Solving Parity Games using Succinct Progress Measure and Register Games

Satya Prakash Nayak
Chennai Mathematical Institute
satyaprakash@cmi.ac.in

Abstract

The complexity of parity games is a long standing open problem that saw a major breakthrough in 2017 when two quasi-polynomial algorithms were published. This report presents two different approach to solving parity games in quasi-polynomial time.

First approach is based on *progress measures*, which allows us to reduce the space required from quasipolynomial to quasi-linear. Our key technical tools are a novel concept of ordered tree coding, and a *succinct tree coding* result that we prove using bounded adaptive multi-counters.

Second approach is based on the notion of *register game*, a parameterised variant of a parity game. The analysis of register games leads to a quasi-polynomial algorithm for parity games, a measure of complexity, the *register index*, which aims to capture the combined complexity of the priority assignement and the underlying game graph.

1 Introduction

A play in a *parity game* consists of a player, whom we shall call Eve, and her opponent, Adam, moving a token along the edges of a graph labelled with integer priorities, forever, thus forming an infinite path. Eve's objective is to force the highest priority that occurs infinitely often to be even, while Adam tries to stop her.

Parity games are fundamental in logic and verification because they capture—in an easy-to-state combinatorial game form—the intricate expressive power of nesting least and greatest fixpoint operators (interpreted over appropriate complete lattices), which play a central role both in the theory and in the practice of algorithmic verification. In particular, the modal μ-calculus model checking problem is polynomial-time equivalent to solving parity games [3], but parity games are much more broadly applicable to a multitude of modal, temporal, and fixpoint logics, and in the theory of automata on infinite words and trees.

The complexity of solving parity games that is, of deciding which player has a winning strategy still is, despite extended efforts, an open problem: it is in bot NP and coNP yet it is not known to admit a polynomial algorihm. After over twenty-five years of incremental improvements, Calude, Jain, Khoussainov, Li, and Stephan published the first quasi-polynomial solution [2]. Only a little later in the same year, Jurdzinski and Lazic presented an independent progress-measure based algorithm that achieves the same complexity [5]. Then third quasipolynomial algorithm came into existance, when Lehtinen [1] used a notion of register games to solve parity games. The report will represent these two algorihms (second and third).

Throughout the report, we use n to denote the number of vertices; η for numbers of vertices with an odd priority; m for number of edges and d is the smallest even number that is not smaller than the priority of any vertex.

2 Succinct Progress Measures

2.1 Succinct tree coding

An ordered tree is an oriented tree in which the children of a node are somehow ordered. We refer to the sequences of branching directions (linearly ordered numbers labelled on the

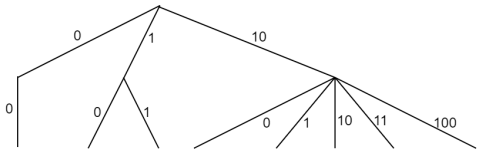


Figure 1

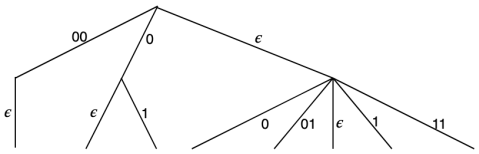


Figure 2

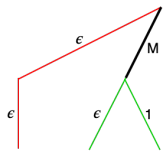
branches) that uniquely identify nodes as their *navigation paths*. For example consider the ordered tree of height 2 and with 8 leaves given in Figure 1 in which the branching directions are binary numbers. The navigation path to the right-most leaf in the tree is $(10, 100)$, since it is the child of the node with navigation path 10 reached from it via branching direction 100.

Our main idea is to use an order-preserving relabelling of the branching direction so that we can encode the navigation paths using less bits and more succinctly. We can see the ordered tree given in Figure 2 which is the same tree as in Figure 1 but ordered succinctly so that all the navigation paths used at most 2 bits.

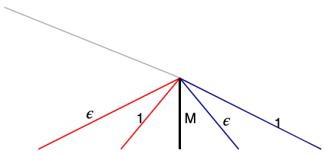
Now the question is how to encode a tree succinctly. Here we give an idea to do that recursively. First we define a strict linear ordering $<$ on finite binary strings as follows, for both binary digits b, and for all binary strings s and s':

$$0s < \epsilon,\ \epsilon < 1s,\ bs < bs' \text{ iff } s < s'.$$

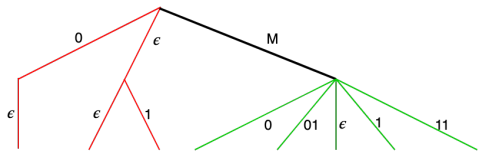
We have illustrated the algorithm with an example. For a given tree T (let that be the tree given in Figures 3c or 1), let M (as shown in Figure 3c) be a branching direction from the root such that both the subtrees: $T_<$ (tree given in Figure 3a) which is obtained by removing all the branches with first branching direction $\geq M$, and $T_>$ (it is null tree for our example) which is obtained by removing all the branches with first branching direction $\leq M$ has at most $\frac{l}{2}$ leaves where l is the number of leaves in T. Also let $T_=$ (tree given in Figure 3b) be the subtree which is obtained by taking the node which is reached by branching direction M as root. First we will encode subtres $T_<$ and $T_>$ and $T_=$ recursively (as shown in figures 3a, 3b and 3c) and append one leading 0 to the binary strings that code the first branching direction of $T_<$ and append one leading 1 to the first branching direction of $T_>$. Then we will labell the branch with branching direction M as ϵ (which gives us the succinctly coded tree in Figure 2). It then follows the next lemma.



(a) Subtree $T_<$



(b) Subtree $T_=$



(c) final step encoding Tree T

Figure 3 Step-wise succinctly encoding of a tree T. In all 3 figures, red colored is the left subtree ($T'_<$), green colored is the middle subtree ($T'_=$) and blue colored is the right subtree ($T'_>$).

▶ **Lemma 1.** *(Succinct tree coding). For every ordered tree of height h and with at most l leaves there is a tree coding in which every navigation path is an i-tuple of binary strings whose total length is at most $\lceil \log_2 l \rceil$, where $i \leq h$ is the length of the path.*

2.2 Succinct progress measure

For finite parity game graphs, a progress measure is a mapping from the n vertices to $\frac{d}{2}$-tuples of non-negative integers (that also satisfies the so-called progressiveness conditions on an appropriate set of edges [4]). An alternative interpretation is that a progress measure maps every vertex to a leaf in an ordered tree T in which each of the at most n leaves has a navigation path of length $\frac{d}{2}$. So the main idea is to use (succinctly) ordered tree T to represent the progress measure.

It is well known that existence of a progress measure is sufficient and necessary for existence of a winning strategy for Even from every starting vertex [4]. Similarly we can show that this is also true for existence of a *succinct progress measure* [5] (we are considering the trimmed version of progress measure) in which the ordered tree T is such that

- finite binary strings ordered as in section 2.1 are used as branching directions instead of non-negative integers, and
- every navigation path in T is a i-tuple of binary strings whose total length is bounded by $\lceil \log_2 \eta \rceil$, for some i, $0 \leq i \leq \frac{d}{2}$.

Since the values of a succinct progress measure (navigation paths in the tree) has length bounded by $\lceil \log_2 \eta \rceil$, this allow us to reduce the work space required from quasipolynomial to quasi-linear but the running time is still quasipolynomial.

We have illustrated the lifting algorihm for succinct progress measure in the following example.

2.3 Example

Consider the parity game $G = (V, V_E, V_A, E, \Omega)$ in Figure 4, where $V_A = \{v_0, v_2, v_3, v_4, v_7, v_9\}$; $V_E = \{v_1, v_5, v_6, v_8\}$ and priorities of vertices are as shown in the figure.

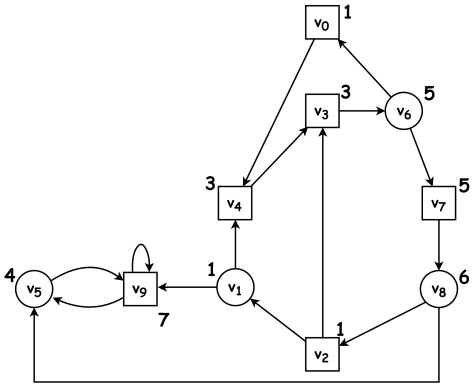


Figure 4

We will solve the parity game using succinct progress measure. So first we will assign $\mu(v)$ = empty sequence for all vertex v, then we have to choose a vertex v such that $\text{Lift}_v(\mu) \neq \mu$ and lift it in each iteration. Note that the number of vertices with odd priority η in graph G is 8, hence $\log_2(\eta) = 3$.

Since each operator for any vertex v, Lift_v will only change $\mu(v)$, if we choose a vertex $v \in V$ and lift it in an iteration then we can just write $\text{Lift}(\mu, v)$ = (value of) $\text{Lift}_v(\mu)(v)$ to decribe the lifting of the vertex v. In the first iteration we choose v_9, so we can write

$\text{Lift}(\mu, v_9) = \text{Lift}_v(\mu)(v_9) = \max(\mu(v_5)|_7, \mu(v_9)|_7) + 1 =$ empty sequence $+1 = (000)$ ($+1$ denotes the next binary tuple in succinct order whose length is bounded by 3.) Since $\text{Lift}_v(\mu)(v_9) > \mu(v_9)|_7$, if we keep lifting v_9 it will keep on increasing untill it becomes T, hence $\mu(v_9) = \mathsf{T}$. In the next iteration we can lift the vertex v_5, so $\text{Lift}(\mu, v_5) = \text{Lift}_v(\mu)(v_5) = \mu(v_9)|_4 = \mathsf{T}$. Similarly the following equations represent the liftings of vertices in the next iterations.

1. $\text{Lift}(\mu, v_0) = \mu(v_4)|_1 + 1 = (000)$
2. $\text{Lift}(\mu, v_7) = \mu(v_8)|_5 + 1 = (000)$
3. $\text{Lift}(\mu, v_6) = \min(\mu(v_0)|_5, \mu(v_7)|_5) + 1 = (000) + 1 = (000, \epsilon)$
4. $\text{Lift}(\mu, v_3) = \mu(v_6)|_1 + 1 = (000, \epsilon) + 1 = (000, \epsilon, \epsilon)$
5. $\text{Lift}(\mu, v_2) = \max(\mu(v_1)|_1, \mu(v_3)|_1) + 1 = (000, \epsilon, \epsilon) + 1 = (000, \epsilon, \epsilon, \epsilon)$
6. $\text{Lift}(\mu, v_8) = \min(\mu(v_2)|_6, \mu(v_5)|_6) = (000)$
7. $\text{Lift}(\mu, v_4) = \mu(v_3)|_3 + 1 = (000, \epsilon, \epsilon) + 1 = (00)$
8. $\text{Lift}(\mu, v_1) = \mu(v_4)|_1 + 1 = (00) + 1 = (00, 0)$
9. $\text{Lift}(\mu, v_0) = (00, 0)$
10. $\text{Lift}(\mu, v_7) = (000, \epsilon)$
11. $\text{Lift}(\mu, v_6) = (00)$
12. $\text{Lift}(\mu, v_3) = (00, 0)$
13. $\text{Lift}(\mu, v_2) = (00, 0, \epsilon)$
14. $\text{Lift}(\mu, v_8) = (00)$
15. $\text{Lift}(\mu, v_7) = (00, 0)$
16. $\text{Lift}(\mu, v_6) = (00, \epsilon)$
17. $\text{Lift}(\mu, v_3) = (00, \epsilon, 0)$
18. $\text{Lift}(\mu, v_2) = (00, \epsilon, 0, \epsilon)$
19. $\text{Lift}(\mu, v_4) = (00, \epsilon, \epsilon)$
20. $\text{Lift}(\mu, v_0) = (00, \epsilon, \epsilon, 0)$
21. $\text{Lift}(\mu, v_1) = (00, \epsilon, \epsilon, 0)$
22. $\text{Lift}(\mu, v_2) = (00, \epsilon, \epsilon, \epsilon)$

The algorithms terminates after the above steps and the winning region for Eve is $\text{dom}(\mu*) = \{v_0, v_1, v_2, v_3, v_4, v_6, v_7, v_8\}$. The small progress measure algorithm will also take nearly these many steps but here we can see that for any vertex v, $\mu(v)$ is a tuple whose length is bounded by 3 which affects the space and time complexity of the algorithm.

3 Register Games

3.1 Definitions

The *k-register game* consists of a normal parity game, augmented with a tuple $(r_0, \ldots, r_k)$ of registers that keeps a partial record of the history of the game. During a turn, several things happen: the player whose turn it is in the parity game moves onto a successor position of their choice, which has some priority p and Eve chooses an index i, $0 \leq i \leq k$; then, the registers get updated according to both i and p, and an output between 0 and $2k + 1$ is produced, also according to i and p.

The update wipes out the contents of registers with index lower than i: for $j < i$, r_j is set to 0. Meanwhile r_i is set to p and r_j for $j > i$ to $\max(r_j, p)$. In other words, each register r_j keeps track of the highest priority seen since Eve last chose r_i with $i \geq j$. The output is $2i$ if $\max(r_i, p)$ is even, and $2i + 1$ otherwise. Then, in the limit, Eve wins a play if the largest output that occurs infinitely often is even.

Since the winning condition of the register game is a parity condition, we can formally define the k-register game on a parity game arena G as a parity game on an arena $R_E^k(G)$, of which the positions are $(v, \bar{r}, t) \in V^G \times I^{k+1} \times \{0, 1\}$ where $\bar{r}$ that represent the contents of the registers and the binary variable t indicates whether the next move consists of Eve's choice of register ($t = 0$), or a move in the underlying parity game ($t = 1$); Ω assigns priorities from $[0 \ldots 2k + 1]$; edges can be defined accordingly.

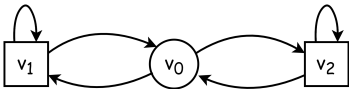


Figure 5 Example

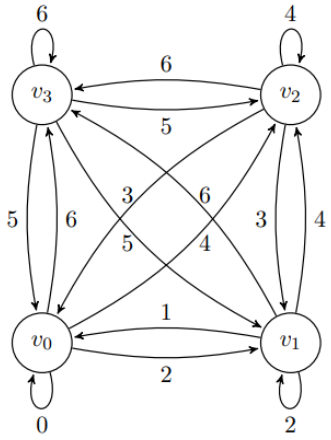


Figure 6 Parity game of register-index 1

3.2 Register index

We can show that if Adam has a winning strategy from a position v in a parity game G, then he has a winning strategy from all positions $(v, \bar{r}, t)$ in $R_E^k(G)$ for any k. However, if Eve has a winning strategy in G from v, then she also has a winning strategy from all positions $(v, \bar{r}, t)$ in $R_E^k(G)$ as long as k is taken to be large enough. Exactly how many register she needs depends on the parity game arena. We call this the *register-index* of the parity game it is a measure of complexity which takes into account both the priority assignment of the parity game and the structure of the underlying graph, and we will see later that it is bounded logarithmically in the number of disjoint cycles of a parity game arena.

For finite arenas, where the number of disjoint cycles is bounded by the number of positions, solving parity games reduces to solving the register game with a number of registers logarithmic in the size of the game. This results in a quasi-polynomial parity game algorithm with running time in $2^{O((\log n)^3)}$ [1]. We have illustrated this with the following example.

3.3 Example

Consider the the parity game $G = (V^G, V_E^G, V_A^G, E^G, \Omega^G)$ in Figure 5, where $V_E^G = \{0\}$; $V_A^G = \{1, 2\}$ and for each $0 \leq i \leq 2$, $\Omega^G(v_i) = i$.

We can see that the maximal number of vertex-disjoint cycles in the underlying graph of G is 2 (i.e the cycle with one vertex v_1 and cycle with one vertex v_2.) So the register index is atmax $1 + \log_2 2 = 2$. So we should solve 2-register game $R_E^2(G)$ to solve the parity game G.

Now the arena of the 2-register game $R_E^2(G)$ on G in which Eve controls the registers, consists of $R_E^k(G) = (V, V_E, V_A, E, \Omega)$ as follows.

- $V = \{v_0, v_1, v_2\} \times \{0, 1, 2\}^3 \times \{0, 1\}$
- $V_A = \{v_1\} \times \{0, 1, 2\}^3 \times \{1\}$
- $V_E = V \setminus V_A = \{v_0, v_1, v_2\} \times \{0, 1, 2\}^3 \times \{0\} \cup \{v_1, v_2\} \times \{0, 1, 2\}^3 \times \{1\}$
- E is the disjoint union of sets of edges E_{move} and E_i for all $i = 0, 1, 2$ where:
  E_{move} consists of edges $((v_1, \bar{r}, 1), (v_1, \bar{r}, 0))$, $((v_1, \bar{r}, 1), (v_0, \bar{r}, 0))$, $((v_0, \bar{r}, 1), (v_1, \bar{r}, 0))$, $((v_0, \bar{r}, 1), (v_2, \bar{r}, 0))$, $((v_2, \bar{r}, 1), (v_0, \bar{r}, 0))$, $((v_2, \bar{r}, 1), (v_2, \bar{r}, 0))$, for each $\bar{r} \in \{0, 1, 2\}^3$.
  For each $i = 0, 1, 2$, E_i consists of edges $((v, \bar{r}, 0), (v, \bar{r}', 1))$ for each $\bar{r} \in \{0, 1, 2\}^3$ such that:

- $r'_j = \max(r_j, \Omega^G(v))$ for $j > i$,
- $r'_j = \Omega^G(v))$ for $j = i$, and
- $r'_j = 0$ for $j < i$,

- $\Omega(e) = \begin{cases} 0 & \text{if } e \in E_{move}, \\ 2i & \text{if } e = ((v, \bar{r}, 0), (v, \bar{r}', 1)) \in E \text{ and } \max(r_i, \Omega^G(v)) \text{ is even}, \\ 2i+1 & \text{if } e = ((v, \bar{r}, 0), (v, \bar{r}', 1)) \in E \text{ and } \max(r_i, \Omega^G(v)) \text{ is odd}. \end{cases}$

To solve the parity game G, we will first solve the register game $R^2_E(G)$ (which is also a parity game) using small progress measure [4]. Since $v_1 \in V_A$ and all the cycles containing the vertex $(v_1, \bar{r}, t)$ are odd cycle, we can find that the least progress measure μ^* returned by the lifting algorihm is such that $\text{dom}(\mu^*) = \{v_0, v_2\} \times \{0, 1, 2\}^3 \times \{0, 1\}$, hence the winning region for Eve in the parity game G are $\{0, 2\}$.

The size of the priority co-domain is perhaps the simplest way of measuring the complexity of parity games: many parity game algorithms are exponential in the number of priorities, and therefore polynomial on parity games with a fixed number of priorities. And as we can see that the number of priorities in a k-register game is $2k + 2$, where k is the register-index. In this example the number priorities in the parity game was small, but there are games with large number of priorities but has small register-index in those cases ths algorithm will be useful.

3.4 Conclusion

The complexity of this algorithm to solve parity games is not competitive with respect to existing quasi-polynomial algorithms and it is not clear how to avoid the space-complexity involved in building $R_E^{1+\log n}(G)$. This algorihm is not useful to solve parity games in practice. However many games have low register-index, including those that exhibit worst-case complexity for other algorithms; on such games this approach is expected to work well.

For example, we can construct a little complicated family of parity games, which has linear entanglement, tree-width and number of priorities, yet constant register-index 1. It consists arenas with vertices $v_0 \dots v_n$ with priority edges (v_j, v_i) of priority $2i$ for $j \leq i$ and $2j - 1$ for $j > i$. For $n = 3$, consider the parity game given in Figure 6.

For the space-complexity issue, we can see that the register index approach seems suited for a symbolic implementation: given a symbolically represented parity game, the k-register game can also be represented symbolically without significant blow-up. This would bypass the space-complexity of the algorithm for parity games that benefit from concise symbolic representations.

We also propose some other ways to imporve the algorithm. We saw that if a player has a winning strategy in G from v, then she also has a winning strategy from all positions $(v, \bar{r}, t)$ in $R^k_E(G)$, where k is the register-index of G. So while solving the k-register game $R^k_E(G)$ using progress measure, we can change the lifing algorihm a little bit so that for a vertex v whenever we assign $\mu((v, \bar{r_0}, t_0)) = \mathsf{T}$ for some t_0 and register values $\bar{r_0}$, we also assign $\mu((v, \bar{r}, t)) = \mathsf{T}$ for all t and for all register values $\bar{r}$. This may reduce the running time to some extent. We can see that in the example given in Section 3.3, this method will reduce number of iterations by a lot, because we have to consider only one odd cycle containing $(v_1, \bar{r}, t)$ then $\mu((v_1, \bar{r}', t'))$ for all $\bar{r}'$ and t' will become T. We may also do something similar for the other vertices also (not sure). And also to reduce the work space required, we can use succinct progress measure instead of small progress measure, ofcourse the work space required will still be quasipolynomial but it can be a little effective for small examples.

References

1 Udi Boker and Karoliina Lehtinen. Register games. *CoRR*, abs/1902.10654, 2019. URL: http://arxiv.org/abs/1902.10654, arXiv:1902.10654.

2 Cristian S. Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. Deciding parity games in quasipolynomial time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, pages 252–263, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3055399.3055409, doi:10.1145/3055399.3055409.

3 E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of μ-calculus. In Costas Courcoubetis, editor, *Computer Aided Verification*, pages 385–396, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

4 Marcin Jurdziński. Small progress measures for solving parity games. In Horst Reichel and Sophie Tison, editors, *STACS 2000*, pages 290–301, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

5 Marcin Jurdzinski and Ranko Lazic. Succinct progress measures for solving parity games. *CoRR*, abs/1702.05051, 2017. URL: http://arxiv.org/abs/1702.05051, arXiv:1702.05051.