Exploring Worst Cases of Self-stabilizing Algorithms using Simulations

Erwan Jahier, Stéphane Devismes, Karine Altisen

Exploring Worst Cases of Self-stabilizing Algorithms using Simulations*

Erwan Jahier[1], Karine Altisen[1], and Stéphane Devismes[2]

[1] Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, France
[2] Université de Picardie Jules Verne, MIS, France

Abstract. Self-stabilization qualifies the ability of a distributed system to recover after transient failures. SASA is a simulator of self-stabilizing algorithms written in the atomic-state model, the most commonly used model in the self-stabilizing area.
A simulator is, in particular, useful to study the time complexity of algorithms. For example, one can experimentally check whether existing theoretical bounds are correct or tight. Simulations are also useful to get insights when no bound is known.
In this paper, we use SASA to investigate the worst cases of various well-known self-stabilization algorithms. We apply classical optimization methods (such as local search, branch and bound, Tabu list) on the two sources of non-determinism: the choice of initial configuration and the scheduling of process activations (daemon). We propose a methodology based on heuristics and an open-source tool to find tighter worst-case lower bounds.

1 Introduction

Usually, simulator engines are employed to test and find flaws early in the design process. Another popular usage of simulators is the empirical evaluation of average-case time complexity via simulation campaigns [3,6,9]. In this paper, we propose to investigate how to build worst-case executions of self-stabilizing algorithms using a simulator engine. For that purpose, we will apply classical optimization methods and heuristics on the two sources of non-determinism: the choice of the initial configuration and the scheduling of process activations. To that goal, we consider SASA [9], an open-source and versatile simulator dedicated to self-stabilizing algorithms written in the atomic-state model, the most commonly used model in self-stabilization. In this model, in one atomic step, a process can read its state and that of its neighbors, perform some local computations, and update its state accordingly. Local algorithms are defined as set of rules of the form $\langle Guard \rangle \rightarrow \langle Statement \rangle$. The guard is a Boolean predicate on the states of the process and its neighbors. The statement is a list of assignments on all or a part of the process' variables. A process is said to be enabled if the guard of at least one of its rules evaluates to true. Executions proceed in atomic steps in which at least one enabled process *moves*, *i.e.*, executes an enabled rule.

Self-stabilization [15] qualifies the ability of a distributed system to recover within finite time after transient faults. Starting from an arbitrary configuration, a self-stabilizing

* This work has been partially funded by the ANR project SkyData (ANR-22-CE25-0008-01).

algorithm makes the system eventually reach a so-called *legitimate* configuration from which every possible execution suffix satisfies the intended specification. Self-stabilizing algorithms are mainly compared according to their *stabilization time*, *i.e.*, the maximum time, starting from an arbitrary configuration, before reaching a legitimate configuration. The stabilization time of algorithms written in the atomic-state model is commonly evaluated in terms of rounds, which measure the execution time according to the speed of the slowest processes. Another crucial issue is the *number of moves* which captures the number of local state updates. By definition, the stabilization time in moves exhibits the amount of computations an algorithm needs to recover a correct behavior. Hence, the move complexity is rather a measure of work than a measure of time: minimizing the number of state modifications allows the algorithm to use less communication operations and communication bandwidth [16].

In the atomic-state model, the asynchrony of the system is materialized by the notion of *daemon*. This latter is an adversary that decides which enabled processes move at each step. The most general daemon is the *distributed unfair* one. It only imposes the progress in the execution, *i.e.*, while there are enabled processes, at least one moves during the next step. Algorithms stabilizing under such an assumption are highly desirable because they work under any daemon assumption. Finally, since it does not impose fairness among process activations, the stabilization time of every self-stabilizing algorithm working under the distributed unfair daemon is necessarily finite in terms of moves.[3]

There are many self-stabilizing algorithms proven under the distributed unfair daemon [6,11,13,19]. However, analyses of the stabilization time in moves remain rather unusual and this is sometime an important issue. Indeed, several self-stabilizing algorithms working under a distributed unfair daemon have been shown to have an exponential stabilization time in moves in the worst case [6] for silent self-stabilizing leader election algorithms given in [11,13], [14] for the BFS spanning tree construction of Huang and Chen [22], and [20] for the silent self-stabilizing algorithm they proposed in [19].

Methods and Contributions. Exhibiting worst-case executions in terms of stabilization time in moves is usually a difficult task since the executions of numerous interconnected processes involve many possible interleavings in executions. The combinatorics induced by such distributed executions is sometime hard to capture in order to prove a relevant lower bound. Hence, we propose here to use the simulator engine SASA to give some insights about worst-case scenarios. By judiciously exploring the transition system, we can expect to quickly find bad scenarios that can be generalized afterwards to obtain tighter bounds.

We consider here self-stabilizing algorithms working under the unfair daemon. Hence, the subgraph of the transition system induced by the set of illegitimate configurations is a directed acyclic graph (DAG). This subgraph can be huge and also dense since from a given configuration we may have up to $2^n - 1$ possible directed edges, where n is the number of nodes. Note that worst-case scenarios in moves are frequently central [6,5]: at each step, exactly one process moves. Therefore, and because it limits the number of possible steps from a configuration to at most n, we focus in the experiments on central

[3] The (classical) weakly fair daemon, for example, does not provide such a guarantee.

schedulers – even if the methods presented in the article actually work for other unfair daemons.

Even with this restriction, the space to explore remains huge. Since worst cases depend on the initial configuration and the scheduling of moves, we propose exploration heuristics targeting these two sources of non-determinism. The goal is to get some insights on algorithms upper bounds, or to assess how tight known upper bounds are.

One of the proposed heuristics relies on so-called *potential functions*. A potential function is a classical technique used to prove convergence (and stabilization) of algorithms: it provides an evaluation of any configuration and decreases along all paths made of illegitimate configurations. We use them to guide the state space exploration, and to use classical optimization techniques (branch and bound). Note that potential functions usually give a rough upper bound on the stabilization time. Again, our approach allows to refine such a bound.

We also propose heuristics based on local search to speed-up the finding of worst-case initial configurations. All those heuristics are implemented into the open-source simulator SASA, and conclusive experiments on well-known self-stabilization algorithms are performed.

Related Work. SASA [9] is an open-source and versatile simulator dedicated to self-stabilizing algorithms written in the atomic-state model. All important concepts used in the model are available in SASA: simulations can be run and evaluated in moves, atomic steps, and rounds. Classical daemons are available: central, locally central, distributed, and synchronous daemons. Every level of anonymity can be considered, from fully anonymous to (partially or fully) identified. Finally, distributed algorithms can be either uniform (all nodes execute the same local algorithm) or non-uniform. SASA can be used to perform batch simulations which can use test oracles to check expected properties. For example, one can check that the stabilization time in rounds is upper bounded by a given function. The distribution provides several facilities to achieve batch-mode simulation campaigns. Simulations can also be run interactively, step by step (forward or backward), for debugging purposes.

Only a few other simulators dedicated to self-stabilization in locally shared memory models, such as the atomic-state model, have been proposed. None of them offers features to deal with worst-case scenarios. Flatebo and Datta [18] propose a simulator of the atomic-state model to evaluate leader election, mutual exclusion, and ℓ-exclusion algorithms on restricted topologies, mainly rings. This simulator has limited facilities including classical daemons and evaluation of stabilization time in moves only. It is not available anymore. Müllner *et al.* [24] present a simulator of the register model, a computational model which is close to the atomic-state model. This simulator does not allow to evaluate stabilization time. Actually, it focuses on three fault tolerance measures initially devoted to masking fault-tolerant systems (namely, reliability, instantaneous availability, and limiting availability [25]) to evaluate them on self-stabilizing systems. These measures are still uncommon today in analyses of self-stabilizing algorithms. The simulator proposed by Har-Tal [21] allows to run self-stabilizing algorithms in the register model on small networks (around 10 nodes). It proposes a small amount of facilities, *i.e.*, the execution scheduling is either synchronous, or controlled step by step

by the user. Only the legitimacy of the current configuration can be tested. It provides neither batch mode, nor debugging tools. Evcimen *et al.* describe in [17] a simulation engine for self-stabilizing algorithms in message passing. Their simulator uses heavy mechanisms to implement this model, such as queue of events, threads, and fault injection. In the Evcimen *et al.*'s simulator, the execution scheduler can be only fully asynchronous. Being corner cases, central and synchronous executions are very useful to find bugs or to exhibit a worst-case scenario.

Several other studies deal with the empirical evaluation of self-stabilizing algorithms [1,2,3]. However, these studies focus on the average-case time complexity. Note that SASA has been also used to tackle average-case stabilization times through simulation campaigns [9].

2 Exploring Daemons

For a given topology T and an initial configuration c_{init}, the stabilization time in moves of an algorithm A depends on the choices made by the daemon at each step. Finding a worst-case stabilization time requires to explore all the illegitimate configurations of the transition system. Hence, we define $\mathscr{R}(A,T,c_{init})$ as the transition system where all legitimate configurations are collapsed into one node, as illustrated in Fig. 1. As the size of $\mathscr{R}(A,T,c_{init})$ grows exponentially, we need exploration heuristics. The goal of those heuristics is to build a scheduling of actions and thus to implement a daemon. We call *exhaustive daemon* the algorithm that builds a central daemon by exploring $\mathscr{R}(A,T,c_{init})$ until finding a longest path; we also use *random daemons* which, at each configuration, pick the next move uniformly at random.

Greedy Daemons. In self-stabilization, a *potential function* ϕ maps configurations to the set of natural integers and satisfies the following two properties: (1) if $\phi(c)$ is minimum, then c is legitimate; (2) ϕ is decreasing over illegitimate configurations, *i.e.*, for every execution $c_0,\ldots c_i,c_{i+1},\ldots$, for every $i \geq 0$, if c_i is illegitimate, then $\phi(c_i) > \phi(c_{i+1})$. Exhibiting such a function is a classical technique to prove the self-stabilization of an algorithm. The idea here is to use potential functions during simulations, and define *greedy daemons* that always choose configurations that maximize ϕ. As shown by the experiments we perform below, for most algorithms, greedy daemons find longer *paths* in $\mathscr{R}(A,T,c_{init})$ than random ones – but not necessarily the longest.

Cutting Exploration Branches. Using a greedy daemon is of course a heuristic that can miss the longest path. To find it, we need to backtrack (branch) in the exploration of $\mathscr{R}(A,T,c_{init})$. A first simple optimization is the following: (1) perform a greedy traversal of $\mathscr{R}(A,T,c_{init})$ to get a lower bound on the maximum number of moves to stabilization; (2) then, during the remaining of the exploration, all configurations which depth (*i.e.*, the distance to c_{init}) plus its potential is less than or equal to the known lower bound will never lead to a longer path: the corresponding branches can then be cut (bound) without missing the worst-case. This can reduce a lot the number of steps necessary to explore exhaustively $\mathscr{R}(A,T,c_{init})$; see experiments below.

Perfect Potential Functions. Given an algorithm, a topology and an initial configuration, we say that the potential function is *perfect* if the corresponding greedy traversal finds in n moves a legitimate configuration when the potential of the initial configuration is n (*i.e.,* if it decreases by one at each move). In such cases, which are easy to detect during simulations, it is useless to continue the search as no better (longer) path can be found.

Tabu List. A classical optimization used to explore graphs is to maintain a (tabu) list of visited nodes in order to avoid to revisit the same node twice. A classical heuristic to prevent this list to grow too much is to keep only the more recently visited nodes. When a configuration α in the tabu list is reached, the length of the path associated to α just need to be updated. This often reduces drastically the exploration time measured in terms of number of visited edges of $\mathscr{R}(A,T,c_{init})$.

Promising Daemons. Consider Fig. 1; according to the values of ϕ, a greedy daemon would choose the path $c_{init}-c_5-c_6-\mathscr{C}_s$. In order to search for a better solution, one could backtrack to the last choice point (c_5), which amounts to perform a depth-first traversal of $\mathscr{R}(A,T,c_{init})$.

As $\mathscr{R}(A,T,c_{init})$ can be huge, exploring it exhaustively can be very long, and the use of a timeout is necessary in practice. In this context, it is better to explore the most promising configurations first. The next configurations that would be explored by a depth-first traversal would be c_7 or c_8; but they do not look promising, as their potential is 2 – which means that at most two more moves would be needed to reach the set of legitimate configurations $\mathscr{C}_s$, and will not lead to big improvements.

By taking into account the depth d in $\mathscr{R}(A,T,c_{init})$ and the potential ϕ, we can choose to backtrack to a more promising configuration. In order to have a choice criterion, we can remark that so far (once the greedy daemon found a path of length 3), each move consumed $18/3=6$ of the initial potential; we denote by s_ϕ this quantity. By choosing the configuration which maximizes the *promise* computed by $d+\phi/s_\phi$, we can hope to do better than a simple depth-first-search and find better solutions first. We now detail the behavior of this heuristic on the $\mathscr{R}(A,T,c_{init})$ of Fig. 1.

1. At the beginning, from c_{init}, we need to consider configurations that all have the same depth (c_1, c_2, c_3, c_4, c_5); the one with the highest promise is therefore the one with the highest potential, c_5.
2. c_7 and c_8 are thus added to the set of configurations to be considered (c_6 has already been visited during the initial greedy traversal), but their promises ($2+2/6=2.33$) are lower than the promise of c_2 ($1+14/6=3.34$).
3. c_2 is therefore selected, which adds c_9, c_{10}, c_{11} in the configurations set to be explored.
4. c_{11} has a promise of $2+11/6=3.83$, and is thus preferred over c_4, which has a promise of $1+13/6=3.17$.
5. Then c_{14} ($3+9/6=4.5$) and c_{15} ($4+4/6=4.67$) are selected, a new path of length 5 is found and s_ϕ is updated.

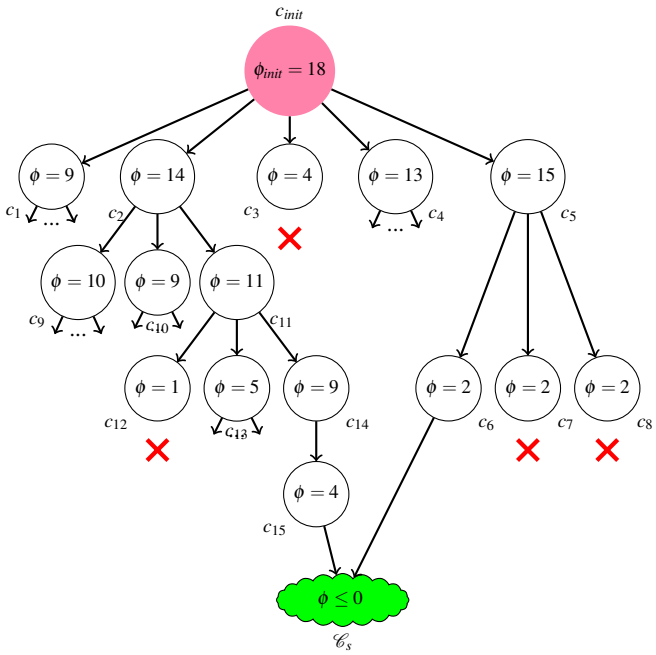

Fig. 1: Selected nodes in the graph $\mathscr{R}(A,T,c_{init})$

At this stage, all configurations for which the sum of the depth and the potential is smaller or equal than 5 can be cut (*cf.* red crosses in Fig. 1).

This algorithm is an heuristic in the sense that it sometimes finds the worst-case faster, but the exploration remains exhaustive as only branches that cannot lead to a worst-case are cut. Another exploration heuristics would have been to select configurations according to the sum of their depth and their potential. But using such a heuristic would delay the discovering of new longest paths (in step 4 above, c_4 would have been chosen over c_{11}), which in turn would prevent to cut branches. More generally, favoring depth over breadth allows to find longer paths sooner, which allows to cut more branches sooner and speed up the exhaustive exploration – which make this idea interesting even without using timeouts.

When No Potential Function is Available. Finding a potential function can be challenging for some algorithms. But note that any function that is able to approximate accurately enough the distance between a configuration and the set of legitimate configurations could be used to guide the exploration of $\mathscr{R}(A,T,c_{init})$ with the heuristics described above. The result of an exhaustive exploration using such a *pseudo-potential function* should be interpreted with care using the optimization described so far since the actual best solution can be missed.

Benchmark Algorithms. Those ideas have been implemented in SASA. We propose to experiment them on the following set of algorithms:

1. `token` is the first token ring algorithm proposed by Dijkstra [15]. It stabilizes to a legitimate configuration from which a unique token circulates in the ring. We use the potential function given in [5].
2. `coloring` is a vertex coloring algorithm [7]. The potential function, proposed in [7], counts the number of conflicting nodes.
3. `te-a5sf` consists of the two last layers of the algorithm given in chapter 7 of [7] to illustrate some algorithm composition. It consists of a bottom-up computation followed by a top-down computation of the rooted tree. Its potential function is inspired from the general method proposed in [8].
4. `k-clust` computes sub-graphs (clusters) of radius at most k [12] in rooted trees. Its potential function is made of the sum of enabled node levels in the tree, as described in [4].
5. `st-algo1` computes a spanning tree (the first of the 2 algorithms proposed in [23]). Its potential function, also given in [23], consists of counting the number of correctly directed edges.
6. `unison` is a clock synchronization algorithm that stabilizes to a configuration from which clocks of neighboring nodes differ from at most one [10]. To the best of our knowledge, no potential function has been ever proposed for this algorithm. We use instead a pseudo-potential function that consists of counting the number of nodes that are not synchronized.

Simulations are performed on different topologies: directed rings (noted diring), random rooted trees (rtree), Erdős–Rényi random graphs (er), lines, grids and stars; in the sequel, the size of those graphs (in number of nodes) is noted aside the graph name. For example, diring5 denotes a directed ring with 5 nodes.

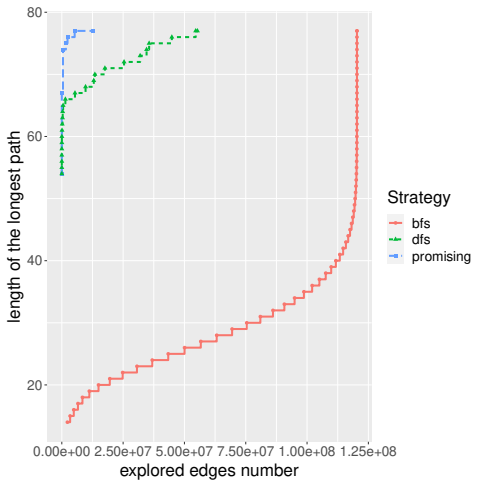

Fig. 2: Comparing exhaustive exploration strategies on `te-a5sf`/rtree5.

Finding Longest Paths First. The motivation for defining promising daemons is to find the longest paths as soon as possible during the exploration. In order to assess our design choices, we have conducted an experiment where, during a simulation of the `te-a5sf` algorithm on a rooted tree of size 5 under the promising daemon, we store the number of edges explored in $\mathscr{R}(A,T,c_{init})$ each time a new longest path is found. Fig. 2 shows the result of this experiment together with the result obtained with a Depth-First Search (DFS) and a Breadth-First Search (BFS) exploration using the same parameters. One can see on Fig. 2 that indeed, on this particular example at least, the promising heuristic is better than a DFS exploration: the longest paths are discovered sooner, which allows to cut more branches and leads to

less explored edges (less than 12 millions versus more than 55 millions for DFS and 120 millions for BFS) to perform the exhaustive exploration. Notice that it is just an heuristic, that sometimes gives better result, and sometimes doesn't.

Measuring the Effect of Branch Cuts and Tabu Lists. We ran the promising heuristic (values are similar with a Depth-First Search) with and without the optimizations of branch-cuts and tabu list.

Table 1 contains the results of the following experiments. We chose small enough topologies to get a result in a reasonable amount of time when computing without optimization (we use random rooted trees for `unison`, as for `k-clust` and `te-a5sf`). For a given algorithm and once the topology is fixed, an experiment consists of picking an initial configuration uniformly at random and running a simulation on it with four different sets of options. Column 2 contains the number of edges explored during the simulation using a promising daemon with no optimization at all. Column 3 contains the gain factor compared to the values of Column 2 using the branch-cuts *and* the tabu list optimizations. Column 4 (resp. 5) contains the same gain factor but using only the branch-cuts (resp. the tabu list) optimization. Each number in this table is the average of x experiments that were performed with different initial configurations picked at random. It is given (at the right-hand-side of the $\pm$ symbol) with the bounds of the corresponding confidence interval at 95% ($1.96 \times \sigma/\sqrt{x}$, where σ is the standard deviation). In order to get a tight enough interval, experiments were repeated from $x = 200$ to $x = 100\,000$ times.

Table 1: Measuring the effect of branch-cuts and tabu list.

algo/topology	no optimization edges number	cut + tabu gain factor	cut gain factor	tabu gain factor
`token`/diring5	$1\,400 \pm 100$	11 ± 0.7	11 ± 0.7	6 ± 0.3
`token`/diring6	$4 \cdot 10^6 \pm 2 \cdot 10^6$	$8\,500 \pm 4\,000$	$4\,700 \pm 3\,000$	$2\,200 \pm 800$
`k-clust`/rtree5	380 ± 30	5.4 ± 0.3	2.9 ± 0.1	4 ± 0.2
`k-clust`/rtree6	$2\,900 \pm 70$	18 ± 0.4	5.6 ± 0.2	12 ± 0.2
`k-clust`/rtree7	$2 \cdot 10^4 \pm 2 \cdot 10^3$	58 ± 5	7 ± 0.8	39 ± 3
`k-clust`/rtree8	$7 \cdot 10^5 \pm 8 \cdot 10^4$	850 ± 90	59 ± 20	400 ± 30
`te-a5sf`/rtree3	$2\,800 \pm 7$	12 ± 0.02	3 ± 0	10 ± 0.01
`te-a5sf`/rtree4	$6 \cdot 10^6 \pm 6 \cdot 10^4$	$2\,000 \pm 10$	6.2 ± 0.05	$1\,800 \pm 10$
`unison`/rtree3	12 ± 0.1	1.2 ± 0	1.2 ± 0	1.1 ± 0
`unison`/rtree4	$3\,900 \pm 100$	76 ± 2	73 ± 2	13 ± 0.3
`unison`/rtree5	$4 \cdot 10^7 \pm 1 \cdot 10^7$	$3 \cdot 10^5 \pm 2 \cdot 10^5$	$3 \cdot 10^5 \pm 2 \cdot 10^5$	$1 \cdot 10^4 \pm 7 \cdot 10^3$

Table 1 shows that, on those algorithms and topologies, the optimization gain factor grows exponentially with the topology size. It also shows that the two optimizations are complementary in the sense that their effects are cumulative.

Note that we do not show any result for `coloring` nor `st-algo1` since their potential functions are perfect which makes promising exploration useless.

Daemons Comparison. Given an algorithm and a topology with a particular initial configuration, the simplest way to search for the longest path is to perform several

simulations using a random daemon. In order to assess the idea of using more elaborated methods based on a potential function and use greedy or promising daemons, we ran another set of simulations. Note that, as for the random daemon, we have performed several runs of the greedy one since it also has a random behavior with SASA: indeed, when several choices lead to the same potential, one is chosen uniformly at random. On the contrary, the promising daemon only needs to be run once.

The results obtained with those three kinds of daemons are provided in Table 2. This table is obtained, for each algorithm and topology, by repeating 1 000 times the following experiment:

1. choose an initial configuration I;
2. run once the promising daemon on I and report the resulting move number in column 2;
3. run n_1 (resp. n_2) times the algorithm on I with a greedy (resp. random) daemon and report the maximum move number in column 3 (resp. column 4). The numbers of simulations n_1 and n_2 are chosen in such a way that the same amount of CPU time budget is used for the three daemons.

Table 2 the shows average values for those 1 000 experiments, as well as the corresponding 95% confidence interval bounds. The last column indicates the total wall-clock simulation time.

Table 2: The longest path obtained with different daemons.

algo/topology	Promising	Greedy	Random	time
`coloring`/er11	4.6 ± 0.1	4.6 ± 0.1	4.2 ± 0.08	$28m$
`st-algo1`/er20	24 ± 0.4	24 ± 0.4	23 ± 0.4	$1h$
`k-clust`/rtree14	24 ± 0.3	22 ± 0.3	18 ± 0.2	$3h$
`token`/diring12	69 ± 0.8	66 ± 0.8	28 ± 0.4	$47m$
`te-a5sf`/rtree5	32 ± 0.07	30 ± 0.05	22 ± 0.1	$1h$

The potential functions of `coloring` and `st-algo1` being perfect, the result is the same for greedy and promising daemons. Moreover, they do not significantly improve the result of the random daemon.

For `token`, `k-clust`, and `te-a5sf`, greedy daemons give better results than random ones, but fail to find the best solution given by the promising daemon.

The case of `unison` and its pseudo-potential function requires a special attention. Indeed, as previously noticed, the promising daemon with the branch-cut optimization is not exhaustive. This is why the results of the experimentation of this algorithm is in a different table (Table 3), which has an extra column (Column 2) that contains the result of the promising daemon run without this optimization. We can observe that the promising daemon finds much better solutions than the greedy daemon, provided that the cut optimization is inhibited. The greedy daemon is itself much better than the random daemon on most topologies. Note that the greedy daemon sometimes does better than the promising daemon (*e.g.,* on `grid16`), which seems counter-intuitive as the promising daemon starts its exploration by a greedy search. This can be explained since greedy daemons have randomness, and in Column 4, each experiment consists of several (n_1) simulations.

Table 3: The longest path obtained with different daemons for `unison`.

algo/topology	Promising no cut	Promising	Greedy	Random	time
`unison`/ring10	42 ± 2	27 ± 0.3	26 ± 0.2	18 ± 0.4	$19m$
`unison`/er11	83 ± 4	35 ± 0.6	39 ± 0.6	22 ± 0.5	$6h$
`unison`/grid16	59 ± 1	37 ± 0.5	40 ± 0.3	29 ± 0.4	$29h$
`unison`/chain10	75 ± 2	36 ± 0.4	37 ± 0.4	19 ± 0.3	$3h$
`unison`/rtree10	55 ± 3	27 ± 0.6	24 ± 0.7	19 ± 0.4	$7h$
`unison`/star8	23 ± 0.6	15 ± 0.3	16 ± 0.4	14 ± 0.2	$4h$

3 Exploring Initial Configurations

For a given topology, the stabilization time is also impacted by the choice of the initial configuration. Here again, simulations can help to experimentally figure out the worst case, and check whether the known bounds are tight.

3.1 Assessing Initial Configurations

Given an initial configuration I, a way to evaluate its ability to lead to worst case stabilization time is to exhaustively explore $\mathscr{R}(A,T,c_I)$ and seek for the longest path. Of course this is costly[4], in particular if we want to do that on a large number of configurations. Using a greedy or a random daemon to approximate this configuration evaluation would be cheaper, but how could we know if a *good* configuration, that leads to a long path, for a random or a greedy daemon, is also a good configuration w.r.t. an exhaustive exploration? In other words, are the results of those different evaluations methods correlated? In order to get some insights on this question, we study this hypothesis experimentally, by running simulations using those three different daemons, and looking at the resulting stabilization time.

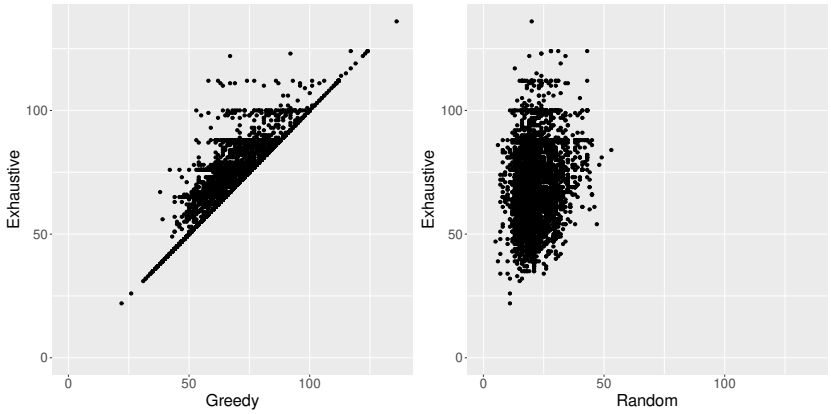

Fig. 3: Comparing configuration evaluation methods by running simulations of `token` on a directed ring of size 12.

[4] Even using the optimizations of Section 2.

Fig. 3 shows the result of such an experiment on the `token` algorithm over a directed ring of size 12. Each of the 1 000 points in both graphics is computed by running a simulation from a different random configuration; its abscissa corresponds to the stabilization time (in moves) obtained using a greedy (left) and a random (right) daemon; its ordinate corresponds to the stabilization time obtained using a promising (and thus exhaustive) daemon. One can notice that on this set of simulations, the worst case obtained by the greedy daemon (rightmost point of the left-hand-side graphics) is also the worst case for the exhaustive daemon (topmost point), whereas it is not the case for the worst case of the random daemon (*cf.* the rightmost point of the right-hand-side graphics). In order to synthesize this analysis numerically, we propose the following experiment. Given an algorithm A and a topology T:

- choose n initial configurations of the system at random;
- for each configuration j, measure (by simulations) the stabilization time R_j, G_j, and E_j using respectively a random, a greedy and an exhaustive daemon;
- let j_r, j_g, j_e be the index of the maximum of the sets $\{R_j\}_j$, $\{G_j\}_j$, and $\{E_j\}_j$ respectively; compute $\tau(R,E) = E_{j_r}/E_{j_e}$ and $\tau(G,E) = E_{j_g}/E_{j_e}$ to estimate the loss ratio that one gets by approximating the exhaustive daemon based evaluation of a configuration by a random and a greedy daemon.

Table 4: Measuring the correlation between worst cases obtained with various daemons

algo/topology	$\tau(G,E)$	$\tau(R,E)$
`te-a5sf`/rtree5	0.91	0.91
`k-clust`/rtree10	1.00	0.73
`token`/diring12	1.00	0.62
`unison`/er20	0.85	0.53
`unison`/ring20	1.00	0.57
`unison`/rtree20	0.99	0.47

Table 4 shows the result of such an experiment performed with 10 000 random initial configurations. We did not use the `coloring` nor the `st-algo1`, as greedy and exhaustive (with perfect ϕ) daemons produce the same results. One can observe that the worst cases obtained with exhaustive daemon-based evaluation and the greedy daemon-based one are indeed very well correlated. It is therefore interesting to approximate exhaustive daemons with greedy ones during the search of worst-case initial configuration.

When no potential function is available, using random daemons may be a stopgap, even if the correlation between random and exhaustive daemons is weaker. The exhaustive exploration of the daemons state space can then always be done, but without the branch-cut optimization. Smaller topologies should thus be considered.

3.2 Local Search

We can define a notion of distance between two configurations, for instance by counting the number of node states that differ (Hamming distance), and by which amount

they differ (using a user-defined function). One practical question to find efficiently worst-case configurations is then the following: do close configurations lead to similar stabilization time in moves? If the answer is yes, it means that it should be interesting to perform a so-called *local search*, which consists of:

1. choosing a configuration I;
2. choosing a configuration I' that is close to I;
3. evaluating I by running a simulation (using, *e.g.*, a greedy daemon);
4. continuing in step 1 with the configuration I' if it leads to a higher stabilization time than I, and with I otherwise.

It is difficult to answer to such a question in the general case, as the answer might differ on the algorithm or on the topology. Once again, simulations can be useful to get insights.

SASA implements the local search described above. The SASA API requires the user to define a distance between two states, so that SASA can compute the configuration neighborhood. In order to take advantage of multi-core architectures, SASA tries several neighbors at each step, and puts them into a priority queue. The number of elements that are tried at each step depends on the number of cores that are used. The priority in the queue is computed by launching a simulation from the corresponding initial configuration (and any daemon available in SASA can be used).

An Experiment to Compare Global and Local Search. In order to assess the idea of using local search to speed up the discovery of worst-case initial configurations, we ran another experiment on a set of algorithms and topologies. For each experiment, we use 10 000 initial configurations. Each experiment has been repeated between 100 and 1 000 times, in order to obtain 95% confidence intervals (at the right-hand-side of the ± sign) that are small enough.

Table 5: Comparing global and local searches of the initial configuration: number of moves [simulation index]

algo/topology	global		local	
`coloring`/er20	17 ± 0.1	$[3\,200 \pm 500]$	17 ± 0.2	$[1\,100 \pm 100]$
`token`/diring12	130 ± 0.6	$[4\,400 \pm 300]$	140 ± 1	$[3\,800 \pm 300]$
`k-clust`/rtree10	25 ± 0.2	$[3\,700 \pm 600]$	26 ± 0.1	$[1\,900 \pm 400]$
`st-algo1`/er20	44 ± 0.3	$[4\,200 \pm 500]$	60 ± 0.03	$[1\,600 \pm 100]$
`te-a5sf`/rtree5	31 ± 0	$[100 \pm 20]$	31 ± 0.02	$[330 \pm 100]$
`unison`/er20	99 ± 0.6	$[4\,800 \pm 200]$	150 ± 3	$[3\,800 \pm 200]$

The result of those experiments is provided in Table 5. The second column contains moves numbers obtained by taking the maximum number of moves (the bigger, the better) among the ones obtained by running a greedy daemon on 10 000 random initial configurations. The values between square brackets correspond to the simulation indices $j \in [1, 10\,000]$ where the worst cases occur, in average (the smaller, the better). The third column contains the same information as the second one, except that configurations are chosen via a local search, as described above.

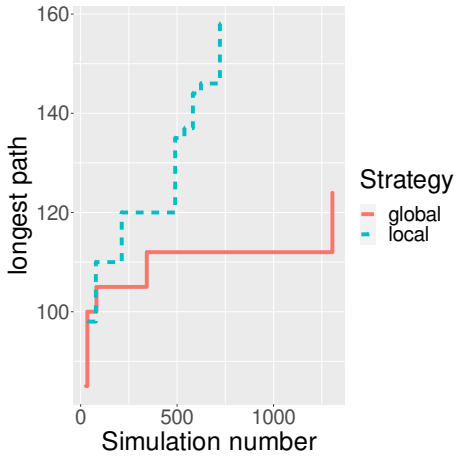

Fig. 4: One of the simulations performed for generating Table 5: `token`/diring12.

For `token`/diring12, `st-algo1`/er20, and `unison`/er20, we can observe in Table 5 that the local search is better, as we obtain better worst cases using less initial configurations. For `coloring`/er20 and `k-clust`/rtree10, the resulting worst cases are similar, but they are obtained quicker. For `te-a5sf`/rtree5 on the other hand, the global search is slightly better.

Fig. 4 details those results on one of the (thousands of) experiments that were run to compute the values in Table 5 in the particular case of `token`/diring12. The more we try initial configurations (in abscissa) the longer path we find (in ordinate). On this figure, we can see the local search approach winning on both counts: higher worst cases that are found using less initial configurations.

4 Conclusion

In this paper, we present a methodology based on heuristics and an open-source tool [9] to find or refine worst-case stabilization times of self-stabilizing algorithms implemented in the atomic-state model using simulations.

We show how potential functions, designed for proving algorithm termination, can also be used to improve simulation worst cases, using greedy or exhaustive explorations of daemon behaviors. We propose a heuristic to speed up the exhaustive exploration and to potentially find the best solution early in the exploration process, which is a desirable property when timeouts are used. We experimentally show several results of practical interests.

- When a potential function is available, it can significantly speed up the search of the worst case.
- When no potential function is available, the use of a pseudo-potential function can still enhance the worst-case search.
- Local search can speed up the search for worst-case initial configurations.
- The worst cases obtained by greedy daemons are correlated (on the algorithms and topologies we tried) to the ones of (more costly) exhaustive daemons. This means

that we can use greedy daemons to search for worst-case initial configurations, and then use an exhaustive daemon only on the resulting configuration.
- The same result can be observed, to a lesser extent, for random daemons. This is interesting as it allows to apply this idea on algorithms that have no known potential (nor pseudo-potential) function.

Future Work. On the algorithms and topologies we have considered, local searches are always better than global (*i.e.*, fully random) searches, except for the `te-a5sf`/rtree4, where the same worst case is found, but a little bit faster. Cases were global searches give better results certainly exist, if the local search starts in a configuration that is far from the ones that produce the worst cases. In such a case, it is easy to combine both heuristics, and to start the local search with the best result found by the global one. Another classical heuristic to combine local and global search would be to perform simulated-annealing.

References

1. Jordan Adamek, Giovanni Farina, Mikhail Nesterenko, and Sébastien Tixeuil. Evaluating and optimizing stabilizing dining philosophers. *Journ. of Parallel and Distributed Computing*, 109, 2017.
2. Jordan Adamek, Mikhail Nesterenko, and Sébastien Tixeuil. Evaluating practical tolerance properties of stabilizing programs through simulation: The case of propagation of information with feedback. In *Symp. on Stabilization, Safety, and Security of Distributed Systems*, volume 7596, 2012.
3. Saba Aflaki, Borzoo Bonakdarpour, and Sébastien Tixeuil. Automated analysis of impact of scheduling on performance of self-stabilizing protocols. In *Symp. on Stabilization, Safety, and Security of Distributed Systems*, 2015.
4. Karine Altisen, Pierre Corbineau, and Stéphane Devismes. A framework for certified self-stabilization. *Log. Methods Comput. Sci.*, 13(4), 2017.
5. Karine Altisen, Pierre Corbineau, and Stéphane Devismes. Certification of an exact worst-case self-stabilization time. *Theoretical Computer Science*, 941, 2023.
6. Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, and Franck Petit. Self-Stabilizing Leader Election in Polynomial Steps. *Information and Computation*, 254, Part 3, 2017.
7. Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. *Introduction to Distributed Self-Stabilizing Algorithms*, volume 8 of *Synthesis Lectures on Distributed Computing Theory*. 2019.
8. Karine Altisen, Stéphane Devismes, and Anaïs Durand. Acyclic strategy for silent self-stabilization in spanning forests. In *Symp. on Stabilization, Safety, and Security of Distributed Systems*, volume 11201, 2018.
9. Karine Altisen, Stéphane Devismes, and Erwan Jahier. sasa: a SimulAtor of Self-stabilizing Algorithms. *The Computer Journal*, 2022.
10. Jean-Michel Couvreur, Nissim Francez, and Mohamed G. Gouda. Asynchronous unison (extended abstract). In *ICDCS'92*, 1992.
11. Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. An o(n)-time self-stabilizing leader election algorithm. *Journ. of Parallel and Distributed Computing*, 71(11), 2011.

12. Ajoy Kumar Datta, Stéphane Devismes, Karel Heurtefeux, Lawrence L. Larmore, and Yvan Rivierre. Competitive self-stabilizing k-clustering. *Theoretical Computer Science*, 626, 2016.
13. Ajoy Kumar Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theoretical Computer Science*, 412(40), 2011.
14. Stéphane Devismes and Colette Johnen. Silent self-stabilizing {BFS} tree algorithms revisited. *Journ. of Parallel and Distributed Computing*, 97, 2016.
15. Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11), 1974.
16. Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory Requirements for Silent Stabilization. *Acta Informatica*, 36(6), 1999.
17. Huseyin Tolga Evcimen, Ozkan Arapoglu, and Orhan Dagdeviren. Selfsim: A discrete-event simulator for distributed self-stabilizing algorithms. In *International Conf. on Artificial Intelligence and Data Processing*, 2018.
18. Mitchell Flatebo and Ajoy Kumar Datta. Simulation of self-stabilizing algorithms in distributed systems. In *Annual Simulation Symposium*, 1992.
19. Christian Glacet, Nicolas Hanusse, David Ilcinkas, and Colette Johnen. Disconnected components detection and rooted shortest-path tree maintenance in networks. In *Symp. on Stabilization, Safety, and Security of Distributed Systems*, 2014.
20. Christian Glacet, Nicolas Hanusse, David Ilcinkas, and Colette Johnen. Disconnected components detection and rooted shortest-path tree maintenance in networks. *Journ. of Parallel and Distributed Computing*, 2019.
21. Oded Har-Tal. A simulator for self-stabilizing distributed algorithms. https://www.cs.bgu.ac.il/~projects/projects/odedha/html/, 2000.
22. Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41(2), 1992.
23. Adrian Kosowski and Lukasz Kuszner. A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves. In *Parallel Processing and Applied Mathematics, 6th International Conf., PPAM*, volume 3911, 2005.
24. Nils Müllner, Abhishek Dhama, and Oliver E. Theel. Derivation of fault tolerance measures of self-stabilizing algorithms by simulation. In *Annual Simulation Symposium*, 2008.
25. Kishor S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications, Second Edition*. Wiley, 2002.