On the Complexity of Monitoring Orchids Signatures, and Recurrence Equations*

Jean Goubault-Larrecq[1] and Jean-Philippe Lachance[2]
[1] LSV, ENS Paris-Saclay, CNRS, Université Paris-Saclay
ENS Paris-Saclay
61, avenue du président Wilson, 94230 Cachan Cedex, France
Tel.: +1-47402260
Fax: +1-47402464
goubault@lsv.fr
[2] Coveo Solutions, Inc., Québec City, QC G1W 2K7, Canada
jplachance@coveo.com

Abstract

Modern monitoring tools such as our intrusion detection tool Orchids work by firing new monitor instances dynamically. Given an Orchids signature (a.k.a. a rule, a specification), what is the complexity of checking that specification, that signature? In other words, let $f(n)$ be the maximum number of monitor instances that can be fired on a sequence of n events: we design an algorithm that decides whether $f(n)$ is asymptotically exponential or polynomial, and in the latter case returns an exponent d such that $f(n) = \Theta(n^d)$. Ultimately, the problem reduces to the following mathematical question, which may have other uses in other domains: given a system of recurrence equations described using the operators $+$ and max, and defining integer sequences u_n, what is the asymptotic behavior of u_n as n tends to infinity? We show that, under simple assumptions, u_n is either exponential or polynomial, and that this can be decided, and the exponent computed, using a simple modification of Tarjan's strongly connected components algorithm, in linear time.

1 Introduction

Orchids [13, 8] is an intrusion detection system. Given a trace σ of events, typically obtained in real-time, and a family of so-called signatures (variously otherwise called rules or specifications), Orchids tries to find a subsequence of σ that

*Partially funded by INRIA-DGA grant 12 81 0312 (2013-2016). The second author also thanks Hydro-Québec and Les Offices jeunesse internationaux du Québec (LOJIQ) for their financial support.

satisfies one of the signatures. Each signature is described as an automaton—not a finite-state automaton, though: each state comes with a piece of code, in a simple but expressive imperative language, that is executed whenever control flow enters that state. Orchids then waits for an event matching one of the transitions going out of the state.

See Figure 1 for a slightly edited example of a signature that monitors legal user id (uid) and group id (gid) changes, and reports any system call done with an unexpected uid or gid (at state `alert`). Events are records with fields such as `.syscall` or `.euid` (we have slightly simplified the syntax), and variable names start with a dollar sign. Transitions are introduced by the **expect** keyword, so, e.g., the start state `init` has one outgoing transition, and `wait` has five. Informally, Orchids will wait in state `init` for an event whose `.syscall` field is equal to the predefined constant `SYS_clone`: that is the indication that some process called the Linux system call `clone()`, creating a new process. Once this is detected, Orchids will then go to state `newpid`, where it will first record the values of the field `.exit` in variable `$pid` (the process id of the created process), and similarly obtain the values of the user id and group id of the created process. Then it goes to state `wait`, which is the core monitoring loop: while in state `wait`, Orchids will wait, concurrently, for five possible conditions, described by so-called **expect** transitions, on future events. The first three are cases where process number `$pid` (as checked by the condition `.pid==$pid`, which checks that the current event is one that concerns process `$pid`) can legally change its user id or group id: either by calling `execve` on a executable with the setuid or setgid bit set, or by calling one of the functions `setresuid()` or `setresgid()`. The fourth **expect** transitions catches the case where process `$pid` exits, in which case Orchids will cease monitoring it. The last of the transitions of state `wait` is triggered whenever the process `$pid` executes an action with an effective user id `.euid` that is not the one we expected (in `$uid`), or with an unexpected effective group id. Orchids then goes to state `alert`, where an alert report is produced, and then ceases monitoring that process.

Since Orchids cannot predict which of the five **expect** transitions will be matched by a subsequent event (and in fact, since, in principle, an event might match several of those transitions), Orchids must monitor all five. To implement that, the Orchids engine simulates so-called *threads*, and *forks* a new thread for each pending transition, each new thread waiting for a matching event[1]. That is, on entering state `wait`, Orchids will create five threads.

This description of the working of Orchids is, of course, oversimplified, but is enough to explain the problem we attack in this paper: evaluating the *complexity* of detecting a subsequence that matches one of the signatures. A similar question occurs naturally in other modern monitors, such as JavaMOP [11] or the more recent RV-Monitor [12], where signatures are called specifications, and threads are called monitor instances. As the authors argue, and as our own ex-

[1]To dispel a possible misunderstanding, these threads are not system-level threads, rather pairs of a pending transition and an environment holding the values of variables, and which are handled and scheduled by the Orchids engine. Similarly, "forking" an Orchids thread simply means making a copy of the current thread and adding it to the Orchids thread queue.

```
rule pidtrack {
  state init {
    expect (.syscall ==
            SYS_clone)
      goto newpid;
  }
  state newpid! {
    $pid = .exit;
    $uid = .euid;
    $gid = .egid;
    goto wait;
  }
  state update_uid_gid! {
    $uid = .euid;
    $gid = .egid;
    goto wait;
  }
  state update_setuid! {
    case (.egid != $gid)
      goto alert;
    else
      goto update_uid_gid;
  }
  state update_setgid! {
    case (.euid != $uid)
      goto alert;
    else
      goto update_uid_gid;
  }
  state wait! {
    expect (.pid == $pid &&
            .syscall == SYS_execve &&
            (.uid != .euid ||
             .gid != .egid))
      goto update_uid_gid;
    expect (.pid == $pid &&
            .syscall == SYS_setresuid)
      goto update_setuid;
    expect (.pid == $pid &&
            .syscall == SYS_setresgid)
      goto update_setgid;
    expect (.pid == $pid &&
            .syscall == SYS_exit)
      goto end;
    expect (.pid == $pid &&
            (.euid != $uid ||
             .egid != $gid))
      goto alert;
  }

  state alert! { report(); }

  state end! { }
}
```

Figure 1: The pid tracker signature

perience confirms, the main function that has to be estimated is the *number of threads* that the engine may create after reading n events.

In the worst case, for a signature S, the Orchids algorithm may create a number of threads $f_S(n)$ that is exponential in n, and that would be untenable. For an intrusion detection system, that would be dangerous, too, as that would open the door to an easy denial-of-service attack on the system itself.

Experience with practical signatures S shows that $f_S(n)$ is most often a polynomial of low degree. The exponential worst case behavior just means that one could instead craft specific signatures S such that $f_S(n)$ would be exponential. Most signatures are not of this kind. But can we warn a signature writer of the complexity of his signatures? I.e., how does $f_S(n)$ vary as a function of S?

Our main contribution is the design, and proof, of a linear time algorithm that, given a signature S, computes the asymptotic behavior of $f_S(n)$ as n tends to $+\infty$. We shall see that $f_S(n)$ is either exponential or polynomial. In the second case, our algorithm computes the unique exponent d such that $f_S(n) = \Theta(n^d)$.

Our algorithm works in two phases. First, it computes a set of recurrence equations from S, defining sequences u_n, v_n, ..., indexed by $n \in \mathbb{N}$. Then, and this is the core of our work, it finds the asymptotic behaviors of those sequences. In the example of Figure 1, the recurrence equations are:

$$u_{n+1}^{\tau_{0\texttt{pidtrack}}} = u_n^{\tau_{0\texttt{pidtrack}}} + u_n^{\texttt{init}} \quad one_{n+1} = one_n \quad u_{n+1}^{\tau_0} = u_{n+1}^{\tau_1} = \ldots = u_{n+1}^{\tau_5} = one_n$$
$$u_n^{\texttt{init}} = u_n^{\tau_0} \qquad u_n^{\texttt{wait}} = u_n^{\tau_1} + \cdots + u_n^{\tau_5}$$
$$u_n^{\texttt{newpid}} = u_n^{\texttt{update_uid_gid}} = u_n^{\texttt{setuid}} = u_n^{\texttt{setgid}} = u_n^{\texttt{alert}} = u_n^{\texttt{end}} = one_n$$
$$r_n = \max(u_n^{\tau_{0\texttt{pidtrack}}}, u_n^{\texttt{init}}, u_n^{\texttt{newpid}}, u_n^{\texttt{update_uid_gid}}, u_n^{\texttt{setuid}}, u_n^{\texttt{wait}}, u_n^{\texttt{setgid}}, u_n^{\texttt{alert}}, u_n^{\texttt{end}})$$

with the initialization conditions $u_0^{\tau_{0\texttt{pidtrack}}} = one_0 = u_0^{\tau_0} = u_0^{\tau_1} = \ldots = u_0^{\tau_5} = 0$. (We shall explain how those are found in Section 6: to connect with the explanation there, τ_0 is the unique transition out of state `init`, and τ_1, ..., τ_5 are the five transitions out of state `wait`.) Of the above sequences, r_n is $f_{\texttt{pidtrack}}(n)$. Then, our algorithm determines that $r_n = \Theta(n)$, showing that rule `pidtrack` creates a number of threads that is linear in the number n of events read in the worst case.

Outline. We review some recent related work in Section 2, describe the Orchids algorithm in Section 3, and define our format of systems of recurrence equations in Section 4. Those systems will be definitions of the sequence $f_S(n)$ by induction on n, together with several other auxiliary sequences, such as $u_n^{\texttt{newpid}}$ or $u_n^{\tau_0}$ above. Any system of recurrence equations Σ in our format gives rise to a graph $G(\Sigma)$ which will be instrumental in our study, and which we define in Section 5. Translating Orchids signatures to systems of recurrence equations and their associated graph is described in Section 6. The real work begins in Section 7, where we shall examine the possible asymptotic behaviors of our sequences, by looking finely at the structure of the strongly connected components (scc) of $G(\Sigma)$, carefully distinguishing between trivial and non-trivial

sccs, and so-called cheap and expensive edges. The algorithm quickly follows in Section 8, as an easy adaptation of Tarjan's algorithm, which we describe in full. We also report on our implementation and the result it gives on the ten standard Orchids signatures. We conclude in Section 9.

2 Related Work

The question of evaluating the complexity of monitors at this level of detail does not seem to have been addressed already. Efficiency has always been an important subject in the field, and RV-Monitor [12] was recently advocated as a fast implementation of monitors, able to sustain a large number of monitor instances (a.k.a., our threads). This is backed by experimental evidence.

RV-Monitor's algorithm is data-driven. Given a specification with parameters x_1, x_2, ..., x_k, RV-Monitor organizes monitor instances inside an indexing tree, and if we agree to call N the maximal number of different values that parameters can take over an n event run, there can be at most N^k monitor instances at any given time. If we assume no fixed bound on N, it is however clear that $N = O(n)$, and that the RV-Monitor analogue of our function $f_S(n)$ above is polynomial in all cases (assuming the specification S fixed).

The Orchids algorithm is not data-driven, but trace-driven. That is, Orchids does not look merely for *values* of parameters that make a match, but for a *subsequence* of the input sequence. This is important for security. See Section 3.1 of [8] for a precise explanation: as we have argued there, Orchids needs to be able to sort matching subsequences (even with different sets of parameter values), so that only the smallest, lexicographically, is eventually reported: this is in most cases the most informative subsequence of events that characterizes a successful attack. Orchids signature matching is therefore necessarily more complex in general, and one can craft signatures that would make Orchids generate exponentially many threads. This is why the algorithm presented here is needed.

Efficiency is also one of the main concerns behind the MonPoly-Reg and MonPoly-Fin tools [4]. The signature language there, MFOTL, is a real-time logic. Each variable varies in a domain of at most N elements, and it is assumed that there is an upper bound m on the number of successive events with the same timestamp. Time complexity is always polynomial. By carefully reading Section 5 of [4], one sees that the polynomial degree is linear in the maximal number k of free variables in the monitored formula and in the number c of connectives of the formula. When $m = 1$ (i.e., event timestamps are strictly increasing) and there are no temporal future operators, the complexity is comparable with the $O(N^k)$ bound given for RV-Monitor: if $t(n)$ is the time to check one RV-Monitor instance, so that RV-Monitor takes time $O(N^k t(n))$, the MFOTL-based tools run in time $O(N^{O(k+c)})$.

We will not cite any other paper on the question of monitor complexity. Other (parametric) monitors such as QEA or LogFire are described in [10], with some features in common with Orchids and RV-Monitor respectively. Extra

information can be gleaned by following references from the above papers.

Later, we will argue that our problem reduces to finding asymptotic estimates for sequences $(u_n)_{n\in\mathbb{N}}$ defined by so-called *recurrence equations*. Those are (systems of) equations of the form $u_{n+1} = f(u_n, v_n, w_n, \cdots)$, $v_{n+1} = g(u_n, v_n, w_n, \cdots)$, etc., where f, g, ..., are some explicitly given functions. There is a huge body of literature, specially in the mathematical literature, on those objects. One of the most relevant source is Flajolet and Sedgwick's book on analytic combinatorics [6]. Unfortunately, we shall need to deal with recurrence equations where u_{n+1} depend on u_n, v_n, w_n, etc., by using both the + and max operations. The latter seems to be out of scope of what is known in analytic combinatorics.

Estimating the asymptotics of recurrence equations defined using + and max can also be done using the spectral theory of max-plus algebras, see [1]. However, this only handles *linear* max-plus equations, i.e., equations of the form $u_{n+1} = \max(a_{uu} + u_n, a_{uv} + v_n, a_{uw} + w_n, \cdots)$, where a_{uu}, a_{uv}, a_{uw}, ..., are constants. (In max-plus algebra, max takes the rôle of addition and + of multiplication. Hence the latter equation is the max-plus equivalent of $a_{uu}u_n + a_{uv}v_n + a_{uw}w_n + \cdots$) We will allow equations of the form $u_{n+1} = \max(3 + 2u_n + v_n, 1 + u_n + w_n)$, for example, which are not linear, and might be called *max-plus polynomial equations*. (This example is the max-plus equivalent of $u_{n+1} = a^3u_n^2v_n + au_nw_n$, for some unspecified constant a.)

Complexity analysis, and specifically of forms close to our own work, is not limited to monitors. A whole part of the literature is devoted to static analysis of the time and space taken by programs. See [7] for a recent paper on the topic, based on finding upper approximations of solutions to so-called *cost equations* [2]. Other papers can be found by following references. Probably the piece of work closest to ours is due to Brockschmidt *et al.* [5], who propose a static analysis framework to infer polynomial upper bounds on the time taken by programs, and (at the same time) on the size of integer values. Both the purpose (complexity evaluation) and the tools of that paper are similar to ours—notably the role of sccs—and this is uncanny: the quantities they evaluate seem to bear no relationship with those we are interested in. Recall that we are interested in evaluating the asymptotic behavior of functions $f_S(n)$ where $n \in \mathbb{N}$ is some measure of time (number of events received in our case), and which are naturally defined by induction on n. Instead, Brockschmidt *et al.* compute an over-approximation $\mathcal{R}(t)(\vec{m})$ of the number of times a given transition t can be taken in an execution trace that starts with inputs of sizes bounded by the vector of sizes $\vec{m}$, and to that end they also compute an over-approximation $\mathcal{S}(t, v)(\vec{m})$ of the size of variable v after transition t has just been taken, starting with inputs of sizes bounded by $\vec{m}$. That is a rather different approach.

As a final note, we have paid special attention to dealing with the issue at the right level of generality. Rather than reporting on a perhaps clever, but highly specialized algorithm devoted to the complexity analysis of a specific tool (Orchids), we identify the core of the problem as finding the asymptotic behavior of sequences defined by recurrence equations. This allows us to study the latter in a proper, well-defined mathematical way, giving not only upper

bounds but also matching lower bounds, and a simple self-contained algorithm that computes them, in linear time. Because it is done at the right level of generality, we believe that the present work may have other uses beyond Orchids.

3 Orchids

Let us have a quick look at how Orchids creates and handles threads (a.k.a., monitor instances). We shall ignore most optimizations, both algorithmic and implementation-related, at least those that do not change the worst-case behavior of Orchids.

3.1 The History of Orchids

Orchids evolved from previous attempts at building intrusion detection systems from a model-checking approach, starting from [16], which presents two approaches. The second approach was a forerunner of the Orchids tool, which was presented in 2005 [13], and whose algorithm and optimizations were described in 2008 [8]. That second approach corrected a few flaws from the first approach. That same first approach (not the second one) is covered by a patent [15], which does not cover Orchids: the main claims of that patent require a means of generating propositional Horn clauses from formulae in a temporal logic for each new event read. The Orchids algorithm is not based on any such mechanism.

We would like to take the 2008 paper [8] as a reference to the Orchids algorithm. However, we need to explain it in a different light in the next subsections, so as to make the complexity analysis clearer.

3.2 A High-Level View

An Orchids signature consists in finitely many *states*. We will implicitly refer to Figure 1 to illustrate the notions. One of those states, `init`, is the initial state. Each state starts with an optional piece of code (for instance, `$pid` = `.exit`; `$uid` = `.euid`; `$gid` = `.egid`; in state `newpid`), which gets executed on entering the state. That code can contain elementary computations, tests, but no loops: we consider that piece of code irrelevant as far as complexity is concerned.

The second part of the description of a state defines its outgoing transitions, and comes in two flavors. We may either see a block of **expect** clauses, labeled with conditions that must be satisfied to launch the transition, as in state `wait`; or a **case**-delimited multi-way conditional, as in state `update_setuid` for example (or as in the degenerate case of state `update_uid_gid`, where there is just one branch, hence no **case** keyword).

The latter kind of state has an obvious semantics. For example, a thread entering state `update_setuid` will compare the field `.egid` with the value of

the variable `$gid`[2], and branch to state `alert` if they are different (a system call was made with a group id that is not what it was expected to be), or to `update_uid_gid` otherwise—such transitions were called *ϵ-transitions* in [8].

The former kind of state will *wait* for a subsequent event matching one of the **expect** clauses. A same event may match several **expect** clauses at once, and accordingly Orchids will fork as many threads as needed. We shall ignore the semantics of the tests performed by those clauses, and therefore a state with 5 **expect** clauses such as `wait` may fork 5 new threads.

Orchids works slightly differently. Mainly, Orchids threads wait on events, not when they enter a state, rather when they reach an **expect** clause. That may seem surprising, but look at it this way: **expect** clauses are the only *quiescent* places in an Orchids signature, namely the only places where execution does not proceed by itself.

Additional differences occur because of the '!' marker next to the state names, and of the `NO_WAIT` flag that the Orchids rule compiler uses to decorate transitions, and which is used to avoid forking certain threads while leaving the semantics unchanged. Setting that flag or not is based on the outcome of static analyses, as described in [8]. We will briefly explain those points below.

3.3 The Orchids Algorithm

Let us give an informal description of how the Orchids algorithm works. We avoid giving a formal description, which would be lengthy and of little interest, and are content with a description that allows us to compute the recurrence equations of Section 6.

Orchids maintains a queue of threads. Each thread θ is a tuple (τ, ρ, n) where τ is an **expect** transition usually out of some state q, ρ is an environment, binding variables to values, and n is a natural number, the *thread id*. The latter allows us to consider *thread groups*, namely the collection of threads that have the same thread id. One of the things Orchids will do, at specific points, is *committing* to a thread: this consists in killing (i.e., removing from the queue) all the threads in the same thread group as the current thread θ, except θ itself.

Committing is done when an Orchids thread enters a state with the '!' marker—so-called *commit states*. One can see that as a generalization of the final states considered in previous papers on Orchids. In particular, final states are now coded as commit states with no outgoing transition, such as state `end` or state `alert` in Figure 1. But it is also interesting to have intermediate commit states, such as states where we can be certain that an attack has been detected, but we wish to proceed with extra actions, typically to analyze what the attacker does once the attack has succeeded (forensics). In general, a commit state q can be seen as a state that is both final (killing all threads in q's thread group) and initial (restarting a thread, by going to state q—we shall define what "going to" means below).

[2]Variables are thread-local: if an Orchids thread modifies one of its variables, this does not affect any other thread.

Any monitoring algorithm needs to delete threads. Apart from the commit mechanism, this is implemented in Orchids by states with no outgoing transition, such as `end` or state `alert`. As the following description will make clear, any Orchids thread that enters such a state will just execute the code written there, then be deleted.

Abstracting away from details, we may describe the Orchids algorithm as follows. For each signature S, we assume an additional transition τ_{0S} outside of the signature. Notionally, this transition has the following description:

```
expect (1) goto init;
```

meaning that it will wait for an arbitrary event (1 is always true), and go to the initial state `init` of S. What is not covered by this description is that going to `init` will be accompanied with the creation of a new thread group, by changing the thread id to a fresh value. That is unimportant for our purposes here.

The Orchids algorithm starts out by creating one thread waiting on τ_{0S}, and with an empty environment. This is used to populate the initial thread queue.

For each new event read e, the Orchids algorithm sweeps through the threads (τ, ρ, n) in the thread queue. Ordering is important in order to implement the shortest run semantics of [8], and as can be inferred from loc. cit., not an entirely trivial matter. For the purpose of complexity analysis, this is unimportant here again, and we shall ignore this point.

For each thread (τ, ρ, n) in the thread queue, if the Boolean condition b labeling the **expect** clause τ is satisfied by e, then Orchids *goes to* the target state of transition τ (relatively to ρ, n), possibly inserting new threads into the thread queue. The process of going to a state is described below. If additionally τ has the `NO_WAIT` flag set, then the original thread (τ, ρ, n) is removed from the queue; otherwise it is kept. If b is not satisfied, then the thread is kept, unchanged.

This `NO_WAIT` flag deserves an explanation: a transition of the form **expect** (b) **goto** q will wait for a later event that satisfies condition b, then go to state q. Imagine b is satisfied at the current event, but also 7 events from now. We must spawn one thread for each of the two possibilities: perhaps the second case will detect a successful run, while the first one will not. This default mechanism is implemented by keeping the original thread (τ, ρ, n) in the queue, so as to detect such a second case. However, there are situations where one can prove that any such second case will either never lead to a successful run, or will lead to one, but one which will never be shortest. In those situations, which can be detected by an appropriate static analyzer (see [8] again, and replace the notion of final states there by the new notion of commit state), the Orchids compiler flags the transition with `NO_WAIT`, instructing Orchids to remove (τ, ρ, n) from the queue at run-time. Then Orchids will simply not look for a second case as above, but that is all right: no shortest run can be found this way, and the shortest run semantics is therefore preserved. In Section 8.2, we shall see an example where this trick decreases the complexity of detection from exponential to polynomial.

We promised we would explain what *going to* a state q' (relatively to ρ, n) would mean. If q' is a commit state, then kill all the threads of the same group,

i.e., remove all threads with the same n component from the thread queue, except the current thread. That being done, if q' is a state of the form:

⟨code⟩
case (b_1) **goto** q_1;
else case (b_2) **goto** q_2;
...
else case (b_{k-1}) **goto** q_{k-1};
else goto q_k;

then run ⟨code⟩ starting from environment ρ, producing a new environment ρ', then check the Boolean conditions b_1, b_2, ..., in turn in environment ρ' (all Boolean conditions are constrained by the type-checker to be side-effect free), until one matches, say b_i, or until the final **else** clause is reached (in which case we agree that $i = k$). Then the Orchids algorithm *goes to* state q_i (relatively to ρ', n), recursively. The process must eventually stop: Orchids rejects all signatures that contain cycles of `goto`s at compile-time; the check is done by a simple graph reachability algorithm.

If q' is a state of the form:

⟨code⟩
expect (b_1) **goto** q_1;
expect (b_2) **goto** q_2;
...
expect (b_k) **goto** q_k;

then run ⟨code⟩ starting from environment ρ, producing a new environment ρ'. Now do *not* check any of the conditions b_1, b_2, ..., b_k, corresponding any of the **expect** transitions τ_1, τ_2, ..., τ_k out of q'. Instead, create new threads (τ_i, ρ', n) for each i, $1 \leq i \leq k$, and insert them into the thread queue.

In particular, when $k = 0$ (look at states `end` and `alert` in Figure 1), no thread at all will be enqueued, and this is the way threads are eventually deleted.

4 Systems of Recurrence Equations

A *sequence* is an infinite family of natural numbers $(u_n)_{n\in\mathbb{N}}$ indexed by natural numbers. We say that a property P holds of u_n *for n large enough* if and only if there is an $n_0 \in \mathbb{N}$ such that P holds of u_n for every $n \geq n_0$. For a function $f\colon \mathbb{N} \to \mathbb{R}$, $u_n = \Theta(f(n))$ means that there are two real constants $m, M > 0$ such that, for n large enough, $mf(n) \leq u_n \leq Mf(n)$. If only the left-hand inequality is assumed, then we write $u_n = \Omega(f(n))$, and if only the right-hand inequality is assumed, then we write $u_n = O(f(n))$.

We shall say that $(u_n)_{n\in\mathbb{N}}$ has *exponential behavior* if and only if $u_n = \Omega(a^n)$ for some constant $a > 1$. It has *polynomial behavior* if and only if $u_n = \Theta(n^k)$ for some constant $k \in \mathbb{N}$.

Let $Q = \{u, v, \cdots\}$ be a finite non-empty set of symbols. Each symbol $u \in Q$ is meant to denote a sequence $(u_n)_{n\in\mathbb{N}}$ of natural numbers. A *system of recurrence equations* Σ for Q is, at least informally:

- an initial condition of the form $u_0 = a_u$, where $a_u \in \mathbb{N} \setminus \{0\}$, one for each $u \in Q$;
- for each $u \in Q$, an equation that defines u_{n+1} in terms of the terms v_n, $v \in Q$, and natural number constants, using the operations max and $+$. Semantically, since max distributes over $+$, this means defining u_{n+1} as $\max_{i=1}^{m_u}(\sum_{v \in Q} a_{uiv} v_n + b_{ui})$, where a_{uiv} and b_{ui} are natural number constants. For reasons explained below, we require $m_u \neq 0$, and for each u and i, either $b_{ui} \neq 0$ or $a_{uiv} \neq 0$ for some $v \in Q$.

We shall use a slightly different *formal* definition below (Definition 4.1), and that will be the only authoritative definition for all our mathematical developments, notably in all proofs. We use the above definition for now to illustrate our goals and some inevitable issues, as well as to vindicate the upcoming formal definition itself.

Sticking to the above definition for now, Σ defines a unique family of sequences $(u_n)_{n \in \mathbb{N}}$, one for each $u \in Q$, in the obvious way. Our purpose is to show that one can decide, in linear time, which of these sequences have exponential behavior, and which have polynomial behavior; in the latter case, our algorithm will return a natural number d such that $u_n = \Theta(n^d)$. Note that this will imply that $(u_n)_{n \in \mathbb{N}}$ has either exponential or polynomial behavior, nothing else—e.g., not logarithmic, $\Theta(2^{\sqrt{n}})$ or $\Theta(n^{\log n})$ for example.

Example 4.1 *Consider $Q = \{u\}$, the system $u_0 = 1$, $u_{n+1} = 2u_n$ is a system of recurrence equations; it defines a unique sequence $u_n = 2^n$, which has exponential behavior.*

Example 4.2 *Instead, consider $Q = \{u, v, w\}$ and the system $u_0 = 1$, $v_0 = 1$, $w_0 = 1$, $u_{n+1} = v_n + 1$, $v_{n+1} = u_n + w_n$, $w_{n+1} = w_n + 2$. Its unique solution is given by $w_n = 2n + 1$, $v_n = \frac{1}{2}n^2 + n + 1$ if n is even, $v_n = \frac{1}{2}n^2 + n + \frac{1}{2}$ if n is odd, $u_n = \frac{1}{2}n^2 + 1$ if n is even, $u_n = \frac{1}{2}n^2 + \frac{3}{2}$ if n is odd. In that case, $w_n = \Theta(n)$, $u_n = \Theta(n^2)$, $v_n = \Theta(n^2)$ all have polynomial behavior.*

Notice the slightly oscillating behavior of $(u_n)_{n \in \mathbb{N}}$ and $(v_n)_{n \in \mathbb{N}}$. Although those sequences have polynomial behavior, we cannot find an actual, unique polynomial $p(n)$ such that $u_n = p(n)$ for every $n \in \mathbb{N}$.

Our recurrence equations have a few constraints attached: a_u is non-zero, m_u is non-zero, and either $b_{ui} \neq 0$ or $a_{uiv} \neq 0$ for some $v \in Q$. This will be the case in all applications. Without this condition, the behaviors of the corresponding sequences might be much wilder, as exemplified below.

Example 4.3 *Consider $Q = \{u, v\}$ with $u_0 = 0$, $v_0 = 1$, $u_{n+1} = 2v_n$, $v_{n+1} = u_n$. This is not a system of recurrence equations in our sense, because the initial value a_u for u is equal to 0. Its unique solution is $u_n = 0$ if n is even, $2^{(n+1)/2}$ if n is odd; $v_n = 2^{n/2}$ if n is even, 0 if n is odd. Note that $(u_n)_{n \in \mathbb{N}}$ exhibits neither polynomial nor exponential behavior, as it oscillates between the two. Such a system is forbidden by our definition.*

We claimed that, for every vertex $u \in Q$, $(u_n)_{n\in\mathbb{N}}$ would either have exponential or polynomial behavior, and that in the latter case, we would be able to find a degree $d \in \mathbb{N}$ such that $u_n = \Theta(n^d)$. One may wonder whether it would be possible to refine this, and to also find a coefficient a such that $u_n \sim an^d$ (meaning that $u/(an^d)$ would tend to 1 as n tends to $+\infty$). This is not possible, as the following example shows.

Example 4.4 *Let $Q = \{u, v, s, t\}$ with $u_0 = 1$, $v_0 = 2$, $s_0 = 1$, $t_0 = 1$, $u_{n+1} = v_n$, $v_{n+1} = u_n$, $s_{n+1} = t_n$, $t_{n+1} = s_n + u_n$. Its unique solution is: $u_n = 1$ if n is even, 2 if n is odd; $v_n = 2$ if n is even, 1 if n is odd; $s_n = n/2 + 1$ if n is even, n if n is odd; $t_n = n + 1$ if n is even, $(n + 3)/2$ if n is odd. Note that both s and t exhibit polynomial behavior, as they are $\Theta(n)$, but we cannot find an a such that $s_n \sim an$ or $t_n \sim an$: for example, s_n oscillates between $n/2 + 1$ and n.*

Let us give a formal definition of systems of recurrence equations. One might do this in the obvious way, using Q and families of numbers a_{uiv} and b_{ui}. However, we would also like some equations such as

$$u_{n+1} = u_n + \max(v_n, \max(u_n, w_n + 2) + \max(2u_n, w_n)) \tag{1}$$

where the max and + operators are freely mixed. Distributing max over + would produce an equivalent system of the right shape, but this transformation takes exponential time and space in the worst case.

Instead, we use the following folklore transform, which works in linear time, at the expense of introducing new symbols to Q (clearly, only linearly many more). For each non-variable proper subexpression of the term on the right (here, $u_n+\max(v_n, \max(u_n, w_n+2)+\max(2u_n, w_n))$), we introduce a fresh symbol. By *non-variable* we mean any subexpression except the non-constant leaves (here, u_n, v_n, w_n); this includes all non-leaf expressions, such as $\max(u_n, w_n+2)$, and all constant leaves, such as 2. Let us do so on (1). There are seven non-variable proper subexpressions there, and we create seven fresh symbols, call them a, b, c, d, e, f and two. The sequence two_n is meant to be the constant sequence equal to 2, and is defined by $two_0 = 2$, $two_{n+1} = two_n$. The sequence a_n denotes $w_n + 2$, b_n denotes $\max(u_n, w_n + 2)$, c_n denotes $2u_n$, d_n denotes $\max(2u_n, w_n)$, e_n denotes $\max(u_n, w_n + 2) + \max(2u_n, w_n)$, and f_n denotes $\max(v_n, \max(u_n, w_n + 2) + \max(2u_n, w_n))$. Accordingly, we replace (1) by the following eight equations:

$$\begin{array}{ccc} u_{n+1} = u_n + f_n & f_n = \max(v_n, e_n) & e_n = b_n + d_n \\ d_n = \max(c_n, w_n) & c_n = 2u_n & b_n = \max(u_n, a_n) \\ a_n = w_n + two_n & two_{n+1} = two_n & \end{array}$$

plus the initial condition $two_0 = 2$.

Doing so only requires us to be able to state two kinds of recurrence equations: equations of the form $u_{n+k} = \max(v_n, w_n, \cdots)$, and equations of the form $u_{n+k} = \sum_{v\in Q} a_{uv} v_n$, obeying some natural conditions. (Note that constants b_{ui} have disappeared in the process, being replaced by fresh symbols, such as two in the above example.) This leads us to the following, formal, definition.

Definition 4.1 (System of Recurrence Equations) *Let Q be some set of so-called symbols.*

A recurrence equation *on Q is:*

1. *either an equation of the form $u_{n+k} = \max(v_n, w_n, \cdots)$, for some non-empty subset of symbols $v, w, \cdots \in Q$,*
2. *or an equation of the form $u_{n+k} = \sum_{v \in Q} a_{uv} v_n$, where at least one a_{uv}, $v \in Q$, is non-zero,*

and where in each case $k = k_u$ is equal to 0 or 1.

A system of recurrence equations Σ *on the set of symbols Q is a Q-indexed family of* recurrence equations E_u*, plus initial conditions $u_0 = a_u$ for each $u \in Q$ such that $k_u \neq 0$, where $a_u \in \mathbb{N} \setminus \{0\}$ is a constant.*

We represent equations $u_{n+k} = \sum_{v \in Q} a_{uv} v_n$ in sparse form, that is, as a list of pairs (v, a_{uv}) for each $v \in Q$ such that $a_{uv} \neq 0$.

Our formal definition includes strictly more systems than our previous, informal definition. Systems defined per our previous definition always have exactly one solution; in other words, for each $u \in Q$, they define exactly one sequence $(u_n)_{n \in \mathbb{N}}$. A contrario, our new definition allows for systems of the form $u_n = u_n$, which have infinitely many solutions; or of the form $u_n = u_n + one_n$, $one_{n+1} = one_n$, $one_0 = 1$, which have no solution. We repair this shortly.

5 Graphs

Given a system Σ of recurrence equations on the set of symbols Q, let us define its *graph* $G(\Sigma)$ as follows. We write $s \to t$ to say there is an edge from s to t. $G(\Sigma)$ is a labelled directed graph, and both vertices and edges receive labels. Its vertices are the elements of Q, and are split in two kinds, corresponding to the two kinds of allowed equations:

1. the *max* vertices u are those whose associated equation E_u is of the form $u_{n+k} = \max(v_n, w_n, \cdots)$; there is one edge from u to v, one from u to w, and so on; u itself is labeled with k, and the edges receive label 1;
2. the *plus* vertices u are those whose associated equation E_u is of the form $u_{n+k} = \sum_{v \in Q} a_{uv} v_n$; there is one edge from u to each $v \in Q$ such that $a_{uv} \neq 0$, and it is labeled with a_{uv}; the vertex u itself is labeled with k;
3. there is no other edge.

Introducing auxiliary symbols as necessary, Example 4.1 is really the system:

$$u_{n+1} = 2u_n \qquad u_0 = 1$$

Its graph is shown on the top left of Figure 2. We distinguish the plus vertices by showing them on a light grey background. We also distinguish the vertex

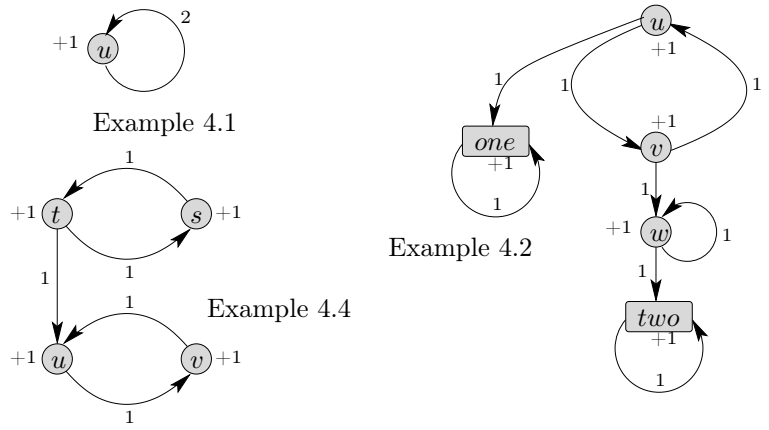


Figure 2: Three examples of graphs $G(\Sigma)$

labels by writing them with a plus sign, viz., +1, not 1. The right-hand graph is that of the system of Example 4.2, put into the adequate form:

$$\begin{array}{lllll} u_{n+1}{=}v_n + one_n & u_0{=}1 & v_{n+1}{=}u_n + w_n & v_0{=}1 & w_{n+1}{=}w_n + two_n \\ one_{n+1}{=}one_n & one_0{=}1 & two_{n+1}{=}two_n & two_0{=}2 & w_0{=}1 \end{array}$$

Similarly for the graph of Example 4.4, shown at the bottom left.

Using the graph $G(\Sigma)$, we evacuate the problem of those systems Σ that have non-unique solutions, or no solution: we say that Σ is *well-formed* if and only if there is no cycle in the graph that goes only through vertices labeled +0.

Proposition 5.1 *Every well-formed system Σ has a unique solution, consisting of uniquely-defined sequences $(u_n)_{n\in\mathbb{N}}$ for each $u \in Q$, which satisfy all the equations in Σ.*

Proof. Define a relation $\succ$ by $u \succ v$ if and only if there is a non-empty path from u to v in $G(\Sigma)$ whose vertices, including u but excluding v, are labeled +0. Write $v \prec u$ for $u \succ v$. Since Σ is well-formed, $\prec$ is irreflexive, hence defines a well-founded strict ordering. Then, u_n is defined by Σ, by induction on the pair $(n, u) \in \mathbb{N} \times Q$, ordered by the lexicographic product $(< \times \prec)_{\text{lex}}$ of the usual ordering $<$ on $\mathbb{N}$ and of $\prec$. Since the latter is well-founded, u_n is defined uniquely. □

In proofs to come, we shall use the relation $\succ$ several times, and each time this will be to do well-founded inductions along $(< \times \prec)_{\text{lex}}$. In all the examples shown in Figure 2, $\succ$ is the trivial relation ($u \succ v$ is always false), and such inductions boil down to ordinary inductions on $n \in \mathbb{N}$.

Note that $G(\Sigma)$ has no provision for specifying initial conditions such as $u_0 = 1$. They are not needed for Proposition 5.1. They will be useless in subsequent developments as well: the asymptotic behavior of u_n will be independent of u_0, provided $u_0 \neq 0$.

6 Generating Recurrence Equations from Orchids Signatures

Let us describe how we generate the recurrence equations that define the sequences we are interested in from a given Orchids signature. The translation takes linear time in the size of the signature, defined as the sum of the number of its states and its transitions.

We first define the set Q of symbols, that is, the names u of sequences $(u_n)_{n\in\mathbb{N}}$: r for the final complexity of signature S, u^q for each state q, u^τ for each **expect** transition τ (including τ_{0S}), *one* for a special sequence defining the constant 1. For the latter, generate the recurrence equations:

$$one_{n+1} \quad = \quad one_n \tag{2}$$

The initialization condition ($one_0 = 1$) is irrelevant to the asymptotic behavior of sequences. In fact, they are discarded when we construct the graph of our system of recurrence equations. We shall therefore omit them.

The *meaning* of symbol u^τ is as follows. Imagine the Orchids algorithm is launched on a single thread (τ, ρ, id) (with the same τ as in u^τ), and feed it n events. Now count how many threads have been created by Orchids once it has read and processed those n events: u^τ_n is designed so as to be a precise upper bound on that number. The symbol u^q has a similar meaning, starting from state q instead of transition τ.

In each state, for each transition τ, say with target state q', we create the following recurrence equations (3)–(8). If q' is not a commit state, and if τ has the `NO_WAIT` flag set, then we generate:

$$u^\tau_{n+1} \quad = \quad u^{q'}_n \tag{3}$$

Indeed, given $n+1$ events, Orchids will simply read the first one, trigger the transition and go to state q', where n events will remain to be read.

If q' is not a commit state and if τ does not have the `NO_WAIT` flag set, then we generate instead:

$$u^\tau_{n+1} \quad = \quad u^\tau_n + u^{q'}_n \tag{4}$$

witnessing the fact that we also keep the current thread, waiting on transition τ, so that all threads descending from τ count (and are accounted through the term u^τ_n).

If q' is a commit state, then, as we have said earlier, we can consider q' to be a final state, and accordingly we write:

$$u^\tau_{n+1} \quad = \quad one_n \tag{5}$$

if τ has the `NO_WAIT` flag, and:

$$u^\tau_{n+1} \quad = \quad u^\tau_n + one_n \tag{6}$$

otherwise. We also need to consider q' as an initial state, and this will be taken care of in Equation (9) below.

For each state q with k outgoing **expect** transitions τ_1, τ_2, ..., τ_k, we generate:

$$u_n^q \quad = \quad u_n^{\tau_1} + u_n^{\tau_2} + \cdots + u_n^{\tau_k} \tag{7}$$

For each state q with outgoing **case** transitions, enumerate the target states that are not commit states as q_1, q_2, ..., q_k, and generate:

$$u_n^q \quad = \quad \max(one_n, u_n^{q_1}, u_n^{q_2}, \cdots, u_n^{q_k}) \tag{8}$$

Indeed all commit states only contribute one thread, hence the maximum with one_n.

Finally, we write:

$$r_n \quad = \quad \max(u_n^{\tau_{0S}}, u_n^{q_1}, u_n^{q_2}, \cdots, u_n^{q_m}) \tag{9}$$

where q_1, q_2, ..., q_m are all the commit states of S. This is meant to take into account the fact that each commit state is not only a final state, but also an initial state, and may therefore contribute to r_n.

Equation (9) is the only equation presented here whose soundness is not obvious. Indeed, what we have is that the maximum number of threads s_n created by signature S is the maximum of $u_n^{\tau_{0S}}$, and of quantities $u_{n-j}^{q_i}$, for $1 \leq i \leq m$, and j varying over $0 \cdots n$: j is, intuitively, the event number at which state q_i is entered. We shall see that the asymptotic behavior of those sequences are either exponential or polynomial. For i fixed, $u_{n-j}^{q_i}$ is an $\Omega(a^{n-j})$ for some $a > 1$, or a $\Theta((n-j)^d)$ for some degree $d \in \mathbb{N}$, as we have seen. When j varies, a^{n-j} (resp., $(n-j)^d$) reaches its maximum when j is minimal. We see that the minimal value of j is the length of the shortest series of transitions leading from state `init` to state q_i in S, which is a constant. Since $a^{n-j} = \Theta(a^n)$ and $(n-j)^d = \Theta(n^d)$, s_n is, in any case, a Θ of r_n, as described in (9).

As a final note to this section, observe that all the recurrence equations described here are well-formed in the sense of Proposition 5.1.

The reader is invited to practice, to generate the equations corresponding to the signature of Figure 1, and, finally, to check that those are exactly those given at the end of the introduction. To this end, one needs to know that the Orchids compiler manages to set the `NO_WAIT` flag on every transition.

We shall give more details, on practical examples, in Section 8.2. The equations we have just described are an upper bound on the actual complexities we are interested in, and a natural question is whether they incur an acceptable loss of precision, compared to the complexities experienced in practice. This is a legitimate concern, and perhaps the most pressing source of worry is as follows. Monitors not only create, but also remove threads (monitor instances) at run-time, whereas our equations only seem to upper-bound the number of created threads, ignoring deletions. This would be understanding our algorithm wrongly. The main mechanism Orchids uses to delete threads is entering committed states. In that case, recall that all threads in the current thread group

except the current thread are killed. Dealing with that case is the purpose of equation (5), (6), and (9). Our complexity evaluation mechanism simulates that bulk removal by considering that all threads of the same group are removed in one go—equations (5), (6)—and that one is born again, as described in equation (9). Hence that particular form of deletion is, in fact, correctly handled. We shall discuss precision issues in more detail Section 8.2, resting on the examples given there.

7 Sccs, and asymptotics

We shall see that the key to understanding the asymptotic behavior of sequences defined by a well-formed system of recurrence equations Σ lies in the strongly connected components of the graph $G(\Sigma)$, introduced in Section 5.

We fix a well-formed system Σ of recurrence equations for the rest of the section, as well as its set of symbols Q, and the unique sequences $(u_n)_{n\in\mathbb{N}}$ that it defines. The following trivial lemma is crucial.

Lemma 7.1 *Assume two vertices u, v in Q such that v is reachable from u, namely, such that there is a path from u to v in $G(\Sigma)$. There is a constant $k \in \mathbb{N}$ such that, for every $n \in \mathbb{N}$, $u_{n+k} \geq v_n$.*

More precisely, k can be taken as the sum of vertex labels on any given path from u to v, including u but excluding v.

Proof. The key argument is that for every edge of the form $s \to t$, where s is labeled a, $s_{n+a} \geq t_n$ for every $n \in \mathbb{N}$. This holds whether s is a max or a plus vertex. □

Before we start doing any real proof, we must mention that the pathological case of a behavior that is neither polynomial nor exponential (see Example 4.3) will not happen. This will rest on the following casual-looking observation, which we shall need in the proof of Proposition 7.5.

Lemma 7.2 *For every $u \in Q$, for every $n \in \mathbb{N}$, $u_n \geq 1$.*

Proof. By well-founded induction on (n, u) along $(< \times \prec)_{\text{lex}}$.

If u is a max vertex labeled $+0$, then $u_n = \max(v_n, w_n, \cdots)$ where v, w, ..., form a non-empty subset of Q. Note that $u \succ v, w, \cdots$ By induction hypothesis, $v_n \geq 1$, $w_n \geq 1$, and so on. Since they form a non-empty subset, $\max(v_n, w_n, \cdots) \geq 1$, so $u_n \geq 1$.

If u is a max vertex labeled $+1$, then either $n = 0$ and $u_0 = a_u \geq 1$, or $u_n = \max(v_{n-1}, w_{n-1}, \cdots) \geq 1$ by a similar argument as above (but using the fact that $n > n - 1$, not $u \succ v, w, \cdots$ as a reason to invoke the induction hypothesis).

If u is a plus vertex labeled $+0$, then $u_n = \sum_{v\in Q} a_{uv} v_n$, and some a_{uv} is non-zero. For this v, $u \succ v$, so by induction hypothesis $v_n \geq 1$. We now have $u_n \geq a_{uv} v_n \geq 1$.

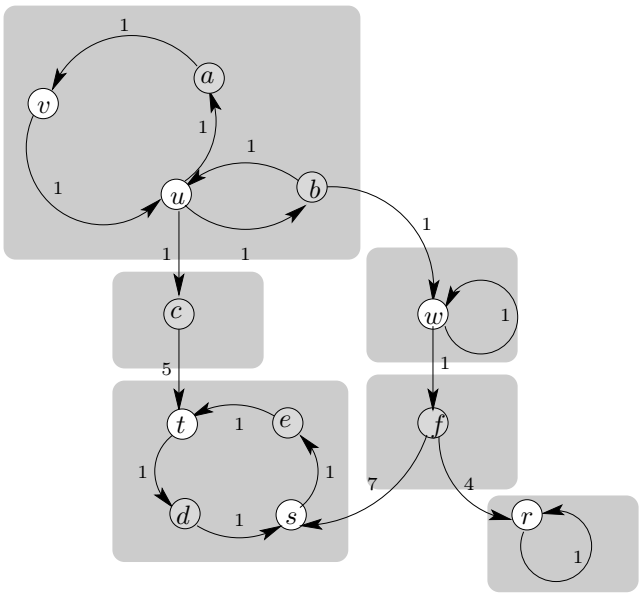


Figure 3: Sccs in a graph $G(\Sigma)$

If u is a plus vertex labeled $+1$, then either $n = 0$ and $u_0 = a_u \geq 1$, or $u_n = \sum_{v \in Q} a_{uv} v_{n-1} \geq 1$, by a similar argument as above. □

A subset A of vertices is *strongly connected* if and only if every vertex from A is reachable from any other vertex of A. The maximal strongly connected subsets are called the *strongly connected components* of the graph, and we abbreviate that as *scc*. They partition the graph, and in particular every vertex u belongs to a unique scc, which we write $scc(u)$.

On Figure 2, the top left graph is its unique scc. The right-hand graph has four sccs, the top cycle $\{u, v\}$, the middle cycle $\{w\}$, and the two cycles $\{one\}$ and $\{two\}$. The bottom left graph has two sccs, the top cycle $\{s, t\}$, and the bottom cycle $\{u, v\}$.

In general, an scc can be more complex than a mere cycle, but if one wants to picture a non-trivial scc, a cycle is a good first approximation. Sccs can also be *trivial*, i.e., consist of no cycle at all, just a single vertex with no self-loop. Figure 3 displays a more complex graph, with sccs shown as darker gray rectangles. (We haven't shown any vertex labels on this example, as they would distract us somehow. You can put any vertex labels so long as no cycle goes only through vertices labeled $+0$, for example by labeling each vertex $+1$.) The topmost scc is an example of an scc that is not just a cycle. There are also trivial sccs: $\{c\}$ and $\{f\}$. We let the reader reconstruct a system of recurrence equations associated with this graph. Note that, contrarily to previous examples, this one involves the max operator in order to define $(u_n)_{n\in\mathbb{N}}$, $(v_n)_{n\in\mathbb{N}}$, $(s_n)_{n\in\mathbb{N}}$, $(t_n)_{n\in\mathbb{N}}$, $(w_n)_{n\in\mathbb{N}}$, $(r_n)_{n\in\mathbb{N}}$. Although this may seem like a complicated graph,

it will follow from our algorithm that $u_n = \Theta(n)$, $a_n = \Theta(n)$, $b_n = \Theta(n)$ and $v_n = \Theta(n)$, and all other vertices have constant behavior.

Definition 7.3 (Bad Vertex) *Call a vertex $u \in Q$* bad *if and only if it is a plus vertex, and given its associated equation $u_{n+k} = \sum_{v \in Q} a_{uv} v_n$, the sum of the coefficients a_{uv} where v ranges over $scc(u)$ is at least 2.*

Equivalently, u is bad if and only if it is a plus vertex, and at least one of the following possibilities occurs:

1. there is an edge (u, v) of label at least 2 to a vertex v in the same scc as u,
2. or there are at least two edges $u \to v$ and $u \to w$ to vertices v and w that are both in the same scc as u.

In Example 4.1, u is bad. There is no bad vertex in Example 4.2 or in Example 4.4. There is no bad vertex either in Figure 3: although there are several edges with label at least 2, they all go out of their start scc.

Definition 7.4 (Bad* Vertex) *Say that a vertex $u \in Q$ is* bad* *if and only if some bad vertex is reachable from it, namely if and only if there is a path from u to some bad vertex v.*

We shall see that the bad* vertices u are exactly those such that $(u_n)_{n \in \mathbb{N}}$ has exponential behavior. In Example 4.1, u is bad*. There is no bad* vertex in Example 4.2 or in Example 4.4, or in Figure 3.

Proposition 7.5 *For every bad* vertex u in Q, $(u_n)_{n \in \mathbb{N}}$ has exponential behavior.*

Proof. We shall show that this is the case if u is bad. If u is bad*, then there is a path from u to some bad vertex v, so that $(v_n)_{n \in \mathbb{N}}$ will have exponential behavior. By Lemma 7.1, for some $k \in \mathbb{N}$, for every $n \in \mathbb{N}$, $u_{n+k} \geq v_n$, so that $(u_n)_{n \in \mathbb{N}}$ will also have exponential behavior.

So let us assume that u is bad. In particular, u is a plus vertex, and we consider two cases.

Case 1. If there is an edge $u \to v$ with label a_{uv} at least 2 for some v in the same scc as u, then there is also a path π from v to u. Concatenating π with the edge $u \to v$, we obtain a cycle $u \to v \xrightarrow{\pi} u$. Let a be the label of u, and k be the sum of the vertex labels on π, including v but excluding u. Since Σ is well-formed, $k + a \geq 1$. By the defining equation for u, $u_{n+k+a} \geq a_{uv} v_{n+k} \geq 2v_{n+k}$, and by Lemma 7.1, $v_{n+k} \geq u_n$, for every $n \in \mathbb{N}$. By induction on p, it follows that $u_{(k+a)p+q} \geq 2^p u_q$ for all $p, q \in \mathbb{N}$, and since $u_0, u_1, \ldots, u_{k+a-1} \geq 1$ (Lemma 7.2), $u_{(k+a)p+q} \geq 2^p$ for all $p, q \in \mathbb{N}$. This implies that $u_n \geq ma^n$ for every $n \in \mathbb{N}$, where $a = \sqrt[k+a]{2}$ and $m = 1/a^{k+a-1}$. Hence $(u_n)_{n \in \mathbb{N}}$ has exponential behavior in this case.

Case 2. If instead there are at least two edges $u \to v$ and $u \to w$ to vertices v and w that are both in the same scc as u, then there are two cycles $u \to v \xrightarrow{\pi_1} u$

and $u \to w \xrightarrow{\pi_2} u$. Let k_1 be the sum of the vertex labels on π_1, including v but excluding u, and similarly for k_2 and π_2. Let also a be the vertex weight of u. Since Σ is well-formed, $k_1 + a$ and $k_2 + a$ are both larger than or equal to 1. Using Lemma 7.1, we obtain that for every $n \in \mathbb{N}$, $v_{n+k_1+a} \geq u_{n+a} \geq v_n$ and $w_{n+k_2+a} \geq u_{n+a} \geq w_n$. If k_1 and k_2 were equal, we could reuse the same argument as in Case 1... but we have to work a bit more.

By induction on p_1, it follows that $v_{n+p_1(k_1+a)} \geq v_n$ and similarly, $w_{n+p_2(k_2+a)} \geq w_n$, for all p_1 and p_2 in $\mathbb{N}$. Pick p_1 and p_2 so that $(p_1+1)(k_1+a) = (p_2+1)(k_2+a)$ (by taking least common multiples), and let k be equal to the latter minus a. Then $v_{n+k} = v_{n+p_1(k_1+a)+k_1} \geq v_{n+k_1}$; by Lemma 7.1 applied to π_1, $v_{n+k_1} \geq u_n$, so $v_{n+k} \geq u_n$. Similarly, $w_{n+k} \geq u_n$. By the defining equation of u, $u_{n+k+a} \geq a_{uv}v_{n+k} + a_{uw}w_{n+k} \geq v_{n+k} + w_{n+k} \geq 2u_n$. We obtain the same inequality $u_{n+k+a} \geq 2u_n$ as in Case 1. It follows that $(u_n)_{n\in\mathbb{N}}$ has exponential behavior again, by the same reasoning. □

If an scc contains a bad* vertex, then all its vertices are bad*. Let us consider the case of sccs A without any bad* vertex. We shall illustrate the various cases we need to consider on the graph shown in Figure 3. The idea of our algorithm is that we shall iterate on all sccs A, from bottom to top, deducing a characteristic degree d_A such that $u_n = \Theta(n^{d_A})$ for every vertex u in A from the characteristic degrees of sccs below A.

We first deal with the case of *trivial* sccs, i.e., sccs with only one vertex and no self-loop. By abuse of language, say that u is a trivial scc iff $\{u\}$ is.

Proposition 7.6 *Assume $u \in Q$ is a trivial scc, and that for every edge $u \to v$, the sequence $(v_n)_{n\in\mathbb{N}}$ has polynomial behavior, viz., for some $d_v \in \mathbb{N}$, $v_n = \Theta(n^{d_v})$. Then $(u_n)_{n\in\mathbb{N}}$ has polynomial behavior, too, and $u_n = \Theta(n^{d_u})$, where $d_u = \max\{d_v \mid v \in Q \text{ such that } u \to v\}$.*

Proof. This follows from the fact that $\max(\Theta(n^{d_v}), \Theta(n^{d_w}), \cdots) = \Theta(n^{\max(d_v, d_w, \cdots)})$ for max vertices, and that $\sum_{v\in Q \text{ such that } u\to v} a_{uv}\Theta(n^{d_v}) = \Theta(n^{\max\{d_v \mid u\to v\}})$ for plus vertices, using the fact that $a_{uv} \geq 1$ for every $v \in Q$ such that $u \to v$. □

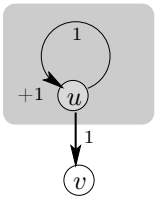
Now assume A is a non-trivial scc without any bad vertex. Such an scc must have a special shape, exemplified on the three sccs of the graph on the right of Figure 2, or on the bottom left graph of the same figure, or on the non-trivial sccs of Figure 3: the weights of all edges between vertices of A must be equal to 1, and every plus vertex has exactly one successor in A (all others are outside A). That it has at most one successor in A (and that all its edge labels are equal to 1) is a consequence of the absence of bad vertices. That it has at least one follows from the fact that A is non-trivial.

Note that plus vertices in A may have more than one successor; but only one can be in A. For example, v has two successors in the graph of Example 4.2 (Figure 2, right), but only u is in the scc $A = \{u, v\}$ that v belongs to. Example 4.4 displays a similar situation. Figure 3 does, too: b has two successors, but only one in its own scc.

Definition 7.7 (Expensive, cheap edges) *Say that an edge $u \to v$* goes out of *A if and only if $u \in A$ and $v \notin A$. If u is a plus vertex, then we say that it is an* expensive *edge out of A, otherwise it is a* cheap *edge out of A.*

In Figure 3, there is only one expensive edge out of a non-trivial scc, namely the edge $b \to w$, with label 1. There are two other edges going out of non-trivial sccs, namely $u \to c$ and $w \to f$. They are both cheap. Note that being cheap or expensive is entirely independent of the label it carries. Here the expensive edge has the lowest possible label of the whole graph.

The key argument in Proposition 7.10 below consists in noting that $\sum_{j=0}^{n} j^N = \Theta(n^{N+1})$ when n tends to $+\infty$—a well-known identity. The way we use that is probably best explained on a small test case. Imagine one of the simplest possible non-trivial sccs: a one-vertex loop, as shown on the right, and assume that we know that $v_n = \Theta(n^N)$, and even, to make things simpler, that $v_n = n^N$.



The equation defining $(u_n)_{n\in\mathbb{N}}$ is $u_{n+1} = u_n + v_n = u_n + n^N$. It follows easily that $u_n = (n-1)^N + (n-2)^N + \cdots + 1^N + 0^N + u_0 = u_0 + \sum_{j=0}^{n-1} j^N$, and that is $\Theta(n^{N+1})$, as we have just seen. Note that the edge $u \to v$ is expensive. The name "expensive" was chosen so as to suggest that those are the edges that are responsible for the increase in the exponent, from N to $N+1$. Cheap edges will incur no such increase.

In general, the proof of the bounds in the following proposition uses a similar argument. The case of non-trivial sccs A without any bad vertex, and with only + vertices (no max vertex) is easier to argue. There is a single elementary cycle from u back to u, and all its weights are equal to 1; all edges out of this path go out of A. Semantically u_{n+a} can be written as u_{n-k} (for some constant k) plus values v_n, for v outside of A. If v_n has polynomial growth, viz. if $v_n = \Theta(n^{d_v})$, then n^{d_v} will enter a summation defining u_{n+a}, and a similar argument as above shows that u_n will grow as a polynomial in n whose degree is the largest exponent d_v plus 1. The fact that + and max vertices can be used freely inside A makes the proof of the Proposition 7.10 below slightly more involved.

The key argument in Proposition 7.10 consists in the following well-known bounds. We take the standard convention that $j^N = 1$ when $N = 0$, even when $j = 0$. The proof is a standard high school exercise.

Lemma 7.8 *Let N be any non-negative real constant. For every $p \in \mathbb{N}$,*

$$\frac{1}{N+1} p^{N+1} \leq \sum_{j=0}^{p} j^N \leq \frac{1}{N+1} (p+1)^{N+1}.$$

Proof. If $N > 0$, by elementary analysis, $\sum_{j=0}^{p} j^N \leq \int_0^{p+1} x^N dx = \frac{1}{N+1}(p+1)^{N+1}$, and $\sum_{j=0}^{p} j^N = \sum_{j=1}^{p} j^N \geq \int_0^p x^N dx = \frac{1}{N+1} p^{N+1}$. When $N = 0$, the claim reduces to the inequalities $p \leq p+1 \leq p+1$, which are obvious. □

We will use the left-hand inequality of Lemma 7.8 to prove lower bounds, but we will not use the right-hand inequality for upper bounds. Regarding the latter, we will find it easier to rely on the following inequality—one that can be used to reprove the right-hand inequality of Lemma 7.8, using a telescoping sum, by the way.

Lemma 7.9 *Let N be any non-zero natural number. For every $x \geq 0$, $x^N + Nx^{N-1} \leq (x+1)^N$.*

Proof. By the binomial formula, $(x+1)^N = \sum_{k=0}^{N} \binom{N}{k} x^k = x^N + \binom{N}{N-1} x^{N-1} + \binom{N}{N-2} x^{N-2} + \cdots \geq x^N + \binom{N}{N-1} x^{N-1}$ (since $N \geq 1$, the sum contains at least those first two terms). We conclude since $\binom{N}{N-1} = N$. □

Proposition 7.10 *Let A be a non-trivial scc of $G(\Sigma)$ without a bad vertex. For each edge $u \to v$ going out of A, assume that $(v_n)_{n\in\mathbb{N}}$ has polynomial behavior. Precisely, assume that $v_n = \Theta(n^{d_v})$.*

Then every vertex of A has polynomial behavior, with the same degree d_A, where d_A is the maximum of:

- *all the quantities d_v, where $u \to v$ ranges over the cheap edges out of A,*
- *and all the quantities $d_v + 1$, where $u \to v$ ranges over the expensive edges out of A.*

To be completely formal, we agree that the maximum of an empty set of numbers is 0; this is needed in case there is no edge going out of A at all.

Proof. We show that, for every $u \in A$, $u_n = \Omega(n^{d_A})$ and that for every $u \in A$, $u_n = O(n^{d_A})$, separately.

Lower bounds. Consider any edge going out of A. If this is a cheap edge $u \to v$, with u a max vertex labeled a, then $u_{n+a} \geq v_n$. By assumption, $v_n = \Omega(n^{d_v})$, hence $u_n = \Omega(n^{d_v})$. By Lemma 7.1, for every $t \in A$, there is a constant $k \in \mathbb{N}$ such that for every $n \in \mathbb{N}$, $t_{n+k} \geq u_n$, whence $t_n = \Omega(n^{d_v})$ as well.

If this is an expensive edge $u \to w$, then u is a plus vertex labeled a, and $u_{n+a} = \sum_{v\in Q} a_{uv} v_n$. Among the summands with $a_{uv} \neq 0$, one is obtained by choosing $v = w$, because of the existence of the edge $u \to w$; and exactly one is a vertex $v \in Q$ that is in A—recall the special shape of non-trivial sccs without a bad vertex. Since $v \in A$ and $w \notin A$, v and w are distinct. It follows that $u_{n+a} \geq v_n + w_n$ for every $n \in \mathbb{N}$. Since A is an scc, u is reachable from v. By Lemma 7.1, there is a $k \in \mathbb{N}$ such that for every $n \in \mathbb{N}$, $v_{n+k} \geq u_n$. Moreover, since Σ is well-formed, $k+a \geq 1$. We have obtained: (*a*) $u_{n+k+a} \geq u_n + w_{n+k}$ for every $n \in \mathbb{N}$. Since $w_n = \Omega(n^{d_w})$, there is a constant $n_0 \in \mathbb{N}$, and a real constant $m > 0$ such that, for every $n \geq n_0$, $w_n \geq m n^{d_w}$. Without loss of generality, we may assume $n_0 \geq k + a$. For n large enough, we may write $n - a$ uniquely as $p(k+a)+q$ for some $p \in \mathbb{N}$, $q < k+a \leq n_0$. Assuming n sufficiently large, p will be non-zero. Using (*a*), $u_n \geq w_{p(k+a)+q} + w_{(p-1)(k+a)+q} + \cdots + w_{j(k+a)+q} + \cdots + w_q \geq m \sum_{j=0}^{p} (j(k+a)+q)^{d_w}$. Since $q \geq 0$, $u_n \geq m(k+a)^{d_w} \sum_{j=0}^{p} j^{d_w}$, and this

is larger than or equal to $\frac{m(k+a)^{d_w}}{d_w+1}p^{d_w+1}$ by Lemma 7.8. Since $p \geq \frac{n-a}{k+a} - 1$, $u_n \geq \frac{m(k+a)^{d_w}}{d_w+1}\left(\frac{n-a}{k+a} - 1\right)^{d_w+1}$, and this is a $\Theta(n^{d_w+1})$, since m, k, a, and d_w are constants. Finally, as for cheap edges, we use Lemma 7.1 to conclude that $t_n = \Omega(n^{d_w+1})$ for every $t \in A$.

For each $t \in A$, we have therefore obtained that $t_n = \Omega(n^{d_w+1})$ for every expensive edge $u \to w$ out of A, and that $t_n = \Omega(n^{d_v})$ for every cheap edge $u \to v$ out of A. Since d_A is the largest of those exponents, $t_n = \Omega(n^{d_A})$.

Upper bounds. There is a constant $n_0 \in \mathbb{N}$, and constants $M_v > 0$, such that, for every $n \geq n_0$, for every edge $u \to v$ out of A, $v_n \leq M_v(n+1)^{d_v}$. (We take $n+1$ instead of n to avoid problems with the case $n = 0$.) Notice that we pick the same n_0 for all edges. That does not restrict generality, since we can always take the maximum of all values of n_0 collected for each edge.

It will be important to realize what the defining equation is for u_n, where u is a plus vertex in A. In principle, letting a be the label of u, this is $u_{n+a} = \sum_{v \in Q} a_{uv}v_n$. Since $a_{uv} \neq 0$ only when there is an edge $u \to v$, this is $u_{n+a} = \sum_{v \in Q/u \to v} a_{uv}v_n$. Since u cannot be bad by assumption, there is a unique $w \in Q$ such that $u \to w$ and for which w is also in A; moreover, $a_{uw} = 1$. (That this w exists is because A is a non-trivial scc.) It follows that the defining equation for such a vertex is of the form $u_{n+a} = w_n + T_n$, where T_n is the sum over all (expensive) edges $u \to v$ out of A of $a_{uv}v_n$.

We first deal with the case where $d_A = 0$. In that case, there cannot be any expensive edge out of A, and for every cheap edge $u \to v$ out of A, $d_v = 0$. Let B be the maximum of all the constants M_v given above, when $u \to v$ ranges over all the cheap edges out of A, and of all the values u_{n_0}, $u \in A$. We claim that, for every $u \in A$, for every $n \geq n_0$, $u_n \leq B(n+1)^{d_A}$, namely that $u_n \leq B$. This is by induction on (n, u) with $n \geq n_0$, ordered by $(< \times \prec)_{\text{lex}}$. For $n = n_0$, $u_{n_0} \leq B$ by the definition of B. For $n > n_0$, either u is a max node (labeled a, say), and $u_n = \max(v_{n-a}, w_{n-a}, \cdots) \leq B$ by induction hypothesis; or u is a plus node, but since there is no expensive edge out of A, the term T_n alluded to above is zero, so the defining equation for u_n is $u_n = w_{n-a}$ for some unique vertex w in A; by induction hypothesis again, this is at most B.

We now deal with the more complicated case $d_A \geq 1$. We deal with that case separately, because we shall need to refer to $d_A - 1$, which would not make sense if $d_A = 0$.

We shall show that $u_n \leq B(n+1)^{d_A} + h(u)(n+1)^{d_A-1}$ for every $n \geq n_0$ and $u \in A$. Here B and $h(u)$, for each $u \in A$, will be non-negative constants that are defined below. What they should be is best found by examining the proof below, and collecting the constraints that they need to obey.

Solving those constraints leads us to define $h(u)$ by induction along $\prec$ by:

1. if u is labeled $+1$, then $h(u) = 0$;
2. if u is a max vertex labeled $+0$, then $h(u) = \max\{h(v) \mid u \to v, v \in A\}$;
3. if u is a plus vertex labeled $+0$, then recall that the defining equation for u is $u_n = w_n + T_n$, where w is in A and $T_n = \sum_v a_{uv}v_n$, where

the sum is taken over all expensive edges $u \to v$ out of A: then we let $h(u) = h(w) + \sum_v a_{uv} M_v$.

And we require B to be so large that:

(a) $u_{n_0} \leq B(n_0+1)^{d_A} + h(u)(n_0+1)^{d_A-1}$ for every $u \in A$;

(b) $B \geq M_v$ for every cheap edge $u \to v$ out of A;

(c) for every plus vertex u labeled $+1$ in A, $h(w) + \sum_v a_{uv} M_v \leq d_A B$, where w is the unique vertex in A such that $u \to w$, and the sum is taken over all expensive edges $u \to v$ out of A;

(d) for every max vertex u labeled $+1$ in A, for every edge $u \to v$ with v in A, $h(v) \leq d_A B$.

Note indeed that all those inequalities are eventually satisfied as B tends to $+\infty$, (a) because $n_0 + 1 > 0$, (c) and (d) because $d_A > 0$.

We now prove that $u_n \leq B(n+1)^{d_A} + h(u)(n+1)^{d_A-1}$ for every $n \geq n_0$ by induction on (n, u) with $n \geq n_0$, ordered by $(< \times \prec)_{\text{lex}}$. The case $n = n_0$ is by (a).

Let us now assume $n > n_0$.

If u is a plus vertex, labeled a, then $u_n = w_{n-a} + T_{n-a}$ where $w \in A$ and $T_{n-a} = \sum_v a_{uv} v_{n-a}$, where the sum is taken over all expensive edges $u \to v$ out of A. By assumption, $v_{n-a} \leq M_v(n-a+1)^{d_v}$ for all the latter vertices v, hence $v_{n-a} \leq M_v(n-a+1)^{d_A-1}$. By induction hypothesis, $w_{n-a} \leq B(n-a+1)^{d_A} + h(w)(n-a+1)^{d_A-1}$. If the label a is $+0$, we obtain $u_n \leq B(n+1)^{d_A} + h(w)(n+1)^{d_A-1} + \sum_v a_{uv} M_v (n+1)^{d_A-1} = B(n+1)^{d_A} + h(u)(n+1)^{d_A-1}$ (item 3 of the definition of h). If the label a is $+1$, we obtain instead $u_n \leq Bn^{d_A} + h(w)n^{d_A-1} + \sum_v a_{uv} M_v n^{d_A-1} \leq Bn^{d_A} + d_A B n^{d_A-1}$, where the latter inequality is by (c); the last term is less than or equal to $B(n+1)^{d_A}$ by Lemma 7.9, and we therefore obtain $u_n \leq B(n+1)^{d_A} + h(u)(n+1)^{d_A-1}$ since $h(u)$ is non-negative. (In fact, $h(u) = 0$, see item 1 of the definition of h.)

If u is a max vertex, labeled a, then $u_n = \max(v_{n-a}, w_{n-a}, \cdots)$. The terms v_{n-a} obtained from (necessarily cheap) edges $u \to v$ out of A are less than or equal to $M_v(n-a+1)^{d_v} \leq M_v(n-a+1)^{d_A} \leq B(n-a+1)^{d_A}$ (by (b)) $\leq B(n-a+1)^{d_A} + h(u)(n-a+1)^{d_A-1}$, since $h(u)$ is non-negative. Whichever the value of a, $+0$ or $+1$, that is less than or equal to $B(n+1)^{d_A} + h(u)(n+1)^{d_A-1}$. The terms v_{n-a} such that $v \in A$ are less than or equal to $B(n-a+1)^{d_A} + h(v)(n-a+1)^{d_A-1}$ by induction hypothesis. If $a = +0$, then this is less than or equal to $B(n+1)^{d_A} + h(u)(n+1)^{d_A-1}$, using the fact that $h(v) \leq h(u)$ in this case (item 2 of the definition of h). If $a = +1$, then this is less than or equal to $Bn^{d_A} + h(v)n^{d_A-1} \leq Bn^{d_A} + d_A B n^{d_A-1}$ by (d). That is less than or equal to $B(n+1)^{d_A}$ by Lemma 7.9, hence to $B(n+1)^{d_A} + h(u)(n+1)^{d_A-1}$, again. It follows that $u_n \leq B(n+1)^{d_A} + h(u)(n+1)^{d_A-1}$. □

Let us use those results to determine the asymptotic behavior of all the sequences defined by the graph of Figure 3. Recall that there is no bad* vertex in that example. We start from the sccs at the bottom, and work our way up:

- The non-trivial sccs $\{r\}$ and $\{s, e, t, d\}$ have no outgoing edge at all, hence their associated sequences are $\Theta(n^0)$ (bounded from below and from above by constants).
- The scc $\{f\}$ is trivial, and its two successors behave as $\Theta(n^0)$, hence it itself behaves as $\Theta(n^0)$, by Proposition 7.6.
- Similarly for the trivial scc $\{c\}$.
- The scc $\{w\}$ is not trivial, but it does not have any expensive edge out of it; by Proposition 7.10, it also behaves as $\Theta(n^0)$.
- The topmost scc is non-trivial, it has one cheap outgoing edge, $u \to c$, and one expensive outgoing edge $b \to w$. By applying Proposition 7.10, all the sequences associated with vertices in that scc behave as $\Theta(n^{\max(0,0+1)})$, that is, $\Theta(n)$.

8 The Algorithm

To conclude, we need a final, standard ingredient: the *condensation* of a directed graph G is the graph whose vertices are the sccs of G, and such that there is an edge from A to B if and only if there are vertices $q \in A$ and $r \in B$ and an edge $q \to r$ in G. The condensation is always acyclic, meaning that working our way up, that is, from the leaves to the roots of the condensation, must terminate (hence terminate in a linear number of steps). In the case of a graph G of the form $G(\Sigma)$, we shall say that an edge $A \to B$ as above in the condensation is *expensive* if and only if A is non-trivial, and we can find a $q \in A$ and an $r \in B$ such that the edge $q \to r$ is expensive in G; it is *cheap* otherwise, namely when A is trivial, or A is non-trivial and all the edges $q \to r$ in G with $q \in A$ and $r \in B$ are cheap.

Theorem 8.1 *Given any system Σ of recurrence equations with set Q of symbols, we can compute a table of numbers $d_u \in \mathbb{N} \cup \{+\infty\}$, $u \in Q$, in linear time, such that $d_u = +\infty$ iff $(u_n)_{n\in\mathbb{N}}$ has exponential behavior, and otherwise $u_n = \Theta(n^{d_u})$.*

The algorithm works as follows:

1. Compute $G(\Sigma)$ and its sccs, building its condensation G'.
2. Traverse G' in reverse topological order (i.e., from the bottom up). For each visited scc A, decide whether A contains a bad vertex. If so, let $d_A := +\infty$. Otherwise, for every successor B of A in G', d_B has already been computed, and let $d_A := \max(\max_{A\to B \text{ cheap}} d_B, \max_{A\to B \text{ expensive}}(d_B + 1))$, where by convention we agree that the maximum of the empty set is zero.
3. Finally, for each $u \in Q$, let $d_u := d_{scc(u)}$.

```
function SCC(v:vertex)
    PUSH(v)
    for each successor w of v do
        if w.index is undefined then
            SCC(w)
            if w.low < v.low then
                v.low ← w.low
            else if w.onStack then
                if w.index < v.low then
                    v.low ← w.index
                end if
            end if
        else
        end if
    end for
    if v.low = v.index then
                ▷ pop the scc with root v from the stack
        l ← S                    ▷ sever scc from rest of stack
        S ← v.next               ▷ l=linked list of nodes in scc
        v.next ← NULL                    ▷ last node in l is v
        w ← l                          ▷ now loop over scc l
        while w ≠ NULL do
            w.onStack ← false
            w.sccRoot ← v
            w ← w.next
        end while
        computeComplexities(l)
    end if
end function
```

```
function PUSH(v:vertex)
    v.index ← index
    v.low ← index
    index ← index + 1
    v.next ← S
    v.onStack ← true
    S ← v
end function
```

Figure 4: Tarjan's algorithm

The correctness of the algorithm is a direct consequence of Proposition 7.5, Proposition 7.6, and Proposition 7.10. That it works in linear time is easy. Notably, the second phase sweeps through all the sccs A once, and for each, takes time proportional to the number of vertices in A plus the number of edges that go out of A. The sum over all sccs A of those values is the size of $G(\Sigma)$.

In practice, this is implemented by simply modifying Tarjan's scc algorithm [17]. In any description of that algorithm, there is a single line of code where it has just found an scc A, and it must emit it by repeatedly popping a stack. It is enough to compute d_A there, by the formula given in Theorem 8.1, item 2, knowing that at that point, all the values d_B will have been computed earlier.

8.1 A Concrete Implementation

We make the algorithm more explicit. Figure 4 displays one possible implementation of Tarjan's algorithm. (Comments start with ▷.) That algorithm works with the help of a global natural number `index`, initialized to 0, and a global stack `S`, which we implement as a linked list of vertices `S`, `S.next`, `S.next.next`, and so on until we reach the special constant `NULL`, denoting the end of the list. We assume that each vertex v of the graph is a record containing the following fields: `onStack`, a Boolean flag which is **true** if and only v is on the

stack; `next`, used for linking purposes, and in the first place to implement the stack itself; and `type`, a flag with only two possible values, `PLUS` or `MAX` depending on whether v is a + or max vertex. Initially, v.`onStack` is **false**, and `S` = `NULL`. The vertex v also contains the following fields, which will be filled in by the algorithm: `index` (depth-first search index), `low` (lowest depth-first search index of vertices in the same scc), `sccRoot` (root vertex of v's scc), and `degree`, all initially undefined. The latter field will eventually contain either a natural number—the degree of the polynomial if the sequence associated with v has asymptotic polynomial behavior—or the special constant `INFTY`, denoting exponential behavior. We agree that `INFTY` is strictly larger than any natural number, and that `INFTY` + 1 = `INFTY`.

Edges are left implicit, and are handled by loops of the form "**for** each successor w" in the code. To obtain the desired degrees, we launch SCC on the root r of our graph; when it returns, we read off the `degree` field from r.`sccRoot`.

The algorithm of Figure 4 is not our contribution. Our contribution is the function COMPUTECOMPLEXITIES, called at the end of the loop in Tarjan's algorithm. At this point, l is guaranteed to hold a list of all the vertices in the just discovered scc, with root v, that list has been severed off the stack and has v as last entry, and for each vertex w in l, w.`sccRoot` = v.

The function COMPUTECOMPLEXITIES is shown in Figure 5, right, and follows our informal description: if the scc l is trivial, then its degree, stored in the `degree` field of its root l.`sccRoot`, is the maximum of all degrees of the successor sccs of l; otherwise, its degree is the maximum of the same degrees (computed as variable dv), possibly incremented by 1 in case of expensive edges. The helper functions TRIVIALSCC and BADVERTEX determine whether their argument is a trivial scc, resp., a bad vertex.

8.2 Experimental Results

We have implemented that algorithm inside Orchids, and its ten standard signatures. Execution time was negligible. We had the pleasant surprise of observing that all our signatures had polynomial thread complexity, confirming our intuition that human experts do not write signatures with exponential behavior.

The largest observed complexity is $\Theta(n^3)$ for `lin24_ptrace.rule`, a signature that attempts to detect the `ptrace` attack [14]. The second largest is $\Theta(n^2)$ for the `apachessl.rule`, a signature that tries to correlate abnormal variations in message entropy [9] with specific failure events from the Apache server. Out of our ten signatures, seven others have linear behavior, including the pid tracker of Figure 1. We have instrumented our algorithm so that it reports the main causes of complexity. For example, on the pid tracker, our algorithm reports:

```
rule pidtrack may have worst case linear behavior, i.e., O(#events).
  each event may fork a new thread going to 'init'.
```

And indeed, each newly created Unix process (through the `clone` system call, a Linux abstraction behind the more well-known `fork` call) may cause the creation of a new Orchids thread for that signature, starting at state `init`.

```
function trivialScc(l:vertex)
    if l.next ≠ NULL then
        ▷ if there is another vertex
                                ▷ in the scc,
                        ▷ then it is not trivial
        return false
    end if
    for each successor w of l do
        if w = l then
                ▷ if there is a self-loop,
                            ▷ then the scc
                            ▷ is not trivial
            return false
        end if
    end for
    return true
end function

function badVertex(w:vertex)
    if w.type ≠ PLUS then
        ▷ MAX vertices are never bad
        return false
    end if
    v ← w.sccRoot
    weight ← 0
    for each edge w --a--> z do
        if z.sccRoot = v then
            weight ← weight + a
        end if
    end for
    return weight ≥ 2
end function
```

```
function computeComplexities(l:vertex)
    v ← l.sccRoot
    v.degree ← 0
    if trivialScc(l) then
                ▷ take max of degrees of successors
        for each successor w of l do
            if w.sccRoot.degree > v.degree then
                v.degree ← w.sccRoot.degree
            end if
        end for
    else                            ▷ loop over w in scc l
        w ← l
        while w ≠ NULL do
            if badVertex(w) then
                            ▷ exponential behavior
                v.degree ← INFTY
                break                ▷ exit while loop
            else
                for each successor z of w do
                    if z.sccRoot ≠ v then
                        dv ← z.sccRoot.degree
                        if w.type = PLUS then
                                    ▷ expensive edge
                            dv ← dv + 1
                        end if
                            ▷ take max(dv, v.degree)
                        if dv > v.degree then
                            v.degree ← dv
                        end if
                    end if
                end for
            end if
            w ← w.next
        end while
    end if
end function
```

Figure 5: Computing complexities

The last of the ten signatures, `taint_auditd.rule`, a tainting mechanism for detecting illegal transitive information flows, is not even flagged by our algorithm: it is correctly classified as generating a *constant* number of threads.

Let us focus on `lin24_ptrace.rule` in order to understand what is going on in the worst experienced case. We will not describe the actual signature: see [13], where we used a version of that rule as a running example.

$\Theta(n^3)$ is an overestimation: the actual complexity of that signature is $\Theta(n^2)$. The source of the overestimation lies in the following piece of Orchids signature:

```
state ptrace_poketext {
  $counter = $counter + 1;

  expect (.pid == $attack_pid &&
          .syscall == SYS_ptrace &&
          .ptrace_req == POKETEXT &&
          .ptrace_pid == $target_pid &&
          $counter < 10)
    goto ptrace_poketext;
  expect (.pid == $attack_pid &&
          .syscall == SYS_ptrace &&
          .ptrace_req == DETACH &&
          .ptrace_pid == $target_pid)
    goto ptrace_detach;
}
```

Let `p` be the state `ptrace_poketext`, and `d` be the state `ptrace_detach` (not shown). Our complexity analyzer correctly determines that $u_n^{\mathtt{d}} = \Theta(1)$, and produces the equations:

$$u_n^{\mathtt{p}} = u_n^{\tau_1} + u_n^{\tau_2} \qquad u_{n+1}^{\tau_1} = u_n^{\mathtt{p}} \qquad u_{n+1}^{\tau_2} = u_n^{\mathtt{d}}$$

where τ_1 and τ_2 are the two **expect** transitions. Both have the `NO_WAIT` flag, as determined by the Orchids signature compiler. Note that, otherwise, the equation defining u^{τ_1} would have to be $u_{n+1}^{\tau_1} = u_n^{\tau_1} + u_n^{\mathtt{p}}$, in which case $u_n^{\tau_1}$ and $u_n^{\mathtt{p}}$ would behave as $\Theta(2^n)$. This dramatically confirms the claim made in [8] that optimizations found by static analysis are crucial: here they allow us to replace an exponential behavior by a polynomial behavior, preserving the semantics.

The above equations yield an asymptotic estimate for $u_n^{\mathtt{p}}$ of $\Theta(n)$. There are two expensive edges (not shown) between the root node and vertex $u^{\mathtt{p}}$, leading to an estimated complexity of $\Theta(n^3)$ for the whole rule. This result is not optimal: as one can see from the use of the variable `$counter` in state p, and since `$counter` starts at 0, the loop on that state can only be taken at most 10 times. A more precise analysis, taking that into account, would generate the following equation for $u_n^{\mathtt{p}}$:

$$u_{n+10}^{\mathtt{p}} = \sum_{i=0}^{9} u_{n+i}^{\mathtt{d}},$$

which would give a more precise estimate $u_n^{\mathtt{p}} = \Theta(1)$, hence an overall complexity of $\Theta(n^2)$. That would be doable, given some additional static analyses.

There is a final source of imprecision in our complexity estimation algorithm. Among the static analyses that Orchids implements, the *monotonicity* analysis allows Orchids to remove threads that are waiting on conditions b that have failed once, provided we can show that b is antitonic [8, Section 5]. Consider an **expect** transition of the form:

```
expect (.syscall == SYS_open &&
        .time <= $start+$delay) goto q;
```

which one would write to detect a call to the `open()` system call within `$delay` time units, assuming `$start` holds the current time. The `.time` field is trusted to evolve in a non-decreasing way as new events flow in. Then, the Boolean condition `.time <= $start+$delay` is antitonic in the sense that its value can only decrease (go from true to false), and never increase. Therefore, if it fails once, it will fail forever. An Orchids thread that is waiting on the above **expect** transition is then reclaimed if `.time <= $start+$delay` is found to be false just once. Contrarily to the `NO_WAIT` flag, Orchids discovers that it can remove the corresponding thread at run-time, not at compile-time. As a consequence, that is ignored by our complexity analyzer, which operates entirely at compile-time: it computes an upper bound on the number of created threads, but ignores deletions (except for the thread killing mechanism arising from commit states, which can be predicted at compile-time).

Should we refine our complexity analyzer to handle such thread deletions? Certainly, if the monotonicity analysis is ever useful, we should do so. We should start by evaluating the gap between the observed number of active threads and the number predicted by our complexity analyzer. However, and perhaps sadly so, the monotonicity analysis may be of limited use. Of the signatures we used, only two would have benefited from that mechanism: the `ssh_failed_burst` rule and the `ssh_failed_long_window` rule, which detect bursts of ssh failures (more than N failed connection attempts during some fixed amount of time, where N is a constant). The author of these rules, Baptiste Gourdin, preferred to write them in a different style. To explain that on the example of the above **expect** transition, he would have written it as:

```
expect (.syscall == SYS_open) goto q;
expect (.time > $start+$delay) goto stop;
```

where `stop` is a commit state. That actually implements the desired timeout by relying on commit states instead: if `.time` ever exceeds `$start+$delay`, then Orchids will enter state `stop`, and that will kill the thread waiting on the condition `.syscall==SYS_open`, among others.

9 Conclusion

We have described, and proved, a linear time algorithm that decides the asymptotic complexity of sequences defined by certain forms of systems of recurrence

equations, using both the + and max operators. Our goal was to analyze, automatically, which Orchids signatures have polynomial detection complexity (in terms of numbers of created Orchids threads), and with which exponent.

This turns out to be an extremely reliable and useful tool to Orchids signature writers. Personal experience shows that a high degree in the polynomial, or worse, an estimation of exponential complexity, is indicative of a mistake in the writing of the signature.

Beyond Orchids, it seems obvious that our simple algorithm for estimating the asymptotic complexity of recurrence equations should find applications outside of security or of runtime verification. Mounir Assaf recently proposed a (yet unpublished) static analysis that detects whether leakage of sensitive data in security programs is negligible or not [3]. This is based on estimating the rate of growth of a sequence u_n as the number of steps taken, n, tends to infinity, and we hope that our algorithm, or similar techniques, apply.

Acknowledgement

The first author would like to thank Mounir Assaf for drawing his attention to analytic combinatorics, and the anonymous referees for their suggestions.

References

[1] Akian M, Bapat R, Gaubert S (2006) Max-plus algebras. In: Hogben L (ed) Handbook of Linear Algebra, Discrete Mathematics and Its Applications, vol 39, Chapman and Hall/CRC, chap 25

[2] Albert E, Arenas P, Genaim S, Puebla G (2011) Closed-form upper bounds in static cost analysis. Journal of Automated Reasoning 46(2):161–203

[3] Assaf M (2015) From qualitative to quantitative program analysis : Permissive enforcement of secure information flow. PhD thesis, Université Rennes I

[4] Basin D, Klaedtke F, Müller S, Zălinescu E (2015) Monitoring metric first-order temporal properties. Journal of the Association for Computing Machinery 62(2):15:1–15:45

[5] Brockschmidt M, Emmes F, Falke S, Fuhs C, Giesl J (2014) Alternating runtime and size complexity analysis of integer programs. In: Proc. Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14), Springer Verlag Lecture Notes in Computer Science 8413, vol 8413

[6] Flajolet P, Sedgwick R (2009) Analytic Combinatorics. Cambridge University Press, iSBN 0521898064. ISBN-13 978-0521898065

[7] Flores-Montoya A, Hähnle R (2014) Resource analysis of complex programs with cost equations. In: Proc. 12th Asian Symposium on Programming Languages and Systems (APLAS'14), Singapore, Singapore, Springer Verlag Lecture Notes in Computer Science 8858

[8] Goubault-Larrecq J, Olivain J (2008) A smell of Orchids. In: Leucker M (ed) Proceedings of the 8th Workshop on Runtime Verification (RV'08), Springer, Budapest, Hungary, Lecture Notes in Computer Science, vol 5289, pp 1–20, DOI 10.1007/978-3-540-89247-2_1

[9] Goubault-Larrecq J, Olivain J (2013) On the efficiency of mathematics in intrusion detection: The NetEntropy case. In: Danger JL, Debbabi M, Marion JY, Garcia-Alfaro J, Zincir-Heywood N (eds) Revised Selected Papers of the 6th International Symposium on Foundations and Practice of Security (FPS'13), Springer, La Rochelle, France, Lecture Notes in Computer Science, vol 8352, pp 3–16, DOI 10.1007/978-3-319-05302-8_1

[10] Havelund K, Reger G (2015) Specification of parametric monitors - quantified event automata versus rule systems. In: Drechsler R, Kuhne U (eds) Formal Modeling and Verification of Cyber-Physical Systems, Springer Verlag, pp 151–189, 1st International Summer School on Methods and Tools for the Design of Digital Systems (SyDe), Bremen, Germany

[11] Jin D, O'Neil Meredith P, Lee C, Roşu G (2012) JavaMOP: Efficient parametric runtime monitoring framework. In: Proceeding of the 34th International Conference on Software Engineering (ICSE'12), IEEE, pp 1427–1430, DOI 10.1109/ICSE.2012.6227231

[12] Luo Q, Zhang Y, Lee C, Jin D, O'Neil Meredith P, Şerbănuţă TF, Roşu G (2014) RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In: Bonakdarpour B, Smolka SA (eds) Proceedings of the 5th International Conference on Runtime Verification (RV'14), Springer Verlag LNCS 8734, Toronto, ON, CA, pp 285–300

[13] Olivain J, Goubault-Larrecq J (2005) The Orchids intrusion detection tool. In: Etessami K, Rajamani S (eds) Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05), Springer, Edinburgh, Scotland, UK, Lecture Notes in Computer Science, vol 3576, pp 286–290, DOI 10.1007/11513988_28

[14] Purczyński W (2003) Linux kernel privileged process hijacking vulnerability. `http://www.securityfocus.com/bid/7112`, bugTraq Id 7112. Last read: september, 2003

[15] Roger M, Goubault-Larrecq J (1999) Procédé et dispositif de résolution de modèles, utilisation pour la détection des attaques contre les systèmes informatiques. Dépôt français du 13 sep. 1999, correspondant Dyade, demandeurs : 1. INRIA 2. Bull S.A. Numéro de publication: 2 798 490. Numéro d'enregistrement national: 99 11716. Classification: G 06 F 19/00.

Date de mise à la disposition du public de la demande: 16 mars 2001, bulletin 01/11.

[16] Roger M, Goubault-Larrecq J (2001) Log auditing through model checking. In: Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW'01), IEEE Computer Society Press, Cape Breton, Nova Scotia, Canada, pp 220–236

[17] Tarjan RE (1972) Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2):146–160