TIME AND PARALLELIZABILITY RESULTS FOR PARITY GAMES WITH BOUNDED TREE AND DAG WIDTH

JOHN FEARNLEY AND SVEN SCHEWE

Department of Computer Science, University of Liverpool, Liverpool, UK
e-mail address: {john.fearnley, sven.schewe}@liverpool.ac.uk

ABSTRACT. Parity games are a much researched class of games in NP ∩ CoNP that are not known to be in P. Consequently, researchers have considered specialised algorithms for the case where certain graph parameters are small. In this paper, we study parity games on graphs with bounded treewidth, and graphs with bounded DAG width. We show that parity games with bounded DAG width can be solved in $O(n^{k+3} \cdot k^{k+2} \cdot (d+1)^{3k+2})$ time, where n, k, and d are the size, treewidth, and number of priorities in the parity game. This is an improvement over the previous best algorithm, given by Berwanger et al., which runs in $n^{O(k^2)}$ time. We also show that, if a tree decomposition is provided, then parity games with bounded treewidth can be solved in $O(n \cdot k^{k+5} \cdot (d+1)^{3k+5})$ time. This improves over previous best algorithm, given by Obdržálek, which runs in $O(n \cdot d^{2(k+1)^2})$ time. Our techniques can also be adapted to show that the problem of solving parity games with bounded treewidth lies in the complexity class NC^2, which is the class of problems that can be efficiently parallelized. This is in stark contrast to the general parity game problem, which is known to be P-hard, and thus unlikely to be contained in NC.

1. INTRODUCTION

A parity game is a two player game that is played on a finite directed graph. The problem of solving a parity game is known to lie in NP ∩ CoNP [McN93], and a sub-exponential time algorithm is known for the problem [JPZ08]. However, despite much effort, this problem is not known to be in P. Due to the apparent difficulty of solving parity games, recent work has considered special cases, where the input is restricted in some way. In particular, people have studied parity games where the input graph is restricted by a graph parameter. For example, parity games have been shown to admit polynomial time algorithms whenever the input graph has bounded treewidth [Obd03], DAG width [BDHK06], clique width [Obd07], Kelly width [HK08], or entanglement [BG04]. In this paper we study the parity game problem for graphs of bounded treewidth and graphs of bounded DAG width.

Parity games are motivated by applications in model checking. The problem of solving a parity game is polynomial-time equivalent to the modal μ-calculus model checking problem [EJS93, Sti95]. In model checking, we typically want to check whether a large

system satisfies a much smaller formula. It has been shown that many practical systems have bounded treewidth. For example, it has been shown that the control flow graphs of goto free Pascal programs have treewidth at most 3, and that the control flow graphs of goto free C programs have treewidth at most 6 [Tho98]. The same paper also shows that tree decompositions, which are costly to compute in general, can be generated in linear time with small constants for these control flow graphs. Moreover, Obdržálek has shown that, if the input system has treewidth k, and if the μ-calculus formula has m sub-formulas, then the modal μ-calculus model checking problem can be solved by determining the winner of a parity game that has treewidth at most $k \cdot m$ [Obd03]. Since m is usually much smaller than the size of the system, we have a strong motivation for solving parity games with bounded treewidth.

Parity games with bounded treewidth were first studied by Obdržálek [Obd03]. He gave an algorithm that runs in $O(n \cdot k^2 \cdot d^{2(k+1)^2})$ time, while using $d^{O(k^2)}$ space, where d is the number of priorities in the parity game, and k is the treewidth of the game. This result shows that, if the treewidth of a parity game is bounded, then there is a polynomial time algorithm for solving the game. However, since the degree of this polynomial depends on k^2, this algorithm is not at all practical. For example, if we wanted to solve the model checking problem for the control flow graph of a C program with treewidth 6, and a μ-calculus formula with a single sub-formula, the running time of the algorithm will already be $O(n \cdot d^{98})$.

DAG width is a generalisation of treewidth to directed graphs. It was defined by Berwanger et al. [BDHK06, BDH$^+$12], and independently by Obdržálek [Obd06]. Since parity games are played on directed graphs, it is natural to ask whether Obdržálek's algorithm can be generalised to parity games with bounded DAG width. Berwanger et al. [BDHK06, BDH$^+$12] showed that the algorithm can indeed be generalised. Their work is a generalisation of Obdržálek's techniques to parity games with bounded DAG width, but since every tree decomposition can also be viewed as a DAG decomposition, their algorithm can also be applied to graphs with bounded treewidth. In comparison to Obdržálek's original algorithm, they achieve an improved space complexity of $d^{O(k)}$. However, some operations of their algorithm still take $d^{O(k^2)}$ time, and thus their algorithm is no faster than the original.

More recently, an algorithm has been proposed for parity games with "medium" tree width [FL11]. This algorithm runs in time $n^{O(k \cdot \log n)}$, and it is therefore better than Obdržálek's algorithm whenever $k \in \omega(\log n)$. On the other hand, this algorithm does not provide an improvement for parity games with small treewidth.

1.1. **Our contribution.** In this paper, we take a step towards providing a practical algorithm for these problems. We show how parity games with bounded treewidth and DAG width can be solved solved in logarithmic space on an alternating Turing machine. We then use different variants of this technique to show three different results. Our first result is for parity games with bounded DAG width. We show that these games can be solved by an alternating Turing machine while using $(k+3) \cdot \log n + (k+2) \cdot \log(k) + (3k+2) \cdot \log(d+1)$ bits of storage. This then implies that there is a deterministic algorithm that runs in $O(n^{k+3} \cdot k^{k+2} \cdot (d+1)^{3k+2})$ time. Since the exponent of n is $k+3$, rather than k^2, this result improves over the $n^{O(k^2)}$ algorithm given by Berwanger et al.

We then turn our attention to parity games with bounded treewidth. Using similar techniques, we are able to obtain an algorithm that solves parity games with bounded treewidth in $O(n \cdot (k+1)^{k+5} \cdot (d+1)^{3k+5})$ time on a deterministic Turing machine. Again, since the exponent of d does not depend on k^2, this is an improvement over the original algorithm of Obdržálek, which runs in $O(n \cdot k^2 \cdot d^{2(k+1)^2})$ time.

For the treewidth result, we must pay attention to the complexity of computing a tree decomposition. In the DAG width result, this did not concern use, because a DAG decomposition can be computed in $O(n^{k+2})$ time, and this is therefore absorbed by the running time of the algorithm. In the treewidth case, however, things are more complicated. From a theoretical point of view, we could apply the algorithm of Bodlaender [Bod96], which runs in $O(n \cdot f(k))$ time. However, the function f lies in $2^{O(k^3)}$, and this has the potential to dwarf any improvement that our algorithm provides. As we have mentioned, in some practical cases, a tree decomposition can be computed cheaply. Otherwise, we suggest that the approximation algorithm of Amir [Ami01] should be used to find a 4.5-approximate tree decomposition. Our algorithm runs in $O(n \cdot k^{4.5k+5} \cdot (d+1)^{13.5k+5})$ time when a 4.5-approximate tree decomposition is used. Note that Obdržálek's algorithm also requires a tree decomposition as input. Thus, in this case, the gap between the two algorithms is even wider, as using a 4.5-approximate tree decomposition causes Obdržálek's algorithm to run in $O(n \cdot d^{40.5k^2+18k+2})$ time.

Finally, we are able to adapt these techniques to show a parallelizability result for parity games with bounded treewidth. We are able to provide an alternating Turing machine that solves the problem in $O(k^2 \cdot (\log n)^2)$ time while using $O(k \cdot \log n)$ space. This version of the algorithm does not require a precomputed tree decomposition. Hence, using standard results in complexity theory [ALR10], we have that the problem lies in the complexity class $\text{NC}^2 \subseteq \text{NC}$, which is the class of problems that can be efficiently parallelized.

This result can be seen in stark contrast to the complexity of parity games on general graphs: parity games are known to be P-hard by a reduction from reachability games, and P-hardness is considered to be strong evidence that an efficient parallel algorithm does not exist. Our result here shows that, while we may be unable to efficiently parallelize the μ-calculus model checking problem itself, we can expect to find efficient parallel algorithms for the model checking problems that appear in practice.

2. Preliminaries

2.1. **Parity games.** A parity game is a tuple $(V, V_0, V_1, E, \text{pri})$, where V is a set of vertices and E is a set of edges, which together form a finite directed graph. The sets V_0 and V_1 partition V into vertices belonging to player Even and player Odd, respectively. The function $\text{pri} : V \to D$ assigns a *priority* to each vertex from the set of priorities $D \subseteq \mathbb{N}$. It is required that the game does not contain any dead ends: for each vertex $v \in V$ there must exist an edge $(v, u) \in E$.

We define the *significance* ordering $\prec$ over D. This ordering represents how attractive each priority is to player Even. For two priorities $a, b \in \mathbb{N}$, we have $a \prec b$ if one of the following conditions holds: (1) a is odd and b is even, (2) a and b are both even and $a < b$, or (3) a and b are both odd and $a > b$. We say that $a \preceq b$ if either $a \prec b$ or $a = b$.

At the beginning of the game, a token is placed on a 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 if it is odd. 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 Σ_0 and Σ_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 $\mathrm{Play}(v_0, \sigma, \tau)$.

For each $\sigma \in \Sigma_0$, we define $G \upharpoonright \sigma$ to be the modification of G where Even is forced to play σ. That is, an edge $(v, u) \in E$ is included in $G \upharpoonright \sigma$ if either $v \in V_1$, or $v \in V_0$ and $\sigma(v) = u$. We define $G \upharpoonright \tau$ for all $\tau \in \Sigma_1$ analogously.

An infinite path $\langle v_0, v_1, \dots \rangle$ is said to be *consistent* with an Even strategy $\sigma \in \Sigma_0$ if $v_{i+1} = \sigma(v_i)$ for every i such that $v_i \in V_0$. If $\sigma \in \Sigma_0$ is a strategy for Even, and v_0 is a starting vertex, then we define $\mathrm{Paths}(v_0, \sigma)$ to give every path starting at v_0 that is consistent with σ. An Even strategy $\sigma \in \Sigma_0$ is called a *winning strategy* for a vertex $v_0 \in V$ if $\max(\mathrm{Inf}(\pi))$ is even for all $\pi \in \mathrm{Paths}_0(v_0, \sigma)$. The strategy σ is said to be winning for a set of vertices $W \subseteq V$ if it is winning for all $v \in W$. 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 2.1 ([EJ91, Mos91])**.** *In every parity game, the set of vertices V can be partitioned into* winning 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.*

In this paper we study the following computational problem for parity games: given a starting vertex s, determine whether $s \in W_0$ or $s \in W_1$.

2.2. **Treewidth.** Treewidth originated from the work of Robertson and Seymour [RS84]. Treewidth is a complexity measure for undirected graphs. Thus, to define the treewidth of a parity game, we will use the treewidth of the undirected graph that is obtained when the orientation of the edges is ignored. We begin by defining tree decompositions.

Definition 2.2 (Tree Decomposition)**.** For each game $G = (V, V_0, V_1, E, \mathrm{pri})$, the pair (T, X), where $T = (I, J)$ 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 $u \in X_i$.
(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 $\max\{|X_i| : i \in I\}$. The *treewidth* of a game G is the smallest width of a tree decomposition of G. Note that this is a slightly unusual definition, because the width of a tree decomposition is usually defined to be $\max\{|X_i| - 1 : i \in I\}$. However, we use our definition in order to keep the definitions the same between treewidth and DAG width.

Let $(T = (I, J), X)$ be a tree decomposition for a parity game $G = (V, V_0, V_1, E, \text{pri})$. Let $i \in I$ be a node in the tree decomposition, and let $v \in V$ be a vertex in the parity game with $v \notin X_i$. Let $k \in I$ be the closest node to i in T such that $x \in X_k$. Since T is a tree decomposition, the node k is well defined. Let $(i, j) \in J$ be the first edge on the path from i to k. We define $\text{Direction}(X_i, v)$ to be the function that outputs the node j.

There are multiple approaches for computing tree decompositions. From a theoretical point of view, the best known algorithm is the algorithm of Bodlaender [Bod96]. If the treewidth is bounded, then this is a linear time algorithm: it runs in $O(n \cdot f(k))$ time. However, the constant factor hidden by the function $f(k)$ is in the order of $2^{O(k^3)}$, which makes the algorithm impractical. For a more practical approach, we can apply the algorithm of Amir [Ami01] to approximate the treewidth of the graph. This algorithm takes a graph G and an integer k, and in $O(2^{3k} \cdot n^2 \cdot k^{3/2})$ time either finds a tree decomposition of width at most $4.5k$ for G, or reports that the tree-width of G is larger than k.

In this paper, we will assume that the size of the tree decomposition is linear in the size of the parity game. More precisely, we assume that, if $(T = (I, J), X)$ is a tree decomposition of a parity game $(V, V_0, V_1, E, \text{pri})$, then we have $|I| \leq |V|$. It has been shown that every tree decomposition can be modified, in polynomial time, to satisfy this property [Bod96, Lemma 2.2][1]. Therefore we can make this assumption without loss of generality.

2.3. **DAG width.** As opposed to treewidth, which is a measure for undirected graphs, DAG width [BDH$^+$12] is a measure for directed graphs. A directed graph $G = (V, E)$ is a DAG if it contains no directed cycles. If G is a DAG, then we define $\sqsubseteq_D$ to be the reflexive transitive closure of the edge relation of G. A *source* in the DAG is a vertex $v \in V$ that is minimal in the $\sqsubseteq_D$ ordering, and a *sink* in the DAG is a vertex that is maximal in the $\sqsubseteq_D$ ordering. Furthermore, given two sets $U, W \subseteq V$, we say that W *guards* U if, for every edge $(v, u) \in E$, where $v \in U$, we have $u \in W \cup U$. In other words, W guards U if the only way to leave U is to pass through a vertex in W. We can now define a DAG decomposition.

Definition 2.3 (DAG Decomposition). Let $G = (V, V_0, V_1, E, \text{pri})$ be a parity game. A DAG decomposition of G is a pair $(\mathcal{D} = (I, J), X)$, where $\mathcal{D}$ is a DAG, and $X = \{X_i : i \in I\}$ is a family of subsets of V, which satisfies the following conditions:

(1) $\bigcup_{i \in I} X_i = V$.
(2) For every edge $(i, j) \in J$, the set $X_i \cap X_j$ guards $(\bigcup_{j \sqsubseteq_D k} X_k) \setminus X_i$.
(3) For every $i, j, k \in I$, if $i \sqsubseteq_D k \sqsubseteq_D j$, then $X_i \cap X_j \subseteq X_k$.

The width of a DAG decomposition (D, X) is $\max\{|X_i| : i \in I\}$. The *DAG width* of a game G is the smallest width of a DAG decomposition of G. In accordance with the second condition in Definition 2.3, for each $X_i \in X$ we define:

$$\text{Guarded}(X_i) = (\bigcup_{(i,j) \in J} (\bigcup_{j \sqsubseteq_D k} X_k)) \setminus X_i.$$

---

[1]Specifically, we refer to Lemma 2.2 in the SIAM Journal on Computing version of this paper.

We can also define Direction for DAG decompositions. Suppose that $(\mathcal{D} = (I, J), X)$ is a DAG decomposition of a parity game $G = (V, V_0, V_1, E, \mathrm{pri})$. Let $i \in I$ be a node in the DAG decomposition, and let $v \in V$ be a vertex in the parity game with $v \in \mathrm{Guarded}(X_i)$. From the properties of a DAG decomposition, there must be at least one $j \in I$ such that $(i, j) \in J$ and either $v \in X_j$ or $v \in \mathrm{Guarded}(X_j)$. We define $\mathrm{Direction}(X_i, v)$ to arbitrarily select a node $j \in I$ that satisfies this property.

The only algorithm for computing DAG decompositions was given by Berwanger et. al. [BDH$^+$12]. They showed that, if a graph has DAG width k, then a DAG decomposition can be computed in $O(n^{k+2})$ time. In the case of DAG width, we cannot assume that the size of the DAG decomposition is linear in the size of the graph. In fact, the best known upper bound on the number of nodes in a DAG decomposition is n^{k+1}, and the number of edges is n^{k+2}, where n is the number of vertices in the graph, and k is the width of the DAG decomposition [BDH$^+$12, Proof of Theorem 16].

3. Strategy Profiles

In this section we define strategy profiles, which are data structures that allow a player to give a compact representation of the relevant properties of their strategy. Our algorithms in Sections 6, 7, and 8 will use strategy profiles to allow the players to declare their strategies in a small amount of space. Throughout this section we will assume that there is a *starting vertex* $s \in V$ and a set of *final vertices* $F \subseteq V$. Let $\sigma \in \Sigma_0$ be a strategy for Even. The strategy profile of σ describes the outcome of a modified parity game that starts at s, terminates whenever a vertex $u \in F$ is encountered, and in which Even is restricted to only play σ.

For each $u \in F$, we define $\mathrm{Paths}(\sigma, s, F, u)$ to be the set of paths from s to u that are consistent with σ and that do not visit a vertex in F. More formally, $\mathrm{Paths}(\sigma, s, F, u)$ contains every path of the form $\langle v_0, v_1, v_2, \dots v_k \rangle$ in $G \upharpoonright \sigma$, for which both of the following conditions hold:

- the vertex $v_0 = s$ and the vertex $v_k = u$, and
- for all i in the range $0 \leq i \leq k - 1$ we have $v_i \notin F$.

For each strategy $\tau \in \Sigma_1$, we define the functions $\mathrm{Paths}(\tau, s, F, u)$ analogously.

Recall that $\preceq$ is the significance ordering over priorities. For each $u \in F$, the function $\mathrm{Exit}(\sigma, s, F, u)$, gives the best possible priority, according to $\preceq$, that Odd can visit when Even plays σ and Odd chooses to move to u. This function either gives a priority $p \in D$, or, if Odd can never move to u when Even plays σ, the function gives a special symbol $-$, which stands for "unreachable". We will also define this function for Odd strategies $\tau \in \Sigma_1$. Formally, for every finite path $\pi = \langle v_1, v_2, \dots, v_k \rangle$, we define $\mathrm{MaxPri}(\pi) = \max\{\mathrm{pri}(v_i) : 1 \leq i \leq k\}$. Furthermore, we define, for $\sigma \in \Sigma_0$ and $\tau \in \Sigma_1$:

$$\mathrm{MinPath}(\sigma, s, F, u) = \min_{\preceq}\{\mathrm{MaxPri}(\pi) : \pi \in \mathrm{Paths}(\sigma, s, F, u)\},$$

$$\mathrm{MaxPath}(\tau, s, F, u) = \max_{\preceq}\{\mathrm{MaxPri}(\pi) : \pi \in \mathrm{Paths}(\tau, s, F, u)\}.$$

For every strategy $\chi \in \Sigma_0 \cup \Sigma_1$ and every $u \in F$ we define:

$$\mathrm{Exit}(\chi, s, F, u) = \begin{cases} - & \text{if } \mathrm{Paths}(\chi, s, F, u) = \emptyset, \\ \mathrm{MinPath}(\chi, s, F, u) & \text{if } \mathrm{Paths}(\chi, s, F, u) \neq \emptyset \text{ and } \chi \in \Sigma_0, \\ \mathrm{MaxPath}(\chi, s, F, u) & \text{if } \mathrm{Paths}(\chi, s, F, u) \neq \emptyset \text{ and } \chi \in \Sigma_1. \end{cases}$$

(A) The simulation game.
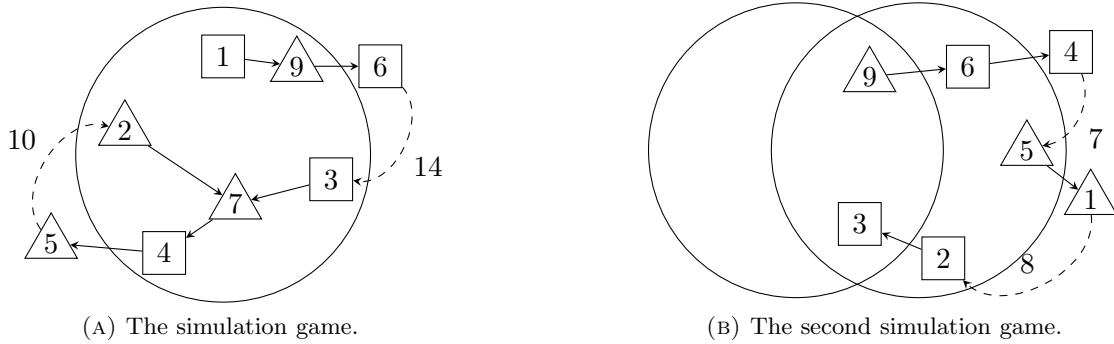
(B) The second simulation game.

FIGURE 1. Example runs of the simulation game.

We can now define the strategy profile for each strategy $\chi \in \Sigma_0 \cup \Sigma_1$. We define $\text{Profile}(\chi, s, F)$ to be a function $F \to D \cup \{-\}$ such that $\text{Profile}(\chi, s, F)(u) = \text{Exit}(\chi, s, F, u)$ for each $u \in F$.

4. Outline

In this section we give an outline of the rest of the paper. We start by describing simulated parity games, which are the foundation upon which all our results are based. Then we describe how this simulation game can be used to prove our three results.

4.1. **Simulated Parity Games.** Suppose that we have a DAG or tree decomposition of width k for our parity game. Suppose further that we want to determine the winner of some vertex $s \in V$ in the parity game. Our approach is to find some set of vertices $S \subseteq V$, where $S = X_i$ for some node i in our decomposition, such that $s \in S$. We then play a *simulation game* on S, which simulates the whole parity game using only the vertices in S.

The general idea behind the simulation game is shown in Figure 1A. The large circle depicts the decomposition node S, the boxes represent Even vertices, and the triangles represent Odd vertices. Since no two vertices share the same priority in this example, we will use the priorities to identify the vertices. As long as both players choose to remain in S, the parity game is played as normal. However, whenever one of the two players chooses to move to a vertex v with $v \notin S$ we simulate the parity game using a strategy profile: Even is required to declare a strategy in the form of a strategy profile P for v and S. Odd then picks some vertex $u \in S$, and moves there with priority $P(u)^2$. In the diagram, the dashed edges represent these simulated decisions. For example, when the play moved to the vertex 6, Even gave a strategy profile P with $P(3) = 14$, and Odd decided to move to the vertex 3. Together, the simulated and real edges will eventually form a cycle, and the winner of the game is the winner of this cycle. In our example, Even wins the game because the largest priority on the cycle is 10.

If Even always gives strategy profiles that correspond to some strategy $\sigma \in \Sigma_0$, then the outcome of our simulated game will match the outcome that would occur in the real parity game. On the other hand, it is possible that Even could lie by giving a strategy profile P

---

[2]The assignment of players here is arbitrary: none of our results would change if we had Odd propose a strategy profile, and Even pick a vertex in response.
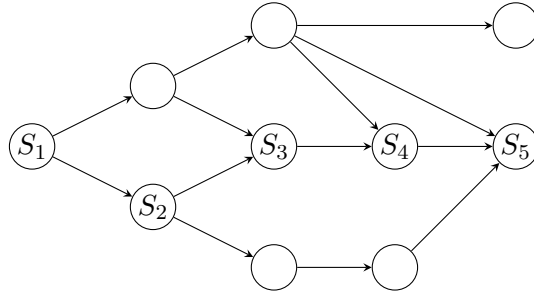
FIGURE 2. An example run of the simulation game for the time complexity result.

for which he has no strategy. To deal with this, Odd is allowed to reject P, which causes the two players to move to a second simulation game, where Even is required to prove that he does have a strategy for P.

Suppose that Odd rejected Even's first strategy profile in the game shown in Figure 1A. Hence, Even must show that his strategy profile P, that contains $P(3) = 14$, is correct. To do this, we select a second node $S' \subseteq V$ from the decomposition, and play a simulation game on S'. An example run of this game is shown in Figure 1B. The left circle represents S, and the right circle represents S'. The game proceeds as before, by simulating a parity game on S'. However, we add the additional constraint that, if a vertex $u \in S$ is visited, then the game ends: Even wins the game if the largest priority p seen on the path to u has $p \succeq P(u)$, and Odd wins otherwise. In this example, Even loses the game because the largest priority seen during the path is 8, and Even's strategy profile in Figure 1A claimed that the largest priority p should satisfy $p \succeq 14$. We also use rejections to deal with the case where Even's strategy profile never returns to S. If Even gives a strategy profile P with $P(u) = -$ for every $u \in S$, then Odd will reject P, and the game will continue on S' as before.

The simulation game will be formally defined in Section 5. We will then go on to prove three different results using three distinct versions of the simulation game. These versions differ in the way that the set S' is selected whenever Odd rejects Even's strategy profile.

4.2. **The Time Complexity Results.** In our first result, to be shown in Section 6, we will show that parity games with DAG width k can be solved in $O(|V|^{k+3} \cdot k^{k+2} \cdot (|D| + 1)^{3k+2})$ time. Figure 2 gives an example run of the simulation game that will be used in this result. The figure shows the DAG decomposition of the parity game, and each circle depicts one of the nodes in the decomposition. At the start of the game, we find a source node i of the DAG decomposition, and we play a simulation game on the set of vertices X_i. This node is shown as S_1 in the figure.

Recall, from Figure 1, that if the game on S_1 ends with Odd rejecting Even's strategy profile, then we must pick a new set of vertices, and play a second simulation game. The rule for picking this new set of vertices will make use of the DAG decomposition. Suppose that the game on S_1 ends when Odd rejects Even's strategy profile for v and S_1. The next set of vertices is chosen to be X_i, where $i = \mathrm{Direction}(S_1, v)$. This set is shown as S_2 in the diagram.

Thus, as we can see in the diagram, as Odd keeps rejecting Even's strategy profiles, we walk along a path in the DAG decomposition. This means that the game must eventually

end. To see this, note that when we play a simulation game on S_5, the properties of a DAG decomposition ensure that we either form a cycle in S_5, or that we visit some vertex in S_4, because S_4 is a guard of S_5. Recall that the simulation game ends whenever we move back to a set that we have already seen. Thus, when we play a simulation game on a sink in the DAG decomposition, there cannot be any simulated moves, and the game will either end in a cycle on the vertices in S_5, or when one of the two players moves to a vertex in S_4.

Our plan is to implement the simulation game on an alternating Turing machine. We will use the non-deterministic and universal states in the machine to implement the moves of the two players. Our goal is to show that this implementation uses at most $O(k \cdot \log |V|)$ space, which would then immediately imply our desired result.

There is one observation about DAG decompositions that is important for obtaining the $O(k \cdot \log |V|)$ alternating space bound. Recall from Figure 1B that if Odd rejects a strategy profile of Even, then we must remember the strategy profile so that we can decide the winner in subsequent simulation games. Our observation is that, if we are playing a simulation game using a DAG decomposition, then we only ever have to remember one previous strategy profile. For example, suppose that we are playing the simulation game on S_3 in Figure 2. Since $S_3 \subseteq \mathrm{Guarded}(S_2)$, we know that we cannot reach a vertex in S_1 without passing through a vertex in S_2. However, the simulation game on S_3 ends immediately when a in S_2 is visited. Hence, we can forget the strategy profile on S_1. This observation is crucial for showing the $O(k \cdot \log |V|)$ alternating space bound.

Our second result, which will be shown in Section 7, uses the same techniques, but applies them to parity games with bounded treewidth. We will show how the amount of space used by the alternating Turing machine can be significantly reduced for the treewidth case, and from this we derive a $O(|V| \cdot (k+1)^{k+5} \cdot (|D|+1)^{3k+5})$ time algorithm for parity games with bounded treewidth.

4.3. **The Parallelizability Result.** In our third result, which will be proved in Section 8, we show that the problem of solving parity games with bounded tree-width lies in the complexity class NC^2. To do this, we will construct a rather different version of the simulation game, which can be solved by an alternating Turing machine in $O(k^2 \cdot (\log |V|)^2)$ time and $O(k \cdot \log |V|)$ space. This then immediately implies that our problem lies in the class NC^2 [ALR10, Theorem 22.15].

In the first and second results, we were able to compute a DAG decomposition or tree decomposition, and then chose the sets S to be nodes in this decomposition. However, we do not take this approach here, because we do not know of a way to compute a tree decomposition in $O(k \cdot \log |V|)$ space on an alternating Turing machine. Instead, we will allow player Odd to chose these sets.

However, giving Odd this freedom comes at a price. In the first result, we only ever had to remember the last strategy profile that was rejected by Odd, and this was critical for showing the required space bounds for the alternating Turing machine. In this result, since Odd could potentially select any set, we may have to remember all of the previous strategy profiles to ensure that the simulation game eventually terminates. But, if we remember too many strategy profiles, then we will be unable to show the required space bound for the alternating Turing machine.

We resolve this by showing the following property: if the parity game has treewidth k, then Odd has a strategy for selecting the sets S such that:

(A) The strategy `Slice`.   (B) The strategy `Reduce`.

FIGURE 3. Odd's strategies for choosing S in the simulation game.

- The simulation game always terminates after $k \cdot \log |V|$ many rounds.
- We never have to remember more than 3 previous strategy profiles at the same time.

These two properties are sufficient to show that our alternating Turing machine meets the required time and space bounds.

We now outline the strategy for Odd that achieves these two properties. In fact, this strategy consists of two different strategies. The first strategy is called `Slice`, and is shown in Figure 3A. The figure shows the nodes in the tree decomposition, which means that each circle represents a set of vertices in the parity game. The nodes F_1 and F_2 represent two previous strategy profiles that we have remembered. The strategy `Slice` is required to select one of the nodes between F_1 and F_2. We do so using the following well known lemma about trees.

Lemma 4.1. *For every tree $T = (I, J)$ with $|I| \geq 3$, there is an $i \in I$ such that removing i from T splits T into parts, where each part has at most $\frac{2}{3}|I|$ vertices.*

We define the strategy `Slice` to select a separator that satisfies Lemma 4.1. For example, in Figure 3A the separator S' splits the tree into 3 parts, having 6, 1, and 1 vertices, respectively. Since there were originally 9 nodes between F_1 and F_2, we have that S' satisfies Lemma 4.1.

The second strategy is called `Reduce`, and it is shown in Figure 3B. It is used whenever we have remembered three strategy profiles. It selects the unique vertex that lies on the paths between them. It can be seen in Figure 3B that the set S' lies on the unique vertex that connects F_1, F_2, and F_3. The purpose of this strategy is to reduce the number of strategy profiles that we must remember. It can be seen that, no matter how the game on S' ends, we will be able to forget at least two of the three strategy profiles, while adding only one new strategy profile for S'.

Odd's overall strategy combines these two sub-strategies: we use `Slice` until three strategy profiles have been remembered, and then we switch between `Reduce` and `Slice`. Applying this strategy ensures that we use `Slice` at least half of the time, and so the game must end after at most $O(\log |V|)$ simulation games have been played. The use of `Reduce` ensures that we never have to remember more than three strategy profiles.

Once these properties have been established, it is then fairly straightforward to show that the game can be implemented by an alternating Turing machine in $O(k^2 \cdot (\log |V|)^2)$ time and $O(k \cdot \log |V|)$ space, which then immediately gives containment in NC^2.

5. Simulated Parity Games

In this section we describe the simulation game, which will be used in all of our subsequent results. We begin by formally defining the simulation game. Since each of our results requires a slightly different version of the simulation game, our definitions will be parameterized by two functions: Next and Hist, and each of our results will provide their own versions of these functions. In the second part of this section, we will show that, no matter which functions are chosen, if the simulation game terminates, then it determines the correct winner. Then, for each of our results, we provide a proof of termination for the relevant Next and Hist functions.

We begin by defining *records*, which allow us to remember the outcome of previous games. A record is a triple (F, p, P), where $F \subseteq V$ is a set of vertices, P is a strategy profile for F, and $p \in D$ is the largest priority that has been seen since P was rejected. When a record is created, we will use $(F, -, P)$ to indicate that no priority has been seen since P was rejected. Given a record (F, p, P), and a priority p', we define $\mathsf{Update}((F, p, P), p') = (F, \max(p, p'), P)$, where we have $\max(-, p') = p'$. A *history* is an ordered sequence of records. Given a history $\mathcal{F}$ and a priority p', we define: $\mathsf{Update}(\mathcal{F}, p') = \{\mathsf{Update}((F, p, P), p') : (F, p, P) \in \mathcal{F}\}$.

We can now define the format for the two functions Next and Hist. The function $\mathsf{Next}(S, v, \mathcal{F})$ takes a set of vertices $S \subseteq V$, a vertex $v \in V$, and a history $\mathcal{F}$. It is required to return a set of vertices $S' \subseteq V$. The history function $\mathsf{Hist}(\mathcal{F})$ allows us to delete certain records from the history $\mathcal{F}$. More formally, the function is required to return a history $\mathcal{F}'$ with $\mathcal{F}' \subseteq \mathcal{F}$.

We can now formally define the simulation game. Let G be a parity game. Given $S \subseteq V$, $s \in V$, a history $\mathcal{F}$, and two functions Next and Hist, we define the game $\mathsf{Simulate}_G(S, \mathcal{F}, s, \mathsf{Next}, \mathsf{Hist})$ as follows. The game maintains a variable c to store the current vertex. It also maintains a sequence of triples Π, where each entry is of the form (v, p, u) with $v, u \in V$ and $p \in D$. The sequence Π represents the simulated path that the two players form during the game.

To define our game, we will require notation to handle the simulated path Π. We extend MaxPri for paths of the form $\Pi = \langle (v_1, p_1, v_2), (v_2, p_2, v_3), \ldots, (v_{j-1}, p_{j-1}, v_j) \rangle$ by defining $\mathrm{MaxPri}(\Pi) = \max\{p_i : 1 \le i \le j - 1\}$. Furthermore, if a path Π consists of an initial path $\langle (v_1, p_1, v_2), \ldots, (v_{j-1}, p_{j-1}, v_j) \rangle$ followed by a cycle $\langle (v_j, p_j, v_{j+1}), \ldots, (v_{c-1}, p_{c-1}, v_j) \rangle$, then we define $\mathrm{Winner}(\Pi)$ to be Even if $\max(\{p_i : j \le i \le c - 1\})$ is even, and Odd otherwise.

We are now able to define $\mathsf{Simulate}_G(S, \mathcal{F}, s, \mathsf{Next}, \mathsf{Hist})$. The game is played in rounds. The first round of the game is slightly different, because it requires a special initialization procedure that will be introduced later. Every other round proceeds as follows.

(1) An edge (c, v) is selected by Even if $c \in V_0$ or by Odd if $c \in V_1$.
(2) If $v \in S$ or if $v \in F$ for some $(F, p, P) \in \mathcal{F}$, then the tuple $(c, \mathrm{pri}(v), v)$ is added to Π, the vertex c is set to v, and the game moves to Step 5.
(3) Even gives a strategy profile P' for the vertex v and the set S.
(4) Odd can either play `accept` for some vertex $u \in S$ with $P'(u) \neq -$, or play `reject`.
   - If Odd plays `accept`, then $(c, \max(\mathrm{pri}(v), P'(u)), u)$ is appended to Π, and c is set to u.
   - If Odd plays `reject`, then:
     - The history $\mathcal{F}'$ is obtained by computing $\mathsf{Update}(\mathcal{F}, \mathrm{MaxPri}(\Pi))$.
     - The history $\mathcal{F}''$ is obtained by adding $(S, -, P')$ to the end of $\mathcal{F}'$.

- The history $\mathcal{F}'''$ is obtained by computing $\mathsf{Hist}(\mathcal{F}'')$.
- The set S' is obtained by computing $\mathsf{Next}(S, v, \mathcal{F}''')$.
- The winner of the game is the winner of $\mathsf{Simulate}_G(S', \mathcal{F}''', v, \mathsf{Next}, \mathsf{Hist})$.

(5) If $c \in F$ for some $(F, p, P) \in \mathcal{F}$, then the game stops. Let (F, p, P) be the final record in $\mathcal{F}$ such that $c \in F$. If $P(c) = -$ then Odd wins the game. Otherwise, let $p' = \max(\mathrm{MaxPri}(\Pi), p)$. Even wins the game if $p' \succeq P(c)$ and Odd wins if $p' \prec P(c)$.

(6) If Π ends in a cycle, then the winner of the game is $\mathrm{Winner}(\Pi)$.

Note that, whenever we move to a new simulation game in Step 4, the variables c and Π are reset to their initial values. We will omit the subscripted parity game G from $\mathsf{Simulate}_G(S, \mathcal{F}, s, \mathsf{Next}, \mathsf{Hist})$ when it is clear from the context.

As we have mentioned, the first round is slightly different. This is because we allow the starting vertex s to be any vertex in V. Thus, we need a procedure to initialize the variable c. If we happen to have $s \in S$, then we can set $c = s$, and start the game in Step 1 as normal. If $s \notin S$, then we start the game in Step 3 with $v = s$. In other words, the game begins by allowing Even to give a strategy profile for v and S. It can be seen that, if this profile is accepted by Odd, then c is set to some vertex $u \in S$. Thus, after this initialization procedure, the game can continue on as normal.

5.1. **Strategies for the simulation game.** Let $\sigma \in \Sigma_0$ be an Even strategy for the original parity game. We define a strategy for Even in $\mathsf{Simulate}$ called $\mathtt{follow}_0(\sigma)$, which follows the moves made by σ. More formally, $\mathtt{follow}_0(\sigma)$ does the following:

- If Even is required to select an edge in Step 1, then the edge $(c, \sigma(c))$ is selected.
- In Step 3, the strategy selects $P' = \mathrm{Profile}(\sigma, v, S)$.

On the other hand, let $\tau \in \Sigma_1$ be a strategy for Odd in the original parity game. We define a strategy for Odd in $\mathsf{Simulate}$ called $\mathtt{follow}_1(\tau)$, which is analogous to the strategy $\mathtt{follow}_0(\sigma)$ that we have just defined. Odd's strategy is more complex, because it must decide whether the strategy profile proposed by Even in Step 3 should be accepted. To aid in this, we use the following definition. Let $P_\tau = \mathrm{Profile}(\tau, s, S)$ for some $s \in V$ and $S \subseteq V$. If P is a strategy profile for S given by Even in Step 3, then τ *refutes* P if one of the following conditions holds.

- For all $u \in S$, we have $P(u) = -$.
- For all $u \in S$, we have either $P_\tau(u) \neq -$ and $P(u) = -$, or $P_\tau(u) \prec P(u)$.

The first condition is necessary to deal with cases where Even gives a strategy profile P with $P(u) = -$ for all $u \in S$. In this case, Odd can not play $\mathtt{accept}$ and therefore has no choice but to reject P in Step 4. The second condition detects whether Even gives a false strategy profile: the condition ensures that if Odd rejects P, and if in a subsequent simulation game we arrive back at a vertex $u \in S$, then Odd will win in Step 5.

The strategy $\mathtt{follow}_1(\tau)$ follows τ for the vertices in S, and whenever Even gives a strategy profile P, Odd plays $\mathtt{reject}$ only when τ refutes P. If τ does not refute P, then there must be at least one $u \in S$ such that $P_\tau(u) \succeq P(u)$. Odd selects one such u and plays $\mathtt{accept}$ for it. Formally, we define $\mathtt{follow}_1(\tau)$ as follows:

- If Odd selects an edge in Step 1, then Odd selects $(c, \tau(c))$.
- In Step 4, Odd's decision is based on the strategy profile P'. If τ refutes P', then odd plays $\mathtt{reject}$. Otherwise, Odd selects a vertex $u \in F$ such that $\mathrm{Profile}(\tau, v, S)(u) \succeq P'(u)$ and plays $\mathtt{accept}$ for u.

5.2. **Plays in the simulation game.** We now describe the outcome when the simulation game $\mathsf{Simulate}(S, \mathcal{F}, s, \mathsf{Next}, \mathsf{Hist})$ is played. The players begin by playing $G_1 = \mathsf{Simulate}(S, \mathcal{F}, s, \mathsf{Next}, \mathsf{Hist})$. This game either ends in Step 5, Step 6, or when Odd plays `reject` in Step 4 and causes the players to move to a new game G_2. The outcome can be described as a (potentially infinite) sequence of games:

$$\langle \mathcal{S}_1 = \mathsf{Simulate}(S_1, \mathcal{F}_1, s_1, \mathsf{Next}, \mathsf{Hist}), \mathcal{S}_2 = \mathsf{Simulate}(S_2, \mathcal{F}_2, s_2, \mathsf{Next}, \mathsf{Hist}), \dots \rangle.$$

We will call this a *play* of the simulation game.

We define some notation for working with simulation game plays. Each of the games $\mathcal{S}_i$ maintained a variable Π. At the start of the game $\mathcal{S}_i$, the variable Π was empty, and at the end of the game, Π contained a sequence of tuples, which represented a sequence of real and simulated moves in S_i. For each i, we define Π_i to be the state of the variable Π at the end of the game $\mathcal{S}_i$. Furthermore, for each i, we will represent Π_i as:

$$\langle (v_{i,1}, p_{i,1}, u_{i,1}), (v_{i,2}, p_{i,2}, u_{i,2}), \dots, (v_{i,|\Pi_i|}, p_{i,|\Pi_i|}, u_{i,|\Pi_i|}) \rangle.$$

We are interested in plays of the simulation game that arise when either Even plays $\mathtt{follow}_0(\sigma)$ for some strategy $\sigma \in \Sigma_0$, or Odd plays $\mathtt{follow}_1(\tau)$ for some strategy $\tau \in \Sigma_1$. For the sake of exposition, we focus on player Even, but we will prove our results for both players. We want to show that, if Even uses $\mathtt{follow}_0(\sigma)$, then we can translate the paths Π_i in the simulation game to paths in the original parity game. More precisely, for each tuple $(v_{i,j}, p_{i,j}, u_{i,j})$ in Π_i, we want to construct a path $\pi_{i,j}$ in $G \restriction \sigma$. For the portions of Π_i that correspond to real edges in the parity game, this is easy. Formally, if $(v_{i,j}, p_{i,j}, u_{i,j})$ was not added to Π_i by a simulated move, then we can define $\pi_{i,j} = \langle v_{i,j}, u_{i,j} \rangle$. This path is obviously a path in $G \restriction \sigma$. The following lemma shows that paths $\pi_{i,j}$ can be defined for the simulated moves in Π_i.

Lemma 5.1. *We have two properties:*
- *If there is a strategy $\sigma \in \Sigma_0$ such that:*

$$\mathrm{Profile}(\sigma, v_{i,j}, S_i)(u_{i,j}) = p_{i,j} \neq -,$$

for some i and j, then there exists a path $\pi_{i,j} \in \mathrm{Paths}(\sigma, v_{i,j}, S_i, u_{i,j})$ such that

$$\mathrm{MaxPri}(\pi_{i,j}) = p_{i,j}.$$

- *If there is a strategy $\tau \in \Sigma_1$ such that:*

$$\mathrm{Profile}(\tau, v_{i,j}, S_i)(u_{i,j}) \succeq p_{i,j} \neq -,$$

for some i and j, then there exists a path $\pi_{i,j} \in \mathrm{Paths}(\tau, v_{i,j}, S_i, u_{i,j})$ such that

$$\mathrm{MaxPri}(\pi_{i,j}) \succeq p_{i,j}.$$

Proof. The first part of this lemma follows directly from the definition of a strategy profile. Since we know that $\mathrm{Profile}(\sigma, v_{i,j}, S_i)(u_{i,j}) \neq -$, we have that $\mathrm{Paths}(\sigma, v_{i,j}, S_i, u_{i,j}) \neq \emptyset$. Furthermore, since $\mathrm{Profile}(\sigma, v_{i,j}, S_i)(u_{i,j})$ considers the minimum path according to the $\preceq$-ordering, there must exist a path $\pi_{i,j}$ in $\mathrm{Paths}(\sigma, v_{i,j}, S_i, u_{i,j})$ such that $\mathrm{MaxPri}(\pi_{i,j}) = p_{i,j}$.

The second part is very similar to the first. Since we know that $\mathrm{Profile}(\tau, v_{i,j}, S_i)(u_{i,j}) \neq -$, we have that $\mathrm{Paths}(\tau, v_{i,j}, S_i, u_{i,j}) \neq \emptyset$. Furthermore, since $\mathrm{Profile}(\tau, v_{i,j}, S_i)(u_{i,j})$ considers the maximum path according to the $\preceq$-ordering, there must exist a path $\pi_{i,j}$ in $\mathrm{Paths}(\tau, v_{i,j}, S_i, u_{i,j})$ such that $\mathrm{MaxPri}(\pi_{i,j}) = \mathrm{Profile}(\tau, v_{i,j}, S_i)(u_{i,j}) \succeq p_{i,j}$. $\square$

For the rest of this section, we will assume that either Even always plays $\texttt{follow}_0(\sigma)$ for some strategy $\sigma \in \Sigma_0$, or that Odd always plays $\texttt{follow}_1(\tau)$ for some strategy $\tau \in \Sigma_1$. We will prove properties about the paths $p_{i,j}$ that are obtained from Lemma 5.1.

Firstly, we note that if $\pi_{i,j}$ is concatenated with $\pi_{i,j+1}$, then the result is still a path in either $G \restriction \sigma$ or $G \restriction \tau$. This is because Steps 2 and 4 of the simulation game ensure that the final vertex in $\pi_{i,j}$ is the first vertex in $\pi_{i,j+1}$. Similarly, the concatenation of $\pi_{i,|\pi_i|}$ and $\pi_{i+1,1}$ always produces a path in either $G \restriction \sigma$, or $G \restriction \tau$, because Step 4 of the simulation game ensures that the final vertex in $\pi_{i,|\Pi_i|}$ must have an edge to the first vertex in $\pi_{i+1,1}$.

With these properties in mind, we define $\pi_{(i,j)\to(i',j')}$, for two pairs of integers where, either $i' > i$, or both $i = i'$ and $j' > j$, to be the concatenation of $\pi_{i,j}$ through $\pi_{i',j'}$. More formally, let $\circ$ be the operator that concatenates two paths. We define:

$$\pi_i = \pi_{i,1} \circ \pi_{i,2} \circ \cdots \circ \pi_{i,|\pi_i|},$$

to be the path from the first vertex in $\pi_{i,1}$ to the final vertex in $\pi_{i,|\pi_i|}$. We then define

$$\pi_{(i,j)\to(i',j')} = \pi_{i,j} \circ \cdots \circ \pi_{i,|\pi_i|} \circ \pi_{i+1} \circ \cdots \circ \pi_{i'-1} \circ \pi_{i',1} \circ \cdots \circ \pi_{i',j'},$$

to be the concatenation of the paths between $\pi_{i,j}$ and $\pi_{i',j'}$.

The following lemma regarding $\mathrm{MaxPri}(\pi_{(i,j)\to(i',j')})$ follows easily from Lemma 5.1.

Lemma 5.2. *If $\sigma \in \Sigma_0$ is an Even strategy, and $\langle G_1 \ldots G_k \rangle$ is obtained when Even plays $\texttt{follow}_0(\sigma)$, then we have:*

$$\mathrm{MaxPri}(\pi_{(i,j)\to(i',j')}) = \max\{p_{x,y} \; : \; i \leq x \leq i' \text{ and}$$
$$(x = i) \Rightarrow j \leq y \text{ and } (x = i') \Rightarrow y \leq j'\}.$$

On the other hand, if $\tau \in \Sigma_1$ is an Odd strategy, and $\langle G_1 \ldots G_k \rangle$ is obtained when Odd plays $\texttt{follow}_1(\tau)$, then we have:

$$\mathrm{MaxPri}(\pi_{(i,j)\to(i',j')}) \succeq \max\{p_{x,y} \; : \; i \leq x \leq i' \text{ and}$$
$$(x = i) \Rightarrow j \leq y \text{ and } (x = i') \Rightarrow y \leq j'\}.$$

5.3. **Winning strategies and the simulation game.** In this section, we show that if one of the two players follows a winning strategy for the original parity game, then that player cannot lose the simulation game. More formally, we show that, if $\sigma \in \Sigma_0$ is a winning strategy for Even for some vertex $s \in V$, and if Even plays $\texttt{follow}_0(\sigma)$, then Even cannot lose $\mathsf{Simulate}(S, \mathcal{F}, s, \mathsf{Next}, \mathsf{Hist})$, for all S, $\mathcal{F}$, Next, and Hist. We also show the analogous result for Odd.

We begin by showing that, if Even never lies about his strategy profile in Step 3, then Even can never lose the game in Step 5.

Lemma 5.3. *If Even plays $\texttt{follow}_0(\sigma)$ for some $\sigma \in \Sigma_0$, then Even can never lose the simulation game in Step 5.*

Proof. Assume, for the sake of contradiction, that Even loses the simulation game in Step 5 while playing $\texttt{follow}_0(\sigma)$. Let

$$\langle \mathcal{S}_1 = \mathsf{Simulate}(S_1, \mathcal{F}_1, s_1, \mathsf{Next}, \mathsf{Hist}), \ldots, \mathcal{S}_k = \mathsf{Simulate}(S_k, \mathcal{F}_k, s_k, \mathsf{Next}, \mathsf{Hist}) \rangle$$

be the play of the simulation game, where Even loses $\mathcal{S}_k$ in Step 5.

Let (F, p, P) be the record used to decide the winner in Step 5. The fact that Odd wins in Step 5 implies that Π_k ends at some vertex $u \in F$, and that either $P(u) = -$ or $\max(\mathrm{MaxPri}(\Pi), p) \prec P(u)$. Let i be the index of the game $\mathcal{S}_i$ in which $(F, -, P)$ is contained in $\mathcal{F}_{i+1} \setminus \mathcal{F}_i$.

We begin with the case where $P(u) = -$. In this case, the fact that $\mathtt{follow}_0(\sigma)$ only ever gives $\mathrm{Profile}(\sigma, s, S)$ for P' in Step 3 allows us to invoke Lemma 5.1 to conclude that the path $\pi_{(i+1,1) \to (k, |\Pi_k|)}$ is a path in $G \upharpoonright \sigma$. Note that the path starts at s_{i+1}, and that u is the first vertex in F that is visited by $\pi_{(i+1,1) \to (k, |\Pi_k|)}$. Hence we must have $\pi_{(i+1,1) \to (k, |\Pi_k|)} \in \mathrm{Paths}(\sigma, s_{i+1}, F, u)$. Therefore, the fact that the set $\mathrm{Paths}(\sigma, s_{i+1}, F, u)$ is non-empty contradicts the fact that $P(u) = \mathrm{Profile}(\sigma, s_{i+1}, F)(u) = -$.

We now consider the case where Odd wins in Step 5 because Π_k ends at some vertex $u \in F$ and $\max(\mathrm{MaxPri}(\Pi), p) \prec P(u)$. In this case we can again invoke Lemma 5.1 to argue that there is a path $\pi_{(i+1,1) \to (k, |\Pi_k|)}$ in $G \upharpoonright \sigma$, and we can use the same arguments as in the previous case to argue that this path is contained in $\mathrm{Paths}(\sigma, s_{i+1}, F_i, u)$. Moreover, Lemma 5.2 implies that

$$p' = \max(p, \mathrm{MaxPri}(\Pi)) = \mathrm{MaxPri}(\pi_{(i+1,1) \to (k, |\pi_k|)}).$$

Since $p' \prec P(u)$, and $P(u) = \mathrm{Profile}(\sigma, s_{i+1}, F)$, we have the following inequality:

$$\mathrm{MaxPri}(\pi_{(i+1,1) \to (k, |\Pi_k|)}) \prec \min_{\preceq} \{\mathrm{MaxPri}(\pi) \ : \ \pi \in \mathrm{Paths}(\sigma, s_{i+1}, F, u)\}.$$

Since $\pi_{(i+1,1) \to (k, |\Pi_k|)}$ is contained in $\mathrm{Paths}(\sigma, s_{i+1}, F, u)$ this inequality is impossible, which yields the required contradiction. $\qquad\square$

We now show a corresponding property for Odd.

Lemma 5.4. *If Odd plays* $\mathtt{follow}_1(\tau)$ *for some strategy* $\tau \in \Sigma_1$*, then Odd can never lose the simulation game in Step 5.*

Proof. Assume, for the sake of contradiction, that Odd loses the simulation game in Step 5 while playing $\mathtt{follow}_1(\tau)$. Let

$$\langle \mathcal{S}_1 = \mathsf{Simulate}(S_1, \mathcal{F}_1, s_1, \mathsf{Next}, \mathsf{Hist}), \ldots, \mathcal{S}_k = \mathsf{Simulate}(S_k, \mathcal{F}_k, s_k, \mathsf{Next}, \mathsf{Hist}) \rangle$$

be the play of the simulation game, where Odd loses $\mathcal{S}_k$ in Step 5.

Let (F, p, P) be the record that is used to determine the winner in Step 5. The fact that Even wins in Step 5 implies that Π_k ends at some vertex $u \in F$, and that $\max(\mathrm{MaxPri}(\Pi), p) \succeq P(u)$. Let i be the index of the game $\mathcal{S}_i$ in which $(F, -, P)$ is contained in $\mathcal{F}_{i+1} \setminus \mathcal{F}_i$. Since $\mathtt{follow}_1(\tau)$ only ever plays $\mathtt{reject}$ in the case where τ refutes P', we can apply Lemma 5.1 to obtain a path $\pi_{(i+1,1) \to (k, |\Pi_k|)}$ in $G \upharpoonright \tau$. Note that the path starts at the vertex s_{i+1}, and that u is the first vertex in F that is visited by $\pi_{(i+1,1) \to (k, |\Pi_k|)}$. Hence we must have $\pi_{(i+1,1) \to (k, |\Pi_k|)} \in \mathrm{Paths}(\tau, s_{i+1}, F, u)$. Furthermore, we can use Lemma 5.2 to conclude:

$$\mathrm{MaxPri}(\pi_{(i+1,1) \to (k, |\Pi_k|)}) \succeq \max(\mathrm{MaxPri}(\Pi), p) \succeq P(u).$$

Since τ refutes P, we must have $P(u) \succ \mathrm{Profile}(\tau, s_{i+1}, F)(u)$. Combining all of these facts yields:

$$\mathrm{MaxPri}(\pi_{(i+1,1) \to (k, |\Pi_k|)}) \succeq P(u)$$
$$\succ \mathrm{Profile}(\tau, s_{i+1}, F)(u)$$
$$= \max_{\preceq} \{\mathrm{MaxPri}(\pi) \ : \ \pi \in \mathrm{Paths}(\tau, s_{i+1}, F, u)\}.$$

However, since $\pi_{(i+1,1)\to(k,|\Pi_k|)} \in \mathrm{Paths}(\tau, s_{i+1}, F_i, u)$, this inequality is impossible. $\qquad\square$

We now turn our attention to Step 6 of the simulation game. The following pair of lemmas will be used to show that, if one of the two players follows a winning strategy, then that player cannot lose the simulation game in Step 6.

Lemma 5.5. *Let $\sigma \in \Sigma_0$ be an Even winning strategy for $s \in V$. If Even plays $\mathtt{follow}_0(\sigma)$, then Even cannot lose $\mathsf{Simulate}(S, \mathcal{F}, s, \mathsf{Next}, \mathsf{Hist})$ is Step 6, for all choices of S, $\mathcal{F}$, Next, and Hist.*

Proof. Assume, for the sake of contradiction, that Even loses the simulation game in Step 6 while playing $\mathtt{follow}_0(\sigma)$. Let

$$\langle \mathcal{S}_1 = \mathsf{Simulate}(S_1, \mathcal{F}_1, s_1, \mathsf{Next}, \mathsf{Hist}), \dots, \mathcal{S}_k = \mathsf{Simulate}(S_k, \mathcal{F}_k, s_k, \mathsf{Next}, \mathsf{Hist}) \rangle$$

be the play of the simulation game, where Even loses $\mathcal{S}_k$ in Step 6.

Since Even loses in Step 6, there must be some index i such that $\langle \Pi_{k,i}, \dots, \Pi_{k,|\Pi_k|} \rangle$ forms a cycle in S_k, and that the highest priority on this cycle is odd. Since Even plays $\mathtt{follow}_0(\sigma)$, we can use Lemma 5.1 to conclude that $\pi_{(k,i)\to(k,|\Pi_k|)}$ is a cycle in $G \upharpoonright \sigma$, and we can apply Lemma 5.2 to conclude that $\mathrm{MaxPri}(\pi_{(k,i)\to(k,|\Pi_k|)})$ is odd. Furthermore, we have that $\pi_{(1,1)\to k,(i-1)}$ is a path from s to the cycle $\pi_{(k,i)\to(k,|\Pi_k|)}$ in $G \upharpoonright \sigma$. This implies that σ is not a winning strategy for s, which provides the required contradiction. $\qquad\square$

Lemma 5.6. *Let $\tau \in \Sigma_1$ be an Odd winning strategy for $s \in V$. If Odd plays $\mathtt{follow}_1(\tau)$, then Odd cannot lose $\mathsf{Simulate}(S, \mathcal{F}, s, \mathsf{Next}, \mathsf{Hist})$ is Step 6, for all choices of S, $\mathcal{F}$, Next, and Hist.*

Proof. Assume, for the sake of contradiction, that Odd loses the simulation game in Step 6 while playing $\mathtt{follow}_1(\tau)$. Let

$$\langle \mathcal{S}_1 = \mathsf{Simulate}(S_1, \mathcal{F}_1, s_1, \mathsf{Next}, \mathsf{Hist}), \dots, \mathcal{S}_k = \mathsf{Simulate}(S_k, \mathcal{F}_k, s_k, \mathsf{Next}, \mathsf{Hist}) \rangle$$

be the play of the simulation game, where Odd loses $\mathcal{S}_k$ in Step 6.

Since Odd loses in Step 6, there must be some index i such that $\langle \Pi_{k,i}, \dots, \Pi_{k,|\Pi_k|} \rangle$ forms a cycle in S_k, and that that the highest priority on this cycle is even. Since Odd plays $\mathtt{follow}_1(\tau)$, we can use Lemma 5.1 to conclude that $\pi_{(k,i)\to(k,|\Pi_k|)}$ is a cycle in $G \upharpoonright \tau$, and we can apply Lemma 5.2 to conclude that $\mathrm{MaxPri}(\pi_{(k,i)\to(k,|\Pi_k|)}) \succeq \mathrm{MaxPri}(\Pi_k)$. Since $\mathrm{MaxPri}(\Pi_k)$ is even, every priority p with $p \succeq \mathrm{MaxPri}(\Pi_k)$ must also be even. Hence, we can conclude that $\mathrm{MaxPri}(\pi_{(k,i)\to(k,|\Pi_k|)})$ is even. Furthermore, we have that $\pi_{(1,1)\to(k,i-1)}$ is a path from s_1 to the cycle $\pi_{(k,i)\to k,(|\Pi_k|)}$ in $G \upharpoonright \tau$. This implies that τ is not a winning strategy for s, which provides the required contradiction. $\qquad\square$

When combined, Lemmas 5.3 through 5.6 imply the following property, which is the main result of this section.

Lemma 5.7. *If $\mathsf{Simulate}_G(S, \mathcal{F}, s, \mathsf{Next}, \mathsf{Hist})$ terminates, then it correctly determines the winner of s.*

6. Time Complexity Results for Parity Games

Let G be a parity game with a DAG decomposition $(\mathcal{D} = (I, J), X)$ of width k. In this section we give a version of the simulation game that can be solved in $(k + 3) \cdot \log |V| + (k + 2) \cdot \log(k) + (3k + 2) \cdot \log(|D| + 1)$ space on an alternating Turing machine, and hence $O(|V|^{k+3} \cdot k^{k+2} \cdot (|D| + 1)^{3k+2})$ time on a deterministic Turing machine.

6.1. **The functions** Next **and** Hist. Recall that the simulation game can be customized by specifying the Next and Hist functions. We begin by defining the versions of these functions that will be used for this result. We start by defining the Next function. Recall, from Section 4.2, that we begin by playing a simulation game on a source in the DAG decomposition. To define this formally, we must be more specific: if we want to determine the winner of some vertex $v \in V$, then we must pick a source i from the DAG decomposition such that $v \in \text{Guarded}(X_i) \cup X_i$, and play a simulation game on X_i.

The definition of Next follows the intuition that was outlined in Section 4.2. Suppose that we are required to compute $\text{Next}(S, v, \mathcal{F})$, for some set $S = X_i$ with $i \in I$, some $v \in V$, and some history $\mathcal{F}$. We find the DAG decomposition node $j \in I$ such that $(i, j) \in J$, and $v \in \text{Guarded}(X_j) \cup X_j$, and set $\text{Next}(S, v, \mathcal{F}) = X_j$. After defining Hist, we will prove Lemma 6.1, which shows that such a j always exists.

To define the Hist function, we use the observation that we made in Section 4.2: we only need to remember at most one record. More precisely, we only need to remember the record corresponding to the last simulation game that was played. Formally, we can define the Hist function as follows.

- The first time that $\text{Hist}(\mathcal{F})$ is called, $\mathcal{F}$ will contain only one record. We do not delete this record, so we set $\text{Hist}(\mathcal{F}) = \mathcal{F}$.
- Every other time that $\text{Hist}(\mathcal{F})$ is called, $\mathcal{F}$ will contain exactly two records. Due to the definition of Next, we know that the two records in $\mathcal{F}$ have the form (X_i, v_1, P_1) and (X_j, v_2, P_2) for some $i, j \in I$, some $v_1, v_2 \in V$, and some strategy profiles P_1 and P_2. Furthermore, we know that $(i, j) \in J$ is a directed edge in the DAG decomposition. We set $\text{Hist}(\mathcal{F}) = \{(X_j, v_2, P_2)\}$.

We now show the correctness of the Next function.

Lemma 6.1. *Let $S = X_i$ for some $i \in I$, and $s \in \text{Guarded}(X_i) \cup X_i$. Suppose that $\text{Simulate}(S, \emptyset, s, \text{Next}, \text{Hist})$ ends when Odd rejects a strategy profile for some vertex $v \in V$. There exists a $j \in I$ with $(i, j) \in J$ and $v \in \text{Guarded}(X_j) \cup X_j$.*

Proof. The definitions of Next and Hist imply that there are two cases to consider:

- The case where X_i is a sink, and $\mathcal{F}$ is empty.
- The case where X_i is not a sink, and $\mathcal{F} = \{(X_k, v', P')\}$, with $(k, i) \in J$.

In the first case, the property follows from the definition of a guarded set: no matter which edges (c, v) are chosen in Step 1, we can never move to a vertex $v \notin \text{Guarded}(X_i)$. In the second case, we can either choose an edge (c, v) with $v \in \text{Guarded}(X_i)$, or an edge (c, v) with $v \in X_k$. However, Step 2 ensures that, if $v \in X_k$, then the game skips directly to Step 5. Thus, in both cases, if Odd rejects a strategy profile for some vertex v, with $(c, v) \in E$, then we have $v \in \text{Guarded}(X_i)$.

To prove our claim, we can apply the properties given in Definition 2.3. They imply that, if there is an edge $(c, v) \in E$ with $c \in X_i$ and $v \in \text{Guarded}(X_i) \setminus X_i$, then there must exist an edge $(i, j) \in J$ with $v \in \text{Guarded}(X_j) \cup X_j$. $\square$

6.2. **Termination and correctness.** We now show that this version of the simulation game terminates, and that it can be used to correctly determine the winner of the parity game. Recall that, in Section 5.3, we showed that if the simulation game terminates, then it correctly determines the winner of the parity game. Thus, we must show that the simulation game terminates when it is equipped with Next and Hist.

Lemma 6.2. *If $S = X_i$ for some source $i \in I$, and $s \in \mathrm{Guarded}(X_i) \cup X_i$ is a starting vertex, then* $\mathsf{Simulate}(S, \emptyset, s, \mathsf{Next}, \mathsf{Hist})$ *terminates.*

Proof. We begin by showing that Odd cannot play `reject` in Step 4 infinitely many times. This follows from the definition of Next. This functions ensures two properties:

- Each simulation game is played on a set of vertices $S = X_j$ for some $j \in I$.
- If odd plays `reject` in a game on $S = X_j$, then the next game will be played on $S' = X_k$, with $(j, k) \in J$.

These two properties ensure that, every time that Odd plays `reject`, we take one step along a directed path in the DAG decomposition. Since the DAG decomposition contains no cycles, we know that Odd can play `reject` at most $|I|$ times before we reach a sink in the DAG decomposition.

The Hist function ensures that, once we reach a sink in the DAG decomposition, the game must end. Suppose that we have reached a simulation game on $S = X_j$ where $j \in I$ is a sink. The history function ensures that $\mathcal{F}$ contains a record (F, p, P) with $F = X_k$, where $(k, j) \in J$. Therefore, since X_j guards X_i, every edge (v, w) with $v \in S$ and $w \notin S$ has $w \in X_j$. Thus, Step 2 of the simulation game ensures that Odd cannot play `reject`.

We now show that if Odd only plays `accept` in a simulation game on S, then the game must terminate after $|S| + 1$ steps. This is because each time Odd plays `accept` for some vertex $u \in S$, the vertex u is appended to Π. Therefore, if the game continues for $|S| + 1$ steps, then Odd will be forced to play `accept` for some vertex u that has been visited by Π, and the game will terminate in Step 6. $\square$

Then, we can apply Lemma 6.2 along with Lemma 5.7 to imply the correctness of this version of the simulation game.

Lemma 6.3. *Let $v \in V$ be a vertex, and let $X_i \in X$ be a set of vertices, where i is a source, and $v \in \mathrm{Guarded}(X_i) \cup X_i$. Even wins* $\mathsf{Simulate}(X_i, \emptyset, v, \mathsf{Next}, \mathsf{Hist})$ *if and only if $v \in W_0$.*

6.3. **Implementation on an alternating Turing machine.** In this section, we show how this version of the simulation game can be implemented on an alternating Turing machine. We use the existential states of the machine to simulate the moves of Even and the universal states of the machine to simulate the moves of Odd. In this way, we ensure that the machine will have an accepting run if and only if Even has a strategy for $\mathsf{Simulate}_G(S, \emptyset, s, \mathsf{Next}, \mathsf{Hist})$ that forces a win no matter how Odd plays.

Theorem 6.4. *There is an algorithm that, when given a parity game G, a DAG decomposition $(\mathcal{D} = (I, J), X)$ of width k for G, and a vertex s, terminates in $O(|V|^{k+3} \cdot k^{k+2} \cdot (|D| + 1)^{3k+2})$ time and outputs the winner of s.*

Proof. We show that $\mathsf{Simulate}_G(S, \emptyset, s, \mathsf{Next}, \mathsf{Hist})$, where S is a source in the DAG decomposition, and $s \in \mathrm{Guarded}(S) \cup S$, can be implemented on an alternating Turing machine in logarithmic space. We split our analysis into two cases: the data that must be stored

throughout the game, and the data that must be stored to implement each iteration of the game.

We begin by considering the data that must be stored throughout. The simulation game maintains the variables S, $\mathcal{F}$, Π and c. Note that, from the definitions of Hist and Next, we have that $S = X_i$ for some $i \in I$. Moreover, we know that $\mathcal{F}$ contains exactly one record (F, p, P), where $F = X_j$ for some j with $(j, i) \in J$. These variables can be stored using the following amount of space.

- We claim that S and F can be remembered using $(k+2) \log |V|$ bits. This is because we always have $S = X_i$ and $F = X_j$ for some edge $(j, i) \in J$, and since since $|J| \leq |V|^{k+2}$, we need at most $(k+2) \log |V|$ bits to store (i, j). Remembering (i, j) is sufficient to identify both S and F.
- Note that c is always a vertex in S. Since we know that $S = X_i$, we can represent c as a number between 1 and k. Thus, c can be stored in at most $\log k$ bits.
- The priority p can be stored in $\log D$ bits.
- The strategy profile P contains $|F|$ mappings from the vertices of F to either a priority d, or $-$. Thus, we can represent a strategy profile as a list of length k, where each element of the list is a number between 1 and $|D| + 1$: the numbers 1 through $|D|$ encode their respective priorities, and the number $|D| + 1$ encodes $-$. Thus, P can be stored using $k \cdot \log(|D| + 1)$ bits.
- Recall that the sequence Π is reset to $\emptyset$ at the start of each simulation game. This implies that Π contains at most $|S| + 1$ tuples of the form (v, p, u). Due to the initialization procedure of the simulation game we can have $v \notin S$ and $u \in S$ in the first element of Π. However, for every subsequent element, we always have $v \in S$ and $u \in S$. Suppose that the first element of Π does have $v \notin S$. We claim that we do not need to remember the vertex v. Note that the simulation game only cares about the largest priority along Π, and whether Π forms a cycle. Since $v \notin S$, Step 2 prevents Π from forming a cycle that includes v. Thus, we do not need to store v.

  Note also that, if (v, p, u) appears in Π, then, the next element in the sequence must be of the form (u, p', u'). Thus, for every tuple in Π other than the first element in the sequence, we only need to store the priority p' and the vertex u'. In total, therefore, we can represent Π by storing at most $|S| + 1$ priorities, and at most $|S| + 1$ vertices in S. These can be stored in $(k+1) \cdot \log |D| + (k+1) \cdot \log k$ bits.

Next, we account for the space used during the execution of the game.

- The vertex v that is selected in Step 1 requires $\log |V|$ bits to store.
- As with the strategy profile P, the strategy profile P' given by Even in Step 3 can be stored in $k \cdot \log(|D| + 1)$ bits.
- We claim that no additional space is needed if Odd rejects the strategy profile P' in step 4. To see why, note that Update will always delete the current record in $\mathcal{F}$. Thus, we can implement Step 4 by first deleting the existing record in $\mathcal{F}$, and then reusing the space for the new record.
- No extra space is needed by Steps 2, 5, and 6.

Thus, in total, we require

$$(k+3) \cdot \log |V| + (k+2) \cdot \log(k) + (3k+2) \cdot \log(|D| + 1)$$

bits of storage for to implement the game on an alternating Turing machine. Using the standard determinization techniques for alternating Turing machines, we can obtain a deterministic algorithm that runs in $O(|V|^{k+3} \cdot k^{k+2} \cdot (|D|+1)^{3k+2})$ time. $\qquad\square$

7. A Faster Algorithm For Parity Games With Bounded Treewidth

In Section 6 we gave an $O(|V|^{k+3} \cdot k^{k+2} \cdot (|D|+1)^{3k+2})$ time algorithm for parity games with bounded DAG width. In this section, we show that, if the parity game has bounded treewidth, then we can produce a version of the simulation game that can be solved in $O(|V| \cdot (k+1)^{k+5} \cdot (|D|+1)^{3k+5})$ time.

7.1. **Outline.** The version of the simulation game used in this section is very similar to the version used in Section 6, but this time we will be using a tree decomposition. In fact, we will convert the tree decomposition to a *rooted* tree decomposition. Suppose that we want to determine the winner of some vertex s in the parity game. We find a tree decomposition node i such that $s \in X_i$, and we declare i to be the root of T. Then, we turn T into a directed tree, by orienting all edges away from the root i. This step is necessary to obtain our desired complexity.

Recall that the factor of $|V|^{k+3}$ in Theorem 6.4 arose from the fact that we needed to use $(k+3) \cdot \log|V|$ bits of storage in the alternating Turing machine:

- We needed $(k+2) \cdot \log|V|$ bits to store an edge from the DAG decomposition.
- We needed $\log|V|$ bits to store the vertex v.

The first observation that we make is that, when we consider parity games with bounded treewidth, we can easily reduce the amount of space that is required. Whereas a DAG decomposition can have up to $|V|^{k+2}$ edges, a tree decomposition can have at most $|V| - 1$ edges. Thus, we can reduce the amount of space needed by the alternating Turing machine to $2 \cdot \log|V| + (k+2) \cdot \log(k) + (3k+2) \cdot \log(|D|+1)$, and thereby obtain a $O(|V|^2 \cdot k^{k+2} \cdot (|D|+1)^{3k+2})$ time algorithm for solving parity games with bounded treewidth.

We can, however, do better than this. For the rest of this section, our goal is to show that the $2 \cdot \log|V|$ term in our space bounds can be reduced to $\log|V|$. This will lower the $|V|^2$ term in the complexity of the algorithm to $|V|$. To do this, we concentrate on the amount of space used by the variable v. The reason why we require $\log|V|$ bits to store v is that the edge chosen in Step 1 can potentially set v to be any vertex in the parity game. Figure 4 illustrates this. It shows a subtree of a tree decomposition: specifically, it shows the tree decomposition nodes i with $u \in X_i$. Each node in the tree decomposition is represented as a circle. For each set X_i, the diagram displays the outgoing edges from u that are contained in that node. That is, for each set X_i, it displays the edges of the form (u, w) with both $u \in X_i$ and $w \in X_i$.

Suppose that we are playing a simulation game on the set X_j, and that the current vertex $c = u$. In Step 1, Even could select the edge (u, h). Since X_m is far from X_j in the tree decomposition, the best we can do is to use $\log|V|$ bits to store the vertex h.

We will fix this problem by modifying the parity game. The modification procedure is illustrated in Figure 5. Each vertex in the graph will have its outgoing edges modified. Figure 5A shows an example vertex u, and Figure 4 gives an example tree decomposition, where only the nodes that contain u are shown. Figures 5A shows the outgoing edges from u
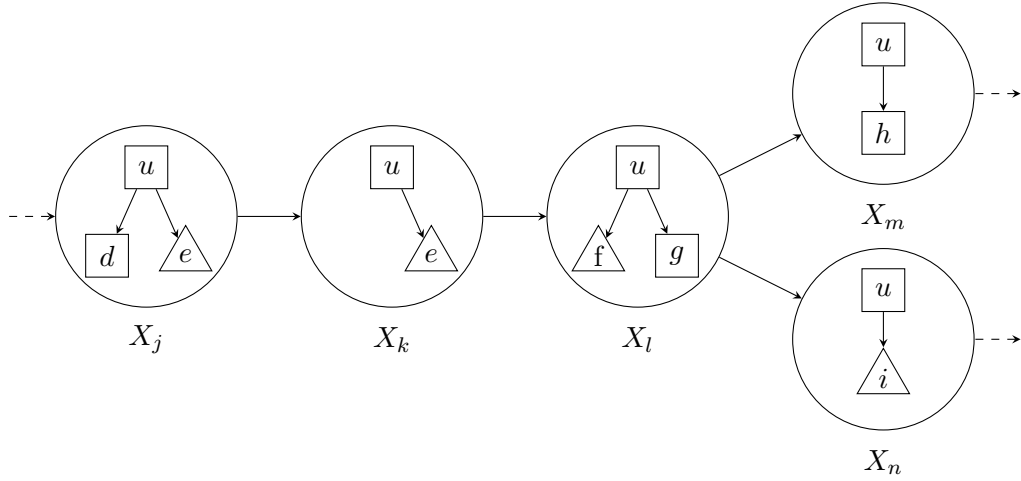
FIGURE 4. The outgoing edges from the vertex u shown in a tree decomposition.

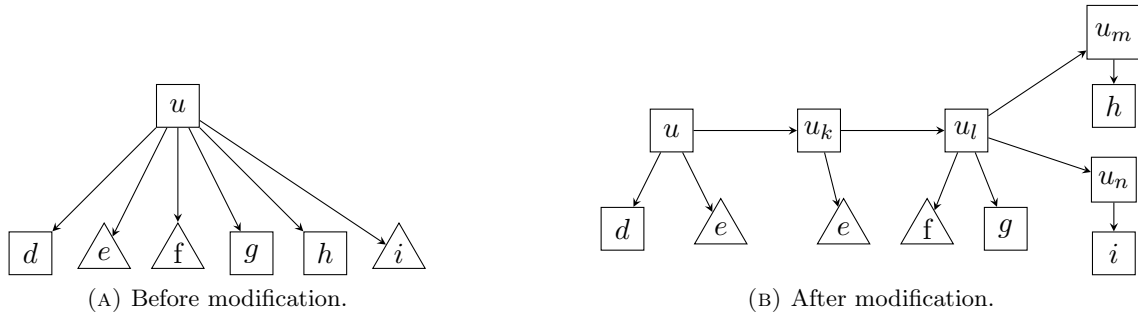(A) Before modification.

(B) After modification.

FIGURE 5. The modification procedure.

in the original parity game, and Figure 5B shows the changes that we make to the outgoing edges from u.

Suppose that we are playing a simulation game on X_j, and that we have arrived at u. Now suppose that Even wants to move to some vertex not in X_j, for example, the vertex g. The modification forces Even to a path in the tree decomposition to a node that contains h, before moving to h.

Without this modification, we would be forced to use $\log v$ bits to hold the variable c, because in Step 1 of the simulation game, Even could potentially pick any vertex in the graph. On the other hand, suppose that we have made the modification, and that are playing a simulation game on X_j and that $c = u$. Let (u, v) be the edge picked by Even in Step 1.

- If $v \in X_j$, then we can represent v using $\log k$ bits.
- If $v \notin X_j$, then we must have that $v = u_i$ for some tree decomposition edge (j, i). In this case, we can store v by remembering u and (j, i).

In the second case, the vertex u can be stored using $\log k$ bits. We claim that the edge (j, i) can be stored without using any additional memory. Recall that we are already required to remember a tree decomposition edge of the form (l, j), in order to represent the current set

S and the set F from the history. Thus, to implement the simulation game on an alternating Turing machine, we need to store two tree decomposition edges, and these edges form a path of length two in the tree decomposition. Our overall space complexity depends on the following simple lemma.

Lemma 7.1. *Let $T = (I, J)$ be a rooted directed tree, with all edges oriented away from the root. There are at most $|I|$ directed two-edge paths in T.*

Proof. Let $i \in I$ be a node. If i is the root, or a successor of the root, then there are no two-edge paths ending at i. Otherwise, i is the is the end of exactly one two-edge path. Therefore, there are at most $|I|$ two-edge paths in T. □

Lemma 7.1 shows that a directed two-edge path in the tree decomposition can be stored in $\log |V|$ bits. This fact then implies the required space bounds for the alternating Turing machine.

The rest of this section proceeds as follows. We begin by formalising the modified parity game. We then show how the simulation game can be used to solve the modified parity game. To do this, we introduce two functions $\mathsf{Next_{TW}}$ and $\mathsf{Hist_{TW}}$, and show that, if the simulation game is equipped with $\mathsf{Next_{TW}}$ and $\mathsf{Hist_{TW}}$, then it always terminates on the modified parity game. Finally, we show that this version of the simulation game can be solved on an alternating Turing machine, and from this we obtain the main result of this section.

7.2. **A modified parity game.** For the rest of this section, we will fix a parity game $G = (V, V_0, V_1, E, \mathrm{pri})$, and a rooted tree decomposition $(T = (I, J), X)$. In this subsection, we describe a modified parity game G', which will be critical in obtaining our result. For every vertex $v \in V$, we define $\mathrm{First}(v, T)$ to give the node $i \in I$ such that $v \in X_i$, and i is closest to the root in T.

We can now define our modified parity game. This definition follows the idea laid out in Figure 5. The first step is, for each vertex $v \in V$, to add extra vertices v_i for every tree decomposition node $i \in I$ other than $\mathrm{First}(v, T)$. Then, we add edges according to the schema laid out by Figure 5.

Definition 7.2. We define $G' = (V', V_0', V_1', E', \mathrm{pri}')$ as follows:

- $V' = V \cup \{v_i \ : \ v \in X_i \text{ for some } i \in I \text{ and } i \neq \mathrm{First}(v, T)\}$.
- $V_0' = V_0 \cup \{v_i \ : \ v \in V_0 \text{ and } v_i \in V' \text{ for some } i \in I\}$
- $V_1' = V_1 \cup \{v_i \ : \ v \in V_1 \text{ and } v_i \in V' \text{ for some } i \in I\}$.
- The set E' is constructed by the following procedures:
  - For every pair of vertices $v_i, v_j \in V'$, if there is an edge $(i, j) \in J$, then we add the edge (v_i, v_j).
  - For every $v_i \in V'$, if there is a vertex $u \in X_i$ and an edge $(v, u) \in E$, then we add (v_i, u) to E'.
  - For every vertex $v \in V$, if there is a vertex $u \in X_{\mathrm{First}(v,T)}$, and an edge $(v, u) \in E$, then we add (v, u) to E'.
  - For every vertex $v \in V$, if there exists a vertex $v_i \in V'$ with $(\mathrm{First}(v, T), i) \in J$, then we add (v, v_i) to E'.
- We set $\mathrm{pri}'(v) = \mathrm{pri}(v)$ for every $v \in V$, and for every $v_i \in V'$ we set $\mathrm{pri}'(v_i) = \mathrm{pri}(v)$.

It is not difficult to see that this construction cannot change the winner of the parity game. This is because, if there is an edge $(v, u) \in E$, then there is a path $\langle v_1, v_2, \ldots, v_k \rangle$ in G' with $v_1 = v$, $v_k = u$, and where v_2 through v_{k-1} belong to the owner of v_1. Thus, if Even has a winning strategy in G that uses the edge (v, u), then we can construct a strategy for Even in G' that follows the corresponding path from v to u. The same property also holds for Odd's winning strategies. Thus, we have the following lemma.

Lemma 7.3. *Let $s \in V$ be a vertex. Even wins from s in G if, and only if, Even wins from s in G'.*

For the rest of this section, we will develop a version of the simulation game that can be used to solve G'. Although we have modified the parity game, we will not compute a new tree decomposition. We will develop an algorithm that uses the original tree decomposition and the modified parity game.

7.3. **The functions $\mathsf{Next}_{\mathsf{TW}}$ and $\mathsf{Hist}_{\mathsf{TW}}$.** We now define two functions, $\mathsf{Next}_{\mathsf{TW}}$ and $\mathsf{Hist}_{\mathsf{TW}}$. The idea is to use the same approach as we did in Section 6. That is, if we want to determine the winner for some vertex s in the parity game, then we find an $i \in I$ with $s \in X_i$, and we play $\mathsf{Simulate}_{G'}(S, \emptyset, s, \mathsf{Next}_{\mathsf{TW}}, \mathsf{Hist}_{\mathsf{TW}})$.

We start by defining the function $\mathsf{Next}_{\mathsf{TW}}$. This function will always choose sets of the form $X_i \cup \{v_i\}$, where $i \in I$ is a tree decomposition node, and $v_i \in V'$ is a vertex. Suppose that we are required to compute $\mathsf{Next}_{\mathsf{TW}}(S, v, \mathcal{F})$, for some set S, some vertex $v \in V'$, and some history $\mathcal{F}$. Since S was either chosen by $\mathsf{Next}_{\mathsf{TW}}$, or was the set chosen at the start of the game, we know that $S \subseteq X_i \cup \{v_i\}$ for some $i \in I$ and some $v_i \in V'$. Let (u, w) be an edge with $u \in S$ and $w \notin S$. The definition of G' implies that w is one of the extra vertices added that were added while constructing G'. That is, we have $w = u_j$ for some tree decomposition edge $(i, j) \in J$. Thus, we know that, if we are required to compute $\mathsf{Next}_{\mathsf{TW}}(S, v, \mathcal{F})$, then we must have $v = u_j$ for some $u \in S$, and some tree decomposition edge (i, j). We define $\mathsf{Next}_{\mathsf{TW}}(S, u_j, \mathcal{F}) = X_j \cup \{u_j\}$.

Next, we define the $\mathsf{Hist}_{\mathsf{TW}}$ function. It is very similar to the Hist function defined in Section 6, and it is defined as follows:

- The first time that $\mathsf{Hist}_{\mathsf{TW}}(\mathcal{F})$ is called, $\mathcal{F}$ will contain only one record. We do not delete this record, so we set $\mathsf{Hist}(\mathcal{F}) = \mathcal{F}$.
- Every other time that $\mathsf{Hist}(\mathcal{F})$ is called, $\mathcal{F}$ will contain exactly two records. Due to the definition of $\mathsf{Next}_{\mathsf{TW}}$, we know that the two records in $\mathcal{F}$ have the form (S_1, v_1, P_1) and (S_2, v_2, P_2), where $X_i \subseteq S_1$ and $X_j \subseteq S_2$, for some $i, j \in I$. Furthermore, we know that $(i, j) \in J$ is a directed edge in the tree decomposition. We set $\mathsf{Hist}(\mathcal{F}) = \{(S_2, v_2, P_2)\}$.

The next lemma shows that, if we equip the simulation game with the functions $\mathsf{Next}_{\mathsf{TW}}$ and $\mathsf{Hist}_{\mathsf{TW}}$, then the simulation game always terminates. Since we are following the approach used in Section 6, this proof is similar in structure to the proof of Lemma 6.2.

Lemma 7.4. *If i is the root of the tree decomposition, and $s \in X_i$ is a vertex, then $\mathsf{Simulate}_{G'}(X_i, \emptyset, s, \mathsf{Next}_{\mathsf{TW}}, \mathsf{Hist}_{\mathsf{TW}})$ terminates.*

Proof. We begin by arguing that Odd cannot play `reject` an infinite number of times. This follows from the definition of $\mathsf{Next}_{\mathsf{TW}}$: we know that if Odd plays `reject` in a simulation game where $X_i \subseteq S$ for some $i \in I$, then the next game will be played on a set S' with $X_j \subseteq S'$ for some directed edge $(i, j) \in J$. Thus, after Odd has played `reject` at most $|I|$ times, we will have arrived at a leaf in the tree decomposition.

The $\mathsf{Hist_{TW}}$ function ensures that, once we reach a leaf in the tree decomposition, the game must end. Suppose that we have reached a simulation game on S, where $S = X_j \cup \{w_j\}$ where $j \in I$ is a leaf, and $w \in X_j$ is a vertex. The history function ensures that $\mathcal{F}$ contains a record (F, p, P) with $X_k \subseteq F$, where $(k, j) \in J$. We now have two properties:

- For every edge (v, u) with $v \in X_j \setminus X_k$ and $u \notin X_j$, we have $u \in X_k$. This follows from the definition of a tree decomposition given in Definition 2.2.
- For every edge (v, u) with $v = w_j$, we have $u \in X_j$. This follows from the definition of E' given in Definition 7.2.

Thus, we have shown that, for every edge (v, u) with $v \in X_j \setminus X_k$ and $u \notin X_j$, we have $u \in X_k$. Therefore, since $X_k \subseteq F$, Step 2 of the simulation game ensures that Odd cannot play `reject`.

Since S contains at most $k+1$ vertices, we know that, if Odd can no longer play `reject`, then Odd can play `accept` at most $k + 2$ times before the game is forced to end in either Step 5 or Step 6. □

Now that we have established termination, we can apply Lemma 5.7 to imply the correctness of this version of the simulation game.

7.4. Implementation on an alternating Turing machine.

Finally, we show how this version of the simulation game can be implemented on an alternating Turing machine. As usual, the existential and universal states of the alternating Turing machine will be used to simulate the moves of Even and Odd. We have the following theorem.

Theorem 7.5. *Given a parity game G, a tree decomposition $(T = (I, J), X)$ of width k for G, and a vertex s, we can determine the winner of s in $O(|V| \cdot (k + 1)^{k+5} \cdot (|D| + 1)^{3k+5})$ time.*

Proof. For the most part, the proof of this theorem is the same as the proof of Theorem 6.4. Recall, from the proof of Theorem 6.4, that we must remember the following variables: S, $\mathcal{F} = \{F, p, P\}$, Π, P', c, and v.

Our central claim is that S, F, and v can be stored in $\log |V| + 2 \cdot \log(k + 1)$ bits. Note that, $\mathsf{Next_{TW}}$ and $\mathsf{Hist_{TW}}$ ensure that $S \subseteq X_i \cup \{s_i\}$ and $F \subseteq X_j \cup \{f_j\}$, for some $(j, i) \in J$, and some vertices $s, f \in V$. By Definition 7.2, there cannot be an edge (v, u) with $v \in S \setminus F$, and $u = f_j$. Thus, our alternating Turing machine can represent F as the set X_j, while forgetting the vertex f_j, and this cannot affect the correctness of the algorithm.

Now we argue that S, F, and v can be stored in $\log |V| + 2 \log(k + 1)$ bits. There are two cases to consider:

- If $v \in S$, then X_i and X_j can be represented by pointing to an edge in J. Since T is a tree, we know that $|J| \leq |I| \leq |V|$. Then, to store s_i we can simply store the vertex s in $\log k$ bits, because we already know X_i. Since $v \in S$, and since $|S| \leq k + 1$, we can store v using $\log(k + 1)$ bits.
- If $v \notin S$, then we know that $v = u_l$ for some $u \in S$, and some $l \in I$. Note that j, i, and l form a directed two-edge path in T. Thus, by Lemma 7.1, we can represent j, i, and l, in $\log |I| \leq \log |V|$ bits. As with the previous case, we can represent the extra vertex s_i using $\log k$ bits. Moreover, since $u \in S$, we can represent v using at most $\log(k + 1)$ bits.

Therefore, in both cases we can store S, F, and v using $\log |V| + \log(k + 1) + \log k$ bits.

The amount of space for the variables p, P, Π, P', and c follow from the arguments made in the proof of Theorem 6.4. However, there is one small difference: in this proof, we

have $|S| = k + 1$ rather than $|S| = k$. Thus, while P could be stored in $k \cdot \log(D + 1)$ bits in Theorem 6.4, we must use $(k + 1) \cdot \log(D + 1)$ bits in this proof. Similar modifications are required for the other variables.

In total, therefore, we require at most $\log |V| + (k + 5) \cdot \log(k + 1) + (3k + 5) \cdot \log(|D| + 1)$ bits of storage in our alternating Turing machine. This alternating Turing machine can be implemented on a deterministic Turing machine in $O(|V| \cdot (k + 1)^{k+5} \cdot (|D| + 1)^{3k+5})$ time. $\square$

8. Parity Games With Bounded Treewidth Are In NC^2

We begin by defining the Next and Hist functions that will be used in by simulation game in this result. Recall, from Section 4.3, that this version of the simulation game gives extra strategic choices to player Odd. We use this idea to define $\mathsf{Next_{NC}}$ and $\mathsf{Hist_{NC}}$ as follows:

- When Step 4 requires a value for $\mathsf{Next_{NC}}(S, v, \mathcal{F}''')$, we allow player Odd to select the set of vertices that will be used. More precisely, we allow Odd to select $S' \subseteq V$ such that $|S'| \leq k$ and $v \in S'$.
- When Step 4 requires a value for $\mathsf{Hist_{NC}}(\mathcal{F}'')$, we allow Odd to to select the set $\mathcal{F}'''$ by deleting records from $\mathcal{F}''$. We insist that, if $|\mathcal{F}''| > 3$, then Odd must delete enough records so that $|\mathcal{F}| \leq 3$.

Thus, we will use $\mathsf{Simulate}(S, \mathcal{F}, s, \mathsf{Next_{NC}}, \mathsf{Hist_{NC}})$ as the simulation game in this section.

Note that this simulation game could go on forever, since Odd could, for example, always play `reject`, and remove all records from $\mathcal{F}$. For this reason, we define a limited-round version of the game, which will be denoted by $\mathsf{Simulate}^r(S, \mathcal{F}, s, \mathsf{Next_{NC}}, \mathsf{Hist_{NC}})$, where $r \in \mathbb{N}$. This game is identical to $\mathsf{Simulate}^r(S, \mathcal{F}, s, \mathsf{Next_{NC}}, \mathsf{Hist_{NC}})$ for the first r rounds. However, if the game has not ended before the start of round $r + 1$, then Even is declared to be the winner.

Let $r \in \mathbb{N}$ be an arbitrarily chosen bound. The following lemma states that, no matter what sets Odd chooses, if the simulation game stops before round $r + 1$, then the winner of the parity game is correctly determined.

Lemma 8.1. *No matter which sets are chosen by Odd for $\mathsf{Next_{NC}}$ and $\mathsf{Hist_{NC}}$ in Step 4:*

- *if $s \in W_0$, then Even has a strategy to win $\mathsf{Simulate}^r(\{s\}, \emptyset, s, \mathsf{Next_{NC}}, \mathsf{Hist_{NC}})$.*
- *If $s \in W_1$, and if the game ends before round $r + 1$, then Odd has a strategy to win $\mathsf{Simulate}^r(\{s\}, \emptyset, s, \mathsf{Next_{NC}}, \mathsf{Hist_{NC}})$.*

The correctness of this lemma follows from Lemma 5.7. Note that this lemma does not guarantee anything if the simulation game terminates in round $r + 1$. Our goal in this section is to show that there is an r such that Odd can always force the game to end before round $r + 1$, which will then allow us to apply Lemma 8.1 to prove correctness of our algorithm.

We will assume that our parity game has a tree decomposition $(T = (I, J), X)$ of width k. Our main goal is to show that Odd has a strategy for choosing $\mathsf{Next_{NC}}$ and $\mathsf{Hist_{NC}}$ that forces the game to terminate within the first r rounds, for some $r \in \mathbb{N}$. In particular, to show that parity games with bounded treewidth lie in NC^2, our strategy must have the following properties:

- The game must end in $O(k \cdot \log |V|)$ rounds. That is, we have $r = c \cdot k \cdot \log |V|$ for some constant c.
- The history may never contain more than three records.

In the following subsections, we will provide such a strategy.

8.1. **Odd's strategy for $\mathsf{Next}_{\mathsf{NC}}$ and $\mathsf{Hist}_{\mathsf{NC}}$.** Recall, from Section 4.3, that the strategy for Odd consists of two sub-strategies: `Slice` and `Reduce`. The strategy `Slice` will be applied whenever $\mathcal{F}$ contains fewer than 3 records, and `Reduce` will be applied whenever $\mathcal{F}$ contains exactly 3 records. Both of these strategies maintain a set of *eligible* tree decomposition nodes $L \subseteq I$. The set of eligible tree decomposition nodes represents the portion of the parity game that can still be reached by the simulation game. An intuitive idea of this is given in Figures 3A and 3B. In both cases, the set of eligible nodes is the subtree between the nodes F_1, F_2, and F_3.

We give a formal definition for how L will be updated. Suppose that Odd rejects the profile P for the vertex v and set X_i. We define $\mathrm{Subtree}(X_i, v)$ to be the set of nodes in the subtree of i rooted at $\mathrm{Direction}(X_i, v)$. We update L as follows.

- The first time that `reject` is played, Odd sets $L = I$.
- In each subsequent iteration, the set L is updated so that $L := L \cap \mathrm{Subtree}(X_i, v)$.

Odd uses L to decide which records should be deleted. Formally, Odd will select $\mathsf{Hist}_{\mathsf{NC}}(\mathcal{F})$ according to the following rules:

- The first time that `reject` is played, Odd removes all records from $\mathcal{F}$.
- In each subsequent iteration, Odd uses the following rule. Let $(F, p, P) \in \mathcal{F}$, and suppose that $X_i \subseteq F$ for some $i \in I$. If there is no edge $(i, j) \in J$ with $j \in L$, then Odd removes (F, p, P) from $\mathcal{F}'$.

We now define the strategy `Slice`. Recall that Lemma 4.1 shows that every tree with more than three nodes can be split into pieces, where each piece has at least two-thirds of the nodes in the tree. We define $\mathrm{Split}(I, J)$ to be a function that returns a vertex $i \in I$ that satisfies the properties given in Lemma 4.1. The strategy `Slice` will use the function Split to remove at least one third of the eligible nodes. Formally, we define $\mathtt{Slice}(S, v, L, \mathcal{F})$. The strategy is defined as follows:

- If $|L| \geq 3$ then $\mathtt{Slice}(S, v, L, \mathcal{F}) = X_i$ where $i = \mathrm{Split}(L, J)$.
- If $|L| < 3$ then $\mathtt{Slice}(S, v, L, \mathcal{F}) = X_i$ for some arbitrarily chosen $i \in L$.

The second clause is necessary, because Lemma 4.1 requires that the tree should have at least three vertices.

We now introduce our second strategy for choosing S', which is called `Reduce`. This strategy will be applied whenever $|\mathcal{F}| = 3$. Recall that the purpose of `Reduce` is to reduce the number of records in $\mathcal{F}$ to at most 2, so that we can continue to apply `Slice`. Suppose that $\mathcal{F} = \{(F_1, p_1, P_1), (F_2, p_2, P_2), (F_3, p_3, P_3)\}$. By assumption we have that $F_1 = X_i$, $F_2 = X_j$, and $F_3 = X_k$, for some $i, j, k \in I$. Note that i, j and k cannot all lie on the same path, because otherwise one of the records in $\mathcal{F}$ would have been deleted by $\mathsf{Hist}_{\mathsf{NC}}$. It is a basic property of trees that, if i, j, and k are not on the same path, then there exists a unique vertex $l \in I$ such that, if l is removed, then i, j, and k, will be pairwise disconnected. For each history $\mathcal{F}$ with $|\mathcal{F}| = 3$, we define $\mathrm{Point}(\mathcal{F})$ to be the function that gives the vertex l. We define the strategy $\mathtt{Reduce}(S, v, L, \mathcal{F}) = X_i$, where $i = \mathrm{Point}(\mathcal{F})$.

Finally, we can give the full strategy for Odd. Odd selects $\mathsf{Next}_{\mathsf{NC}}(S, v, \mathcal{F})$ according to the following rules:

- If $|\mathcal{F}| < 3$, then Odd selects $S' = \mathtt{Slice}(S, v, L, \mathcal{F})$.
- If $|\mathcal{F}| = 3$, then Odd selects $S' = \mathtt{Reduce}(S, v, L, \mathcal{F})$.

8.2. **Correctness of** SR. In this subsection, we show that SR has both of the properties that we desire. Suppose that Odd uses SR, and that the outcome is a play of the simulation game:

$$\langle \mathcal{S}_1 = \mathsf{Simulate}(S_1, \mathcal{F}_1, s_1, \mathsf{Next_{NC}}, \mathsf{Hist_{NC}}), \mathcal{S}_2 = \mathsf{Simulate}(S_2, \mathcal{F}_2, s_2, \mathsf{Next_{NC}}, \mathsf{Hist_{NC}}), \dots \rangle.$$

Furthermore, let L_i be the contents of L at the start of the game $\mathcal{S}_i$, and let $v_i = s_{i+1}$ be the final vertex visited in $\mathcal{S}_i$. The first property that we will prove is that, as long as Odd plays SR, the set L is guarded by the records in $\mathcal{F}$. Informally, this means that, in order to move from a tree decomposition node $i \in L$, to a tree decomposition node $j \notin L$, we must pass through some record in $\mathcal{F}$. Formally, this can be expressed as the following lemma.

Lemma 8.2. *Suppose that Odd plays* SR. *If $z \geq 2$, then for every edge $(i, j) \in J$ with $i \in L_z$, and $j \notin L_z$, there exists a record $(F, p, P) \in \mathcal{F}_z$ with $X_z = F$.*

Proof. We will prove this claim by induction on z. The base case is vacuously true, since SR sets $L_2 = I$, which implies that there cannot exist an edge (i, j) with $j \notin L_2$.

For the inductive step, assume that this lemma holds for some $z > 2$. Since $L_{z+1} = L_z \cap \mathsf{Subtree}(S_z, v_z)$, the only possible edge that could be not covered by a set in $\mathcal{F}_z$ is the edge (i, j), where $X_i = S_{z+1}$, and $j \in \mathsf{Subtree}(S_{z+1}, v_{z+1})$. However, this edge is covered by two properties:

- In Step 4, the simulation game always adds the record $(F, -, P')$ to $\mathcal{F}'$ to create $\mathcal{F}''$.
- When choosing $\mathcal{F}'''$, the strategy SR never removes a record (F, p, P) from $\mathcal{F}''$ with $F \subseteq X_j$, where $(i, j) \in J$ and $i \in L_{z+1}$ and $j \notin L_{z+1}$.

Thus, a record (F, p, P) with $F = X_i$ must be contained in $\mathcal{F}_{z+1}$, which completes the proof of the inductive step. □

Next we will show the correctness of Reduce. In particular, we show how if it is used when $|\mathcal{F}_x| = 3$, it enforces that $|\mathcal{F}_{x+1}| \leq 2$.

Lemma 8.3. *Suppose that Odd plays* SR, *and for some $x \geq 2$ we have $|\mathcal{F}_x| = 3$, and $S_x = \mathsf{Reduce}(S_{x-1}, v_{x-1}, L_{x-1}, \mathcal{F}_x)$. We have $|\mathcal{F}_{x+1}| \leq 2$.*

Proof. Since $x \geq 2$, and since Odd is following SR, we know that there must exist three tree decomposition nodes $i, j, k \in I$, such that, if (F_1, p_1, P_1), (F_2, p_2, P_2), and (F_3, p_3, P_3) are the three records in $\mathcal{F}_x$, then we have $F_1 = X_1$, $F_2 = X_j$, and $F_3 = X_k$. By definition, $\mathsf{Reduce}(S_{x-1}, v_{x-1}, L_{x-1})$ selects a vertex l such that, if l is removed from $T = (I, J)$, then i, j, and k become disconnected. Thus, at most one of i, j, and k can lie in $\mathsf{Subtree}(S_{x-1}, v_{x-1})$, and hence at least two records will be removed by SR. Therefore, since at most one new record is added to $\mathcal{F}_{x+1}$, we must have $|\mathcal{F}_{x+1}| \leq |\mathcal{F}_x| + 1 - 2 = 2$. Thus, we have shown that $|\mathcal{F}_{x+1}| \leq 2$. □

Finally, we are able to prove that SR has the property that we require.

Lemma 8.4. *Suppose that the parity game has a tree decomposition of width k. If Odd plays* SR, *then the game ends in $O(k \cdot \log |V|)$ rounds.*

Proof. Lemma 4.1 implies that, if Slice is invoked at the end of $\mathcal{S}_i$, then we will have $|L_{i+1}| \leq \frac{2}{3}|L_i|$. Each time Slice is used, we add at most one new record to $\mathcal{F}$. Therefore, for each use of Slice, we must use Reduce at most 1 time, in order to keep $|\mathcal{F}| \leq 3$. Thus, after Odd has played reject at most $2 \cdot \log_{\frac{3}{2}}(|I|)$ times, we will arrive at a game $\mathcal{S}_c$ with $|L_c| < 3$.

We claim that once we have $|L_c| < 3$, Odd can play `reject` at most two more times. This follows from the fact that for every edge $(i,j) \in J$ with $i \in L$ and $j \notin L$, we must have $X_j \subseteq F_a$ for some F_a in $\mathcal{F}_c$. Therefore, after Odd plays `reject` two more times, we will arrive at a game $\mathcal{S}_{c+2} = \mathsf{Simulate}_G(S, \mathcal{F}, s, \mathsf{Next}_{\mathsf{NC}}, \mathsf{Hist}_{\mathsf{NC}})$ such that, if $X_i \subseteq S$, then for all edges $(i,j) \in J$ we have $X_j = F_a$ for some set F_a in $\mathcal{F}$. This is because the vertex $i \in I$ such that $S = X_i$ is always removed from L, and so it does not matter whether Odd uses `Slice` or `Reduce` at this point. In $\mathcal{S}_{c+2}$, we know that Step 2 prevents Odd from playing `reject`, and therefore the game will end.

So far, we have that Odd can play `reject` at most $2 \cdot \log_{\frac{3}{2}}(|I|) + 2$ times. Note that between each instance of `reject`, Odd can play `accept` at most $k + 1$ times without triggering Step 5 or Step 6. Since $|I| \leq |V|$ we can therefore conclude that the game can last at most $(k + 1) \cdot (2 \cdot \log_{\frac{3}{2}}(|V|) + 2)$ times, and this is contained in $O(k \cdot \log |V|)$. □

8.3. Implementation on an alternating Turing machine.

Let G be a parity game, and let $G' = (V, V_0', V_1', E, \mathrm{pri}')$ be a parity game that swaps the roles of the two players in G. More formally we have:

- $V_0' = V_1$, and $V_1' = V_0$.
- For each $v \in V$ we set $\mathrm{pri}'(v) = \mathrm{pri}(v) + 1$.

It should be clear that Even can win from a vertex $s \in V$ in G if and only if Odd can win from s in G'.

Suppose that we want to find the winner of a vertex s. We choose $c = O(k \cdot \log |V|)$, where the constant hidden by the $O(\cdot)$ notation is given in the proof of Lemma 8.4. We then solve $\mathsf{Simulate}_G^c(\{s\}, \emptyset, s, \mathsf{Next}_{\mathsf{NC}}, \mathsf{Hist}_{\mathsf{NC}})$ and $\mathsf{Simulate}_{G'}^c(\{s\}, \emptyset, s, \mathsf{Next}_{\mathsf{NC}}, \mathsf{Hist}_{\mathsf{NC}})$. If both games declare that Even wins s, then we declare that the game has treewidth larger than k. Otherwise, we declare that the winner of $\mathsf{Simulate}_G^c(\{s\}, \emptyset, s, \mathsf{Next}_{\mathsf{NC}}, \mathsf{Hist}_{\mathsf{NC}})$ is the winner of s. Lemmas 5.3, 5.4, and 8.4 imply that this procedure is correct.

Thus, to prove our main result, we must show that $\mathsf{Simulate}_G^c(\{s\}, \emptyset, s, \mathsf{Next}_{\mathsf{NC}}, \mathsf{Hist}_{\mathsf{NC}})$ can be solved in $O(k^2 \cdot (\log |V|)^2)$ time and $O(k \cdot \log |V|)$ space by an alternating Turing machine. To implement the game on an alternating Turing machine, we use the existential states to simulate Even's moves and the universal states to simulate Odd's moves. We begin by proving the time bound.

Lemma 8.5. *A simulation of* $\mathsf{Simulate}_G^c(\{s\}, \emptyset, s, \mathsf{Next}_{\mathsf{NC}}, \mathsf{Hist}_{\mathsf{NC}})$ *by an alternating Turing machine uses at most* $O(k^2 \cdot (\log |V|)^2)$ *time.*

Proof. By definition we have that $\mathsf{Simulate}_G^c(\{s\}, \emptyset, s, \mathsf{Next}_{\mathsf{NC}}, \mathsf{Hist}_{\mathsf{NC}})$ allows the game to continue for at most $O(k \cdot \log |V|)$ rounds. Each step of the game takes the following time:

- Since we are only required to guess a vertex in Step 1, this step can be implemented in $\log |V|$ time.
- In Step 2 checking whether $v \in S$ can be done in $k \cdot \log |V|$ time. Note that we can have at most 3 records in $\mathcal{F}$. Therefore, the check for whether $v \in F$ for some $(F, p, P) \in \mathcal{F}$ can also be performed in $3 \cdot k \cdot \log |V|$ time.
- Generating a strategy profile in Step 3 can take at most $k \cdot \log(|D| + 1)$ time.
- If Odd plays `accept` in Step 4, then, if we always maintain a pointer to the end of Π, appending $(v, P(u), u)$ to Π will take $2 \cdot \log |V| + \log |D|$ time.
- If Odd plays `reject` in Step 4 then:

- Since computing $\text{MaxPri}(\Pi)$ can take at most $k \cdot \log |D|$ time, and since $|\mathcal{F}| \le 3$, we have that $\mathcal{F}'$ can computed in $k \cdot \log |D|$ time.
  - Appending $(F, -, P')$ to $\mathcal{F}'$ when creating $\mathcal{F}''$ will take $k \cdot \log |V| + \log(|D| + 1) + k \cdot \log(|D| + 1)$ time.
  - Implementing Odd's strategy for $\mathsf{Hist_{NC}}$ can be done in $4 \cdot k \cdot \log |V| + \log(|D| + 1) + k \cdot \log(|D| + 1)$ time. This is because Odd is only allowed to remove records from $\mathcal{F}''$, and because $|\mathcal{F}''| \le 4$.
  - Implementing Odd's strategy for $\mathsf{Next_{NC}}$ involves picking a subset of vertices with size at most k. This can be done in $k \cdot \log |V|$ time.
- In Step 5, since $|\mathcal{F}_i| \le 3$, we can check whether $c \in F$, for some $(F, p, P) \in \mathcal{F}_i$, in $3 \cdot k \cdot \log |V|$ time. If the game stops here, then computing p', which requires us to find $\text{MaxPri}(\Pi)$, must take at most $k \cdot \log |D|$ time.
- In Step 6, determining if Π forms a cycle can be done by checking for each tuple $(v, p, u) \in \Pi$ if $c = v$. Since there can be at most k tuples in Π, this can be done in $k \cdot \log |V|$ time. Finding the highest priority on the cycle then takes at most $k \cdot \log |D|$ time.

Since $|D| \le |V|$, we have that each round takes at most $O(k \cdot \log |V|)$ time. Since there are at most $O(k \cdot \log |V|)$ rounds, the machine must terminate in $O(k^2 \cdot (\log |V|)^2)$ time. □

Next, we prove the space bounds for the alternating Turing machine.

Lemma 8.6. *A simulation of $\mathsf{Simulate}_G^c(\{s\}, \emptyset, s, \mathsf{Next_{NC}}, \mathsf{Hist_{NC}})$ by an alternating Turing machine uses at most $O(k \cdot \log |V|)$ space.*

Proof. In order to simulate the game, we must remember the set S, the vertex c and the vertex v, the strategy profile P', the path Π, and the history $\mathcal{F}$. It is important to note that we do not need to remember the set L used by SR. In Lemma 8.4 we showed that there *exists* an Odd strategy that forces the game to terminate in $O(k \cdot \log |V|)$. The existence of such a strategy is sufficient to ensure that the alternating Turing machine computes the correct answer, and we do not need to actually implement the strategy on the alternating Turing machine.

Therefore, the amount of space used by the alternating Turing machine is as follows.

- Since we always have $|S| \le k$, we know that S requires at most $k \cdot \log |V|$ bits to store.
- The vertices c and v each require $\log |V|$ bits to store.
- The strategy profile P' contains $|S|$ mappings $u \to p$, where $u \in S$ and $p \in D \cup \{-\}$. Therefore, P' can be stored using $k(\log |V| + \log(|D| + 1))$ bits.
- Since $|S| \le k$, the path Π contains at most k tuples of the form (v, p, u), where $v, u \in V$ and $p \in D \cup \{-\}$. Therefore, storing the path Π requires at most $3 \cdot k \cdot (\log |V| + \log(|D| + 1))$ bits.
- For each record $(F, p, P) \in \mathcal{F}$, the set F requires $k \cdot \log |V|$ bits to store, the priority p requires $\log(|D| + 1)$ bits to store, and we have already argued that the strategy profile P requires at most $k \cdot (\log |V| + \log(|D| + 1))$ bits to store. Since $|\mathcal{F}| \le 3$, we have that $\mathcal{F}$ requires at most $3 \cdot (2 \cdot k \cdot \log |V| + (k + 1) \cdot \log(|D| + 1))$ bits to store.

Since $|D| \le |V|$, we have shown that the alternating Turing machine requires at most $O(k \cdot \log |V|)$ space. □

Having shown these two properties, we now have the main result of this section.

Theorem 8.7. *Let G be a parity game and k be a parameter. There is an alternating Turing machine that takes $O(k^2 \cdot (\log |V|)^2)$ time, and uses $O(k \cdot \log |V|)$ space, to either*

determine the winner of a vertex $s \in V$, or correctly report that G has treewidth larger than k.

A simple corollary of the time and space results for our alternating Turing machine is that our problem lies in the class NC^2 [ALR10, Theorem 22.15].

Corollary 8.8. *The problem of solving a parity game with bounded treewidth lies in NC^2.*

9. CONCLUSION

We have seen three results: a $O(|V|^{k+3} \cdot k^{k+2} \cdot (|D|+1)^{3k+2})$ time algorithm for parity games with bounded DAG width, a $O(|V| \cdot (k+1)^{k+5} \cdot (|D|+1)^{3k+5})$ time algorithm for parity games with bounded treewidth, and a proof that parity games with bounded treewidth lies in NC^2.

It is worth noting that the running time of our algorithm for parity games with bounded DAG width includes a factor of $|V|^{k+2}$, because that is the best known upper bound for the number of edges that can appear in a DAG decomposition. If, in the future, better upper bounds are derived, then the running time of our algorithm will see a corresponding improvement.

An interesting open problem is: are there *fixed parameter tractable* algorithms for solving parity games with bounded treewidth? An algorithm is fixed parameter tractable if its running time can be expressed as $O(poly(V) \cdot f(k))$, where the degree of the polynomial is independent of the treewidth. Neither of the algorithms that we have presented satisfy this property, because they include the terms $|V|^{k+3}$ and $(|D|+1)^{3k+5}$, respectively. In the case of DAG width, if the upper bounds on the size of the DAG decomposition are tight, then it seems unlikely that there exist fixed parameter tractable algorithms. However, for the case of treewidth, it is entirely possible that such algorithms could exist.

REFERENCES

[ALR10]   E. Allender, M. C. Loui, and K. W. Regan. Complexity classes. In Mikhail J. Atallah and Marina Blanton, editors, *Algorithms and theory of computation handbook*, page 22. Chapman & Hall/CRC, 2010.

[Ami01]   E. Amir. Efficient approximation for triangulation of minimum treewidth. In *Proc. of UAI*, pages 7–15, San Francisco, USA, 2001. Morgan Kaufmann Publishers Inc.

[BDH⁺12] D. Berwanger, A. Dawar, P. Hunter, S. Kreutzer, and J. Obdržálek. The DAG-width of directed graphs. *J. Comb. Theory, Ser. B*, 102(4):900–923, 2012.

[BDHK06] D. Berwanger, A. Dawar, P. Hunter, and S. Kreutzer. DAG-width and parity games. In *Proc. of STACS*, pages 524–436. Springer, 2006.

[BG04]   D. Berwanger and E. Grädel. Entanglement - a measure for the complexity of directed graphs with applications to logic and games. In *Proc. of LPAR*, pages 209–223. Springer, 2004.

[Bod96]   H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, December 1996.

[EJ91]   E. A. Emerson and C. S. Jutla. Tree automata, μ-calculus and determinacy. In *Proc. of FOCS*, pages 368–377. IEEE Computer Society Press, October 1991.

[EJS93]   E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of μ-calculus. In *CAV*, pages 385–396, 1993.

[FL11]   J. Fearnley and O. Lachish. Parity games on graphs with medium tree-width. In *Proc. of MFCS*, pages 303–314. Springer, 2011.

[HK08]   P. Hunter and S. Kreutzer. Digraph measures: Kelly decompositions, games, and orderings. *Theor. Comput. Sci.*, 399(3):206–219, 2008.

[JPZ08]   M. Jurdzinski, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. *SIAM J. Comput.*, 38(4):1519–1532, 2008.

[McN93]   R. McNaughton. Infinite games played on finite graphs. *Ann. Pure Appl. Logic*, 65(2):149–184, 1993.

[Mos91]   A. W. Mostowski. Games with forbidden positions. Technical Report 78, University of Gdańsk, 1991.

[Obd03]   J. Obdržálek. Fast mu-calculus model checking when tree-width is bounded. In *Proc. of CAV*, pages 80–92. Springer, 2003.

[Obd06]   J. Obdržálek. Dag-width: connectivity measure for directed graphs. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 814–821. ACM Press, 2006.

[Obd07]   J. Obdržálek. Clique-width and parity games. In *Proc. of CSL*, pages 54–68, 2007.

[RS84]   N. Robertson and P. D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[Sti95]   C. Stirling. Local model checking games (Extended abstract). In *Proc. of CONCUR*, pages 1–11. Springer-Verlag, 1995.

[Tho98]   M. Thorup. All structured programs have small tree width and good register allocation. *Information and Computation*, 142(2):159 – 181, 1998.