Solving Parity Games in Scala

Antonio Di Stasio, Aniello Murano, Vincenzo Prignano and Loredana Sorrentino

Università degli Studi di Napoli “Federico II”

The 11th International Symposium on Formal Aspects of Component Software
10-12 September 2014, Bertinoro, Italy

Importance of Parity Games

- A powerful formalism to synthesize and verify reactive systems
- They express important system requirements (i.e., safety, liveness, ecc.)
- Correspond to the model-checking problem for the modal mu-calculus
- Finding a winning strategy in Parity Games is in UPTime ∩ CoUPTime
- Deciding whether a polynomial time solution exists is a long-standing open question

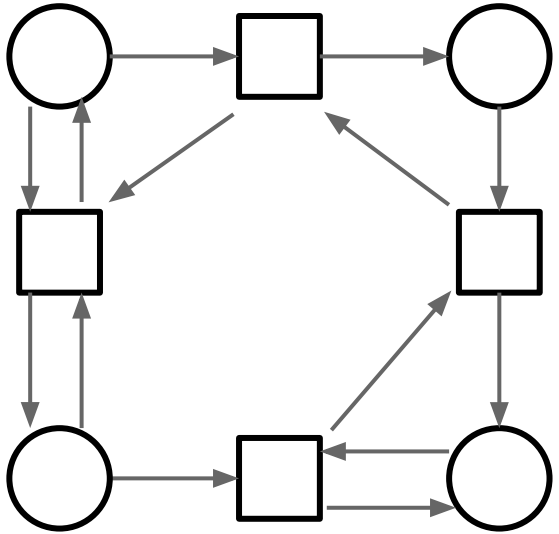
Parity Games

- *Parity Games* are infinite two-player games

  Player 0

  Player 1

- They are played on directed graphs



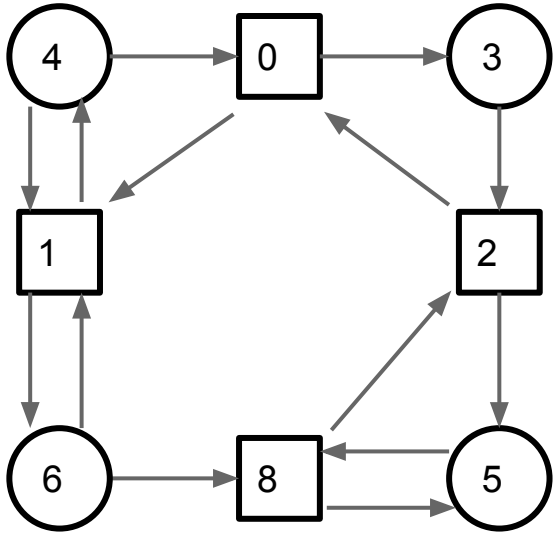
Parity Games

- *Parity Games* are infinite two-player games

  Player 0

  Player 1

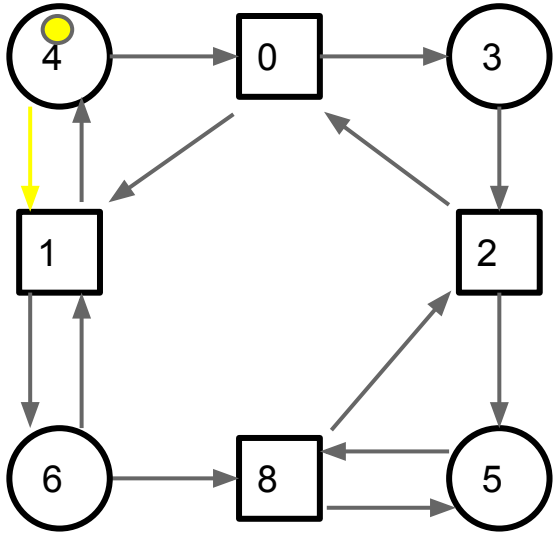
- They are played on directed graphs
- Each node is colored by a priority, i.e., a natural number
- Players have perfect information about the adversary moves



Parity Games

- The players move in turn a token along graph's edges

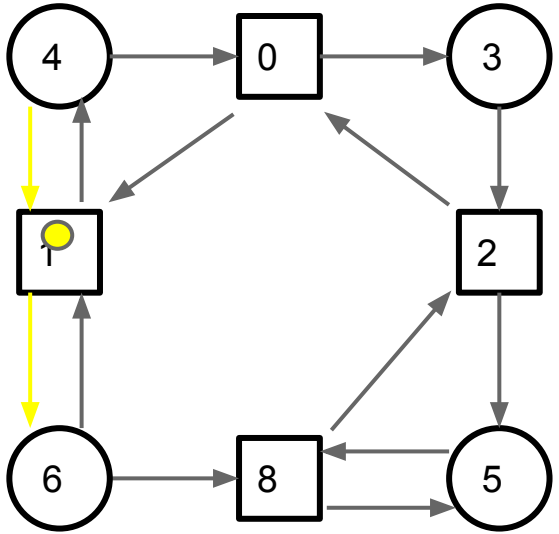
- The *play* induces an infinite path in which
  - Player 0 wins the play if the greatest priority visited infinitely often is even;
  - otherwise, Player 1 wins the play.



Parity Games

- The players move in turn a token along graph's edges

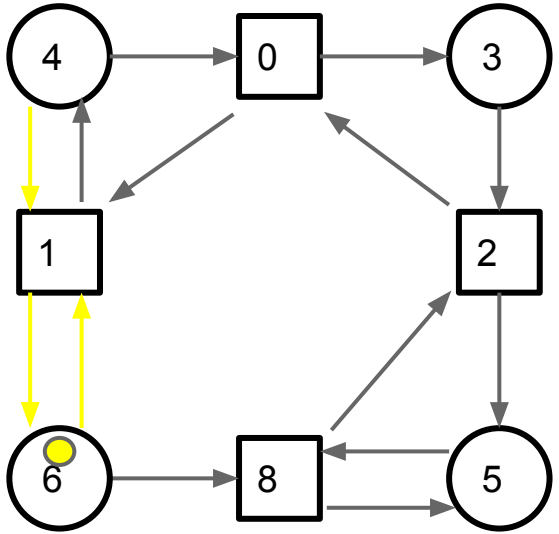
- The *play* induces an infinite path in which
  - Player 0 wins the play if the greatest priority visited infinitely often is even;
  - otherwise, Player 1 wins the play.



Parity Games

- The players move in turn a token along graph's edges

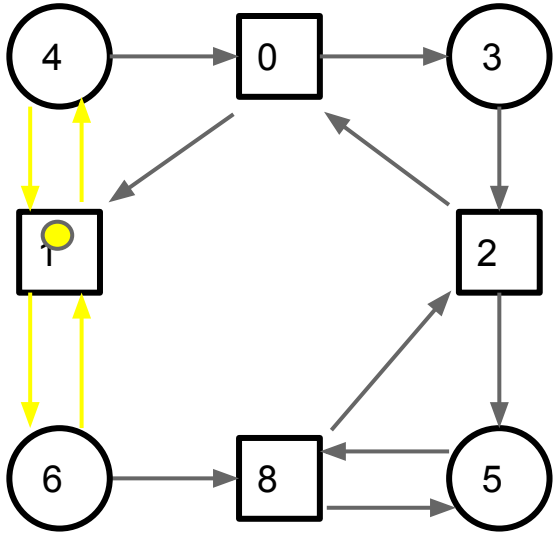
- The *play* induces an infinite path in which
  - Player 0 wins the play if the greatest priority visited infinitely often is even;
  - otherwise, Player 1 wins the play.



Parity Games

- The players move in turn a token along graph's edges

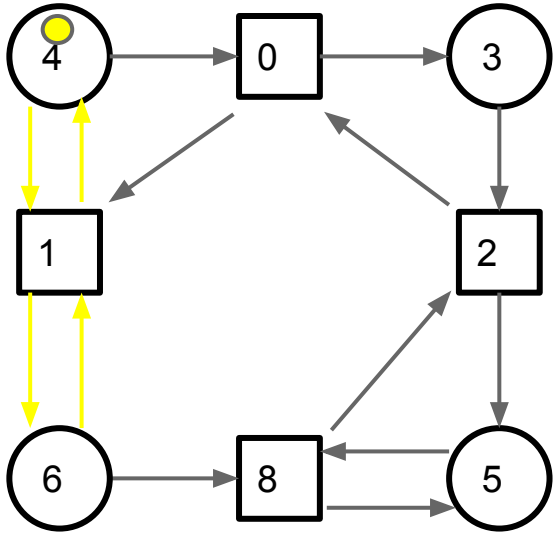
- The *play* induces an infinite path in which
  - Player 0 wins the play if the greatest priority visited infinitely often is even;
  - otherwise, Player 1 wins the play.



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 greatest priority visited infinitely often is even;
  - otherwise, Player 1 wins the play.

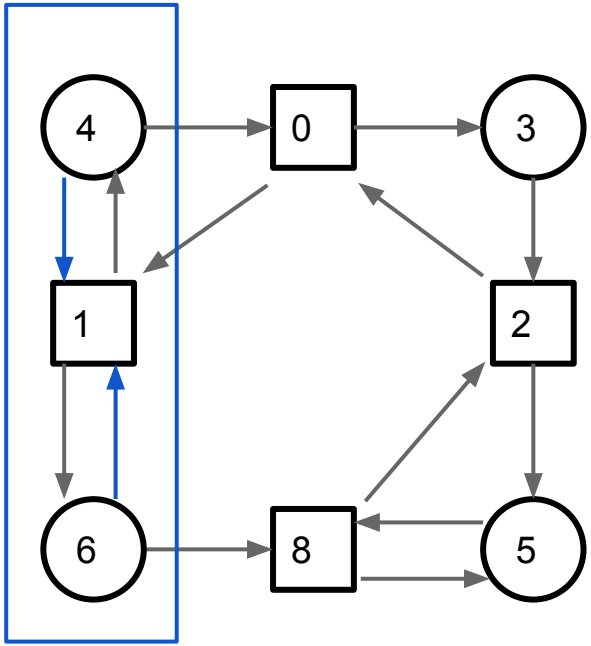


Parity Games

- 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 greatest priority visited infinitely often is even;
  - otherwise, Player 1 wins the play.

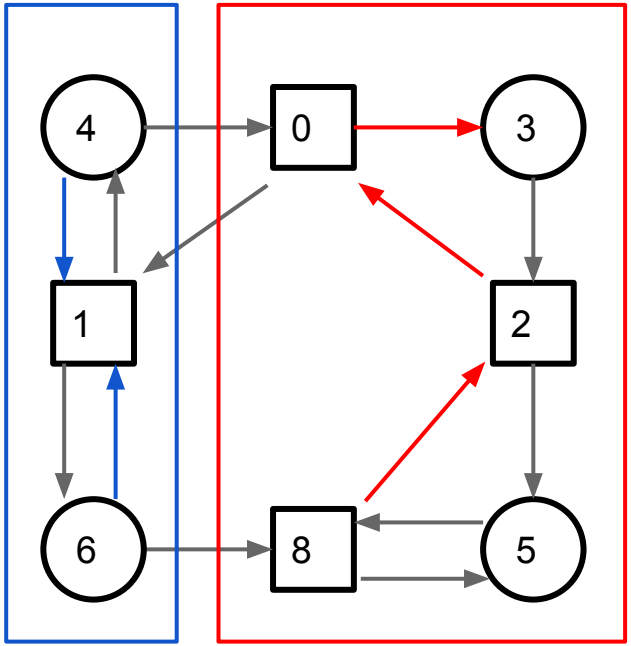
$W_0 = \{v \in V \mid \text{Player 0 wins from } v\}$



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 greatest priority visited infinitely often is even;
  - otherwise, Player 1 wins the play.



$W_0 = \{v \in V \mid \text{Player 0 wins from } v\}$

$W_1 = \{v \in V \mid \text{Player 1 wins from } v\}$

Algorithms for Solving Parity Games

Several algorithms have been proposed for solving parity games in the last two decades:

Condition	**Complexity***
Recursive [Zielonka-1998]	$O(e \cdot n^d)$
Small Progress Measures [Jurdzinski-2000]	$O(d \cdot e \cdot (\frac{n}{d})^{\frac{d}{2}})$
Strategy Improvement [Jurdzinski and J. Voge-2000]	$O(2^e \cdot n \cdot e)$
Dominion Decomposition [Jurdzinski,Paterson and Zwick-2006]	$O(n^{\sqrt{n}})$
Big Step [Schewe-2007]	$O(e \cdot n^{\frac{1}{3}d})$

*__e__ = number of edges, __n__ = nodes, __d__ = priority

Solving Parity Games in Pratice

- The algorithms have been implemented in **PGSolver**, written in OCaml, by Oliver Friedmann and Martin Lange

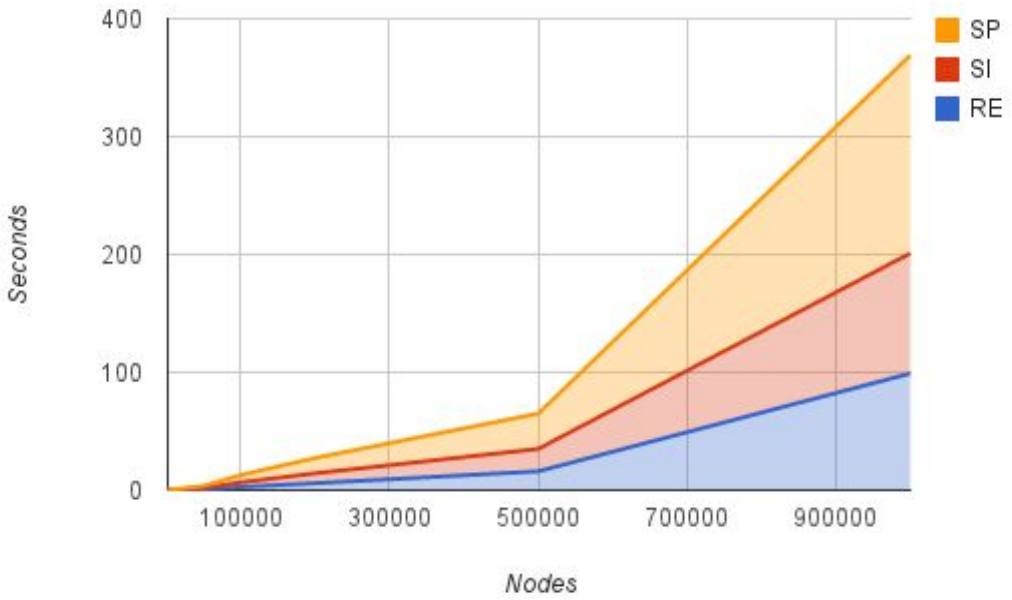
- PGSolver is a collection tools to solve, performe benchmark and generate parity games

- PGSolver has allowed to declare the Zielonka Recursive Algorithm as the **best** performing to solve parity games “in pratice”

Recursive as best algorithm in pratice

Nodes	RE	SI	SP
2000	0	0	0
5000	0,1	0,1	0,1
10000	0,1	0,2	0,4
20000	0,3	0,4	0,7
50000	0,8	1,1	1,7
100000	2,7	3,9	6,2
200000	5,9	8,4	13
500000	16	19	30
1000000	99	102	168

RE = Recursive
SI = Strategy Improvement
SP = Small Progress Measures



How can we improve the running time of the Recursive Algorithm?

Two research directions :

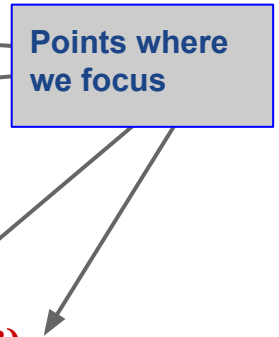
1. The algorithm in the way it is implemented in PGSolver
2. The chosen programming language

Recursive Algorithm

function win(G):
  if V == Ø :
    $(W_0, W_1) == (Ø, Ø)$
  else
    d = maximal priority in G
    U = { v ∈ V | *priority*(v) = d}
    p = d % 2
    j = 1 - p
    A = **$Attr_p(U)$**
    (W_0', W_1') = win **(G \ A)**
    if $W_j' == Ø$
      $W_p = W_p \cup A$
      $W_j = Ø$
    else
      B = **$Attr_j(W_1^j)$**
      (W_0', W_1') = win **(G \ B)**
      $W_p = W_p'$
      $W_j = W_j' \cup B$
    return (W_0, W_1)

Recursive Algorithm

function win(G):
  if V == Ø :
    (W_0,W_1)== (Ø,Ø)
  else
    d = maximal priority in G
    U ={ v∈V | *priority*(v) = d}
    p = d % 2
    j = 1 - p
    A = **$Attr_p(U)$**
    (W_0',W_1') = win **(G \ A)**
    if W_j' == Ø
      $W_p = W_p \cup A$
      W_j = Ø
    else
      B = **$Attr_j(W_1^j)$**
      (W_0',W_1') = win **(G \ B)**
      $W_p = W_p'$
      $W_j = W_j' \cup B$
    return (W_0,W_1)



Recursive Algorithm

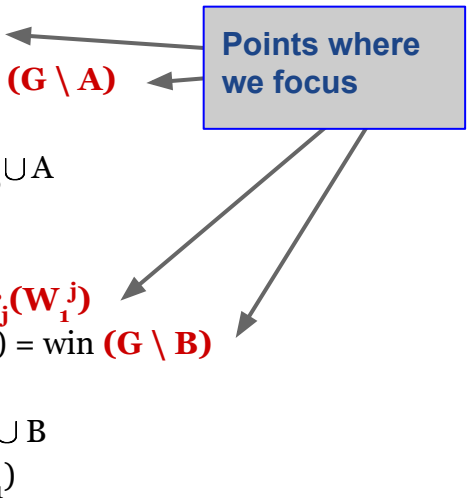
```
function win(G):
  if V == Ø :
     (W_0,W_1)== (Ø,Ø)
  else
       d = maximal priority in G
       U ={ v∈V | priority(v) = d}
       p = d % 2
       j = 1 - p
       A = Attr_p(U)
       (W_0',W_1') = win (G \ A)
       if W_j' == Ø
            W_p = W_p ∪ A
            W_j = Ø
       else
            B = Attr_j(W_1^j)
            (W_0',W_1') = win (G \ B)
            W_p = W_p'
            W_j = W_j' ∪ B
       return (W_0,W_1)
```

Points where we focus

Key issues in PGSolver:

Recursive Algorithm

```
function win(G):
  if V == Ø :
     (W_0,W_1)== (Ø,Ø)
  else
        d = maximal priority in G
        U ={ v∈V | priority(v) = d}
        p = d % 2
        j = 1 - p
        A = Attr_p(U)
        (W_0',W_1') = win (G \ A)
        if W_j' == Ø
              W_p = W_p ∪ A
              W_j = Ø
        else
              B = Attr_j(W_1^j)
              (W_0',W_1') = win (G \ B)
              W_p = W_p'
              W_j = W_j' ∪ B
        return (W_0,W_1)
```

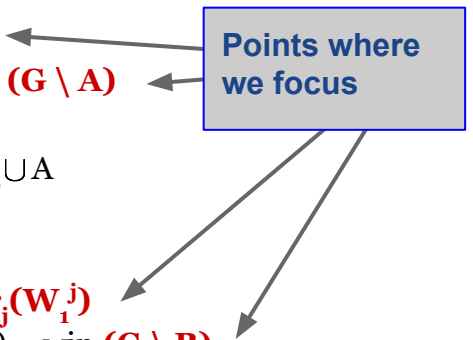


Key issues in PGSolver:

1. The program computes the difference between the graph and the attractor, returning **a new graph**

Recursive Algorithm

```
function win(G):
  if V == Ø :
     (W_0,W_1)== (Ø,Ø)
  else
        d = maximal priority in G
        U ={ v∈V | priority(v) = d}
        p = d % 2
        j = 1 - p
        A = Attr_p(U)
        (W_0',W_1') = win (G \ A)
        if W_j' == Ø
              W_p = W_p ∪ A
              W_j = Ø
        else
              B = Attr_j(W_1^j)
              (W_0',W_1') = win (G \ B)
              W_p = W_p'
              W_j = W_j' ∪ B
        return (W_0,W_1)
```



Key issues in PGSolver:

1. The program computes the difference between the graph and the attractor, returning **a new graph**

2. In **every** call, the attractor function builds the trasposed graph

Recursive Algorithm

```
function win(G):
  if V == Ø :
     (W_0,W_1)== (Ø,Ø)
  else
        d = maximal priority in G
        U ={ v∈V | priority(v) = d}
        p = d % 2
        j = 1 - p
        A = Attr_p(U)
        (W_0',W_1') = win (G \ A)
        if W_j' == Ø
              W_p = W_p ∪ A
              W_j = Ø
        else
              B = Attr_j(W_1^j)
              (W_0',W_1') = win (G \ B)
              W_p = W_p'
              W_j = W_j' ∪ B
        return (W_0,W_1)
```

Key issues in PGSolver:

1. The program computes the difference between the graph and the attractor, returning **a new graph**
2. In **every** call, the attractor function builds the trasposed graph
3. The attractor **calculates** the numbers of successors for the opponent player, in **every** iteration, possibly visiting **several times the same node**

Improved implementation of the Recursive Algoritm in PGSolver (1)

- The algorithm **does not directly modify the graph data structure**, but it uses a set, called *Removed*, to keep track of removed nodes
  - The check if a given node is excluded or not, is performed in **time costant**

- The transposed graph is calculated **only once**

Improved implementation of the Recursive Algoritm in PGSolver (2)

- Attractor function use an **Hashmap** to keep track of the number of successors of the opponent player's nodes and ensure:

  a. A costant time for check if a node was already visited

  b. That nodes present in several adjacency lists of nodes of the attractor, are visited **only once**

The chosen programming language: Scala

Scala is a programming language defines as:

1. A scalable language
2. A high-level language with a **concise and clear syntax**
3. A fusion of an object-oriented language and **a functionl one**
4. Runs on the Java Virtual Machine (JVM)
5. Supports every existing Java library

New Tool: SPGSolver

SPGSolver is a new tool, written in Scala, for solving parity games, and includes:

1. Classic Recursive Algorithm, based on implemantion developed in PGSolver

2. Improved Recursive Algorithm, based on improved implementation developed in PGSolver

Performance

We analyze and evaluate the running time of four implementations:

- Classic Recursive in OCaml (CRO)
- Improved Recursive in OCaml (IRO)
- Classic Recursive in Scala (CRS)
- Improved Recursive in Scala (IRS)

Performance - Benchmarks

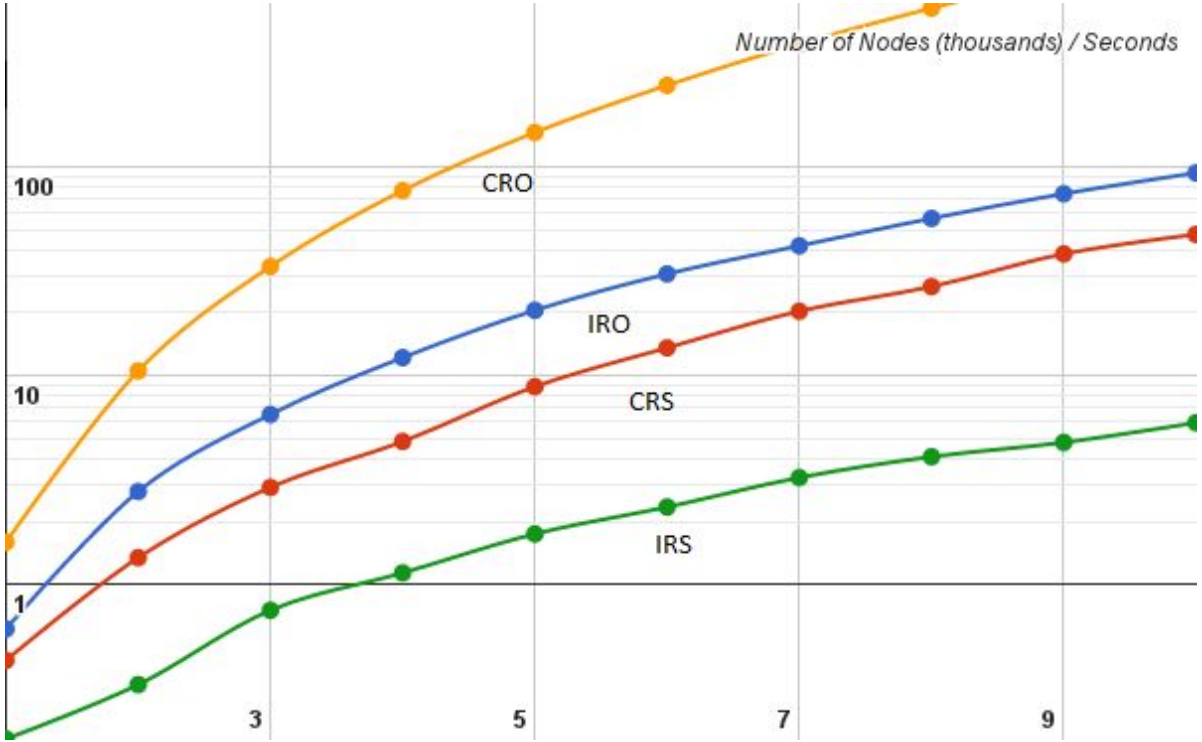
The experiments were performed on multiple instances of random parity games and under fixed parameters :

- Nodes
- Colors
- Minimum and maximum number of adjacent

In this experiment Colors=Nodes , Minumum = N/2, Maximum = N.

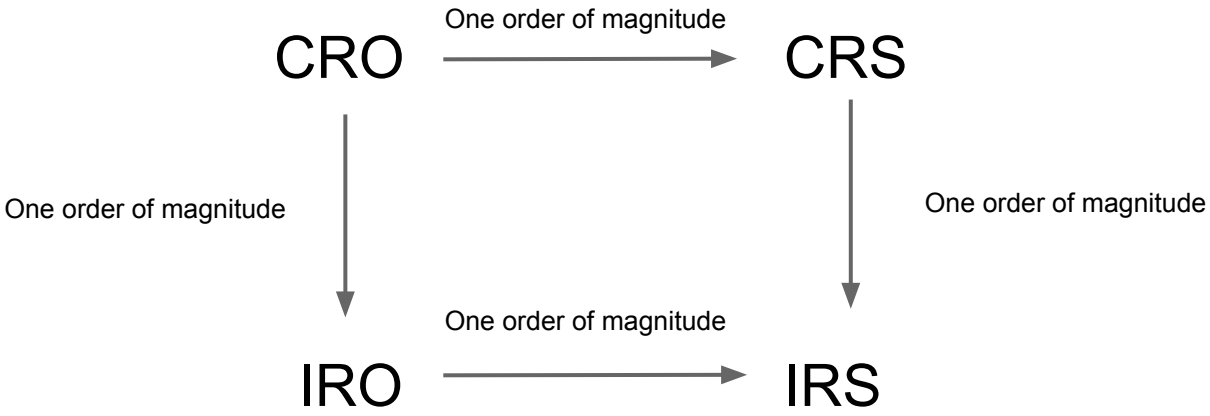
Nodes	IRS	CRS	IRO	CRO
1000	0.204	0.505	0.752	1.99
2000	0.456	1.918	3.664	13.208
3000	1.031	2.656	6.147	41.493
4000	1.879	6.728	15.966	96.847
5000	2.977	12.616	27.272	183.589
6000	3.993	19.032	41.051	306.104
7000	4.989	27.05	50.367	486.368
8000	6.103	36.597	70.972	718.002
9000	7.287	55.171	97.216	1026.404
10000	8.468	68.303	113.36	1422.94

Performance - Random Games Chart



Logarithmic Scaled

Performance - Results



Results show that IRS allows to gain **two orders of magnitude in time**

Conclusions

- PGsolver is the *de facto* tool to solve Parity Games
- We have revisited the implementation of the recursive algorithm in PGSolver
- We have worked on two directions to improve its performance: a smarter implementation and using Scala as programming language
- The combination of both allows to gain two orders of magntude in time

Future work

- There are two orthogonal open questions:

- Can we reproduce the same improvements (or even better) to the other algorithms implemented in OCaml, using Scala instead?

- What about the other programing languages instead of using SCALA? Can they perform better in practice?