Games on graphs from theory to practice: a novel algorithm for solving reachability and safety games

Christian Maria Giannetti
Master in artificial intelligence and robotics
Sapienza University of Rome
giannetti.1904342@studenti.uniroma1.it

***Abstract*—Games on graphs are a topic raising increasing interest in the research community since the mathematical analysis and simulations of such distributed games are crucial in several fields, including economics, politics, and epidemiology. This paper aims to provide a comprehensive documentation to orient the reader through the design choices made during the project's development. The project's goal was to learn the principles at the basis of games on graphs and implement an algorithm to solve reachability and safety games by exploiting their inner duality and a framework for the testing procedure. The paper is structured as follows. The first section introduces the theoretical compendium that rigorously formalizes several fundamental concepts related to reachability and safety games. The second section illustrates the design choices made concerning the construction of the arenas. The third section provides a concrete formulation of reachability and safety games, as well as the algorithms to solve them, in order to bring the reader to the implementation. The fourth section introduces a novel algorithm that combines the logic underpinning reachability and safety games, respectively, and includes several optimizations to increase its performance. Finally, the fifth section presents a wide range of experiments conducted and discusses the collected results.**

I. Introduction

The present section includes a theoretical compendium that introduces and rigorously formalizes several notions: from the definition of deterministic turn-based games to the theorem of determinacy of reachability games with its corollaries and the theorem duality of reachability and safety games. All the information contained in this section are the fruit of personal revision of multiple resources among which [1], [2], [3], [4], [5].

A. Deterministic turn-based games

Fundamental definitions In deterministic games, a game graph is a structure $G = (V, E, V_0, V_1)$, which consist of a directed graph (V, E) and a partition (V_0, V_1) of its set of vertices $V = V_0 \cup V_1$. We denote with *positions* and *moves*, respectively, the vertices and the edges of a game graph. Furthermore, we define with $E^+(v)$ the set of successors of a position v $\in$ V (i.e., the set of nodes $E^+(v) = \{s : (v, s) \in E\}$) and with $E^-(v)$ the set of predecessors of a position v $\in$ V (i.e., the set of nodes $E^-(v) = \{p : (p, v) \in E\}$). A generic play on a game graph involves two players, Player 0 and Player 1, which define a path by transferring a token along the graph's edges. At the beginning of the play, the token is assigned to a position.

Notion of turn-based game We refer to the partition (V_0, V_1) with the term *turn partition* and the notion of *turn-based* play is defined according to the player who controls the node containing the token. Specifically, if the current position v belongs to V_0, Player 0 is in turn to move the token to a successor $s \in E^+(v)$; otherwise, if $v \in V_1$, Player 1 is in turn to move. Hence, a play starting from a designated position $v_0 \in V$ in G is an infinite sequence of positions $\pi = v_0, v_1, \ \dots \ , v_n$ that represents a path on the graph G. In this respect, we define an *initial play* as a prefix of a play.

Definition of strategy We define a *strategy* for Player 0 as a function $\sigma : V^* V_0 \mapsto V$ that maps every initial play $v_0, v_1, \ \dots \ , v_n$ terminating in a position $v_n \in V_0$ to a successor position $v_{n+1} \in E^+(v)$. A play $\pi = v_0, v_1, \ \dots$ follows the strategy σ if $v_{n+1} = \sigma(v_0, \ \dots \ , v_n)$ for all n with $v_n \in V_0$. Analogously, we define a strategy for Player 1. Once a starting position is fixed, any pair of strategies (σ, τ) for Player 0 and Player 1, respectively, determines a play. Furthermore, we denominate the unique play that follows both σ and τ the *outcome* of the two strategies and indicate it by $\sigma^\wedge\tau$.

Definition of winning condition A winning condition on G is a set $W \subseteq V^\omega$ of plays. Player 0 wins a play π if $\pi \in W$, otherwise Player 1 is the winner of the play. We designate a strategy as winning for a player if all plays that follow the strategy are winning.
Conclusively, we represent a game $G = (G, W)$ by defining a game graph and a winning condition.

B. Reachability and safety games

We define a reachability condition through a set $F \subseteq V$ of *target* positions that describes the following winning condition:

$$\{v_0, v_1, \ \dots, : v_n \in \ F \ for \ some \ n \ \geq 0\}$$

Analogously we denote a safety condition through a set $F \subseteq V$ of *safe* positions that describes the following winning condition:

$$\{v_0, v_1, \ \dots, : v_n \in \ F \ for \ all \ n \ \geq 0\}$$

For the sake of completeness, we define infinitary conditions by referring to the set of elements that occur infinitely often in a sequence:

$$Inf(v_0, v_1, v_2, \dots) : \{v = v_n \text{ for infinitely many } n \geq 0\}$$

However, we remark that only a finite number of moves is needed to establish whether a reachability condition is fulfilled or a safety condition is violated.

Determinacy of reachability games Note that concluding if a game is determined or not corresponds to discern if, given a class Γ of games, it is the case that for any $G \in \Gamma$, either Player 0 or Player 1 has a winning strategy. For the sake of completeness, we point out that an alternative formulation corresponds to investigate whether, given the game $G = (V, W)$, there exists a partition $V = W_0 \cup W_1$ such that Player 0 has a winning strategy from any position $v \in W_0$, whereas Player 1 has a winning strategy from any position $v \in W_1$. If this is the case, we denote the sets W_0, W_1 as the *winning regions* of Player 0 and Player 1, respectively.

In this respect, reachability games (i.e., games with reachability winning conditions) are determined, as specified by the following theorem:

Theorem of determinacy of reachability games *For every reachability game $G = (V, V_0, E, F)$, there exists a partition $W_0 \cup W_1 = V$ such that Player 0 has a winning strategy from any starting position $v \in W_0$ and Player 1 has a winning strategy from any starting position $v \in W_1$.*

Theorem of determinacy of reachability games: proof
This paragraph presents the proof of the theorem of determinacy of reachability games on finite graphs. Specifically, we construct inductively a sequence of sets $((Attr^0_i(F))_{i \geq 0}$ with the property that from any position of $Attr^0_i(F)$, Player 0 can ensure to reach F in at most i many steps:

$$\begin{aligned}
&Attr^0_0(F) := F;\\
&Attr^0_{i+1}(F) := Attr^0_i(F)\\
&\cup \{v \in V_0 : E^+(v) \cap Attr^0_i(F) \neq \emptyset\}\\
&\cup \{v \in V_1 : E^+(v) \subseteq Attr^0_i(F)\}
\end{aligned}$$

The sequence is increasing until it reaches a fixed point $Attr^0_i(F) = Attr^0_{i+1}(F)$ after at most $|V|$ many stages. We denote this fixed point by $Attr^0(F)$ and call it the $attractor$ of F for Player 0.
We claim that:

$W_0 := Attr^0(F)$ is the winning region of Player 0 on G
$W_1 := V \backslash Attr^0(F)$ is the winning region of Player 1 on G.

To demonstrate this, we define a function $rank : V \rightarrow \omega \cup \{\infty\}$ that associates to every position $v \in V$ the stage at which it was included into the attractor,

$$rank(v) \triangleq \begin{cases} min\{i : v \in Attr^0_i(F)\} & \text{if v} \in Attr^0(F) \\ \infty & \text{if v} \notin Attr^0(F) \end{cases}$$

The above-mentioned function has the following property:

(i) for every $v \in Attr^0(F)$, either

- $v \in F$, or
- $v \in V_0 \backslash F$ and for some successor w $\in E^+(v)$, $rank(w) < rank(v)$, or
- $v \in V_1 \backslash F$ and for all successors w $\in E^+(v)$, $rank(w) < rank(v)$;

(ii) for every v $\in V \backslash Attr^0(F)$, either

- $v \in V_0$, and, for all successors $w \in E^+(v)$, $rank(w) = \infty$, or
- $v \in V_1$, and, for some successor $w \in E^+(v)$, $rank(w) = \infty$.

Accordingly, we can define a function $f : V_0 \rightarrow V$ that selects for every position $v \in V_0$ a successor $f(v) \in E^+(v)$ such that $rank(f(v)) < rank(v)$ whenever $v \in Attr^0(F) \backslash F$.
Then, we consider the reachability game G with an arbitrary starting position $v_0 \in Attr^0(F)$, and let $\sigma : V^*V_0 \rightarrow V$ be the strategy for Player 0 that chooses for every initial play $\pi = v_0, v_1, \dots, v_l$ with $v_l \in V_0$ the successor $f(v_l)$. Then, any play $v_0, v_1, v_2, \dots$ that follows σ must reach a position of F. Otherwise, we had a strictly decreasing sequence

$$rank(v_0) > rank(v_1) > rank(v_2) > \dots$$

of infinite length, which cannot happen. Thus, σ is a winning strategy for Player 0 in G, v_0.
Conversely, for Player 1, let $g : V_1 \rightarrow V$ be a function that selects for every $v \in V_1$ a successor $g(v)$ such that $rank(g(v)) = \infty$ whenever $v \in V \backslash Attr^0(F)$. For the game G starting at an arbitrary position $v_0 \in F \backslash Attr^0(F)$, define a strategy $\tau : V^*V_1 \rightarrow V$ for Player 1 by associating to every initial play $v_0, v_1, \dots v_l$ with $v_l \in V_1$ the successor $g(v_l)$. Then, in any play $v_0, v_1, v_2, \dots$ following τ, we have $rank(v_i) = \infty$ at all indices $i \geq 0$, which means that F is never reached. Hence, τ is a winning strategy for Player 1 in the reachability game G, v_0.

Memoryless strategy A strategy $\sigma : V^*V_0$ for a game G, v_0 is defined *positional* or *memoryless* if there exists a function $f : V_0 \mapsto V$ such that $\sigma(v_0, v_1, v_2, \dots, v_n) = f(v_n)$ for all initial plays $v_0, v_1, \dots, v_n$ with $v_n \in V_0$. The existence of such function f induce a memoryless strategy.

Given a fixed game G, a memoryless strategy $f : V_0 \mapsto V$

induces a (proper) strategy $\sigma : V^*V_0 \mapsto V$ for each game G, v_0 with $v_0 \in V$. We denote such a memoryless strategy as *uniformly winning* over a set of positions $U \subseteq V$ if the (induced proper) strategy is winning in every game G, v_0 with $v_0 \in U$.

First corollary of reachability games' theorem of determinacy

For every reachability game G, the set V of positions can be partitioned into $W_0 \cup W_1 = V$ such that Player 0 has a uniform memoryless winning strategy over W_0 and Player 1 has a uniform memoryless winning strategy over W_1. The winning regions, jointly with the memoryless strategies, can be computed in time $\mathcal{O}(|V| + |E|)$.

Duality of reachability and safety games Every result drawn for reachability games applies readily to safety games since reachability games and safety games are dual. Specifically, given a fixed game G, it is possible to solve the game from both the players' perspectives.

Formally, we first define the following propositions:

$P_1 =$ Player 0 has a winning strategy in the safety game (V, V_0, V_1, E, F)

$P_2 =$ Player 1 has a winning strategy in the reachability game $(V, V_1, V_0, F_0 = V \backslash F)$, where the players' roles are switched.

Then, we observe that the following condition applies:

$$P_1 \iff P_2$$

Second corollary of reachability games' theorem of determinacy

For every safety game G, the set V of positions can be partitioned into $W_0 \cup W_1 = V$ such that Player 0 has a uniform memoryless winning strategy over W_0 and Player 1 has a uniform memoryless winning strategy over W_1. The winning regions, jointly with the memoryless strategies, can be computed in time $\mathcal{O}(|V| + |E|)$.

II. From Theory to Practice: Building the Arena

A. Representing the graph: data structures employed

This paragraph describes the data structure employed to represent the graph. The function that builds the graph takes as input:

- The nodes controlled by the reachability player, labelled with an integer.
- The nodes controlled by the safety player, labelled with an integer.
- The edges between the nodes present in the graph.

Given these inputs, the function generates two graphs: the straight and transpose graphs. The latter is employed in the optimized versions of the presented algorithms to limit the computational resources in specific scenarios and decrease the time required to solve the game. At the implementation level, we model each graph through a dictionary that associates an integer representing the node's label with a set of integers that include all the nodes' neighbors. In other terms, each integer in the set represents a graph node. We address in detail the reasons why it is convenient to exploit the transpose graph in section IV.

B. Random graph generation: automatic experiments

In order to facilitate and speed up the testing procedure, we provide a function that automatically generates a random graph. We have integrated this function into the code to allow the user to generate large graphs quickly. Precisely, the function takes as input the following parameters:

- *Number of nodes*: an integer that specifies the number of nodes.
- *Number of edges*: an integer that specifies the number of edges.
- *Self-loops*: a Boolean that regulates the presence of self-loops in the graph.
- *Isolated nodes*: a Boolean that regulates the presence of isolated nodes in the graph.

An equally crucial matter during the testing phase is to guarantee repeatable and unbiased tests. For this reason, we also provide a function responsible for executing a repeatable, automatic, and unbiased battery of tests. Specifically, the function takes the following inputs:

a := *num_nodes_min*
b := *num_nodes_max*
c := *avg_edges_per_node_min*
d := *avg_edges_per_node_max*
e := *target_safe_ratio_min*
f := *target_safe_ratio_max*

Given the inputs listed above, the function generates multiple graphs and uses them as arenas to benchmark multiple algorithms. Specifically, for each experiment requested by the user, it computes the following parameters:

- *num_nodes:=* $randInt(a, b)$. This parameter specifies the number of nodes of the graph.
- *num_edges:=* $\lfloor num\_nodes * randFloat(c, d) \rfloor$. This parameter specifies the number of edges of the graph.
- *target_safe_size:=* $\lfloor num\_nodes * randFloat(e, f) \rfloor$ This parameter defines the ratio that regulates the dimension of the target set for the safety player with respect to the total number of nodes in the graph (e.g., a ratio of 0.7 indicates that the target set has a dimension equal to the 70% of the total number of nodes in the graph).

C. Random graph generation: avoiding isolated nodes

The randomly generated graph employed in the experimental phase may contain isolated nodes. Their presence increases the computational cost of the algorithm's execution, but it does not add any informative content to the experiments. Since this phenomenon becomes significant in large graphs, we propose a method that, given a randomly generated disconnected graph, returns a connected graph that is finally used to perform the desired experiment.

We explain in the following lines the simple logic underpinning the method. We first generate an NxN adjacency matrix whose element $(i,j)=1$ if the edge (i,j) is present in the graph, otherwise $(i,j)=0$. Then, for each (i,j), we add 1 with a probability such that the expected value of the edges' number is equal to an integer parameter taken as input, denoted as E. At this point, if the option $no\_isolated$ is set to $true$, we iterate through the matrix to verify if it contains isolated nodes. Specifically, for each row of the matrix, we check that all the row's elements are equal to 0. If this is the case, we randomly select an integer j to determine which column will be modified. Then, we flip a fair coin to decide whether to add the edge (i,j) or (j,i). This process is repeated until the graph has no longer isolated nodes.

III. From theory to practice: problem formulation from different perspectives

A. *Safety-reachability duality: Reachability problem formulation*

Given a game, seen from the reachability player point of view, it is possible to decompose the set $Force_R(X)$ as follows:

$$\begin{aligned}
&Reach_comp(X)=\{v\in V_R : E(v)\cap X\neq\emptyset\}\\
&Safety_comp(X)=\{v\in V_S : E(v)\subseteq X\}\\
&Force_R(X)=Reach_comp(X)\cup Safety_comp(X)
\end{aligned}\tag{1}$$

We observe that the $Reach\_comp(X)$ includes all the nodes with at least a successor in the reachability player's winning set, i.e., the nodes for which the reachability player has a move to enter in the region X. On the other hand, $Safety\_comp(X)$ includes all the nodes whose all successors are in the reachability player's winning set, i.e., the nodes the safety player cannot avoid entering the region X. It follows that the set $Force_R(X)$ contains the nodes for which the reachability player can enforce the token to move in the reachability player's winning set.

B. *Safety-reachability duality: naive purely backward algorithm*

The algorithm presented below computes the winning set for the reachability player, i.e., the set of nodes from which the reachability player can reach the target set's nodes. Note that the target set is given as input. At each iteration, the algorithm computes $Force_R(X)$, and it adds to the winning set those nodes from which the reachability player has a move to enter $Force_R(X)$. In this respect, we denote this paradigm to solve the game as *proceeding backward*, since the first iteration will add the nodes for which it is possible to reach the target set through a single edge, the second iteration will include the nodes from which the reachability player can reach the target set through two edges and so forth.

Naive purely backward algorithm

Input:
Graph $G(V,E)$: The graph representing the arena.
Target_reach: The target set for the reachability player.
Begin:
Win = Target_reach
while (Win $\neq$ (Win $\cup$ force(Win)): **do**
    Win = Win $\cup$ force(Win)
end while
return Win

C. *Safety-reachability duality: naive purely forward algorithm*

Given a game, seen from the safety player point of view, it is possible to decompose the set $Force_S(X)$ as follows:

$$\begin{aligned}
&Reach_comp(X)=\{v\in V_R : E(v)\subseteq X\}\\
&Safety_comp(X)=\{v\in V_S : E(v)\cap X\neq\emptyset\}\\
&Force_S(X)=Reach_comp(X)\cup Safety_comp(X)
\end{aligned}\tag{2}$$

We remark that in a safety game, player's roles are inverted. In fact, in this scenario, the reachability component $Reach\_comp(X)$ includes all the nodes whose all successors are in the safety player's winning set, i.e., the nodes the reachability player cannot avoid entering the region X. On the other hand, the safety component $Safety\_comp(X)$ includes all the nodes with at least a successor in the safety player's winning set, i.e., the nodes for which the safety player has a move to enter in the region X. It follows that the set $Force_S(X)$ contains the nodes for which the safety player can enforce the token to remain in his winning set forever.

D. *Safety-reachability duality: naive purely forward algorithm*

As previously pointed out, the algorithm below computes the set of nodes from which the safety player can remain in the target set forever. Given as input the target set, a set of candidate safe nodes, the algorithm removes at each iteration those nodes from which the reachability player has a move to enter his target set (i.e., the unsafe nodes for the safety player). In doing so, as the game progresses, each unsafe node is removed from the target set, and the algorithm finally determines the winning set for the safety player. In this respect, we denote this paradigm to solve the game as *proceeding forward*.

Naive purely forward algorithm

Input:
Graph $G(V,E)$: The graph representing the arena.
Target_safe: The target set for the safety player.
Begin:
Win = Target_safe
while (Win $\neq$ (Win $\cap$ force(Win)): **do**
    Win = Win $\cap$ force(Win)
end while
return Win

IV. Optimizations

The following sections discuss in detail the algorithmic optimizations proposed and also provide their theoretical justifications. We remark that all the proposed optimizations are not mutually exclusive, thus it is possible to use them simultaneously.

A. Transpose graph optimization

The following analysis aims to provide the reader the intuition for which it is convenient to employ the transpose graph in solving reachability or safety games. For the sake of clarity, we provide an example by using formulation (1). To compute $Reach\_comp(X)$, that is the component of $Force_R(X)$ related to the reachability player, it is necessary to iterate through the list of all the nodes in the straight graph and verify that from the node v, it is possible to reach the region X. We observe that, by construction, the target set is always a subset of the complete set of the graph nodes (i.e., the total number of nodes in the graphs is always greater or equal than the number of nodes belonging to the target set). Note that their difference, in terms of nodes' number, can also span several orders of magnitude.

We provide an alternative method that aims to reduce the computational cost of computing $Reach\_comp(X)$. Specifically, when computing nodes to add in $Reach\_comp(X)$, we would like to consider only those nodes in V that do have an edge to a node that belongs to the target set X. From this observation follows that it is beneficial to employ a transpose graph to compute these nodes directly. More precisely, instead of considering each node $v \subseteq V$ from which it is possible to reach the target set X in the straight graph, it is much more convenient to examine each node $v \subseteq V$ reached from a node $x \subseteq X$ in the transpose graph. This optimization exploits the fact that the target set's size, on average, is smaller than the size of V. It follows that the optimization is particularly effective in the algorithm's first iterations because the target set's size increases at each consecutive iteration.[1] Specifically, the only scenario in which the two approaches have exactly the same computational complexity is the one in which the graph $G(V, E)$ is complete (i.e., every node $v \subseteq V$ is connected to all the remaining graph nodes). In this latter case, the computational complexity of both approaches becomes quadratic in the number of nodes. However, the occurrence of such a scenario is exceptionally unlikely in the case of randomly generated graphs, which are generally sparse.

We note that this optimization becomes less and less effective the denser the graph, but it does increase the algorithm's scalability on average.

[1]This observation holds if the game is solved from the reachability player's point of view. Conversely, if the game is solved from the safety player's point of view, the winning set's size decreases at each iteration.

B. Transpose graph optimization: simultaneous usage of multiple graphs

Given a graph G and an input target set X, we define the neighborhood of the target set in the transpose graph G^T as follows:

$$N_{G^T} := \{x \in X \mid (x \subseteq Neighbors(G^T, x))\}$$

Specifically, N_{G^T} indicates the set of nodes reachable from a node belonging to the target set through a single edge in the transpose graph.

To provide the reader an intuition of this hybrid approach, we take as an example a reachability game. With reference to formulation 1, the following lines investigate the conditions the nodes have to satisfy to be included in a particular component of $Force_R(X)$, either $Reach\_comp(X)$ or $Safety\_comp(X)$.

Concerning $Reach\_comp(X)$, the set will include all the nodes $v \subseteq V$ that satisfy the following conditions:

a) The node v is controlled by the reachability player (i.e., $v \cap R$).
b) The node v belongs to the set N_{G^T} (i.e., $v \in N_{G^T}$).

As pointed out in section IV-A, it is convenient to employ the transpose graph instead of the straight graph to verify that the condition *b)* is fulfilled. This design choice has the advantage of reducing the computations required for this last check.

On the other hand, $Safety\_comp(X)$ will include all the nodes $v \subseteq V$ that satisfy the following conditions:

a) The node v is controlled by the safety player (i.e., $v \cap S$).
b) The node v belongs to the set N_{G^T} (i.e., $v \in N_{G^T}$).
c) All the outgoing edges of the node $v \subseteq V$ lead to nodes contained in the target set X (i.e., $v \in V \mid E(v) \subseteq X$).

As seen above, the computation of $Safety\_comp(X)$ imposes to verify an additional condition, for which it is strictly necessary to use the straight graph. Hence, we propose a hybrid approach. We employ the transpose graph to check the fulfillment of condition *b)*, while we use the straight graph to check if condition *c)* is verified. We act in this way to optimize the process to the maximum extent possible.

C. Transpose graph optimization: applicative example

To provide the reader the proof of the effectiveness of such optimization, we show an applicative example. Given the arena depicted in figure 1, it is needed to compute the nodes that satisfy the conditions *a)* and *b)* with the least possible computations. However, while employing the transpose graph generally decreases the computational cost of verifying whether condition *b)* is fulfilled, the optimization does not affect the computational cost of checking the condition *a)*. With reference to the figure 1, we distinguish the following scenarios depending on which data structure is used to execute the required computations:

1) *Straight graph usage:* When using the straight graph to compute which nodes satisfy the condition *b)*, we will have the following scenario. The algorithm iterates on all the nodes in the graph controlled by the reachability player, and for each node, checks if the node has an edge to a node contained in the target set (i.e., the region colored in green). Thus, the set of processed nodes will be $V_p = \{v_1, v_3, v_7, v_8\}$.
2) *Transpose graph usage:* On the contrary, when using the transpose graph, the scenario evolves as follows: the algorithm iterates only on the nodes reachable from the target set in the transpose graph. Hence, the set of nodes that must be processed will be restricted to $V_p' = \{v_3, v_8\}$.

Note that, regardless of the used data structure, $Reach\_comp$ will contain the same nodes, i.e., $Reach\_comp = \{v_3, v_8\}$.

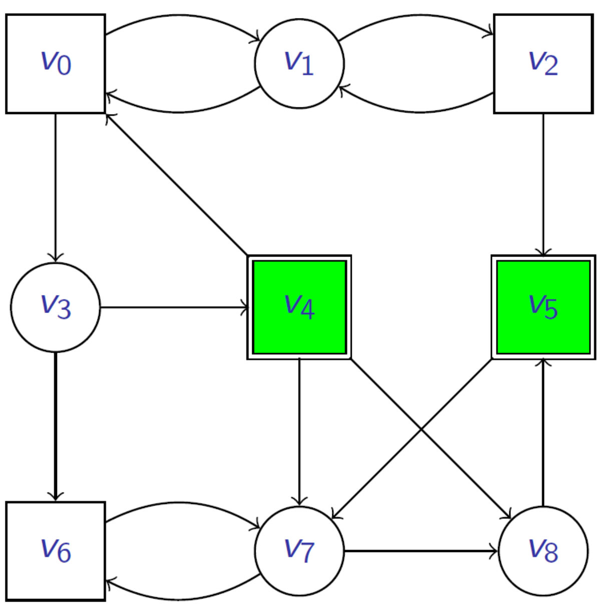

Fig. 1. Graphical representation of the given arena. The round-shaped nodes are the nodes controlled by the reachability player, while the square-shaped nodes indicate the nodes controlled by the safety player. Lastly, the region colored in green identifies the reachability player's target set.

D. Optimizations for reachability games

The following paragraphs discuss the algorithmic optimizations designed for solving reachability games more efficiently. The integration of these optimizations into the *multiple-perspective* algorithm has led to improve the efficiency of the *step backward*, which corresponds to solve the game from the reachability player's point of view.

E. Optimizations for reachability games: Current set optimization

Bearing in mind the formulation (1), we define the *Current set* as *the $Force_R$ computed in the previous iteration of the algorithm*. Concisely, it denotes the set of nodes added in the last iteration to the reachability player's winning set. From this observation follows that the Current set is generally a subset of the target set. This other optimization improves the efficiency of the computation of $Reach\_comp$ and can be combined efficaciously with the transpose graph optimization. Precisely, this optimization consists of verifying the fulfillment of the conditions *a)* and *b)* exclusively for the nodes reachable from the Current set, instead of verifying whether these conditions hold for the set of nodes reachable from the entire target set.[2] In fact, we observe that the nodes candidate to enter the target set are only the nodes that have not been considered yet (i.e., the nodes reachable from the target set through a single edge). This occurs because the nodes added in the previous iteration (i.e., the nodes reachable from the target set through two edges) fall necessarily within the following cases:

1) *The node is not reachable from the target set with a single edge:* The node has not been added to the target set in the previous iteration.
2) *The node is not reachable from the target set through two edges:* The node must be necessarily added to the force set (and thus to the target set) in the previous iteration. In fact, since the graph edges do not change between the iterations, it is impossible that a node was not reachable from the target set through a single edge in the previous iteration but that becomes reachable in the current iteration.

For the reasons presented in this paragraph, it is convenient to verify whether the conditions *a)* and *b)* hold *only for the nodes contained in the Current set*.

Figure 2 provides a graphical explanation of the optimization presented above. With reference to the figure, suppose the algorithm is processing the node *u*. The only candidate to the entrance in the force set we would like to consider are the nodes *a* and *b*, circled in red. The reasons are the following. The nodes *c* and *d* cannot reach the node *u*, so they will be taken into account when processing the node *u'*. On the other hand, the nodes *e, f, and g* could be taken into account in the current passage, but they are already in the winning set; hence adding them into the force set would not be beneficial. More precisely, it would be detrimental because the computational cost of performing the union would increase for considering some nodes that are already in the winning set.

F. Optimizations for reachability games: Processed list optimization

Keeping in mind the formulation (1), we first introduce the following points:

- We suppose to solve the game as a safety game.[3] Hence, we denote the winning set for the safety player as W, and its complement, the losing set, as L.

[2] Please note that we are reasoning by using the transpose graph. Hence, instead of stating that a node v *can reach* the target set X, we state that a node v *is reachable from* the target set X.

[3] We think from this perspective to be consistent with the code provided. In fact, the optimization in question is integrated in the *multiple-perspective algorithm*, which starts solving the game from the safety player's perspective. In doing so, the explanation is consistent with the code provided.

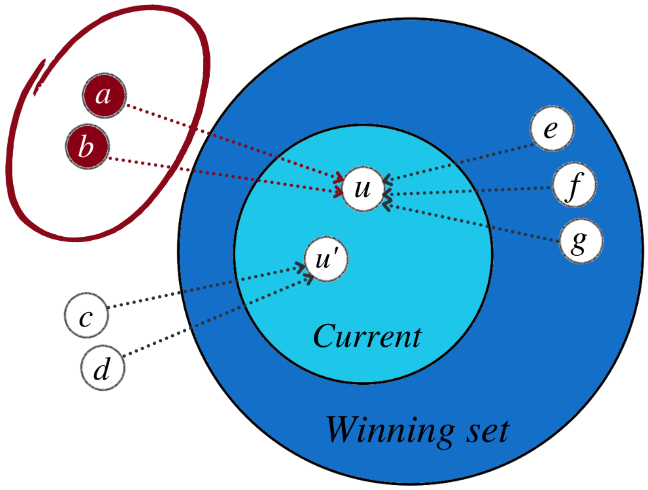

Fig. 2. Graphical representation of the functioning of the Current set optimization.

- In the present paragraph, we will indicate with V_S the *set nodes controlled by the safety player* and with *SCL* the *set of nodes controlled by the safety player that are solely connected with nodes belonging to L.*

The present optimization is related to the computation of $Safety\_comp$ in the function *FORCE_SAFE_BACKWARD* of the *multiple-perspective* algorithm. The function's goal is to compute the set *SCL*.

The first step is to verify whether the conditions *a)* and *b)* hold. Instead of examining the whole set V_S, we examine only the nodes that can reach a losing node through a single edge by exploiting the transpose graph, as explained in the previous paragraphs. At this point, it is necessary to verify whether the nodes considered satisfy the condition *c)*, i.e., to identify all the nodes whose outgoing edges lead to nodes belonging to L. Specifically, we apply the following procedure.

For each node $u \in L$, we examine all the nodes controlled by the safety player solely connected with $u \in L$ or with another node $u' \in L$. We conduct this check by iterating through the set V_S. However, it may occur that a node v is encountered, firstly, as a neighbor of the node u and, secondly, of the node u'. Since the result of the operations carried out will be independent of the fact that the node v is a neighbor either of the node u or the node u', we insert each considered node into *Processed*, a list that contains the processed nodes. In doing so, once a node has been already examined as the neighbor of a node u, it will not be considered again, if it is encountered as the neighbor of another node u'.

G. Safety-reachability duality: improved purely backward algorithm

The algorithm presented in this paragraph follows strictly the logic of the *purely backward* algorithm, but it integrates the optimizations discussed above to improve its performance. In the code provided, the implementation of this optimized version of the algorithm is applied to reachability games. However, the algorithm can also solve safety games by simply returning the complement of the computed winning set. We also provide below the code of the function *NAIVE_FORCE_REACH*, which is the main target of most of the optimizations. We encourage the reader to compare this function with the optimized *FORCE_REACH* integrated into the *improved backward algorithm* to fully understand why the presented optimizations are particularly effective.

H. Optimizations for reachability games: multiple-perspective algorithm

The present algorithm combines the two logics underpinning the algorithms previously presented: *pure forward* and *improved pure backward*. The fundamental idea underlying the design of the present algorithm is to take the best from both approaches. Precisely, at each iteration, the algorithm determines whether it is convenient to tackle the problem from either the reachability or safety player's point of view, depending on a heuristic based on the winning set's size. As a design choice, the algorithm starts to solve the game as a safety game, thus taking as input the target set for the safety player. At each iteration, it determines if it is convenient to proceed *backward* (i.e., solving the problem from the reachability player's point of view) or *forward* (i.e., solving the problem from the safety player's point of view). In fact, once computed the safety player's winning set for the current iteration, the algorithm computes automatically the other player's target set, which is, symmetrically, named as the *losing set*, finally allowing to switch the point of view, if needed.

I. Multiple-perspective algorithm: the employed heuristic

Since the complexity of computing either the sets of both players is dominated by the number of nodes in the winning set, the algorithm determines whether it is convenient to perform the *forward* or *backward* steps depending on the winning set's size. More precisely, whenever the winning set's size is less than or equal to half the number of the nodes in V, it is convenient to perform the forward step; otherwise, it is convenient to execute the backward step. Note that the actual complexity of each step depends on the degree of the nodes involved in the computation. Hence, this criterion is not an accurate indicator, but it is just used as a heuristic. Lastly, emphasis must be placed on the following point: *taking a step instead of another does not undermine the correctness of the algorithm*. To provide the reader a less formal but immediate intuition, we point out that changing the point of view at each iteration can be viewed as generating at each iteration a new game, in which the target set given in input is the winning set returned by the step executed at the previous iteration. In this respect, each iteration is independent of the previous one, hence switching between the two points of view as the game progresses does not affect the algorithm's correctness.

V. EXPERIMENTAL PHASE

Concisely, the experimental phase is structured as follows. In the first place, the script automatically generates an arena (i.e., a graph), where it launches multiple algorithms that solve the same game. At the end of each execution, it collects their results to provide a set of statistics to the user. Precisely, we benchmark the proposed algorithms by measuring the time required to solve the same game. Concerning the experiments' structure, we have made the design choice to not include the improved version of the backward algorithm in the battery of tests proposed. The reason is the following. The design of the improved algorithm to solve the reachability game has been undoubtedly propaedeutic to the design of the *multiple-perspective* algorithm. However, we have preferred to evaluate the combined action of the multiple optimizations techniques employed than testing the effects of every single optimization in a wide range of scenarios. This design choice lies in the fact that all the optimizations proposed are not mutually exclusive (i.e., using a technique does not preclude applying another technique). Specifically, each experiment envisages a run of the following algorithms:

a) *Naive purely forward algorithm*
b) *Naive purely backward algorithm*
c) *Multiple-perspective algorithm*

Moreover, comparing the *multiple-perspective* algorithm with the naive versions of the algorithms for solving reachability and safety games is helpful to provide the reader a concise but practical proof of the effectiveness of the proposed approach. Since all the proposed algorithms, starting with the naive versions, are guaranteed to solve reachability and safety games, the criterion through which we benchmark the previously by measuring the time required to solve the same game.

A. Presentation of the results

The present subsection provides a concise and practical overview of the different algorithms' performance during the benchmark process. Before presenting the results, we point out a fundamental design choice regarding the evaluation method: *all the algorithms have provided the same inputs regardless of their inner functioning*. Specifically, to evaluate the actual effectiveness of employing a more sophisticated approach, the transpose graph is not given as input to the multiple-perspective algorithm, which has to generate it on its own. Please note that *the time required by the algorithm to solve the game also includes the time required for generating the transpose graph*. This design choice concerning the evaluation method has been carried for two distinct reasons. Firstly, to obtain unbiased results and, secondly, to determine whether it is convenient to use the multiple perspective algorithm, although it has to autonomously generate at runtime the transpose graph, which is indeed a computationally expensive procedure when handling massive graphs. In other terms, we aim to verify whether the multiple-perspective algorithm performs well, at least as its naive counterparts, despite the additional burden of generating the transpose graph. Since the employed threshold that regulates the switch of point of view has been set to half of the total number of nodes, we aimed to investigate in which cases the algorithm performs statistically better than its naive counterparts. In the items below, we briefly summarize the content of the tables that report the obtained results.

1) Table I presents the results for games on graphs with a number of nodes ranging from 100 and 1000 and with a target_safe_ratio was randomly ranging between 0.01 and 0.5.
2) Table II shows the results for games on graphs with a number of nodes ranging from 100 and 1000 and with a target_safe_ratio was randomly ranging between 0.5 and 0.1.
3) Table III illustrates the results for games on graphs with a number of nodes ranging from 100 and 1000 and with a target_safe_ratio was randomly ranging between 0.01 and 1.0.
4) Table IV presents the results for games on graphs with a number of nodes ranging from 5000 and 6500 and with a target_safe_ratio was randomly ranging between 0.01 and 1.0.

The experiments performed under settings (1) and (2) aimed to verify the performances of the multiple perspective algorithm when the algorithm was forced to perform only forward steps (case 1) and when the algorithm was obliged to perform at least one backward step (case 2). On the other hand, the experiments conducted in circumstances (3) and (4) intended to examine the algorithm's performance under general settings to test whether applying the multiple perspective algorithm was statistically convenient in medium-size and considerably large graphs, respectively.

B. Discussion of the results

With reference to the presented results, it is possible to make the following observations:

- Under the settings (1) [Table I], the multiple-perspective algorithm performs statistically better than both the forward and backward algorithms. This result is presumably related to the processed list optimization that enables the multiple-perspective algorithm to avoid performing the same operation several times. Please refer section IV for further details.
- Under the settings (2) [Table II], the multiple-perspective algorithm performs statistically better than both the forward and backward algorithms eight times out of ten. On the other hand, in two cases, the forward algorithm performs better than the multiple-perspective algorithm. However, from the magnitude of the percentages, we can see that the difference in terms of performances in these last two cases is not particularly pronounced. We hypothesize that this last result is because the multiple-perspective algorithm has two additional costs when solving games. The first one consists of the generation of the transpose graph and the second one of the cost

of switching point of view (i.e., computing the complement of the winning set).[4] In other terms, the multiple-perspective algorithm is slower than its naive counterparts when the time required to compute the additional data structures exceeds the time saved through the carried-out optimizations.

- The experiments conducted under the settings (3) [Table III] and (4) [Table IV] demonstrate that employing the multiple-perspective algorithm performs better than its naive counterparts on the average case. In fact, the cases in which the multiple-perspective algorithm is slower than the naive ones show that the difference in terms of the required time is not particularly relevant, but there are some cases in which the algorithm profoundly outperforms both the naive counterparts (e.g., Table III - experiment 1, Table IV - experiment 10).
- From all the experiments conducted, it is possible to see that the multiple-perspective algorithm profoundly outperforms the backward algorithm in every conducted experiment. This result is plausibly related to the fact that most of the optimizations proper of the multiple-perspective algorithm have been designed to improve the algorithm's capabilities to solve reachability games; hence the multiple-perspective algorithm is more efficient under these settings.
  Moreover, we remark another crucial factor. As a design choice, the game starts as a safety game; hence it is given as input the target_set for the safety player. In this respect, it is clear that the backward algorithm is disadvantaged because it has to compute the complement of the target_set and then solve the game as a reachability game. Please note that the multiple perspective algorithm experiences the same disadvantage if, at the start, the size of the winning set is greater than half of the total number of nodes, but it succeeds in performing better than the backward algorithm thanks to the implemented optimizations.

VI. Conclusions

This paper has addressed the tasks of implementing multiple algorithms to solve reachability and safety games by exploiting their inner duality and building a framework for testing purposes. In the first section, we have introduced the theoretical compendium accompanying this work. The second section has presented the design choices related to the construction of the arenas. In the third section, we have provided a concrete formulation of reachability and safety games, as well as the algorithms to solve them with a view to the implementation. The fourth section has introduced a novel, personally designed algorithm that combines the logic underpinning reachability and safety games and includes various optimizations to increase its efficiency. Finally, the fifth section has presented a wide range of experiments and has discussed the obtained results.

[4] Please note that the algorithm can switch the point of view only a single time during a game, according to the chosen heuristic.

References

[1] Erich Gradel and Wolfgang Thomas. Automata, logics, and infinite games: a guide to current research. *Springer*, 2002.

[2] Wolfgang Thomas. Infinite games and verification. In *International Conference on Computer Aided Verification*, pages 58–65. Springer, 2002.

[3] Kousha Etessami. Algorithmic game theory and applications: games on graphs. https://www.inf.ed.ac.uk/teaching/courses/agta/lec12_11.pdf, 2014. Online; accessed 11 June 2021.

[4] Yurii Khomskii. Infinite games: lecture notes. https://www.math.uni-hamburg.de/home/khomskii/infinitegames2010/Infinite%20Games%20Sofia.pdf, 2010. Online; accessed 11 June 2021.

[5] Martin Zimmermann . Infinite games: lecture notes. https://www.react.uni-saarland.de/teaching/infinite-games-16/lecture-notes.pdf, 2016. Online; accessed 11 June 2021.

$Experiment\_label$	$Total\_nodes$	$Total\_edges$	$Target\_nodes$	$Safety\_nodes$	$Reachability\_nodes$
1	557	768	77	279	278
2	148	413	72	74	74
3	656	1811	312	327	329
4	910	4134	218	454	456
5	800	2405	231	399	401
6	260	345	86	130	130
7	356	495	167	179	177
8	965	2656	198	482	483
9	319	1422	139	159	160
10	674	2250	291	337	337

$Experiment\_label$	FW_time	BW_time	MP_time	$Time\_saving\_wrt\_FW$	$Time\_saving\_wrt\_BW$
1	0.0011s	0.0225s	0.0007s	31.68%	96.71%
2	0.0010s	0.0018s	0.0007s	35.50%	62.76%
3	0.0026s	0.0281s	0.0025s	2.87%	91.11%
4	0.0027s	0.0557s	0.0024s	10.74%	95.66%
5	0.0027s	0.0438s	0.0023s	13.72%	94.69%
6	0.0008s	0.0050s	0.0004s	48.05%	91.29%
7	0.0013s	0.0068s	0.0009s	26.61%	86.32%
8	0.0032s	0.0650s	0.0024s	26.21%	96.34%
9	0.0012s	0.0059s	0.0010s	17.82%	83.77%
10	0.0031s	0.0324s	0.0024s	21.25%	92.52%

TABLE I
FIRST EXPERIMENTS BATTERY. NUMBER OF NODES RANGING BETWEEN [100, 1000], TARGET_SAFE_RATIO RANGING BETWEEN [0.1-0.5].

$Experiment\_label$	$Total\_nodes$	$Total\_edges$	$Target\_nodes$	$Safety\_nodes$	$Reachability\_nodes$
1	220	461	219	111	109
2	110	512	108	55	55
3	911	2140	810	455	456
4	130	414	127	65	65
5	266	598	133	134	132
6	116	161	96	59	57
7	269	629	218	135	134
8	191	675	173	95	96
9	164	493	145	82	82
10	148	419	145	74	74

$Experiment\_label$	FW_time	BW_time	MP_time	$Time\_saving\_wrt\_FW$	$Time\_saving\_wrt\_BW$
1	0.0013s	0.0013s	0.0010s	18.79%	23.99%
2	0.0004s	0.0004s	0.0004s	1.06%	12.34%
3	0.0130s	0.0196s	0.0143s	-10.33 %	26.75%
4	0.0005s	0.0007s	0.0004s	1.44%	32.25%
5	0.0007s	0.0037s	0.0006s	3.96%	82.61%
6	0.0004s	0.0007s	0.0004s	11.17%	46.40%
7	0.0012s	0.0021s	0.0015s	-17.69 %	32.20%
8	0.0010s	0.0014s	0.0008s	20.63%	43.66%
9	0.0009s	0.0012s	0.0008s	10.43%	23.59%
10	0.0007s	0.0005s	0.0005s	30.19%	4.15%

TABLE II
SECOND EXPERIMENTS BATTERY. NUMBER OF NODES RANGING BETWEEN [100, 1000], TARGET_SAFE_RATIO RANGING BETWEEN [0.5,1.0].

$Experiment\_label$	$Total\_nodes$	$Total\_edges$	$Target\_nodes$	$Safety\_nodes$	$Reachability\_nodes$
1	996	2432	23	26	970
2	573	1734	119	30	543
3	789	2882	218	361	428
4	514	914	55	112	402
5	529	2517	2	196	333
6	174	420	49	49	125
7	502	2167	77	26	476
8	793	2768	357	614	179
9	313	489	24	9	304
10	203	445	196	11	192

$Experiment\_label$	FW_time	BW_time	MP_time	$Time\_saving\_wrt\_FW$	$Time\_saving\_wrt\_BW$
1	0.0012s	0.0096s	0.0003s	76.19%	97.01%
2	0.0004s	0.0070s	0.0004s	−4.65 %	94.29%
3	0.0026s	0.0389s	0.0023s	13.79%	94.17%
4	0.0005s	0.0127s	0.0005s	3.10%	96.11%
5	0.0002s	0.0106s	0.0001s	14.94%	98.75%
6	0.0005s	0.0020s	0.0005s	1.72%	74.65%
7	0.0004s	0.0056s	0.0003s	8.29%	94.07%
8	0.0033s	0.0429s	0.0033s	1.49%	92.32%
9	0.0001s	0.0026s	0.0001s	11.52%	95.06%
10	0.0006s	0.0011s	0.0005s	15.38%	50.55%

TABLE III
THIRD EXPERIMENTS BATTERY. NUMBER OF NODES RANGING BETWEEN [100, 1000], TARGET_SAFE_RATIO RANGING BETWEEN [0.1,1.0].

$Experiment\_label$	$Total\_nodes$	$Total\_edges$	$Target\_nodes$	$Safety\_nodes$	$Reachability\_nodes$
1	5341	9515	1226	4587	754
2	6368	8771	1037	4883	1485
3	5887	28 147	1887	5452	435
4	5990	8531	148	1111	4879
5	6275	25 183	325	1038	5237
6	5667	20 984	2613	2933	2734
7	5961	23 184	1461	5679	282
8	5543	7214	554	3868	1675
9	5318	16 082	2270	1648	3670
10	5581	20 494	1236	61	5520

$Experiment\_label$	FW_time	BW_time	MP_time	$Time\_saving\_wrt\_FW$	$Time\_saving\_wrt\_BW$
1	0.0494s	2.8728s	0.0409s	17.06%	98.57%
2	0.0689s	4.3557s	0.0596s	13.40%	98.63%
3	0.0479s	3.7482s	0.0438s	8.68%	98.83%
4	0.0101s	1.1958s	0.0113s	−12.04 %	99.06%
5	0.0196s	1.6087s	0.0177s	9.72%	98.90%
6	0.1544s	1.8811s	0.1561s	−1.08 %	91.70%
7	0.0275s	3.9568s	0.0231s	16.22%	99.42%
8	0.0368s	3.0642s	0.0316s	14.10%	98.97%
9	0.1412s	1.3840s	0.1292s	8.48%	90.66%
10	12.5055s	0.7117s	0.0048s	99.96%	99.33%

TABLE IV
FOURTH EXPERIMENTS BATTERY. NUMBER OF NODES RANGING BETWEEN [5000, 6500], TARGET_SAFE_RATIO RANGING BETWEEN [0.1,1.0].

Appendix: Pseudocode

NAIVE_FORCE_REACH

Unoptimized version of the FORCE_REACH function.

function NAIVE_FORCE_REACH(G^T, V_R, C)
Input:
G^T : (V, E^T) s.t. V set of nodes, E^T set of edges obtained by inverting the edges in G
$V_R \subseteq$ V //Set of nodes controlled by the reachability player
Q $\subseteq$ V //Target set for the reachability player
Output:
F $\subseteq$ V //F = $\{$v $\in V_R$: $\exists$ u $\in$ Q $|$ (u,v) $\in E^T\}$
  F $\leftarrow \emptyset$
  for u $\in$ Q **do**
    for u $\in V_R$ **do**
      if u $\in$ *Neighbors(*u, G*)* **then**
        F $\leftarrow$ F $\cup$ $\{$v$\}$
      end if
    end for
  end for
  return F
end function

Improved purely backward algorithm

Input:
G : (V, E) s.t. V set of nodes, E set of edges bw nodes in V
T $\subseteq$ V: Target set for the reachability player
V_S, V_R := Sets of nodes $v \in V$ s.t. $V_S \cup V_R = V$ [4]
Output:
Win $\subseteq$ V: Winning set for the reachability player

Begin:
$G^T = (V, E^T) \leftarrow$ transpose(G)
Q $\leftarrow$ T // Q:= Target set for the reachability player
C $\leftarrow$ T // C:= Set of nodes found reachable in the last iteration
$F_{reach} \leftarrow$ FORCE_REACH(G^T, V_R, C)
$F_{safe} \leftarrow$ FORCE_SAFE(G, G^T, V_S, Q)
F $\leftarrow F_{reach} \cup F_{safe}$
Q' $\leftarrow$ Q $\cup$ F
while Q != Q': **do**
  C $\leftarrow$ F
  Q $\leftarrow$ Q'
  $F_{reach} \leftarrow$ FORCE_REACH(G^T, V_R, C)
  $F_{safe} \leftarrow$ FORCE_SAFE(G, G^T, V_S, Q)
  F $\leftarrow F_{reach} \cup F_{safe}$
end while
return Q
End

Improved purely backward algorithm: auxiliary functions

function FORCE_REACH(G^T, V_R, C)
Input:
G^T : (V, E^T) s.t. V set of nodes, E^T set of edges obtained by inverting the edges in G
$V_R \subseteq$ V //Set of nodes controlled by the reachability player
C $\subseteq$ V //Set of nodes found reachable in the last iteration
Output:
F $\subseteq$ V //F = $\{$v $\in V_R$: $\exists$ u $\in$ C $|$ (u,v) $\in E^T\}$ [5]
  F $\leftarrow \emptyset$
  for u $\in$ C **do**
    F $\leftarrow$ F $\cup$ (*Neighbors(u,* $G^T \cap V_R$*))*
  end for
  return F
end function

function FORCE_SAFE(G, G^T, V_S, Q)
Input:
G : (V, E) s.t. V set of nodes, E set of edges bw nodes in V
G^T : (V, E^T) s.t. V set of nodes, E^T set of edges obtained by inverting the edges in G
$V_S \subseteq$ V //Set of nodes controlled by the safety player
Q $\subseteq$ V //Target set for the reachability player
Output:
F $\subseteq$ V //F = $\{$v $\in V_S$ $|$ $\{$ $u \,|\, (v, u) \in$ E $\}$ $\subseteq$ Q $\}$ [6]
  for u $\in$ Q **do**
    N $\leftarrow$ *Neighbors(*u, G^T*)* $\cap V_S$
    for v $\in$ N: **do**
      if *Neighbors(*v, G*)* $\subseteq$ Q **then**
        F $\leftarrow$ F $\cup$ $\{$v$\}$
      end if
    end for
  end for
  return F
end function

[4]We define V_S as the set of nodes controlled by the safety player, while we indicate with V_R the set of nodes controlled by the reachability player.

[5]We indicate with F all the nodes controlled by the reachability player that have an edge to a node in C. Please notice we employ the transpose graph as an optimization.

[6]We indicate with F all the nodes controlled by the safety player that have an edge to a node in Q. Please notice we employ the direct graph in this case.

Multiple perspective algorithm

Input:
$G : (V,\, E)$ s.t. V set of nodes, E set of edges bw nodes in V
$V_S \subseteq V$: Set of nodes controlled by the safety player
$V_R \subseteq V$: Set of nodes controlled by the reachablity player
Target_safe: Set of target nodes for the safety player
Threshold: Threshold that regulates the strategy shift
Output:
Win: The winning set of nodes for the safety player

Begin:
Win $\leftarrow$ Target_safe
Lose $\leftarrow$ $V \setminus$ Target_safe
Last_force_reach $\leftarrow$ Lose
while True **do**
  if $Card(Win)$[6] $\leq$ Threshold **then**
    F $\leftarrow$ STEP_FORWARD($G = (V,\, E),\, Win$)
    Win_new := Win $\cap$ F
    Win $\leftarrow$ Win_new
    Lose $\leftarrow$ Lose $\cup$ [Win $\triangle$ F]
  end if
  else //$Card(Win) >$ Threshold
    F$\leftarrow$STEP_BACKWARD($G,\, G^T,\, Lose,\, Last\_force\_reach$)
    Win $\leftarrow$ Win $\setminus$ F
    Lose $\leftarrow$ Lose $\cup$ F
    Last_force_reach $\leftarrow$ F
  if $Card(Win\_new) = Card(Win)$ [7] **then**
    return Win
  end if
end while

function STEP_FORWARD($G = (V,\, E),\, Win$):
  F_s $\leftarrow$ FORCE_SAFE_FORWARD(Win)
  F_r $\leftarrow$ FORCE_REACH_FORWARD($Win \cap V_R$)
  F $\leftarrow$ F_s $\cup$ F_r
  return F
end function

function STEP_BACKWARD($G = (V,\, E),\, G^T,\, Lose,\, Last\_force\_reach$):
  F_r $\leftarrow$ FORCE_REACH_BACKWARD($G^T,\, Lose,\, Last\_force\_reach$)
  F_s $\leftarrow$ FORCE_SAFE_BACKWARD($G,\, G^T,\, Lose$)
  F $\leftarrow$ F_s $\cup$ F_r
  return F
end function

[6]The operator $Card(.)$ returns the cardinality of a given input set.
[7]For the sake of efficiency, we compare the sets' cardinalities instead of comparing their elements to verify if the fix-point has been reached.
[8]If a safe node is isolated, then it is clearly safe.

Multiple perspective algorithm: auxiliary functions

function FORCE_SAFE_FORWARD($G = (V,\, E),\, Win$):
  F $\leftarrow \emptyset$
  Win_safety $\leftarrow$ Win $\cap\, V_S$:
  for u $\in$ Win_safety **do**
    if $Card(Neighbors(u, G)) = 0$ [8] **then**
      F $\leftarrow$ F $\cup\, \{u\}$
    end if
    if ($\exists\, u \in Neighbors(u,\, G) \,|\, v \in Win$): **then**
      F $\leftarrow$ F $\cup\, \{u\}$
    end if
  end for
  return F
end function

function FORCE_REACH_FORWARD($G = (V,\, E),\, Win$):
  F $\leftarrow \emptyset$
  Win_reach $\leftarrow$ Win $\cap\, V_R$:
  for u $\in$ Win_reach: **do**
    if $Neighbors(u,\, G) \subseteq$ Win **then**
      F $\leftarrow$ F $\cup\, \{u\}$
    end if
  end for
  return F
end function

function FORCE_REACH_BACKWARD($G^T = (V,\, E^T),\, V_R,\, Lose,\, Last\_force\_reach$):
  F $\leftarrow \emptyset$
  for u $\in$ Last_force_reach: **do**
    F $\leftarrow$ F $\cup$ ($Neighbors(u,\, G^T) \cap V_R$)
  end for
  return F
end function

function FORCE_SAFE_BACKWARD($G^T = (V,\, E^T),\, Lose$):
  F $\leftarrow \emptyset$
  Processed $\leftarrow \emptyset$
  for u $\in$ Lose: **do**
    for v $\in$ ($Neighbors(u,\, G^T) \cap V_S$) **do**
      if v $\notin$ (Processed $\cup$ Lose) **then**
        if $Neighbors(v,\, G) \subseteq Lose$ **then**
          F $\leftarrow$ F $\cup$ {v}
        end if
        Processed $\leftarrow$ Processed $\cup$ {v}
      end if
    end for
  end for
  return F
end function