Solving Parity Games Using An Automata-Based Algorithm

Antonio Di Stasio[1], Aniello Murano[1], Giuseppe Perelli[2], Moshe Vardi[3]

[1]University of Naples "Federico II"

[2]University of Oxford

[3]Rice University

CIAA, 2016

Outline

Outline

Importance of Parity Games

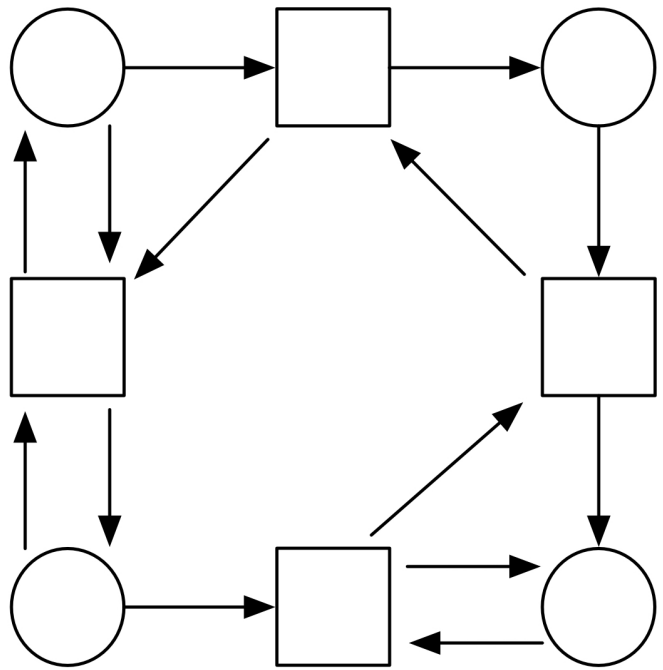
- A powerful formalism to synthesize and verify reactive systems.
- They express important system requirements (safety, liveness, etc.).
- Correspond to the model-checking problem for the modal mu-calculus.

However,

- Finding a winning strategy in Parity Games is in UP $\cap$ CoUP.
- Deciding whether a polynomial time solution exists is a long-standing open question.

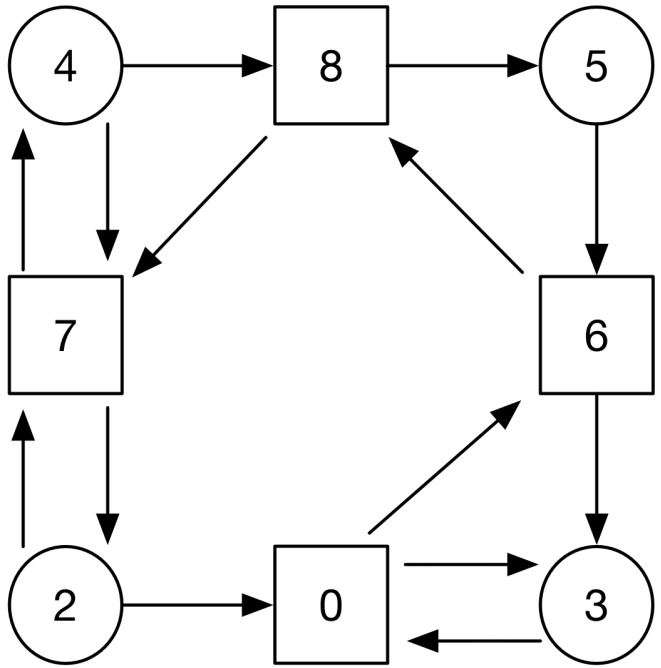
Parity Games

- Parity Games are infinite two-player:
  1. Player 0
  2. Player 1
- They are played on directed graphs.



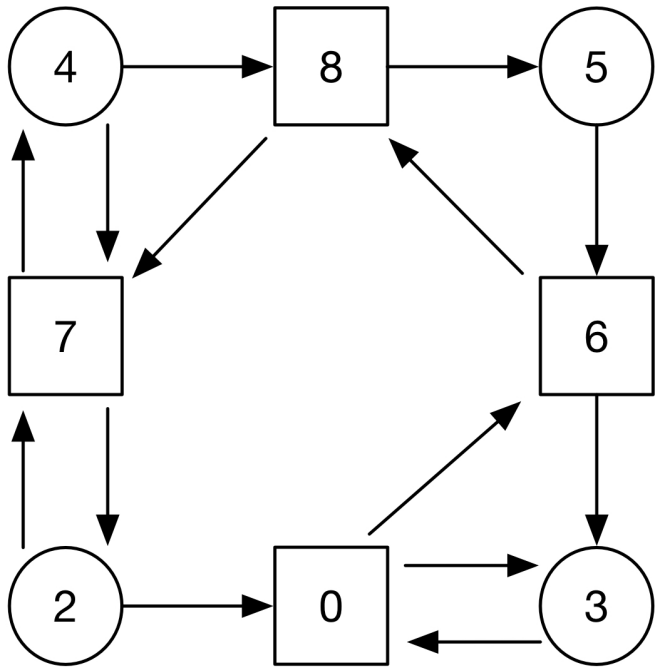
Parity Games

- Parity Games are infinite two-player:
  1. Player 0
  2. Player 1
- They are played on directed graphs.
- Each node is colored by a priority, i.e., a natural number.
- Players have perfect information about the adversary moves.



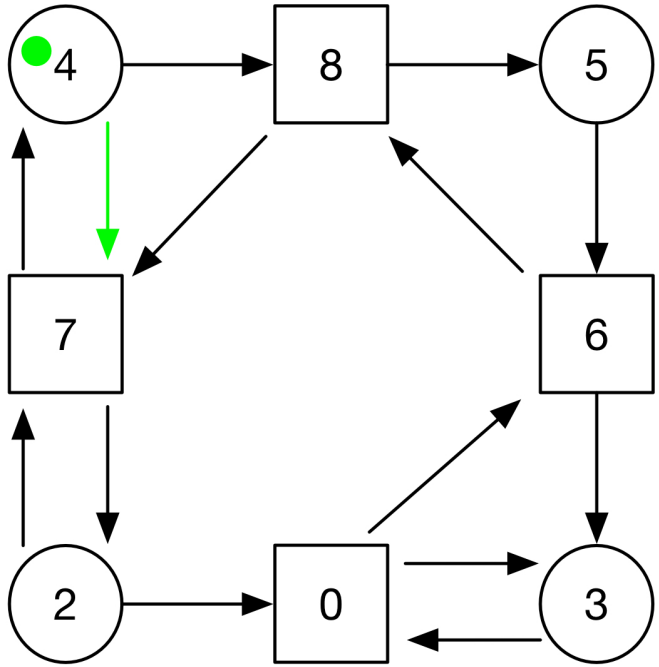
Parity Games

- The players move in turn a token along graph's edges
- The **play** induces an infinite path in which:
  - Player 0 wins the play if the smallest priority visited infinitely often is even;
  - otherwise, Player 1 wins the play.



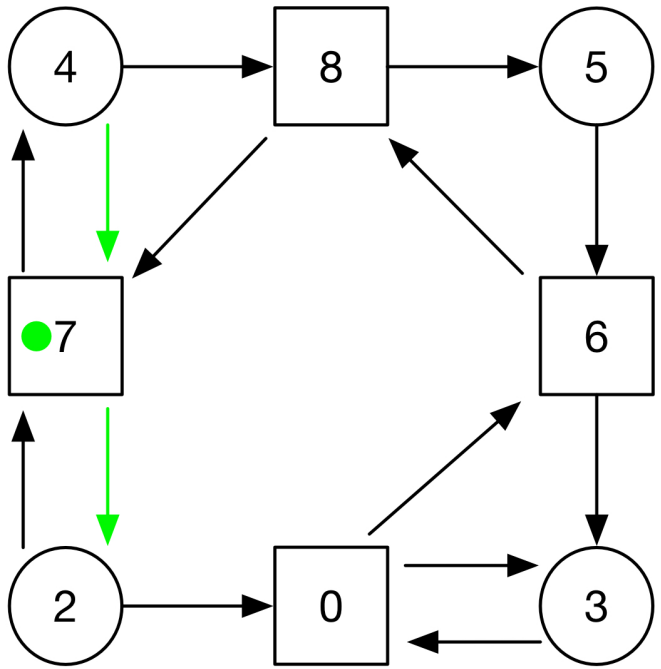
Parity Games

- The players move in turn a token along graph's edges
- The **play** induces an infinite path in which:
  - Player 0 wins the play if the smallest priority visited infinitely often is even;
  - otherwise, Player 1 wins the play.



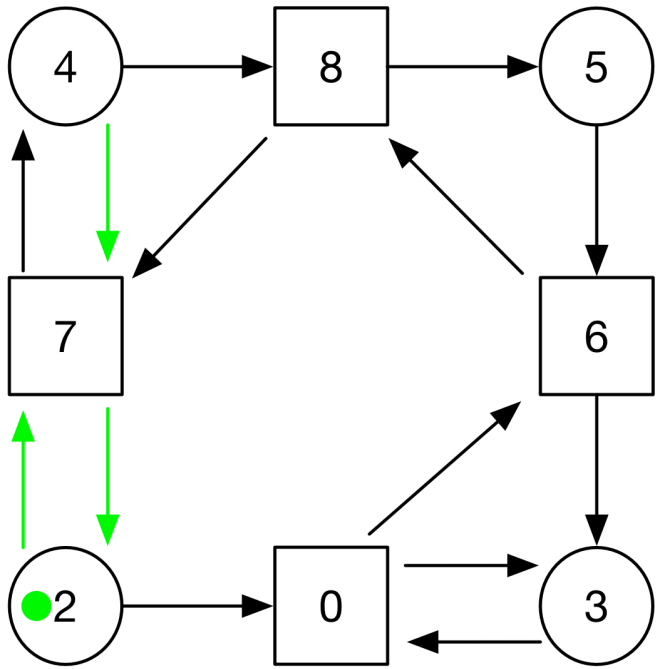
Parity Games

- The players move in turn a token along graph's edges
- The **play** induces an infinite path in which:
  - Player 0 wins the play if the smallest priority visited infinitely often is even;
  - otherwise, Player 1 wins the play.



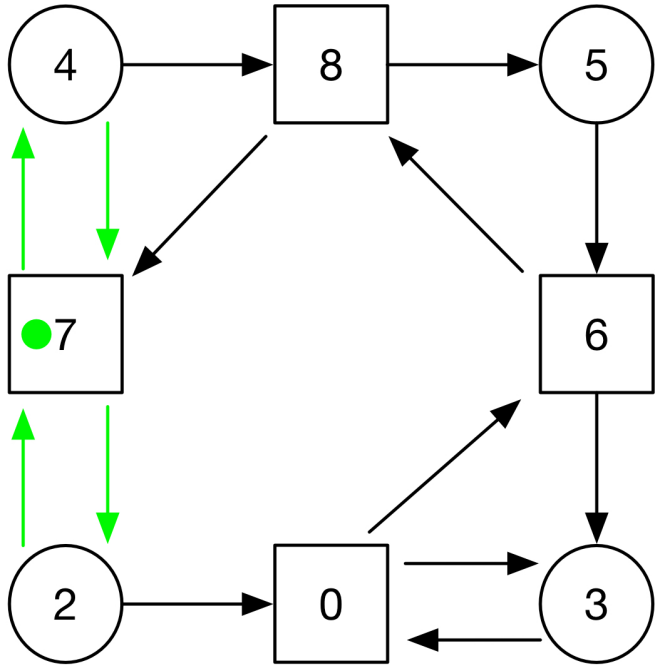
Parity Games

- The players move in turn a token along graph's edges
- The **play** induces an infinite path in which:
  - Player 0 wins the play if the smallest priority visited infinitely often is even;
  - otherwise, Player 1 wins the play.



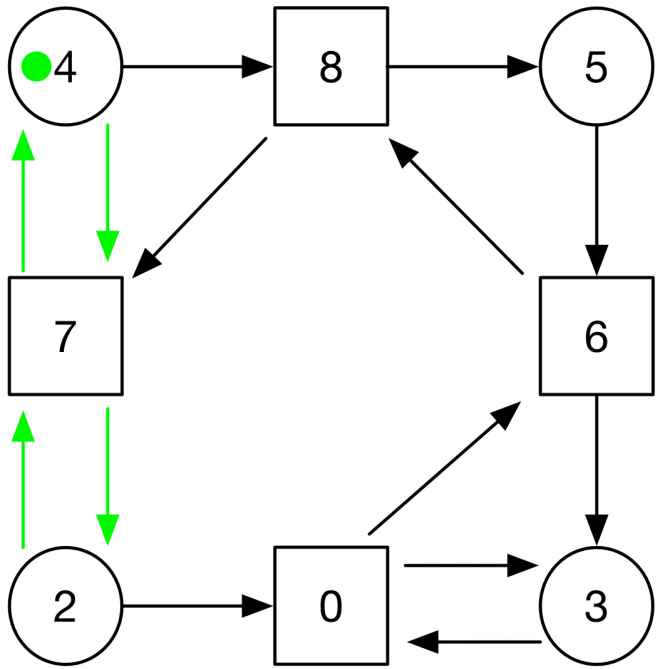
Parity Games

- The players move in turn a token along graph's edges
- The **play** induces an infinite path in which:
  - Player 0 wins the play if the smallest priority visited infinitely often is even;
  - otherwise, Player 1 wins the play.



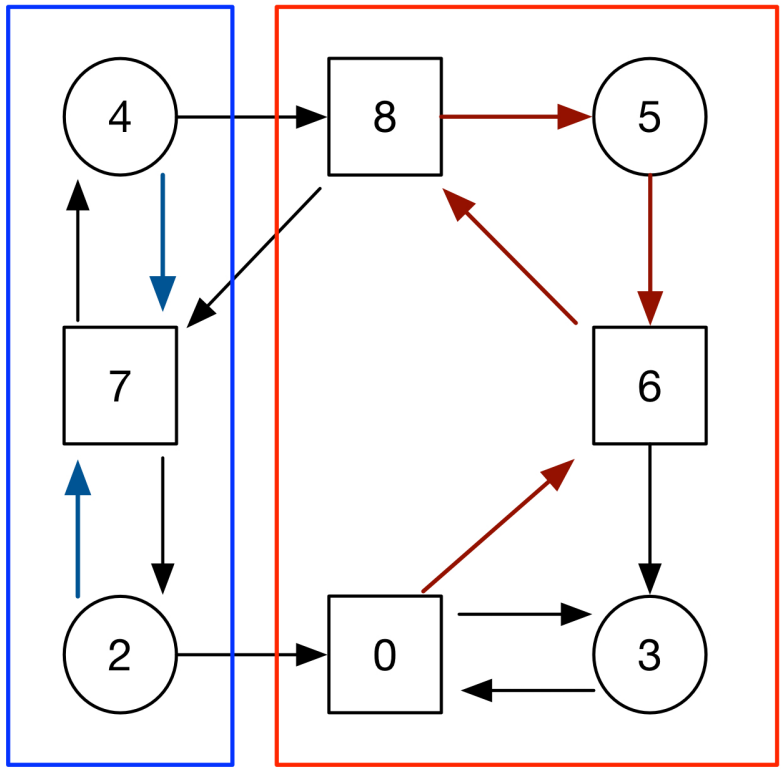
Parity Games

- The players move in turn a token along graph's edges
- The **play** induces an infinite path in which:
  - Player 0 wins the play if the smallest priority visited infinitely often is even;
  - otherwise, Player 1 wins the play.



Parity Games

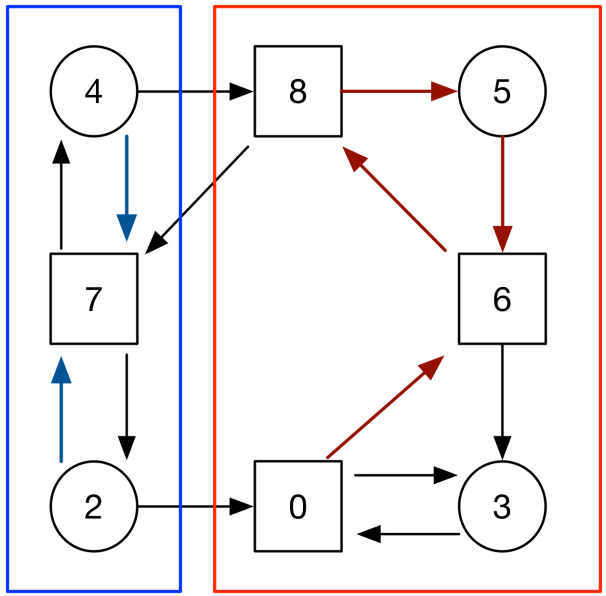
- The players move in turn a token along graph's edges
- The **play** induces an infinite path in which:
  - Player 0 wins the play if the smallest priority visited infinitely often is even;
  - otherwise, Player 1 wins the play.
- W_0 : Set of winning nodes for player 0.
- W_1 : Set of winning nodes for player 1.



Parity Games - Determinacy

A game is determined if for every node v either

- Player 0 can ensure winning from v or
- Player 1 can ensure winning from v.



Theorem ([Martin, 1975])

Parity games are determined

Algorithms for Parity Games

Algorithm	Computational Complexity [1]
Recursive (Rec)	$O(e \cdot n^d)$
Small Progress Measures (SPM)	$O(d \cdot e \cdot (\frac{n}{d})^{\frac{d}{2}})$
Strategy Improvement (SI)	$O(2^e \cdot n \cdot e)$
Dominion Decomposition (DD)	$O(n^{\sqrt{n}})$
Big Step (BS)	$O(e \cdot n^{\frac{1}{3}d})$

Table: Parity algorithms along with their computational complexities.

[1] n = nodes, e = number of edges, d = priorities

Our starting point and main results

- In *Weak Alternating Automata and Tree Automata Emptiness* [KV98] Orna Kupferman and Moshe Y. Vardi intruduced the `APT` algorithm.
- `APT` solves the parity games via emptiness checking of alternating parity automata.
- In [KV98] this algorithm has been just sketched but not spelled out in detail and without a correctness proof.
- Two major gaps that we fill in this work.

APT Algorithm

Idea...

The APT algorithm makes use of two special (incomparable) sets of nodes, denoted by V and A, and called set of *Visiting* and *Avoiding*.
Intuitively,

- nodes in V are those from which player 0 surely wins the game;
- nodes in A are those from which player 0 surely loses the game (and so need to be avoided).

The reasoning is symmetric for the player 1.

APT Algorithm

- Let $\alpha = \mathrm{F}_1 \cdot \ldots \cdot \mathrm{F}_k$ (parity condition), where $\mathrm{F}_i = \mathsf{p}^{-1}(i)$, i.e., the set of all nodes labeled with i.
- The algorithm is based on the function $\mathrm{Win}_0(\alpha, \mathrm{V}, \mathrm{A})$ that computes the set of nodes from which the player 0 has a strategy that:
  - avoids the nodes in A and;
  - either forces a visit in V or he wins the parity condition α.
- The definition is symmetric for the function $\mathrm{Win}_1(\alpha, \mathrm{A}, \mathrm{V})$.

APT Algorithm

- **Base Case**: If α is empty and the nodes of the game are partitioned in V and A, then

Dfn: $\mathrm{Win}_i(\alpha, \mathrm{V}, \mathrm{A})$

$\mathrm{Win}_i(\alpha, \mathrm{V}, \mathrm{A})$ is set of nodes from which the player *i* can force to move, in one step, to the set V.

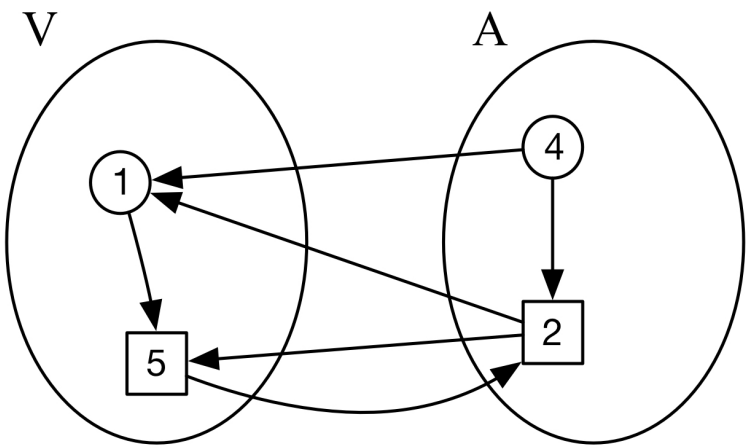
APT Algorithm

- **Base Case**: If α is empty and the nodes of the game are partitioned in V and A, then

Dfn: $\mathrm{Win}_i(\alpha, \mathrm{V}, \mathrm{A})$

$\mathrm{Win}_i(\alpha, \mathrm{V}, \mathrm{A})$ is set of nodes from which the player *i* can force to move, in one step, to the set V.

Example: Let $i = 1$:



$\mathrm{Win}_1(\alpha, \mathrm{V}, \mathrm{A}) = \{1, 2\}$

APT Algorithm

- **Inductive step**: $\alpha = \mathrm{F}_i \cdot \alpha'$, with $\alpha' = \mathrm{F}_{i+1} \cdot \ldots \cdot \mathrm{F}_k$, and $\beta \equiv_2 i$ a player.

Visiting and Avoiding

- The set V contains the nodes with priorities lower of i **good** for the player $1 - \beta$.
- The set A contains the nodes with priorities lower of i **bad** for the player $1 - \beta$.

APT Algorithm

- **Inductive step**: $\alpha = \mathrm{F}_i \cdot \alpha'$, with $\alpha' = \mathrm{F}_{i+1} \cdot \ldots \cdot \mathrm{F}_k$, and $\beta \equiv_2 i$ a player.

Visiting and Avoiding

- The set V contains the nodes with priorities lower of i **good** for the player $1 - \beta$.
- The set A contains the nodes with priorities lower of i **bad** for the player $1 - \beta$.

Dfn: $\mathrm{Win}_i(\alpha, \mathrm{V}, \mathrm{A})$

1. The set F_i (nodes with lower priority) is removed from α.

APT Algorithm

- **Inductive step**: $\alpha = \mathrm{F}_i \cdot \alpha'$, with $\alpha' = \mathrm{F}_{i+1} \cdot \ldots \cdot \mathrm{F}_k$, and $\beta \equiv_2 i$ a player.

Visiting and Avoiding

- The set V contains the nodes with priorities lower of i **good** for the player $1 - \beta$.
- The set A contains the nodes with priorities lower of i **bad** for the player $1 - \beta$.

Dfn: $\mathrm{Win}_i(\alpha, \mathrm{V}, \mathrm{A})$

1. The set F_i (nodes with lower priority) is removed from α.
2. F_i is declared to be avoiding.

APT Algorithm

- **Inductive step**: $\alpha = \mathrm{F}_i \cdot \alpha'$, with $\alpha' = \mathrm{F}_{i+1} \cdot \ldots \cdot \mathrm{F}_k$, and $\beta \equiv_2 i$ a player.

Visiting and Avoiding

- The set V contains the nodes with priorities lower of i **good** for the player $1 - \beta$.
- The set A contains the nodes with priorities lower of i **bad** for the player $1 - \beta$.

Dfn: $\mathrm{Win}_i(\alpha, \mathrm{V}, \mathrm{A})$

1. The set F_i (nodes with lower priority) is removed from α.
2. F_i is declared to be avoiding.
3. Using a fixpoint computation, the **only** nodes in F_i are revisited to be visiting.

Benchmarks

- The `APT` algorithm has been implemented in the well-know platform `PGSolver` developed in Ocaml by O. Friedmann and M. Lange.
- `PGSolver` is a collection of tools to solve, performe benchmark and generate parity games.
- We evaluated our implementation of the `APT` algorithm over several random game instances, comparing it with Zielonka (`RE`) and Small Progress Measure (`SP`) algorithms.

Benchmarks

- The random games instances were generated by `PGSolver`.
- Given a number n of nodes and the number k of priority as parameters, the generator works as follows:
  - For each node v, the priority $\mathrm{p}(v)$ is choosen uniformly between 0 and $k-1$.
  - For each node v, a number d from 1 to n is choosen uniformly and d distinct successor of v are randomly selected.

Benchmarks

	2 Pr			3 Pr			5 Pr		
n	RE	SP	APT	RE	SP	APT	RE	SP	APT
2000	4.94	5.05	0.10	4.85	5.20	0.15	4.47	4.75	0.42
4000	31.91	32.92	0.17	31.63	31.74	0.22	31.13	32.02	0.82
6000	107.06	108.67	0.29	100.61	102.87	0.35	100.81	101.04	1.39
8000	229.70	239.83	0.44	242.24	253.16	0.5	228.48	245.24	2.73
10000	429.24	443.42	0.61	482.27	501.20	0.85	449.26	464.36	3.61
12000	772.60	773.76	0.87	797.07	808.96	0.98	762.89	782.53	6.81
14000	1185.81	1242.56	1.09	1227.34	1245.39	1.15	1256.32	1292.80	10.02

Table: Runtime executions with fixed priorities 2, 3 and 5

Benchmarks

n	Pr	RE	SP	APT
$n = 2^k$				
1024	10	1.25	1.25	8.58
2048	11	7.90	8.21	71.08
4096	12	52.29	52.32	1505.75
8192	13	359.29	372.16	$\texttt{abort}_T$[2]
16384	14	2605.04	2609.29	$\texttt{abort}_T$
32768	15	$\texttt{abort}_T$	$\texttt{abort}_T$	$\texttt{abort}_T$
$n = 10^k$				
10	1	0	0	0
100	2	0	0	0
1000	3	1.3	1.3	0.04
10000	4	738.86	718.24	4.91
100000	5	$\texttt{abort}_M$[3]	$\texttt{abort}_M$	66.4

Table: Runtime executions with $n = 2^k$ and $n = 10^k$

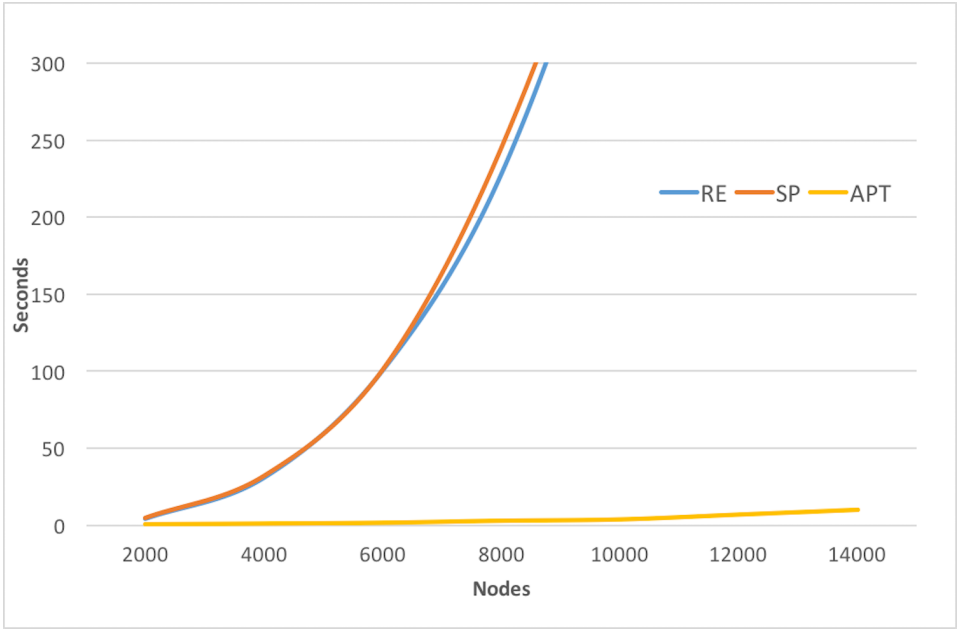


Figure: Runtime executions with $n = e^k$

[2] abortT: the program has been aborted due to timeout (greater than 3600 seconds).

[3] abortM: the program has been killed due to excessive memout (greater than 100GB).

Evaluation of the Results - 1

- Our results show that, under specific parameters, the APT algorithm performs better than others when the number of priorities is much smaller than the number of nodes in the graph.
- Moreover, the results of the exponential-scaling experiments show that the APT agorithm is the best performing one when $n = e^k$ and $n = 10^k$, but underperforms when $n = 2^k$.
- Then, APT has superior performance when the number of priorities is logarithmic in the number of game-graph nodes but with the base of logarithmic large enough.

Evaluation of the Results - 2

This results are an important development in many real applications of parity games, in particular:

- In the automata-theoretic approach to μ-calculus model checking, where the translation usually results in a parity automaton with few priorities, but with a huge number of nodes;
- In controller synthesis where we get low priorities (usually $\log_i|\text{states}|$ with $i \gg 2$) when we reduce the problem to computing winning strategy in a parity game.

Summary

We have:

- Used for a first time an automata-based algorithm to solve parity games: the `APT` algorithm.
- Implemented the `APT` algorithm in the platform `PGSolver`.
- By means of benchmarks, shown that `APT` is the best performing algorithm for solving parity games when:
  - the number of priorities is much smaller than the number of nodes in the graph;
  - the number of priorities is logarithmic in the number of the nodes but with the base of logarithmic large enough.

.

Conclusion and Future Work

- Study why the performance of `APT` algorithm is so sensitive to the relative number of priorities
- Analyze specialized optimization technique for the specific setting of a small number of priorities.
- Translate the `APT` algorithm to a symbolic setting.