http://www.aimspress.com/journal/mine

Mathematics in Engineering, 6(1): 28–44.
DOI:10.3934/mine.2024002
Received: 01 November 2022
Revised: 03 November 2023
Accepted: 08 January 2024
Published: 11 January 2024

Research article

A graph-based framework for complex system simulating and diagnosis with automatic reconfiguration

Martina Teruzzi, Nicola Demo and Gianluigi Rozza*

Mathematics Area, mathLab, SISSA, via Bonomea 265, I-34136 Trieste, Italy

* **Correspondence:** Email: gianluigi.rozza@sissa.it.

Abstract: In this work we present a novel approach for modeling complex industrial plants, employing directed graphs to the simulation and automatic reconfiguration after failures. The framework offers the possibility to model the failure propagation, estimating the overall condition of the system before and after the damage and exploit such a health index for dynamic recalibration. To model the typical operation of industrial plants, we propose several additions with respect to the standard graphs: *i*) a quantitative measure to control the overall condition of the system *ii*) nodes of different categories–and then different behaviors–and *iii*) a fault propagation procedure based on the predecessors and the redundancy of the system. The obtained graph is able to mimic the behavior of the real target plant when one or more faults occur. Additionally, we also implement a generative approach capable of activating a particular category of nodes in order to contain the issue propagation, equipping the network with the capability of reconfiguring itself and resulting in a mathematical tool useful not only for simulating and monitoring but also to design and optimize complex plants. The final asset of the system is provided in the output with its complete diagnostics and a detailed description of the steps that have been carried out to obtain the final realization.

Keywords: risk-analysis; graph-theory; fault-diagnosis; automatic reconfiguration; industrial applications; signed-directed-graphs

1. Introduction and motivations

In many operative and industrial fields dealing with complex systems, one of the fundamental aspects during the life cycle of a product is the individuation of faults. A quick detection of any generic fault could lead to avoiding destructive situations and to a dynamical re-calibration of the system, minimizing the impact of the failures. For this reason, a lot of effort has been spent in its modeling and diagnostics [1–3]. Graphs have met the necessity to model precisely process plants, leading the path to the creation of ad-hoc software [4,5]. Moreover, the qualitative paradigm of signed

directed graphs has been enriched over the years with high-resolution techniques, able to provide a quantitative description [6–8].

Besides a precise description of fault consequences, the ability to simulate these kinds of fatal events creates the opportunity for real-time intervention, aimed at an automatic reconfiguration of the system with its largest residual operational capacity.

In this work, we will present a screening tool we have developed within this framework, able to mimic different kinds of perturbations in interconnected systems, such as industrial plants. The dependencies between the network elements is indeed depicted employing graph theory, such that the results obtained from the graph analysis can therefore be used to improve the robustness and resilience profile of industrial facilities against domino effect propagation. Not only: we coupled the graph model with a custom fault propagation procedure–based on logical relations between the components– to provide a simplified *digital twin* of the studied plant that behaves like the physical one. This finally allowed us to employ a genetic procedure that, controlling the active propagation resistances, is able to dynamically change such resistance states in order to limit the effect of a generic fault.

All the features described in the present work have been implemented in SAFEX software, within project SAFE, "Realtime Damage Manager And Decision Support". An open-source restricted version of the software can be found in GRAPE (GRAph Parallel Environment) software package [9–11].

The paper is organized as follows: in Section 2, after an initial digression on graph definitions, we present the features we have introduced for the failure propagation and the dynamic optimization. We highlight that, for the sake of clarity, we provide in the same section a toy problem to show the purpose of the new additions. In Section 3 we finally apply the proposed framework to an industrial plant and, to have more robust results from a statistical perspective, on several random graphs. The document ends with Section 4, where we summarize the content of the manuscript and highlight the future continuation of this work.

2. Graph theory for modeling complex systems

Graph theory is a well-established mathematical framework. Its origins date back to Euler in 1736 [12]; from the very beginning its rigorous formalization has been entangled with scientific and technological development in various fields, including mathematics [13, 14], chemistry [15], physics and engineering [16]. This section is devoted to introduce the methodology we have applied for the modeling of complex systems: initially we provide a formal overview about graph theory, defining the main features of a generic graph, to then focus about the introduced characteristics and the genetic calibration of the graph itself. A graph can be defined as a general representation of a structure describing relations between objects. For its generality, it is well-suited for the description of very diverse structures at many different scales [17–19]. In the current application, graphs have been employed in the description of interconnected systems, such as industrial plants.

We define a graph G as formed by a set of nodes N and a set of edges E; each edge connects two nodes. If a *weight* is assigned to each edge (usually a positive integer value), the graph is called *weighted*; *unweighted*, otherwise. A directed graph (or *digraph*) is a graph whose edges have a direction, with one node as head and the other as tail. The graph is called *sparse* if $|E| \propto |N|$, while it is called dense if $|E| \propto |N|^2$. The length of a *walk* (intended as a succession of alternating nodes and edges) between two nodes u and v is given by the number of the edges traversed, if the graph is

unweighted. Otherwise, the length is given by the sum of the edge weights. A node v is said to be reachable from a node u if there exists at least one walk from u to v. A walk with no repeated nodes is called a *path*. The *distance* between two nodes u and v is the length of the shortest walk containing them. Within the current application, we are always going to address weighted directed graphs.

2.1. Model industrial plants with graphs

The proposed methodology aims to model a generic plant through a graph: the components that compose such plant–e.g., a water pump, a valve–are indeed modeled as nodes, while the connections between such objects–the reader can think of cables and wires–are simulated through the graph edges. In order to provide a more realistic description of how commodities flow within an industrial plant, the graphs have been equipped with weighted edges. Weight can be very useful in specifying preferable itineraries for the service stream, with respect to areas in which its course can be less favored.

A shortest path is defined as the walk between two vertices that minimizes the sum of the weights of the edges traversed. An example is given in Figure 1. Shortest-path calculation plays a crucial role in graph analysis. It is necessary in order to be able to compute a series of quantities, known as efficiency and centrality measures, for system diagnostics.

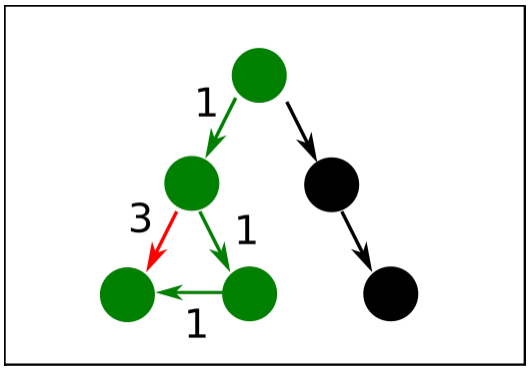


Figure 1. Shortest Path. In this weighted digraph $G(N, E)$ the shortest path is indicated in green, with respect to the red path, which implies a larger weighted distance between the nodes.

For weighted graphs we have implemented two algorithms for shortest-path calculation: Dijkstra's method [20] for sparse graphs and Floyd-Warshall [21–23] for dense graphs.

Dijkstra's algorithm assigns some initial distance values, improving them step by step. It starts with marking all the nodes as unvisited, assigning to every node a tentative distance value, which is set to zero for the initial node (that will be labeled as current), while infinity for all other nodes. The search proceeds between all the unvisited neighbors of the current node, calculating their tentative distance through it, and updating if smaller than the previous one. When all its neighbors have been visited, the current node is marked as visited and never checked again. For all-pairs distance (the distance computed over all possible nodes) this algorithm is of order $O(|N|(|E| + |N|log(|N|)))$.

Floyd-Warshall algorithm, instead, involves the comparison of all the possible routes between a pair of vertices i and j. If the distance is smaller than the previous one, distance and predecessor matrices get updated. A final path reconstruction returns the shortest paths for the graph.

From shortest paths, it is possible to extract the path *efficiency*, which is given by the inverse of its length, when different from zero, and zero otherwise. We recall that, for weighted graphs, the path length does not depend on the number of edges traversed, but on the sum of their weights. From this local measure, further measures can be obtained, providing a more global view. Our implementation also provides *centrality* measures. These quantities are more related to how well-connected a node is, and their comparison before and after a fault can be a useful hint on how the situation has changed due to the perturbation. Shortest-path and measures calculations are repeated after the propagation of damage, and their description for the integer and damaged graph represents an important part of the system diagnosis. More details about efficiencies and centralities can be found in [9].

Such measures are surely very important to analyze and detect possible criticalities but are not sufficient to express the propagation of a certain failure inside the graph or to estimate its health condition. The next sections highlight the additions we implemented in order to have a dynamic model of the entire plant. The basic rule is that any node of the plant that is affected by damage propagates the failure to its successor nodes before becoming a *dead* node (not working) and being removed by the graph.

2.2. Node types and residual service

In modeling complex systems, it becomes fundamental the estimate the health condition, especially before and after damage occurs, in order to quantify the condition decrease. In addition to the aforementioned measures, we then introduce a feature representing the system's residual capacity. We refer to this measure for the rest of the article as residual service, or just *service*. To our knowledge, no similar property has been implemented before for the flow of commodities.

The importance of service, together with its intrinsic relation with fault propagation, has an immediate effect on the graph characterization, acting as a sort of health index of the system. Mimicking the commodities flow, the basic idea is to have a limited set of nodes that provide *service* to the entire system–i.e., a real positive quantity–a set of nodes that needs a certain amount of service to stay active and working. All the rest of the nodes are in charge of propagating the service to their neighbors. In order to implement the computation of the *residual service* of the system, we classify all the nodes into several categories: Nodes have been labeled regarding their role when service is taken into account:

- $SOURCE$ nodes provide service to the network, acting as a source for the entire system. Some examples are water tanks or electric batteries since they enter water or electricity.
- $USER$ nodes need a certain service to stay alive. Electric devices that use the electric current are concrete examples of this type of node.
- HUB nodes represent the rest, and they usually constitute the paths going from $SOURCE$ to $USER$.

The total service of the whole plant is then computed as the sum of the residual service in all the $SOURCE$ nodes.

It is important to highlight that $SOURCE$ and $USER$ nodes can be seen not just as components, but also as the input and output coming from other sub-systems. In this way, we can connect different sub-structures, which exchange information using service. It is also important to specify that for each $SOURCE$, the service has to be split between all surviving $USER$ nodes connected to it. If some

USER nodes are broken due to a perturbation, the residual service is re-computed taking just the available ones into account.

2.3. Edge types and fault propagation

In addition to the definition of service, we add the possibility to specify three different logical relations between nodes connected by an edge (see Figure 2), allowing for modeling a larger variety of possible systems.

- *SINGLE* edge connects a node to its unique predecessor.
- *AND* edge reports the fact that a node has more than one predecessor: all the predecessors are necessary for the functioning of that component.
- *OR* edge also reports the fact that a node has more than one predecessor: however, in this case, just one of the predecessors should be active to guarantee the functioning of that component.

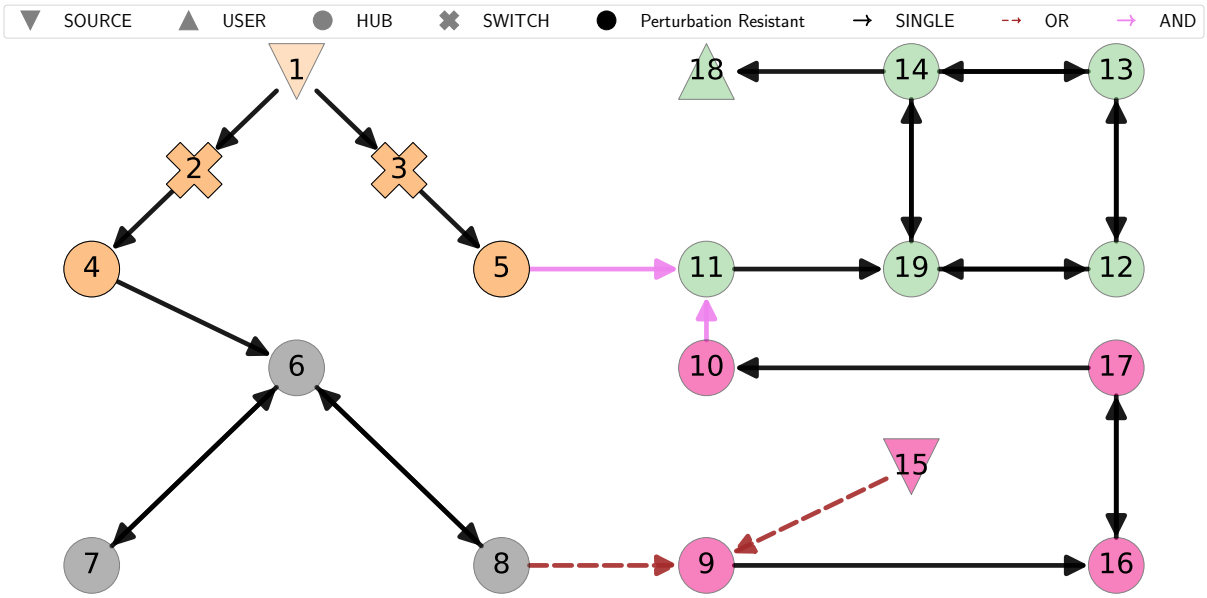


Figure 2. Example graph. The four different colors identify different areas. Nodes are labeled as *SOURCE* (reverse triangle), *HUB* (circle), or *USER* (up triangle), while the edges can describe *OR*, *SINGLE*, or *AND* relations between linked nodes. *SWITCH* nodes (cross) are a particular type of *HUB* nodes. Nodes with no transparency are perturbation-resistant nodes.

Finally, if a node has no predecessor it is labeled in the input as *ORPHAN*. The perturbation of any damage is propagated using the graph connectivity enriched by these logical relationships. Once the node is perturbed by a fault, after propagating it to its successors, is removed by the graph, marking the related component as not working. Regarding the behavior of the graph dealing with a generic fault, we introduce two kinds of perturbation resistances that can confine the cascade effects. The first one is passive resistance, for nodes that are not affected by the perturbation. This kind of resistance is intrinsic to the component itself (one example can be a fire-proof element within a plant). The second

resistance is called active, implying the voluntary activation of some graph elements in order to block the failure propagation. We introduce a new type of nodes, the $SWITCH$ ones, a subset of the HUB's that have a binary state. For a concrete example, the reader can think of a hydraulic valve or electric switch. Their state can be changed in order to stop the failure propagation. By convention in SAFEX we set a $SWITCH$ to *True* if it allows the flow of commodities (and, in this way, also the propagation of damage), like in a closed circuit. Vice-versa, a $SWITCH$ is set to *False*.

2.4. Dynamic recalibration after damage: a genetic approach

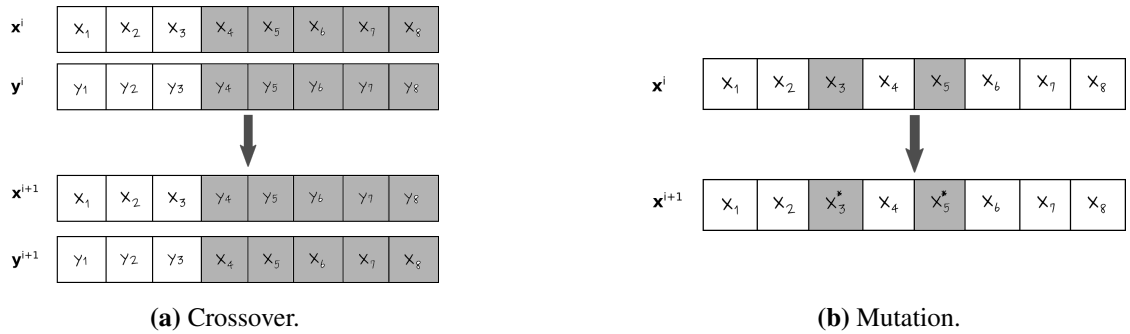
The example graph in Figure 2 shows just two nodes labeled as $SWITCH$. For this simple case, we can list all the possible configurations that they can assume very easily. Increasing the number of $SWITCH$ nodes, the number of possible realizations increases exponentially. Therefore, determining the best configuration for a certain simulated fault is not straightforward.

In order to solve this problem in general, in SAFEX we employ a generative approach. Genetic algorithms date back to 1973 [24, 25], and they have had a strong development within the years [26–30], still maintaining their basic concept, which we are going to summarize here to help a better understanding.

These kinds of algorithms generate a population of individuals with random genes and make them evolve mimicking Darwin's theory of evolution. The main steps are *selection*, *crossover*, and *mutation*.

We can define a population as composed by N individuals $\mathbf{x}_i \in \mathbb{R}^P$ with P genes as $\mathbf{X} = \{\mathbf{x}_i, ..., \mathbf{x}_N\}$. We express the fitnesses of each individual using a scalar function $f : \mathbb{R}^P \to \mathbb{R}$. The first generation $\mathbf{X}^1$ is generated randomly, and for every individual in the population the respective fitness is computed: $y_i = f(\mathbf{x}_i)$ for $i = 1, ..., N$. Every iteration (called *generation*), implies the following steps:

- *selection*: the best individuals of the previous generation are chosen, according to their fitnesses, in order to form the new one. The number of individuals chosen and the approach can vary;
- *crossover*: the selected individuals are combined in pairs, with a certain probability. Mixing the genetic information of the two parents allows the creation of one or more new offspring, which are going to constitute the new generation $\mathbf{X}^{i+1}$. You can see a graphical sketch of this procedure in Figure 3a;
- *mutation*: one or more individuals evolve changing part of their genes, according to some probability. You can see a graphical sketch of this procedure in Figure 3b.



(a) Crossover. **(b)** Mutation.

Figure 3. Graphical sketch of crossover and mutation procedures.

The total number of generations is itself a parameter of the algorithm. Iterating over generations, it is possible to optimize the original population. In the actual implementation of the genetic algorithm in SAFEX for the optimization of $SWITCH$ nodes, we have taken advantage of DEAP (Distributed Evolutionary Algorithms in Python) library [31].

We still need to define what is an individual, and how fitness is evaluated for it, in the current framework. A *state* is defined as one of the possible realizations for $SWITCH$ nodes: hence, $\mathbf{x}_i$ is a list of $True$ and $False$ values, with dimension equal to the total number of these nodes. The set of SWITCH nodes is defined in the input. For each of the states, three quantities constitute the fitness:

- $n_{actions}$: the total number of "flips" necessary in order to turn the initial configuration (given in input) into the one represented by the individual. The best configuration is supposed to be the closest to the initial with the lowest fitness. Within a real industrial plant, whether the activation of a $SWITCH$ is manual or automatic, the quickest solution remains the one with the smallest intervention;
- S_{tot}: total final residual service, given by the sum of the residual service at all $USER$ nodes;
- n_{alive}: total number of survived nodes after the perturbation, for the considered configuration of $SWITCH$ nodes.

The weights for these three different quantities are set by the user; in any case, we want to minimize the first quantity, while maximizing the other two, so that the fitness reads:

$$y_i = f(\mathbf{x}_i) = w_1 \cdot n_{actions}(\mathbf{x}_i) - w_2 \cdot S_{tot}(\mathbf{x}_i) - w_3 \cdot n_{alive}(\mathbf{x}_i). \tag{2.1}$$

3. Numerical results

In this section, we show the numerical results we obtained by applying the described modeling technique to different test cases. We remark again that, concerning the implementation side, our framework has been implemented in SAFEX, which has been employed also to collect the results we are going to discuss. Initially, we provide a toy example to show the propagation of the failures and the residual service. Then, we focus on a demonstration of the activation of $SWITCH$ nodes on a small graph that emulates real-world plants, testing the introduced methodologies for simulating a perturbation. We have chosen a plant with limited dimension (therefore also the corresponding graph) to show in detail all the steps computed for propagating such perturbation, as well as a qualitative discussion about the outcome of the genetic re-calibration. In the end, we analyze larger graphs in order to quantitatively measure the effectiveness of the proposed framework. It must be said that this last test, since the randomness in the graphs generation, wants to explore the robustness of the proposed optimization on a large variety of possible scenarios. The graphs do not represent any existing real plant, but contain, by constructions, all the possible structures, estimating from a statistical perspective the effectiveness of the method.

3.1. Toy example

After having explained all the main characteristics of our graphs, in this section we are going to present one simple fault event, in order to give a picture of the propagation procedure. The analyzed graph is shown in Figure 2. Despite its reduced dimension, it contains all the elements introduced

in the previous section, allowing us to clearly emphasize their role in the procedure. We label all the nodes with integer numbers ({1..18}) in order to easily indicate them. The four colors of the nodes indicate the node attribute *area*, which is the room (spatial location) to which the corresponding component belongs. The $SOURCE$ nodes, represented by the reverse triangle, are the 1 and 15 and they are connected to the only $USER$, node 18 (up triangle). The other nodes are $SWITCH$ and HUB, respectively the cross and dot symbols in the figure. The opacity indicates the resistance to perturbation: in this example, the only node not resistant to perturbation is the 1. The edges instead indicate the logical relations: solid black for $SINGLE$, solid pink for AND (the node is alive if all the parent nodes are alive) or dashed for OR (the node is alive if any of the parent nodes is alive).

We consider the perturbation of one node, namely node 1. In Figure 4, on the left, the whole graph is shown, before and after the perturbation. Initially, the $SOURCE$ nodes (1, 15) provide the service $S_{node1} = 1$ and $S_{node15} = 2$, respectively, so that the total service at the user 18 is 3. When node 1 is perturbed, not possessing any resistance to the damage, it gets broken. $SWITCH$ nodes 2 and 3 are opened (e.g., set to *False*) so that failure propagation gets blocked and the damage is limited to node 1 only.

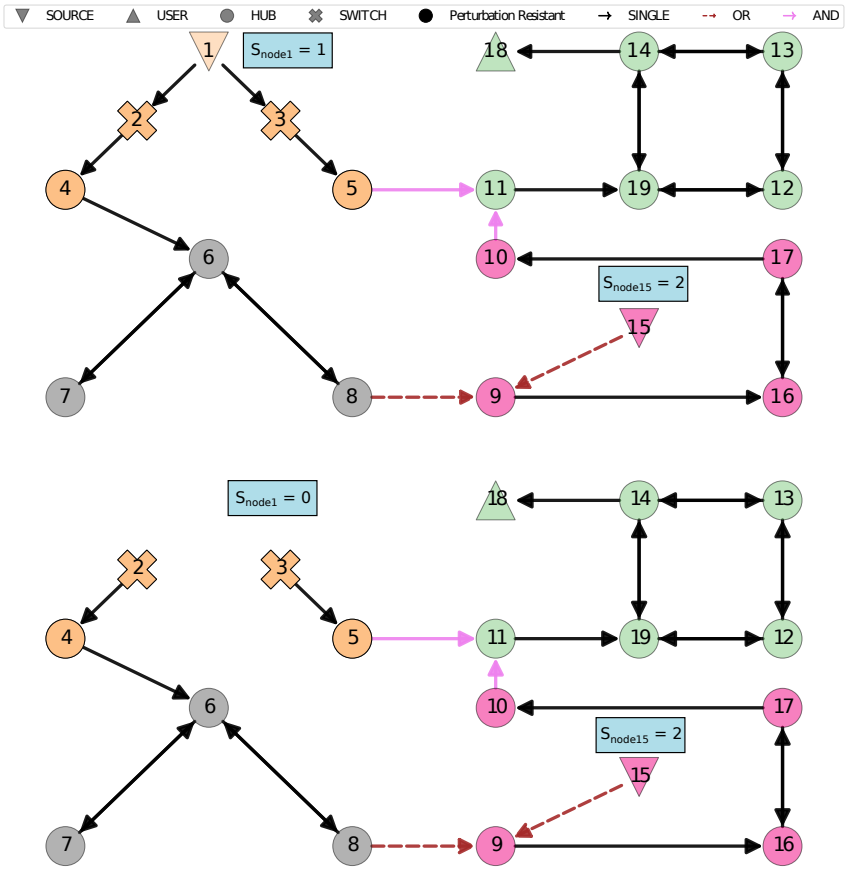


Figure 4. Example of a fault event, the damage of one node, namely $SOURCE$ 1. On top, is the graph referring to the healthy plant, while the damaged one is at the bottom. A damage is induced in the node 1. S_{node1} and S_{node15} represent the service provided by nodes 1 and 15, before and after the perturbation.

3.2. Switch line

The first system under study is a very good illustration of the mechanism for the activation of $SWITCH$ nodes. The graph in Figure 5 shows, in fact, a sequence of nodes interposed with $SWITCH$ nodes. The possible states are many: for this reason, we can clearly see the genetic algorithm described in Section 2.4 in action.

The graph in Figure 5 includes 25 elements connected by direct edges that reflect the hierarchy of the system in a parent-child fashion. The nodes are distributed in adjacent areas:

- $area1$ includes 6 nodes, namely A, 1, S_1, 2, S_2, and 10;
- $area2$ includes 5 nodes, namely 3, S_3, 4, S_4, and 11;
- $area3$ includes 6 nodes, namely B, 5, S_5, 6, S_6 and 12;
- $area4$ includes 6 nodes, namely C, 7, S_7, 8, S_8 and 13;
- $area5$ includes 2 nodes, namely 9 and 14.

In the graph in Figure 5, all edges are AND edges. This choice has been made in order to have the widest possible spread of any perturbation. A perturbation of one or multiple elements in one area may exceed the area boundaries and propagate to other components connected to it, located in other areas. Nodes 10, 11, 12, 13, and 14 are perturbation-resistant nodes, showing passive resistance. Hence, these nodes will not be affected by the simulated perturbation. Nodes S_1, S_2, S_3, S_4, S_5, S_6, S_7 and S_8 are $SWITCH$ isolating elements.

As stated previously, $SWITCH$ nodes activation can stop the propagation of a fault in a graph, avoiding a very dangerous cascade effect. They are characterized by two possible values: $True$ or $False$. For this example, we are going to set the genetic algorithm weights to unity ($w_1 = w_2 = w_3 = 1$). Moreover, we set the $SWITCH$ nodes all to $True$ as initial condition:

$$\mathbf{x}_{initial} = \{S_i = True, \quad \forall i = 1, ..., 8\}. \tag{3.1}$$

If all the three sources A, B and C in Figure 5 provide a service equal to 1, then for the initial configuration chosen each of the users obtains an amount of service equal to $3/5 = 0.6$.

The first perturbation we would like to analyze is the perturbation of node 1. Node 1 has no fault resistance of any kind; hence, it gets broken.

After that, the perturbation could in principle propagate to all nodes (if not limited); however, the genetic evolution identifies the following best state: all the switches but S_1 remain $True$ (hence, closed). This configuration has the following characteristics in terms of fitness: $n_{actions} = 1$ with respect to the initial condition, total final service $S_{tot} = 2$ ($SOURCE$ node A is no more connected to any of the $USER$ nodes), and $n_{alive} = 24$ nodes surviving to the perturbation. In conclusion, $y_{best} = -25$ is the fitness of the best configuration. Notice that setting a $SWITCH$ node to $False$ cuts off the edges between that node and its predecessors.

The second result for the switch line graph concerns the perturbation of node 2. This time, we perturb again a single node; however, this node is surrounded by more switches. For this reason, in order to isolate the broken node, we need to open two switches: both S_1 and S_2 get activated. The fitness of the best state can be computed $y_{best} = 2 - 3 - 24 = -25$. It is important to notice that opening S_1 and S_2 has indeed modified the flow in the graph. In fact, the final service is $S_{tot} = 3$, as for the initial configuration, but differently distributed between the $USER$ nodes with respect to the starting point. While initially all the $USER$ nodes were evenly receiving $1/5$ of the total service, in the final

situation node 10 is the only one still connected to $SOURCE\ A$, receiving final service 1, while the other four $USER$ nodes have all final service equal to 0.5, splitting evenly the service provided by B and C.

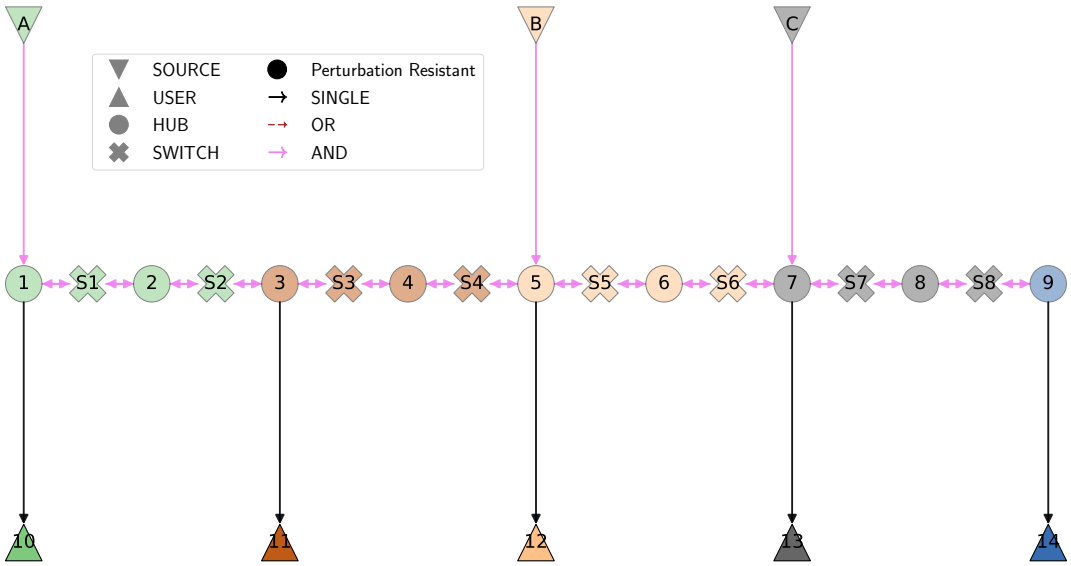


Figure 5. Switch line. This graph is divided into five areas, which are identified by the different colors. Nodes are labeled as $SOURCE$ (down triangle), HUB (circle), $SWITCH$ (cross), or $USER$ (up triangle), while the edges can describe OR, $SINGLE$, or AND relations between linked nodes. Nodes with no transparency are passive-resistant nodes.

As a final result for the switch line, we would like to perturb more than one node, namely nodes 2 and 3. This kind of perturbation is interesting because, differently from the previous one inspected here, if we set weights $w_1 = w_2 = w_3 = 1$, then two states have the same, lowest, fitness. The concerned states are the following:

$$x_{best}{}^1 = \{False, False, False, True, True, True, True, True\}. \tag{3.2}$$

$$x_{best}{}^2 = \{False, True, False, True, True, True, True, True\}. \tag{3.3}$$

Both of them have fitness $y_{best}{}^1 = y_{best}{}^2 = -23$. We have presented this corner case in order to highlight the role of weights: in situations like the one just shown, assigning a positive weight larger than 1 to the number of actions is going to make $x_{best}{}^2$ prevail as best state; vice-versa if the residual service matters more.

3.3. Random graphs

After showing the capabilities of SAFEX in detail, we would like to present an overview analysis on random graphs. We have focused our analysis on sparse random graphs of size 100.

We have generated sparse directed graphs using $fast\_gnp\_random\_graph$ function of $NetworkX$. This method, for a fixed value of the number of nodes n, and of the probability p of creating edges, returns a $G_{n,p}$ random graph, also known as an Erdős-Rényi graph or binomial graph [14, 32].

In the $G_{n,p}$ model, a graph is constructed by connecting labeled nodes randomly. Each edge is included in the graph with probability p, independently from every other edge. In this study, we have always chosen $p = 1/n$.

We have started with sparse graphs of size 100 nodes. We have created different versions of the same random graph, each one with an increasing percentage of $SWITCH$ nodes, from 0.1 to 0.9. $SOURCE$ and $USER$ nodes have not been modified in this process, in order to be able to compare the total service before and after the perturbation. The percentage of $SWITCH$ nodes is relative to the total number of HUB nodes, which have been turned into larger and larger numbers of switches. It is important to specify that for increasing the percentage of switches, the set of $SWITCH$ nodes contains the ones of smaller percentages as a subset.

All the nodes without fathers have been labeled as $SOURCE$, and the ones with no children have been labeled as $USER$. The rest of the nodes are all HUB nodes initially; a percentage larger and larger is turned into $SWITCH$ nodes, as stated previously. No node has been labeled as perturbation resistant: the reason for this choice is that we were interested in looking at the propagation of the perturbation in the graph, modulated by the presence of switches. All the edges have a weight equal to one. The parameters of the genetic algorithm are the following: $npop$ individuals for the initial population, $ngen$ generations, probability $indpb$ for the attributes to be changed, and threshold $tresh$ for applying crossover/mutation. The population to be iterated through the generations is created first from the initial population, choosing the best $nsel$ selected children, which get modified by mutation or crossover at each step, according to the defined probabilities.

The first graph that we experience is a sparse random graph with only AND logic relationships between nodes. This is the worst-case scenario: in fact, in the absence of perturbation resistances of any kind, the fault propagation can stop just when arriving at the end of a certain path, after having run it all. In the case of OR fathers, instead, just one of the OR fathers has to remain alive in order for the node not to be deleted.

We present as the first example the perturbation of node 83 with and without switches. This node is the one with the highest out-degree centrality in the graph. Since the fault propagates from one node to all its children, the breakage of this node is the most disruptive for the graph. Disabling switches, the perturbation of node 83 causes the break of 17 nodes: $[3, 9, 26, 42, 46, 48, 51, 53, 60, 70, 75, 77, 83, 84, 87, 89, 94]$. The original service, equal to 27, turns out to be 14 in the end. In Table 1 we show instead the results for increasing the percentage of switches in the graph. The results shown in the table are the best of 10 runs, meaning the ones with the lowest fitness. Moreover, it is worth mentioning that while obtaining these results, we have fixed the same weights $w_1 = w_2 = w_3 = 1$ for the three contributions of the fitness. This implies that the genetic algorithm is going to take into account equally the total number of actions and the total number of survived nodes in order to identify the best state, not only the total final service.

What we immediately notice is the fact that, even for the smallest percentage of switches, the number of broken nodes is reduced with respect to their total absence. Moreover, the number of broken nodes diminishes for increasing switch percentage, while the service increases.

Given the set of switches for fixed percentages, the set for smaller percentages is a subset of the

former. Hence, we expect that for increasing the percentage of switches the fitness must remain the same or decrease, if increasing the number of $SWITCH$ nodes a lower minimum has been reached. We notice this behavior in Table 1.

The second example that we would like to show is the perturbation of a single node with the smallest out-degree centrality, equal to 1. A fault departing from this node should have a less severe effect on the graph. Disabling switches, the perturbation of node 93 causes the break of 4 nodes: $[25, 34, 70, 93]$. The original service, equal to 27, turns out to be 24 in the end. In Table 2 we show instead the results for increasing the percentage of switches in the graph. We notice that for this perturbation the genetic algorithm converges immediately to a situation in which the only broken node is the one from which the fault starts. This is the best possible result: in fact, no other node is affected. Node 93 is a $SOURCE$ node, and its only connection is to node 25, which is a $SWITCH$ node for all percentages.

The lower value of service for 80% of switches can be justified by a smaller number of actions. We again recall that the fitness takes into account the number of actions, the total number of survived nodes, and the total final service with the same weight.

We are now going to present the same perturbation as above, nodes 83 and 93, for graphs that present both *OR* and *AND* edges. These types of graphs have been created from the ones with just *AND* edges, turning half of the edges to *OR* edges.

The first perturbation studied is as before the one of node 83, the most connected. In the absence of switches, this perturbation causes the breakage of 12 nodes, with a total final service of 21. As expected, the presence of *OR* edges mitigates the consequences of the perturbation, even without switches. In Table 1 we can see that this effect is perpetuated even in the presence of switches. With respect to Table 1, the service starts higher even for the smallest percentage of switches and is always larger compared to the same percentage of switches.

Regarding the perturbation of node 93 instead, in the absence of switches three nodes get broken, with a final service of 24. We can say that the presence of *OR* edges improves the final state in absence of switches, while in their presence the situation with or without *OR* edges is more or less the same. However, we have to recall node 93 presents a favorable situation, being linked by its only edge to a $SWITCH$ node. We can conclude that in any case, the presence of switches improves the final state, both in terms of the total number of survived nodes and in terms of total final service.

Tables 3 and 4 show the cases in which one of the fitness weights has been alternately set equal to 10, while the other two remain equal to 1. Table 3 exhibits the fact that rewarding the survival of nodes is somehow equivalent to promoting a larger amount of total final service. Table 4 instead underlines the fact that when the cost of switch flips is large, the need to keep a low number of actions takes place at the expense of the survival of nodes and of the total final service.

Table 1. Results for the best switch state for increasing percentage of switches. In the table header, *#acts* represents the number of actions, *surv* the number of survived nodes after the perturbation, S_{tot} is the total final service, while y_{best} is the best fitness of 10 runs. This example refers to the perturbation of node 83 (out-degree centrality equal to 4), for a graph of size 100 nodes. On the left, results for just *AND* edges are shown, while on the right results for both *OR* and *AND* edges are shown. Parameters: $npop = 400$, $ngen = 200$, $indpb = 0.7$, $tresh = 0.4$, $nsel = 100$.

No switches: 83 survived nodes, $S_{tot} = 14$ (*AND*); 88 survived nodes, $S_{tot} = 21$ (*OR* and *AND*)								
%	**# acts**	**surv.**	S_{tot}	y_{best}	**# acts**	**surv.**	S_{tot}	y_{best}
10	1	84	14	−97	1	90	22	−111
20	2	94	22	−114	1	95	26	−120
30	2	94	22	−114	2	96	26	−120
40	2	94	22	−114	2	96	26	−120
50	3	96	22	−115	3	98	26	−121
60	3	96	22	−115	3	98	26	−121
70	3	96	22	−115	3	98	26	−121
80	3	98	23	−118	3	99	26	−122
90	3	98	23	−118	3	99	26	−122

Table 2. Results for the best switch state for increasing percentage of switches. In the table header, *#acts* represents the number of actions, *surv* the number of survived nodes after the perturbation, S_{tot} is the total final service, while y_{best} is the best fitness of 10 runs. This example refers to the perturbation of node 93 (out-degree centrality equal to 1), for a graph of size 100 nodes. On the left, results for just *AND* edges are shown, while on the right results for both *OR* and *AND* edges are shown. Parameters: $npop = 400$, $ngen = 200$, $indpb = 0.7$, $tresh = 0.4$, $nsel = 100$.

No switches: 96 survived nodes, $S_{tot} = 24$ (AND); 97 survived nodes, $S_{tot} = 24$ (*OR* and *AND*)								
%	**# acts**	**surv.**	S_{tot}	y_{best}	**# acts**	**surv.**	S_{tot}	y_{best}
10	1	99	25	−123	1	99	25	−123
20	1	99	25	−123	1	99	25	−123
30	1	99	25	−123	1	99	25	−123
40	1	99	25	−123	1	99	25	−123
50	1	99	25	−123	1	99	25	−123
60	2	99	25	−122	1	99	25	−123
70	2	99	25	−122	1	99	25	−123
80	6	99	25	−118	1	99	25	−123
90	5	99	25	−119	1	99	25	−123

Table 3. Results for the best switch state for increasing percentage of switches. In the table header, *#acts* represents the number of actions, *surv* the number of survived nodes after the perturbation, S_{tot} is the total final service, while y_{best} is the best fitness of 10 runs. This example refers to the perturbation of node 83 (out-degree centrality equal to 4), for a graph of size 100 nodes. On the left, weights $w_1 = w_3 = 1$, $w_2 = 10$, while on the right weights $w_1 = w_2 = 1$, $w_3 = 10$. Parameters: $npop = 400$, $ngen = 200$, $indpb = 0.7$, $tresh = 0.4$, $nsel = 100$.

No switches case: 83 survived nodes, $S_{tot} = 14$								
%	**# acts**	**surv.**	S_{tot}	y_{best}	**# acts**	**surv.**	S_{tot}	y_{best}
10	1	84	14	−223	1	84	14	−853
20	1	91	22	−310	1	91	22	−931
30	2	94	22	−312	2	94	22	−960
40	2	94	22	−312	2	94	22	−960
50	3	96	22	−313	3	96	22	−979
60	3	96	22	−313	3	96	22	−979
70	3	96	22	−313	3	96	22	−979
80	3	98	23	−325	3	98	23	−1000
90	3	98	23	−325	3	98	23	−1000

Table 4. Results for the best switch state for increasing percentage of switches. In the table header, *#acts* represents the number of actions, *surv* the number of survived nodes after the perturbation, S_{tot} is the total final service, while y_{best} is the best fitness of 10 runs. This example refers to the perturbation of node 83 (out-degree centrality equal to 4), for a graph of size 100 nodes, with weights $w_2 = w_3 = 1$, $w_1 = 10$. Parameters: $npop = 400$, $ngen = 200$, $indpb = 0.7$, $tresh = 0.4$, $nsel = 100$.

No switches case: 83 survived nodes, $S_{tot} = 14$				
%	**# acts**	**surv.**	S_{tot}	y_{best}
10	0	83	14	−97
20	1	91	22	−103
30	1	91	22	−103
40	1	91	22	−103
50	1	91	22	−103
60	1	91	22	−103
70	1	91	22	−103
80	1	91	22	−103
90	1	91	22	−103

4. Conclusions

In this work, we have presented a method able to perform risk analysis, together with the consequent fault system diagnostics, on complex interconnected systems. Everything that has been presented in this paper is implemented in GRAPE software [9].

Weighted graphs describe preferable itineraries for the flow of commodities, concerning areas in which the course can be less favored; everything is efficiently integrated in the graph paradigm. Our focus in this work has been devoted to the introduction of the new concept of *service*. Its creation redefines the different roles of the nodes within the network and allows quantification of its residual operational capacity. In particular, its value is preserved and maximized by an automatic identification of the best configuration of the set of $SWITCH$ nodes, that are involved in limiting the cascade effect within GRAPE simulation of a fault propagation in the system.

The different types of perturbations available in the software are explained, and several examples are given on model graphs. Results obtained on random sparse graphs are also presented, highlighting how fault limitation is improved by the presence of an increasing number of switches. The methodology illustrated here is of immediate application to industrial plant systems of different natures. The graph standard is general enough to be applied to diverse cases, integrated with a precise estimate of the resources still available in the system. For these reasons, this software can be employed in the development of a digital twin for fault analysis, from the early stages. The scalability of the graph concept towards larger systems is of immediate transfer and is going to be analyzed in even further detail in the future. Future developments are going to include querying the software with realistic plants of increasing complexity. This implies the extension of GRAPE to graphs of increasing size.

Finally, another possible extension of this work can be the coupling of graph topology with realistic physical quantities on nodes or edges; it can be of relatively easy implementation within our software and could be of wide use within realistic industrial applications.

Use of AI tools declaration

The authors declare they have not used Artificial Intelligence (AI) tools in the creation of this article.

Acknowledgements

The authors want to thank Francesco Andreuzzi for the productive discussions and comments, as well as the CETENA team for its support during the development. This work was partially supported by the project SAFE, “Realtime Damage Manager And Decision Support”, supported by Regione FVG, POR-FESR 2014-2020, Piano Operativo Regionale Fondo Europeo per lo Sviluppo Regionale, and by European Union Funding for Research and Innovation–Horizon 2020 Program–in the framework of European Research Council Executive Agency: H2020 ERC CoG 2015 AROMA-CFD project 681447 “Advanced Reduced Order Methods with Applications in Computational Fluid Dynamics” P.I. Gianluigi Rozza.

Conflict of interest

All authors declare no conflicts of interest in this paper.

References

1. S. Chechik, M. Langberg, D. Peleg, L. Roditty, Fault tolerant spanners for general graphs, *SIAM J. Comput.*, **39** (2010), 3403–3423. https://doi.org/10.1137/090758039
2. W. Shi, D. B. West, Diagnosis of wiring networks: an optimal randomized algorithm for finding connected components of unknown graphs, *SIAM J. Comput.*, **28** (1999), 1541–1551. https://doi.org/10.1137/S0097539795288118
3. J. Kleinberg, M. Sandler, A. Slivkins, Network failure detection and graph connectivity, *SIAM J. Comput.*, **38** (2008), 1330–1346. https://doi.org/10.1137/070697793
4. C. Reinartz, D. Kirchhübel, O. Ravn, M. Lind, Generation of signed directed graphs using functional models, *IFAC-PapersOnLine*, **52** (2019), 37–42. https://doi.org/10.1016/j.ifacol.2019.09.115
5. C. Palmer, P. W. H. Chung, Creating signed directed graph models for process plants, *Ind. Eng. Chem. Res.*, **39** (2000), 2548–2558. https://doi.org/10.1021/ie990637v
6. M. R. Maurya, R. Rengaswamy, V. Venkatasubramanian, A signed directed graph and qualitative trend analysis-based framework for incipient fault diagnosis, *Chem. Eng. Res. Des.*, **85** (2007), 1407–1422. https://doi.org/10.1016/S0263-8762(07)73181-7
7. D. Peng, Z. Geng, Q. Zhu, A multilogic probabilistic signed directed graph fault diagnosis approach based on bayesian inference, *Ind. Eng. Chem. Res.*, **53** (2014), 9792–9804. https://doi.org/10.1021/ie403608a
8. X. Ma, D. Li, A hybrid fault diagnosis method based on fuzzy signed directed graph and neighborhood rough set, *2017 6th Data Driven Control and Learning Systems (DDCLS)*, Chongqing, China, 2017, 253–258. https://doi.org/10.1109/DDCLS.2017.8068078
9. mathLab, GRAPE: GRAph Parallel Environment. Available from: `https://github.com/mathLab/GRAPE`.
10. A. Maurizio, *Representation of distribution networks of ships using graph-theory*, MS. Thesis, SISSA (International School for advanced Studies), 2018.
11. M. Teruzzi, *Parallel implementations for complex graph analysis with application in modern passenger ship safety management*, MS. Thesis, SISSA (International School for advanced Studies), 2020.
12. L. Euler, Solutio problematis ad geometriam situs pertinentis, *Comment. Acad. Sci. Petropolitanae*, 1736, 128–140.
13. J. J. Sylvester, Chemistry and algebra, *Nature*, **17** (1878), 284. https://doi.org/10.1038/017284a0
14. P. Erdős, A. Rényi, On random graphs I, *Publ. Math.*, **6** (1959), 290–297.
15. A. Cayley, On the theory of the analytical forms called trees, *Phil. Mag.*, **13** (1857), 172–176.

16. G. Kirchhoff, Ueber den durchgang eines elektrischen stromes durch eine ebene, insbesonere durch eine kreisförmige, *Ann. Phys.*, **140** (1845), 487–514. https://doi.org/10.1002/andp.18451400402
17. F. Dörfler, J. W. Simpson-Porco, F. Bullo, Electrical networks and algebraic graph theory: Models, properties, and applications, *Proceedings of the IEEE*, **106** (2018), 977–1005. https://doi.org/10.1109/JPROC.2018.2821924
18. W. Huber, V. J. Carey, L. Long, S. Falcon, R. Gentleman, Graphs in molecular biology, *BMC Bioinformatics*, **8** (2007), S8. https://doi.org/10.1186/1471-2105-8-S6-S8
19. E. Otte, R. Rousseau, Social network analysis: a powerful strategy, also for the information sciences, *J. Inform. Sci.*, **28** (2002), 441–453. https://doi.org/10.1177/016555150202800601
20. E. W. Dijkstra, A note on two problems in connexion with graphs, *Numer. Math.*, **1** (1959), 269–271. https://doi.org/10.1007/BF01386390
21. R. Floyd, Algorithm 97: shortest path, *Communications of the ACM*, **5** (1962), 345. https://doi.org/10.1145/367766.368168
22. B. Roy, Transitivité et connexité, *C.-R. Acad. Sci. Paris*, **249** (1959), 216–218.
23. S. Warshall, A theorem on boolean matrices, *J. ACM*, **9** (1962), 11–12. https://doi.org/10.1145/321105.321107
24. J. H. Holland, Genetic algorithms and the optimal allocation of trials, *SIAM J. Comput.*, **2** (1973), 88–105. https://doi.org/10.1137/0202009
25. J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*, MIT Press, 1992.
26. M. Kumar, M. Husain, N. Upreti, D. Gupta, Genetic algorithm: review and application, *Int. J. Inf. Tech. Knowl. Manage.*, **2** (2010), 451–454.
27. T. A. El-Mihoub, A. A. Hopgood, L. Nolle, A. Battersby, Hybrid genetic algorithms: a review, *Eng. Lett.*, **13** (2006), 124–137.
28. R. Sivaraj, T. Ravichandran, A review of selection methods in genetic algorithm, *Int. J. Eng. Sci. Tech.*, **3** (2011), 3792–3797.
29. S. M. Elsayed, R. A. Sarker, D. L. Essam, A new genetic algorithm for solving optimization problems, *Eng. Appl. Artif. Intel.*, **27** (2014), 57–69. https://doi.org/10.1016/j.engappai.2013.09.013
30. Z. Drezner, A new genetic algorithm for the quadratic assignment problem, *INFORMS J. Comput.*, **15** (2003), 320–330. https://doi.org/10.1287/ijoc.15.3.320.16076
31. F. A. Fortin, F. M. De Rainville, M. A. Gardner, M. Parizeau, C. Gagné, DEAP: evolutionary algorithms made easy, *J. Mach. Learn. Res.*, **13** (2012), 2171–2175.
32. V. Batagelj, U. Brandes, Efficient generation of large random networks, *Phys. Rev. E*, **71** (2005), 036113. https://doi.org/10.1103/PhysRevE.71.036113

AIMS Press