A technique to speed up symmetric attractor-based algorithms for parity games

K. S. Thejaswini ✉
Department of Computer Science, University of Warwick

Pierre Ohlmann ✉
University of Warsaw, Poland

Marcin Jurdziński ✉
Department of Computer Science, University of Warwick

Abstract

The classic McNaughton-Zielonka algorithm for solving parity games has excellent performance in practice, but its worst-case asymptotic complexity is worse than that of the state-of-the-art algorithms. This work pinpoints the mechanism that is responsible for this relative underperformance and proposes a new technique that eliminates it. The culprit is the wasteful manner in which the results obtained from recursive calls are indiscriminately discarded by the algorithm whenever subgames on which the algorithm is run change. Our new technique is based on firstly enhancing the algorithm to compute attractor decompositions of subgames instead of just winning strategies on them, and then on making it carefully use attractor decompositions computed in prior recursive calls to reduce the size of subgames on which further recursive calls are made. We illustrate the new technique on the classic example of the recursive McNaughton-Zielonka algorithm, but it can be applied to other symmetric attractor-based algorithms that were inspired by it, such as the quasi-polynomial versions of the McNaughton-Zielonka algorithm based on universal trees.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Design and analysis of algorithms; Theory of computation → Logic and verification

Keywords and phrases Parity games, Attractor decomposition, Quasipolynomial Algorithms, Universal trees

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2022.32

Related Version The full version with proofs will be available on arXiv [30]

Funding EPSRC grant EP/P020992/1 (Solving Parity Games in Theory and Practice).
Pierre Ohlmann: Project BOBR that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 948057)

Acknowledgements We thank Rémi Morvan for contributing to several simulating discussions. We are grateful to Aditya Prakash for proof reading the current version of the paper, and to Nathanaël Fijalkow and Olivier Serre for doing the same with an earlier version of the paper. We also thank anonymous referees for pointing out some missing references from our previous draft as well as for their valuable comments to improve our current presentation. The authors are listed in reverse alphabetical order.

1 Context and contributions

Parity games are two-player games on graphs, which have been studied since early 1990's [10, 11] and have many applications in automata theory on infinite trees [15], fixpoint logics [9, 4], verification and synthesis [28, 29]. They are intimately linked to the problems of emptiness and complementation of nondeterministic automata on trees [10, 33], model checking [11, 4, 16] and satisfiability checking of fixpoint logics, or fair simulation relations [12].

Determining the winner of a parity games are one of the few problems known to lie in the complexity class $\mathsf{NP} \cap \mathsf{coNP}$ as well as $\mathsf{UP} \cap \mathsf{coUP}$ but not known to have a polynomial algorithm. Existence of a polynomial algorithm for solving parity games, which has been an important open problem for nearly three decades, has recently gained a lot of attention after the major breakthrough of Calude, Jain, Khoussainov, Li and Stephan [5], who provided a quasi-polynomial solution to the problem. Several algorithms [18, 23, 27, 24] followed this work, each providing a different perspective to solving parity games. Czerwiński, Daviaud, Fijalkow, Jurdziński, Lazić, and Parys [6] originally exhibited an underlying combinatorial structure of universal trees, provably underlying the techniques of Calude et al. of Jurdziński and Lazić, and of Lehtinen. Czerwiński et al. have also established a quasi-polynomial lower bound for the size of smallest universal trees, providing evidence that the techniques developed in these papers may be insufficient for leading to further improvements in the complexity of solving parity games. Following their work, Jurdziński, Morvan and Thejaswini [20] extended this result to formulate attractor based algorithms, as that of Parys [27] and also Lehtinen, Schewe, and Wojtczak [24] using universal trees too.

We propose a new technique for speeding up symmetric attractor-based algorithms for solving parity games. We focus on illustrating the technique on the example of the classic McNaughton-Zielonka algorithm [33], but we argue that it is applicable to other symmetric attractor-based algorithms that were inspired by McNaughton-Zielonka [1, 27, 25, 20, 22]. Many such algorithms have exhibited excellent performance in practice, significantly beating other classes of algorithms on standard benchmarks [21, 31]. On the other hand, their worst-case asymptotic running time is typically worse than that of asymmetric algorithms [17, 5, 18, 8]. More specifically, using the perspective of how various algorithms for parity games are related to universal trees [6], while the running time of state-of-the-art asymmetric algorithms is dominated by the size of a universal tree [18, 8], it is the square of the size of a universal tree for symmetric algorithms [20, 25]. Our technique allows to reduce the worst-case running time of symmetric attractor-based algorithms to match the linear dependence on the size of a universal tree enjoyed by asymmetric algorithms.

Our technique is based on making a better use of structural information obtained from earlier recursive calls to significantly reduce the worst-case overall size of the tree of recursive calls of the algorithm. While existing symmetric attractor-based algorithms are typically computing just the winning sets of positions or positional winning strategies, following [20], we propose to enhance them to explicitly record more finely structured witnesses of winning strategies called attractor decompositions. Moreover, we show how witnesses for both players from recursive calls on subgames can be meaningfully used to reduce the sizes of subgames on which further recursive calls are made, even if their key properties are damaged by the removal of some vertices from subgames on which they were computed. In contrast, other symmetric attractor-based algorithms are wasteful by routinely discarding witnesses for one of the players that are computed in recursive calls; in the worst case, this results in repeatedly solving large subgames from scratch. Our technique is robust and it applies to both the classic exponential-time McNaughton-Zielonka algorithm [33] and its more recent quasi-polynomial variants [27, 20, 25]. We are also confident that it is applicable to other symmetric attractor-based algorithms such as priority promotion [1]. Such algorithms can be interpreted as variants of the McNaughton-Zielonka algorithm that are enhanced by ad-hoc heuristics to construct attractor decompositions which are more robust to the wasteful behaviour described above. Our technique offers a more principled approach, in which decompositions of subgames computed in previous recursive calls are never discarded and are instead used in a systematic manner to speed up and reduce the number of further recursive calls.

2 Games, Strategies, Attractor decomposition

Parity Games A parity game $\mathcal{G}$ consists of a finite directed graph (V, E), where the vertices are partitioned into V_{Even} and V_{Odd} where these belong to the Even player and the Odd player respectively along with a mapping π, from V to the set $\{1, \ldots h\}$ that labels every vertex with a positive integer, called its priority. A token is moved from a designated start vertex into a neighbour by Even or Odd depending on who owns the vertex, forming a sequence of vertices, which we will a *play*. An infinite play is said to be winning for Even if the highest priority that occurs infinitely often is Even and Odd wins otherwise.

A (positional) *strategy* $\sigma \subseteq E$ for Even is a subset of edges originating from Even owned vertices. A game restricted to an Even strategy refers to the sub graph induced by considering only edges in σ along with edges in which originate in Odd's vertex. We call a cycle in a game $\mathcal{G}$ an even cycle if the highest priority of the cycle is Even. A strategy is said to be winning for Even from a vertex v, if the game restricted to the Even strategy σ is such that the only cycles reachable from v are such that the highest priority in the cycle is even.

Attractors, traps, and dominions In a parity game $\mathcal{G}$, for a target set of vertices B and a set of vertices A such that $B \subseteq A$, we say that an Even strategy σ is an Even reachability strategy to B from A if every infinite path in the subgraph restricted to σ along with all edges from all Odd vertices, that starts from a vertex in A contains at least one vertex in B. We call the largest such set A for which there is a reachability strategy, the *Even attractor* of B and we denote it by $\text{Attr}^{\mathcal{G}}_{\text{Even}}(B)$.

A trap for Odd is a subgame T of $\mathcal{G}$, where Even has a strategy σ, on restricted to which all paths from T remain in T. A subgame D in a game $\mathcal{G}$ is said to be an *Even dominion* if it is an Odd trap and moreover, every cycle that can be reached from any vertex in D using the strategy used to trap Odd is an even cycle.

We also say that a set of vertices C is a *quasi-dominion* for Even if the subgame $\mathcal{G} \cap C$ is winning for Even. Note that all dominions of Even are also quasi-dominions but not all quasi-dominions of Even are dominions, since Odd might be able to escape a quasi-dominion.

If we do not mention if an object defined is for Even or for Odd, we assume Even by default. All of the above can be defined analogously for the other player.

Trees Throughout this paper, trees only refer to rooted ordered trees. For a totally ordered set Σ, an *ordered tree* $\mathcal{T}$ is a finite prefix closed set consisting of sequences of elements from Σ. We say that the *root* of the tree is the empty sequence $\langle\rangle$. We call an element of this prefix closed set, a *node* and use $\eta, \gamma, \epsilon \ldots$ to refer to it. The leaves of a tree are the maximal elements of $\mathcal{T}$. For a node $\eta \in \mathcal{T}$, the subtree rooted at η is the tree $\eta^{-1} \cdot \mathcal{T}$. We say η' is an element of the subtree rooted at η if η' in $\mathcal{T}$ can be obtained by extending η with a sequence from Σ. For a node η that is not a leaf, the *children* of η are elements η_i such that $\eta_i = \eta \cdot \langle a \rangle$ for $a \in \Sigma$. The height of a tree is defined as the maximum length sequence in it. A leaf has height 1, and every node has a height one more than any of its children. We separately define levels of nodes in a tree for Even and Odd inductively as follows. An *Even level* of a node is 2 if it is a leaf, and it is two more than its children's Even level if it is not a leaf. The Odd level is defined similarly, starting at level 1 for leaves. Note that the height of a tree is at most half of either the Even or Odd level.

We only consider trees such that the children of each node all have the same level. For any node, we usually use the same variable with subscripts to list their children in order i.e., for η, we use $\eta_1, \ldots, \eta_k$ to denotes its first k children

A complete n-ary tree of height $2h$, is the tree where each node has n children and the height of the root is $2h$. For this an n-ary tree, we use $\eta \cdot \langle i \rangle$ to denote the i^{th} child of η. A tree is (n, h)-universal if there is an order preserving injective homomorphism from any tree of height h and with n leaves into it.

Attractor decomposition A structurally simplest quasi-dominion is an *atomic quasi-dominion*: it is a set $T \cup H$ such that T is an Even attractor to H in $T \cup H$, and all vertex priorities in H are even and larger than all vertex priorities in T. It is a quasi-dominion because if Even uses an attractor strategy in T and an arbitrary strategy in H, then a play that stays in $T \cup H$ forever visits set H infinitely many times and hence it is winning for Even. Consider the following three ways of composing quasi-dominions into structurally more complex ones.

- If Q_1 and Q_2 are quasi-dominions and Q_1 is a trap for Odd in $Q_1 \cup Q_2$, then $Q_1 \cup Q_2$ is a quasi-dominion. We say that $Q_1 \cup Q_2$ is a *sequential composition* of Q_1 and Q_2.
- If Q is a quasi-dominion, S is an attractor to Q in $Q \cup S$, and Q is a trap for Odd in $Q \cup S$, then $Q \cup S$ is a quasi-dominion. We say that $Q \cup S$ is a *side-attractor composition* of Q and S.
- If Q is a quasi-dominion, T is an attractor to H in $Q \cup (T \cup H)$, and if all vertex priorities in H are even and larger than all vertex priorities in $Q \cup T$, then $Q \cup (T \cup H)$ is a quasi-dominion. We say that $Q \cup (T \cup H)$ is a *top-attractor composition* of Q and $T \cup H$, and that T is a *top attractor* to H.

Henceforth, when we say that a quasi-dominion can be obtained by sequential, side-attractor, or top-attractor composition, we implicitly assume that the suitable applicability conditions listed above hold.

Note that an atomic Even quasi-dominion is a special case of the top-attractor composition, in which $Q = \emptyset$. A sequential composition is a quasi-dominion because if Even uses quasi-dominion strategies in Q_1 and Q_2, then every play that stays in $Q_1 \cup Q_2$ forever, either stays in Q_2 forever, or it eventually stays in Q_1 forever, and hence it is winning for Even. A side-attractor composition is a quasi-dominion because if Even uses quasi-dominion strategy in Q and an attractor strategy in S, then every play that stays in $Q \cup S$ forever, eventually stays in Q forever, and hence it is winning for Even. A top-attractor composition is a quasi-dominion because if Even uses a quasi-dominion strategy in Q, an attractor strategy in T, and an arbitrary strategy in H, then every play that stays in $Q \cup (T \cup H)$ forever, either eventually stays in Q forever, or it visits set H infinitely many times, and hence it is winning for Even.

It is folklore that the three composition operations are complete in the sense that every quasi-dominion can be obtained by a sequence of such composition operations [26, 33]. More specifically, the following concept of an attractor decomposition [7, 20] is a technically convenient normal form, in which the hierarchical structure of quasi-dominions resulting from the composition operations is captured by an ordered tree.

▶ **Definition 1** (Attractor decomposition). *An attractor decomposition of a game $\mathcal{G}$ consists of an ordered tree, and for each node ϵ of the tree, three mutually disjoint sets of vertices H^ϵ (high even priority set), T^ϵ (top attractor), and S^ϵ (side attractor) that satisfy the following conditions. Note that these vertices can be potentially empty for some nodes ϵ.*

1. *The root node has some even level, and the children of every node at level k have level $k-2$. For every node ϵ, if its level is k, then all vertex priorities in H^ϵ are k, all vertex priorities in T^ϵ are at*

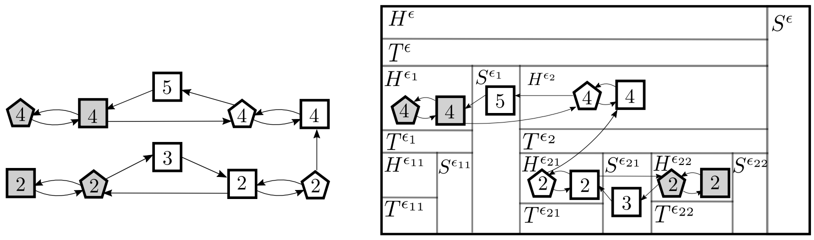

Figure 1 On the left, a game where Even wins from every vertex and on the right, its attractor decomposition

most $k-1$, and all vertex priorities in S^ϵ are at most $k+1$.

2. *If node ϵ is a leaf, then $T^\epsilon \cup H^\epsilon$ is an atomic quasi-dominion, and quasi-dominion Q^ϵ can be obtained by side-attractor composition of $T^\epsilon \cup H^\epsilon$ and S^ϵ.*
3. *If ϵ is node that is not a leaf, then quasi-dominion Q^ϵ can be obtained in the following way. Firstly, quasi-dominion P can be obtained by the sequential composition of quasi-dominions Q^η, where η ranges over the children of ϵ in the tree order. Then, quasi-dominion R can be obtained by the top-attractor composition of P and $T^\epsilon \cup H^\epsilon$. Finally, quasi-dominion Q^ϵ can be obtained by the side-attractor composition of R and S^ϵ.*

Consider an example of a game and an attractor decomposition described in Figure 1. The priorities are written in the vertices. Player Even owns all square vertices and Odd owns all the pentagons. Some vertices are shaded to be able to identify each vertex in the game uniquely. In this game, Even has a strategy to win from everywhere. An attractor decomposition of this game is depicted to its right. The nodes of the tree are depicted by $\epsilon, \epsilon_1, \epsilon_2, \epsilon_{11}, \epsilon_{21}$ and ϵ_{22}. The corresponding sets are depicted in the figure with some of them being empty. This attractor decomposition satisfies properties 1, 2 and 3 in Definition 1.

McNaughton-Zielonka algorithm We recall the classical recursive algorithm of McNaughton and Zielonka [26, 33]. However, this is not similar to a description of McNaughton and Zielonka one would find in the wild. It is enhanced to produce attractor decompositions for both the players and also return the winning sets for players Even and Odd. This is accomplished in Algorithm 1 where the attractor decomposition produced is similar to the one defined before. Note that the work of Jurdziński, Morvan and Thejaswini [20] also produce attractor decompositions explicitly for McNaughton and Zielonka.

The algorithm uses two mutually recursive calls, $\texttt{McNZ}_{\text{Even}}$ and $\texttt{McNZ}_{\text{Odd}}$ which takes as input a game $\mathcal{G}$, the highest priority h. We also further include as input to these calls, two nodes ϵ and ω where ϵ belongs to an Even n-ary tree and ω to an Odd n-ary tree. The respective levels of these nodes in the tree are h and $h+1$. These trees are used to store the structure of the quasi-dominions obtained in the algorithm thus far. Moreover, we assume that the algorithm has access to the recursive structure of the quasi-dominions computed by it in previous recursive subcalls and can access the sets corresponding to this quasi-dominion with the help of the nodes of the tree. The quasi-dominions computed up until a point in the algorithm are stored as a disjoint partition of sets, based on the underlying complete tree. These disjoint sets are represent with H^ϵ_{Even}, T^ϵ_{Even} and S^ϵ_{Even} for each node ϵ belonging to the n-ary tree used for Even tree and H^ω_{Odd}, T^ω_{Odd} and S^ω_{Odd} for ω belonging to the Odd tree. We exclusively use ϵ and its variants for Even trees and ω and its variants for Odd trees to avoid confusion. The statement $S^{\omega\cdot\langle i\rangle}_{\text{Odd}} \leftarrow A'_i \setminus U'_i$ performs the side attractor composition for Odd and the statements $H^\epsilon_{\text{Even}} \leftarrow H_i$, and $T^\epsilon_{\text{Even}} \leftarrow A_i \setminus H_i$ together perform a top attractor composition for the Even attractor decomposition.

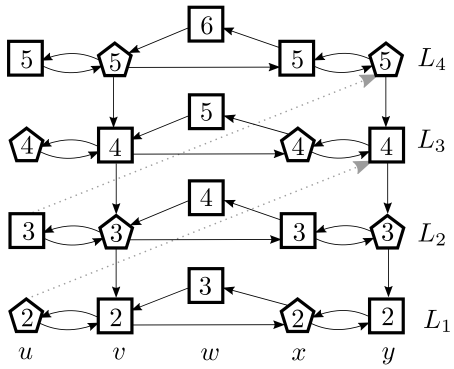

Figure 2 The game $\mathscr{H}_4$

Algorithm 1 The McNaughton-Zielonka Algorithm

procedure $\texttt{McNZ}_{\text{Even}}(\mathscr{G}, h, \epsilon, \omega)$**:**
  if $h = 0$ **then**
    return $\emptyset$
  $\mathscr{G}_1 \leftarrow \mathscr{G}$, $i = 0$
  repeat
    $i \leftarrow i + 1$
    $H_i \leftarrow \pi^{-1}(h) \cap \mathscr{G}_i$
    $A_i \leftarrow \mathrm{Attr}^{\mathscr{G}_i}_{\text{Even}}(H_i)$
    $\mathscr{G}'_i \leftarrow \mathscr{G}_i \setminus A_i$
    $U'_i \leftarrow$
     $\texttt{McNZ}_{\text{Odd}}(\mathscr{G}'_i, h-1, \omega \cdot \langle i \rangle, \epsilon)$
    $A'_i \leftarrow \mathrm{Attr}^{\mathscr{G}_i}_{\text{Odd}}(U'_i)$
    $S^{\omega \cdot \langle i \rangle}_{\text{Odd}} \leftarrow A'_i \setminus U'_i$
    $\mathscr{G}_{i+1} \leftarrow \mathscr{G}_i \setminus A'_i$
  until $\mathscr{G}_i = \mathscr{G}_{i+1}$
  $H^{\epsilon}_{\text{Even}} \leftarrow H_i$
  $T^{\epsilon}_{\text{Even}} \leftarrow A_i \setminus H_i$
  return $V(\mathscr{G}_i)$

We only describe the Even recursive call since the Odd recursive subcall can be described similarly. The correctness of the above algorithm are well known and we refer an interested reader to several pre-existing works [26, 33, 19, 20] proving the correctness of the algorithm.

Hard examples for McNaughton and Zielonka The McNaughton-Zielonka algorithm outperforms several other algorithms in practice, but there are several families of games on which it takes exponential time. Some examples are those found in the paper of Friedmann [13], Gzada and Willemse [14] and van Dijk [31], Benerecetti et al [2]. We add to this list, a family of games on which McNaughton-Zielonka makes exponentially many recursive subcalls. We focus our attention in this section to this family and use it as a running example to highlights the exponential complexity of McNaughton-Zielonka and to motivate the idea behind our technique in the following sections.

Consider the following family of games $\mathscr{H}_k$ for each k which we define below. The game $\mathscr{H}_k$ consists of $5k$ vertices, with the highest priority being $k+2$. For each i at most k, the vertex set consists of 5 vertices, and they are $\{u_i, v_i, w_i, x_i, y_i\}$. We call this set L_i and refer to it as the i^{th} layer of the game. The priority of w_i is $i+2$ and all the other nodes in L_i have priority $i+1$. For even values of i, Odd owns v_i, y_i and Even owns u_i, x_i. For odd values of i, this is swapped. The ownership of w_i is irrelevant. The edges within a layers L_i are: $\{(u_i, v_i), (v_i, u_i), (v_i, x_i), (x_i, w_i), (w_i, v_i), (x_i, y_i), (y_i, x_i)\}$. Between layers,

- for each $i \leqslant k-2$, there is an edge (u_i, y_{i+2});
- for each $1 < i \leqslant k$, there are edges (v_i, v_{i-1}) and (y_i, y_{i-1}).

An example of the game $\mathscr{H}_4$ is shown in Figure 2, where square vertices are owned by Even and pentagon by Odd. The odd layers in the game are winning for Even, whereas the Even layers are winning for Odd. A strategy witnessing the above of each player is for a player to move to the vertex to their left. More formally, for each player and an appropriate j, the edges (v_j, u_j) and (y_j, x_j) turns out to be a winning strategy in their respective dominions.

▶ **Lemma 2.** *For the family of games $\mathcal{H}_n$ for $n \geqslant 1$ Algorithm 1 makes $O(2^n)$ recursive subcalls to* $\texttt{McNZ}_{\text{Even}}$ *and* $\texttt{McNZ}_{\text{Odd}}$

We provide the proof for the above in the full version of the paper but restrict ourselves to a discussion to highlight the idea behind the exponential complexity of McNaughton-Zielonka. For an odd value k, the procedure $\texttt{McNZ}_{\text{Even}}$ on $\mathcal{H}_k$ makes two $\texttt{McNZ}_{\text{Odd}}$ subcalls and these are on the subgames $\mathcal{H}_{k-1} \cup \{u_k, v_k, y_k\}$ and $\mathcal{H}_{k-1} \cup \{x_k, y_k\}$ in succession. Notice that these games have a large intersection, which includes $\mathcal{H}_{k-1}$ and the vertex y_k, leading to an exponential complexity of this easy-to-describe algorithm. The first recursive call $\mathcal{H}_{k-1} \cup \{u_k, v_k, y_k\}$ indeed identifies winning sets for this subgame for both players. Moreover, for the Even player, the strategies that are winning in this subgame is in fact winning in $\mathcal{H}_k$ too. Unfortunately, the next recursive subcall promptly discards this information.

It is natural to ask if there is some way we can utilise the progress we make in the first recursive subcall to provide a head start for the following recursive subcalls. Several enhancements of McNaughton-Zielonka exist, all of which attempt to utilise some information from recursive subcalls. A notable few include the Priority promotion algorithms and its variants [1, 3] along with the Tangle learning algorithms by van Dijk [31], which all perform well in practice. The key idea behind these algorithms is to identify quasi-dominions in their recursive subcalls and increase the quasi-dominions obtained so far carefully. Another idea was to remember the strategy obtained in the recursive subcall. In a recent work by Lapauw, Bruynooghe, and Denecker [22], they show that by remembering and modifying strategies obtained recursively in a calculated manner, they too obtained faster practical performances. But all of these have worst case exponential running time comparable to McNaughton-Zielonka, as these heuristics do not lead to a provable increase in the running time of either of these algorithms.

We propose a technique in the next section, which remembers some information obtained from the structure of the attractor decompositions computed from previous recursive subcalls. This turns out to provide a quadratic gain in the worst case runtime complexity.

3 Making McNaughton-Zielonka faster with some memory

The crucial object that we will be dealing with for the rest of this paper are *decompositions*, which forms the central theme of this section. We define this object as a generalisation of an attractor decomposition. With the help of our definition, we carefully modify the classical algorithm to make at most $O\left(\left(\frac{n}{h}\right)^{h/2}\right)$ many recursive calls. This is comparable to the runtime of the first progress measure algorithm by Jurdziński [17]. For the proof of correctness and runtime of the algorithm, we need the machinery that we develop in Section 4. This mathematical tool-kit helps accurately pin point the guarantees of the algorithm and also analyse the running time. We however state these guarantees in this Section and with these guarantees use them to demonstrate how our algorithm works faster on families of games which are hard for several attractor based algorithms. We show that this modification to McNaughton-Zielonka makes it run in polynomial time for two specific families.

A relaxation of attractor decomposition

Recall the procedure $\texttt{McNZ}_{\text{Even}}$ on example $\mathcal{H}_k$ from Section 2. This procedure makes two recursive subcalls of $\texttt{McNZ}_{\text{Odd}}$ to subgames each containing $\mathcal{H}_{k-1}$. Notice that although $\texttt{McNZ}_{\text{Odd}}$ returned an attractor decomposition on the first iteration, we discarded the Even attractor decomposition obtained from this recursive subcall until an empty Odd dominion was returned by the recursive subcall. In fact, this algorithm on any game repeatedly discards this information until U_i' returned is empty.

Indeed, this wasteful discarding of attractor decompositions seems necessary to prove correctness at first sight. On removing vertices from an Even attractor decomposition it might no longer satisfy all the properties of an attractor decomposition. Since after each recursive subcall, we remove a non-empty Odd attractor A_i' from it, it seems natural that this information must now be discarded. To circumvent this wastage, we propose to store an object that loosely resembles an attractor decomposition but with the requirements about quasi-dominions and attractors relaxed. Below, we define an Even decomposition to closely resemble our definition of attractor decomposition from the previous section.

▶ **Definition 3** (Decomposition). *A decomposition consists of an ordered tree, and for each node ϵ of the tree, three mutually disjoint sets of vertices H^ϵ (high even priority set), T^ϵ (top set), and S^ϵ (side set) that satisfy only the following condition.*

- *The root node has some even level, and the children of every node at level k have level $k-2$. For every node t, if its level is k, then all vertex priorities in H^ϵ are k, all vertex priorities in T^ϵ are at most $k-1$, and all vertex priorities in S^ϵ are at most $k+1$.*

Notice that this definition is more general than an attractor decomposition. A decomposition which satisfies the attractor composition properties (items 2 and 3) in Definition 1 is an attractor decomposition. More importantly, since a decomposition no longer needs to satisfy properties about traps and attractors, on restricting it to a subset of vertices, it still remains a decomposition. This helps us maintain this structure even after the algorithm removes some vertices from it.

A faster algorithm

The modification we propose for McNaughton-Zielonka is described in Algorithm 2. This algorithm, at a high level resembles Algorithm 1. The significant difference is that it starts with a decomposition and modifies it by calling recursive subcalls until this decomposition is an attractor decomposition.

For a parity game $\mathcal{G}$, the algorithm maintains two decompositions with n-ary trees for both the players. This is done globally and refers to the sets associated to these decompositions using the nodes of the tree. We refer to decompositions $\mathcal{D}^\epsilon_{\text{Even}}$ or $\mathcal{D}^\omega_{\text{Odd}}$ for ϵ and ω which are nodes in the Even and Odd tree respectively. Note that although we use $\mathcal{D}^\epsilon_{\text{Even}}$ or $\mathcal{D}^\omega_{\text{Odd}}$, these are two distinct decompositions. Moreover, we use ϵ along with Even in the subscript, along with its variations like $\epsilon', \epsilon_i, \ldots$ for nodes in the Even tree and similarly for Odd with ω, this enables us to refer to both the Even and Odd decompositions without confusion.

We call the partitions in these decomposition H^ϵ_{Even}, T^ϵ_{Even} and S^ϵ_{Even} for ϵ, a node in the Even tree, and H^ω_{Odd}, T^ω_{Odd} and S^ω_{Odd} for ω in the Odd tree. We use $[\mathcal{D}^\epsilon_{\text{Even}}]$ to denote the set of of vertices associated with some node in the subtree of the tree rooted at ϵ but without the vertices in S^ϵ_{Even}. More formally, for a decomposition $\mathcal{D}^\epsilon_{\text{Even}}$, we define $[\mathcal{D}^\epsilon_{\text{Even}}]$ inductively, where

- $[\mathcal{D}^\epsilon_{\text{Even}}]$ consists of $H^\epsilon \cup T^\epsilon$ if ϵ is a leaf;
- $[\mathcal{D}^\epsilon_{\text{Even}}]$ consists of $H^\epsilon_{\text{Even}} \cup T^\epsilon_{\text{Even}}$ along with all vertices in $[\mathcal{D}^{\epsilon_i}_{\text{Even}}] \cup S^{\epsilon_i}_{\text{Even}}$ for ϵ_i ranging over the children of ϵ.

The procedure $\texttt{McNZFast}_{\text{Even}}$ in Algorithm 2 works using decompositions $\mathcal{D}^\epsilon_{\text{Even}}$ and $\mathcal{D}^\omega_{\text{Odd}}$ on a subgame $\mathcal{G}$. We modify these decompositions with operations like 'Set' and 'Move'. We only give an intuition of the operations here, but we make these precise in the appendix of the paper. The highest priority in the game $\mathcal{G}$ is at most the level of Even tree's root: ϵ. Moreover, the level of Odd tree's root ω is one more than the level of ϵ.

Algorithm 2 McNaughton and Zielonka Algorithm with memory

procedure $\texttt{McNZFast}_{\text{Even}}(\mathcal{G}, h, \epsilon, \omega)$:
- **if** $\mathcal{D}^{\epsilon}_{\text{Even}}$ *restricted to* $\mathcal{G}$ *is an attractor decomposition* **then**
  - Set S^{ω}_{Odd} to $V(\mathcal{G})$
  - **return** $V(\mathcal{G})$
- **else if** $\mathcal{D}^{\omega}_{\text{Odd}}$ *restricted to* $\mathcal{G}$ *is an attractor decomposition* **then**
  - Set $S^{\epsilon}_{\text{Even}}$ to $V(\mathcal{G})$
  - **return** $\emptyset$
- **else**
  - $\mathcal{G}_1 \leftarrow \mathcal{G}$; $i = 0$
  - **repeat**
    - $i \leftarrow i + 1$
    - $H_i \leftarrow \pi^{-1}(h) \cap \mathcal{G}_i$
    - $T_i \leftarrow \text{Attr}^{\mathcal{G}_i}_{\text{Even}}(H_i)$
    - Set $T^{\epsilon}_{\text{Even}}$ to $T_i \setminus H_i$
    - $S_i \leftarrow \text{Attr}^{\mathcal{G}_i}_{\text{Even}}\left(\mathcal{G}_i \setminus \left[\mathcal{D}^{\omega\cdot\langle i\rangle}_{\text{Odd}}\right]\right)$
    - Move S_i to at least $S^{\omega\cdot\langle i\rangle}_{\text{Odd}}$
    - $\mathcal{G}'_i \leftarrow (\mathcal{G}_i \setminus S_i)$
    - $U'_i \leftarrow$ $\texttt{McNZFast}_{\text{Odd}}(\mathcal{G}'_i, h-1, \epsilon, \omega\cdot\langle i\rangle)$
    - $S'_i \leftarrow \text{Attr}^{\mathcal{G}_i}_{\text{Odd}}(U'_i)$
    - Set $S^{\omega\cdot\langle i\rangle}_{\text{Odd}}$ to $S'_i \setminus U'_i$
    - Move S'_i to at least $S^{\epsilon}_{\text{Even}}$
    - $\mathcal{G}_{i+1} \leftarrow \mathcal{G}_i \setminus S'_i$
  - **until** $\mathcal{D}^{\epsilon}_{\text{Even}}$ *restricted to* $\mathcal{G}_{i+1}$ *is an attractor decomposition*
  - **return** $V(\mathcal{G}_i)$

Algorithm 3 Universal Attractor Decomposition algorithm with memory

procedure $\texttt{UnivFast}_{\text{Even}}(\mathcal{G}, h, \epsilon, \omega)$:
- **if** $\mathcal{D}^{\epsilon}_{\text{Even}}$ *is an attractor decomposition for* $\mathcal{G}$ **then**
  - Set S^{ω}_{Odd} to $V(\mathcal{G})$
  - **return** $V(\mathcal{G})$
- **else if** $\mathcal{D}^{\omega}_{\text{Odd}}$ *is an attractor decomposition for* $\mathcal{G}$ **then**
  - Set $S^{\epsilon}_{\text{Even}}$ to $V(\mathcal{G})$
  - **return** $\emptyset$
- **else**
  - $\mathcal{G}_1 \leftarrow \mathcal{G}$
  - Let $\omega_1, \ldots, \omega_k$ be children of ω in the Odd tree
  - **for** $i \leftarrow 1$ **to** k **do**
    - $H_i \leftarrow \pi^{-1}(h) \cap \mathcal{G}_i$
    - $T_i \leftarrow \text{Attr}^{\mathcal{G}_i}_{\text{Even}}(H_i)$
    - Set $T^{\epsilon}_{\text{Even}}$ to $T_i \setminus H_i$
    - $S_i \leftarrow \text{Attr}^{\mathcal{G}_i}_{\text{Even}}\left(\mathcal{G}_i \setminus \left[\mathcal{D}^{\omega_i}_{\text{Odd}}\right]\right)$
    - Move S_i to at least $S^{\omega_i}_{\text{Odd}}$
    - $\mathcal{G}'_i \leftarrow (\mathcal{G}_i \setminus S_i)$
    - $U'_i \leftarrow$ $\texttt{UnivFast}_{\text{Odd}}(\mathcal{G}'_i, h-1, \epsilon, \omega_i)$
    - $S'_i \leftarrow \text{Attr}^{\mathcal{G}_i}_{\text{Odd}}(U'_i)$
    - Set $S^{\omega_i}_{\text{Odd}}$ to $S'_i \setminus U'_i$
    - Move S'_i to at least $S^{\epsilon}_{\text{Even}}$
    - $\mathcal{G}_{i+1} \leftarrow \mathcal{G}_i \setminus S'_i$
  - **return** $V(\mathcal{G}_{i+1})$

If the decompositions computed so far already forms an attractor decomposition for either player, the algorithm stops and returns the corresponding set, since having an attractor decomposition implies that we have a witness of winning for either player in the current subgame. We claim that one can check if a decomposition is an attractor decomposition efficiently, in polynomial time in the number of vertices of the game rather than the size of the tree, but postpone the details on how until the next section.

On the other hand, if both players only have decompositions that are not attractor decompositions on the current subset of vertices, we first compute the Even attractor to the top priority in the current subgame, closely mirroring Algorithm 1. However, we deviate from McNaughton-Zielonka, by updating this information by modifying the partition $T^{\epsilon}_{\text{Even}}$ of the Even decomposition.

The line Set $T^{\epsilon}_{\text{Even}}$ to $T_i \setminus H_i$ results in the current decomposition of even being modified so that the 'top set' $T^{\epsilon}_{\text{Even}}$ now contains exactly the vertices $T_i \setminus H_i$, which could be attracted to the set of vertices of top priority H_i in the current subgame. Doing this updates requires modification of the decomposition to relocate the other vertices that were previously at $T^{\epsilon}_{\text{Even}}$ to sets associated to the children of ϵ.

In Algorithm 1, the next recursive subcall would work on the complement of the attractor set T_i. However, for this algorithm, the next recursive subcall at one level lower, we considers a subset of the set $\left[\mathscr{D}^{\omega\cdot\langle i\rangle}_{\text{Odd}}\right]$. Since this need not be a subgame, the procedure computes the Even attractor to all vertices not in this set. The complement of this Even attractor results in S_i, a trap for Even. Intuitively, the next line: Move S_i to at least $S^{\omega\cdot\langle i\rangle}_{\text{Odd}}$ modifies the decomposition such that in the new decomposition obtained, the vertices in S_i are no longer in the sets at $\left[\mathscr{D}^{\omega\cdot\langle i\rangle}_{\text{Odd}}\right]$. Instead, these vertices are such that assigned the vertices in S_i to the sets corresponding to either move these vertices to $S^{\omega\cdot\langle i\rangle}_{\text{Odd}}$ in the Odd decomposition. The other partitions in the decomposition are left unchanged.

We then perform an Odd recursive subcall, by calling the procedure $\texttt{McNZFast}_{\text{Odd}}$ on $\mathscr{G}'_i$, the trap for Even obtained above. After the Odd recursive subcall, which returns the Odd winning set U'_i in the subgame $\mathscr{G}'_i$, we compute the Odd attractor to this set which is S'_i. Since S'_i is the set of vertices from which Odd has a strategy to reach U'_i, we adjust the Odd decomposition such that the side set $S^{\omega\cdot\langle i\rangle}_{\text{Odd}}$ is exactly $S'_i \setminus U'_i$ and all vertices in S'_i are then relocated to $S^{\epsilon}_{\text{Even}}$ for the Even decomposition. This is captured by the line Move S'_i to at least $S^{\epsilon}_{\text{Even}}$. This is done similarly to the above move operator, where all vertices in S'_i in the newly obtained decomposition

Now, all we need to explain is how we initialise these decompositions for the algorithm. The following definition seem technical, although intuitively, we start with an 'optimistic' decomposition, which starts with the simplest structure of a decomposition for both Odd and Even. We call the *initial Even decomposition* for the game $\mathscr{G}$ with highest priority h, where the Even node at level h is ϵ, and the Odd node at level $h+1$ is ω, as the following:

- all vertices of priority h are at $H^{\epsilon}_{\text{Even}}$ for Even.
- all vertices with priority strictly smaller than $h-1$ are at $T^{\epsilon}_{\text{Even}}$ for Even.

Similarly, the *initial Even decomposition* is defined as

- all vertices of priority h are at S^{ω}_{Odd} for Odd.
- all vertices of priority exactly $h-1$ are at $H^{\omega_1}_{\text{Odd}}$ and all vertices of priority strictly smaller than $h-1$ are at $T^{\omega_1}_{\text{Odd}}$ for Odd

With this, we can state the guarantees our modification provides.

▶ **Theorem 4.** *Let $\mathscr{G}$ be a parity game with priorities no larger than an even number h, for two n-ary trees of height $h/2$ and $h/2+1$ with roots ϵ and ω respectively. Procedure* $\texttt{McNZFast}_{\text{Even}}(\mathscr{G}, h, \epsilon, \omega)$, *with the underlying decomposition being the initial Even and Odd decomposition of $\mathscr{G}$ into these trees output the winning set of Even, and the decomposition computed is the same as the one computed by procedure* $\texttt{McNZ}_{\text{Even}}$ *in Algorithm 1.*

The following statement proves runtime of this algorithm, and shows the quadratic improvement we gain compared to McNaughton-Zielonka, and comparable to the the small progress measure algorithm by Jurdziński [17].

▶ **Lemma 5.** *On a game $\mathcal{G}$ with n vertices with priority at most h, the number of recursive subcalls by either* $\texttt{McNZFast}_{\text{Even}}$ *or* $\texttt{McNZFast}_{\text{Odd}}$ *is at most the product of a polynomial in n and $O\left(\frac{n}{h}\right)^{\lceil h/2 \rceil}$.*

Examples of faster termination

We will demonstrate the algorithm on two examples in this subsection to understand this. These two examples are going to be the family of games $\mathcal{H}_k$ introduced in Section 2 and the family of games, we will call $\mathcal{F}_k$, which was introduced by Friedmann [13].

We recall that in his work, Friedmann had provided a family of games on which McNaughton-Zielonka takes exponential time [13]. We will call this family of examples $\mathcal{F}_k$ for $k \in \mathbb{N}$ but will not describe this family in detail, and instead refer the reader to the work Friedman [13]. Friedmann, showed in Theorem 4.3, of [13] that on the game $\mathcal{F}_k$, the algorithms makes F_k recursive subcalls where F_k denotes the n^{th} Fibonacci number, bounded by approximately $(1.618)^k$. This is smaller than the 2^n time for the previous example we have shown, however, one can remark that because of the structure of the game, it makes it difficult for several algorithms to solve this game. Indeed, the priority promotion algorithms without any enhancements such as memoisation or delayed promotion also takes exponential time on these games. This exponential behaviour is due to their 'rests' performed. We show that our algorithm solves this family of games provided by Friedmann in polynomial time as well as our running example of $\mathcal{H}_k$ in the next two lemmas.

▶ **Lemma 6.** *Algorithm 2 when initialised with the trivial decomposition for both players solves the family of games $\mathcal{H}_n$ in time that is polynomial in n.*

▶ **Lemma 7.** *Algorithm 2 when initialised with the trivial decomposition for both players solves the family of games $\mathcal{F}_n$, introduced by Friedmann [13] in time that is polynomial in n.*

The key idea behind both these proofs is that we do enough work by maintaining a decomposition in an initial recursive subcall. In the future recursive subcalls, when this decomposition is our starting point, the algorithm needs to do at most polynomial amount of processing in the above family of games by showing that they fit one of the following criterion:

- already have a witness for winning in the form of an attractor decomposition from a previous recursive subcall (To prove polynomial termination of $\mathcal{H}_k$ in Lemma 6);
- although the current decomposition of an Even dominion, is the initial decomposition, the decomposition maintained for Odd is robust enough to conclude quickly that it is losing for Odd, and therefore winning for Even; (used in the proof of Lemma 7).
- each of the next several recursive subcalls made are on a significantly smaller subgames due to the structure of the available decompositions. Some of these subcalls might even turn out to be empty (Lemma 6 used to prove fast termination of $\mathcal{F}_k$). This happens dually with the above mentioned phenomenon;
- starting from a specific decomposition for the game, we only require polynomial time using our algorithm to arrive at an attractor decomposition (used in the proof of Lemma 6).

The exact details of these proof requires us to identify specific subgames in the recursive calls and are available in the full version of the paper. Although the proof of both Lemmas rely on induction which in turn depends on the regularity of the subgames in recursive calls, this is not essential for our algorithm to terminate faster and is just an easier proof mechanism. This distinguishes our algorithm from targeted tricks aimed at solving specific families of games faster.

4 Using smaller trees

Attractor decompositions function as a proof of winning for parity games, and can be defined on the recursive structure of trees. Universal trees have been key to all known quasi-polynomial algorithms [6]. The use of small universal trees in the attractor based quasi-polynomial algorithms of Lehtinen, Parys, Schewe and Wojtczak [25] was highlighted in works of Jurdziński, Morvan and Thejaswini [20]. Their work also captured the recursive structure of these witnesses with the help of trees. Moreover, they generalise their algorithm to work on arbitrary trees, and show stronger guarantees. As a corollary, they obtain that using two universal trees in their algorithm gives a correct algorithm for solving parity games. However, the theoretical complexity of these algorithms do not match the run-time of the state of the art. For two trees $\mathcal{T}^{\text{Even}}$ and $\mathcal{T}^{\text{Odd}}$, these algorithms take time proportional to the interleaving of the two trees. This object has size equal to the product of the trees, $|\mathcal{T}^{\text{Even}}| \cdot |\mathcal{T}^{\text{Odd}}|$, as opposed to the algorithms with state of the art [18] worst case complexity of time which is linear in the size of each tree. In fact, in the journal version of the paper of Lehtinen, Parys, Schewe and Wojtczak [25], they emphasise in their introduction that their algorithm has a gap when compared to other quasi-polynomial algorithms. We describe how our technique can be applied to the algorithms of Lehtinen, Parys, Schewe and Wojtczak [25] as well as Jurdziński, Morvan and Thejaswini [20] in this section which would closes this gap. We also achieve this by extending our results to arbitrary trees, of which their algorithms form specific instances. For a detailed report on which universal trees correspond to which algorithm, we refer the reader to the work of Jurdzińksi, Morvan and Thejaswini [20].

Algorithm using arbitrary trees

To apply the technique of remembering decompositions for quasi-polynomial versions of attractor based algorithm, we need to restrict the exponential branching in the previous algorithm. In our algorithm, we bound the branching in the algorithm to inter-leavings of any two arbitrary trees with minimal modification from our previous algorithms. Instantiating these trees to the universal trees underlying the algorithm of Parys [27], or Lehtinen, Schewe, and Wojtczak [24] results in a speed up compared to these respective algorithms.

The version of our algorithm which uses any two arbitrary trees contains two mutually recursive procedures $\texttt{UnivFast}_{\text{Even}}$ and $\texttt{UnivFast}_{\text{Odd}}$ which take as input a game $\mathcal{G}$, the highest priority h in the game, and two nodes ϵ and ω from $\mathcal{T}^{\text{Odd}}$ and $\mathcal{T}^{\text{Even}}$. The nodes ϵ and ω have level h and $h+1$ respectively in their trees for the Even procedure and vice versa for the Odd procedure. The underlying trees are not passed in a recursive subcall as we assume they are a part of the algorithm and can be accessed globally. Other than the arbitrary trees, this algorithm deviates slightly from our previous algorithms in one way. The main loop in this algorithm, whose branching was earlier (virtually) unbounded, is now determined by the trees $\mathcal{T}^{\text{Odd}}$ and $\mathcal{T}^{\text{Even}}$. The pseudo-code for the Even recursive subcall is specified in Algorithm 3 and is given next to the pseudo-code of Algorithm 2 for ease of comparison.

Decompositions and labels

To prove correctness of the algorithms as well as analyse their runtime more carefully, we offer an alternate way of viewing decompositions. Instead of a partition in the shape of a tree, we can equivalently view it as a map to a specific tree. We define such trees, which we call *leafy trees*. We then show that decompositions are nothing but maps into such specific ordered trees. This ordering on the tree induces a natural ordering on the set of all decompositions.

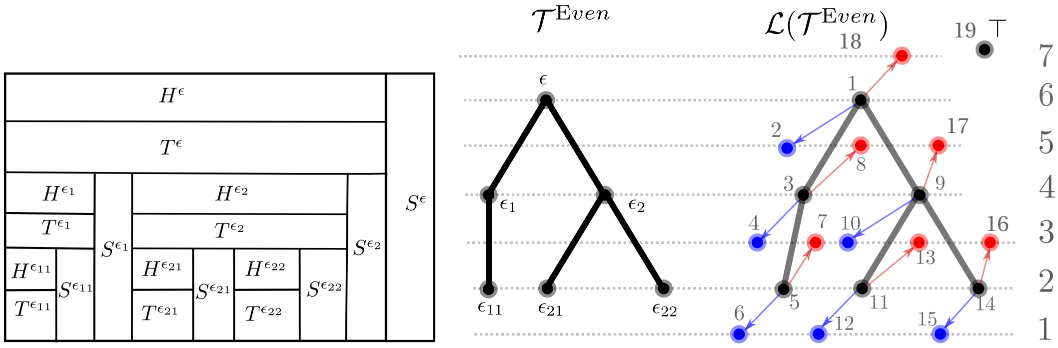

Figure 3 From left to right: A decomposition set into the tree $\mathcal{T}^{\text{Even}}$, tree $\mathcal{T}^{\text{Even}}$ and its corresponding leafy tree with its order

This alternate view also helps us argue correctness and termination using monovariance of the decomposition maintained throughout the algorithm.

Leafy tree Let $\mathcal{T}$ be an ordered tree with a root at level h, an even value. We call the leafy tree of a tree $\mathcal{T}$, denoted by $\mathcal{L}(\mathcal{T})$ to be an ordered set which contains three copies of each element in the tree $\mathcal{T}$ and another element $\top$. We define this more formally below.

▶ **Definition 8.** *Given a tree $\mathcal{T}$, we introduce two additional nodes for each node $\eta \in \mathcal{T}$, which we will call η^T and η^S to denote the nodes corresponding to* top *and* side *nodes respectively. Other than this, we add an element $\top$. We say, that*

$$\mathcal{L}(\mathcal{T}) = \mathcal{T} \cup \{\eta^T \mid \eta \in \mathcal{T}\} \cup \{\eta^S \mid \eta \in \mathcal{T}\} \cup \{\top\}.$$

We sometimes refer to the nodes from $\mathcal{T}$ as a node in the skeleton of the leafy tree. The order of elements in $\mathcal{L}(\mathcal{T})$ is as follows: $\eta \in \mathcal{T}$ and for $\eta' \in \mathcal{L}(\mathcal{T}')$, where $\mathcal{T}'$ is a strict subtree of the tree rooted at η, we have $\eta < \eta^T < \eta' < \eta^S$. Moreover, the order on $\mathcal{T}$ is inherited by $\mathcal{L}(\mathcal{T})$. We also have $\eta < \top$, for any $\eta \in \mathcal{L}(\mathcal{T})$, which is not $\top$. Note that the above conditions ensure that $\mathcal{L}(\mathcal{T})$ is a total order and the smallest element strictly larger than η is well defined. We define level of η^S to be one more than the level of η and the level of η^T to be one less.

For a pictorial representation of a leafy tree, look at Figure 3. Here, the Leafy tree of the tree $\mathcal{T}^{\text{Even}}$ with root ϵ, and its children ϵ_1,ϵ_2 and leaves ϵ_{11}, ϵ_{21}, and ϵ_{22}. In $\mathcal{L}(\mathcal{T}^{\text{Even}})$, we denote elements according to their order, where the elements from $\mathcal{T}^{\text{Even}}$ are the black nodes, η^T with blue and η^S with red. Their level can be inferred by the dotted lines. Also observe a decomposition into the same tree $\mathcal{T}^{\text{Even}}$ placed next to the leafy tree in the picture. We show that a map into a leafy tree can be obtained from the following decomposition.

Labelling We define a labelling for Even into $\mathcal{T}$ as a map from the vertices of a parity game $\mathcal{G}$ to $\mathcal{L}(\mathcal{T})$, which obeys the following conditions:

- if a vertex is mapped to the skeleton of the leafy tree: $\mathcal{T} \subset \mathcal{L}(\mathcal{T})$, then its priority is the same as the level of the node;
- if a vertex is mapped to 'leafy vertex', i.e, η^S or η^T then its priority is at most the level of the leafy vertex.

There are no restrictions on elements mapped to $\top$. One could also equivalently define a labelling for Odd by replacing Even with Odd. When we refer to labellings in the rest of the section, we refer to Even labellings.

Given two labellings for Even, λ_1 and λ_2 into tree $\mathcal{T}$ which are maps from V to $\mathcal{L}(\mathcal{T})$, we say that the order on the elements of $\mathcal{L}(\mathcal{T})$ extend to give a partial order on the labelling. More formally, $\lambda_1 \sqsubseteq \lambda_2$ if and only if for all $v \in V$, $\lambda_1(v) \leq \lambda_2(v)$.

Indeed, the above definition corresponds to a decomposition naturally by the following proposition.

▶ **Proposition 9.** *For a parity game $\mathcal{G}$, with priorities that do not exceed $h+1$ and a tree $\mathcal{T}$ of height $h/2$,*

- *for a labelling λ from $\mathcal{G}$ to $\mathcal{L}(\mathcal{T})$, there is a canonical decomposition $\mathcal{D}_\lambda$ to the tree $\mathcal{T}$;*
- *for a decomposition $\mathcal{D}$ into $\mathcal{T}$, there is a canonical labelling $\lambda_\mathcal{D}$ which maps $V(\mathcal{G})$ to $\mathcal{L}(\mathcal{T})$.*

This allows us to use labellings and decompositions interchangeably, as they have a one to one correspondence with each other. It also induces a partial order $\sqsubseteq$ on decompositions, which is obtained by a the partial order on labellings. We now pinpoint when the decomposition corresponding to a labelling forms an attractor decomposition. We say that a vertex u is *valid* in a labelling λ if $\lambda(u) = \top$ or:

- $\lambda(u) \in \mathcal{T}$ and in one step, Even can ensure that she reaches a vertex v such that $\lambda(v) < \lambda(u)^S$.
- $\lambda(u) = t^T$ or $\lambda(u) = t^S$ and there is a reachability strategy from u to vertices in $\{v \mid \lambda(v) < t\}$ without visiting any vertex v in the path where $\lambda(v) > t$.

In the lemma below, we show that validity of all vertices also corresponds to a witness of winning.

▶ **Lemma 10.** *Given an Even labelling λ, there is a winning strategy for Even from all vertices u such that $\lambda(u) \neq \top$ iff all vertices are valid.*

Not only do such labellings correspond to dominions, the following proposition shows a stronger statement that the decomposition $\mathcal{D}_\lambda$ corresponding to such a labelling λ is an attractor decomposition exactly if and only if every vertex is valid in λ.

▶ **Proposition 11.** *In a parity game $\mathcal{G}$,*

- *an Even decomposition $\mathcal{D}$ to a tree $\mathcal{T}$ is also an attractor decomposition if the corresponding labelling $\lambda_\mathcal{D}$ which maps $V(\mathcal{G})$ to $\mathcal{L}(\mathcal{T})$ is valid for all vertices;*
- *if a labelling λ is valid at all vertices, then the corresponding $\mathcal{D}_\lambda$ is an attractor decomposition.*

As defined in these previous works [7, 20, 8], given a dominion, there could be several attractor decompositions for it. Each of these attractor decompositions could correspond to different trees. In our definition, instead of obtaining the trees from a given attractor decomposition, we view it as a map into a fixed tree. The proposition below helps us show that attractor decompositions defined as labellings is robust.

▶ **Proposition 12.** *Consider a parity game $\mathcal{G}$ with two labellings λ_1 and λ_2 from the vertices to $\mathcal{L}(\mathcal{T})$ for some tree $\mathcal{T}$. Let $\lambda_1 \sqsubseteq \lambda_2$ and $u \in V(\mathcal{G})$ such that $\lambda_1(u) = \lambda_2(u)$. If u is valid in λ_2, then it is also valid in λ_1.*

It can be shown as a corollary of the above proposition, that taking the point-wise minimum of two attractor decompositions is also an attractor decomposition. An interesting consequence of this stated corollary of Proposition 12 is the following lemma, which notes that the set of all attractor decompositions forms a lattice.

▶ **Lemma 13.** *Given a game $\mathcal{G}$ and a tree $\mathcal{T}$, the set of all labellings from $V(\mathcal{G})$ to $\mathcal{L}(\mathcal{T})$ which are also attractor decompositions form a lattice, with the $\sqsubseteq$ order. More specifically, it has a unique maximal and minimal element.*

The meet operator is well defined, which makes the set of attractor decompositions a finite semi-lattice. Since there is a unique maximal element, the trivial labelling which maps all elements to $\top$, this structure forms a lattice. We will focus on this unique minimal attractor decomposition, which plays a key role in our proofs.

Correctness and runtime

With this dual view of an attractor decomposition, we are equipped to prove the correctness and runtime of our algorithms.

Correctness

We prove correctness by proving Theorem 4, which states that on two input trees $\mathcal{T}^{\text{Even}}$ and $\mathcal{T}^{\text{Odd}}$, if there is a dominion of Even that has an attractor decomposition for the tree $\mathcal{T}_{\text{Even}}$, then the decomposition returned contains all vertices in that dominion. Additionally, it contains no vertices from any Odd dominions that has an attractor decomposition for the tree $\mathcal{T}_{\text{Odd}}$. Indeed, if the trees are large enough, and winning set for both Even and Odd have an attractor decomposition into $\mathcal{T}^{\text{Even}}$ and $\mathcal{T}^{\text{Odd}}$ respectively, then the sets D_{Odd} and D_{Even} correspond to winning sets exactly and the algorithm exactly returns the winning sets. On smaller trees that cannot include the entire dominion, the set returned in itself is not a winning set, but it partitions the winning dominions captured by these trees. Therefore, this theorem can be seen as a generalisation of the dominion separation theorem (Theorem 17) in [20].

▶ **Theorem 14.** *Let $\mathcal{G}$ be a parity game with priorities no larger than an even number h, and let $\mathcal{T}^{\text{Even}}$ and $\mathcal{T}^{\text{Odd}}$ be two trees of with roots ϵ and ω respectively. Let D_{Even} be the largest Even dominion in $\mathcal{G}$ that has an attractor decomposition into $\mathcal{T}^{\text{Even}}$ and similarly D_{Odd}, the largest Odd dominion that has an attractor decomposition into $\mathcal{T}^{\text{Odd}}$. Procedure* $\texttt{UnivFast}_{\text{Even}}(\mathcal{G}, h, \epsilon, \omega)$, *with the decomposition being the initial Even and Odd decomposition of $\mathcal{G}$ outputs a set of vertices W such that $D_{\text{Even}} \subseteq W \subseteq V(\mathcal{G} \setminus D_{\text{Odd}})$.*

We prove the above theorem by showing the following stronger statements. We show that for the smallest attractor decompositions $\mathcal{A}^{\epsilon}$ and $\mathcal{A}^{\omega}$ with respect to the corresponding trees, the algorithm maintains the following invariants for the decompositions:

1. $\mathcal{D}^{\epsilon}_{\text{Even}} \sqsubseteq \mathcal{A}^{\epsilon}_{\text{Even}}$ and $\mathcal{D}^{\omega}_{\text{Odd}} \sqsubseteq \mathcal{A}^{\omega}_{\text{Odd}}$;
2. At the end of a recursive subcall of $\texttt{UnivFast}_{\text{Even}}(\mathcal{G}, h, \epsilon, \omega)$, the decompositions at the end of the recursive call are such that $\left[\mathcal{D}^{\epsilon}_{\text{Even}}\right] \cap \left[\mathcal{D}^{\omega}_{\text{Odd}}\right]$ is empty.

The essence of the proof boils down to showing that each operation performed on the decompositions, increases the decomposition without overtaking the smallest attractor decomposition. We achieve this by using the ordering on labellings developed earlier in this section. Additionally, we prove Theorem 4 with an extra invariant that if the trees $\mathcal{T}^{\text{Even}}$ and $\mathcal{T}^{\text{Odd}}$ are complete n-ary trees, then $\mathcal{D}^{\epsilon}_{\text{Even}} = \mathcal{A}^{\epsilon}_{\text{Even}}$ and $\mathcal{D}^{\omega}_{\text{Odd}} = \mathcal{A}^{\omega}_{\text{Odd}}$. This automatically gives us the proof of correctness for Algorithm 2 and that it provides an attractor decomposition.

A notable observation on the proof of Theorem 4 is that these two invariants mentioned hold true for all three of the Algorithms stated in the paper until now, including McNaughton-Zielonka. However, it is only with complete trees that we can prove that both Algorithm 1 and Algorithm 2 produce the minimal attractor decompositions. This shows that McNaugton and Zielonka and its variants provide a winning strategy on termination, whereas the quasi-polynomial algorithms only produce a partition between the winning and loosing vertices, with no strategy for each player. We make precise what guarantees one can achieve using our techniques. We also show a quadratically faster termination for Algorithm 2 and Algorithm 3 than their counterparts which do not maintain a decomposition.

Runtime

We show how our algorithm runs in time that is at most linear in the size of each of the tree, a significant reduction from other attractor based algorithms with a quadratic dependence.

▶ **Theorem 15.** *Consider a game $\mathcal{G}$ with n vertices and priority at most h. Procedure* $\texttt{UnivFast}_{\text{Even}}$ *(resp.* $\texttt{UnivFast}_{\text{Odd}}$*) with trees $\mathcal{T}_{\text{Odd}}$ and $\mathcal{T}_{\text{Even}}$ makes $O(n^c \cdot \max(|\mathcal{T}_{\text{Odd}}|, |\mathcal{T}_{\text{Even}}|))$ many recursive subcalls to* $\texttt{UnivFast}_{\text{Even}}$ *or* $\texttt{UnivFast}_{\text{Odd}}$ *for a constant c. The time taken to perform each operations outside of the recursive subcalls is a polynomial in n, thus making the run-time of this algorithm $O\left(n^d \cdot \max(|\mathcal{T}_{\text{Odd}}|, |\mathcal{T}_{\text{Even}}|)\right)$ for a constant d.*

Our most significant change that contributes to an improvement in the theoretical bound of the running time is that we maintain the decompositions from previous recursive calls. This is crucial in the proof and helps us argue that our modified algorithm does not take too long before there is an increase at least one of our decompositions. We would however like to emphasise that not all implementations of the algorithm would have the claimed run-time, but with carefully designed data structures, the time taken outside a recursive call is at most polynomial. One such implementation would require a data structure which stores each decomposition as a labelling. This labelling is however represented by only maintaining elements in the support of this labelling instead of the whole tree. Since there are at most n vertices, this support is significantly smaller than the size of a potentially quasi-polynomial or exponentially-sized tree. The attractor computations and the respective Set and Move operations performed in the algorithms take time proportional to a polynomial in n, assuming that we have either oracle access or polynomial-time access to queries such as: next sibling of node, parent of a node or child of a node.

5 Outlook

We believe that our technique can be applied to other attractor-based algorithms that were inspired by the McNaughton-Zielonka algorithm [1, 22], but elaborating this in detail is beyond the scope of this paper. The methodology to follow would be analogous to ours: first make explicit how these algorithms are incrementally building attractor decompositions, and then use the decompositions for both players obtained from recursive calls to reduce the sizes of subgames in further recursive calls. Even implementing just the first step of this methodology seems worthwhile: doing so on arbitrary ordered trees, like we did in Section 4, would allow to obtain a generic priority promotion algorithm, which could be made straightforwardly quasi-polynomial by plugging in small universal trees [18, 8], hence generalizing and streamlining the first quasi-polynomial priority promotion algorithm [3].

A weakness of all quasi-polynomial symmetric attractor-based algorithms—including ours—is that they may output correct winning sets, but without constructing winning strategies. This is a major shortcoming in the context of synthesis, where winning strategies correspond to the desired controllers. We argue that our technique, which is based on computing decompositions that are under-approximations of the least attractor decompositions, allows to tackle this weakness with a modest additional computational cost. If the algorithm terminates with a decomposition that is not an attractor decomposition, then the decomposition obtained can serve as a starting point to make further progress. We can run an algorithm for each player that repeatedly increases the decomposition in the underlying order appropriately until an attractor decomposition is obtained. This addition does not increase the worst-case asymptotic running time by more than a polynomial factor.

We have illustrated that our technique, when applied to the standard McNaughton-Zielonka algorithm, yields an algorithm that can solve some hard examples [13] in polynomial time. Other families of hard examples [32, 2] should also be analyzed. We believe that studying the shapes of attractor decompositions of hard examples and how they impact the intricate behaviour of symmetric attractor-based algorithms could shed new light on some central questions in the algorithmic study of parity games, such as how to overcome the quasi-polynomial barrier [6].

References

1 Massimo Benerecetti, Daniele Dell'Erba, and Fabio Mogavero. Solving parity games via priority promotion. *Formal Methods Syst. Des.*, 52(2):193–226, 2018. doi:10.1007/s10703-018-0315-1.

2 Massimo Benerecetti, Daniele Dell'Erba, and Fabio Mogavero. Robust worst cases for parity games algorithms. *Inf. Comput.*, 272:104501, 2020. doi:10.1016/j.ic.2019.104501.

3 Massimo Benerecetti, Daniele Dell'Erba, Fabio Mogavero, Sven Schewe, and Dominik Wojtczak. Priority promotion with parysian flair. *CoRR*, abs/2105.01738, 2021. URL: https://arxiv.org/abs/2105.01738, arXiv:2105.01738.

4 J. C. Bradfield and I. Walukiewicz. *The mu-calculus and Model Checking*, pages 871–919. Springer, 2018. doi:10.1007/978-3-319-10575-8_26.

5 Cristian S. Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. Deciding parity games in quasi-polynomial time. *SIAM Journal on Computing*, 51(2):STOC17–152–STOC17–188, 2022. arXiv:https://doi.org/10.1137/17M1145288, doi:10.1137/17M1145288.

6 W. Czerwiński, L. Daviaud, N. Fijalkow, M. Jurdziński, R. Lazić, and P. Parys. Universal trees grow inside separating automata: Quasi-polynomial lower bounds for parity games. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 2333–2349. SIAM, 2019. doi:10.1137/1.9781611975482.142.

7 L. Daviaud, M. Jurdziński, and K. Lehtinen. Alternating weak automata from universal trees. In *30th International Conference on Concurrency Theory, CONCUR 2019*, volume 140 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:14, Amsterdam, the Netherlands, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

8 Laure Daviaud, Marcin Jurdzinski, and K. S. Thejaswini. The strahler number of a parity game, 2020. doi:10.4230/LIPIcs.ICALP.2020.123.

9 Anuj Dawar and Erich Grädel. The descriptive complexity of parity games. In Michael Kaminski and Simone Martini, editors, *Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings*, volume 5213 of *Lecture Notes in Computer Science*, pages 354–368. Springer, 2008. doi:10.1007/978-3-540-87531-4_26.

10 E. Allen Emerson and Charanjit S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 368–377. IEEE Computer Society, 1991. doi:10.1109/SFCS.1991.185392.

11 E. Allen Emerson, Charanjit S. Jutla, and A. Prasad Sistla. On model-checking for fragments of μ-calculus. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 1993. doi:10.1007/3-540-56922-7_32.

12 Kousha Etessami, Thomas Wilke, and Rebecca A. Schuller. Fair simulation relations, parity games, and state space reduction for büchi automata. In *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*, volume 2076 of *Lecture Notes in Computer Science*, pages 694–707. Springer, 2001. doi:10.1007/3-540-48224-5_57.

13 Oliver Friedmann. Recursive algorithm for parity games requires exponential time. *RAIRO Theor. Informatics Appl.*, 45(4):449–457, 2011. doi:10.1051/ita/2011124.

14 Maciej Gazda and Tim A. C. Willemse. Zielonka's recursive algorithm: dull, weak and solitaire games and tighter bounds. In Gabriele Puppis and Tiziano Villa, editors, *Proceedings Fourth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2013, Borca di Cadore, Dolomites, Italy, 29-31th August 2013*, volume 119 of *EPTCS*, pages 7–20, 2013. doi:10.4204/EPTCS.119.4.

15 Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-36387-4.

16 Ichiro Hasuo, Shunsuke Shimizu, and Corina Cîrstea. Lattice-theoretic progress measures and coalgebraic model checking. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, pages 718–732. ACM, 2016. doi:10.1145/2837614.2837673.

17 M. Jurdziński. Small progress measures for solving parity games. In *17th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1770 of *LNCS*, pages 290–301, Lille, France, 2000. Springer. doi:10.1007/3-540-46541-3_24.

18 M. Jurdziński and R. Lazić. Succinct progress measures for solving parity games. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017*, pages 1–9, Reykjavik, Iceland, 2017. IEEE Computer Society.

19 M. Jurdziński, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. *SIAM Journal on Computing*, 38(4):1519–1532, 2008.

20 Marcin Jurdziński, Rémi Morvan, and K. S. Thejaswini. Universal Algorithms for Parity Games and Nested Fixpoints. *CoRR*, abs/2001.04333, 2020. URL: https://arxiv.org/abs/2001.04333, arXiv:2001.04333.

21 J. J. A. Keiren. Benchmarks for parity games. In *FSEN*, volume 9392 of *LNCS*, pages 127–142, Tehran, Iran, 2015. Springer. doi:10.1007/978-3-319-24644-4_9.

22 Ruben Lapauw, Maurice Bruynooghe, and Marc Denecker. Improving parity game solvers with justifications. In *Verification, Model Checking, and Abstract Interpretation*, page 449–470, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-39322-9_21.

23 K. Lehtinen. A modal μ perspective on solving parity games in quasi-polynomial time. In *33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*, pages 639–648, Oxford, UK, 2018. IEEE. doi:10.1145/3209108.3209115.

24 K. Lehtinen, S. Schewe, and D. Wojtczak. Improving the complexity of Parys' recursive algorithm, 2019. arXiv:1904.11810.

25 Karoliina Lehtinen, Paweł Parys, Sven Schewe, and Dominik Wojtczak. A Recursive Approach to Solving Parity Games in Quasipolynomial Time. *Logical Methods in Computer Science*, Volume 18, Issue 1, January 2022. URL: https://lmcs.episciences.org/8953, doi:10.46298/lmcs-18(1:8)2022.

26 R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993. doi:10.1016/0168-0072(93)90036-D.

27 P. Parys. Parity games: Zielonka's algorithm in quasi-polynomial time. In *MFCS 2019*, volume 138 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:13, Aachen, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.MFCS.2019.10.

28 N. Piterman. From nondeterministic buchi and streett automata to deterministic parity automata. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 255–264, 2006. doi:10.1109/LICS.2006.28.

29 Sven Schewe and Bernd Finkbeiner. Synthesis of asynchronous systems. In *Proceedings of the 16th International Conference on Logic-Based Program Synthesis and Transformation*, LOPSTR'06, page 127–142, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/978-3-540-71410-1_10.

30 K. S. Thejaswini, Marcin Jurdziński, and Pierre Ohlmann. A technique to speed up symmetric attractor-based algorithms for parity games. *CoRR*, abs/2010.08288, 2020. URL: https://arxiv.org/abs/2010.08288, arXiv:2010.08288.

31 T. van Dijk. Oink: An implementation and evaluation of modern parity game solvers. In *Tools and Algorithms for the Construction and Analysis of Systems, 24th International Conference, TACAS 2018*, volume 10805 of *LNCS*, pages 291–308, Thessaloniki, Greece, 2018. Springer. doi:10.1007/978-3-319-89960-2_16.

32 Tom van Dijk. A parity game tale of two counters. In Jérôme Leroux and Jean-François Raskin, editors, *Proceedings Tenth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2019, Bordeaux, France, 2-3rd September 2019*, volume 305 of *EPTCS*, pages 107–122, 2019. doi:10.4204/EPTCS.305.8.

33 W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1–2):135–183, 1998. doi:10.1016/S0304-3975(98)00009-7.

A Set and Move operator

We define operations like "Set T^ϵ to S", "Add S to S^ω_{Odd}" etc., as operations performed on sets rigorously. These short hands helped us capture the essence of these manipulations performed on the decompositions to give a new one. With appropriate data structures, these operations can be performed in nearly linear time, i.e, for a set of size k, it takes time at most $mk\log(n)\log(d)$.

Adding and removing vertices from $\mathcal{D}^\epsilon_{\mathrm{Even}}$ and $\mathcal{D}^\omega_{\mathrm{Odd}}$

To make the definition of Set and Move easier, we define operations that would facilitate this: First, we define $\oplus$ and $\ominus$. We also define an extention of our notation $[\mathcal{D}^\epsilon_{\mathrm{Even}}]$ here. We use $[\![\mathcal{D}^\epsilon_{\mathrm{Even}}]\!]$ to denote the set of of vertices in $[\mathcal{D}^\epsilon_{\mathrm{Even}}]$ along with $S^\epsilon_{\mathrm{Even}}$. We also introduce one extra partition of vertices to the already existing ones in the decompositions: $\top_{\mathrm{Even}}$ for the Even decomposition and $\top_{\mathrm{Odd}}$ for the Odd decomposition to accommodate vertices that no longer fit into a decomposition.

Consider an Even decomposition $\mathcal{D}_{\mathrm{Even}}$ and an element of the Even tree ϵ at level h, and subset of vertices U, whose priorites are at most $h+1$. Intuitively, we modify the decomposition $\mathcal{D}^\epsilon_{\mathrm{Even}}$ by adding elements of a set U to it whilst maintaining all the properties of a decomposition. More rigorously, we let $[\![\mathcal{D}^\epsilon_{\mathrm{Even}}]\!] \leftarrow [\![\mathcal{D}^\epsilon_{\mathrm{Even}}]\!] \oplus U$ denote the following operations in that order:

- $S^\epsilon_{\mathrm{Even}} \leftarrow S^\epsilon_{\mathrm{Even}} \cup \left(U \cap \pi^{-1}(h+1)\right)$;
- $H^\epsilon_{\mathrm{Even}} \leftarrow H^\epsilon_{\mathrm{Even}} \cup \left(U \cap \pi^{-1}(h)\right)$;
- $T^\epsilon_{\mathrm{Even}} \leftarrow T^\epsilon_{\mathrm{Even}} \cup \left(U \cap \pi^{-1}(< h)\right)$.

One could analogously define the operators $\oplus$ for $\mathcal{D}^\omega_{\mathrm{Odd}}$ for an Odd decomposition.

To remove vertices U from the decomposition $\mathcal{D}^\epsilon_{\mathrm{Even}}$, we define $[\![\mathcal{D}^\epsilon_{\mathrm{Even}}]\!] \leftarrow [\![\mathcal{D}^\epsilon_{\mathrm{Even}}]\!] \ominus U$ as follows: for each ϵ' in the subtree of ϵ,

- $S^{\epsilon'}_{\mathrm{Even}} \leftarrow S^{\epsilon'}_{\mathrm{Even}} \setminus U$;
- $H^{\epsilon'}_{\mathrm{Even}} \leftarrow H^{\epsilon'}_{\mathrm{Even}} \setminus U$;
- $T^{\epsilon'}_{\mathrm{Even}} \leftarrow T^{\epsilon'}_{\mathrm{Even}} \setminus U$.

Setting $T^\epsilon_{\mathrm{Even}}$

To give an intuitive definition of "Set $T^\epsilon_{\mathrm{Even}}$ to S", we essentially replace the set of vertices at $T^\epsilon_{\mathrm{Even}}$ with S, and we remove vertices which are originally there which are not S and assign them to the next available positions in the decomposition that would be processed. For a subset of vertices S, we define "Set $T^\epsilon_{\mathrm{Even}}$ to S" where the set $T^\epsilon_{\mathrm{Even}}$ contains S.

- $R \leftarrow T^\epsilon_{\mathrm{Even}} \setminus S$
- $T^\epsilon_{\mathrm{Even}} \leftarrow S$
- $[\![\mathcal{D}^{\epsilon_1}_{\mathrm{Even}}]\!] \leftarrow [\![\mathcal{D}^{\epsilon_1}_{\mathrm{Even}}]\!] \oplus R$, where
  - ϵ_1 is the first child of ϵ in the tree if ϵ is not a leaf,
  - if ϵ is a leaf, then ϵ_1 is the smallest node larger than epsilon, and
  - if the smallest node larger than epsilon does not exist in the tree, we let an element $\top_{\mathrm{Even}} \leftarrow R$.

Setting $S^{\omega_i}_{\mathrm{Odd}}$

This is very similar to the above, the only difference is in the last line. We could unify these setting operations, but we give them separately for more clarity. For a subset S of vertices, we define "Set $S^{\omega_i}_{\mathrm{Odd}}$ to S" to be:

- $R \leftarrow S^{\omega_i}_{\mathrm{Odd}} \setminus S$
- $S^{\omega_i}_{\mathrm{Odd}} \leftarrow S$
- if i is the last child of ω, then $S^\omega_{\mathrm{Odd}} \leftarrow S^\omega_{\mathrm{Odd}} \cup R$

- if not, then $[\![\mathscr{D}^{\omega_{i+1}}_{\text{Odd}}]\!] \leftarrow [\![\mathscr{D}^{\omega_{i+1}}_{\text{Odd}}]\!] \oplus R$

Moving vertices to $S^{\omega_i}_{\text{Odd}}$

For a subset of vertices S, we define " Move S to at least $S^{\omega_i}_{\text{Odd}}$" as follows: we first remove S from all the other positions in the decomposition rooted at ω_i and then add it to $S^{\omega_i}_{\text{Odd}}$ if it was in $[\![\mathscr{D}^{\omega_i}_{\text{Odd}}]\!]$.

- $R \leftarrow [\![\mathscr{D}^{\omega_i}_{\text{Odd}}]\!] \cap S$
- $[\![\mathscr{D}^{\omega_i}_{\text{Odd}}]\!] \leftarrow [\![\mathscr{D}^{\omega_i}_{\text{Odd}}]\!] \ominus S$
- $S^{\omega_i}_{\text{Odd}} \leftarrow S^{\omega_i}_{\text{Odd}} \cup R$

The even counterpart is defined similarly.