GAMES ON GRAPHS

Artur De Luca
Ionut Motoi
Leonardo Saraceni
Renzo Cherubino

INTRODUCTION

In many real-world problems, systems must be able to run for long periods of time (ideally infinite), while operating in an adversarial environment.

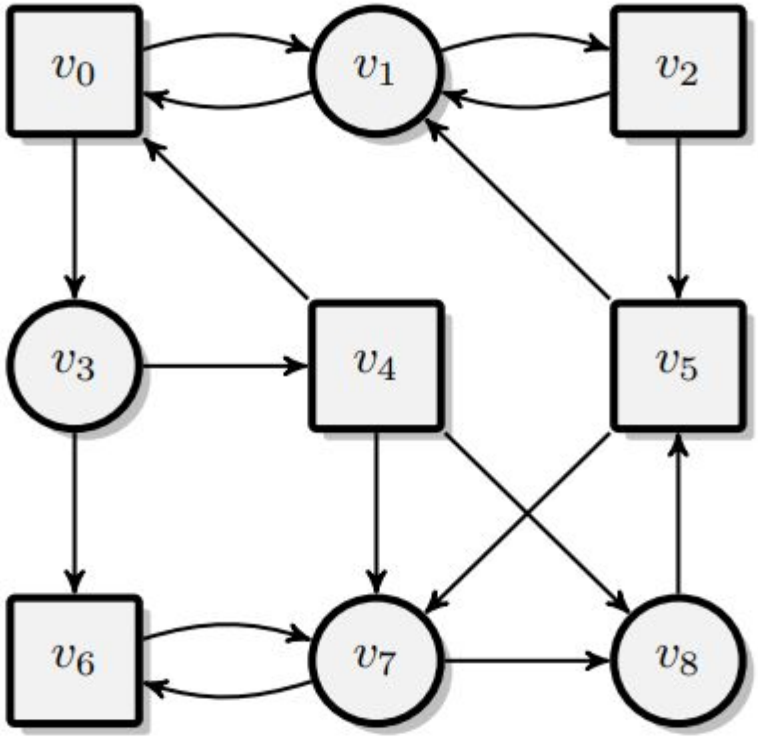
To properly study these settings and solutions, game-theoretic frameworks have been widely employed, modelling problems as a game between two agents, both trying to find a finite-state strategy that meets the winning conditions, if one exists.

This can be done by using graphs, where we can represent plays by moving a token through the graph according to the rules and the desired strategy, forming an infinite path.

INDEX

FOUNDATIONS

- Arena $\mathcal{A} = (V, V_0, V_1, E)$
- Vertices V: (v_0,v_1,v_2,v_3, v_4, v_5, v_6, v_7,v_8)
  - V_0: (v_1,v_3, v_7,v_8) *round vertices*
  - V_1: (v_0,v_2, v_4,v_5,v_6) *square vertices*
  - E: *all edges between elements in V*
- A play is an infinite sequence of valid transitions within vertices
- A strategy is a function mapping a given play to a successor
  - A positional strategy is a weaker representation of strategy that only depends on the current position
- These strategies can pursue a particular objective, depending on the type of game



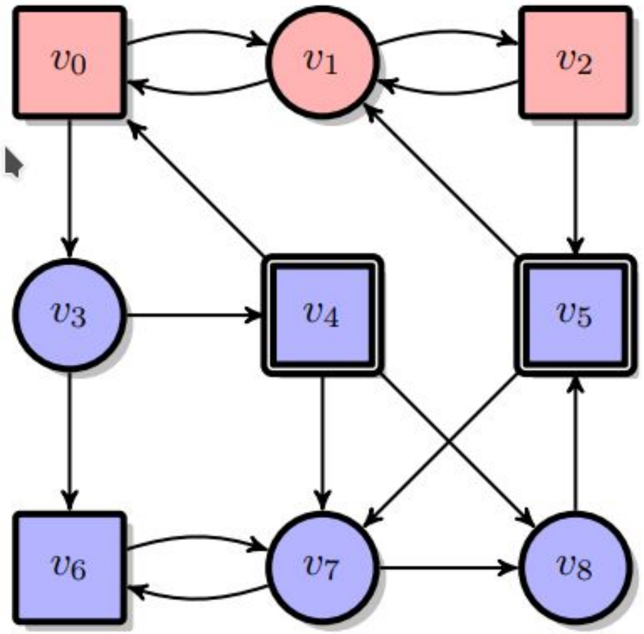
TYPES OF GAMES

REACHABILITY GAMES

In Reachability games, Player 0 has to visit one or more vertices from the set *R* at least once

We can find the solution iteratively by computing the region where Player 0 can attract the token to *R*

This region, called the 0-attractor of R, is obtained by expanding from the *R* vertices in a hierarchical manner



EXAMPLE OF ARENA USED FOR A REACHABILITY GAME

Reachability set: [V_4, V_5]

Winning region Player 0 → [V_3, V_4, V_5, V_6, V_7, V_8]

Winning region Player 1 → [V_0, V_1, V_2]

ALGORITHM

Step 1 — Identify the vertices in R to set up the initial attractor set

Step 2 — Calculate the Controlled Predecessor of the current attractor

Step 3 — Add the Controlled Predecessor to the current attractor

Step 4 — Repeat steps 2 and 3 until the attractor becomes stationary

ALGORITHM

Step 1 — Identify the vertices in R to set up the initial attractor set

Step 2 — Calculate the Controlled Predecessor of the current attractor

Step 3 — Add the Controlled Predecessor to the current attractor

Step 4 — Repeat steps 2 and 3 until the attractor becomes stationary

$$\begin{aligned}\mathrm{CPre}_0(R) = \{v \in V_0 | v' \in R \text{ for some sucesssor of } v\} \cup \\ \{v \in V_1 | v' \in R \text{ for all sucesssors of } v\}\end{aligned}$$

Step 1 — Identify the vertices in R to set up the initial attractor set

Step 2 — Calculate the Controlled Predecessor of the current attractor

Step 3 — Add the Controlled Predecessor to the current attractor

Step 4 — Repeat steps 2 and 3 until the attractor becomes stationary

ALGORITHM

Step 1 Identify the vertices in R to set up the initial attractor set

Step 2 Calculate the Controlled Predecessor of the current attractor

$$\mathrm{Attr}_0^{n+1}(R) = \mathrm{Attr}_0^{n}(R) \cup \mathrm{CPre}_0(\mathrm{Attr}_0^{n}(R))$$

Step 3 Add the Controlled Predecessor to the current attractor

Step 4 Repeat steps 2 and 3 until the attractor becomes stationary

ALGORITHM

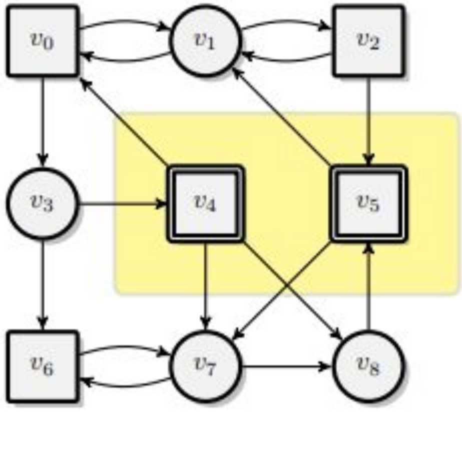
Step 1 — Identify the vertices in R to set up the initial attractor set

Step 2 — Calculate the Controlled Predecessor of the current attractor

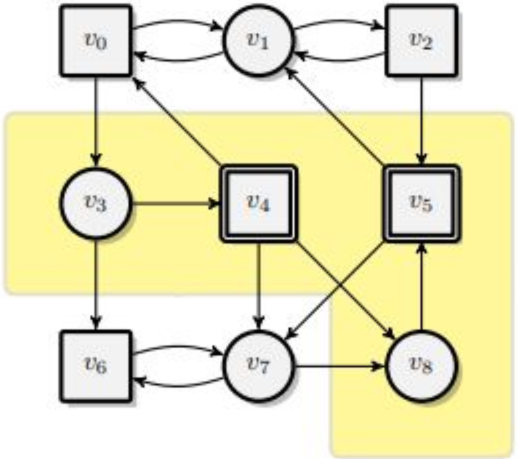
Step 3 — Add the Controlled Predecessor to the current attractor

Step 4 — Repeat steps 2 and 3 until the attractor becomes stationary

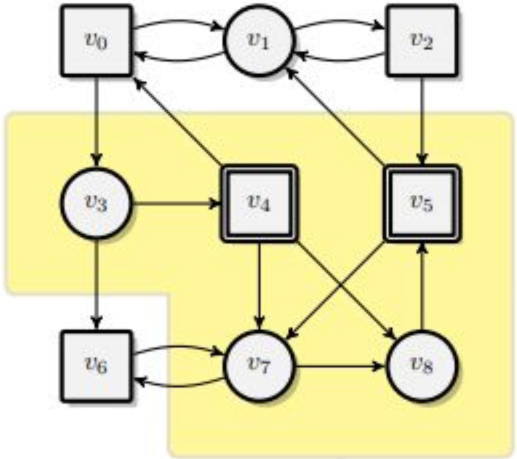
ALGORITHM



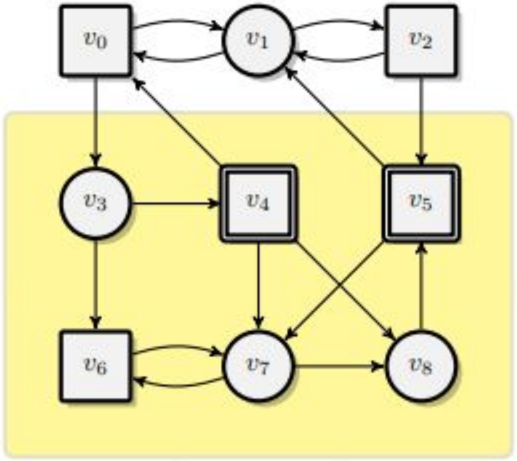
$\mathrm{Attr}_0^0(\{v_4, v_5\}) =$
$\{v_4, v_5\}$



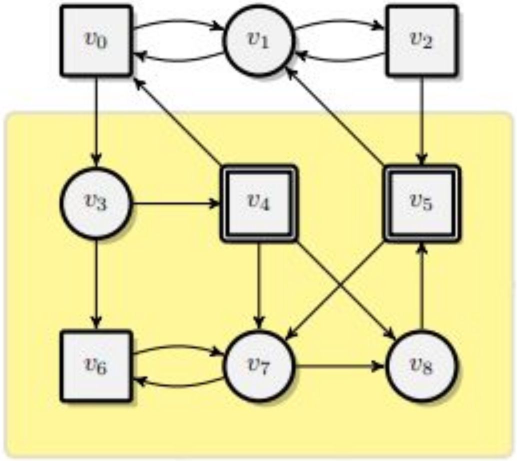
$\mathrm{Attr}_0^1(\{v_4, v_5\}) =$
$\{v_4, v_5\} \cup \{v_3, v_8\}$



$\mathrm{Attr}_0^2(\{v_4, v_5\}) =$
$\{v_3, v_4, v_5, v_8\} \cup \{v_3, v_7, v_8\}$



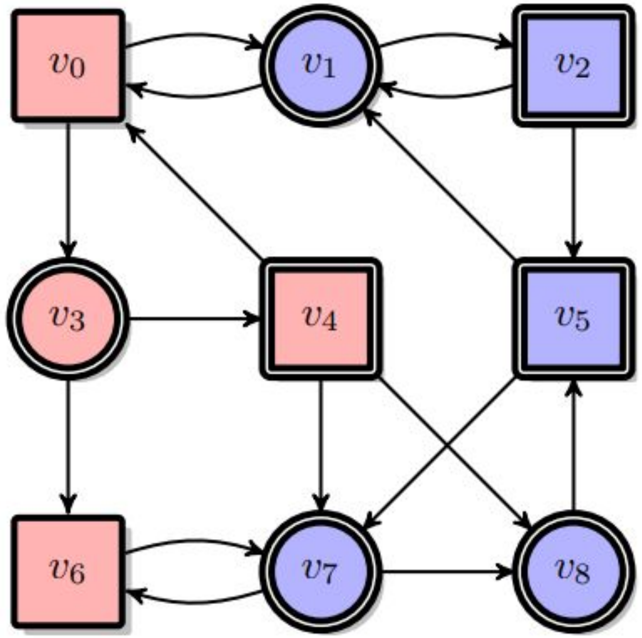
$\mathrm{Attr}_0^3(\{v_4, v_5\}) =$
$\{v_3, v_4, v_5, v_7, v_8\} \cup \{v_3, v_6, v_7, v_8\}$



$\mathrm{Attr}_0^4(\{v_4, v_5\}) =$
$\{v_3, v_4, v_5, v_6, v_7, v_8\}$

SAFETY GAMES

In Safety games, Player 0 has to obey a safety condition, there is, to remain in a set S of safe vertices



EXAMPLE OF ARENA USED FOR A SAFETY GAME

Safe set: [V_1, V_2, V_3, V_4, V_5, V_7, V_8]

Winning region Player 0 → [V_1, V_2, V_5, V_7, V_8]

Winning region Player 1 → [V_0, V_3, V_4, V_6]

DUALITY

Safety games are the dual of Reachability games.

In reachability - Player 1 must stay in region V\R

- Safety condition of Player 0

In safety - Player 1 must reach the set V\S

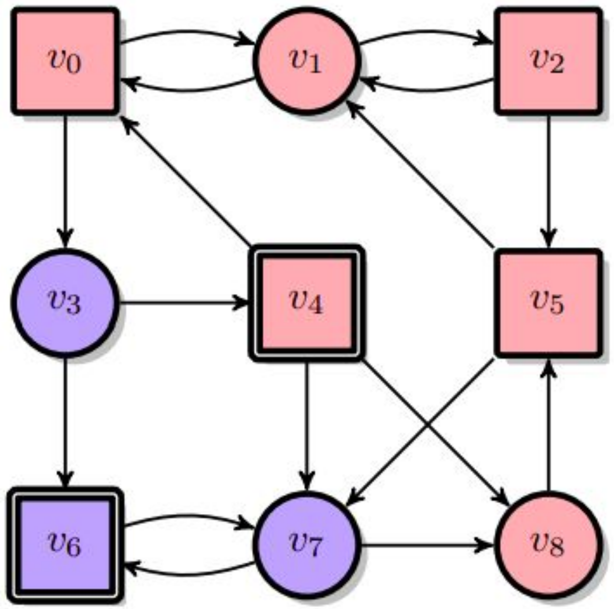
- Reachability condition of Player 0

Thus, we can obtain the winning strategies for safety by swapping the players' vertices and running the reachability algorithm

BÜCHI GAMES

In Büchi games, Player 0 has to visit a set of vertices F infinitely often → infinite plays

The solution can be found iteratively, by computing the increasing winning region of Player 1.



EXAMPLE OF ARENA USED FOR BÜCHI GAME

Recurrence set: [V_4, V_6]

Winning region Player 0 → [V_3, V_6, V_7]

Winning region Player 1 → [V_0, V_1, V_2, V_4, V_5,V_8]

Step 1

Find the vertices from which Player 1 can prevent Player 0 to reach F.

Step 2

Add those vertices to the winning region of Player 1 → W_1.

Step 3

Remove from the recurrence set F the vertices from which Player 1 can reach W_1.

Step 4

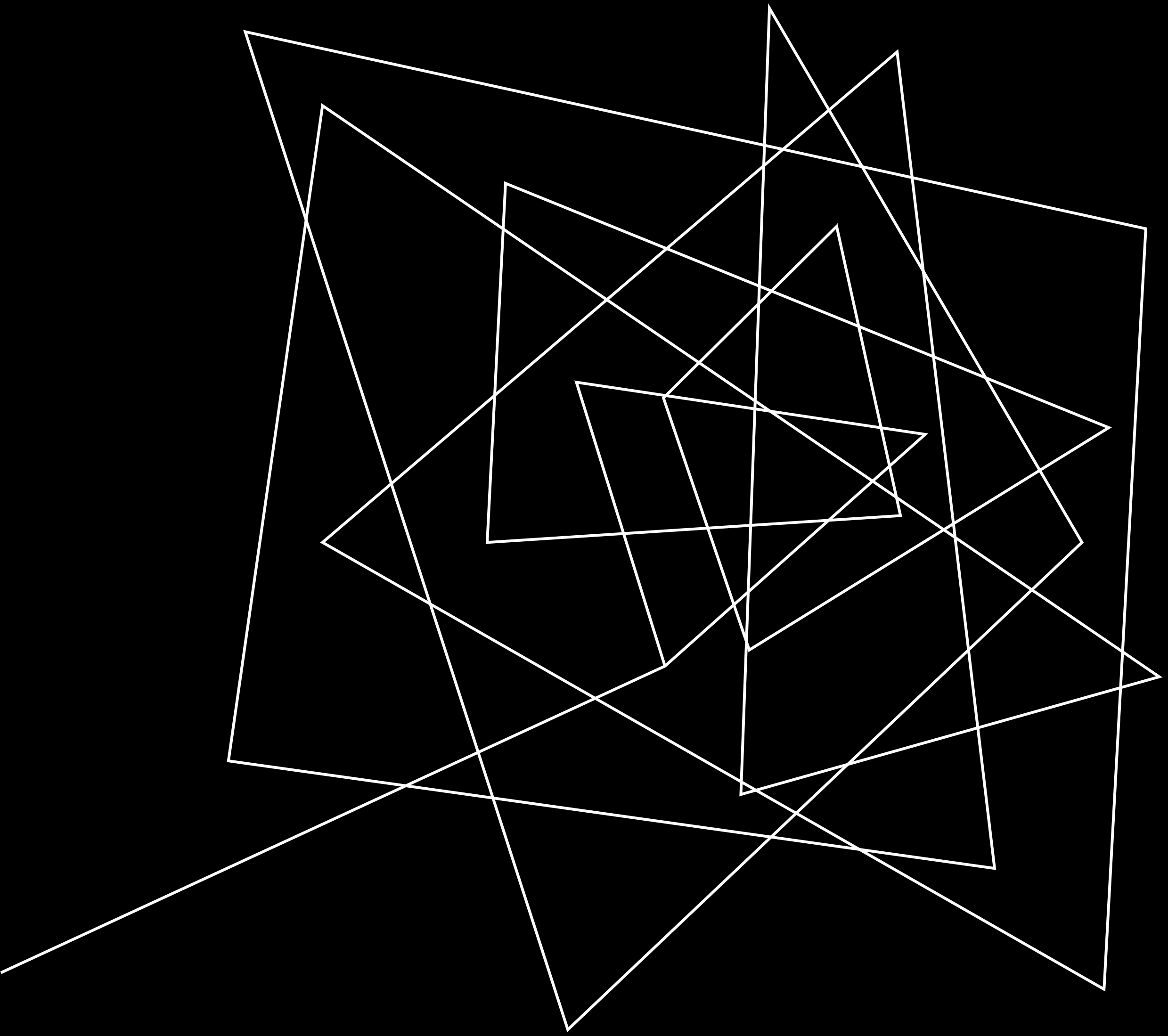
Repeat until F becomes stationary

ALGORITHM

CO-BÜCHI GAMES

In co-Büchi games, the goal of Player 0 is to visit a 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.



DUALITY

Co-Büchi games are the dual of Büchi games.

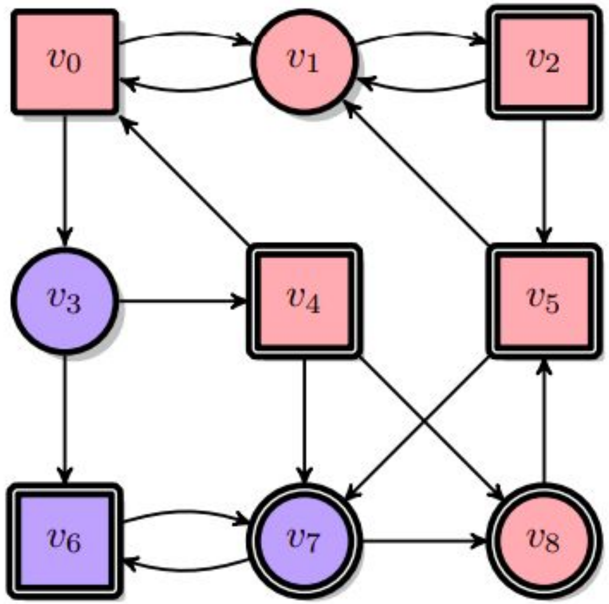
Büchi games - Player 1 must visit set V\F infinitely often

- Co-Büchi condition of Player 0

Co-Büchi games - Player 1 must visit the set V\C finitely often

- Buchi condition of Player 0

Thus, we can obtain the winning strategies for Co-Büchi games by swapping the players' vertices and running the Büchi algorithm



EXAMPLE OF ARENA USED FOR CO-BÜCHI GAME

Recurrence set: [V_2, V_4, V_5, V_8]

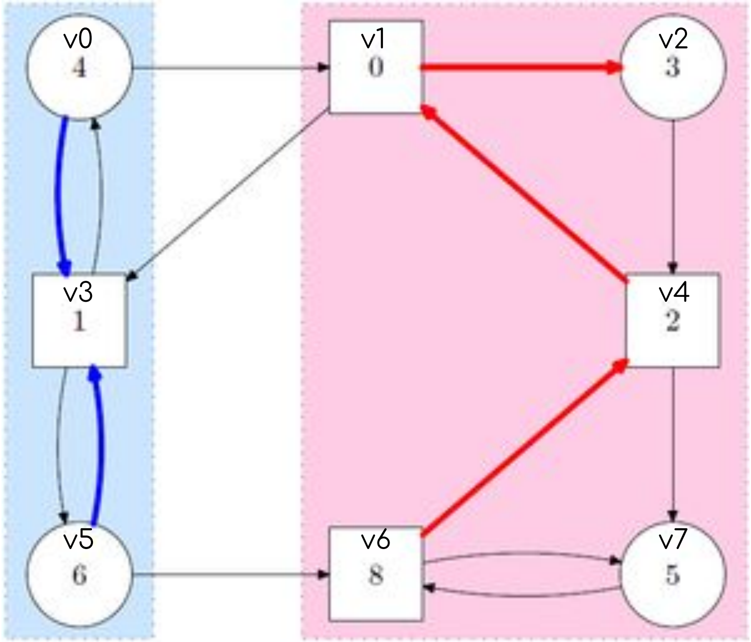
Winning region Player 0 → [V_3, V_6, V_7]

Winning region Player 1 → [V_0, V_1, V_2, V_4, V_5,V_8]

PARITY GAMES

In parity games, each vertex in the arena is "colored" with a natural number that represents its importance.

A generic Player i wins a play ρ if the maximum number seen infinitely often in the play ρ has parity i.



EXAMPLE OF ARENA USED FOR PARITY GAME

Winning region Player 0 → $[V_0, V_3, V_5]$

Winning region Player 1 → $[V_1, V_2, V_4, V_6, V_7]$

ZIELONKA'S RECURSIVE ALGORITHM

Step 1

Let:
- $\rho \rightarrow$ maximal priority
- $i = \rho \bmod 2 \rightarrow$ player associated with the priority
- $U = \{v \mid \Omega(v) = \rho\} \rightarrow$ set of nodes with priority ρ
- $A = Attr_i(U) \rightarrow$ corresponding attractor of player i

Step 2

Recursively solve $G' = G \setminus A \rightarrow$ obtain W'_i and W'_{1-i}

Step 3 case A

If W'_{1-i} is empty $\rightarrow W_{1-i}$ is empty and $W_i = V$

Step 3 Case B

Else compute the attractor $B = Attr_{1-i}(W'_{1-i})$

Recursively solve $G'' = G \setminus B \rightarrow$ obtain W'_i and W'_{1-i}

$W_i = W''_i$ and $W_{1-i} = W''_{1-i} \cup B$

Arena Generator

A python class is used to represent the arena.

Each object of this type, contains all the nodes, together with their successors, predecessors and importance.

Random Arena Generator

generate_random_arena(num_nodes, max_priority, max_successor, folder, pedix)

Takes as input the number of nodes **n,** the maximum number of successors assignable to each node **s**, the maximum number of importance a node can have **ρ**.

Generate a random arena exporting it to a txt file:

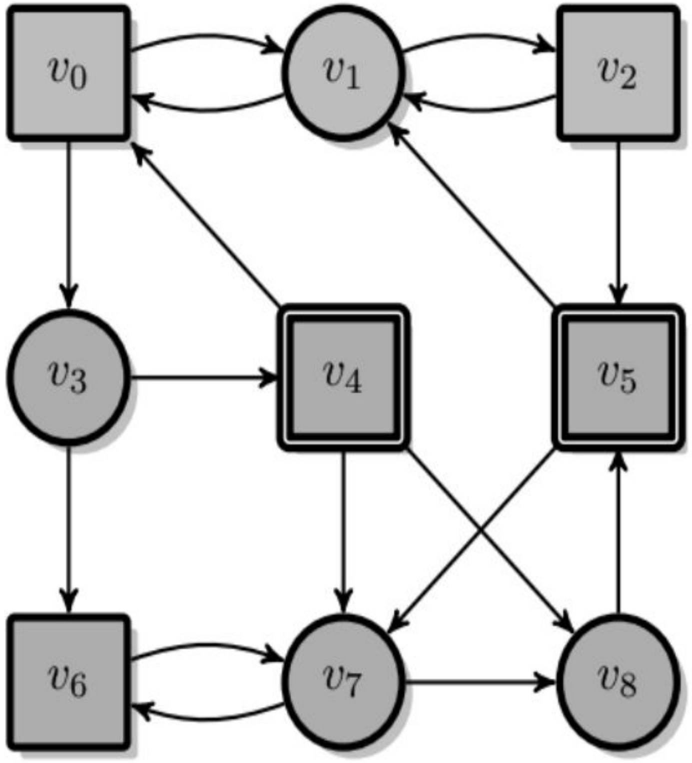
- nodeN player importance successor, successor2 ...
- nodeN+1 player importance successor, successor2 ...

5 arenas have been created with the following setups:

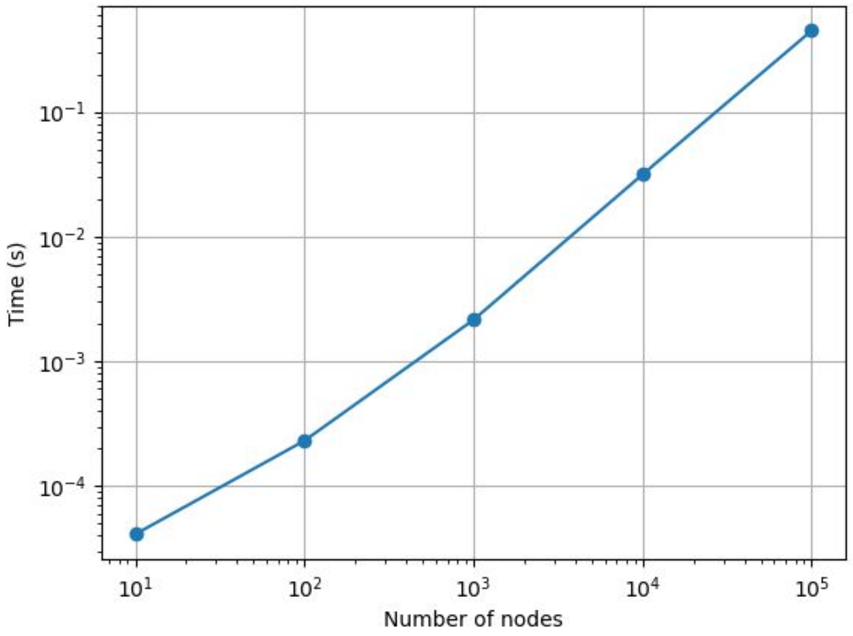
- **arena 1 → n** = 10, **ρ** = 10, **s** = 10
- **arena 2 → n** = 100, **ρ** = 100, **s** = 10
- **arena 3 → n** = 1000, **ρ** = 1000, **s** = 10
- **arena 4 → n** = 10000, **ρ** = 10000, **s** = 10
- **arena 5 → n** = 100000, **ρ** = 100000, **s** = 10

Arena Example

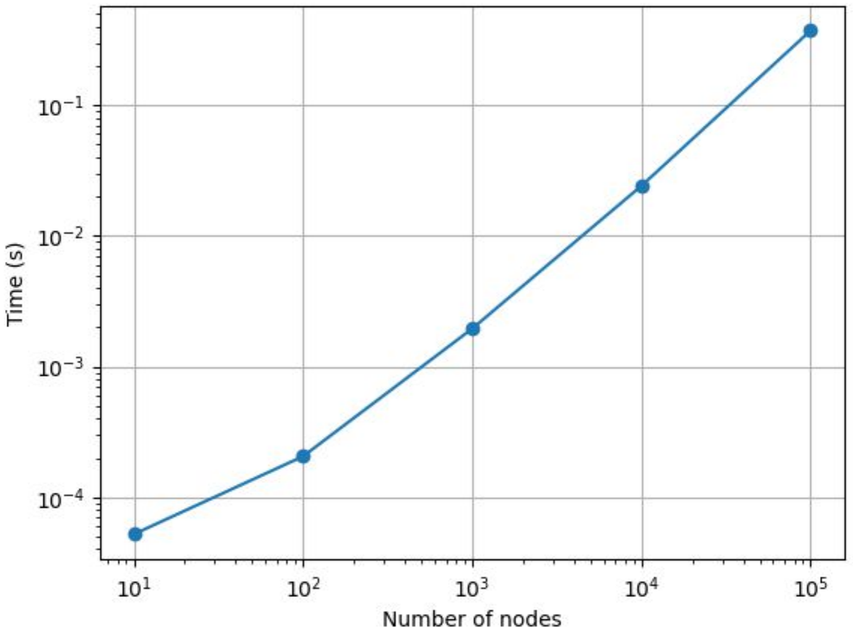
nodes	player	priority	successors
0	1	4	1,3
1	0	3	0,2
2	1	2	1,5
3	0	1	4,6
4	1	0	0,7,8
5	1	1	1,7
6	1	2	7
7	0	3	6,8
8	0	0	5



Scalability Results

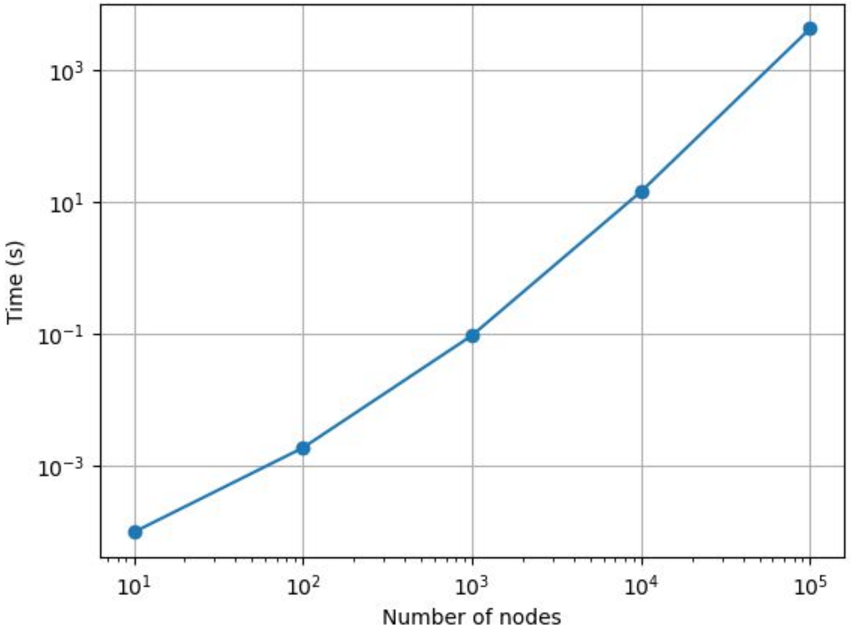


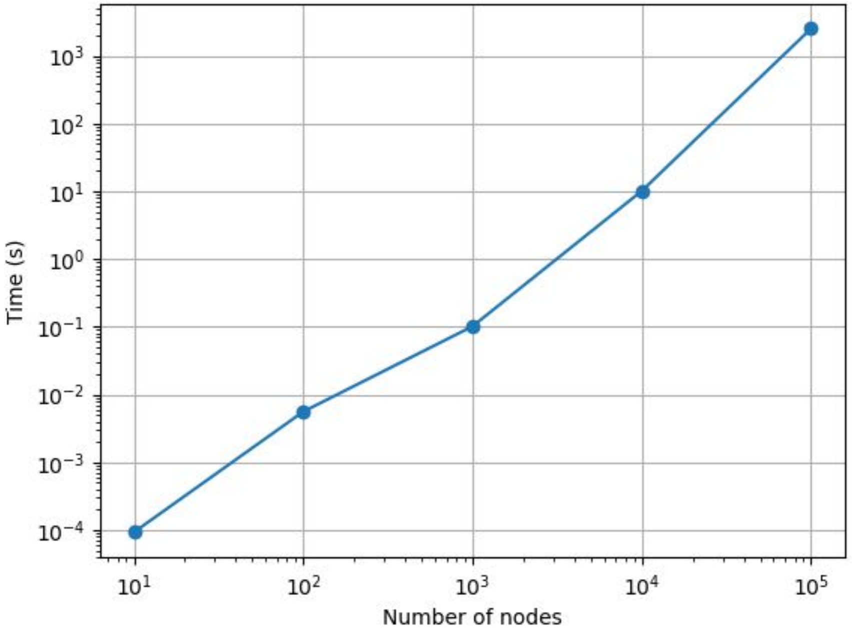
Reachability times:
- Arena 10 nodes → 4.101e-05 (s)
- Arena 100 nodes → 2.303e-4 (s)
- Arena 1000 nodes → 2.147e-3 (s)
- Arena 10000 nodes → 3.149e-2 (s)
- Arena 100000 nodes → 4.491e-1 (s)



Safety times:
- Arena 10 nodes → 5.221e-05 (s)
- Arena 100 nodes → 2.058e-4 (s)
- Arena 1000 nodes → 1.951e-3 (s)
- Arena 10000 nodes → 2.412e-2 (s)
- Arena 100000 nodes → 3.718e-1 (s)

Scalability Results





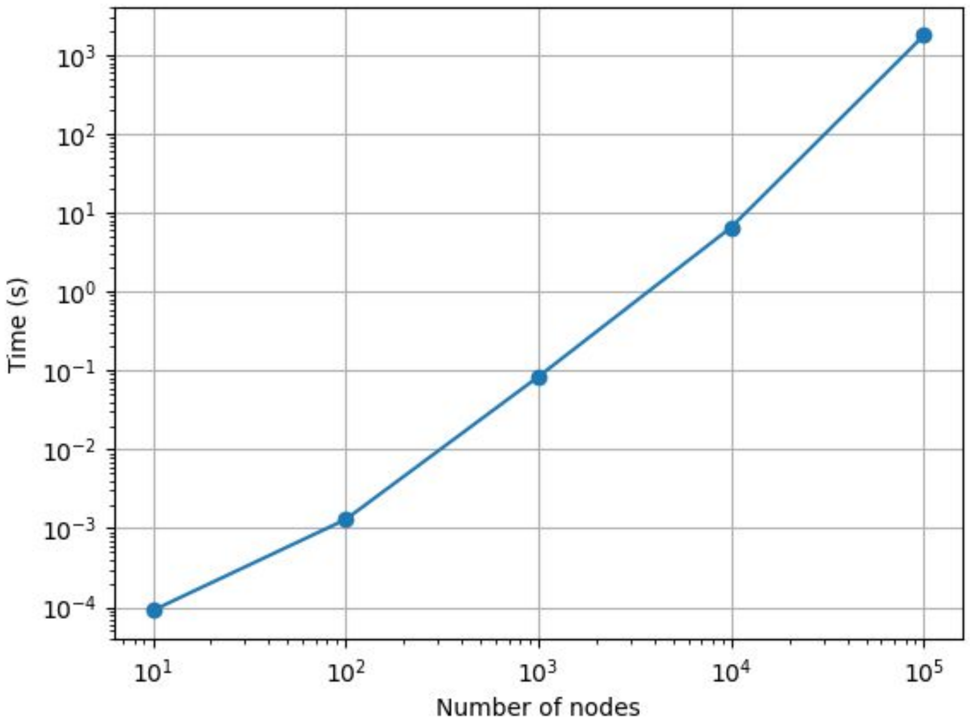
Buchi times:

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

Co-buchi times:

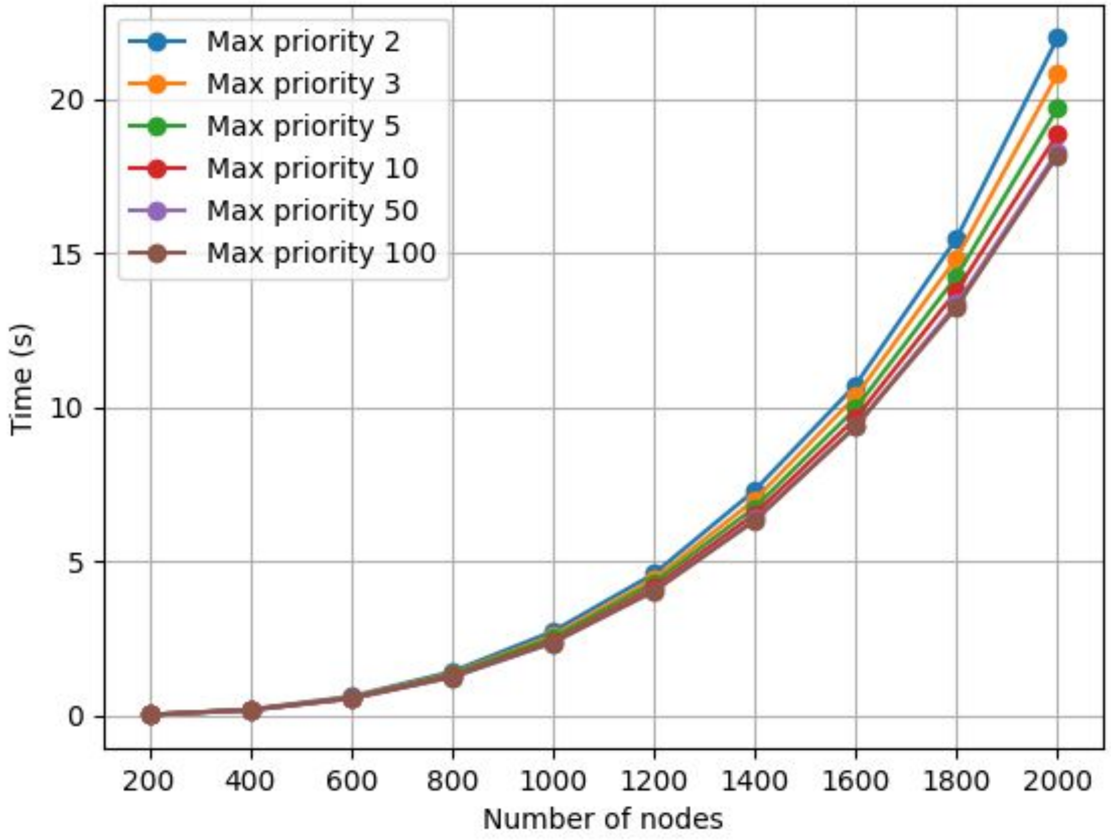
- Arena 10 nodes → 9.274e-05 (s)
- Arena 100 nodes → 5.582e-3 (s)
- Arena 1000 nodes → 1.015e-1 (s)
- Arena 10000 nodes → 10.132 (s)
- Arena 100000 nodes → 2524.564 (s)

Scalability Results



Parity times:
- Arena 10 nodes → 9.107e-05 (s)
- Arena 100 nodes → 1.295e-3 (s)
- Arena 1000 nodes → 8.397e-2 (s)
- Arena 10000 nodes → 6.589 (s)
- Arena 100000 nodes → 1796.324 (s)

Scalability Results

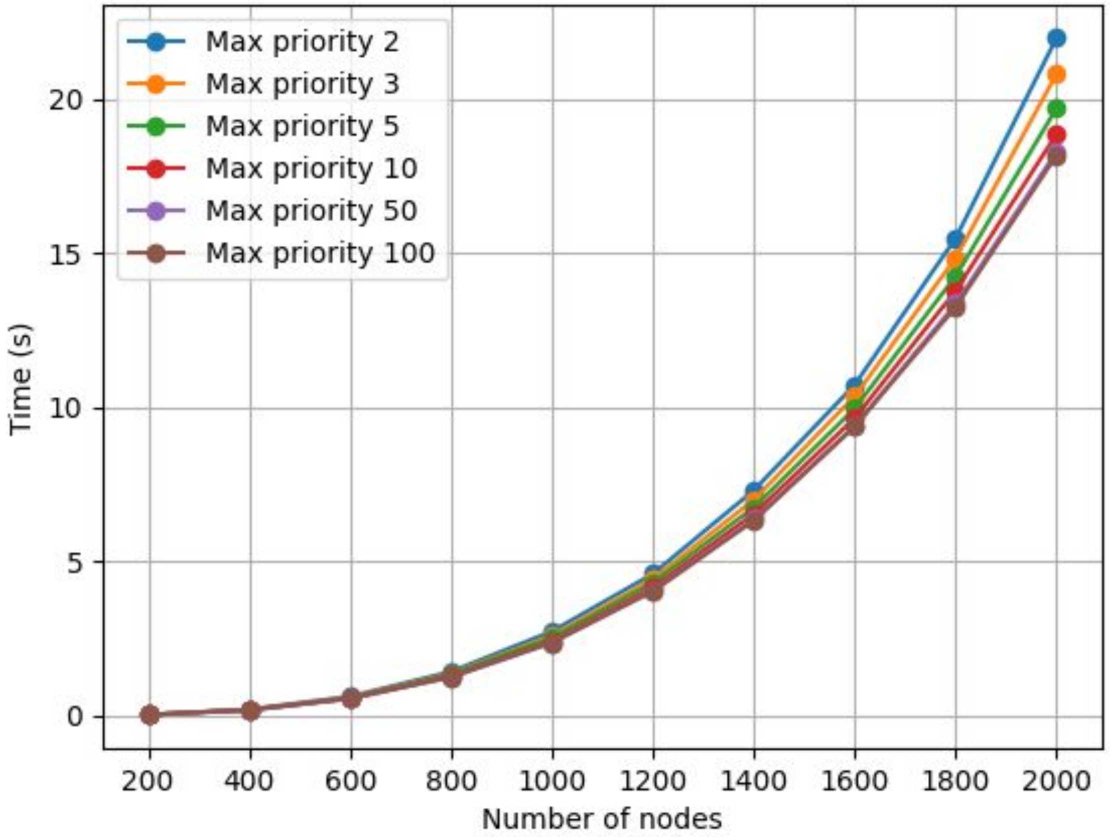


Evaluation of the algorithm's performance over a set of games randomly generated.

10 different game instances for each set of parameters.

The games are tested with priorities
ρ = 2, 3, 5, 10, 50 and 100, 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.

Scalability Results



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

THANK YOU FOR THE ATTENTION