Winning Cores in Parity Games

Steen Vester

Technical University of Denmark, Kgs. Lyngby, Denmark
stve@dtu.dk

Abstract

Whether parity games can be solved by a polynomial-time algorithm is a well-studied problem which has not yet been resolved. In this talk we propose a new direction for approaching this problem based on the novel notion of a winning core.

We give two different, but equivalent, definitions of a winning core and show a number of interesting properties about them. This includes showing that winning cores can be computed in polynomial time if and only if parity games can be solved in polynomial time and that computation of winning cores is in the intersection of \( \text{NP} \) and \( \text{co-NP} \).

We also present a deterministic polynomial-time approximation algorithm for solving parity games based on computing winning cores. It runs in time \( O(d \cdot n^2 \cdot m) \) where \( d \) is the number of colors, \( n \) is the number of states and \( m \) is the number of transitions. The algorithm returns under-approximations of the winning regions in parity games. It works remarkably well in practice as it solves all benchmark games from the PGSolver framework in our experiments completely and outperforms existing algorithms in most cases. Correctness of the output of the algorithm can be checked efficiently.

1 Introduction

Solving parity games [1] is an important problem of both theoretical and practical interest. This is known to be in \( \text{NP} \cap \text{co-NP} \) [2] and \( \text{UP} \cap \text{co-UP} \) [7] but in spite of the development of many different algorithms (see e.g. [13, 18, 8, 17, 9, 15]), frameworks for benchmarking such algorithms [6, 10] and families of parity games designed to expose the worst-case behaviour of existing algorithms [8, 4, 5] it has remained an open problem whether a polynomial-time algorithm exists.

Various problems for which polynomial-time algorithms are not known can been reduced in polynomial time to the problem of solving parity games. Among these are model-checking of the propositional \( \mu \)-calculus [11, 3, 16], the emptiness problem for parity automata on infinite binary trees [14, 2] and solving boolean equation systems [12].

Some of the most notable algorithms from the litterature of solving parity games include the recursive algorithms from [13, 15] using \( O(n^d) \) time, the small progress measures algorithm [8] using \( O(d \cdot m \cdot (n/d)^{d/2}) \) time, the strategy improvement algorithm [17] using \( O(n \cdot m \cdot 2^n) \) time, the big step algorithm [15] using \( O(m \cdot n^{d/3}) \) time and the dominion decomposition algorithm [9] using \( O(n^{\sqrt{d}}) \) time. Here, \( n \) is the number of states in the game, \( m \) is the number of transitions and \( d \) is the maximal color occurring in the game.

2 Contributions

First, we introduce some notation. In the following we fix a finite parity game \( G \) (for a definition, see e.g. [13]) with colors in \{1, ..., \( d \)\}. The set of winning states for player \( j \) in \( G \) is denoted \( W_j(G) \). We say that a (finite or infinite) sequence \( \rho = s_0s_1... \) of states with at least one transition is 0-dominating if the greatest color \( e = \max\{c(s_i) \mid i > 0\} \) occurring in a non-initial state of \( \rho \) is even and 1-dominating if it is odd. Examples are shown in Figure [1].
We say that a play $\rho$ begins with $k$ consecutive $j$-dominating sequences if there exist indices $i_0 < i_1 < \ldots < i_k$ with $i_0 = 0$ such that $\rho_{i_\ell} \rho_{i_\ell+1} \ldots \rho_{i_{\ell+1}}$ is $j$-dominating for all $0 \leq \ell < k$. This definition is straightforwardly extended to an infinite number of consecutive $j$-dominating sequences. As examples, the sequence on the left in Figure 1 begins with two consecutive 0-dominating sequences $s_0 s_1$ and $s_1 s_2 s_3$ whereas the sequence to the right begins with only one 0-dominating sequence $t_0 t_1$, but not two consecutive 0-dominating sequences.

For a player $j$ and a parity game $G$ the winning core $A_j(G)$ is defined as the set of states in $G$ from which player $j$ can force that the play begins with an infinite number of consecutive $j$-dominating sequences. Our main results on winning cores are the following.

**Proposition 1.** Let $\rho$ be a play. Then $\rho$ begins with an infinite number of consecutive $j$-dominating sequences if and only if $\rho$ is $j$-dominating and winning for player $j$.

**Theorem 1.** Let $G$ be a parity game and $j$ be a player. Then

1. $A_j(G) \subseteq W_j(G)$
2. $A_j(G) = \emptyset$ if and only if $W_j(G) = \emptyset$

**Proposition 2.** There exists a parity game $G$ where $A_j(G)$ is not a $j$-dominion.

**Theorem 2.** Computing winning cores is in $\text{NP} \cap \text{co-NP}$. 

**Theorem 3.** Computing winning cores can be done in polynomial time if and only if parity games can be solved in polynomial time.

Proposition 1 gives us two equivalent definitions of the same concept which is not immediately obvious. Theorem 1 provides us with valuable information about the winning cores and, further, it is used to design an algorithm for solving parity games based on computing winning cores. Proposition 2 is interesting as many algorithms for solving parity games focus on finding $j$-dominions whereas the winning core is a subset of winning states that is not necessarily a $j$-dominion. Finally, Theorem 2 and 3 make the search for a polynomial-time algorithm for computing winning cores a viable direction in the search for a polynomial-time algorithm for solving parity games.

The results are used to develop a new fixpoint algorithm that calculates under-approximations of the winning regions in parity games. This algorithm runs in time $O(d \cdot n^2 \cdot m)$ and is very fast in practice. Further, it can be efficiently checked whether the output of the algorithm is correct. This means that it can be applied with confidence when it outputs the correct result.

In our experiments the algorithm performs remarkably well returning the complete winning region in most cases. In Figure 2 are experimental results on correctness in randomly generated games. Further, the algorithm returns the correct results on all other benchmark games from the PGSolver framework on which it has been tested. A comparison of running times for some of the benchmarks can be seen below. These benchmarks are games designed to be difficult for some of the existing algorithms to solve. Our algorithm has been implemented in OCaml as a part of the PGSolver framework using the same basic data structures as the other algorithms.
\[d = 4\]
\[d = \lceil \sqrt{n} \rceil\]
\[d = n\]
\[n \backslash b\]
<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>5</th>
<th>10</th>
<th>2</th>
<th>5</th>
<th>10</th>
<th>2</th>
<th>5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.002%</td>
<td>0.000%</td>
<td>0.000%</td>
<td>0.114%</td>
<td>0.001%</td>
<td>0.000%</td>
<td>0.559%</td>
<td>0.011%</td>
<td>0.000%</td>
</tr>
<tr>
<td>1000</td>
<td>0.000%</td>
<td>0.000%</td>
<td>0.000%</td>
<td>0.571%</td>
<td>0.000%</td>
<td>0.000%</td>
<td>2.113%</td>
<td>0.000%</td>
<td>0.000%</td>
</tr>
</tbody>
</table>

Figure 2: Ratio of games where the algorithm did not return the entire winning region. \(n\) is the number of states, \(d\) is the number of colors and \(b\) is the out-degree. For every fixed \(n\), \(d\) and \(b\) the experiments were done on 100,000 games generated randomly by PGSolver [6].
References


