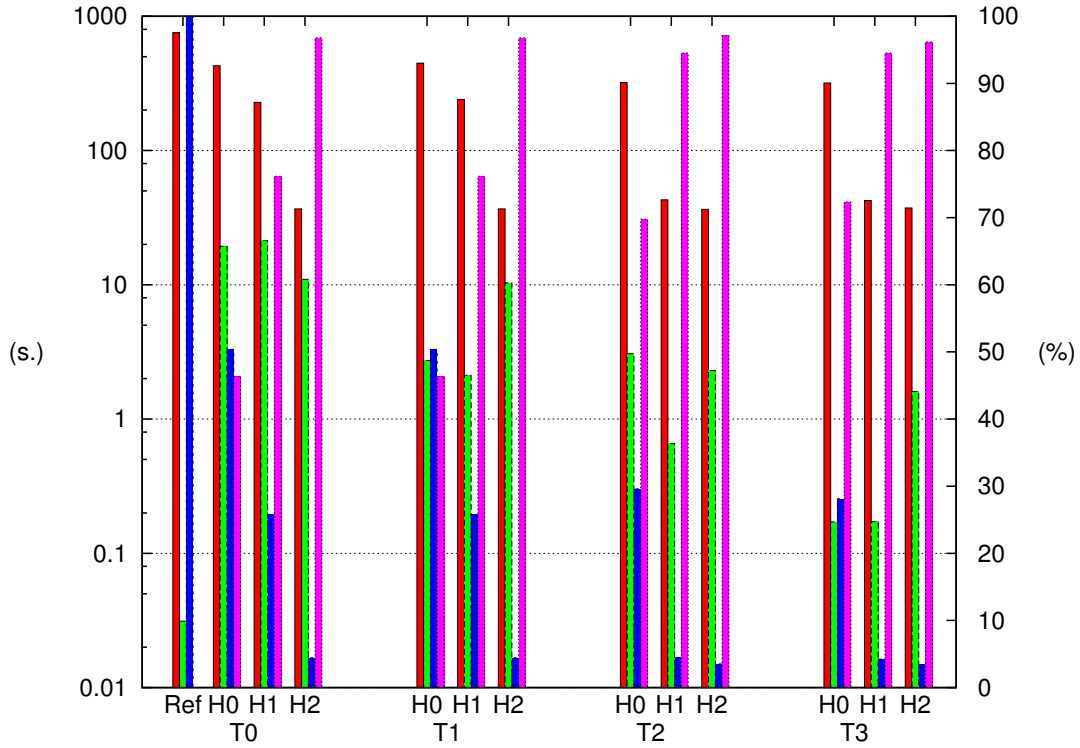

Figure 6: (Exp. 1) Number of states: 50 000, breadth: 2

The fact that each patch only contains one update means the UBA stays almost the same after each patch, but the great amount of patches makes it changes quite significantly in the end.

The first interesting result of this experiment is how the potential speed-up can be orders of magnitude higher than the reference computation. The other interesting result is how the heuristics can greatly improve the re-computation avoidance rate from H_0 to H_1 and from H_1 to H_2.

But, not everything is ideal: in Figure 8 the time taken by H_2 in T_0 and T_1 is quite high and get very close the time taken by reference computation.

Second experiment. This experiment tests the effect of having only 10 patches made of 1 000 000 updates each. Like previously, the underlying graph of the UBA varies in breadth and number of states so that there at most 2 000 000 transitions. Also like previously, there is only one accepting state.

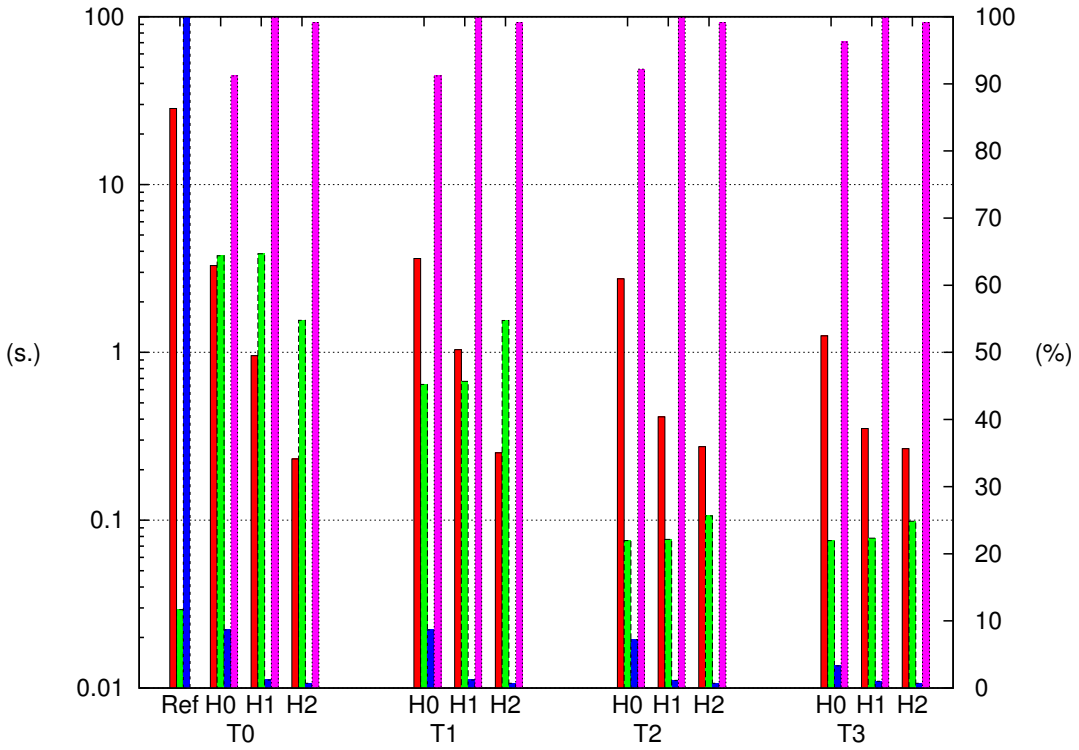

Figure 7: (Exp. 1) Number of states: 10 000, breadth: 10

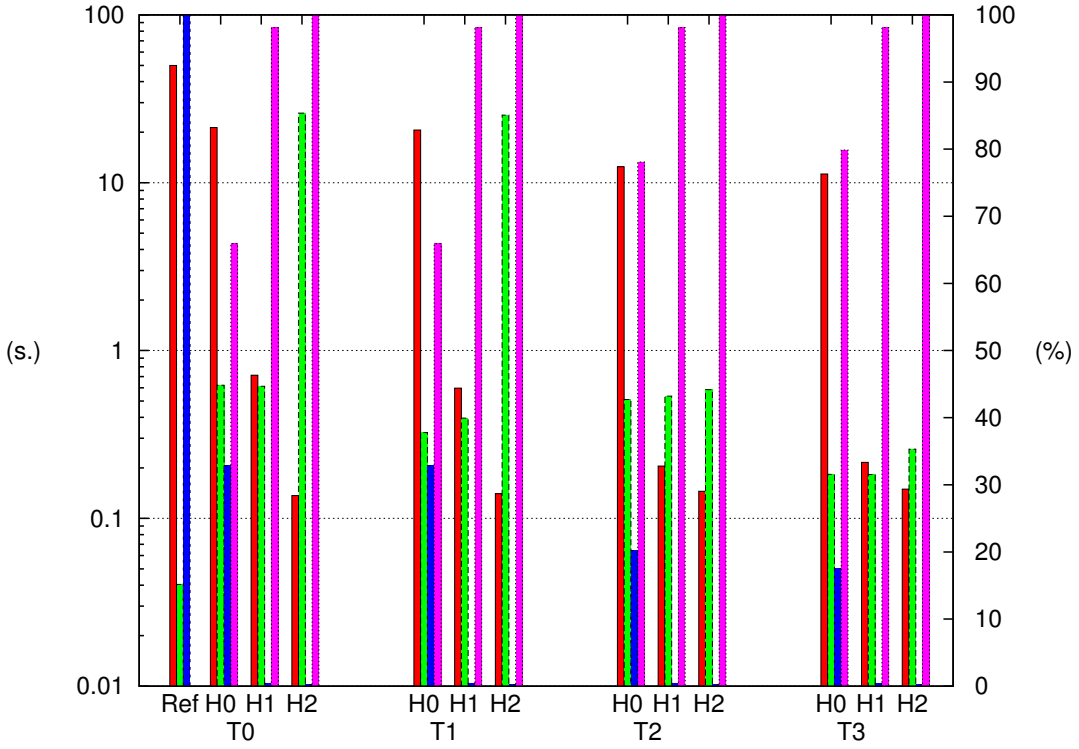

Figure 8: (Exp. 1) Number of states: 1 000, breadth: 100

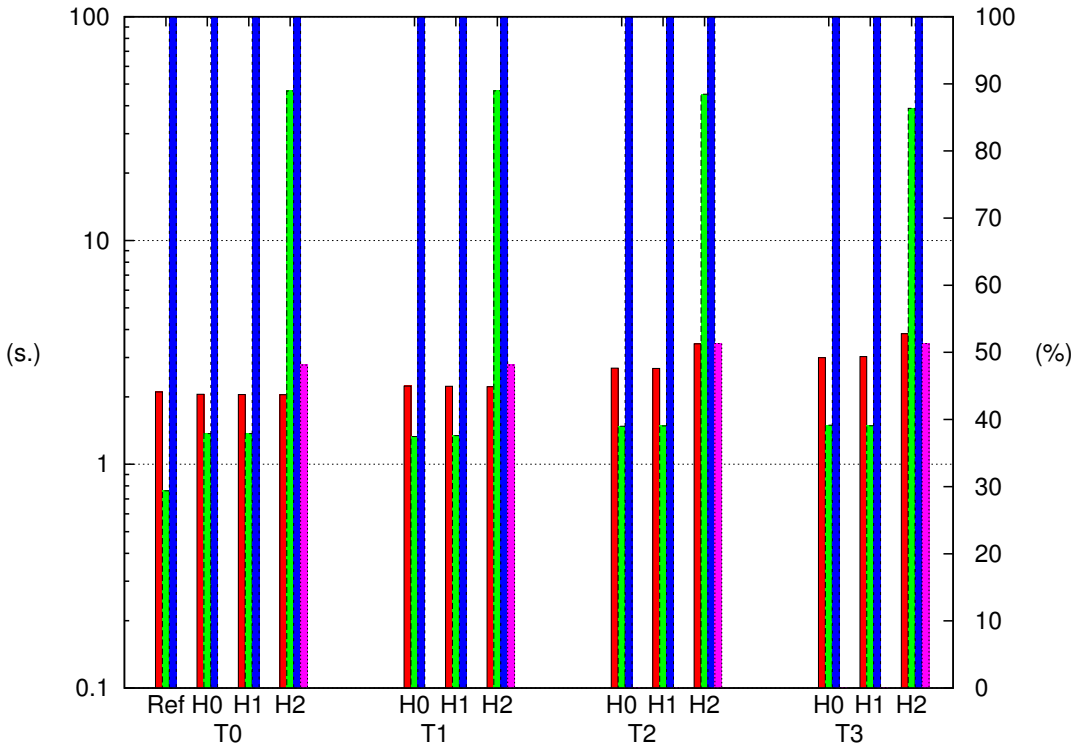

Figure 9: (Exp. 2) Number of states: 1 000 000, breadth: 2

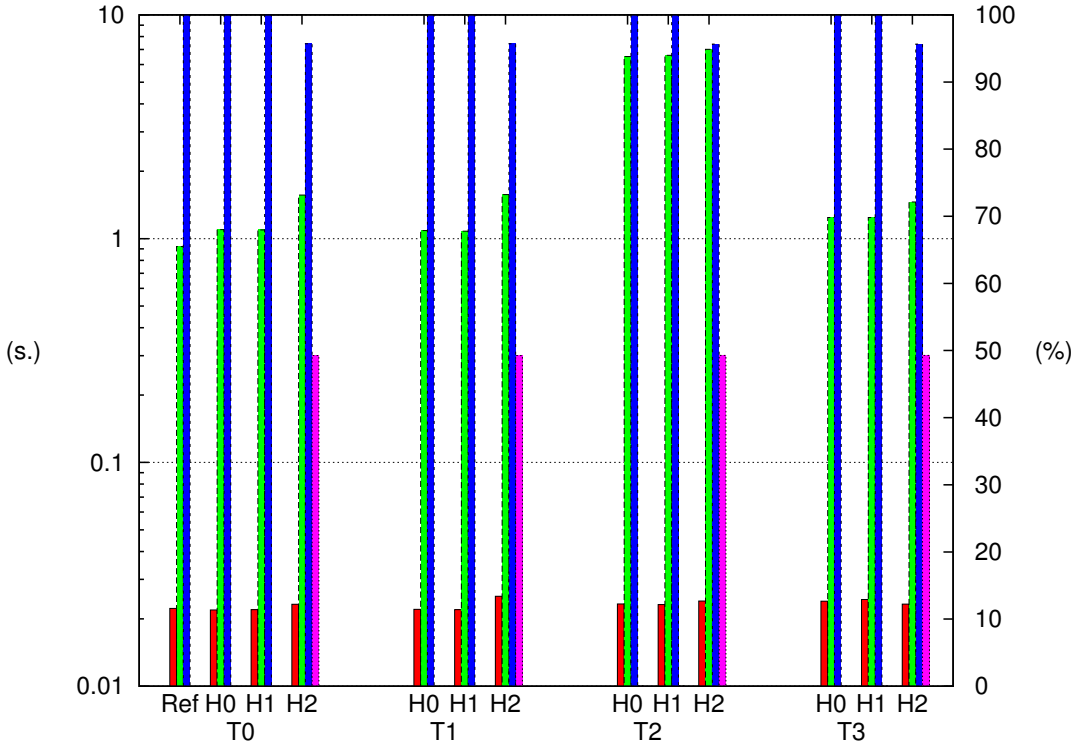

Figure 10: (Exp. 2) Number of states: 10 000, breadth: 200

The interesting result of this experiment, depicted by Figures 9 and 10 is the overall weak performance of all the algorithms when compared to the reference computation, but it especially demonstrate that H_2 can have a really bad performance. This result is due to the heavy modification of the UBA: each path can modify at most half the transitions. This heavy modification, because it is random, have a great chance to have impact state very close to the initial state.

As for H_2, even though its avoidance rate is 50%, it requires a lot more re-computations and ends up with a 128% of global avoidance rate in Figure 9 i.e. it computes more state than the reference computation. All that is due to its exessively aggressive laziness when it comes to check which update has the oldest impact.

As for Figure 10 the inefficiency mostly comes from the backtracking itself and shows how big its impact on the timings can be.

Third experiment. This experiment tests the effect of a big UBA, comprising 20 000 000, 200 000 000 and 500 000 000 transitions with a significant amount of accepting states (1 000 000). Like previously, this transition number is an upper bound and is made of a fixed amount of states (10 000 000) and of variations on breadth (2, 20 and 50). 1 000 patches each constituted of 5 000 updates are applied for the experiment of Figures 11 and 12, while 10 000 patches of 500 updates were applied for the experiment of Figure 13.

The amount of updates is quite small compared to size of the UBA, which leads to very good results in the experiment. The only inefficiency is located to T_0 and is linked to its inefficient backtracking.

Fourth experiment. Unlike the previous experiments, this one keeps the same state/breadth ratio but instead the patches are generated so that the source state of each update is within some varying distance from the last impact state. Distance is to be understood as a distance in the state ordering: the last visited state s ($\mathrm{ranks}(s) = \mathrm{index}-1$) is the most distant from initial state. The patches in this experiment vary according to a given *minimal* distance from the initial state. This minimal distance is stated in percentage: 100% is only possible for the last visited state and 0% means no restriction on the the distance (the patches source and target states can be any visited states younger than the initial state: all the states verify that). The distance restriction applied in this experiment are 0%, 30%, 60% and 90%.

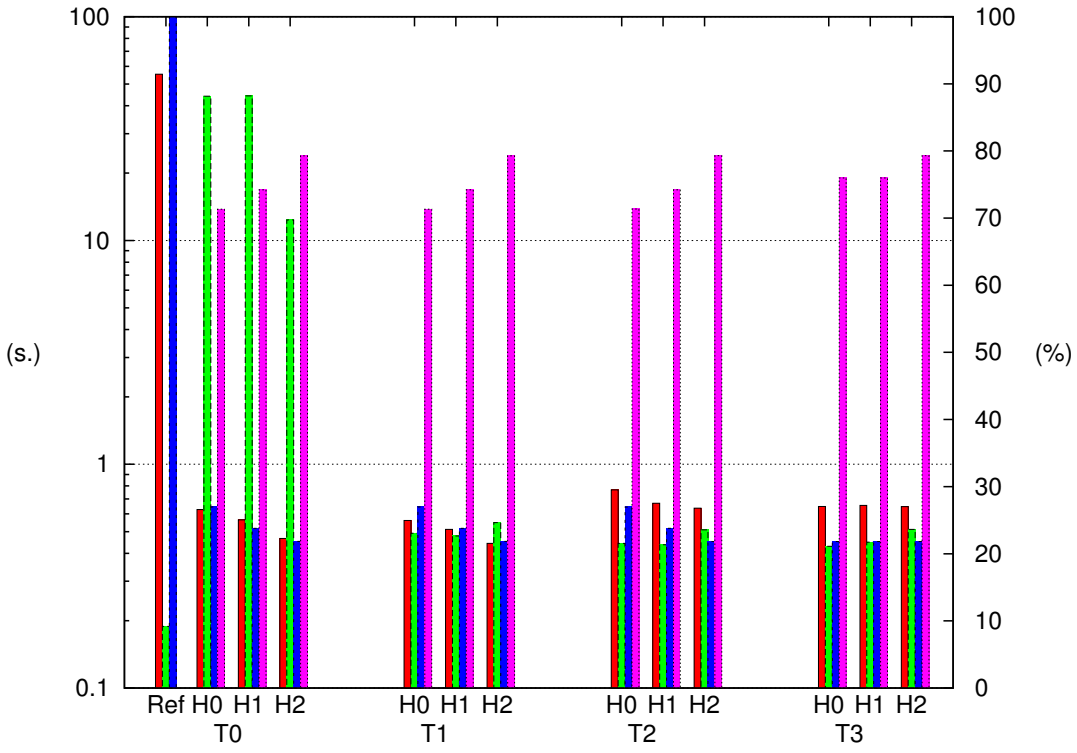

Figure 11: (Exp. 3) Number of states: 10 000 000, breadth: 2

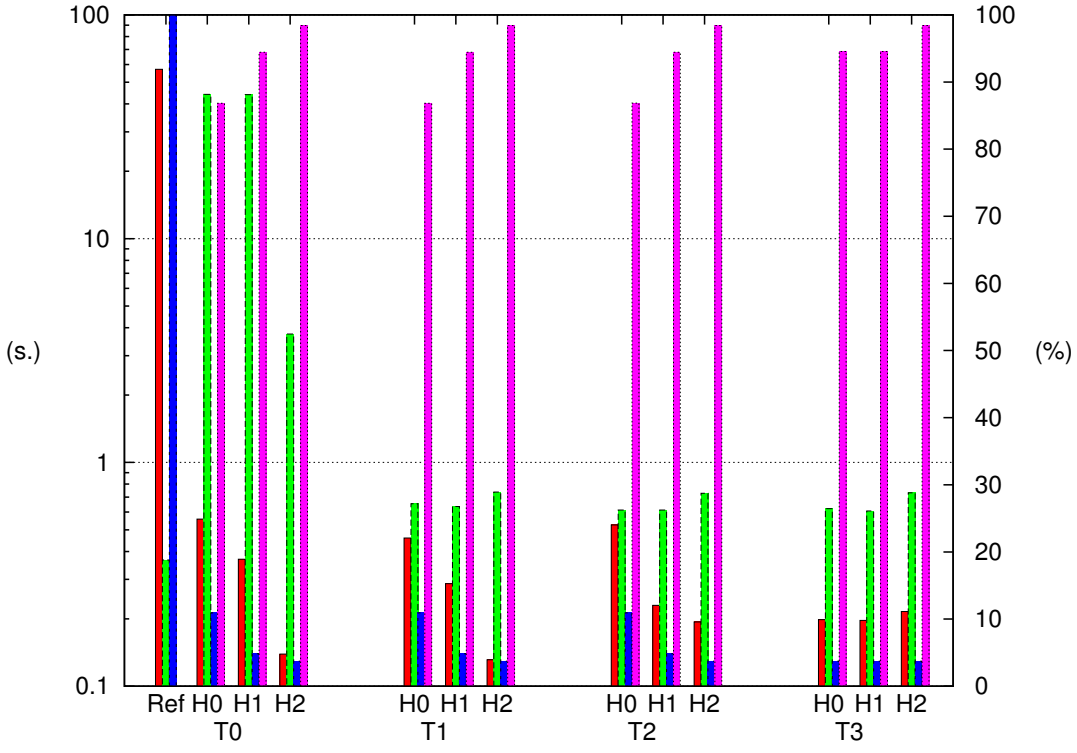

Figure 12: (Exp. 3) Number of states: 10 000 000, breadth: 20

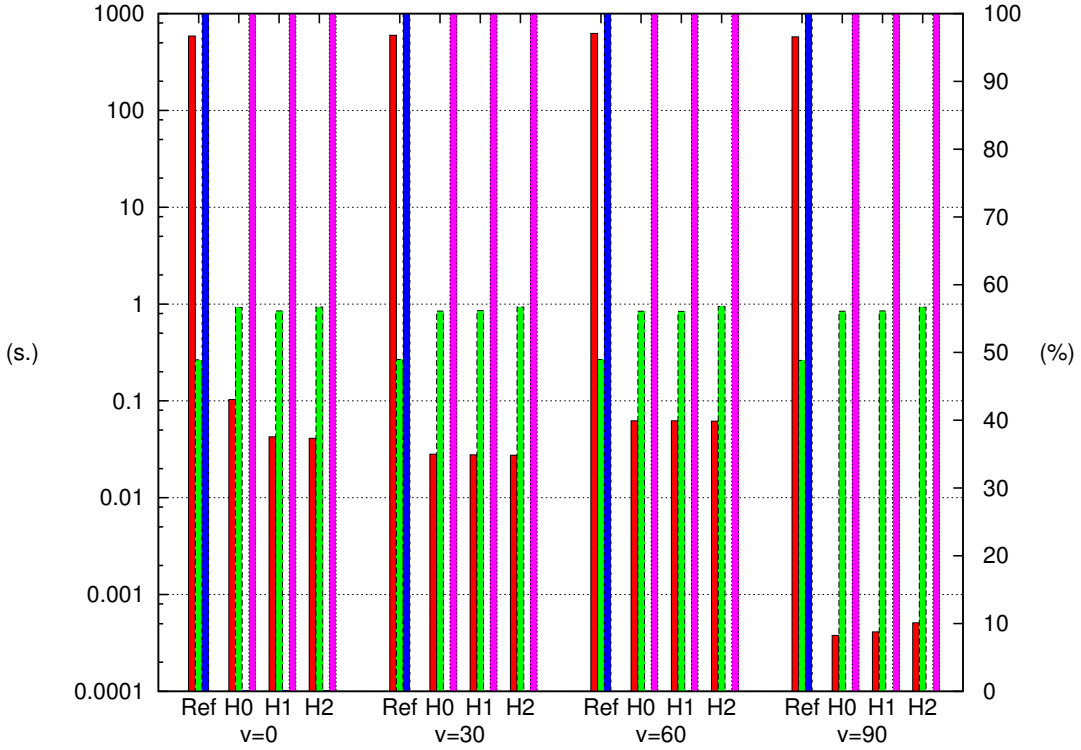

Figure 13: (Exp. 3) Number of states: 10 000 000, breadth: 50

And additional rule is that even if the update (Rem, s, t) respects the distance restriction, it is only kept if it respects $\mathrm{src}(s) \neq t$ *or* $\mathrm{low}(s)$ respects the distance limitation. By doing this the percentage will also represent the minimal avoidance percentage of the patches.

In this experiment, the graph contains 10 000 000 states and maximaly 20 000 000 transitions. Ten patches are applies each of which contains at most 500 000 updates.

The fourth experiment shows how badly the algorithm can behave, even when the patches are following a profitable pattern. In Figure 14, when d reaches 90% no more recomputation is necessary *but* knowing this requires a computation that can be expensive and may in facttake more time than the basic exploration, as it can be seen in Figure 14 for $d = 60$ and $d = 90$.

But there is two observations which can be done that soften this bad result. For the most part, this inefficiency comes from testing whether one update breaks the accepting run or not. Because there is quite a lot of updates, testing whether a patch is impacting for H_1 and H_2 becomes very costly. But, even though the prototype does optimize the test by using a binary search when possible, this may not be the only optimization pos-

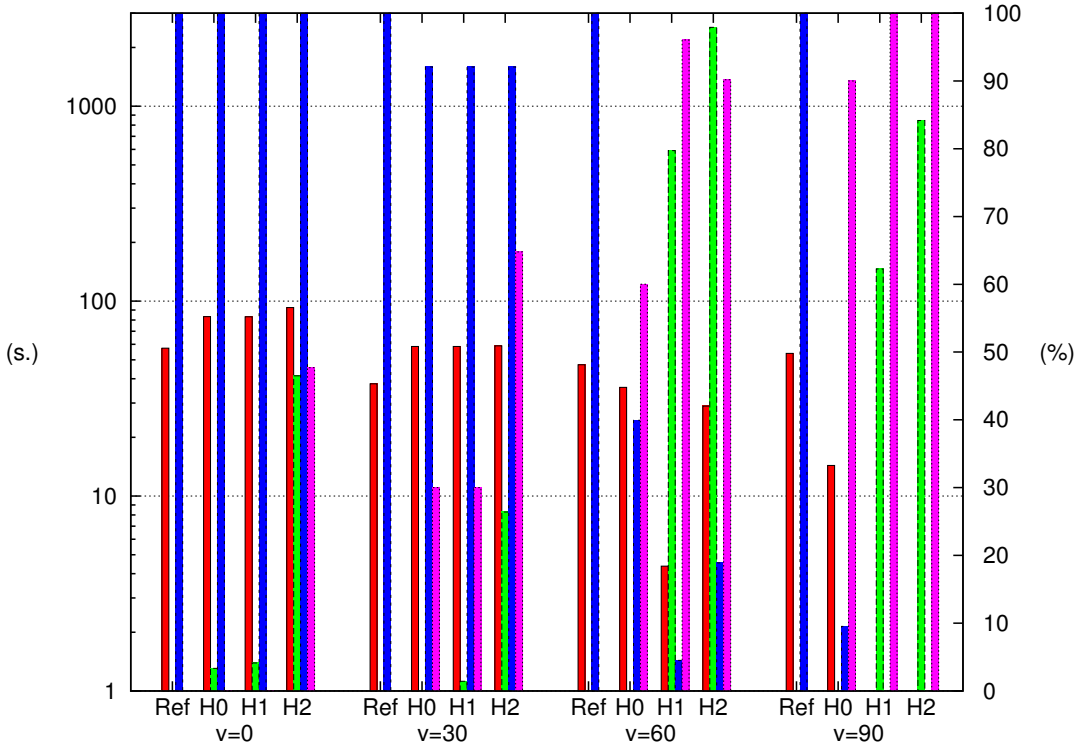

Figure 14: (Exp. 4) The v% closest state to s_0 cannot be used in a patch

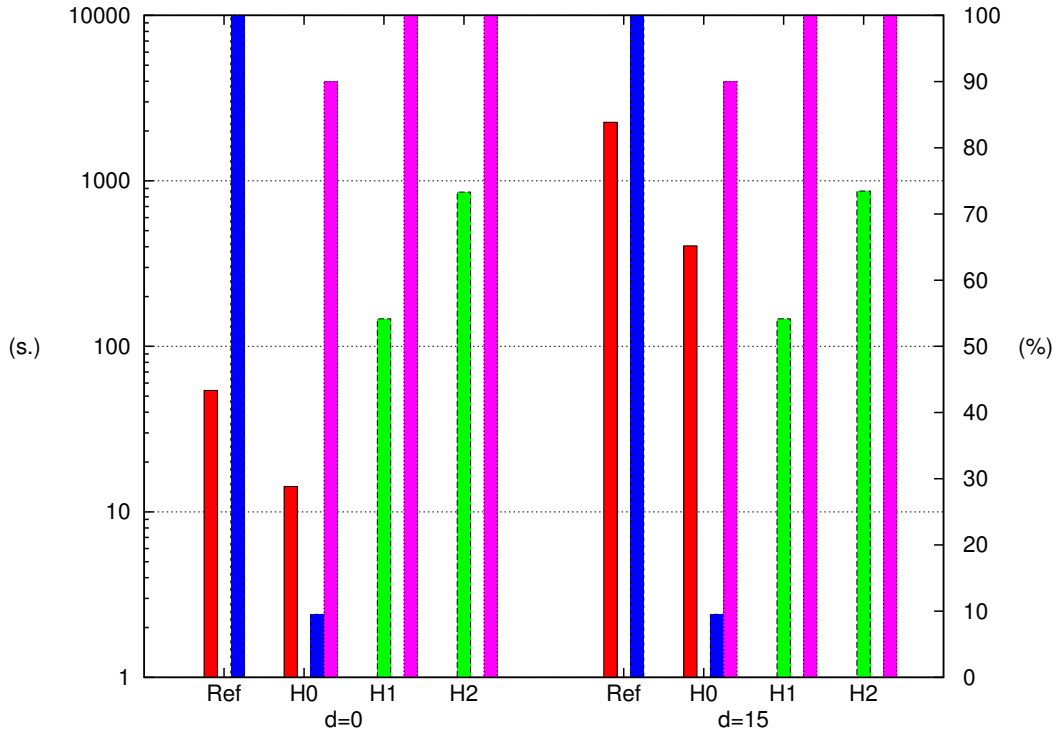

Figure 15: (Exp. 4) Experiment of Fig. 14 (v=90) with a delay d (in microseconds) added to the computation of the next state

sible. In fact, here parallellization of this test may yield very good results since the test is essentially a membership test.

Also, and *more importantly*, the prototype is written in such a way that retrieving the next state is only a matter of accessing two vectors in a sequence. Getting the next state is then very fast, which is highly improbable in the context of on-the-fly model-checking for example.

The results depicted in Figure 15, are a demonstration of how adding a delay of 15 microseconds to the computation of the next state compensates the computation overhead of H_1 and H_2 .

Fifth experiment. This experiment is a slight modification of the previous one to show a favorable case. Compared to the previous experiment, the number of accepting state has been augmented to 100 making is slightly easier to find an accepting run. The number of update per patch was reduced to 5 000 while the patch number was augmented to 1 000 (this means that the total number of updates is maximaly the same).

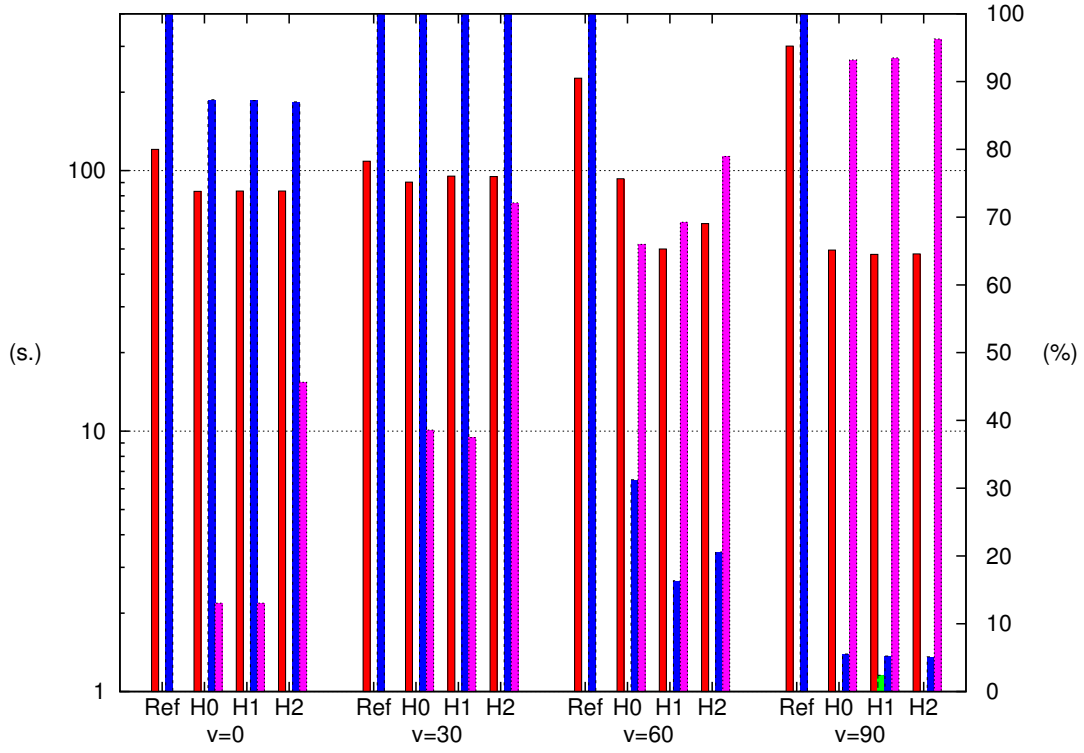

Figure 16: (Exp. 5) Less updates per patch than Exp. 4

In the end, in this experiment we can see that the update number is no longer a problem (the backtracking taking less than one second is not shown) and the re-computation

avoidance is good enough to have a speed-up of more than 5 in $d = 60$ and $d = 90$. Also notice the high avoidance rate, that is very benefical in situations where the computation of one state is slow.

Remark. One important remark is that we did not use the ancestor check ($s \prec^+ t$) because it turned out to be quite unefficient in a lot of situations. Unless the computation of a successor state takes a lot of time, or if the graph depth is quite small, then this checking should be avoided.

As an example of how bad this can be, a similar computation takes places in the backtracking for T_0 and T_1 whose bad effect can be seen in Figure 7 for example.

9 Conclusion

This work presented a technique that can be used to maintain an accepting run after some updates in the underlying graph of an UBA. As expected, the technique can greatly improve the performance when the updates are in small number or adequately localized. On the contrary the performance is negligeable or even worse in some cases when the updates are too numerous. Overall, the presented technique performs best in situations in which computing a state takes a long time.

One of the major issue remaining with this approach is to link it to a model-checking methodology. Indeed, we considered the product as an input, whereas to be more useful for model-checking we still need some way to map modifications of the model or modifications of the property to modifications in the product.

References

[1] Oleg V. Sokolsky and Scott A. Smolka, *Incremental Model-Checking in the modal μ-calculus* In CAV'94, LNCS 818

[2] Harry Li and Shriram Krishnamurthi and Kathi Fisler, *Verifying Cross-cutting Features as open systems* In SIGSOFT Softw. Eng. Notes, vol. 27–6, 2002

[3] Liam Roditty and Uri Zwick *A fully dynamic reachability algorithm for directed graphs with an almost linear update time* In Proc. of ACM Symposium on Theory of Computing, 2004

[4] Valerie King *Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs* In Proc. of FOCS 1999, pages 81–91,

[5] Liam Roditty *A faster and simpler fully dynamic transitive closure* In Proc. of SODA 2003, pages 404–412,

[6] Christopher L. Conway and Kedar S. Namjoshi and Dennis Dams and Stephen A. Edwards, *Incremental Algorithms for Inter Procedural Analysis of Safety Properties* In proceeding of CAV'05, LNCS 3576.

[7] Jaco Geldenhuys and Antti Valmari, *More efficient on-the-fly LTL verification with Tarjan's algorithm* TCS 345, pp. 60–82 (2005)

[8] Hana Chockler and Alexander Ivrii and Arie Matsliah and Shiri Moran and Ziv Nevo, *Incremental formal verification of hardware* In proceedings of FMCAD '11

[9] Krishnendu Chatterjee and Monika Henzinger *Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification* In proceedings of SODA '11

[10] Maxime Cordy and Pierre-Yves Schobbens and Patrick Heymans and Axel Legay *Towards an incremental automata-based approach for software product-line model checking* In proceedings of SPLC '12, vol. 2