Parity Games On Graphs With Medium Tree-width

John Fearnley[1] and Oded Lachish[2]

[1] Department of Computer Science, University of Liverpool, UK
[2] Department of Computer Science and Information Systems, Birkbeck, University of London, UK

Abstract. This paper studies the problem of solving parity games on graphs with bounded tree-width. Previous work by Obdržálek has produced an algorithm that uses $n^{O(k^2)}$ time and $n^{O(k^2)}$ space, where k is the tree-width of the graph that the game is played on. This paper presents an algorithm that uses $n^{O(k \log n)}$ time and $O(n + k \log n)$ space. This is the fastest known algorithm for parity games whose tree-width k satisfies (in standard asymptotic notation) $k \in \omega(\log n)$ and $k \in o(\sqrt{n}/\log n)$.

1 Introduction

In this paper we study the problem of solving parity games on graphs with bounded tree-width. A parity game is a two player game that is played on a finite directed graph. Parity games have received much attention due to the fact that they share a polynomial time equivalence with the μ-calculus model checking problem [7, 16], and with the non-emptiness problem for non-deterministic parity tree automata [8]. The complexity of solving parity games is also interesting, because it is one of the few non-cryptographic problems that lies in NP $\cap$ co-NP (and even in UP $\cap$ co-UP [9]). This implies that they are highly unlikely to be either NP-complete or co-NP-complete. However, despite much effort from the community, no polynomial time algorithm has been found.

Problems that are not known to admit polynomial time algorithms for general inputs are often studied from the perspective of *parametrised complexity* [5]. Here, the objective is to find a class of restricted inputs for which a polynomial time algorithm can be devised, and tree-width has turned out to be one of the most successful restrictions for this purpose. The tree-width of a graph is a parameter that measures how close the graph is to being a tree, which originated from the work of Robertson and Seymour on graph minors [14].

When we are given a guarantee that the input graph has tree-width bounded by some constant k, we want to find algorithm whose running time can be expressed as $O(f(k) \cdot n^{g(k)})$. Since k is constant, this is a polynomial time upper bound. A more restrictive notion is *fixed parameter tractability*, where we are looking for an algorithm whose running time can be expressed as $O(f(k) \cdot n^c)$, where c is a constant that is independent of k. Many problems that are NP-hard in general have been found to admit fixed parameter tractable algorithms when the input is restricted to be a graph with bounded tree-width [4].

Since parity games have no known polynomial time algorithm, it is natural to ask whether the problem becomes easier when the game graph has bounded tree-width. Although no fixed parameter tractable algorithm is known for parity games, Obdržálek has found a polynomial time algorithm [13]. His algorithm uses a dynamic programming technique that exploits the structure of a graph with bounded tree-width. This yields an algorithm that solves a parity game on a graph with tree-width k in $n^{O(k^2)}$ time and $n^{O(k^2)}$ space.

Our contribution. While Obdržálek's algorithm uses dynamic programming, which is a technique that is traditionally applied to graphs of bounded tree-width, we approach the problem using the techniques that have been developed for solving parity games. Our work builds upon the algorithm of Jurdziński, Paterson, and Zwick, that solves parity games in sub-exponential time for arbitrary graphs [10]. Their algorithm is a modification of McNaughton's exponential-time recursive algorithm for parity games [11]. They introduced the concept of a dominion, which is a structure that appears in parity games. They showed that if McNaughton's algorithm is equipped with a preprocessing procedure that removes all small dominions, then the resulting algorithm runs in $n^{O(\sqrt{n})}$ time. This is currently the fastest algorithm for solving parity games on general graphs. These techniques have since been adapted by Schewe to produce the best known algorithm for parity games with a small number of distinct priorities [15].

Our algorithm applies these techniques to parity games on graphs of bounded tree-width. Instead of using preprocessing to remove small dominions, our algorithm uses the fact that the graph has bounded tree-width to break it into smaller pieces, and then uses preprocessing to remove every dominion that is entirely contained within each piece. We show that equipping McNaughton's algorithm with this preprocessing procedure produces an algorithm that runs in $n^{O(k \log n)}$ time for graphs with tree-width bounded by k. Therefore, our algorithm is asymptotically faster than Obdržálek's algorithm whenever $k \in \omega(\log n)$, where ω is standard asymptotic notation. It is also asymptotically faster than the algorithm of Jurdziński, Paterson, and Zwick whenever $k \in o(\sqrt{n}/\log n)$.

The other advantage that our algorithm has over Obdržálek's algorithm is the amount of space that is uses. Each step in Obdržálek's dynamic programming creates a vast amount of information about the game that must be stored, which leads to the $n^{O(k^2)}$ space complexity of the algorithm. By contrast, our algorithm always uses $O(n^2)$ space. This may make our algorithm more attractive even in the cases where $k \notin \omega(\log n)$.

2 Parity Games

A parity game is defined by a tuple $(V, V_0, V_1, E, \mathrm{pri})$, where V is a set of vertices and E is a set of edges, which together form a finite directed graph. We assume that every vertex has at least one outgoing edge. The sets V_0 and V_1 partition V into vertices belonging to player Even and vertices belonging to player Odd, respectively. The function $\mathrm{pri} : V \to \mathbb{N}$ assigns a *priority* to each vertex. We will use $\mathrm{MaxPri}(G) = \max\{\mathrm{pri}(v) \ : \ v \in V\}$ to denote the largest priority in G.

At the beginning of the game a token is placed on the starting vertex v_0. In each step, the owner of the vertex that holds the token must choose one outgoing edge from that vertex and move the token along it. In this fashion, the two players form an infinite path $\pi = \langle v_0, v_1, v_2, \dots \rangle$, where $(v_i, v_{i+1}) \in E$ for every $i \in \mathbb{N}$. To determine the winner of the game, we consider the set of priorities that occur *infinitely often* along the path. This is defined to be:

$$\mathrm{Inf}(\pi) = \{d \in \mathbb{N} \ : \ \text{For all } j \in \mathbb{N} \text{ there is an } i > j \text{ such that } \mathrm{pri}(v_i) = d\}.$$

Player Even wins the game if the highest priority occurring infinitely often is even, and player Odd wins the game otherwise. In other words, player Even wins the game if and only if $\max(\mathrm{Inf}(\pi))$ is even.

A positional strategy for Even is a function that chooses one outgoing edge for every vertex in V_0. A strategy is denoted by $\sigma : V_0 \to V$, with the condition that $(v, \sigma(v)) \in E$, for every Even vertex v. Positional strategies for player Odd are defined analogously. The sets of positional strategies for Even and Odd are denoted by Π_0 and Π_1, respectively. Given two positional strategies σ and τ, for Even and Odd respectively, and a starting vertex v_0, there is a unique path $\langle v_0, v_1, v_2 \dots \rangle$, where $v_{i+1} = \sigma(v_i)$ if v_i is owned by Even and $v_{i+1} = \tau(v_i)$ if v_i is owned by Odd. This path is known as the *play* induced by the two strategies σ and τ, and will be denoted by $\mathrm{Play}(v_0, \sigma, \tau)$.

An infinite path $\langle v_0, v_1, \dots \rangle$ is said to be consistent with an Even strategy $\sigma \in \Pi_0$ if $v_{i+1} = \sigma(v_i)$ for every i such that $v_i \in V_0$. If $\sigma \in \Pi_0$ is strategy for Even, and v_0 is a starting vertex then we define $\mathrm{Paths}(v_0, \sigma)$ to be the function that gives every path starting at v_0 that is consistent with σ:

$$\begin{aligned}\mathrm{Paths}_0(v_0, \sigma) = \{\langle v_0, v_1, \dots \rangle \in V^\omega \ : \ & \text{for all } i \in \mathbb{N}, \text{ if } v_i \in V_0 \\ & \text{then } v_{i+1} = \sigma(v_i), \text{ and if } v_i \in V_1 \text{ then } (v_i, v_{i+1}) \in E)\}.\end{aligned}$$

An Even strategy is called a *winning strategy* from a given starting vertex if player Even can use the strategy to ensure a win when the game is started at that vertex, no matter how player Odd plays in response. In other words, a strategy $\sigma \in \Pi_0$ is a winning strategy for player Even from the starting vertex v_0 if $\max(\mathrm{Inf}(\pi))$ is even for every path $\pi \in \mathrm{Paths}_0(v_0, \sigma)$. The strategy σ is said to be winning for a set of vertices $W \subseteq V$ if it is winning for every vertex $v \in W$. Winning strategies for player Odd are defined analogously.

A game is said to be *positionally determined* if one of the two players always has a positional winning strategy. We now give a fundamental theorem, which states that parity games are positionally determined.

Theorem 1 ([6, 12]). *In every parity game, the set of vertices V can be partitioned into two sets (W_0, W_1), where Even has a positional winning strategy for W_0, and Odd has a positional winning strategy for W_1.*

The sets W_0 and W_1, whose existence is implied by Theorem 1, are called *winning sets*, and our objective is to find an algorithm that computes this partition. Occasionally, we will want to refer to winning sets from different games, and we will use W_i^G to refer to the winning set for player i in the game G.

3 McNaughton's Algorithm for Parity Games

In this section we describe the McNaughton's algorithm for parity games. This algorithm was first formulated by McNaughton [11], and was presented for parity games by Zielonka [17]. We will also describe the improvements that have been made to this algorithm by Jurdziński, Paterson, and Zwick. Our algorithm will build upon the techniques that have been developed by these authors.

To begin, we must define the *attractor* of a set of vertices W. This is the set of vertices from which one of the players can force the token to arrive at W, no matter how the opponent plays in response. We will give the definitions for attractors of player Even. To define the attractor we use the Pre^0 operator:

$$\begin{aligned}\mathrm{Pre}^0(W) = \{v \in V_0 \ : \ &\text{There exists } (v,u) \in E \text{ such that } u \in W\} \cup \\ &\{v \in V_1 \ : \ \text{For all } (v,u) \in E \text{ we have } u \in W\}\end{aligned}$$

This operator gives the set of vertices from which Even can force play into the set W in exactly one step. To compute the attractor, we iteratively apply this operator until a fixed point is reached:

$$\begin{aligned}W_0 &= W \\ W_{i+1} &= \mathrm{Pre}^0(W_i) \cup W_i\end{aligned}$$

Since each set $W_i \subseteq W_{i+1}$, we know that a fixed point must be reached after at most $|V|$ iterations. Therefore, we define the attractor for player Even of a set of vertices W to be $\mathrm{Attr}^0(W) = W_{|V|}$. The corresponding function $\mathrm{Attr}^1(W)$, which gives attractors for player Odd, can be defined analogously.

Algorithm 1 `McNaughton`(G)

Input: A parity game $G = (V, V_0, V_1, E, \mathrm{pri})$
Output: The partition of winning sets (W_0, W_1)

```
p := MaxPri(G); d := p mod 2
A := Attr^d({v ∈ V : pri(v) = p})
if A = V then
    (W_d, W_{1-d}) := (V, ∅)
    return (W_0, W_1)
end if
(W'_0, W'_1) := McNaughton(G ↾ (V \ A))
if W'_{1-d} = ∅ then
    (W_d, W_{1-d}) := (V, ∅)
else
    D := Attr^{1-d}(W'_{1-d})
    (W''_0, W''_1) := McNaughton(G ↾ (V \ D))
    (W_d, W_{1-d}) := (W''_d, W''_{1-d} ∪ D)
end if
return (W_0, W_1)
```

McNaughton's algorithm is shown as Algorithm 1. Given a set of vertices $W \subseteq V$ such that for each $v \in W$ there is an edge (v, u) with $u \in W$, we use $G \restriction W$ to denote the *sub-game* induced by the set W. In other words, $G \restriction W$ is the game obtained by removing every vertex in $V \setminus W$ from the game G, and all edges that are incident to these vertices. The algorithm makes two recursive calls on sub-games of size at most $|V| - 1$, which implies the following theorem.

Theorem 2 ([11, 17]). *If McNaughton's algorithm is applied to a parity game with n vertices, then it will compute the partition into winning sets in $O(2^n)$ time.*

Jurdziński, Paterson, and Zwick introduced the concept of a *dominion* [10], which is useful for analysing McNaughton's algorithm. A set of vertices W is a *trap* for player d if there is no edge (v, u) such that $v \in V_d \cap W$ and $u \notin W$, and if for every vertex $v \in V_{1-d} \cap W$ there is an edge (v, u) such that $u \in W$. In other words, the set of vertices W is a trap for a player d if player $1 - d$ has a strategy to ensure that player d cannot leave W. A trap W for player d is a dominion if player $1 - d$ wins everywhere in the game $G \restriction W$.

Definition 3 (Dominion ([10])). *A set $W \subseteq V$ is a dominion for player d if W is a trap for player $1 - d$, and player d has a winning strategy for every vertex in the game $G \restriction W$.*

To see why dominions are useful for analysing McNaughton's algorithm, we will give two trivial facts about traps in parity games. Firstly, the complement (with respect to the set V) of an Even attractor is a trap for Even, and the complement of an Odd attractor is a trap for Odd. Secondly, the set W_0 is a trap for Odd and the set W_1 is a trap for Even. These two facts imply that the set W'_{1-d} computed by the first recursive call of the algorithm is a dominion.

Proposition 4. *The set W'_{1-d} is a dominion for player $1 - d$.*

Jurdziński, Paterson, and Zwick used this idea to speed up McNaughton's algorithm. Before making the first recursive call, their algorithm performs a pre-processing step that removes all player $1 - d$ dominions that are smaller than some threshold. This guarantees that the size of the set W'_{1-d}, which is returned by the first recursive call, must be larger than this threshold. Since the set W'_{1-d} is removed from the game, this means that the second recursive call will be performed on a significantly smaller game. By carefully balancing the amount of time spent on pre-processing and the size of the dominions that are removed, they obtained a bound of $n^{O(\sqrt{n})}$ on the running time of their algorithm.

4 Tree Width

To define the tree-width of a parity game, we will use the tree-width of the undirected graph that is obtained when the orientation of the edges in the game is ignored. Therefore, we will use the following definition of a tree decomposition.

Definition 5 (Tree Decomposition). *For each game G, the pair (T, X), where $T = (I, F)$ is an undirected tree and $X = \{X_i : i \in I\}$ is a family of subsets of V, is a tree decomposition of G if all of the following hold:*

1. $\bigcup_{i \in I} X_i = V$,
2. *for every $(v, u) \in E$ there is an $i \in I$ such that $v \in X_i$ and $u \in X_i$, and*
3. *for every $i, j \in I$, if $k \in I$ is on the unique path from i to j in T, then $X_i \cap X_j \subseteq X_k$.*

The *width* of a tree decomposition (T, X) is $\max\{|X_i| - 1 : i \in I\}$, which is the cardinality of the largest set contained in X minus 1. The *tree-width* of a game G is the smallest width that is obtained by a tree decomposition of G. We will discuss the computation of a tree decomposition for a parity game in Section 6, and for the time being we will assume that a tree decomposition is given along with the input to our algorithm.

We can now explain the properties of a tree decomposition that will be used in our algorithm. Suppose that $i \in I$ is some vertex in the tree decomposition. If we remove i from the graph T, then we will produce a forest, and each of the trees in this forest will have exactly one vertex that is adjacent to i in T. This allows us to use the edges of i to identify each tree in the forest. For each edge $(i, j) \in F$, we use $T_{(i,j)}$ to denote the tree rooted at j that appears when i is removed from T. For each tree $T_{(i,j)}$, we define $V_{(i,j)} = \{v \in V : v \in X_l \text{ for some } l \in T_{(i,j)}\}$ to be the set of vertices in G that is contained in $T_{(i,j)}$.

For our algorithm, the most important property of a tree decomposition is that each set X_i is a *separator* in the graph G. This means that in order to move from some vertex $v \in V_{(i,j)}$ to some vertex $u \in V_{(i,l)}$, where $j \neq l$, we must pass through some vertex in X_i. In other words, if every vertex in X_i is removed from the graph, then the sets $V_{(i,j)}$ will be disconnected from each other.

Proposition 6. *Suppose that $i \in I$ and let (i, j) and (i, l) be two edges in F such that $j \neq l$. If $v \in V_{(i,j)}$ and $u \in V_{(i,l)}$, then every path from v to u must pass through at least one vertex in X_i.*

Our algorithm will use separators to break the graph into smaller pieces. In particular, it will find a separator that splits the graph into at least two pieces, where each piece contains at most two-thirds of the vertices in the original graph. The following proposition is a standard result for graphs of bounded tree width [14], which shows that such a separator must always exist.

Proposition 7. *Let G be a game with at least $3k+3$ vertices, and let (T, X) be a tree decomposition of G with width k. There is some $i \in I$ such that $|V_{(i,j)}| \leq 2|V|/3$ for all $(i, j) \in F$.*

We define $\text{Split}(G, (T, X))$ to be a function that, given a game G with tree decomposition (T, X), selects some vertex $i \in I$ that satisfies the condition given by Proposition 7. Obviously, this function can be computed in polynomial time.

5 Our Algorithm

In this section, we will describe the approach that our algorithm takes in order to solve a game $G = (V, V_0, V_1, E, \text{pri})$ with tree decomposition $(T = (I, F), X)$ of width k, where $i = \text{Split}(G)$ and $d = \text{MaxPri}(G) \bmod 2$. The key idea behind our algorithm is to break the game into sub-games using the separator X_i. Each sub-game will be preprocessed separately in order to remove every player $1 - d$ dominion that is entirely contained within the sub-game.

We begin by describing the preprocessing procedure that is used to remove every dominion for player $1 - d$ that does not contain a vertex in X_i. For each edge $(i, j) \in F$, we define a preprocessing game G_j on the vertices in $V_{(i,j)} \cup X_i$. The only difference between $G \restriction (V_{(i,j)} \cup X_i)$ and G_j is that all of the vertices in X_i are given to player d. Moreover, every vertex in X_i is given a self loop edge, and its priority is changed to 0 if d is even, and 1 if d is odd. These changes is to allow player d to win the game if the token arrives at a vertex in X_i.

Definition 8 (Preprocessing Game G_j). *Let $G = (V, V_0, V_1, E, \text{pri})$ be a game where $d = \text{MaxPri}(G) \bmod 2$, let (T, X) be a tree decomposition of G, and let $i = \text{Split}(G)$. For each edge $(i, j) \in F$ we define the game $G_j = (V', V'_0, V'_1, E', \text{pri}')$ as follows:*

$$
\begin{aligned}
V' &= V_{(i,j)} \cup X_i \\
V'_d &= (V_0 \cap V') \cup X_i, \\
V'_{1-d} &= (V_1 \cap V') \setminus X_i, \\
E' &= (E \cap (V' \times V')) \cup \{(v, v) \ : \ v \in X_i\} \\
\text{pri}'(v) &= \begin{cases} 0 & \text{if } d \text{ is even and } v \in X_i, \\ 1 & \text{if } d \text{ is odd and } v \in X_i, \\ \text{pri}(v) & \text{otherwise.} \end{cases}
\end{aligned}
$$

Note that the definition of a preprocessing game ensures that every vertex must have at least one outgoing edge. This is because the fact that X_i is a separator implies that the preprocessing game must include all outgoing edges from the vertices in $V_{(i,j)}$, and every vertex in $v \in X_i$ is given a self loop (v, v). Moreover, since no new vertices are added, and the only new edges are self loops, if (T, X) is a tree decomposition of width k for the original game, then (T, X) is also a tree decomposition of width k for the preprocessing game.

The algorithm will call itself recursively in order to solve each preprocessing game G_j. It will therefore compute a partition $(W_0^{G_j}, W_1^{G_j})$, which are the winning sets for the two players in the game G_j. The first thing that we can prove is that the winning set for player $1 - d$ in the preprocessing game G_j must be contained in the winning set for player $1 - d$ in the game G. Therefore, we can remove the vertices in the set $W_{1-d}^{G_j}$ from the game G and add them to the set W_{1-d}, which is the winning set for player $1 - d$ in the game G.

Proposition 9. *If $d = \text{MaxPri}(G) \bmod 2$, then we have $W_{1-d}^{G_j} \subseteq W_{1-d}^{G}$ for every $(i, j) \in F$.*

Proof. We begin by arguing that $W_{1-d}^{G_j}$ is a trap for player d in the game G. We will prove this claim by contradiction. Suppose that there is an edge (v, u) with $v \in V_d \cap W_{1-d}^{G_j}$ and $u \in V \setminus W_{1-d}^{G_j}$. We must have $v \in X_i$, because otherwise the edge (v, u) would be contained in E', and this would imply that $W_{1-d}^{G_j}$ is not a trap for player d in G_j. Since $v \in X_i$, we know that $(v, v) \in E'$ and that $\mathrm{pri}'(v) = d \bmod 2$. Therefore, player d has a winning strategy for the vertex v in the game G_j, which contradicts the fact that $v \in W_{1-d}^{G_j}$.

Since $W_{1-d}^{G_j}$ is a trap for player d in the game G, player $1-d$ can use the winning strategy for $W_{1-d}^{G_j}$ to force a win from that set in the game G. This proves the claim that $W_{1-d}^{G_j} \subseteq W_{1-d}^{G}$. □

The next proposition gives the reason why this preprocessing procedure is useful. It states that, after each of the sets $W_{1-d}^{G_j}$ has been removed, every dominion for player $1-d$ must use at least one vertex from the separator X_i.

Proposition 10. *If $d = \mathrm{MaxPri}(G) \bmod 2$ then for every player $1-d$ dominion D that is contained in $V \setminus \bigcup_{(i,j)\in F} W_{1-d}^{G_j}$ we have $D \cap X_i \neq \emptyset$.*

Proof. We will prove this claim by contradiction. Suppose that there is a player $1-d$ dominion D that is contained in $V \setminus \bigcup_{(i,j)\in F} W_{1-d}^{G_j}$ with the property $D \cap X_i = \emptyset$. If D is not entirely contained in some set $V_{(i,j)}$, then we will consider the dominion $D' = D \cap V_{(i,j)}$ for some edge $(i, j) \in F$ such that $D \cap V_{(i,j)} \neq \emptyset$. The fact that X_i is a separator in G implies that D' is also a dominion for player $1-d$. Therefore, from now on, we can assume that $D \subseteq V_{(i,j)}$.

Since D is a player $1-d$ dominion, Definition 3 implies that player $1-d$ has a winning strategy for every vertex in the sub-game $G \restriction D$. Since D is a trap for player d and $D \cap X_i = \emptyset$, this strategy allows player $1-d$ to win from every vertex in D in the preprocessing game G_j. This implies that $D \subseteq W_{1-d}^{G_j}$, which contradicts the fact that $D \subseteq V \setminus \bigcup_{(i,j)\in F} W_{1-d}^{G_j}$. □

Recall that Proposition 4 implies that the winning set for player $1-d$ that is returned by the first recursive call of McNaughton's algorithm must be a dominion for player $1-d$. The property given by Proposition 10 allows us to conclude that this winning set must contain at least one vertex in the separator X_i. Since the algorithm removes the winning set for player $1-d$ from the game, we know that at least one vertex will be removed from the separator X_i. If the graph has tree width k, then repeating this procedure $k+1$ times must remove every vertex from the set X_i. Since X_i is a separator, this implies that the game will have been split into at least two disjoint parts, which can be solved separately.

Algorithm 2 shows the algorithm `Preprocess`(G, i), which performs preprocessing for the game G around the separator X_i. It returns the set W_{1-d}, which is the attractor for player $1-d$ to the union of the winning sets $W_{1-d}^{G_j}$. Algorithm 3 is identical to Algorithm 1, except that calls to `McNaughton` have been replaced by calls to `Solve`.

Algorithm 4 shows the main algorithm. It has two special cases: when the number of vertices is small the algorithm applies McNaughton's algorithm, and when the separator X_i is empty the algorithm calls itself recursively to solve each piece independently. If neither of the two special cases are applicable, then the algorithm runs the preprocessing procedure on the game. Note that the preprocessing procedure may remove every vertex v that has $\mathrm{pri}(v) = \mathrm{MaxPri}(G)$, which could cause the player $d := \mathrm{MaxPri}(G) \bmod 2$ to change. If this were to occur, then we would have to run preprocessing for the other player to ensure that Proposition 10 can be applied. Therefore, the algorithm repeats the preprocessing procedure until the player d does not change. Once preprocessing is complete, the algorithm runs `New-McNaughton` on the remaining game.

Algorithm 2 `Preprocess`$(G, (T, X), i)$

Input: A parity game $G = (V, V_0, V_1, E, \mathrm{pri})$, a tree decomposition $(T = (I, F), X)$, and the index of a separator X_i
Output: Every player-$(p \bmod 2)$ dominion in G that is disjoint from X_i

```
d := p mod 2
for all (i, j) ∈ F do
    (W_0^{G_j}, W_1^{G_j}) := Solve(G_j, (T, X), Split(G_j, (T, X)))
end for
W_{1−d} := Attr^{1−d}(⋃_{(i,j)∈F} W_{1−d}^{G_j})
return W_{1−d}
```

Algorithm 3 `New-McNaughton`$(G, (T, X), i)$

Input: A parity game $G = (V, V_0, V_1, E, \mathrm{pri})$, a tree decomposition $(T = (I, F), X)$, and the index of a separator X_i
Output: The partition of winning sets (W_0, W_1)

```
p := MaxPri(G); d := p mod 2
A := Attr^d({v ∈ V : pri(v) = p})
if A = V then
    (W_d, W_{1−d}) := (V, ∅)
    return (W_0, W_1)
end if
(W'_0, W'_1) := Solve(G ↾ (V \ A), (T, X), i) {Call Solve instead of McNaughton.}
if W'_{1−d} = ∅ then
    (W_d, W_{1−d}) := (V, ∅)
else
    D := Attr^{1−d}(W'_{1−d})
    (W''_0, W''_1) := Solve(G ↾ (V \ D), (T, X), i) {Call Solve instead of McNaughton.}
    (W_d, W_{1−d}) := (W''_d, W''_{1−d} ∪ D)
end if
return (W_0, W_1)
```

Algorithm 4 `Solve`$(G, (T, X), i)$

Input: A parity game $G = (V, V_0, V_1, E, \text{pri})$, a tree decomposition $(T = (I, F), X)$ and the index of a separator X_i
Output: The partition of winning sets (W_0, W_1)

```
if |V| < 12k then
  (W_0, W_1) := McNaughton(G)
else if X_i = ∅ then
  for all (i, j) ∈ F do
    (W_0^{G_j}, W_1^{G_j}) := Solve(G ↾ V_{(i,j)}, (T, X), Split(G ↾ V_{(i,j)}, (T, X)))
  end for
  W_0 := ⋃_{(i,j)∈F} W_0^{G_j}; W_1 := ⋃_{(i,j)∈F} W_1^{G_j}
else
  (W_0, W_1) := (∅, ∅)
  repeat
    p := MaxPri(G); d := p mod 2
    W_{1-d} := W_{1-d} ∪ Preprocess(G, (T, X), i)
    V := V \ W_{1-d}
  until MaxPri(G) mod 2 = d
  (W'_0, W'_1) := New-McNaughton(G, (T, X), i)
  return (W_0 ∪ W'_0, W_1 ∪ W'_1)
end if
return (W_0, W_1)
```

We will use the notation $T(n, l)$ to denote the running time of Algorithm 4 when the input graph G has n vertices, and when $|X_i| = l$. Since the best bound for $|X_i|$ that we have is $|X_i| \leq k$, the running time of our algorithm is given by $T(n, k)$, where k is the width of the tree decomposition. However, we must still consider $T(n, l)$ for $l < k$, because our algorithm reduces the size of $|X_i|$ as it progresses. We have the following recurrence for $T(n, l)$:

$$T(n,l) \leq \begin{cases} n \cdot T(2n/3, k) & \text{if } l = 0, \\ 2^{12k} & \text{if } n \leq 12k, \\ n^2 \cdot T(2n/3 + k, k) + T(n-1, l) + T(n-1, l-1) + n^2 & \text{otherwise.} \end{cases}$$

The first case follows from the fact that the algorithm solves each piece of the game independently when the separator X_i is empty. Proposition 7 implies that each of these pieces can have size at most $2n/3$, and no separator in the piece can contain more than k vertices. The second case follows from the use of McNaughton's algorithm to solve games that have fewer than $12k$ vertices. The running time of McNaughton's algorithm is given by Theorem 2.

The final case of the recurrence deals with the case where X_i is non-empty, and where there are a large number of vertices. The term $n^2 \cdot T(2n/3+k, k)$ represents the cost of preprocessing. The algorithm `Solve`, can invoke `Preprocess` at most $|V|$ times before calling `New-McNaughton`. This is because if MaxPri(G) mod $2 \neq d$ then at least one vertex must have been removed by the previous call to `Preprocess`. The algorithm `Preprocess` itself solves at most n preprocessing

games, each of which has at most $2n/3 + k$ vertices and tree-width at most k. The term $T(n-1, l)$ comes from the first recursive call of `New-McNaughton`, where the removal of an attractor is guaranteed to reduce the number of vertices by 1. The term $T(n-1, l-1)$ is for the second recursive call of `New-McNaughton`, where Proposition 10 guarantees that at least one vertex has been removed from X_i. Finally, in each application of `New-McNaughton` must compute at least one attractor, and an attractor can be computed in $O(n^2)$ time. Solving the recurrance yields the following theorem.

Theorem 11. *When given a parity game with a tree decomposition of width k, Algorithm 4 will compute the partition into winning sets in time $n^{O(k \log n)}$ and $O(n^2)$ space.*

6 Computing The Tree Decomposition

It has been shown that the problem of deciding whether a graph has tree-width smaller than some bound k is NP-hard [2], which poses a significant problem for our algorithm, since it requires a tree decomposition as an input. Bodlaender has given an exact algorithm for computing tree decompositions [3]: If the tree-width of the graph is bounded by some constant k, then his algorithm will run in time $O(n \cdot f(k))$. However, the constant hidden by the $O(\cdot)$ notation is in the order 2^{k^3}. Thus, the cost of computing a tree decomposition using this algorithm dwarfs any potential advantage that our algorithm can offer over Obdržálek's algorithm, which must also compute a tree decomposition for the input graph.

As a solution to this problem, we propose that an *approximate* tree decomposition should be computed instead. A paper by Amir gives a plethora of approximation algorithms for this purpose [1]. One such algorithm takes an arbitrary graph G and an integer k, and in $O(2^{3k} n^2 k^{3/2})$ time provides either a tree decomposition of width at most $4.5k$ for G, or reports that the tree-width of G is larger than k.

Therefore, we can take the following approach to computing our tree decomposition. Apply Amir's algorithm with the input k set to 1. If a tree decomposition is found then halt, otherwise increase k by 1 and repeat. Once $k > \sqrt{n}/(4.5 \log n \cdot c)$, where c is the constant hidden by the $O(\cdot)$ notation in Theorem 11, we stop attempting to find a tree decomposition and apply the algorithm of Jurdziński, Paterson, and Zwick. If the procedure produces a tree decomposition for some k, then we apply either Obdržálek's algorithm or our algorithm, depending on how large k is.[3]

If the procedure halts at the value k, either because a tree decomposition has been found or because k is to large, then the total amount of time spent computing the tree decomposition is $\sum_{i=1}^{k} O(2^{3k} n^2 k^{3/2}) \in O(2^{3(k+1)} n^2 k^3)$. Thus, if the

[3] A more efficient technique would be to double k in each iteration and, once a tree decomposition is found, to use binary search to find the smallest value of k for which Amir's algorithm produces a tree decomposition. However, using this approach does not lead to better asymptotic running times for the algorithms.

input graph has treewidth k and Obdržálek's algorithm is applied, then the running time will be $O(2^{3(k+1)}n^2k^3)+n^{O((4.5k)^2)} \in n^{O(k^2)}$. If our algorithm is applied then the total running time will be $O(2^{3(k+1)}n^2k^3) + n^{O(4.5k \log n)} \in n^{O(k \log n)}$. Since the final value $k < \sqrt{n}$, if the algorithm of Jurdziński, Paterson, and Zwick is applied then the total running time will be $O(2^{3(k+1)}n^2k^3)+n^{O(\sqrt{n})} \in n^{O(\sqrt{n})}$.

References

1. E. Amir. Efficient approximation for triangulation of minimum treewidth. In *Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence*, pages 7–15, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
2. S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8(2):277–284, April 1987.
3. H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, December 1996.
4. H. L. Bodlaender. Treewidth: Algorithmic techniques and results. In *Mathematical Foundations of Computer Science 1997*, volume 1295 of *LNCS*, pages 19–36. Springer, 1997.
5. R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.
6. E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 368–377, Washington, DC, USA, 1991. IEEE Computer Society Press.
7. E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of μ-calculus. In *Computer Aided Verification, 5th International Conference, CAV'93*, volume 697 of *LNCS*, pages 385–396. Springer-Verlag, 1993.
8. E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games. A Guide to Current Research*, volume 2500 of *LNCS*. Springer, 2002.
9. M. Jurdziński. Deciding the winner in parity games is in UP ∩ co-UP. *Information Processing Letters*, 68(3):119–124, 1998.
10. M. Jurdziński, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 117–123. ACM/SIAM, 2006.
11. R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.
12. A. W. Mostowski. Games with forbidden positions. Technical Report 78, University of Gdańsk, 1991.
13. J. Obdržálek. Fast mu-calculus model checking when tree-width is bounded. In *CAV'03*, volume 2725 of *LNCS*, pages 80–92. Springer, 2003.
14. N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, September 1986.
15. S. Schewe. Solving parity games in big steps. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, volume 4855 of *LNCS*, pages 449–460. Springer, 2007.
16. C. Stirling. Local model checking games (Extended abstract). In *CONCUR'95: Concurrency Theory, 6th International Conference*, volume 962 of *LNCS*, pages 1–11. Springer-Verlag, 1995.
17. W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200:135–183, 1998.

A Proof of Theorem 11

Proof. We begin by arguing that the algorithm is correct. The correctness of removing the sets found by the preprocessing procedure follows from Proposition 9. Otherwise, the algorithm performs the same steps as McNaughton's algorithm, and therefore correctness follows from Theorem 2.

We now turn our attention to the running time of the algorithm. We begin with the recurrence $T(n,l) \leq n \cdot T(2n/3+k,k)+T(n-1,l)+T(n-1,l-1)+n^2$. We obtain the following by repeated substitution of $T(n-1,l)$:

$$\begin{aligned} T(n,l) &\leq n^2 \cdot T(2n/3+k,k) + T(n-1,l) + T(n-1,l-1) + n^2 \\ &\leq \sum_{i=0}^{n-12k} (n^2 \cdot T(2(n-i)/3+k,k) + T(n-i-1,l-1) + n^2) + 2^{12k} \\ &\leq n^3 \cdot T(2n/3+k,k) + n \cdot T(n-1,l-1) + n^3 + 2^{12k} \end{aligned}$$

Then, repeated substitution of $T(n-1,l-1)$ yields:

$$\begin{aligned} T(n,l) &\leq n^3 \cdot T(2n/3+k,k) + n \cdot T(n-1,l-1) + n^3 + 2^{12k} \\ &\leq \sum_{i=0}^{l-1} (n^{3+i} \cdot T(2(n-i)/3+k,k) + n^i(n^3+2^{12k})) + n^{l+1} \cdot T(2n/3,k) \\ &\leq n^{l+3} \cdot T(2n/3+k,k) + n^l(n^3+2^{12k}) + n^{l+1} \cdot T(2n/3,k) \\ &\leq n^{l+4} \cdot T(2n/3+k,k) + n^{l+3} + n^l \cdot 2^{12k} \end{aligned}$$

Since $n \geq 12k$, we have $2n/3+k \leq 3n/4$, and therefore the term $T(2n/3+k,k)$ is smaller than or equal to $T(3n/4,k)$. We can now perform repeated substitution on this term, to obtain the following expression for $T(n,k)$.

$$\begin{aligned} T(n,k) &\leq n^{k+4} \cdot T(3n/4,k) + n^{k+3} + n^k \cdot 2^{12k} \\ &\leq n^{(k+4)\cdot \log_{4/3}(n)} \cdot 2^{12k} + \log_{4/3}(n) \cdot (n^{k+3} + n^k \cdot 2^{12k}) \end{aligned}$$

Therefore, we have $T(n,k) \in n^{O(k \log n)}$.

We now argue that the algorithm uses a polynomial amount of space. It can easily be verified that the operations performed by each instance of `Preprocess`, `New-McNaughton`, and `Solve`, such as computing an attractor, can be performed in $O(n)$ space. Note that the maximum possible recursion depth of our algorithm is $2 \cdot n$. This is because each call to `Preprocess` made by `Solve` is on a game with at least one vertex removed, and the two calls to `Solve` made by `New-McNaughton` are made on games with at least one vertex removed. Since each instance of the algorithm uses at most $O(n)$ space, the total amount of space used at each point can be at most $2 \cdot n \cdot O(n) \in O(n^2)$.