SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER, CONTROL AND MANAGEMENT ENGINEERING

Reasoning Agents Report

GAMES ON GRAPHS

REACHABILITY, SAFETY, BÜCHI, CO-BÜCHI, AND PARITY GAMES

Professor:
Giuseppe De Giacomo

Supervisor:
Giuseppe Perelli

Students:
Ionut Marian Motoi
Leonardo Saraceni
Renzo Cherubino
Arthur Back De Luca

Academic Year 2020/2021

Contents

1 Introduction

Many real-world problems in computer science can't be dealt with by means of programs that start and then terminate at some point. In those cases, the system must be able to run for long periods of time (ideally infinite), while contrasting an antagonistic environment. An example is given by the ABS system of cars, which has to constantly read the wheel speed, and apply the appropriate correction to ensure uniformity in the wheel movement. In the last few years, game-theoretic frameworks have been widely employed since they are well suited to represent the adversarial interaction between the system and its environment. In a more general way, a problem of this type can be modelled as a game between two agents, both trying to find a finite-state strategy that meets the winning conditions, if one exists. This can be done by using a valid graph, i.e. without dead ends, where each vertex is owned by one of the two players. Those, aim to build a play by moving a token through the graph according to the rules, forming an infinite path. A strategy from a vertex is winning for the Player i, if every play that begins from there and is played according to the strategy, meets the winning conditions of Player i. A game is determined if, from each vertex, one of the players has a winning strategy.

The aim of this project, is to create a flexible tool to solve different type of games, namely reachability, safety, Büchi, co-Büchi and parity. Therefore we provide 5 different type of solvers with the possibility of inserting custom arenas and target sets.

2 Arenas and Strategies

For this project, we focused on solving different types of objectives for 2-players, turn-based, deterministic games. The arena is the main component of a game since it describes the rules and the order the players have to follow. Arenas $A = (V_0, V_1, E)$ are graphically represented as directed graphs made of vertices and edges. Each vertex is assigned to one of the two players: round vertices belong to player 0, while square ones belong to player 1. The edges describe the moves the players can perform.

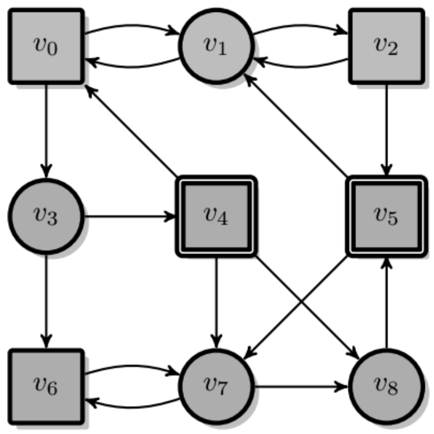

Figure 1: Example of an arena.
The round vertices belong to the Player 0,
while the square vertices belong to the Player 1.

A game proceeds by placing a token at some initial vertex, then the player that owns that vertex moves it in the graph by following one of the outgoing edges. Moreover, since we want to play infinitely long games we only consider **valid** arenas, i.e. there is at least one outgoing edge for each vertex. By doing so, we ensure that the token does not end up in a deadlock. In order to solve games, we sometimes need to restrict our focus to only a small part of the arena, known as the **sub-arena**, which must be valid as well.

Players play in the arena by moving a token through it. After infinitely many moves, the players have produced an infinite path, called a **play** $S \in V^\omega$ in the arena.

The most important aspect of a game are its **strategies** $\sigma : V^* * V_0 \to V$ for Player 0 and $\tau : V^* * V_1 \to V$ for Player 1. In a game, the strategy states how to move the token through the arena. On the one hand, this decision depends on the structure of the arena, which describes the possible moves. On the other hand, it may also depend on the decisions made in the past, on the vertices already visited. Formally, a strategy is a function mapping vertices to a successor. By convention, we denote the strategies for Player 0 by σ and strategies for Player 1 by τ, but in general we refer to an arbitrary Player $i \in \{0, 1\}$ and its opponent Player $i - 1$. In this case we refer to strategies for Player i as σ, and for Player $i - 1$ as τ.

The definition of strategy given before is not feasible for practical uses, therefore we need a weaker one, called **positional strategies** $\sigma : V_0 \to V$ for Player 0 and $\tau : V_1 \to V$ for Player 1. Those don't depend on the whole story of the play, but only on the vertex the token is currently at. Therefore, the strategy will always return the same successor for a given vertex.

Strategies are important since they allow to define the winner of a play. A **winning condition** $W_0 \subseteq V^\omega$ is defined as a subset of the arena plays. If a play is in the winning condition, then Player 0 wins, otherwise Player 1 wins. When playing a game, both players are trying to guarantee a win against the possible moves of the opponent. This notion is formalized under the name of **winning strategy**.

The **winning region** of Player i is defined by collecting all the vertices from which Player i can enforce a win. It is important to notice that the winning regions of the two players are always disjoint.

A **trap** for Player i is defined as a region of an arena which Player i cannot leave without the help of the opponent $i-1$. Therefore the token can be kept in that region indefinitely.

Solving the game G amounts to compute the winning regions and strategies for both players.

3 Games

We consider five types of infinite games, namely reachability and safety games, Büchi and co-Büchi games, and parity games.

3.1 Reachability Games

In a reachability game the set of winning plays for Player 0 is induced by a set R of vertices and consists of all the plays that visit R at least once. The objective of Player 0 is to reach R, while Player 1 has to avoid reaching that region of the arena.

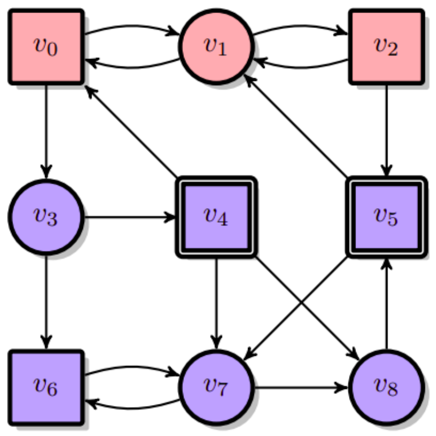

Figure 2: Example of a reachability game.
The winning region of Player 0 is shown in blue,
while the winning region of Player 1 is red.

An example of reachability game is given in Fig.[2], where the reachability set is represented as double-framed vertices. In this game, Player 0 has to reach either v_4 or v_5. It is clear that if the game starts in one of those two nodes then Player 0 has already won since the token is already in the reachability set R. Moreover, Player 0 wins also if he is in a vertex that belongs to himself and that has a successor from which we know it is possible to enforce a win. An example is given by the node v_7, from which Player 0 can move to v_8, and then to v_5, that belongs to R. In general, if Player 1 has a vertex v that only has successors from which Player 0 can enforce a win, then the vertex v is winning for Player 0. Following this mechanism, it is easy to conclude that Player 0 can enforce a win from every vertex in the set $V' = \{v_3, v_4, v_5, v_6, v_7, v_8\}$. All the remaining vertices, namely $\{v_0, v_1, v_2\}$ belong to the set $V \setminus V'$, and therefore Player 1 can use these to keep the token in the region $V \setminus V'$, from which Player 0 can't bring out the token. Thus $V \setminus V'$ is winning for Player 1. In a more formal way, we can define the positional strategies for both players as:

Positional strategy Player 0:

- $\sigma(v_1) = v_0$
- $\sigma(v_3) = v_4$
- $\sigma(v_7) = v_8$
- $\sigma(v_8) = v_5$

Positional strategy Player 1:

- $\tau(v_0) = v_1$
- $\tau(v_2) = v_1$
- $\tau(v_4) = v_8$
- $\tau(v_5) = v_7$
- $\tau(v_6) = v_7$

Both are positional winning strategies for the winning regions $W_0 = V'$ and $W_1 = V \setminus V'$. The values of $\sigma(v_1), \tau(v_4), \tau(v_5)$ and $\tau(v_6)$ can be defined arbitrarily since any choice taken by the player loses. The winning region for Player 0 V' contains all vertices from where Player 0 can attract the token to the reachability set R, and is called the attractor of R. The construction of the attractor is hierarchical, that means new vertices are included at each step, expanding the frontier.

(**Controlled Predecessor**)
We define the controlled predecessor $CPre_i(R)$ as:

$$\begin{aligned} CPre_i(R) = &\{v \in V_i \mid v' \in R \text{ for some successor } v' \text{ of } v\} \cup \\ &\{v \in V_{i-1} \mid v' \in R \text{ for all successors } v' \text{ of } v\} \end{aligned}$$

(**Attractor**) The i-th attractor $Attr_i(R)$ is defined inductively as:

- $Attr_i^0(R) = R$
- $Attr_i^{n+1}(R) = Attr_i^n(R) \cup CPre_i(Attr_i^n(R))$
- $Attr_i(R) = \bigcup_{n \in N} Attr_i^n(R)$

In general, the attractor becomes stationary after at most $|V|$ steps. The attractor is an important concept since the winning region of Player 0 in a reachability game is just the attractor of the reachability set R, while the winning region of Player 1 is the complement of the attractor.

A strategy σ for Player 0 is an attractor strategy, as it attracts the token to the reachability set R. Instead, a strategy τ for Player 1, traps the token away from R, hence is a **trap** for Player 0, as he cannot escape from it, if Player 1 plays according to τ.

The pseudocode for the reachability solver algorithm is presented on the next page.

Algorithm 1 Reachability algorithm

```
function REACHABILITY_SOLVER(arena, R):
    for $V \in R$ do
        $W_0 \leftarrow W_0 \cup V$
        $Queue \leftarrow Queue \cup V$
        if $Player(V) == 0$ then
            $\sigma(V) \leftarrow$ arbitrary successor
    while $Queue \neq \emptyset$ do
        for $node \in Queue$ do
            $Q \leftarrow Predecessor(node)$
            if $Player(Q)) \leftarrow 0$ then
                $W_0 \leftarrow W_0 \cup node$
                $\sigma(Q) \leftarrow node$
            else if $Player(Q) == 1$ then
                $W_1 \leftarrow W_1 \cup node$
                $\tau(Q) \leftarrow node$
    return $W_0, \sigma, W_1, \tau$
```

3.2 Safety Games

Safety is another type of winning condition. While in a reachability game the goal of the Player 0 is to reach a specific part of the arena, in a safety game the Player 0 should not leave a region S made of safe vertices. An important thing to notice is the duality between reachability and safety. In a reachability game, Player 1's goal is to stay forever in a region V\R, that is a safety condition. Dually, in a safety game the goal of Player 1 is to reach the set V\S, that is a reachability condition. By exploiting duality, is possible to turn a reachability game into a safety game and viceversa, simply by swapping the roles of the players (the set of vertices V_0 and V_1) and substituting the winning set by its complement.

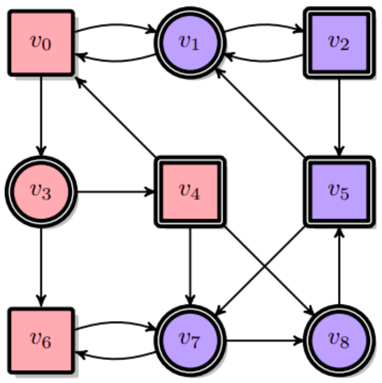

Figure 3: Example of a safety game.
The winning region of Player 0 is shown in blue,
while the winning region of Player 1 is red.

An example is given in Fig.[3], where the elements in the safe set are represented as double-framed vertices. Starting from v_4, this is winning for Player 1 since he can move to unsafe vertex v_0. The previous vertex v_3 is therefore losing since from there Player 0 can only move to v_6 or to v_4 which are both unsafe. Following the reasoning, the winning region for Player 1 simply is its attractor to the region V\S, which is $W_1 = \{v_0, v_3, v_4, v_6\}$. Exploiting duality, the winning region for Player 0 is the complement of W_1: $W_0 = \{v_1, v_2, v_5, v_7, v_8\}$.

Thanks to duality, is possible to transfer all the properties and results obtained for reachability games, to safety games. Therefore, safety games are determined with positional winning strategies.

3.3 Büchi Games

In both reachability or safety games, we study a game where one of the two players has a **reachability condition**. This means that a play is essentially over as soon as one of the two players reaches for the first time a set of vertices, since the winner is determined and all the future moves are irrelevant. This means that at that point the winner is determined and all the future moves are irrelevant. In Büchi games Player 0 has to visit a set of vertices F infinitely often, so we have to consider the infinite play.

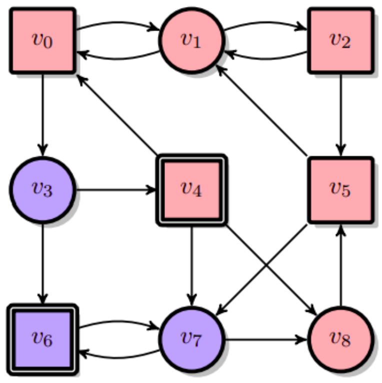

Figure 4: Example of a Büchi game.
The winning region of Player 0 is shown in blue,
while the winning region of Player 1 is red.

An example is given in figure Fig.[4], where the goal of Player 0 is to visit $F = \{v_4, v_6\}$ infinitely often. This is only possible from vertices in $Attr_0(F)$, therefore $V \setminus Attr_0(F) = \{v_0, v_1, v_2, v_5, v_8\} \subseteq W_1$. We start the analysis from the vertex v_4, which belongs to F, and therefore is potentially desirable for Player 0 to reach it. We observe that v_4 is a Player 1 vertex, and has only successors belonging to the winning region of Player 1, thus v_4 is not desirable at all for Player 0 and can be removed from the set F. Considering now v_6, the other vertex in F, which is a Player 1 vertex, the only successor $v7$ is not yet known to be winning for Player 1. Therefore, we can keep v_6 in the recurrence set F. Player 1 wins from every vertex from which Player 0 cannot attract to $F = \{v_6\}$. Consequently $V \setminus Attr_0(F) = \{v_0, v_1, v_2, v_4, v_5, v_8\} \subseteq W_1$. At this point we still have not determined that all the successors of v_6 are in the winning region of Player 1 so we have reached a stationary situation. On the other hand, Player 0 has a winning strategy from the set of vertices that attract to v_6, namely $W_0 = \{v_3, v_6, v_7\}$.

The reasoning to follow is to compute the increasing under-approximations winning region of Player 1 W_1^n, by starting from $F_0 = F$, since at the beginning every vertex is still desirable. At each iteration, we need to find the vertices from which Player 1 can prevent reaching F^n, and add those to the n-th under-approximation W_1^n. Then we remove from the recurrence set F^n all the vertices from which Player 1 can reach W_1^n, since those are winning for Player 1, despite being in F. Then we repeat this construction until the set F^{n+1} becomes stationary.

A pseudocode for the recurrence part is briefly reported here:

Algorithm 2 Büchi solver

```
function BÜCHI_SOLVER(arena, F):
    while $CPre_1(W_1) \neq W_1$ do
        $W_1 \leftarrow V \setminus Attr_0(F)$
        $F \leftarrow F \setminus CPre_1(W_1)$
    $W_0 \leftarrow V \setminus W_1$
    ...
```

3.4 Co-Büchi Games

As safety games were the dual of reachability games, co-Büchi games are the dual of Büchi Games. The goal of Player 0 in a co-Büchi game is to visit a persistence set of vertices C finitely often. Therefore, co-Büchi condition can be seen as a generalization of the safety condition, that allows a finite number of visits to unsafe vertices.

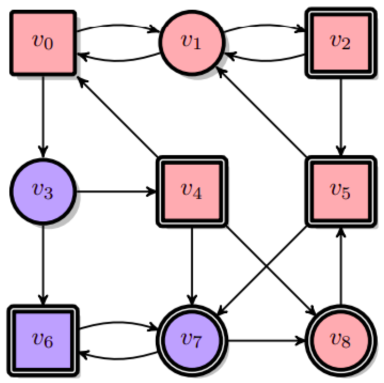

Figure 5: Example of a co-Büchi game.
The winning region of Player 0 is shown in blue,
while the winning region of Player 1 is red.

Exploring the example given in figure Fig.[5], Player 0 wins the game if the vertices outside of its persistence set, i.e. v_0, v_1 and v_3 are visited only finitely often. Therefore intuitively Player 0 wins from the vertices v_6 and v_7 by always moving back and forth, and Player 1 has no way to escape this trap. For this reason, Player 0 wins also from v_3, by moving to v_6 that is winning as stated above. Those are the winning vertices for Player 0, from every other vertex Player 1 wins by playing accordingly to the attractor strategy for v_4 at the vertices v_3, v_2, v_5 and v_8. In fact, Player 1 can positionally move from v_4 to v_0, and from v_0 to v_1. In this way, the vertex v_1 that is not in the persistence set, is visited infinitely often, resulting in a win for Player 1.

3.5 Parity Games

Parity games are a generalization of Büchi games, and are really important in the theory of infinite games, and find a lot of applicability in logics and automata theory. In such games, the vertices of the arena are labeled with a natural number, and the goal of Player 0 is to ensure that the maximum number seen infinitely often is even. Therefore, a generic Player i wins a play ρ if the maximum number seen infinitely often in the play ρ has

parity i. The number on every vertex represents its importance, and large numbers are more important than smaller ones. Even numbers are preferable for Player 0 and odd numbers are desirable for Player 1. For this reason, the numbers assigned to the vertices are usually called **priorities**.

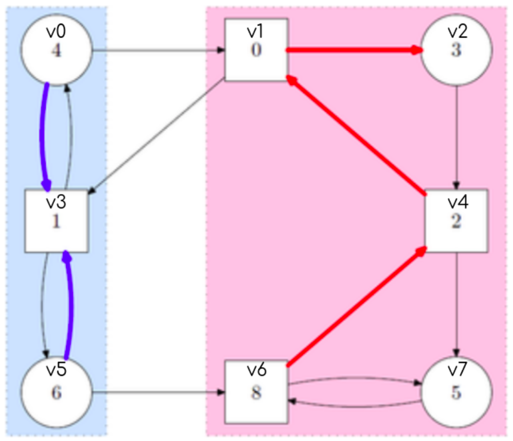

Figure 6: Example of a Parity game.
The winning region of Player 0 is shown in blue,
while the winning region of Player 1 is red.

An example is given in figure Fig.[6]. As we can see, every vertex has a label with the number associated to that vertex. Starting the analysis from the vertex v_0, we can see that from here Player 0 can enforce a win, since he can go to vertex v_3, from which Player 1 can only go to v_0 and v_5. This is a trap, since both those nodes belong to Player 0, that can go back and forth, without allowing Player 1 to escape from this zone, and is winning for Player 0 since he visits infinitely often vertices with an even label (4 for v_0 and 6 for v_5, which are larger than the odd number. All the other nodes are winning for Player 1, since from v_1 he can go to v_2 (which is labeled with the odd number 3), from which Player 0 is forced to v_4 since is the only outgoing edge, and from there Player 1 can go back to v_1 and repeat the loop again. Hence, here Player 0 is trapped. Also v_6 and v_7 are winning for Player 1, since from there the token can always be brought to v_5, from which Player 0 is trapped again.

Following this reasoning we can define the positional winning strategy for both players as:

Positional strategy Player 0:

- $\sigma(v_0) = v_3$
- $\sigma(v_5) = v_3$

Positional strategy Player 1:

- $\tau(v_1) = v_2$
- $\tau(v_4) = v_1$
- $\tau(v_6) = v_4$

For solving parity games we implemented Zielonka's recursive algorithm, reported in [3]. The algorithm is based on a recursive descent on the number of priorities.

Let p be the maximal priority and let $i = p \bmod 2$ be the player associated with the priority. If $p = 0$ then the Player 0 wins from every vertex in the game and $W_0 = V$ and $W_1 = \emptyset$.

Now let $U = \{v \mid \Omega(v) = p\}$ be the set of nodes with priority p and let $A = Attr_i(U)$ be the corresponding attractor of player i. Player i can now ensure that every play that visits A infinitely often is won by player i.

Let $G' = G \setminus A$ be the game from which the nodes in A were removed. We can solve the smaller game G' by recursion and obtain a pair of winning sets W'_i and W'_{1-i}. In case $W'_{1-i} = \emptyset$, also $W_{1-i} = \emptyset$ for the game G.

Otherwise, if $W'_{1-i} \neq \emptyset$, then we compute the attractor $B = Attr_{1-i}(W'_{1-i})$ and remove it from the game to obtain $G'' = G \backslash B$. We also solve the game G'' recursively, obtaining the winning regions W''_i and W''_{1-i}. Finally the winning regions for the game G are obtained as $W_i = W''_i$ and $W_{1-i} = W''_{1-i} \cup B$.

Algorithm 3 Parity solver

```
function PARITY_SOLVER(arena, G):
    $p \leftarrow$ max priority in $G$
    if p = 0 then
        $W_0 \leftarrow V$
        $W_1 \leftarrow \{\}$
        return $W_0, W_1$
    else
        $U \leftarrow$ nodes in $G$ with priority $p$
        $i \leftarrow p \mod 2$
        $A \leftarrow Attr_i(U)$
        $W_0', W_1' \leftarrow$ parity_solver$(G \setminus A)$
        if $W_{1-i}' = \{\}$ then
            $W_i \leftarrow V$
            $W_{1-i} \leftarrow \{\}$
            return $W_i, W_{1-i}$
        $B \leftarrow Attr_{1-i}(W_{1-i}')$
        $W_0'', W_1'' \leftarrow$ parity_solver$(G \setminus B)$
        $W_i \leftarrow W_i''$
        $W_{1-i} \leftarrow W_{1-i}'' \cup B$
        return $W_i, W_{1-i}$
```

4 How to Run the Code

In this section we show how to execute the solver. Is important to notice that the only required argument is --game, since all the others have some default values.

python3 solver.py --game ... --arena ... --target ...

This has associated 3 possible flags.

- **--game:** here is possible to select the game we want to solve among reachability, safety, Büchi, co-Büchi and parity.
- **--arena:** allows to select the arena to play into, if none one is passed by default. We provide a total of 8 arenas.
- **--target:** this argument passes the set of interest of the game chosen.

Here we provide some **templates** of code to execute each game:

- python3 solver.py --game reachability --arena assets/arena0.txt
  --target 4, 5
- python3 solver.py --game safety --arena assets/arena0.txt
  --target 1, 2, 3, 4, 5, 7, 8
- python3 solver.py --game buchi --arena assets/arena0.txt
  --target 4, 6
- python3 solver.py --game cobuchi --arena assets/arena0.txt
  --target 2, 4, 5, 6, 7, 8
- python3 solver.py --game parity --arena assets/arenawiki.txt

5 Implementation details

In our implementation, a python class is used to represent the arena. Each object of this type, contains all the nodes, together with the successors, the predecessors and the importance of all of these. Moreover, there are a lot of methods that are necessary in order to properly design the arena to solve. Here are listed all the methods:

- **add_node** $\rightarrow$ adds a node to the arena with the relative player
- **add_successor** $\rightarrow$ adds a successor to the successors list of the given node
- **add_predecessor** $\rightarrow$ adds a predecessor to the predecessors list of the given node
- **get_nodes** $\rightarrow$ returns the list of all the nodes in the arena
- **get_successors** $\rightarrow$ returns the list of the successors of the given node
- **get_predecessors** $\rightarrow$ returns the list of the predecessors of the given node
- **get_player** $\rightarrow$ returns the player to which the node belongs
- **set_player** $\rightarrow$ assigns the node to a player
- **set_importance** $\rightarrow$ adds the importance (priority) to a node, that is used only in parity games
- **get_importance** $\rightarrow$ returns the importance of the given node
- **get_sub_arena** $\rightarrow$ creates a sub-arena from the current game. The sub-game will contain all nodes in the provided set.
- **get_max_importance** $\rightarrow$ returns the value of the highest importance

In order to verify the scalability of the various solvers, we created a function called **generate_random_arena**, that takes as inputs the number of nodes n, the maximum number of successors that can be assigned to each node s, and the maximum importance that a node can have p. Then the function generates a random arena following the parameters passed to it. To verify the results, 5 arenas have been created with the following setups:

- **arena_1** $\rightarrow$ n = 10, p = 10, s = 10
- **arena_2** $\rightarrow$ n = 100, p = 100, s = 10
- **arena_3** $\rightarrow$ n = 1000, p = 1000, s = 10
- **arena_4** $\rightarrow$ n = 10000, p = 10000, s = 10
- **arena_5** $\rightarrow$ n = 100000, p = 100000, s = 10

The results obtained by running the benchmarks, shows that by increasing the number of nodes in the arena of a multiplicative factor 10, the time required to solve the game doesn't increases in an exponential way.

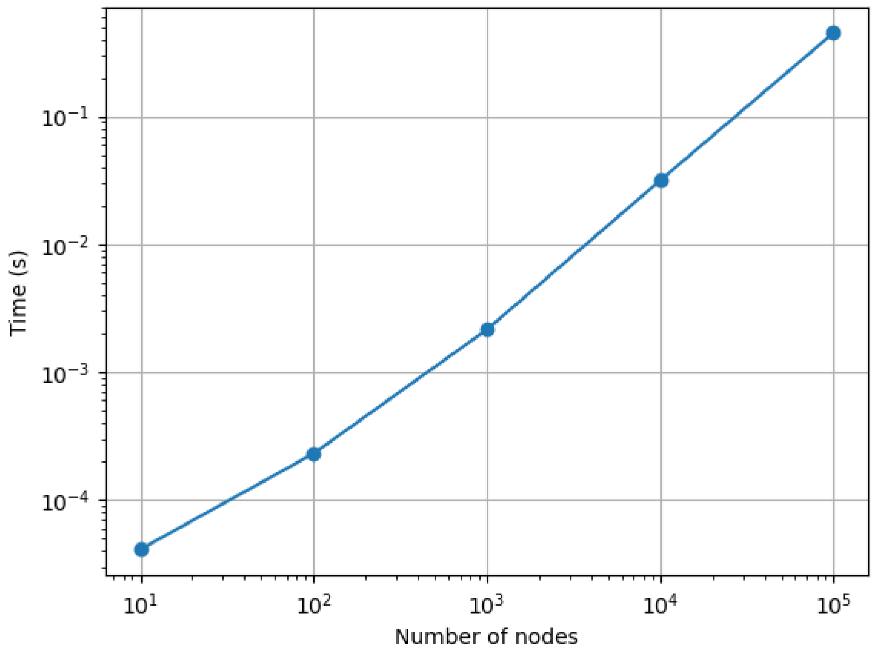

Figure 7: Reachability

Reachability times:

- **Arena 10 nodes** → 4.101e-05 (s)
- **Arena 100 nodes** → 2.303e-4 (s)
- **Arena 1000 nodes** → 2.147e-3 (s)
- **Arena 10000 nodes** → 3.149e-2 (s)
- **Arena 100000 nodes** → 4.491e-1 (s)

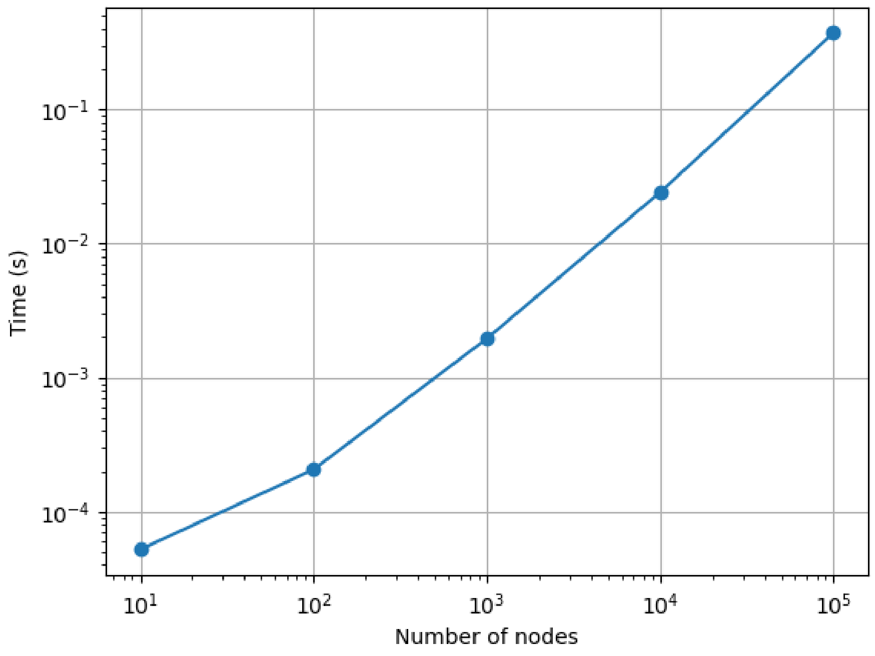

Figure 8: Safety

Safety times:

- **Arena 10 nodes** $\rightarrow$ 5.221e-05 (s)
- **Arena 100 nodes** $\rightarrow$ 2.058e-4 (s)
- **Arena 1000 nodes** $\rightarrow$ 1.951e-3 (s)
- **Arena 10000 nodes** $\rightarrow$ 2.412e-2 (s)
- **Arena 100000 nodes** $\rightarrow$ 3.718e-1 (s)

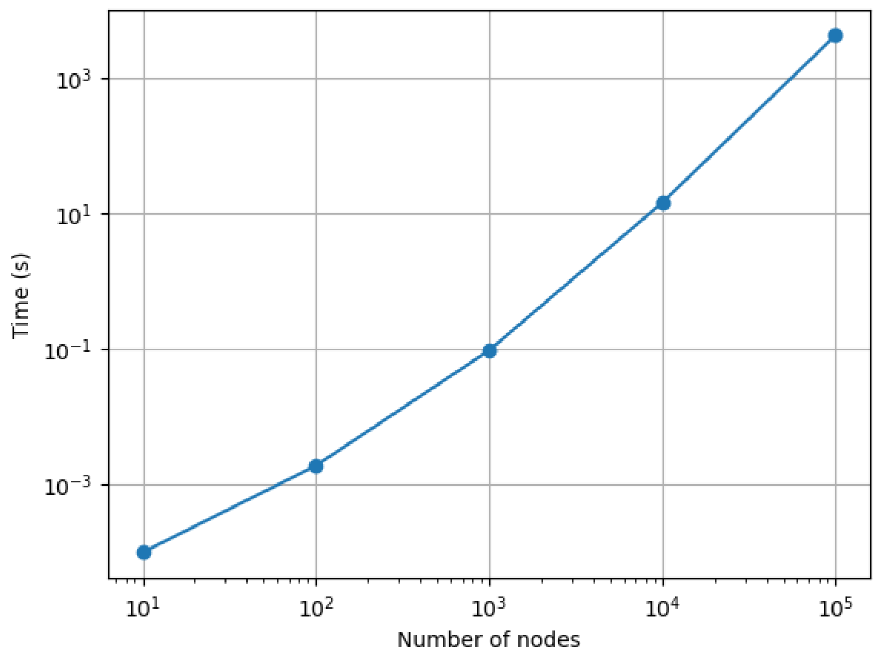

Figure 9: Buchi

Buchi times:

- **Arena 10 nodes** $\rightarrow$ 9.680e-05 (s)
- **Arena 100 nodes** $\rightarrow$ 1.853e-3 (s)
- **Arena 1000 nodes** $\rightarrow$ 9.426e-2 (s)
- **Arena 10000 nodes** $\rightarrow$ 14.502 (s)
- **Arena 100000 nodes** $\rightarrow$ 4328.881 (s)

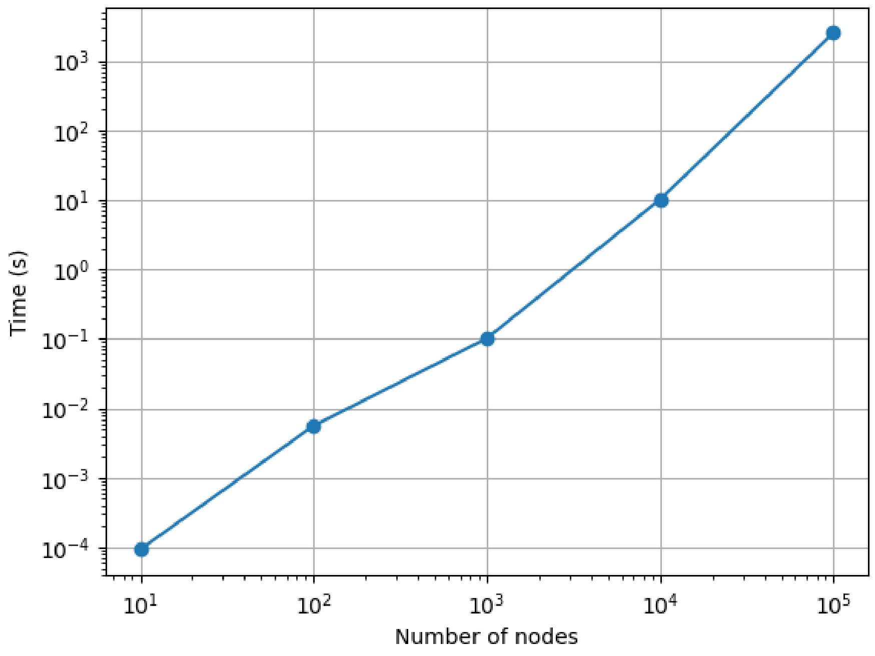

Figure 10: Co-Buchi

Co-buchi times:

- **Arena 10 nodes** $\rightarrow$ 9.274e-05 (s)
- **Arena 100 nodes** $\rightarrow$ 5.582e-3 (s)
- **Arena 1000 nodes** $\rightarrow$ 1.015e-1 (s)
- **Arena 10000 nodes** $\rightarrow$ 10.132 (s)
- **Arena 100000 nodes** $\rightarrow$ 2524.564 (s)

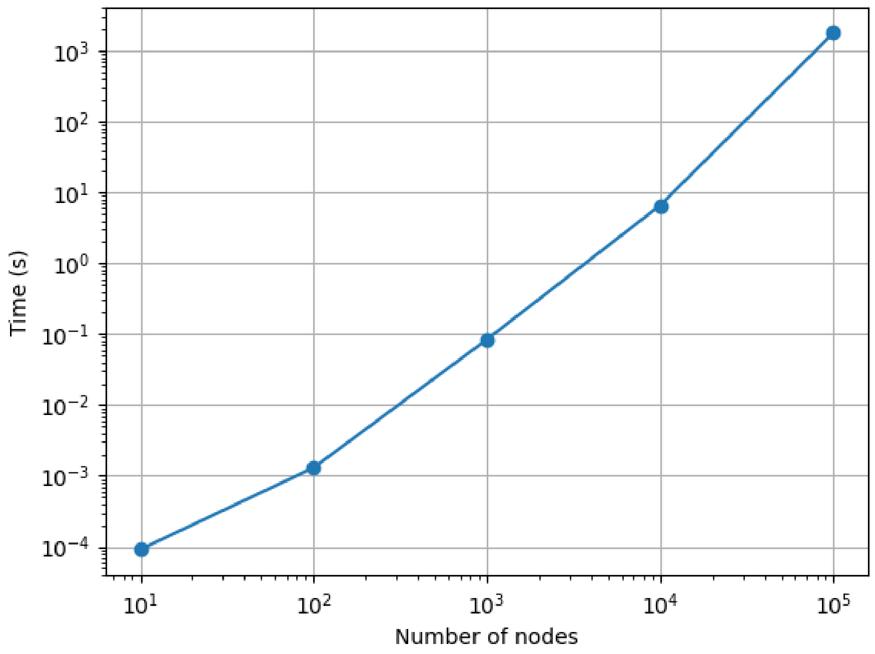

Figure 11: Parity

Parity times:

- **Arena 10 nodes** $\rightarrow$ 9.107e-05 (s)
- **Arena 100 nodes** $\rightarrow$ 1.295e-3 (s)
- **Arena 1000 nodes** $\rightarrow$ 8.397e-2 (s)
- **Arena 10000 nodes** $\rightarrow$ 6.589 (s)
- **Arena 100000 nodes** $\rightarrow$ 1796.324 (s)

For what concerns parity games, we performed a more complete and systematic analysis. We evaluated the performance of the algorithm over a set of games that are randomly generated by our function **generate_random_arena**, described before. In order to obtain unbiased results, we have taken 10 different game instances for each set of parameters and used the average time among them returned by the tool.

We tested games with priorities $p = 2, 3, 5, 10, 50, 100$, where for each of them we measured run-time performance for graphs of different sizes, ranging in 200, 400, 600, 800, 1000, 1200, 1400, 1800, 2000 nodes. The obtained results are reported in Fig.[12].

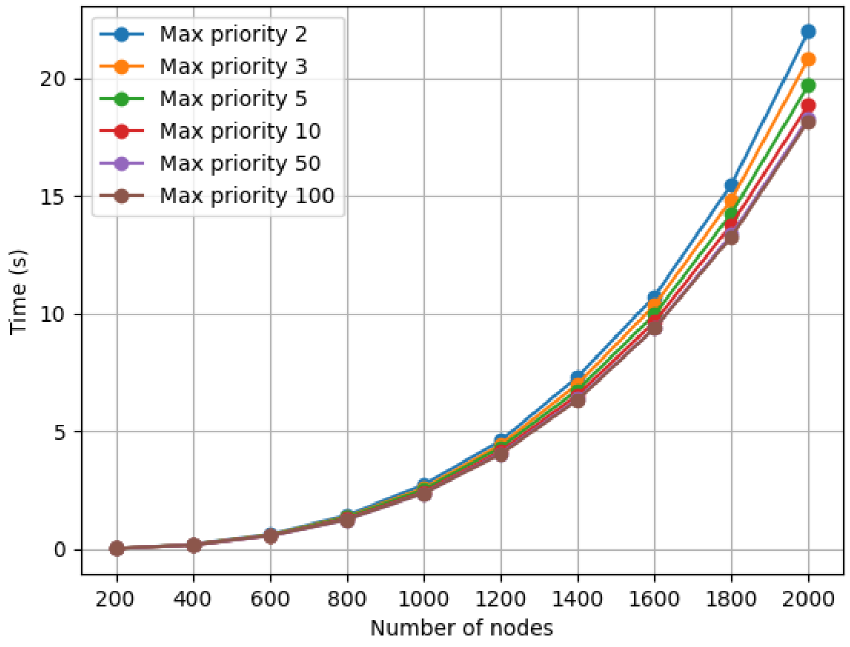

Figure 12: Parity performance on different priorities

The results are shown in Table [1], in which the number of states is reported in column 1, while the others contain the runtime executions, expressed in seconds, while the maximum priority is reported in the others.

n	**2 Pr**	**3 Pr**	**5 Pr**	**10 Pr**	**50 Pr**	**100 Pr**
200	0.0286	0.0290	0.0275	0.027	0.029	0.030
400	0.198	0.190	0.186	0.181	0.178	0.181
600	0.619	0.595	0.582	0.561	0.549	0.559
800	1.424	1.363	1.342	1.286	1.247	1.251
1000	2.731	2.591	2.526	2.418	2.369	2.367
1200	4.595	4.425	4.308	4.154	4.061	4.027
1400	7.305	7.000	6.747	6.540	6.366	6.321
1600	10.729	10.356	9.961	9.645	9.381	9.374
1800	15.483	14.837	14.248	13.775	13.350	13.247
2000	21.974	20.807	19.696	18.860	18.299	18.170

Table 1: Parity runtime executions with fixed maximum priorities