Exploring Worst-Case Scenarios of Self-Stabilizing Algorithms

Erwan Jahier

Verimag - Joint work with Karine Altisen and Stéphane Devismes

Synchron La Rochette

November 22, 2021

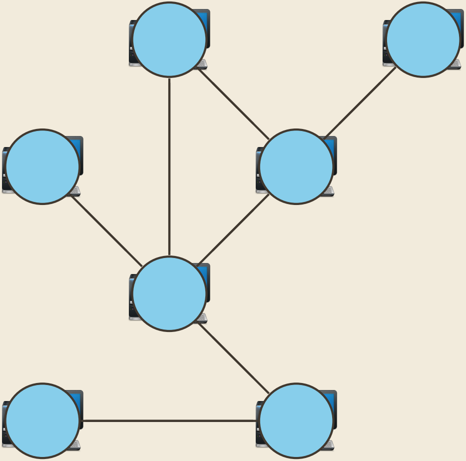
Outline

1. Self-stabilizing Algorithms in the Atomic-State Model
2. SASA: Simulation of Self-stabilizing Algorithms
3. Exploring Worst-cases via Local Search
4. Exploring Worst Initial Configurations
5. Exploring Worst-Cases via Model-Checking
6. Exploring Asymptotic Behavior via Simulation Campaigns
7. Conclusion

Plan

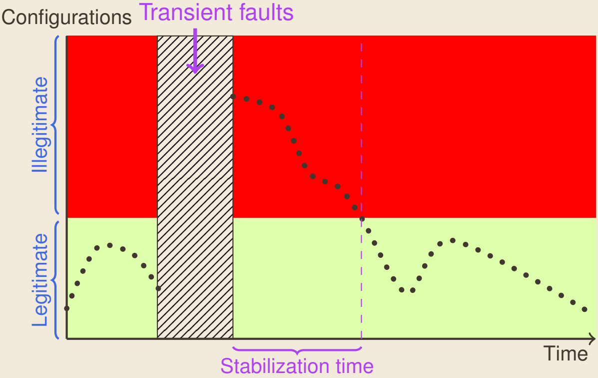
1. Self-stabilizing Algorithms in the Atomic-State Model
2. SASA: Simulation of Self-stabilizing Algorithms
3. Exploring Worst-cases via Local Search
4. Exploring Worst Initial Configurations
5. Exploring Worst-Cases via Model-Checking
6. Exploring Asymptotic Behavior via Simulation Campaigns
7. Conclusion

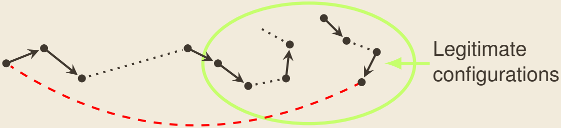
Distributed Systems Algorithms



- Process
  - Autonomous
  - Interconnected
- Hypotheses
  - Connected
  - Bidirectional
- Expected Property
  - Fault-tolerance

Self-Stabilizing Algorithms

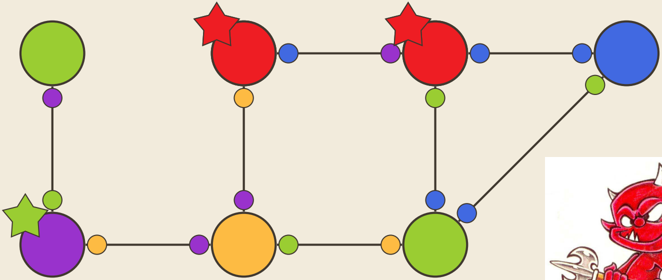




Atomic (Synchronous?) State Model

Performing an Atomic Step consists in:

1. Reading neighbors variables
2. Computing enabled nodes
3. Choosing nodes to activate: a Daemon models the asynchronism
4. Computing a new configuration



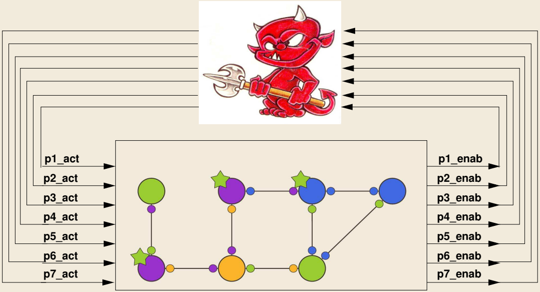
Algorithms in the ASM viewed as Reactive programs

loop:

1. Reads neighbors vars
2. Computes `pi_enab`
3. Chooses `pi_act` (Daemon)
4. Computes states (`pi_act`)

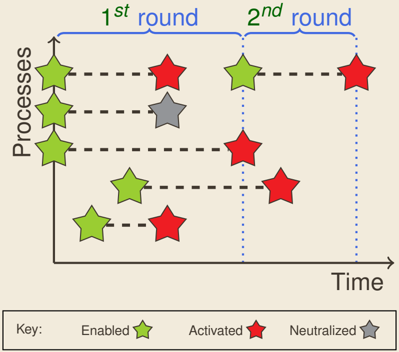
loop:

- 4. **Init** -> Computes states (`pi_act`)
- 1. Reads neighbors vars
- 2. Computes `pi_enab`
- 3. Chooses `pi_act` (Daemon)



Goal: Study the Algorithm Complexity

- Space Complexity: memory requirement in **bits**
- Time Complexity (stabilization time) in
  - **steps**, **moves**
  - **rounds**: capture the execution time of the slowest processes



Message Passing Versus Atomic State Models

- Message Passing Model (MPM)
    - Used in the Distributed Algorithms community
    - Lower-level: queues of events
- Atomic State Model (ASM):
    - Used in the Self-Stabilizing Algorithms community
    - Higher-level: atomic instantaneous communications
- General Algorithms transformations exist

Reminiscent to the synchronous / a-synchronous point of view

Some Classical Examples

- Dijkstra’s Token Ring
- Coloring Algo
- Synchronous or A-Synchronous Unison
- BFS or DFS spanning tree construction
- etc.

Coloring Algo

For each process p

- Parameters:
  - K : an integer such that $K \geq \Delta$
  - $p.N$: the set of p's neighbors
- Local Variable:
  - $p.c \in \{0, ..., K\}$ holds the color of p
- Local functions:
  - $Used(p) = \{q.c : q \in p.N\}$
  - $Free(p) = \{0, ..., K\} \setminus Used(p)$
  - $Conflict(p) = \exists q \in p.N : q.c = p.c$
- Action:
  - Color :: Conflict(p) $\hookrightarrow p.c \leftarrow min(Free(p))$

```
cd test/coloring; rdbgui4sasa -sut "sasa grid4.dot
-locally-central-demon"
```

Plan

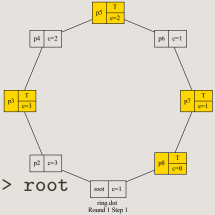
Simulating Self-stabilizing Algorithms: What for?

- Debugging
    - Simulate existing algorithms
    - Design new algorithms
- Get Insights on the Algorithms Complexity
    - Average case Complexity
    - Check if the theoretical worst case is good/correct
    - etc.

Defining The Network Topology

- Take advantage of the GraphViz `dot` language
    - Simple syntax
    - Open-source
    - Plenty of visualizers, editors, parsers, exporters

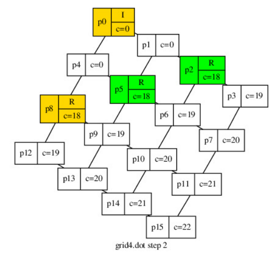
```
digraph ring {
 root [ algo="root.ml" init="1" ]
 p2   [ algo="p.ml"    init="3" ]
 p3   [ algo="p.ml"    init="3" ]
 p4   [ algo="p.ml"    init="2" ]
 p5   [ algo="p.ml"    init="2" ]
 p6   [ algo="p.ml"    init="1" ]
 p7   [ algo="p.ml"    init="1" ]
 p8   [ algo="p.ml"    init="0" ]
 root -> p2 -> p3 -> p4 -> p5 -> p6 -> p7 -> p8 -> root
}
```



- `dot` attributes
    - name-value pairs that can be ignored (as C `#pragmas`)
    - node attributes: `algo`, `init`
    - graph attributes: global simulation parameters

A Topology Example: a 4x4 grid

```
graph g {
  p0  [algo="p.ml"  init="0"]      p0 -- p1 -- p2 -- p3 -- p7
  p1  [algo="p.ml"  init="17"]     p0 -- p4 -- p5 -- p6
  p2  [algo="p.ml"  init="18"]     p11-- p15
  p3  [algo="p.ml"  init="19"]     p1 -- p5 -- p9
  p4  [algo="p.ml"  init="17"]     p10 -- p11 -- p7
  p5  [algo="p.ml"  init="18"]     p10 -- p14 -- p15
  p6  [algo="p.ml"  init="19"]     p10 -- p6
  p7  [algo="p.ml"  init="20"]     p10 -- p9
  p8  [algo="p.ml"  init="18"]     p12 -- p13 -- p14
  p9  [algo="p.ml"  init="19"]     p12 -- p8 -- p9
  p10 [algo="p.ml"  init="20"]     p13 -- p9
  p11 [algo="p.ml"  init="21"]     p2 -- p6 -- p7
  p12 [algo="p.ml"  init="19"]     p4 -- p8
  p13 [algo="p.ml"  init="20"]     }
  p14 [algo="p.ml"  init="21"]
  p15 [algo="p.ml"  init="22"]
```



grid4.dot step 2

Algorithm Programming Interface (1/2): **to provide**

- 42 straightforward loc of Ocaml Interface (`mli`) file
- The Local State Type is **polymorphic**

```
type 's neighbor (* bool, int, float, array, struct, etc. *)
```

- For each local algorithm, one need to define 3 functions:
  1. an **enable** function, which encodes the **guards** of the algorithm
  2. a **step** function, that **triggers** enabled actions
  3. a state **initialization** function
     - used if no initial value is provided in the DOT file

```
type 's enable_fun = 's -> 's neighbor list -> action list
type 's step_fun   = 's -> 's neighbor list -> action -> 's
type 's state_init_fun = int -> string -> 's
```

Algorithm Programming Interface (2/2): **can be used**

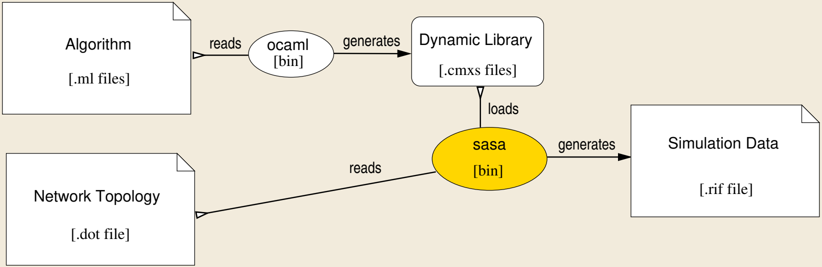
Each node can access to information concerning **neighbors**. . .

```
val state : 's neighbor -> 's
```

. . . or **topology**

```
val card: unit -> int
val links_number : unit -> int
val diameter: unit -> int
val min_degree : unit -> int
val mean_degree : unit -> float
val max_degree: unit -> int
val is_cyclic: unit -> bool
val is_connected : unit -> bool
val is_tree : unit -> bool
...
val get_graph_attribute : string -> string
```

The SASA Core Simulator Architecture



Dijkstra's Token Ring For **each Non-Root** (2/2):noexport:

- Parameters:
  k : a positive integer
  $p.Pred$: the predecessor of p in the ring
- Local Variable:
  $p.v \in \{0, ..., k-1\}$
- Action:
  "a" :: $p.v \neq p.Pred.v \hookrightarrow p.v \leftarrow p.Pred.v$

```
open Algo
let k = 42
let init_state _ _ = Random.int k
let enable_f e nl =
 if e<>state (List.hd nl) then ["a"]
                          else []
let step_f e nl a = state (List.hd nl)
```

[batch demo]

[manual daemon demo]

[interactive demo (distributed)]

[interactive demo (central)]

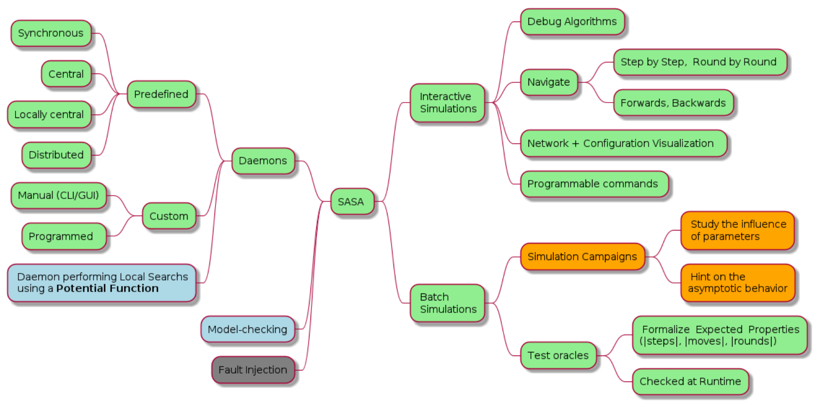
[interactive demo (synchronous)]

Graph Coloring

- Parameters:
  $p.N$: the set of p's neighbors ;
  k : an integer such that $k \geq \Delta$
- Local Variable:
  $p.c \in \{0, ..., k\}$ holds the color of p
- Local functions:
  $Used(p) = \{q.c : q \in p.N\}$
  $Free(p) = \{0, ..., k\} \setminus Used(p)$
  $Conflict(p) = \exists q \in p.N : q.c = p.c$
- Action:
  "conflict" :: Conflict(p)
  $\hookrightarrow p.c \leftarrow min(Free(p))$

```
open Algo
let k=3
let init_state _ _ = Random.int k
let neigbhors_vals nl = List.map (fun n -> state n) nl
let confl v nl = List.mem v (neigbhors_vals nl)
let free nl =
 let confll = List.sort_uniq compare (neigbhors_vals nl) in
 let rec aux free confl i =
   if i > k then free else
    (match confl with
      | x::tail ->
       if x=i then aux free tail (i+1)
             else aux (i::free) confl (i+1)
      | [] -> aux (i::free) confl (i+1)
  )
  in
  List.rev (aux [] confll 0)
let enable_f e nl=if (confl e nl) then ["conflict"] else []
let step_f e nl a = if free nl = [] then e else List.hd f
let actions = Some ["conflict"]
```

SASA Main Features



- green: SYNCHRON’19, TAP’20
- orange: SSS’20 tutorial, The Computer Journal (to appear)
- blue: today

Plan

1. Self-stabilizing Algorithms in the Atomic-State Model
2. SASA: Simulation of Self-stabilizing Algorithms
3. Exploring Worst-cases via Local Search
4. Exploring Worst Initial Configurations
5. Exploring Worst-Cases via Model-Checking
6. Exploring Asymptotic Behavior via Simulation Campaigns
7. Conclusion

Potential Functions

- Potential Functions (Φ)
  - map configurations to a numeric value
  - null for legitimate configurations
- SS Algorithms correctness Proof technique:

$$\exists \Phi \ \forall i \ \Phi(c_{i+1}) - \Phi(c_i) \leq 1$$

- Can be used to implement daemons that explore **Wort-Cases scenarios**

Greedy Daemon

- At each step, chooses the one that maximizes the PF
- nb: for most algorithms, the worst daemon is **central**
- of course, greedy daemons **may miss the true worst-case**
- The initial configuration influences **a lot** the worst-case

→ We need a **Local Search** Infrastructure

nb: configurations are **often** made of bounded integers

Local Search

- **Solve optimization problems**, i.e., find the minimum (or max) of some cost function
- Explore the search space via **neighbors** ($\neq$ Global Search)
- Do not explore neighbors that can not lead the best solution (branch-and-bound)
- Various heuristics
  - DFS, BFS, Stochastic Hill-climbing, Beam-search (explore **some of the neighbors of higher** cost)
  - Tabu list: remember **some of** the recently visited nodes
  - Simulated Annealing: choose bad neighbors from times to times (to exit from local minimum)

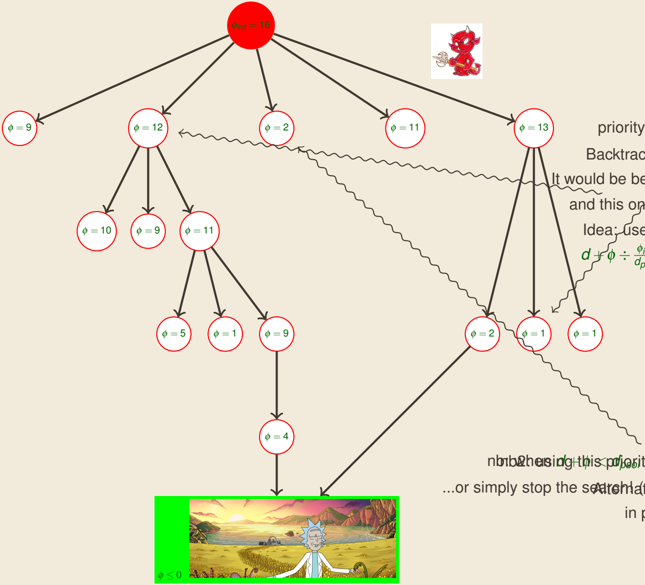
A LocalSearch ocaml API (1/2)

```
type (’n, ’tv, ’v) t = {
  init : ’n * ’tv * ’v;
  succ : ’n -> ’n list; (* returns (all or some) neighbors *)

  is_goal : ’n -> bool;    (* is the node a solution of the problem *)
  stop : ’n -> ’n -> bool; (* if [stop pre_sol n], stop the search *)
  cut: ’n -> ’n -> bool;   (* if [cut pre_sol n], don’t explore n *)

  push : ’tv -> ’n -> ’tv; (* add the node in the set of nodes to visit *)
  pop  : ’tv -> (’n * ’tv) option; (* pick a node to visit *)

  visiting : ’n -> ’v -> ’v;   (* mark a node as visited *)
  visited  : ’n -> ’v -> bool; (* check if a node has been visited *)
}

type ’n sol = Stopped | NoMore | Sol of ’n * ’n moresol
and ’n moresol = ’n option -> ’n sol

val run : (’n, ’tv, ’v) t -> ’n moresol
```

A `LocalSearch` ocaml API (2/2)

- Storing Already Visited nodes (`'v`)
  - None
  - All (depending on the configuration space size)
  - Tabu lists
- Storing nodes to be visited (`'tv`)
  - only keep the best cost (greedy)
  - priority queues (exhaustive)
    - -depth $\otimes$ ϕ: BFS
    - depth $\otimes$ ϕ: DFS (less memory, get first solutions faster)
    - ϕ: ~BFS
    - depth+ϕ: ~DFS

Using `LocalSearch` to implement wort-case daemons



priority=$d + \phi$ ⇒ Greedy daemon!

Backtrack for more solutions?

It would be better to find the best sol sooner (timeouts)

and this one looks more promising

Idea: use previous solution to estimate ϕ expectation

$d + \phi \div \frac{\phi_{init}}{d_{psol}} = 1 + \frac{12}{5.333...} = 3.25 > 2.375$

nb: using this priority favors a Breadth Search...

nb2: when $d + \phi < d_{psol} = 5$ we can cut the search...

...or simply stop the search! (as its priority would be low)

Alternate with DFS?

in parallel?

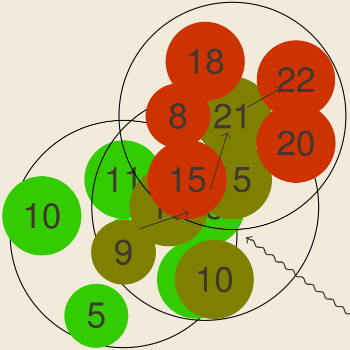
Plan

1. Self-stabilizing Algorithms in the Atomic-State Model
2. SASA: Simulation of Self-stabilizing Algorithms
3. Exploring Worst-cases via Local Search
4. Exploring Worst Initial Configurations
5. Exploring Worst-Cases via Model-Checking
6. Exploring Asymptotic Behavior via Simulation Campaigns
7. Conclusion

Search for Bad initial configurations via Local Search

- Given an initial configuration `c`
- in loop:
  - Choose a **neighbor** `nc` of `c`
  - compute their **cost**: **simulation step number** using some **daemon**
    - central, distributed, etc.
    - greedy, exhaustive
  - keep the **best** config: `c` $\leftarrow$ `max_cost(c,nc)`

Stochastic Beam Search Local



Compute the cost (in parallel) and
... and pop the best one
push the more costly in a `PQ.t`...

Stochastic Beam Search Local: a possible variant

- nb: we push n_i with priority $cost(n_i)$ if $cost(n_i) > cost(n)$
  - a variant to (try to) exit from local maxima: with probability ($e^{-\Delta cost/T}$, where T $\searrow_0$) restart from n_i when $cost(n_i) < cost(n)$) (*Simulated Annealing*)
- nb 2: the cost of neighbors can easily be computed in **parallel** (`functory`)

The Stochastic Beam Search LocalSearch parameters

- Set by users
  - The daemon used to compute the cost of each initial configuration
    - central, distributed, etc.
    - greedy, exhaustive
  - The size of the beam
  - The number of simulations `ns`
- (currently) Hard-coded
  - The number `n` of neighbors in the beam (depends on `ns`)
  - The choice of neighbors in the beam
    - 1 by changing **all** values of the configuration
    - 1 by changing **each** value of the configuration with a probability 0.5
    - n-2 by changing **one** value of the configuration

Demo

```
cd ~/sasa/test/dijkstra-ring/
make ring_noinit.cmxs
sasa -cd ring_noinit.dot
sasa -gcd ring_noinit.dot
sasa -is 1000 -cd ring_noinit.dot
sasa -is 1000 -gcd ring_noinit.dot
sasa -is 100 -ed -q ring_noinit.dot # not interesting?
```

Plan

1. Self-stabilizing Algorithms in the Atomic-State Model
2. SASA: Simulation of Self-stabilizing Algorithms
3. Exploring Worst-cases via Local Search
4. Exploring Worst Initial Configurations
5. Exploring Worst-Cases via Model-Checking
6. Exploring Asymptotic Behavior via Simulation Campaigns
7. Conclusion

SALut - Self-stabilizing Algorithms in LUsTre

- From
    - An API to program Self-Stabilizing algorithms in Lustre
    - a dot 2 Lustre compiler (trainee project, 1 day/week × 3 months)
- Be able to
    - perform efficient simulation (?)
    - perform model-checking

The Lustre sasa API

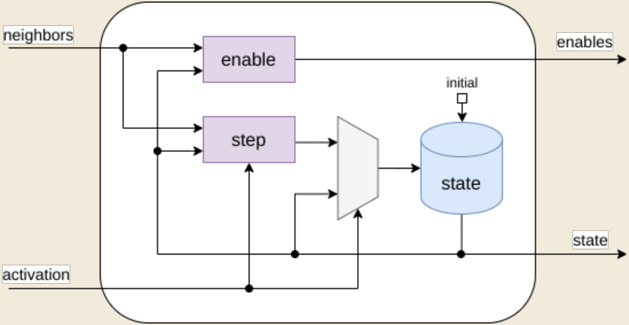
To be provided:

```
(* Computes the set of actions enabled in this configuration. *)
function p_enable<<const degree:int>>(
   this : state;
   neighbors : state^degree
) returns (enabled : bool^actions_number);

(* Executes the given action, returning the updated node state. *)
function p_step<<const degree:int>>(
   this : state;
   neighbors : state^degree;
   action : action
) returns (new : state);
```

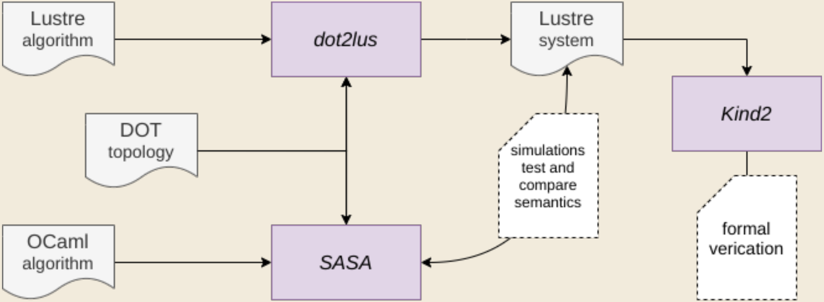
A Lustre sasa API

Generated by `dot2lus`

```
node network(
   activate : bool^actions_number^card; config_init : state^card
) returns (
   enable : bool^actions_number^card; config : state^card;
);
```



Validating the Lustre translations (algo+network)



Proving an (easy) open-problem

A least upper bound of three steps on the stabilization time of Dijkstra's token ring algorithm in a ring with three nodes.

Plan

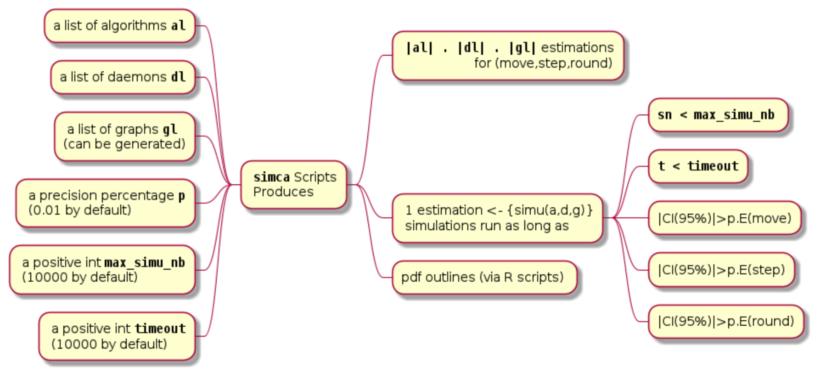
Simulation Campaigns

The distribution contains scripts to support **SIMulation CAMpaigns**:

- the `sasa/tools/simca/` directory of the git repository
  - Ocaml scripts to **automate the running** of simulations
  - R scripts to produce graphical outlines
- cf in the set of sasa tutorials (*), the ones named:
  - " Simulation Campaigns with sasa "
  - " Comparing Spanning Trees Construction "

(*) `https://verimag.gricad-pages.univ-grenoble-alpes.fr/vtt/tags/sasa/`

Simulation Campaigns



a list of algorithms **al**

a list of daemons **dl**

a list of graphs **gl**
(can be generated)

a precision percentage **p**
(0.01 by default)

a positive int **max_simu_nb**
(10000 by default)

a positive int **timeout**
(10000 by default)

simca Scripts
Produces

|al| . |dl| . |gl| estimations
for (move,step,round)

1 estimation <- {simu(a,d,g)}
simulations run as long as

sn < max_simu_nb

t < timeout

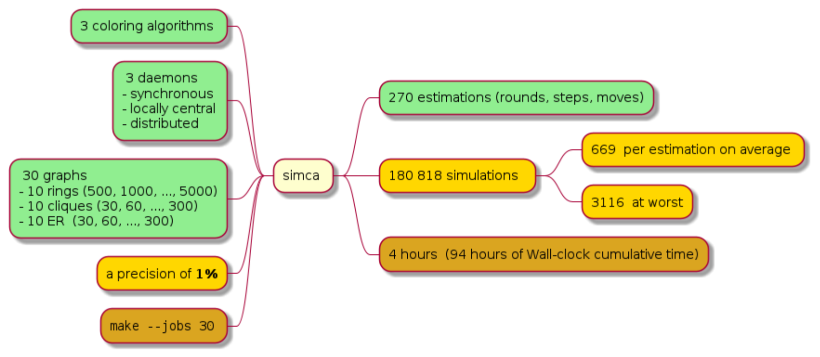
|CI(95%)|>p.E(move)

|CI(95%)|>p.E(step)

|CI(95%)|>p.E(round)

pdf outlines (via R scripts)

a Simulation Campaign: comparing 3 coloring Algos



3 coloring algorithms

3 daemons
- synchronous
- locally central
- distributed

30 graphs
- 10 rings (500, 1000, ..., 5000)
- 10 cliques (30, 60, ..., 300)
- 10 ER (30, 60, ..., 300)

a precision of **1%**

`make --jobs 30`

simca

270 estimations (rounds, steps, moves)

180 818 simulations

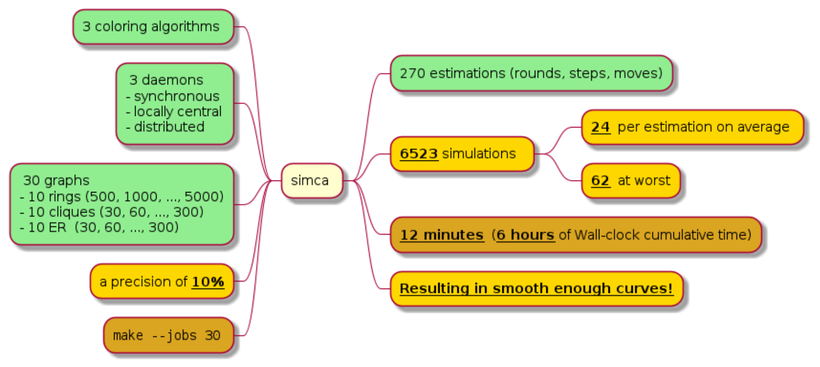
669 per estimation on average

3116 at worst

4 hours (94 hours of Wall-clock cumulative time)

a Simulation Campaign on 3 coloring Algorithms

Using a lower precision (1% → 10%)



3 coloring algorithms

3 daemons
- synchronous
- locally central
- distributed

30 graphs
- 10 rings (500, 1000, ..., 5000)
- 10 cliques (30, 60, ..., 300)
- 10 ER (30, 60, ..., 300)

a precision of **10%**

`make --jobs 30`

simca

270 estimations (rounds, steps, moves)

6523 simulations

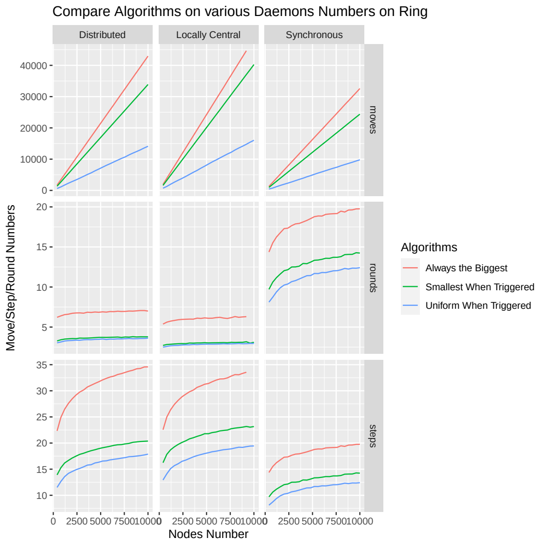
24 per estimation on average

62 at worst

12 minutes (**6 hours** of Wall-clock cumulative time)

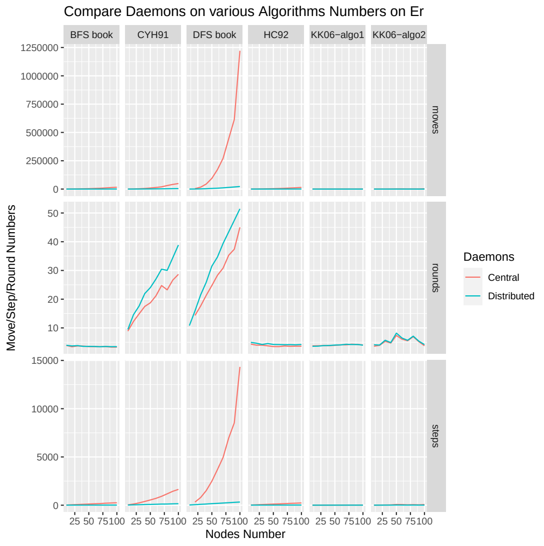
Resulting in smooth enough curves!

Some of the generated graphics



“ Simulation Campaigns with sasa ”

Ditto on 6 **Spanning Trees Construction** Algorithms



“ Comparing Spanning Trees Construction ”

Plan

Further work

- Experiments
- Local Search Tuning
- A more efficient dot to Lustre translation schema
- Take advantage of this Local Search AP (Design pattern?) to use Quantitative oracles in Lurette

Conclusion

- An open-source SimulAtor of **Self-stabilizing Algorithms**
- writen using the **atomic-state** model (the most commonly used in Self-Stab)
- Rely on **existing** tools as much as possible
  - `dot` for Graphs
  - `ocaml` for programming local algorithms
  - *Synchrone (Verimag)* Team Tools for simulation
- Installation via
  - `docker`
  - `opam`
  - `git`

`https://verimag.gricad-pages.univ-grenoble-alpes.fr/synchrone/sasa`