universität wien

Dissertation / Doctoral Thesis

Titel der Dissertation / Title of the doctoral thesis

Leveraging the Power of Graph Algorithms: Efficient Algorithms for Computer-Aided Verification

verfasst von / submitted by

Dipl. Ing. Alexander Svozil, BSc

angestrebter akademischer Grad / in partial fulfillment of the requirement for the degree of

Doktor der Technischen Wissenschaften (Dr. techn.)

Wien, 2021 / Vienna, 2021

Studienkennzahl lt. Studienblatt: / degree programme code as it appears on the student record sheet:	A 786 880
Dissertationsgebiet lt. Studienblatt: / field of study as it appears on the student record sheet:	Informatik
Betreuerin: / Supervisor:	Univ.-Prof. Dr. Monika Henzinger

Abstract

Model checking verifies whether a given *model* of a system ensures a *specification*. In *reactive synthesis* the input is a specification and the goal is to construct a correct *reactive system*.

We consider a variety of models: *(state-transition) graphs* where vertices represent the states of a system and edges are the transitions from one state to the next. *Markov Decision Processes (MDPs)* extend graphs to model probabilistic systems, e.g. randomized protocols. *Game graphs* extend graphs to model interactions with the environment and are a central tool for reactive synthesis.

Objectives are sets of traces in the model and represent the specification. The class of *ω-regular objectives* expresses all commonly used functional specifications for reactive systems in model checking and synthesis. Examples for ω-regular objectives we consider are *reachability* objectives, *Büchi* objectives, *bounded Büchi* objectives, *Streett* objectives and *parity* objectives. To model functional and efficient reactive systems we use the combination of *quantitative (mean-payoff)* objectives and ω-regular objectives. For the verification of MDPs with ω-regular objectives, we consider computing the *maximal end-component (MEC)* decomposition of MDPs.

The goal of the thesis is to leverage fast graph algorithms and modern algorithmic techniques for problems in model checking and synthesis on graphs, MDPs, and game graphs. The results include *symbolic algorithms*, a well-known class of algorithms in model checking that trades limited access to the input model for an efficient representation. In particular, we present the following results:

- Algorithms for game graphs with mean-payoff Büchi objectives and mean-payoff coBüchi objectives which match one of the best running time bounds for mean-payoff objectives.
- A near-linear time randomized algorithm for Streett objectives in graphs and MDPs.
- A sub-cubic time algorithm for bounded Büchi objectives in graphs and a cubic time algorithm for game graphs.
- Conditional lower bounds for queries of reachability objectives in game graphs and MDPs. Linear and near-linear time algorithms for sequential reachability objectives in graphs and MDPs respectively.
- The first quasi-polynomial time symbolic algorithm for parity objectives in game graphs.
- We break a long-standing running time bound for MEC decomposition from the '90s by providing a sub-quadratic time symbolic algorithm.

Zusammenfassung

Ein *Modellprüfer* kontrolliert ob ein gegebenes *Modell* eines Systems eine *Anforderung* erfüllt. In der *reaktiven Synthese* ist die Eingabe eine Anforderung und das Ziel ist ein korrektes *reaktives System* zu erzeugen. Wir betrachten folgende Modelle: *(Zustand-Transition) Graphen* wo Knoten die Zustände des Systems darstellen und Kanten die Transitionen von einem Zustand zum Nächsten sind. *Markow-Entscheidungsprozesse* (MEPs) erweitern Graphen um probabilistische Systeme modellieren zu können, z.B. randomisierte Protokolle. *Spielgraphen* erweitern Graphen um Interaktionen mit der Umwelt zu modellieren und sind ein zentrales Werkzeug für die reaktive Synthese.

Zielvorgaben sind Mengen von Abläufen in einem Modell und repräsentieren die Anforderungen. Die Klasse der *ω-regulären Zielvorgaben* drückt alle üblichen funktionalen Anforderungen für reaktive Systeme in der Modellprüfung und in der Synthese aus. Beispiele für ω-reguläre Zielvorgaben die wir betrachten sind *Erreichbarkeits-*, *Büchi-*, *eingeschränkte Büchi-*, *Streett-* und *Paritäts-*Zielvorgaben. Um funktionale und effiziente reaktive Systeme zu modellieren benutzen wir die Kombination von *quantitativen (Mittelwert-)* Zielvorgaben und ω-regulären Zielvorgaben. Für die Prüfung von MEPs mit ω-regulären Zielvorgaben betrachten wir die Berechnung von der *maximalen Schluss-Komponenten (MSK) Dekomposition* eines MEPs.

Das Ziel der Arbeit ist es schnelle Graphalgorithmen und moderne algorithmische Techniken für Probleme in der Modellprüfung und Synthese in Graphen, MEPs und Spielgraphen wirksam einzusetzen. Unter den Resultate sind auch *symbolische Algorithmen*, eine bekannte Klasse von Algorithmen in der Modellprüfung die einen begrenzten Zugang zum Eingabemodell für eine effiziente Repräsentation eintauschen. Wir stellen folgende Ergebnisse vor:

- Algorithmen für Spielgraphen mit Mittelwert-Büchi-Zielvorgaben und Mittelwert-coBüchi-Zielvorgaben die einem der schnellsten Algorithmen für Mittelwert-Zielvorgaben in der Laufzeit gleichziehen.
- Ein randomisierter Algorithmus für Streett-Zielvorgaben in Graphen und MEPs in fast linearer Laufzeit.
- Ein Algorithmus für eingeschränkte Büchi Zielvorgaben in subkubischer Laufzeit für Graphen und kubischer Laufzeit in Spielgraphen.
- Konditionale untere Schranken für Anfragen von Erreichbarkeits-Zielvorgaben in Spielgraphen und MEPs. Algorithmen für sequenzielle Erreichbarkeits-Zielvorgaben in Graphen und MEPs in jeweils linearer Zeit und fast linearer Zeit.

- Der erste symbolische Algorithmus für Paritäts-Zielvorgaben in quasi-polynomieller Zeit.
- Wir durchbrechen eine langstehende Laufzeitschranke für die MSK Dekomposition von den 90er-Jahren indem wir einen symbolischen Algorithmus in sub-quadratischer Laufzeit präsentieren.

Acknowledgments

I am deeply grateful to my advisor Monika Henzinger. The years as a Ph.D. student passed by extremely fast due to her outstanding guidance and support. During all these years I was very proud to have her as my advisor: She continuously pushed me into the right challenges which sharpened my problem-solving skills and taught me essential research skills in computer science. I hope that a small portion of her brilliance which I witnessed many times during our research sessions and convinced me to do the Ph.D. under her guidance in the first place rubbed off on me.

I am deeply thankful to Krishnendu Chatterjee for being a great mentor and for all the invaluable guidance throughout this research. I thank Wolfgang Dvořák for all the hours we spent discussing research problems and writing papers together: Collaborating with him is an awesome experience. I thank Gramoz Goranci for the collaboration and for being a cheerful roommate. I thank Christian for the collaboration and for introducing me to algorithm engineering. I thank Sagar Kale for the collaboration and for improving my writing style.

Special thanks go to Christel Baier and Véronique Bruyère who have agreed to review this thesis and be part of my thesis committee, especially in these crazy times: I could have not wished for a better thesis committee.

Finally I would like to thank all my colleagues over the years: Stefan Neumann, Alexander Noe, Kathrin Hanauer, Sebastian Forster, Pan Peng, Dariusz Leniowski, Veronika Loitzenbauer, Shahbaz Khan, Sebastian Lamm, Marcelo Faraj, and Xiaowei Wu who contributed with questions, suggestions, clarifications and attention.

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007–2013) / ERC Grant Agreement no. 340506. and from the Vienna Science and Technology Fund (WWTF) through project ICT15–003.

Bibliographic Note

Several results of this thesis were already published in conference papers and thus the chapters of this thesis are based on the following papers:

- **Chapter 3:** K. Chatterjee, M. Henzinger, and A. Svozil. "Faster Algorithms for Mean-Payoff Parity Games". In: *MFCS*. vol. 83. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 39:1–39:14.
- **Chapter 4:** K. Chatterjee, W. Dvořák, M. Henzinger, and A. Svozil. "Near-Linear Time Algorithms for Streett Objectives in Graphs and MDPs". In: *CONCUR*. 2019, 7:1–7:16.
- **Chapter 5:** K. Chatterjee, M. Henzinger, S. Kale, and A. Svozil. "Faster Algorithms for Bounded Liveness in Graphs and Game Graphs". In: *unpublished*. 2021.
- **Chapter 6:** K. Chatterjee, W. Dvorák, M. Henzinger, and A. Svozil. "Algorithms and Conditional Lower Bounds for Planning Problems". In: *ICAPS*. AAAI Press, 2018, pp. 56–64.
- **Chapter 7:** K. Chatterjee, W. Dvorák, M. Henzinger, and A. Svozil. "Quasipolynomial Set-Based Symbolic Algorithms for Parity Games". In: *LPAR*. vol. 57. EPiC Series in Computing. EasyChair, 2018, pp. 233–253.
- **Chapter 8:** K. Chatterjee, W. Dvořák, M. Henzinger, and A. Svozil. "Symbolic Time and Space Tradeoffs for Probabilistic Verification". In: *unpublished*. 2021.

Contents

1 Introduction **1**
- 1.1 Related Work . . . 5
- 1.2 Results and Outline. . . . 7

2 Preliminaries **9**
- 2.1 Models . . . 9
- 2.2 Plays and Strategies. . . . 10
- 2.3 Objectives. . . . 10
- 2.4 Winning Strategies, Winning Sets, Queries, and Value Functions . 12
- 2.5 Basic Algorithmic Results . . . 13
  - 2.5.1 Graphs: SCCs, decremental algorithms and Graph Reachability 13
  - 2.5.2 MDPs: MECs and Random Attractors . . . 14
  - 2.5.3 Game Graphs: Closed Sets, Attractors . . . 15
- 2.6 Symbolic Model of Computation . . . 16
- 2.7 Conjectured Lower Bounds . . . 18

3 Faster Algorithms for Mean-Payoff Parity Games **21**
- 3.1 Introduction . . . 21
- 3.2 Decremental Algorithm for Threshold Mean-Payoff Games . . . 23
- 3.3 Threshold Mean-Payoff Parity Games . . . 27
  - 3.3.1 Threshold Mean-Payoff Büchi Games . . . 27
  - 3.3.2 Threshold Mean-Payoff coBüchi Games . . . 28
  - 3.3.3 Threshold Mean-Payoff Parity Games . . . 31
- 3.4 Optimal Values for Mean-payoff Parity Games . . . 32
- 3.5 Conclusion . . . 34

4 Near-Linear Time Algorithms for Streett Objectives in Graphs and MDPs **35**
- 4.1 Introduction . . . 35
- 4.2 Decremental SCCs . . . 37
- 4.3 Graphs with Streett Objectives . . . 40
- 4.4 Algorithms for MDPs . . . 46
  - 4.4.1 Maximal End-Component Decomposition . . . 46
  - 4.4.2 Almost-Sure Reachability . . . 47

4.4.3 Decremental Maximal End-Component Decomposition . . . 48
4.4.4 MDPs with Streett Objectives . . . 53

5 Faster Algorithms for Bounded Liveness in Graphs and Game Graphs 57
5.1 Introduction . . . 57
5.2 Algorithms for Graphs . . . 59
5.2.1 The Bounded Büchi Objective . . . 60
5.2.2 The Bounded coBüchi Objective . . . 66
5.3 Algorithms for Game Graphs . . . 68
5.3.1 An $O(n^2d^2)$-time Algorithm for Bounded Büchi in Games . 69
5.3.2 An $O(n^2d)$-time Algorithm for Bounded Büchi in Games . . 71

6 Algorithms and Conditional Lower Bounds for Planning Problems 81
6.1 Introduction . . . 81
6.2 Coverage Problem . . . 85
6.2.1 Algorithms . . . 86
6.2.2 Conditional Lower Bounds . . . 86
6.3 AllCoverage Problem . . . 90
6.3.1 Algorithms . . . 91
6.3.2 Conditional Lower Bounds . . . 91
6.4 Sequential Reachability Problem . . . 93
6.4.1 Algorithms . . . 94
6.4.2 Conditional Lower Bounds . . . 104
6.5 Discussion and Conclusion . . . 107
6.5.1 Implications for factored models . . . 107
6.5.2 Concluding Remarks . . . 108

7 Quasipolynomial Set-Based Symbolic Algorithms for Parity Games 111
7.1 Introduction . . . 111
7.2 The Progress Measure Algorithm . . . 115
7.3 Set-Based Symbolic Black Box Progress Measure Algorithm . . . 117
7.3.1 Improving the Basic Algorithm . . . 117
7.3.2 Correctness . . . 120
7.4 Implementing the Ordered Progress Measure . . . 125
7.5 Reducing the Number of Sets for the OPM . . . 126
7.5.1 Algorithmic Details for the Reduced Symbolic Space Algorithm 1 . . . 127
7.6 Algorithmic Details for the Reduced Symbolic Space Algorithm 2 . 130
7.6.1 Data Structure. . . . 130
7.6.2 Correctness . . . 132
7.6.3 Symbolic Resource consumption . . . 133
7.7 Conclusion . . . 133

8 Symbolic Time and Space Tradeoffs for Probabilistic Verification 135

8.1 Introduction . . . 135
8.2 Algorithmic Tools . . . 138
8.2.1 Symbolic SCCs Algorithm . . . 138
8.2.2 Symbolic Random Attractors . . . 138
8.2.3 Separators . . . 139
8.3 Symbolic MEC decomposition . . . 141
8.3.1 Collapsing End-components . . . 141
8.3.2 Algorithm Description . . . 142
8.3.3 Correctness . . . 144
8.3.4 Symbolic Operations Analysis . . . 148
8.3.5 Symbolic Space . . . 150
8.4 Symbolic Qualitative Analysis of Parity Objectives . . . 151
8.4.1 Almost-sure Reachability. . . . 152
8.4.2 Parity Objectives. . . . 153
8.5 Conclusion . . . 158

9 Conclusion **159**

Bibliography **161**

CHAPTER 1

Introduction

In the last decades, computer systems enriched the world in many aspects: Transportation (e.g. self-driving cars), entertainment (e.g. decades of online video content) and production (e.g. enterprise resource planning) are just a few examples. Because humans create most of the aforementioned systems, coding mistakes and conceptual errors are frequent and hard to avoid. Overlooked side cases, misspelled variable names and uninitialized pointers are part and parcel of the systems produced nowadays. Especially in safety-critical environments, bugs have detrimental outcomes. For example, in 1996 the launch of the Ariane 5 rocket failed because its navigation system tried to convert a 64-bit number to a 16-bit number without checking if 16 bits are sufficient to hold the value [29]. Other examples of similar disasters include bugs in medical equipment, electric power transmission, and automated trading systems which cost human lives and millions of dollars [215].

In practice, computer scientists usually write *tests* to avoid errors. A test succeeds if the input matches the expected output. On the upside, tests are efficient and easy to write. On the downside, tests cannot, in general, certify the correctness of a system. In particular, bugs in a concurrent system are notoriously hard to find with tests: In one instance, a test of a concurrent system may pass but, on the next instance, the same test may fail due to concurrency issues. In contrast to the testing paradigm, the field of computer-aided verification *proves* that a system is correct.

Due to the undecidability of the Halting Problem [101, 220] and Rice's Theorem [199], we know that certifying if a given arbitrary system has a property is undecidable. Even when we accept these harsh limitations and shift the focus to systems with finitely many states where the verification of properties becomes decidable, many questions in computer-aided verification are immensely difficult to resolve due to results in complexity theory.

Despite these sobering general results, Clarke, Emerson, Sifakis and others invented a practical approach to tackle the challenges in computer-aided verification called "model checking" [106, 109, 194, 196]. Given a system and a system specifi-

cation the "model checking workflow" is as follows: First, we compile the system into a *finite state-transition graph* (modeling) and translate the system specification into *properties* of the model. The *model checker* takes the state-transition graph and the properties as input. The output is either (a) that the given state-transition graph fulfills the desired properties, i.e., the system ensures the system specification or (b) that there is a counterexample of one of the properties because, e.g., the system has a bug or the model does not sufficiently represent the system.

In what follows, we consider *reactive systems*, i.e., systems that continuously interact with the environment (other parallel processes, user input, etc.). A state of a reactive system consists of the variable valuations at a point in time. An example of a rudimentary reactive system is a semaphore, a reactive system that makes sure that only one out of many parallel processes enters a critical section at each point in time. For reactive systems *terminal states* are undesirable (e.g., the semaphore is in a deadlock) and, thus, every state has a successor state. More sophisticated examples for reactive systems are, for example, an aircraft or a flight-control systems.

Models. The standard model is a *(state-transition) graph* where a vertex represents a state of the reactive system. The edges of the graph represent transitions between states, e.g., when a variable is incremented. A *play* is an infinite path starting at an *initial vertex* in the graph which represents the initial state of the reactive system. A play represents a run of the reactive system. While many reactive systems can be modeled using state-transition graphs, there are cases where graphs are not able to express the desired properties of a reactive system: For example, if the state transition depends on the action of an adversarial actor or an uncertain environment. When a reactive system interacts with an adversarial environment (e.g., user input), *game graphs* extend the notion of graphs [2, 119]. Another prominent use-case for game graphs is *Synthesizing* [36, 49]: Synthesizing is the ambitious approach to build a reactive system from scratch given a specification [53, 100]. The process of synthesizing is a turn-based game where the environment controls possible inputs for the system and the system controls the output [110]. If a state transition depends on a given probability distribution or non-deterministic behavior, *Markov Decision Processes (MDPs)* extend graphs [113, 221]. Model Checking in MDPs is a key component in establishing the correctness of randomized distributed algorithms [169], studying biological processes [137], analyzing security protocols [189], optimizing power management [188] and many more [23, 25, 26, 65].

Games Graphs. In game graphs, there are two players. Player 1 represents the choices of the system and player 2 represents the environment. *Player-1 vertices* and *player-2 vertices* partition the vertices in the graph. At the beginning of a play, a token is placed on the vertex which represents the initial state. When the token is on a player-1 vertex, player 1 moves the token along one of its outgoing edges and the system goes to the next state. When the token is on a player-2 vertex, player 2 (the environment) moves the token along one of the outgoing edges.

Markov Decision Processes (MDPs). When the system interacts with a non-deterministic or an uncertain component we model it with MDPs. That is, vertices are partitioned

into player-1 vertices and *random vertices*. Again, at the start of a play, we place a token on the initial state. For player-1 vertices, the system has the choice to go to the next state along its outgoing edges. When the token is on a random vertex, a probability distribution determines the next vertex along one of its outgoing edges. For the problems we consider in MDPs, we can assume without loss of generality that the probability distribution is uniform.

Specification. The system specification defines the desired properties of a model and is input in both model checking and reactive synthesis. An example of such a property is the *safety property* where a system must not reach a set of undesired states. *Objectives* define the desired properties of a reactive system as a set of plays. *Qualitative objectives*, such as ω-regular objectives describe the functional behavior of a reactive system. Throughout, we consider mostly ω-regular objectives which are canonical because they express most functional requirements for model checking and synthesis [181, 193]. *Quantitative objectives* describe desired behavior for the performance or resource consumption of a reactive system [68] and are often combined with qualitative objectives [37, 83]. We consider the following objectives:

Reachability and Safety objectives. Given a set of "good vertices", a *reachability objective* is defined by the set of plays, which include a good vertex. Dually, given a set of "safe vertices", a *safety objective* is the set of plays, which visit only safe vertices.

Sequential Reachability objectives. Given k sets of vertices, the *sequential reachability objective* is the set of plays that visit at some point in time a vertex in the first set, then, later on, a vertex in the second set and so on until the play visits a vertex in the kth set of vertices.

Büchi and coBüchi objectives. Given a set of "Büchi vertices", *Büchi objectives* are the set of plays which visit a Büchi vertex infinitely often. Dually, given a set of "coBüchi vertices", *coBüchi objectives* describe the set of plays which visits only the coBüchi vertices infinitely often.

Bounded Büchi and bounded coBüchi objectives. Bounded Büchi objectives and *bounded coBüchi objectives* extend the Büchi and coBüchi objectives: Given an integer d and a set of Büchi vertices a play in the bounded Büchi objective includes a Büchi vertex every at most d steps after finitely many steps. Dually, the bounded coBüchi objectives contains a play if it visits d coBüchi vertices consecutively infinitely often.

Parity objectives. For the *parity objective*, every vertex in the model has an integer priority. The parity objective includes plays where the vertex with minimum priority visited infinitely often is even.

Streett objectives. Streett objectives have a set of requests and corresponding grants. Each request and corresponding grant is a set of vertices in the model. Streett objectives include a play if it reaches for every request occurring infinitely often the corresponding grant infinitely often.

Mean-Payoff objectives. Mean-Payoff objectives are quantitative objectives where every edge in the model has an associated reward. The *payoff* of a play is the limit

of the long-run average of the rewards in the play. Mean-payoff objectives have a threshold and include a play if the payoff is above the threshold.

Mean-Payoff Parity objectives. Mean-payoff parity objectives combine mean-payoff objectives and parity objectives: A play is in the mean-payoff parity objective if it is in the parity objective and the mean-payoff objective.

Algorithmic Questions. We perform an algorithmic study of the following two questions:

- Given a model, an objective, and an initial vertex in the model we compute whether player 1 can "force" a play in the objective (despite the non-determinism in an MDP or the adversarial choices of player 2 in a game graph). We say that the play *ensures* the objective and that the vertex is *winning* for player 1. A natural extension of this question is to compute *all* vertices for which player 1 can ensure the given objective in the model, i.e., the *winning set*.
- Given an MDP, we compute the *maximal end-component (MEC) decomposition* of an MDP. Intuitively, a MEC describes a maximal (under set inclusion) set of vertices in the MDP for which player 1 can from any vertex in the MEC "force" to reach every other vertex in the MEC despite the probabilistic elements of the model, i.e., it generalizes strongly connected components in graphs. Computing the MECs of an MDP is a key component for computing winning sets of ω-regular objectives [8, 69].

Symbolic Algorithms and the state explosion problem. A state of a reactive system consists of a valuation for each variable at a point in time. Note that this causes huge numbers of vertices in the induced model because the number of states grow exponentially with the number variables, e.g., for a bit array with 20 entries and two variables with values in $\{0, \ldots, 9\}$ we have at least $2^{20} \cdot 10^2$ states. This huge induced model is a major drawback in model-checking because the model might not fit into the main memory. To tackle this problem, a successful approach is to express the sets of states and transition relations *implicitly* instead of *explicitly* in terms of BDDs [24, 51, 52, 56]. We consider a theoretical model for algorithms that works for this implicit representation without considering specifics of the representation called *symbolic model of computation* [74, 79, 94, 131, 153]. A *symbolic algorithm* can use the same operations as a regular RAM algorithm, except for the access to model: To access the input graph a symbolic algorithm must use the following two types of *symbolic operations*:

1. *One-step operations $Pre(\cdot)$ and $Post(\cdot)$.* Given a set of vertices X, the predecessor operation $Pre(X)$ returns the set of vertices with an edge to a vertex in X. Similarly, the successor operation $Post(X)$ returns the set of vertices with an edge from some vertex in X.
2. *Basic set operations.* Basic set operations have as input one or two sets of vertices or edges and perform, for example, the union, intersection, and complement on the set(s).

In the symbolic model, running time is defined as the *number of symbolic operations*. The symbolic model defines one unit of space as one set (not the size of the set) due

to the implicit representation as BDD. *Symbolic space* is the number of sets stored simultaneously at any point of the algorithm.

Modern Graph Algorithms. We leverage the power of modern graph algorithms to obtain faster algorithms for the algorithmic questions. We also provide negative results called "conditional lower bounds", i.e., we show that an improvement in running time for the algorithmic questions implies an algorithm that breaks a long-standing running time barrier for well-studied problems like SAT. We describe three algorithmic concepts which contributed to the results over the algorithmic questions:

Dynamic Graph Algorithms. A *dynamic graph algorithm*, in contrast to a *static* algorithm, maintains a property of a graph, e.g. strongly connected components (SCCs), while edges of a graph are removed and added. The most naïve dynamic graph algorithm computes the property from scratch every time an edge is removed or added. Finding fast dynamic algorithms is an exciting mathematical challenge because they are relevant in practice [135] and can be used as subroutines in static algorithms, for example, when vertices are repeatedly deleted in a static algorithm.

Hierarchical Graph Decomposition. The *hierarchical graph decomposition* is a technique originally introduced for dynamic graph algorithms [139] but a breakthrough result showed that a similar technique can be used to obtain faster algorithms for computing winning sets in game graphs with Büchi objectives [91] and other problems [81, 140] where we repeatedly remove sets of vertices. Given a model with n vertices, the technique consists of iteratively constructing $\log n$ subgraphs with only $O(2^i)$ ($1 \leq i \leq \log n$) edges for each vertex of the original graph (the vertices remain unmodified in the subgraphs). The subgraph G_i is defined by the edges in iteration i. The goal is to find a set of vertices that fulfills a given desired property and to show that the size of this set is proportional to the number of edges in G_i. We save running time by repeatedly looking at subgraphs of the original graph with a smaller amount of edges to find vertices with the desired property and making sufficient progress by removing at least $O(2^i)$ vertices from the graph.

Conditional Lower Bounds. Similar to reductions from an NP-hard problem, *conditional lower bounds* give running time guarantees conditioned on the fact that there is no significant improvement in running time over decades for certain well-studied problems [222]. Examples for the "well-studied" problems include *all-pair shortest paths in graphs*, *3-SUM* and *CNF-SAT*. In particular, if we have a conditional lower bound for an algorithmic question, we obtain better algorithms for a long-standing well-studied problem if there is a substantial improvement in running time for the algorithmic question which is highly unlikely.

1.1 Related Work

In this section, we present an overview of the results for the considered algorithmic questions. We describe the running time of the algorithm for a model with n vertices and m edges. For parity objectives, we consider models with d priorities and for

mean-payoff objectives W denotes the maximum weight of an edge. For Streett objectives, we denote with b the size of the input sets consisting of the requests and grants.

For explicit algorithms, in *graphs*, the following results are known:

- The winning set of reachability, safety, Büchi and coBüchi objectives can be computed in linear time using depth first search and strongly connected components [216].
- For Streett objectives, the best algorithms compute the winning set in time $O(m^{1.5}\sqrt{\log n})$ and $O(n^2)$ [75, 138].

In *MDPs*:

- The MEC-decomposition can be determined in $O(min(m^{1.5}, n^2))$ [91].
- Computing the winning set of reachability objectives can be done in time $O(m + \mathsf{MEC})$ where MEC denotes the running time of the best algorithm for MEC-decomposition [75].
- Computing the winning set of Streett objectives can be done in time $O(m^{1.5}\sqrt{\log n})$, $O(n^2)$ [75].

In *game graphs*:

- Computing the winning set of reachability and safety objectives can be done in linear time [28, 147].
- For Büchi and coBüchi objectives, the winning set can be computed in $O(n^2)$ time [91].
- For computing the winning set of parity objectives there is a long line of work which improves the running time [151, 154, 208]. The most important recent development is an algorithm in quasi-polynomial running time ($O(n^{\log d+6})$) [59]. Since this breakthrough, many other quasi-polynomial algorithms were discovered [116, 128, 152, 174, 192]. The long-standing open question is if there is a polynomial-time algorithm for computing the winning set of parity objectives in game graphs.
- Similarly, for computing the winning set of mean-payoff objectives it is not known whether there exists a polynomial time algorithm. The two fastest algorithms are $O(mn2^{n/2})$ [120, 161] and $O(mnW)$ [44].
- For the intersection of mean-payoff objectives and parity objectives, i.e., mean-payoff parity objectives the best algorithm for computing the winning set is in $O(dmn^{\lg(d/\lg n)+2.45}W)$ [115].

For *symbolic algorithms* the following results are known:

- In MDPs, there are two results for computing the MEC-decomposition: First, the classical symbolic algorithm which performs $O(n)$ symbolic SCC decompositions which can be done in $O(n)$ symbolic operations and $O(\log n)$ symbolic space [131]. Second, an algorithm which uses $O(n\sqrt{m})$ symbolic operations and $O(\sqrt{m})$ symbolic space [79].
- In game graphs, the best algorithm to compute parity objectives uses $O(\lg d)$ symbolic space and $n^{2\lg(d/\lg n)+O(1)}$ symbolic operations [153]. Another, older algorithm for parity games before the first explicit quasi-polynomial algo-

rithm [59] uses $\min\{n^{O(\sqrt{n})}, O(n^{d/3+1})\}$ symbolic operations and $O(n)$ symbolic space [73].

1.2 Results and Outline.

In this section, we give an overview of the chapters in the thesis. The $\widetilde{O}(\cdot)$ notation hides poly-logarithmic factors.

- In Chapter 2, we introduce the necessary definitions to describe the results.
- In Chapter 3, we present algorithms which improve the running time for computing the winning sets of mean-payoff Büchi objectives and mean-payoff co-Büchi objectives from $O(n^3mW)$ to $O(nmW)$. Furthermore, we present an $O(n^{d-1}mW)$ time algorithm for the winning set of mean-payoff parity objectives.
- In Chapter 4, we present near-linear algorithms, i.e., a $\widetilde{O}(m+b)$ time algorithm for computing the winning set of a Streett objective in graphs and MDPs. Towards that goal, we present $\widetilde{O}(m)$ time algorithms for computing the MEC decomposition of an MDP, computing the MEC decomposition under edge deletions, and computing the winning set of a reachability objective in MDPs.
- We present the first sub-cubic time algorithm for computing the winning set of bounded Büchi objectives in graphs in Chapter 5. For the same problem in game graphs, we present a new $O(n^2d)$ time algorithm using hierarchical graph decomposition.
- We study reachability problems with k sets of vertices in graph games and MDPs in Chapter 6: In particular, we present algorithms for computing the winning set of sequential reachability objectives and the *coverage problem* where, instead of one reachability objective we are given k reachability objectives which must be satisfied at the same time. For computing the winning set of sequential reachability objectives we present conditional lower bounds for game graphs. For MDPs, we present a subcubic time algorithm which rules out conditional lower bounds. For the coverage problem we provide novel conditional lower bounds for game graphs and MDPs and argue why the problem can be solved in linear time in graphs.
- In Chapter 7, we shift the focus to symbolic algorithms. We present the first *symbolic* algorithm for computing the winning set of parity objectives in graph games with a quasi-polynomial number of symbolic operations and $O(d \log n)$ symbolic space.
- In Chapter 8, we present two symbolic algorithms for MDPs. The first algorithm computes the MEC decomposition with a symbolic space and symbolic operation trade-off: It requires $\widetilde{O}(n^{2-\epsilon})$ symbolic operations and $\widetilde{O}(n^{\epsilon})$ symbolic space for $0 < \epsilon \leq 1/2$. The second algorithm computes the winning set of parity objectives with $\log d$ computations of the MEC decomposition and improves the time-space product from $\widetilde{O}(n^2d)$ to $\widetilde{O}(n^2)$.

CHAPTER 2

Preliminaries

In this chapter, we introduce necessary definitions for the following chapters. Since the notation and definitions are standard, we base them on similar definition sections [74, 86, 91, 178].

2.1 Models

Game Graphs. Game graphs $\Gamma = ((V, E), \langle V_1, V_2 \rangle)$ consist of a finite set of vertices V, a finite set of edges E and partitions V into player-1 vertices V_1 and the adversarial player-2 vertices V_2.

Markov decision process (MDP). An MDP $P = ((V, E), \langle V_1, V_R \rangle, \delta)$ has a finite set of vertices V which we partition into player-1 vertices V_1 and random vertices V_R, a finite set of edges $E \subseteq (V \times V)$, and a probabilistic transition function δ. The probabilistic transition function maps V_R to $\mathcal{D}(V)$, i.e., the set of probability distributions over the set of vertices V. There is an edge from a random vertex v to a vertex w, i.e. $(v, w) \in E$ if and only if $\delta(v)[w] > 0$. We call an edge $e = (u, v)$ *random edge* if $u \in V_R$. Otherwise it is a *player-1 edge.* For simplicity, we let $\delta(v)$ be the uniform distribution over vertices u with $(v, u) \in E$: this common technical assumption is without loss of generality for the qualitative analysis of MDPs.

Remark 2.1.1. A standard way to define MDPs, e.g. [134], is to consider vertices with actions to define a probabilistic transition function for every vertex and action. In our model, the choice of actions is represented as the choice of edges at player-1 vertices and the probabilistic transition function is represented by the random vertices. This allows us to treat MDPs and game graphs uniformly, and graphs can be described easily as a special case of MDPs.

We also follow the common technical assumption that vertices in game graphs and MDPs do not have self-loops and that every vertex has an outgoing edge.

Graphs. *Graphs* are the special case of MDPs with $V_R = \emptyset$ and game graphs with $V_2 = \emptyset$. The set $Out(v) = \{u \in V \mid (v,u) \in E\}$ describes the set of successors of v and the set $In(v) = \{u \in V \mid (u,v) \in E\}$ describes the set of predecessors of v. When U is a set of vertices, we define $E(U)$ to be the set of all edges incident to the vertices in U, i.e., $E(U) = \{(u,v) \in E \mid u \in U \text{ or } v \in U\}$. With $G[S]$ we denote the subgraph of $G = (V,E)$ induced by the set of vertices $S \subseteq V$, i.e., $G[S] = (S, E_S)$ where $E_S = \{(u,v) \mid u,v \in S\}$. A graph is *strongly connected* if there is a path between every pair of vertices. We denote with $n = |V|$ the number of vertices and with $m = |E|$ the number of edges.

2.2 Plays and Strategies.

Plays. An infinite sequence $\omega = \langle v_0, v_1, v_2, \ldots \rangle$ of vertices such that each $(v_{i-1}, v_i) \in E$ for all $i \geq 1$ is called *play*. The set of all plays is denoted with Ω. A *finite play* V^* is a finite prefix of a play.

Strategies. We call the recipes that extend finite plays *strategies* and there are player-1 strategies and player-2 strategies. A player-1 *strategy* is a function $\sigma : V^* \cdot V_1 \mapsto V$ and maps every finite play $\omega \in V^* \cdot V_1$ that ends in a player-1 vertex v to a successor vertex $\sigma(\omega)$, i.e., $(v, \sigma(\omega)) \in E$. We define player-2 strategies analogously. A player-1 strategy is *memoryless* if $\sigma_1(\omega) = \sigma_1(\omega')$ for all $\omega, \omega' \in V^* \cdot V_1$ that end in the same vertex $v \in V_1$, that is, the strategy does not depend on the entire finite play, but only on the last vertex. We define memoryless player-2 strategies analogously. We denote with Σ and Π the sets of all strategies for player 1 and player 2 respectively.

Outcome of Strategies. The *outcome of strategies* is a unique play starting at an initial vertex which we define for all models: In graphs, given an initial vertex, a player-1 strategy induces a unique play in the graph. In MDPs, given an initial vertex v and a player-1 strategy σ, we obtain a set of possible plays $\omega(v,\sigma)$ if player 1 follows σ since random vertices choose their successor according to a probability distribution. An event is a measurable subset of $\omega(v,\sigma)$ and the probabilities of events are uniquely defined [221]. For a vertex v, strategy σ and an event $\mathscr{A} \subseteq \Omega$, we denote by $\Pr_v^\sigma(\mathscr{A})$ the probability that a play belongs to $\mathscr{A}$ if the game starts at v and player 1 follows σ. In game graphs, given a starting vertex v and strategies player-1 strategy σ and player-2 strategy π, the unique play $\omega(v,\sigma,\pi) = \langle v_0, v_1, v_2, \ldots, \rangle$ is defined as $v_0 = v$ and for all $i > 0$ if $v_i \in V_1$ then $\sigma(\langle v_0, \ldots, v_i \rangle) = v_{i+1}$ and if $v_i \in V_2$, then $\pi(\langle v_0, \ldots, v_i \rangle) = v_{i+1}$.

2.3 Objectives.

An *objective* $\Phi \subseteq \Omega$ is the set of "winning plays". The play $\omega \in \Omega$ *satisfies* the objective if $\omega \in \Phi$. For a play $\omega = \langle v_0, v_1, v_2, \ldots \rangle$ define $Inf(\omega) = \{v \in V \mid v_i = v \text{ for infinitely many } i \geq 0\}$ to be the set of vertices that occur infinitely often in ω. For the definitions of the following objectives let Γ be an MDP or a game graph.

1. *Reachability and Safety objectives.* A *Reachability objective* $\text{Reach}(T, \Gamma)$ requires, given a set T of vertices, that a play visits *at least one* vertex in T. Dually, for a set of vertices C, a play in the *safety objective* $\text{Safety}(C, \Gamma)$ visits *only* vertices in C. Formally, $\text{Reach}(T, \Gamma) = \{\langle v_0, v_1, v_2, \dots \rangle \in \Omega \mid \exists k \geq 0 : v_k \in T\}$ and $\text{Safety}(C, \Gamma) = \{\langle v_0, v_1, v_2, \dots \rangle \in \Omega \mid \forall k \geq 0 : v_k \in C\}$. The two objectives are dual, i.e., $\text{Reach}(T, \Gamma) = \Omega \setminus \text{Safety}(V \setminus T, \Gamma)$.

2. *Sequential Reachability.* For a tuple of vertex sets $\mathcal{T} = (T_1, T_2, \dots, T_k)$ in Γ the *sequential reachability objective* is the set of infinite plays that contain a vertex of T_1 followed by a vertex of T_2 and so on up to a vertex of T_k, i.e., $Seq(\mathcal{T}, \Gamma) = \{\langle v_0, v_1, v_2, \dots \rangle \in \Omega \mid \exists j_1, j_2, \dots j_k : v_{j_1} \in T_1, v_{j_2} \in T_2, \dots, v_{j_k} \in T_k \text{ and } j_1 \leq j_2 \leq \dots \leq j_k\}$.

3. *Büchi and coBüchi objectives.* Given a set B of vertices called "Büchi vertices", the *Büchi objective* $\text{Büchi}(B, \Gamma)$ contains all plays which visit a vertex in B infinitely often. Dually, the *coBüchi objective* $\text{coBüchi}(C, \Gamma)$ contains a play if it visits only vertices in C infinitely often. Formally, $\text{Büchi}(B, \Gamma) = \{\omega \in \Omega \mid Inf(\omega) \cap B \neq \emptyset\}$ and $\text{coBüchi}(C, \Gamma) = \{\omega \in \Omega \mid Inf(\omega) \subseteq C\}$. The two objectives are dual, i.e., $\text{Büchi}(B, \Gamma) = \Omega \setminus \text{coBüchi}(V \setminus B, \Gamma)$.

4. *Bounded Büchi and bounded coBüchi objectives.* Given a set of "Büchi vertices" B and an integer $d \geq 0$, the *bounded Büchi objective* $\text{boundedBüchi}(B, d, \Gamma)$ includes a play if, after visiting finitely many arbitrary vertices, the distance between any two consecutive Büchi vertices in the play is at most d. Dually, the *bounded coBüchi objective* $\text{boundedcoBüchi}(C, d, \Gamma)$ includes a play if it visits at least d consecutive vertices in C infinitely often.

   Formally, the sets of "winning plays" are

   $$\text{boundedBüchi}(B, d, \Gamma) = \{\omega \in \Omega \mid \exists i \geq 0 \forall j \geq i : \{v_j, v_{j+1}, \dots, v_{j+d-1}\} \cap B \neq \emptyset\}$$
   $$\text{boundedcoBüchi}(C, d, \Gamma) = \{\omega \in \Omega \mid \forall i \geq 0 \exists j \geq i : \{v_j, v_{j+1}, \dots, v_{j+d-1}\} \subseteq C\}.$$

5. *Parity objectives.* Given a *priority function* p that maps every vertex to a non-negative integer priority, a play satisfies the *parity objective* if the minimum priority vertex that appears infinitely often is even. Formally, the parity objective is the set $\text{Parity}(p, \Gamma) = \{\omega \in \Omega \mid \min\{p(v) \mid v \in Inf(\omega)\} \text{ is even}\}$. The Büchi and coBüchi objectives are special cases of parity objectives with two priorities: For Büchi objectives the image of p is $\{0, 1\}$ and for coBüchi objectives it is $\{1, 2\}$. Given a game graph Γ with parity function p, we call (Γ, p) a *parity game.*

6. *Threshold mean-payoff objectives.* A *weight function* $w : E \mapsto \mathbb{Z}$ maps edges to integers.

   Mean-Payoff function. Given (Γ, w), the *mean-payoff function* $MP(\omega, w, \Gamma)$ maps a play ω and a weight function w to the long-run average weight of the play: $MP(\omega, w, \Gamma) = \liminf_{n \mapsto \infty} \frac{1}{n} \cdot \sum_{i=0}^{n-1} w(v_i, v_{i+1})$.

Mean-payoff objectives. Given a threshold $\nu \in \mathbb{Q}$ and a weight function w, the *threshold mean-payoff objective* includes plays with a mean-payoff value of at least ν, i.e., MeanPayoff$(\nu, w, \Gamma) = \{\omega \in \Omega \mid MP(\omega, w, \Gamma) \geq \nu\}$. When Γ is a game graph, (Γ, w) is a *mean-payoff game*.

7. *Threshold mean-payoff parity objectives.* Given a priority function p and a weight function w in a game graph Γ we call the triple (Γ, p, w) a *mean-payoff parity game*.

   Mean-Payoff Parity function. Given (Γ, p, w) the *mean-payoff parity function* maps every play in Γ to a real-number or $-\infty$ as follows: If the play satisfies the parity objective, then the value of the play is the mean-payoff value, else it is $-\infty$. Formally, for a play ω we have

$$MPP(\omega, p, w, \Gamma) = \begin{cases} MP(\omega, w, \Gamma) & \text{if } \omega \in \text{Parity}(p, \Gamma) \\ -\infty & \text{if } \omega \notin \text{Parity}(p, \Gamma). \end{cases}$$

   Mean-Payoff Parity objective. Given a priority function p for Γ, a weight function w for Γ and a threshold ν, the *threshold mean-payoff parity objective* combines parity and mean-payoff objectives, i.e., MeanPayoffParity$(\nu, p, w, \Gamma) = \{\omega \in \Omega \mid MPP(\omega, p, w, \Gamma) \geq \nu\}$.

8. *k-pair Streett objective.* In the *Streett objective* we are given a set of k pairs of vertex sets, i.e., $\{(L_1, U_1), \ldots, (L_k, U_k)\}$ such that $L_i, U_i \subseteq V$ for $1 \leq i \leq k$. The objective includes a play if, whenever some vertex in L_i is visited infinitely often, then also some vertex of U_i is visited infinitely often (for all $1 \leq i \leq k$). More formally, Streett$(\{(L_i, U_i) \mid 1 \leq i \leq k\}, \Gamma) = \{\omega \in \Omega \mid L_i \cap Inf(\omega) = \emptyset$ or $U_i(\omega) \neq \emptyset$ for all $1 \leq i \leq k\}$.

We sometimes omit the MDP or game graph Γ from the objective when it is obvious on which game graph the objective is defined on.

2.4 Winning Strategies, Winning Sets, Queries, and Value Functions

Winning Strategies and Winning Sets. In MDPs, a player-1 strategy σ is *almost-sure (a.s.) winning* from a starting vertex v for an objective ϕ iff $\Pr_v^\sigma(\phi) = 1$. The *winning set* $\langle\langle 1 \rangle\rangle_{a.s.}(\phi)$ for player 1 is the set of vertices from which player 1 has an almost-sure winning strategy.

In game graphs, given an objective $\Phi \subseteq \Omega$ for player 1, a strategy $\sigma \in \Sigma$ is a *winning strategy for player 1* from vertex v if for all player-2 strategies $\pi \in \Pi$ the play $\omega(v, \sigma, \pi)$ satisfies Φ. We define the winning strategies of player 2 analogously. A *vertex is winning* for player 1 for Φ if player 1 has a winning strategy from v. Formally, the *set of winning vertices for player 1* for the objective Φ is $W_1(\Phi) =$

$\{v \in V \mid \exists \sigma \in \Sigma \text{ s.t. } \forall \pi \in \Pi : \omega(v, \sigma, \pi) \in \Phi\}$. We define the set of winning vertices for player 2 with respect to the objective Φ with $W_2(\Phi) = \{v \in V \mid \exists \pi \in \Pi \text{ s.t. } \forall \sigma \in \Sigma : \omega(v, \sigma, \pi) \in \Phi\}$. Due to a seminal result by Martin [182] every vertex in V belongs to the winning set of player 1 or the winning set of player 2 and, thus, the two sets form a partition of V. We say that a vertex is either *winning for player 1* if it is in $W_1(\cdot)$ or *winning for player 2* if it is in $W_2(\cdot)$.

Coverage and AllCoverage. In the *coverage problem* the input consists of the vertex sets $T_1, \ldots, T_k$ and a start vertex s. The coverage problem is not a single objective but is a query involving several objectives. That is, for k different vertex sets, namely $T_1, T_2, \ldots, T_k$, the coverage query $Coverage(T_1, \ldots, T_k)$ asks whether s is a winning vertex for player 1 for all reachability objective Reach(T_i) ($1 \leq i \leq k$). If s is winning for player 1 for all the reachability objectives we say that s winning for player 1 regarding the query $Coverage(T_1, \ldots, T_k)$. In the *AllCoverage* problem the input are target sets $T_1, T_2, \ldots, T_k$ and we determine the player-1 winning set of the coverage problem, i.e., all vertices with a player-1 winning strategy for $Coverage(T_1, \ldots, T_k)$.

Value Functions. Given a payoff function f and a vertex v (such as the mean-payoff function, or the mean-payoff parity function), the value for player 1 at v is the maximal payoff that she can guarantee against all strategies of player 2. Formally,

$$val(f)(v) = \sup_{\sigma \in \Sigma} \inf_{\pi \in \Pi} f(\omega(v, \sigma, \pi)).$$

2.5 Basic Algorithmic Results

In this section, we introduce special subsets of vertices for the models and basic algorithmic concepts which we need to prove the theoretical results in the following chapters.

2.5.1 Graphs: SCCs, decremental algorithms and Graph Reachability

For the following definitions we are given a graph $G = (V, E)$. *Strongly connected components (SCCs), bottom SCCs and the condensation of a Graph* A set of vertices $X \subseteq V$ forms a *strongly connected subgraph* (SCS) if the induced subgraph $G[X]$ is strongly connected. An SCS is *trivial* if it contains a single vertex only and all other SCSs are *non-trivial*. A *strongly connected component* (SCCS) is a set of vertices C such that $G[C]$ is an SCS and C is a maximal set (under set inclusion) in V such that $G[C]$ is an SCS. The SCC C is a *bottom SCC* if no vertex $v \in C$ has an edge to a vertex in $V \setminus C$. The *SCC decomposition* partitions the vertices of V into the corresponding SCCs and can be determined in $O(m)$ time [216]. The condensation of G, denoted by $\mathsf{CONDENSE}(G)$ is the graph where all vertices in the same SCC in G are contracted: We call the vertices of $\mathsf{CONDENSE}(G)$ *nodes* to distinguish them from the vertices in G.

Graph Reachability. Let $GraphReach(S, G)$ be the set of vertices in G that can reach a vertex of $S \subseteq V$. Depth-first search determines the set $GraphReach(S, G)$ in linear time [216].

Decremental Graph Algorithm. A *decremental graph algorithm* is a data structure which supports the deletion of *player-1 edges* while it maintains the solution to a graph problem. A decremental graph algorithm usually allows three kinds of operations: (1) *preprocessing*, which is computed when it receives the initial input graph, (2) *delete*, which deletes a player-1 edge and updates the data structure, and (3) *query*, which computes the answer to the problem. The *query time* is the time that the decremental graph algorithm needs to compute the answer to the query. The *update time* of a decremental algorithm is the running time for a single *delete* operation. We sometimes refer to the delete operations as *update operation*. The running time of a decremental algorithm is characterized by the *total update time*, i.e., the sum of the update time over the worst-case sequence of deletions. Sometimes a decremental algorithm is randomized and the provided running time guarantees hold for an *oblivious adversary* who fixes the sequence of updates in advance. When we use a randomized decremental algorithm assuming an oblivious adversary as a subprocedure, the sequence of deleted edges must not depend on the random choices of the decremental algorithm.

2.5.2 MDPs: MECs and Random Attractors

For the following definitions let $P = (V, E, \langle V_1, V_R \rangle, \delta)$ be an MDP.

Maximal End-Components. An *end-component* is a set of vertices $X \subseteq V$ such that (1) $P[X]$ strongly connected and (2) all *random vertices* have their outgoing edges in X, that is, for all $v \in X \cap V_R$ and all $(v, u) \in E$ we have $u \in X$. An end-component is *trivial* if it has size one. All other end-components are *non-trivial*. An end-component maximal under set inclusion is a *maximal end-component (MEC)*. MECs generalize SCCs in graphs and, in a MEC X, player-1 can *almost-surely reach* (reach with probability 1) all vertices $u \in X$ from every vertex $v \in X$ because random vertices do not leave X. The MEC-decomposition of an MDP is the partition of V into MECs and the set of vertices which do not belong to any MEC. Every bottom SCC C in (V, E) is a MEC because no vertex (and thus no random vertex) has an outgoing edge.

Example 2.5.1 (Key difference of SCCs and MECs)**.** *In the SCC decomposition each vertex belongs to exactly one SCC (which might be a trivial SCC just containing that vertex) but for the MEC decompositions a non-empty set of random vertices which do not belong to any MEC can exist (still each vertex belongs to at most one MEC). Consequently, if we contract each MECs into a player-1 vertex, the resulting MDP is not necessarily acyclic which is in contrast to the acyclic condensation graph that we obtain from contracting the SCCs. In Figure 2.1 we demonstrate this key difference: When we contract all SCCS of the graph G into player-1 vertices yields the DAG G'. Consider the MDP P where the vertices $\{v_1, v_2, v_4, v_7\}$ of G are random vertices and all*

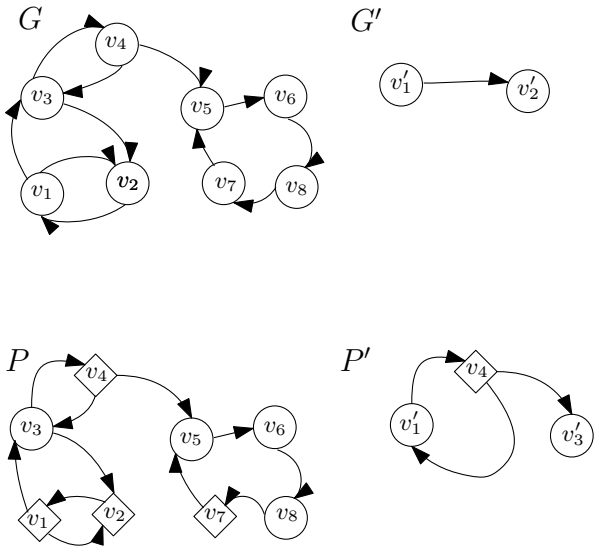


Figure 2.1: A graph G and an MDP P where we contract all SCCs and MECs respectively into player-1 vertices (removing self-loops) to obtain G' and P'. The graph G' is acyclic whereas the MDP P' contains a cycle.

edges remain unchanged. If we contract the MECS, i.e. $\{\{v_1, v_2, v_3\}, \{v_5, v_6, v_7, v8\}\}$ *into player-1 vertices* $\{v'_1, v'_3\}$ *we obtain the MDP* P' *which has a cycle. Note that this is because* v_4 *does not belong to a MEC and is strongly connected with* v'_1 *or* v_3 *respectively in* P' *and* P.

Random attractor. A *random attractor* $attr_R(T, P)$ of a vertex set T in P is a vertex set. It includes all vertices (1) in T, (2) random vertices with an edge to the random attractor and player-1 vertices with all outgoing edges in the random attractor. We define the random attractor $A = attr_R(T, P)$ inductively as follows: $A_0 = T$ and $A_{i+1} = A_i \cup \{v \in V_R \mid Out(v) \cap A_i \neq \emptyset\} \cup \{v \in V_1 \mid Out(v) \subseteq A_i\}$ for all $i > 0$. Due to [28, 147] we can compute the random attractor $A = attr_R(T, P)$ of a set T in time $O(\sum_{v \in A} In(v))$.

Reachability in MDPs. Given a vertex set T, the set of vertices from which T can be *reached almost-surely* (i.e., with probability 1) can be computed in $O(m)$ time given the MEC-decomposition of P [86, Theorem 4.1].

2.5.3 Game Graphs: Closed Sets, Attractors

For the following definitions, let $\Gamma = (V, E, \langle V_1, V_2 \rangle)$ be a game graph.

Closed Sets. A set $U \subseteq V$ of vertices is a *closed set for player 1* if the following two conditions hold.

1. For all vertices $u \in (U \cap V_1)$ we have $Out(u) \subseteq U$, that is, all successors of player-1 vertices are again in U and

2. For all $u \in (U \cap V_2)$ we have that $Out(u) \cap V_2 \neq \emptyset$, that is, every player-2 vertex in U has a successor in U.

Player-2 closed sets are defined analogously by exchanging roles of player 1 and player 2. Every closed set U for player $x \in \{1, 2\}$ induces a subgame graph which we denote $\Gamma \restriction U$.

The following proposition establishes the connection between closed sets, winning for safety, reachability, and coBüchi objectives. The proof of the proposition is straightforward and can be found in [91].

Proposition 2.5.2 ([91]). *Consider a game graph Γ, and a closed set U for player 1. Then, the following assertions hold:*

1. *Player 2 has a winning strategy for the objective Safety(U) for all vertices in U, that is, player 2 can ensure that if the play starts in U, then the play never leaves the set U.*

2. *For all $T \subseteq V \setminus U$, we have $W_1(Reach(T)) \cap U = \emptyset$, that is, for any set T of vertices outside U, player 1 does not have a strategy from vertices in U to ensure to reach T.*

3. *If $U \cap B = \emptyset$ (i.e., there is no Büchi vertex in U), then every vertex in U is winning for player 2 for the coBüchi objective.*

Attractors and Reachability in Game Graphs. Given a set of vertices $T \subseteq V$, the set of vertices from which player x can reach T against all strategies of the other player, is the *player-x attractor* of T, i.e., $attr_x(T, \Gamma) = W_x(\text{Reach}(T, \Gamma))$. Formally, the player-$x$ attractor ($x \in \{1, 2\}$) $attr_x(T, \Gamma)$ of a given vertex set T is the limit of the sequence $A_0 = T$; $A_{i+1} = A_i \cup \{v \in V_x \mid Out(v) \cap A_i \neq \emptyset\} \cup \{v \in V_{\bar{x}} \mid Out(v) \subseteq A_i\}$ for all $i \geq 0$. The running time for computing an attractor $A = attr_x(T, \Gamma)$ is $O(m)$ [28, 147].

The following observation connects closed sets and attractors.

Observation 2.5.3 ([91]). *For all game graphs Γ, all players $\ell \in \{1, 2\}$, and all sets $U \subseteq V$ we have the following: The set $V \setminus attr_\ell(U, \Gamma)$ is a closed set for player ℓ, i.e., no player ℓ vertex in $V \setminus attr_\ell(U, \Gamma)$ has an edge to $attr_\ell(U, \Gamma)$ and every vertex of the other player in $V \setminus attr_\ell(U, \Gamma)$ has an edge in $V \setminus attr_\ell(U, \Gamma)$.*

2.6 Symbolic Model of Computation

In the *(set-based) symbolic model of computation*, we store the model with implicitly represented sets of vertices and edges. An *symbolic algorithm* accesses vertices and edges of the MDP not explicitly but with *(set-based) symbolic operations*. We characterize the resources in the symbolic model of computation by the *number of (set-based) symbolic operations* and *(set-based) symbolic space*.

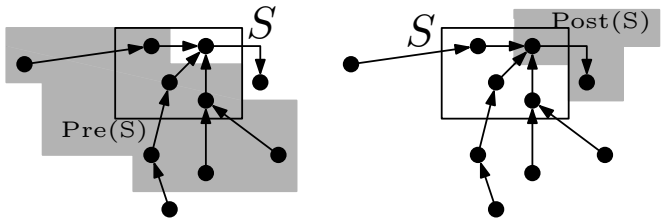


Figure 2.2: Illustration of the one-step operations.

Set-Based Symbolic Operations. A symbolic algorithm can use the same mathematical, memory, access, and logical operations as a regular RAM algorithm, except for the access to the graph.

An input model with vertices V and edges E can be accessed only by the following types of operations:

1. The algorithm can combine two sets of vertices or edges with *basic set operations*: $\cup, \cap, \subseteq, \setminus, \times$ and $=$.

2. To obtain the predecessors/successors of a vertex set S with regard to the edge set E, the algorithm uses the *one-step operation*. The predecessor and successor operation of a vertex set S over a specific edge set E are:

$$Pre_E(S) = \{v \in V \mid Out(v) \cap S \neq \emptyset\} \text{ and}$$
$$Post_E(S) = \{v \in V \mid In(v) \cap S \neq \emptyset\}$$

3. For a vertex set S, the $Pick(S)$ operation which returns an arbitrary vertex and the cardinality operation $|S|$ which returns cardinality of S.

4. When the model is a game graph, an algorithm can use the *controllable predecessor operation* of a vertex set S (z and $\bar{z}$ denote the two players)

$$CPre_z(S) = \{v \in V_z \mid Out(v) \cap S \neq \emptyset\} \cup \{v \in V_{\bar{z}} \mid Out(v) \subseteq S\}.$$

We can express the set $CPre_z(S)$ using only $Pre_E(\cdot)$ and basic set operations.

Sometimes we omit the subscript E of the one-step operations when the edge set is clear from the context.

Set-based Symbolic Space. The basic unit of space for a symbolic algorithm is a set [47, 73]. For example, a BDD can represent a set symbolically [51, 52, 55, 79, 105, 107, 108, 132, 211]. It is hard to correlate the size of the set with the size of the BDD: Consider for example a model whose state-space consists of valuations of N-boolean variables. The set of all vertices is simply represented as a true BDD. Also, the set of all vertices where the kth bit is false is represented by a BDD which depending

on the value of the kth bit chooses true or false. Again, a constant-size BDD represents this set. Therefore, even large sets of vertices can sometimes be represented as constant-size BDDs. In general, the size of the smallest BDD representing a set is computationally hard to determine and depends on the variable reordering [105]. We represent each set, thus, as a unit data structure to obtain a clean theoretical model for the algorithmic analysis. It follows that the *symbolic space requirements* of a symbolic algorithm, is the *maximal number of sets* the algorithm stores simultaneously.

2.7 Conjectured Lower Bounds

Many results from classical complexity are based on standard complexity-theoretic assumptions, e.g., P $\neq$ NP. Likewise, we derive polynomial lower bounds which are based on widely believed, conjectured lower bounds on well-studied algorithmic problems. The lower bounds we derive depend on the popular conjectures below:

We first consider the conjectures on Boolean Matrix Multiplication [228, Theorem 6.1] and *triangle detection* in graphs [5, Conjecture 2]. A *triangle* in a graph is a triple x, y, z of vertices such that $(x, y), (y, z), (z, x) \in E$. In *triangle detection* we the input is a graph and the question is if a triangle exists in the graph. We remove, with linear-time preprocessing, all self-loops in instances of *triangle detection* exist.

Remark 2.7.1 (Combinatorial algorithm). The notion of combinatorial algorithm is often used in the field of fine-grained complexity community [4, 45, 177], despite the lack of a formal definition. Intuitively, combinatorial algorithms do not use fast matrix multiplication [172, 227]. While non-combinatorial algorithms have the matrix multiplication exponent ω in the running time. To the best of our knowledge, all algorithms for deciding (almost-sure) winning conditions in game graphs and MDPs are combinatorial. Therefore, lower bounds for combinatorial algorithms are of particular interest in our setting. For more details on the notion of a combinatorial algorithm, we direct the reader to [27, 141].

Conjecture 2.7.2 (Comb. Boolean Matrix Multiplication Conjecture (BMM))**.** *A $O(n^{3-\epsilon})$ time combinatorial algorithm for computing the boolean product of two $n \times n$ matrices for any $\epsilon > 0$ does not exist.*

Conjecture 2.7.3 (Strong Triangle Conjecture (STC))**.** *Neither a $O(\min\{n^{\omega-\epsilon}, m^{2\omega/(\omega+1)-\epsilon}\})$ expected time nor a $O(n^{3-\epsilon})$ ($\epsilon > 0$) time combinatorial algorithm that can detect whether a graph contains a triangle exist. ($\omega < 2.373$ is the matrix multiplication exponent.)*

Williams and Williams [228, Theorem 6.1] showed that the BMM is equivalent to the combinatorial part of STC. Also, if we do not restrict ourselves to combinatorial algorithms, STC, still gives a super-linear lower bound.

We also consider the Strong Exponential Time Hypothesis (SETH) used in [5, Conjecture 1] and introduced by [148, 149] for the satisfiability problem of propositional logic and the Orthogonal Vector Conjecture.

Conjecture 2.7.4 (Strong Exponential Time Hypothesis (SETH))**.** *For $\epsilon > 0$ there is a k such that k-CNF-SAT on n variables and m clauses cannot be solved in $O(2^{(1-\epsilon)n} poly(m))$ time.*

The Orthogonal Vectors Problem (OV). Given two sets S_1, S_2 of d-bit vectors with $|S_1| = |S_2| = N$ and $d = \omega(\log N)$, are there $u \in S_1$ and $v \in S_2$ such that $\sum_{i=1}^{d} u_i \cdot v_i = 0$?

Conjecture 2.7.5 (Orthogonal Vectors Conjecture (OVC))**.** *A $O(N^{2-\epsilon})$ time algorithm for the Orthogonal Vectors Problem for any $\epsilon > 0$ does not exist.*

The SETH implies the OVC [226, Theorem 5]. An explicit reduction is given in the survey article [222, Theorem 3.1]. Whenever a problem is provably hard assuming OVC, is also hard when assuming SETH.

Remark 2.7.6. The conjectures make sure that no *polynomial* improvements over the best-known running times are possible but do not exclude improvements by sub-polynomial factors such as poly-logarithmic factors or factors like $2^{\sqrt{\log n}}$.

CHAPTER 3

Faster Algorithms for Mean-Payoff Parity Games

In this chapter, we consider computing the winning region for threshold mean-payoff parity objectives, and the value function for mean-payoff parity objectives.

3.1 Introduction

Graph games in Reactive Synthesis. There has been a long history of using graph games for modeling and synthesizing reactive processes [54, 193, 198]: a reactive system and its environment represent the two players, whose states and transitions are specified by the vertices and edges of a game graph. Consequently, graph games provide the theoretical foundation for modeling and synthesizing reactive processes.

Qualitative and quantitative objectives. For reactive systems, the objective is given as a set of desired paths (such as ω-regular specifications), or as a quantitative optimization objective with a payoff function on the paths. The class of ω-regular specifications provides a robust framework to express all commonly used specifications for reactive systems in verification and synthesis. Parity objectives are a canonical way to express ω-regular objectives [219], where an integer priority is assigned to every vertex, and a path satisfies the parity objective for player 1 if the minimum priority visited infinitely often is even. One of the classical and most well-studied quantitative objectives is the mean-payoff objective, where a reward is associated with every edge, and the payoff of a path is the long-run average of the rewards of the path.

Mean-payoff parity objectives. Traditionally the verification and the synthesis problems were considered with qualitative objectives. However, recently combinations

of qualitative and quantitative objectives have received a lot of attention. Qualitative objectives such as ω-regular objectives specify the functional requirements of reactive systems, whereas the quantitative objectives specify resource consumption requirements (such as for embedded systems or power-limited systems). Combining quantitative and qualitative objectives is crucial in the design of reactive systems with both resource constraints and functional requirements [34, 41, 67, 83]. For example, mean-payoff parity objectives are relevant in the synthesis of optimal performance lock-synchronization for programs [66], where one player is the synchronizer, the opponent is the environment; the performance criteria are specified as mean-payoff objective; and the functional requirement (e.g., data-race freedom or liveness) as an ω-regular objective. Mean-payoff parity objectives have been used in several other applications, e.g., defining permissivity for parity games [42] and for the robustness in synthesis [35].

Threshold and value problems. For graph games with mean-payoff and parity objectives, there are two variants of the problem. First, the *threshold* problem, where a threshold ν is given for the mean-payoff objective and player 1 must ensure the parity objective and that the mean-payoff is at least ν. Second, the *value* problem, where player 1 maximizes the mean-payoff value while ensuring the parity objective. In the sequel of this section, we will refer to graph games with mean-payoff and parity objectives as mean-payoff parity games.

Previous results. Mean-payoff parity games were first studied in [83], and algorithms for the value problem were presented. It was shown in [71] that the decision problem for mean-payoff parity games lies in NP $\cap$ coNP (similar to the status of mean-payoff games and parity games). For game graphs with n vertices, m edges, parity objectives with d priorities, and maximal absolute reward value W for the mean-payoff objective, the previous known algorithmic bounds for mean-payoff parity games are as follows: For the threshold problem, the results of [71] give an $O(n^{d+4}mdW)$-time algorithm. This algorithmic bound was improved in [42] where an $O(n^{d+2}mW)$-time algorithm was presented for the value problem. The result of [42] does not explicitly present any other better bound for the threshold problem. However, the recursive algorithm of [42] uses value mean-payoff games as a sub-routine, and replacing value mean-payoff games with threshold mean-payoff games gives an $O(n)$-factor saving, and yields an $O(n^{d+1}mW)$-time algorithm for the threshold problem for mean-payoff parity games.

Contributions. In this chapter, our main contributions are faster algorithms to solve mean-payoff parity games. Previous and our results are summarized in Table 3.1.

1. *Threshold problem.* We present an $O(n^{d-1}mW)$-time algorithm for the threshold problem for mean-payoff parity games, improving the previous $O(n^{d+1}mW)$ bound. The important special case of parity objectives with two priorities correspond to Büchi and coBüchi objectives. Our bound for mean-payoff Büchi games and mean-payoff coBüchi games is $O(nmW)$, which matches the best-

known bound to solve the threshold problem for mean-payoff objectives [44], and improves the previous known $O(n^3mW)$ bound [42].

2. *Value problem.* We present an $O(n^d mW \log(nW))$-time algorithm for the value problem for mean-payoff parity games, improving the previous $O(n^{d+2}mW)$ bound. Our bound for mean-payoff Büchi games and mean-payoff coBüchi games is $O(n^2mW \log(nW))$, which matches the bound of [44] to solve the value problem for mean-payoff objectives, and improves the previous known $O(n^4mW)$ bound.

Technical contributions. Our main technical contributions are as follows:

1. First, for the threshold problem, we present a decremental algorithm for mean-payoff games that supports a sequence of vertex-set deletions along with their player-2 reachability set. We show that the total running time is $O(nmW)$, which matches the best-known bound for the static algorithm to solve mean-payoff games. We show that using our decremental algorithm we can solve the threshold problem for mean-payoff Büchi games in time $O(nmW)$.

2. Second, for mean-payoff coBüchi games, the decremental approach does not work. We present a new static algorithm for threshold mean-payoff games that identifies subsets X of the winning set for player 1, where the time complexity is $O(|X|mW)$, i.e., it replaces n with the size of the set identified. We show that with our new static algorithm we can solve the threshold problem for mean-payoff coBüchi games in time $O(nmW)$.

3. Finally, we show for all mean-payoff parity objectives, given an algorithm for the threshold problem, that the value problem can be solved in time $n \log(nW)$ times the complexity of the threshold problem.

Related works. The problem of graph games with mean-payoff parity objectives was first studied in [83]. The NP ∩ coNP complexity bound was established in [71], and an improved algorithm for the problem was given in [42]. The mean-payoff parity objectives have also been considered in other stochastic setting such as Markov decision processes [70, 84] and stochastic games [85]. The algorithmic approaches for stochastic games build on the results for non-stochastic games. In this chapter, we present faster algorithms for mean-payoff parity games.

3.2 Decremental Algorithm for Threshold Mean-Payoff Games

In this section, we present a decremental algorithm for threshold mean-payoff games that supports deleting a sequence of sets of vertices along with their player-2 attractors. The overall running time of the algorithm is $O(nmW)$.

	threshold problem		value problem	
	Previous	New	Previous	New
MP-Büchi	$O(n^3mW)$	$O(nmW)$	$O(n^4mW)$	$O(n^2mW \log(nW))$
MP-coBüchi	$O(n^3mW)$	$O(nmW)$	$O(n^4mW)$	$O(n^2mW \log(nW))$
MP-parity	$O(n^{d+1}mW)$	$O(n^{d-1}mW)$	$O(n^{d+2}mW)$	$O(n^dmW \log(nW))$

Table 3.1: Algorithmic bounds for mean-payoff (MP) and parity objectives: In a game graph Γ with weight function w, n denotes the number of vertices, m denotes the number of edges, d denotes the number of priorities of the parity function p, and W is the maximum absolute value of the weight function w.

Key idea. A static algorithm based on the notion of progress measure for mean-payoff games was presented in [44]. We show that the progress measure is monotonic with regard to the deletion of vertices and their player-2 attractors. We use an amortized analysis to obtain the running time of our algorithm.

Mean-payoff progress measure. Let (Γ, w) be a mean-payoff game with threshold ν. Progress measure is a function f which maps every vertex in Γ to an element of the set $C_\Gamma = \{i \in \mathbb{N} \mid i \leq nW\} \cup \{\top\}$, i.e., $f : V \mapsto C_\Gamma$. Let $(\preceq, C_\Gamma)$ be a total order, where $x \preceq y$ for $x, y \in C_\Gamma$ holds iff $x \leq y \leq nW$ or $y = \top$. We define the operation $\ominus : C_\Gamma \times \mathbb{Z} \mapsto C_\Gamma$ for all $a \in C_\Gamma$ and $b \in \mathbb{Z}$ as follows:

$$a \ominus b = \begin{cases} \max(0, a-b) & \text{if } a \neq \top \text{ and } a - b \leq nW, \\ \top & \text{otherwise.} \end{cases}$$

A player-1 vertex v is *consistent* if $f(v) \geq f(v') \ominus w(v, v')$ for *any* $v' \in Out(v)$. A player-2 vertex v is *consistent* if $f(v) \geq f(v') \ominus w(v, v')$ for *all* $v' \in Out(v)$. Let $v \in V$, we define $\mathrm{Lift}(f, v) : [V \mapsto C_\Gamma \times V] \mapsto [V \mapsto C_\Gamma]$ as $\mathrm{Lift}(f, v) = g$ where:

$$g(u) = \begin{cases} f(u) & \text{if } u \neq v, \\ \min\{f(v') \ominus w(v, v') \mid (v, v') \in E\} & \text{if } u = v \text{ and } v \in V_1, \\ \max\{f(v') \ominus w(v, v') \mid (v, v') \in E\} & \text{if } u = v \text{ and } v \in V_2. \end{cases}$$

Static Algorithm. The static algorithm by Brim et al. [44] is an iterative algorithm that maintains and returns a progress measure f and a list L of vertices that are not consistent. The initial progress measure of every vertex is set to zero. Also, all the weights of all edges are subtracted by the value ν, i.e., $w(e) \leftarrow w(e) - \nu$ for all edges e in E. The list L is initialized with the vertices which are not consistent considering the initial progress measure. Then the following steps are executed in a while-loop:

1. If L is empty, return f.
2. Take out a vertex v of L.

3. Perform the Lift-operation on the vertex, i.e., $f \leftarrow \text{Lift}(f, v)$.

4. If a vertex v' in $In(v)$ is not consistent, put v' into L.

If every vertex is consistent, i.e., the list L is empty, Brim et al. show that the winning region of player 1 is the set of vertices which are not set to $\top$ in f, i.e., $W_1(\text{MeanPayoff}(\nu, w, \Gamma)) = \{v \in V \mid f(v) \neq \top\}$.

Decremental input/output. Let (Γ, w) be a mean-payoff game with threshold ν. The *input* to the decremental algorithm is a sequence of sets $A_1, A_2, \ldots, A_k$, such that each A_i is a player-2 attractor of a set X_i in the game $\Gamma_i = \Gamma \upharpoonright (V \setminus \bigcup_{j<i} A_j)$. The *output requirement* is the player-1 winning set after the deletion of $\bigcup_{j<i} A_j$ for $i = 1, \ldots, k$, i.e., the output requirement is the sequence $Z_1, Z_2, \ldots, Z_k$, where $Z_i = W_1(\text{MeanPayoff}(\nu, w, \Gamma_i))$ in $\Gamma_i = \Gamma \upharpoonright (V \setminus \bigcup_{j<i} A_j)$. In other words, we repeatedly delete a vertex set X_i along with its player-2 attractor A_i from the current game graph Γ_i, and require the winning set of player 1 for the mean-payoff objective as an output after each deletion.

Decremental algorithm. We maintain a progress measure f_i, $1 \leq i \leq k$, during the whole sequence of deletions. The initial progress measure f_1 for the mean-payoff game (Γ, w) with threshold mean-payoff objective $\text{MeanPayoff}(\nu, w, \Gamma)$ is computed with the static algorithm of Brim et al [44].

For all edges e in E, we set $w(e) \leftarrow w(e) - \nu$. In iteration i with input A_i, in the game Γ_i with its corresponding vertex set V_i the following steps are executed:

1. If a vertex in the set $\{v \in V_i \setminus A_i \mid \exists v' : v' \in Out(v) \text{ and } v' \in A_i\}$ is not consistent in f_i without the set A_i, put it in the list L_i.

2. Delete the set A_i from Γ_i to receive Γ_{i+1} (and thus V_{i+1}).

3. Execute the while-loop with steps (1) — (4) of the above described iterative algorithm by Brim et al. [44] initialized with Γ_{i+1}, L_i and f_i restricted to the vertices in V_{i+1} to obtain f_{i+1}.

4. Output the set $\{v \in V_{i+1} \mid f(v) \neq \top\}$ from f_{i+1}.

Correctness. Let (Γ, w) be a mean-payoff game, $\text{MeanPayoff}(\nu, w\Gamma)$ be a threshold objective and $A_1, A_2, \ldots, A_k$ a sequence of sets, such that each A_i is a player-2 attractor in the game $\Gamma_i = \Gamma \upharpoonright (V \setminus \bigcup_{j<i} A_j)$. To show the correctness of the decremental algorithm we show that the condition that the list L contains all vertices which are not consistent is an invariant of the decremental algorithm at line 3. This property was proved for the static algorithm in [44].

Lemma 3.2.1. *The condition that L_i contains all vertices which are not consistent with the progress measure f_i restricted to V_{i+1} in Γ_{i+1} is an invariant of the static algorithm called in step 3 of the decremental algorithm for $1 \leq i \leq k-1$.*

Proof. The fact that the static algorithm correctly returns a progress measure with only consistent vertices when the invariant holds was shown in [44]. It was also shown in [44] that the invariant is maintained in the loop. It remains to show that the condition holds when we call the static algorithm at step 3. For the base case, let $i = 1$. In the initial progress measure f_1 and the initial game graph Γ_1, every vertex is consistent. By the definition of a player-2 attractor, deleting the set A_1 potentially removes edges (v, v') where v is a player-1 vertex in $V \setminus A_1$ and v' is in A_1. (Note that v cannot be a player-2 vertex.) All of the vertices not consistent anymore are added to L_i in step 1 of the decremental algorithm. For the inductive step let $i = j$. By the induction hypothesis, all vertices which were not consistent with the progress measures f_{h-1} restricted to V_h for $2 \leq h \leq j$ were added to the corresponding lists. Thus by the correctness of the static algorithm, it correctly computes the new progress measure f_h for the game graph Γ_h where every vertex is consistent. Thus also every vertex in the progress measure f_j restricted to V_j is consistent. Again the player-2 attractor is removed and vertices that are not consistent with progress measure f_j restricted to V_{j+1} are put into L_j by step 1 of the algorithm. □

Thus we proved that the static algorithm always correctly updates to the new progress measure in each iteration. The winning region of player-1 is obtained by the returned progress measure (step 4). The decremental algorithm thus correctly computes the sequence $Z_1, Z_2, \ldots Z_k$, where $Z_i = W_1(\text{MeanPayoff}(\nu, w, \Gamma_i))$.

Running Time. The calculation of the initial progress measure for the mean-payoff game Γ with threshold ν is in time $O(nmW)$. The vertices which are not consistent anymore after the deletion of A_i can be found in time $O(m)$ (step 1). As at most n such sets A_i exist, the running time is $O(mn)$. In step 3 the static algorithm is executed with the current progress measure f_i: Every time a vertex v is picked from the list L_i it costs $O(|Out(v) + In(v)|)$ time to use Lift on it and to look for vertices in $In(v)$ which are not consistent anymore (steps 1–3 in the static algorithm). We charge the cost for each vertex to its incident edges. Note that deleting a set of vertices and their corresponding player-2 attractor will only potentially *increase* the progress measure of some player-1 vertices. As we can increase the progress measure of every vertex only nW times before it is set to $\top$ where it is always consistent, we get the desired time bound of $O(mnW)$.
Thus our decremental algorithm for threshold mean-payoff games works as desired and we obtain the following result:

Theorem 3.2.2. *Given a mean-payoff game (Γ, w), a threshold mean-payoff objective ϕ and a sequence of sets $A_1, A_2, \ldots, A_k$ such that each A_i is a player-2 attractor of a set X_i in the game $\Gamma_i = \Gamma \restriction (V \setminus \bigcup_{j<i} A_j)$, the sequence $Z_1, Z_2, \ldots, Z_k$, where $Z_i = W_1(\phi)$ in Γ_i can be computed in $O(nmW)$ time.*

Remark 3.2.3. Note that the running time analysis of our decremental algorithm crucially depends on the monotonicity property of the progress measure. If edges are both added and deleted, then the monotonicity property does not hold. Hence

obtaining a fully dynamic algorithm that supports both addition/deletion of vertices/edges with running time $O(nmW)$ is an interesting open problem. However, we will show that for solving mean-payoff parity games, the decremental algorithm plays a crucial part.

3.3 Threshold Mean-Payoff Parity Games

In this section, we present algorithms for threshold mean-payoff parity games. Our most interesting contributions are for the base case of mean-payoff Büchi objectives and mean-payoff coBüchi objectives, and the general case follows a standard recursive argument.

3.3.1 Threshold Mean-Payoff Büchi Games

In this section, we consider threshold mean-payoff Büchi games.

Algorithm for threshold mean-payoff Büchi games. The basic algorithm is an iterative algorithm that deletes player-2 attractors. The algorithm proceeds in iterations. In iteration i, let D_i be the set of vertices already deleted. Consider the subgame $\Gamma_i = \Gamma \restriction (V \setminus D_i)$. Then the following steps are executed:

1. Let $V^i = V \setminus D_i$ and B_i denote the set of Büchi vertices (or vertices with priority 0) in Γ_i. Compute $Y_i = attr_1(B_i, \Gamma_i)$ the player-1 attractor to B_i in Γ_i.

2. Let $X_i = V^i \setminus Y_i$. If X_i is non-empty, remove $A_i = attr_2(X_i, \Gamma_i)$ from Γ_i, and proceed to the next iteration in Step 1 otherwise go to Step 3.

3. Else $V^i = Y_i$. Compute $U_i = W_1(\text{MeanPayoff}(v, w, \Gamma_i))$, i.e., the winning region for the threshold mean-payoff objective in Γ_i. Let $X_i = V^i \setminus U_i$. If X_i is non-empty, remove $A_i = attr_2(X_i, \Gamma_i)$ from the game graph Γ_i, and proceed to the next iteration. If X_i is empty, then the algorithm stops and all the remaining vertices are winning for player 1 for the threshold mean-payoff Büchi objective.

Correctness. Since the correctness argument has been used before [83], we only present a brief sketch: The basic correctness argument shows for $i > 0$ that all vertices removed from Γ_i do not belong to the winning set for player 1. In the end, for the remaining vertices, player 1 can ensure to reach the Büchi vertices, and ensures the threshold mean-payoff objectives. A strategy that plays for the threshold mean-payoff objectives longer and longer, and in between visits the Büchi vertices, ensures that the threshold mean-payoff Büchi objective is satisfied.

Running time analysis. We observe that the total running time to compute all attractors is at most $O(nm)$, since the algorithm runs for $O(n)$ iterations and each attractor computation is linear time. In step 3, the algorithm needs to compute the winning

region for threshold mean-payoff objective. The algorithm always removes a set X_i and its player-2 attractor A_i, and requires the winning set for player 1. Thus we can use the decremental algorithm from Section 3.2, which precisely supports these operations. Hence using Theorem 3.2.2 in the algorithm for threshold mean-payoff Büchi games, we obtain the following result.

Theorem 3.3.1. *Given a mean-payoff game graph (Γ, w) and a threshold mean-payoff Büchi objective MeanPayoff(v, w, Γ), the winning set W_1(MeanPayoff(v, w, Γ)) can be computed in $O(mnW)$ time.*

3.3.2 Threshold Mean-Payoff coBüchi Games

In this section, we will present an $O(nmW)$-time algorithm for threshold mean-payoff coBüchi games. We start with the description of the basic algorithm for threshold mean-payoff coBüchi games.

Algorithm for threshold mean-payoff coBüchi games. The basic algorithm is an iterative algorithm that deletes player-1 attractors. The algorithm proceeds in iteration. In iteration i, let D_i be the set of vertices already deleted. Consider the subgame $\Gamma_i = \Gamma \restriction (V \setminus D_i)$. Then the following steps are executed:

1. Let $V^i = V \setminus D_i$ and C_i denote the set of non coBüchi vertices (the set of vertices player 1 must avoid to visit infinitely often, i.e., priority-1 vertices) in Γ_i. Compute $Y_i = attr_2(C_i, \Gamma_i)$ the player-2 attractor to C_i in Γ_i.

2. Let $X_i = V^i \setminus Y_i$. Consider the subgame $\widehat{\Gamma}_i = \Gamma_i \restriction X_i$ and compute the winning region of the threshhold mean-payoff objective $Z_i = W_1(\text{MeanPayoff}(v, w, \widehat{\Gamma}))$ for player 1.

3. If Z_i is non-empty, remove $attr_1(Z_i, \Gamma_i)$ from Γ_i, and proceed to the next iteration. Else if Z_i is empty, then all remaining vertices are winning for player 2.

Correctness argument. Consider the subgame Γ_i. In each subgame $\widehat{\Gamma}_i$ of Γ_i all edges of player 2 are intact since it is obtained after removing a player-2 attractor Y_i. Moreover, there is no priority-1 vertex in $\widehat{\Gamma}_i$. Hence, ensuring the threshold mean-payoff objective in $\widehat{\Gamma}_i$ for player 1 implies satisfying the threshold mean-payoff coBüchi objective. The set Z_i and its player-1 attractor belongs to the winning set of player 1 and can be removed. Thus, all vertices removed are part of the winning region for player 1. Upon termination, in $\widehat{\Gamma}_i$, player 1 cannot satisfy the threshold mean-payoff condition from any vertex. Consider a player-2 strategy, where in $\widehat{\Gamma}_i$ player 2 falsifies the threshold mean-payoff condition, and in Y_i plays an attractor strategy to reach C_i (the non coBüchi vertices, i.e., priority-1 vertices). Given such a strategy, either (a) Y_i is visited infinitely often, and then the coBüchi objective is violated; or (b) from some point on the play stays in $\widehat{\Gamma}_i$ forever, and then the threshold mean-payoff objective is violated. This shows the correctness of the algorithm.

However, the running time of this algorithm is not $O(nmW)$. We now present the key ideas to obtain an $O(nmW)$-time algorithm.

First intuition. Our first intuition is as follows. In step 2 of the above algorithm, instead of obtaining the whole winning region $W_1(\text{MeanPayoff}(v, w, \widehat{\Gamma}))$ it suffices to identify a subset $X_i \subseteq Z_i$ of the winning region (if it is non-empty) and remove its player-1 attractor. We call this the modified algorithm for threshold mean-payoff coBüchi games. We first describe why we cannot use the decremental approach in the following remark.

Remark 3.3.2. Consider the subgames for which the threshold mean-payoff objective must be solved. Consider Figure 3.1. The first player-2 attractor removal induces subgame $\widehat{\Gamma}_1$. After identifying a winning region X_1 of $\widehat{\Gamma}_1$ we remove its player-1 attractor A_1. After removal of A_1, we consider the second player-2 attractor to the priority-1 vertices. The removal of this attractor induces $\widehat{\Gamma}_2$. We observe comparing $\widehat{\Gamma}_1$ and $\widehat{\Gamma}_2$ that certain vertices are removed, whereas other vertices are added. Thus the subgames to be solved for threshold mean-payoff objectives do not satisfy the condition of decremental or incremental algorithms (see Remark 3.2.3).

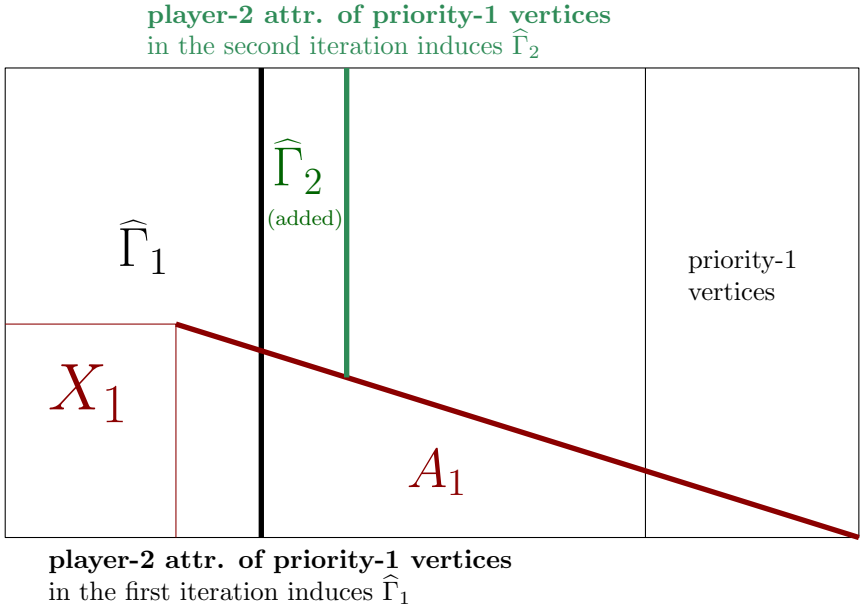


Figure 3.1: Pictorial illustration of threshold mean-payoff coBüchi games. The subgames $\widehat{\Gamma}_1$ and $\widehat{\Gamma}_2$ are shown. We observe that $\widehat{\Gamma}_2$ is obtained both by addition and deletion of game parts to $\widehat{\Gamma}_1$.

Second intuition. While we cannot use the decremental algorithm, we can solve the problem in $O(nmW)$ time, if we have a modified static algorithm for threshold mean-payoff games, with the following property: (a) it identifies a subset of the winning region X for player 1, if the winning region is non-empty, in time $O(|X|mW)$;

(b) if the winning region is empty, it returns the empty set, and then it takes time $O(nmW)$. With such an algorithm we analyze the running time of the above modified algorithm for threshold mean-payoff coBüchi games. The total time required for all attractor computations is again $O(nm)$. Otherwise, we use the modified static algorithm to remove vertices of player-1 and to remove a set of size $|X|$ we take $O(|X|mW)$ time, and thus we can charge each vertex $O(mW)$ time. Hence the total time required is $O(nmW)$. In the rest of the section, we present this modified static algorithm for threshold mean-payoff games.

Problem Statement.

Input:	Mean-payoff game (Γ, w) with threshold v.
Question:	If $W_1(\text{MeanPayoff}(v))$ is non-empty, return a nonempty set $X \subseteq W_1(\text{MeanPayoff}(v))$ in time $O(\|X\|mW)$, else return $\emptyset$ in time $O(nmW)$.

Modified static algorithm for threshold mean-payoff games. The basic algorithm for threshold mean-payoff games computes a progress measure, with a defined top element value $\top$. If the progress measure has the value $\top$ for a vertex, then the vertex is declared as winning for player 2. With value $\top = n \cdot W$, the correct winning region for both players can be identified. Moreover, for a given value α for $\top$, the progress measure algorithm requires $O(\alpha \cdot m)$ time. Our modified static algorithm is based on the following idea:

1. Consider a value $\alpha \leq n \cdot W$ for the top element. With this reduced value for the top element, if a winning region is identified for player 1, then it is a subset of the whole winning region for player 1.

2. We will iteratively double the value for the top element.

Given the above ideas our algorithm is an iterative algorithm defined as follows: Initialize top value $\top_0 = W$. The i-th iteration is as follows:

1. Run the progress measure algorithm with top value $\top_i$.

2. If a winning region X for player 1 is identified, return X.

3. Else $\top_{i+1} = 2\top_i$ (i.e., the top value is doubled).

4. If $\top_{i+1} \geq 2nW$, stop the algorithm and return $\emptyset$, else proceed to the next iteration.

Correctness and running time analysis. The key steps of the correctness argument and the running time analysis are as follows:

1. The above algorithm is correct, since if it returns a set X then it is a subset of the winning set for player 1.

2. If the algorithm returns a winning set with top value α, then the total running time till this iteration is $m \cdot (\alpha + \alpha/2 + \alpha/4 + \cdots)$, because the progress with top value α requires time $O(\alpha m)$. Hence the total running time if a set X is returned with top value α is $O(\alpha \cdot m)$.

3. Let Z be a set of vertices such that no player-2 vertex in Z has an edge out of Z, and the whole subgame $\Gamma \restriction Z$ is winning for player 1. Then a winning strategy in Z ensures that a progress measure with top value $|Z|W$ would identify the set Z as a winning set.

4. From above it follows that if the winning set X is identified at top value α, but no winning set was identified with top value $\alpha/2$, then the size of the winning set is at least $\alpha/(2W)$.

5. It follows from above that if a set X is identified, then the total running time to obtain set X is $O(|X|mW)$.

6. Moreover, the total running time of the algorithm when no set X is identified is in $O(nmW)$, and in this case, the winning region is empty.

Thus we solved the modified static algorithm for threshold mean-payoff games as desired and obtain the following result.

Theorem 3.3.3. *Let $Z = W_1(\mathit{MeanPayoff}((\Gamma, w, v)))$. If $Z \neq \emptyset$, then a non-empty set $X \subseteq Z$ can be computed in time $O(|X|mW)$, else an empty set is returned if $Z = \emptyset$, which takes time $O(nmW)$.*

Using the above algorithm to compute the winning set for player 1 in the subgames, we obtain an algorithm for threshold mean-payoff coBüchi games in time $O(nmW)$.

Theorem 3.3.4. *Given a mean-payoff cobüchi game the winning set of a threshold mean-payoff coBüchi objective, can be computed in $O(nmW)$ time.*

3.3.3 Threshold Mean-Payoff Parity Games

The algorithm for threshold mean-payoff parity games is the standard recursive algorithm [83] (classical parity game-style algorithm) that generalizes the Büchi and coBüchi cases (which are the base cases). The running time recurrence is as follows: $T(n, d, m, w) = n(T(n, d-1, m) + O(m)) + O(nmW)$. Using our approach we obtain the following result.

Theorem 3.3.5. *Given a mean-payoff parity game the winning set of a threshold mean-payoff parity objective can be computed in $O(n^{d-1}mW)$ time.*

3.4 Optimal Values for Mean-payoff Parity Games

In this section, we present an algorithm that computes the value function for mean-payoff parity games. For mean-payoff games a dichotomic search approach was presented in [44]. We show that such an approach can be generalized to mean-payoff parity games.

Range of Values for the Dichotomic Search. To describe the algorithm we recall a lemma about the possible range of optimal values of a mean-payoff parity game. The lemma is an easy consequence of the characterization of [83] that the mean-payoff parity value coincides with the mean-payoff value, and the possible range of value for mean-payoff games.

Lemma 3.4.1 ([83, 121, 175])**.** *Let (Γ, p, w) be a mean-payoff parity game. For each vertex v, the optimal value $val(MPP(\cdot, p, w, \Gamma))(v)$ is a rational number $\frac{y}{z}$ such that $1 \leq z \leq n$ and $|y| \leq z \cdot W$.*

By Lemma 3.4.1 the value of each vertex $v \in V$, is contained in the following set of rationals

$$S^{(\Gamma,p,w)} = \left\{ \frac{y}{z} \;\middle|\; y, z \in \mathbb{Z}, 1 \leq z \leq n \wedge -z \cdot W \leq y \leq z \cdot W \right\}.$$

Definition 3.4.2. *Let (Γ, p, w) be a mean-payoff parity game. We denote the set of vertices $v \in V$ such that $val(MPP(\cdot, p, w, \Gamma))(v) \circ \mu$ where $\circ \in \{<, \leq, =, \geq, >\}$ with $V_\Gamma^{\circ\mu}$.*

Key Observation.

Let $((\Gamma = (V, E, \langle V_1, V_2\rangle)), p, w)$ be a mean-payoff parity game. Let $\mu \in [-W, W]$. The sets $V_\Gamma^{>\mu}, V_\Gamma^{=\mu}$ and $V_\Gamma^{<\mu}$ can be computed using any algorithm for threshold mean-payoff parity games twice (for example using Theorem 3.3.5). To calculate $V_\Gamma^{\geq\mu}$ and $V_\Gamma^{<\mu}$ use the algorithm on (Γ, p, w) with the mean-payoff parity objective $\phi = \text{Parity}(p, \Gamma) \cap \text{MeanPayoff}(\mu, w, \Gamma)$. Consider $((\Gamma' = (V, E, \langle V_2, V_1\rangle)), p, w')$, where $w'(e) = -w(e)$ for all edges $e \in E$ and player-1 and player-2 vertices are swapped. To calculate $V_\Gamma^{\leq\mu}$ and $V_\Gamma^{>\mu}$ use the algorithm on (Γ', p, w) with mean-payoff parity objective $\phi = \text{Parity}(p, \Gamma') \cap \text{MeanPayoff}(-\mu, w', \Gamma')$. Given the sets $V_\Gamma^{\leq\mu}$, $V_\Gamma^{>\mu}, V_\Gamma^{\geq\mu}$ and $V_\Gamma^{<\mu}$ we can extract the sets $V_\Gamma^{>\mu}, V_\Gamma^{=\mu}$ and $V_\Gamma^{<\mu}$.

All values μ' in $S^{(\Gamma,p,w)}$ are of the form $\frac{y}{z}$. For those values we can determine whether $v \in V_\Gamma^{\geq\mu'}$ by applying the algorithm for threshold mean-payoff parity games on $((\Gamma' = (V, E, \langle V_2, V_1\rangle)), p, w')$ where $w'(e) = w(e) \cdot z$ for all $e \in E$ with the mean-payoff parity objectives $\phi = \text{Parity}(p, \Gamma) \cap \text{MeanPayoff}(y, w', \Gamma)$. Note that in the worst case, the weight function w' of Γ' is in $O(nW)$.

Dichotomic Search. Let (Γ, p, w) be a mean-payoff parity game. The dichotomic search algorithm is recursive algorithm initialized with $\Gamma_0 = \Gamma$ and $S_0 = S^{(\Gamma,p,w)}$. In recursive call i the following steps are executed:

1. Let $r_i = \min(S_i)$ and $s_i = \max(S_i)$.
2. Determine a_1, the largest element in S_i less than or equal to $\frac{r_i+s_i}{2}$ and a_2, the smallest element in S_i greater than or equal to $\frac{r_i+s_i}{2}$.
3. Determine the partitions $V_{\Gamma_i}^{<a_1}$, $V_{\Gamma_i}^{=a_1}$, $V_{\Gamma_i}^{=a_2}$, $V_{\Gamma_i}^{>a_2}$ using the key observation.
4. For all $v \in V_{\Gamma_i}^{=a_1}$ set the value to a_1, for all $v \in V_{\Gamma_i}^{=a_2}$ set the value to a_2 and set the value to $-\infty$ for all vertices v which are not in any set calculated in step 3.
5. Recurse upon $\Gamma_i \upharpoonright V_{\Gamma_i}^{<a_1}$ and $\Gamma_i \upharpoonright V_{\Gamma_i}^{>a_2}$.

Correctness. Let (Γ, p, w) be a mean-payoff parity game. We prove that the dichotomic search algorithm correctly calculates $val(MPP(\cdot, p, w, \Gamma))(v)$ for all $v \in V$. The algorithm is initialized with Γ and $S^{(\Gamma,p,w)}$. By Lemma 3.4.1 the values of the vertices $v \in V$ are in the set $S^{(\Gamma,p,w)}$. Because we perform a binary search over the set $S^{(\Gamma,p,w)}$ we can guarantee the termination of the algorithm. Notice that we need to show that the values calculated in the subgames constructed in step 4 are identical to the values in the original game. Then correctness follows immediately by our key observation and because we perform a binary search over the set $S^{(\Gamma,p,w)}$.

Lemma 3.4.3. *Given a mean-payoff parity game (Γ, p, w) and $\mu \in \mathbb{Q}$, let $\Gamma' = \Gamma \upharpoonright V_\Gamma^{>\mu}$ and $\Gamma'' = \Gamma \upharpoonright V_\Gamma^{<\mu}$. For all $v \in V_\Gamma^{>\mu}$, we have $val(MPP(\cdot, p, w, \Gamma'))(v) = val(MPP(\cdot, p, w, \Gamma))(v)$ and for all $v \in V_\Gamma^{<\mu}$, we have $val(MPP(\cdot, p, w, \Gamma''))(v) = val(MPP(\cdot, p, w, \Gamma))(v)$.*

Proof. Let $v \in V_\Gamma^{>\mu}$ be arbitrary. We will prove $val(MPP(\cdot, p, w, \Gamma'))(v) = val(MPP(\cdot, p, w, \Gamma))(v)$ by showing the following two cases:

- $val(MPP(\cdot, p, w, \Gamma'))(v) \leq val(MPP(\cdot, p, w, \Gamma))(v)$: Note that there can be no player-2 vertex in $V_\Gamma^{>\mu}$ with an edge to $V_\Gamma^{\leq\mu}$. Thus we cut away only edges of player-1 vertices in Γ'. Consequently player-1 has less choices in Γ' than in Γ at each of her vertices. Thus $val(MPP(\cdot, p, w, \Gamma'))(v) \leq val(MPP(\cdot, p, w, \Gamma))(v)$ holds.
- $val(MPP(\cdot, p, w, \Gamma'))(v) \geq val(MPP(\cdot, p, w, \Gamma))(v)$: Let σ be an optimal strategy for player 1 and let π be an optimal strategy for player 2 which both exist by [83]. We will show that σ produces plays with vertices in $V_\Gamma^{>\mu}$ only, if it starts from v. For the sake of contradiction assume that a play $\rho = \omega(v, \sigma, \pi)$ contains a vertex $v^* \in V_\Gamma^{\leq\mu}$. Notice that there are no player-2 vertices in $V_\Gamma^{>\mu}$ with edges to $V_\Gamma^{\leq\mu}$. Thus σ chose a successor vertex in $V_\Gamma^{\leq\mu}$. But when ρ ends up in $V_\Gamma^{\leq\mu}$ the optimal player-2 strategy π can guarantee that $MPP(\rho, p, w, \Gamma) \leq \mu$ by the definition of $V_\Gamma^{\leq\mu}$. There is a strategy

to keep the value of the play starting at v greater than μ by the definition of $V_\Gamma^{>\mu}$. Thus any play ρ leading to $V_\Gamma^{\leq\mu}$ using σ is not optimal which is a contradiction to our assumption. Consequently $val(MPP(\cdot, p, w, \Gamma'))(v) \geq val(MPP(\cdot, p, w, \Gamma))(v)$ follows.

The fact that for all $v \in V_\Gamma^{<\mu}$, we have $val(MPP(\cdot, p, w, \Gamma''))(v) = val(MPP(\cdot, p, w, \Gamma))(v)$ follows by a symmetric argument.

□

Running Time. The running time of the dichotomic search is $O(n \log(nW)\mathsf{TH})$ where TH is the running time of an algorithm for the threshold mean-payoff parity problem. The additional factor n comes from rescaling the weights of the mean-payoff parity game Γ which is described in the key observation. The factor $O(\log(nW))$ is from using binary search on S as $|S| = O(n^2W)$.

Theorem 3.4.4. *Given a mean-payoff parity game (Γ, p, w) and an algorithm that solves the threshold mean-payoff parity problem in $O(\mathsf{TH})$, the value function of (Γ, p, w) can be computed in time $O(n \log(nW)\mathsf{TH})$.*

As a corollary of the above theorem and Theorem 3.3.5, the value function for mean-payoff parity games can be computed in $O(n^d \cdot m \cdot W \cdot \log(nW))$ time.

3.5 Conclusion

In this chapter, we present faster algorithms for mean-payoff parity games. Our most interesting results are for mean-payoff Büchi and mean-payoff coBüchi games, which are the base cases. For threshold mean-payoff Büchi and mean-payoff coBüchi games, our bound $O(nmW)$ matches the current best-known bound for mean-payoff games. For the value problem, we show the dichotomic search approach of [44] for mean-payoff games can be generalized to mean-payoff parity games. This gives an additional multiplicative factor of $n \log(nW)$ as compared to the threshold problem. A recent work by Comin et al. [111] shows that the value problem for mean-payoff objective can be solved with a multiplicative factor n compared to the threshold objective (i.e., it shaves of the log factor). An interesting question is whether the approach of Comin et al. can be generalized to mean-payoff parity games.

CHAPTER 4

Near-Linear Time Algorithms for Streett Objectives in Graphs and MDPs

In this chapter, we present randomized near-linear time algorithms for Streett objectives in graphs and MDPs.

4.1 Introduction

In this work, we present near-linear (hence near-optimal) randomized algorithms for the strong fairness verification in graphs and Markov Decision Processes (MDPs). In the fundamental model-checking problem, the input is a *model* and a *specification*, and the algorithmic verification problem is to check whether the model *satisfies* the specification. We first describe the models and the specifications we consider, then the notion of satisfaction, and then previous results followed by our contributions.

Models: Graphs and MDPs. Graphs and Markov decision processes (MDPs) are two classical models of reactive systems. The states of a reactive system are represented by the vertices of a graph, the transitions of the system are represented by the edges and non-terminating trajectories of the system are represented as infinite paths of the graph. Graphs are a classical model for reactive systems with nondeterminism, and MDPs extend graphs with probabilistic transitions that represent reactive systems with both nondeterminism and uncertainty. Thus, graphs and MDPs are the standard models of reactive systems with nondeterminism, and nondeterminism with stochastic aspects, respectively [22, 105]. Moreover, MDPs are used as models for concurrent finite-state processes [113, 221] as well as probabilistic systems in open environments [22, 117, 168, 209].

Specification: Strong fairness (aka Streett) objectives. A fundamental specification formalism in the analysis of reactive systems is the *strong fairness condition.* The strong fairness conditions (aka Streett objectives) consist of k types of requests and corresponding grants, and the requirement is that for each type if the request happens infinitely often, then the corresponding grant must also happen infinitely often. Beyond safety, reachability, and liveness objectives, the most standard properties that arise in the analysis of reactive systems are Streett objectives, and chapters of standard textbooks in verification are devoted to it (e.g., [105, Chapter 3.3], [180, Chapter 3], [16, Chapters 8, 10]). Besides, ω-regular objectives can be specified as Streett objectives, e.g., LTL formulas and non-deterministic ω-automata can be translated to deterministic Streett automata [203] and efficient translations have been an active research area [76, 126, 160]. Consequently, Streett objectives are a canonical class of objectives that arise in verification.

Satisfaction. The notions of satisfaction for graphs and MDPs are as follows: For graphs, the notion of satisfaction requires that there is a trajectory (infinite path) that belongs to the set of paths specified by the Streett objective. For MDPs, the satisfaction requires that there is a strategy to resolve the nondeterminism such that the Streett objective is ensured almost-surely (with probability 1). Thus the algorithmic model-checking problem of graphs and MDPs with Streett objectives is a central problem in verification, and is at the heart of many state-of-the-art tools such as SPIN, NuSMV for graphs [104, 145] and PRISM, LiQuor, Storm for MDPs [103, 117, 168].

Our contributions are related to the algorithmic complexity of graphs and MDPs with Streett objectives. We first present previous results and then our contributions.

Previous results. The most basic algorithm for the problem for graphs is based on repeated SCC (strongly connected component) computation, and informally can be described as follows: for a given SCC, (a) if for every request type that is present in the SCC the corresponding grant type is also present in the SCC, then the SCC is identified as "good", (b) else vertices of each request type that have no corresponding grant type in the SCC are removed, and the algorithm recursively proceeds on the remaining graph. Finally, reachability to good SCCs is computed. The algorithm for MDPs is similar where the SCC computation is replaced with maximal end-component (MEC) computation and reachability to good SCCs is replaced with probability 1 reachability to good MECs. The basic algorithms for graphs and MDPs with Streett objective have been improved in several works, such as for graphs in [80, 138], for MEC computation in [77, 78, 91], and MDPs with Streett objectives in [75]. For graphs/MDPs with n vertices, m edges, and k request-grant pairs with b denoting the size to describe the request grant pairs, the current best-known bound is $O(\min(n^2, m\sqrt{m \log n}) + b \log n)$.

Our contributions. In this work, our main contributions are randomized near-linear time (i.e. linear times a polylogarithmic factor) algorithms for graphs

and MDPs with Streett objectives. In detail, our contributions are as follows:

- First, we present a near-linear time randomized algorithm for graphs with Streett objectives where the expected running time is $\widetilde{O}(m+b)$, where the $\widetilde{O}$ notation hides poly-log factors. Our algorithm is based on a recent randomized algorithm for maintaining the SCC decomposition of graphs under edge deletions, where the expected total running time is near linear [31].
- Second, by exploiting the results of [31] we present a randomized near-linear time algorithm for computing the MEC decomposition of an MDP where the expected running time is $\widetilde{O}(m)$. We extend the results of [31] from graphs to MDPs and present a randomized algorithm to maintain the MEC decomposition of an MDP under edge deletions, where the expected total running time is near linear [31].
- Finally, we use the result of the above item to present a near-linear time randomized algorithm for MDPs with Streett objectives where the expected running time is $\widetilde{O}(m+b)$.

All our algorithms are randomized and since they are near-linear in the size of the input, they are optimal up to poly-log factors. An important open question is whether there are deterministic algorithms that can improve the existing running time bound for graphs and MDPs with Streett objectives. Our algorithms are deterministic except for the invocation of the decremental SCC algorithm presented in [31].

Table 4.1: Summary of Results.

Problem	New Running T.	Old Running T.
Streett Objectives on Graphs	$\widetilde{O}(m+b)$	$\widetilde{O}(\min(n^2, m\sqrt{m})+b)$ [81, 138]
Almost-Sure Reachability	$\widetilde{O}(m)$	$O(m \cdot n^{2/3})$ [75, 91]
MEC Decomposition	$\widetilde{O}(m)$	$O(m \cdot n^{2/3})$ [91]
Decremental MEC Decomposition	$\widetilde{O}(m)$	$O(nm)$ [91]
Streett Objectives on MDPs	$\widetilde{O}(m+b)$	$\widetilde{O}(\min(n^2, m\sqrt{m})+b)$ [75]

4.2 Decremental SCCs

We first recall the result about decremental strongly connected components maintenance in [31] (cf. Theorem 4.2.1 below) and then augment the result for our purposes.

Theorem 4.2.1 (Theorem 1.1 in [31]). *Given a graph $G = (V, E)$ with m edges and n vertices, we can maintain a data structure $\mathscr{A}$ that supports the operations*

- DELETE(u, v)*: Deletes the edge (u, v) from the graph G.*

- QUERY(u, v): *Returns whether u and v are in the same SCC in G,*

in total expected update time $O(m \log^4 n)$ and with worst-case constant query time. The bound holds against an oblivious adversary.

The preprocessing time of the algorithm is $O(m + n)$ using [216]. To use this algorithm we extend the query and update operations with three new operations described in Corollary 4.2.2.

Intuitively, the first function is available in the algorithm described in [31]. The second function can be implemented directly from the construction of the data structure maintained in [31]. The key idea for the third function is that when an SCC splits, we consider the new SCCs. We distinguish between the largest of them and the others which we call small SCCs. We then consider all edges incident to the small SCCs: Note that as the new outgoing edges in the large SCC are also incident to a small SCC we can also determine the outgoing edges of the large SCC. Observe that whenever an SCC splits all the small SCCs are at most half the size of the original SCC. That is, each vertex can appear only $O(\log n)$ times in small SCCs during the whole algorithm. As an edge is only considered if one of the incident vertices is in a small SCC each edge is considered $O(\log n)$ times and the additional running time is bounded by $O(m \log n)$. Furthermore, we define T_d as the running time of the best decremental SCC algorithm which supports the operations in Corollary 4.2.2. Currently, $T_d = O(m \log^4 n)$.

Corollary 4.2.2. *Given a graph $G = (V, E)$ with m edges and n vertices, we can maintain a data structure $\mathscr{A}$ that supports the operations*

- REP(u) *(query-operation): Returns a reference to the SCC containing the vertex u.*

- DELETE-ANNOUNCE(E) *(update-operation): Deletes the set E of edges from the graph G. If the edge deletion creates new SCCs $C_1, \ldots, C_k$ the operation returns a list $Q = \{C_1, \ldots, C_k\}$ of references to the new SCCs.*

- DELETE-ANNOUNCE-NO-OUTGOING(E) *(update-operation): Deletes the set E of edges from the graph G. The operation returns a list $Q = \{C_1, \ldots, C_k\}$ of references to all new SCCs with no outgoing edges.*

in total expected update time $O(m \log^4 n)$ and worst-case constant query time for the first operation. The bound holds against an oblivious adaptive adversary.

Proof. We first recall the data structure $\mathscr{A}$ maintained by the algorithm of [31]. We need the following detail about $\mathscr{A}$ to prove the second and third point. The data structure $\mathscr{A}$ maintains a hierarchy of graphs $\hat{G} = \{\hat{G}_0, \ldots, \hat{G}_{\lfloor \log n \rfloor + 1}\}$:

$$\hat{G}_i = \mathsf{CONDENSE}((V, \bigcup_{j<i} E_j)) \cup E_i$$

where the edge sets E_i form a partition of E. Additionally, the top graph $\hat{G}_{\lfloor \log n \rfloor + 1}$ contains all the edges of G and the SCCs in $\hat{G}_{\lfloor \log n \rfloor + 1}$ are thus the same as in G [31, P. 5]. The top level graph thus corresponds to CONDENSE(G).

- The data structure $\mathscr{A}$ maintained in [31] can be extended to support the function REP(v) for some vertex u because $\mathscr{A}$ can in constant time identify the node representing v in the graph $\hat{G}_{\lfloor \log n \rfloor + 1}$ [31, P.23].

- The data structure $\mathscr{A}$ maintained in [31] can be extended to support the modified delete-operation DELETE-ANNOUNCE(E): At the beginning of the operation we initialize a new List $\mathscr{L}$. For each edge $e \in E$ we do the following: We compute $\mathscr{A}$.DELETE(e) and when nodes in CONDENSE(G) are split due to an edge deletion, $\mathscr{A}$ creates new nodes $\{s_1, \ldots, s_k\}$ in CONDENSE(G) by the fact that $\mathscr{A}$ maintains the hierarchy $\hat{G}$. For each $s_i \in \{s_1, \ldots, s_k\}$, store the pointer to the nodes in $\mathscr{L}$. In the end we return $\mathscr{L}$. This can be done with an additional constant effort when maintaining the graph CONDENSE(G). If an SCC C splits which is already in $\mathscr{L}$ due to an edge deletion in C, we create a new node for each new SCCs (and add it to $\mathscr{L}$).

- We prove that the data structure $\mathscr{A}$ maintained in [31] can be extended to support the modified delete operation DELETE-ANNOUNCE-NO-OUTGOING(E) in total expected update time $O(m \log^4 n)$ time: Note that the top level maintained in the hierarchy corresponds to CONDENSE(G). Initially, we store the number of outgoing edges in a counter for all nodes in CONDENSE(G). Clearly, this is in $O(m + n)$ time, the preprocessing time of $\mathscr{A}$. These counters will be maintained throughout the entire sequence of edge-deletions, i.e., for each edge deleted from G. Whenever an edge $e = (u, v) \in E$ is deleted we need to distinguish between the case (i) e is in the graph CONDENSE(G) and (ii) e is in one of the SCCs. In case (i) the counter of the node representing u must be decremented. This is in constant time. For case (ii), when e is inside an SCC we also distinguish between two cases. Either the SCC decomposition does not change (CONDENSE(G) does not change) or the SCC decomposition changes. In the first case, we leave the counters unchanged and are done because CONDENSE(G) does not change. When the SCC decomposition changes, new nodes are created in CONDENSE(G) because by deleting edges we create more SCCs. Thus there is an SCC C_o which is split into new SCCs $C = \{C_1, \ldots, C_k\}$ when we delete the edge e. We initialize the counters for all new SCCs to zero (nodes created in CONDENSE(G)) except for the largest one (i.e., $C_\ell = \arg\max_{C_i \in C} |C_i|$) which we set to the value of the original SCC C_o. We determine the number of outgoing edges of the SCCs in $C \setminus \{C_\ell\}$ by looking at the incident edges. During this process we modify the counter of C_ℓ, which we initialized to the counter of the original SCC C_o in the following two cases:

- If an edge in $C \setminus \{C_\ell\}$ goes to an SCC not in C we decrement the counter of C_ℓ.
- If there is an edge going to an SCC in C we increment the counter of C_ℓ.

Assume the counters were correct before the edge deletion, that is, in CONDENSE(G). Let G' be the graph without e. We prove that we count each outgoing edge in CONDENSE(G') correctly. Let $e' = (u, v)$ be an arbitrary edge in CONDENSE(G'). Either e' is present before the deletion of e or e' is new after the deletion of e.

- Edges present before the deletion of e are of two types, going out of C_o and not going out of C_o: When the edge e' is not going out of C_o the counter of the outgoing node is still correct because the number of outgoing edges did not change. When the edge e' is going out of C_o there are two cases. Either it is now going out of C_ℓ or of some other SCC in $C \setminus \{C_\ell\}$. If $u = C_\ell$, notice that C_ℓ is set to the counter of C_o and we count e'. On the other hand, if u is in $C \setminus \{C_\ell\}$ and v is a node outside of C we count e' when we look at all edges incident to u. Note that we also count it at C_ℓ because this edge is originally incident to C_o and but not to C_ℓ. Thus we decrement the counter of C_ℓ for each such edge.
- Edges new after the deletion of e are also of two types: Going out of C_ℓ and going out of an SCC in $C \setminus \{C_\ell\}$. Note that all edges have their endpoints in C. If u is in $C \setminus \{C_\ell\}$ we count e' correctly because we looked at all edges incident to u. If $u = C_\ell$ we need to increment the counter of u. This is done when we look at all the incident edges of $C \setminus \{C_\ell\}$.

For each edge, we have a constant amount of work: An edge can be in the smaller part of the partition at most $O(\log n)$ times because it is only considered when the corresponding original SCC halves. When we execute the operation DELETE-ANNOUNCE-NO-OUTGOING(E) while deleting edges and we find that an SCC has no outgoing edges, i.e., the stored counter is zero, we add it to a list and return this list. □

4.3 Graphs with Streett Objectives

In this section, we present an algorithm which computes the winning regions for graphs with Streett objectives. The input is a directed graph $G = (V, E)$ and k Streett pairs (L_j, U_j) for $j = 1, \ldots, k$. The size of the input is measured in terms of $m = |E|$, $n = |V|$, k and $b = \sum_{j=1}^{k} (|L_j| + |U_j|) \leq 2nk$.

Algorithm Street and good component detection. Let C be an SCC of G. In the good component detection problem, we compute (a) a non-trivial SCS $G[X] \subseteq C$ induced by the set of vertices X, such that for all $1 \leq j \leq k$ either $L_j \cap X = \emptyset$ or $U_j \cap X \neq \emptyset$ or (b) that no such SCS exists. In the first case, there exists an infinite path that eventually stays in X and satisfies the Streett objective, while in the latter case, there exists no path which satisfies the Streett objective in C. From the results of [16, Chapter 9, Proposition 9.4] the following algorithm, called Algorithm Street, suffices for the winning set computation:

1. Compute the SCC decomposition of the graph;
2. For each SCC C for which the good component detection returns an SCS, label the SCC C as satisfying.
3. Output the set of vertices that can reach a satisfying SCC as the winning set.

Since the first and last step are computable in linear time, the running time of Algorithm Street is dominated by the detection of good components in SCCs. In the following, we assume that the input graph is strongly connected and focus on good component detection.

Bad vertices. A vertex is *bad* if there is some $1 \leq j \leq k$ such that the vertex is in L_j but it is not strongly connected to any vertex of U_j. All other vertices are good. Note that a good vertex might become bad if a vertex deletion disconnects an SCS or a vertex of a set U_j. A good component is a non-trivial SCS that contains only good vertices.

Decremental strongly connected components. Throughout the algorithm, we use the algorithm described in Section 4.2 to maintain the SCCs of a graph when deleting edges. In particular, we use Corollary 4.2.2 to obtain a list of the new SCCs which are created by removing bad vertices. Note that we can 'remove' a vertex by deleting all its incident edges. Because the decremental SCC algorithm assumes an oblivious adversary we sort the list of the new SCCs as, otherwise, the edge deletions performed by our algorithm would depend on the random choices of the decremental SCC algorithm.

Data structure. During the algorithm, we maintain a decomposition of the vertices in $G = (V, E)$: We maintain a list Q of certain sets $S \subseteq V$ such that every SCC of G is contained in some S stored in Q.

The list Q provides two operations: Q.ADD(X) enqueues X to Q; and Q.PULL() dequeues an arbitrary element X from Q. For each set S in the decomposition, we store a data structure $D(S)$ in the list Q. This data structure $D(S)$ supports the following operations

1. CONSTRUCT(S): initializes the data structure for the set S
2. REMOVE(S, B) updates S to $S \setminus B$ for a set $B \subseteq V$ and returns $D(S)$ for the new set S.

3. BAD(S) returns a reference to the set $\{v \in S \mid \exists j$ with $v \in L_j$ and $U_j \cap S = \emptyset\}$

4. D-SCCS(S) returns the set of SCCs currently in $G[S]$. We implement D-SCCS(S) as a balanced binary search tree which allows logarithmic and updates and deletions.

In [138] an implementation of this data structure with functions (1)-(3) is described that achieves the following running times. For a set of vertices $S \subseteq V$, let $bits(S)$ be defined as $\sum_{j=1}^{k}(|S \cap L_j| + |S \cap U_j|)$.

Lemma 4.3.1 (Lemma 2.1 in [138]). *After a one-time preprocessing of time $O(k)$, the data structure $D(S)$ can be implemented in time $O(bits(S)+|S|)$ for* CONSTRUCT(S), *time $O(bits(B) + |B|)$ for* REMOVE(S, B) *and constant running time for* BAD(S).

We augment the data structure with the function D-SCCS(S) which runs in total time of a decremental SCC algorithm supporting the first function in Corollary 4.2.2.

Algorithm Description. The key idea is that the algorithm maintains the list Q of data structures $D(S)$ as described above when deleting bad vertices. Initially, we enqueue the data structure returned by CONSTRUCT(V) to Q. As long as Q is non-empty, the algorithm repeatedly pulls a set S from Q and identifies and deletes bad vertices from $G[S]$. If no edge is contained in $G[S]$, the set S is removed as it can only induce trivial SCCs. Otherwise, the subgraph $G[S]$ is either determined to be strongly connected and output as a good component or we identify and remove an SCC with at most half of the vertices in $G[S]$. Consider Figure 4.1 for an illustration of an example run of Algorithm 4.1.

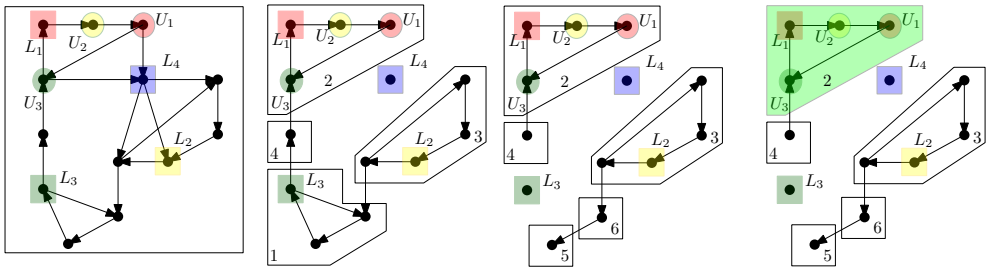


Figure 4.1: Illustration of one run of Algorithm 4.1: The vertex in the set L_4 is a bad vertex and we remove it from the SCC yielding four new SCCs. First, we look in the SCC containing L_3. The vertex in L_3 is a bad vertex because there is no vertex in U_3 in this SCC. Again two SCCs are created after its removal. The next SCC we process is the SCC containing L_1. It is a good component because the vertex in L_1 has a vertex in U_1 in the same SCC. No bad vertices are removed and the whole SCC is identified as a good component.

Outline correctness and running time. In the following, when we talk about the *input graph* $\hat{G}$ we mean the unmodified, strongly connected graph which we use to

Algorithm 4.1: Algorithm GoodComp

Input: Strongly connected graph $G = (V, E)$ and Streett pairs (L_j, U_j) for $j = 1, \ldots, k$
Output: a good component in G if one exists

Invoke an instance $\mathscr{A}$ of the decremental SCC algorithm; Initialize Q as a new list.
$D(V) \leftarrow$ construct(V); $D(V)$.d-sccs$(V) \leftarrow \{\mathscr{A}.\text{rep}(x)\}$ for some $x \in V$
Q.add$(D(V))$
while Q *is not empty* **do**
  $D(S) \leftarrow Q$.pull()
  while $D(S)$.bad(S) *is not empty* **do**
    $B \leftarrow D(S)$.bad(S); $D(S) \leftarrow D(S)$.remove(S, B)
    `// obtain SCCs after deleting bad vertices from` S
    $D(S)$.d-sccs$(S) \leftarrow D(S)$.d-sccs$(S) \setminus \left(\bigcup_{b \in B}\{\mathscr{A}.\text{rep}(b)\}\right)$
    $D(S)$.d-sccs$(S) \leftarrow D(S)$.d-sccs$(S) \cup \mathscr{A}$.delete-announce$(E(B))$
  if $G[S]$ *contains at least one edge* **then**
    Initialize K as a new list
    for $X \leftarrow D(S)$.d-sccs(S) **do**
      if $X = S$ **then output** $G[S]$; `// good component found`
      if $|X| \leq \frac{|S|}{2}$ **then** K.add(X);
    Sort the SCCs in K by vertex id (look at all the vertices in each SCC of K)
    $R \leftarrow \emptyset$ `//` Build $D(X)$ for SCCs X in K and remove X from $S, D(S)$ and d-sccs(S)
    for $X \leftarrow K$.pull() **do**
      $R \leftarrow R \cup X$; $D(X) \leftarrow$ construct(X)
      $D(X)$.d-sccs$(X) \leftarrow \{\mathscr{A}.\text{rep}(x)\}$ for some $x \in X$
      $D(S)$.d-sccs$(S) \leftarrow D(S)$.d-sccs$(S) \setminus \{\mathscr{A}.\text{rep}(x)\}$ for some $x \in X$
      Q.add$(D(X))$
    if $D(S)$.d-sccs$(S) \neq \emptyset$ **then** Q.add$(D(S)$.remove$(S, R))$
return No good component exists.

initialize Algorithm 4.1. In contrast, with the *current* graph G we refer to the graph where we already deleted vertices and their incident edges in the course of finding a good component. For the correctness of Algorithm 4.1, we show that if a good component exists, then there is a set S stored in list Q which contains all vertices of this good component.

To obtain the running time bound of Algorithm 4.1, we use the fact that we can maintain the SCC decomposition under deletions in $O(T_d)$ total time. With the properties of the data structure described in Lemma 4.3.1 we get a running time of $\widetilde{O}(n+b)$ for the maintenance of the data structure and identification of bad vertices over the whole algorithm. Combined, these ideas lead to a total running time of $\widetilde{O}(T_d + n + b)$ which is $\widetilde{O}(m + b)$ using Corollary 4.2.2.

Lemma 4.3.2. *Algorithm 4.1 runs in expected time* $\widetilde{O}(m + b)$.

Proof. The preprocessing and initialization of the data structure D and the removal

of bad vertices in the whole algorithm takes time $O(m + k + b)$ using Lemma 4.3.1. Since each vertex is deleted at most once, the data structure can be constructed and maintained in total time $O(m)$. Announcing the new SCCs after deleting the bad vertices at Line 9 is in $O(T_d) = \widetilde{O}(m)$ total time by Corollary 4.2.2. Consider an iteration of the while loop at Line 4: A set S is removed from Q. Let us denote by n' the number of vertices of S. If $G[S]$ does not contain any edge after the removal of bad vertices, then S is not considered further by the algorithm. Otherwise, the for-loop at Line 12 considers all new SCCs. We can implement the for-loop in a lockstep fashion: In each step for each SCC we access the i-th vertex and as soon as all of the vertices of an SCC are accessed we add it to the list K. When only one SCC is left we compute its size using the original set S and the sizes of the other SCCs. If its size is at most $|S|/2$ we add it to K. Note that this can be done in time proportional to the number of vertices in the SCCs in S of size at most $|S|/2$. The sorting operation at Line 15 takes time $O(|K| \log |K|)$ plus the size of all the SCCs in K, that is $\sum_{K_i \in K} |K_i|$. Note that $O(|K| \log |K|) = O((\sum_{K_i \in K} |K_i|) \log(\sum_{K_i \in K} |K_i|))$. Let $K_i \in K$ be an SCC stored in K. Note that during the algorithm each vertex can appear at most $\log(n)$ times in the list K. This is by the fact that K only contains SCCs that are at most half the size of the original set S. We obtain a running time bound of $O(n(\log n)^2)$ for Lines 12-15.
Consider the second for-loop at Line 17: Let $|X| = n_1$. The operations REMOVE$(\cdot)$ and CONSTRUCT$(\cdot)$ are called once per found SCC $G[X]$ with $X \neq S$ and take by Lemma 4.3.1 $O(|X| + bits(X))$ time. Whenever a vertex is in X, the size of the set in Q containing v originally is reduced by at least a factor of two due to the fact that $|X| = n_1 \leq n'/2$. This happens at most $\lceil \log n \rceil$ times. By charging $O(1)$ to the vertices in X and, respectively, to $bits(X)$, the total running time for Lines 21 & 22 can be bounded by $O((n + b) \log n)$ as each vertex and bit is only charged $O(\log n)$ times. Combining all parts yields the claimed running time bound of $O(T_d + b \log n + n \log^2 n) = \widetilde{O}(m + b)$. □

The correctness of the algorithm is similar to the analysis given in [81, Lemmas 3.6 & 3.7] except that we additionally have to prove that D-SCCS(S) holds the SCCs of $G[S]$. Lemma 4.3.3 shows that we maintain D-SCCS(S) properly for all the data structures in Q.

Lemma 4.3.3. *After each iteration of the outer while-loop every non-trivial SCC of the current graph is contained in one of the subgraphs $G[S]$ for which the data structure $D(S)$ is maintained in Q and* D-SCCS(S) *stores a list of all SCCs contained in S.*

Proof. Initially, D-SCCS(V) stores the whole input graph as one SCC. Thus, by the assumption that the input is a strongly connected graph the claim is true before the first iteration of the while loop. Thus let us assume the statement is true before an iteration of the loop. We will show that then it also holds after the iteration. First, the SCCs of a set S stored in the data structure are only modified when we remove bad vertices at Lines 6-8. Consider such bad vertex $b \in B$: If new SCCs are created due to the deletion of b, the representative SCC of b splits into multiple

SCCs and D-SCCS(S) is correctly updated in Lines 8 and 9. Moreover, for each new non-trivial SCC X that we remove from S, the algorithm adds a new set to Q in Lines 21 & 22. Also, we update the corresponding set D-SCCS(X) at Line 19 and the old set D-SCCS(S) at Line 20. □

We prove the next Lemma by showing that we never remove edges of vertices of good components.

Lemma 4.3.4. *After each iteration of the outer while-loop every good component of the input graph is contained in one of the subgraphs $G[S]$ for which the data structure $D(S)$ is maintained in the list Q.*

Proof. We first show that Algorithm 4.1 never removes edges or vertices that belong to a good component. Consider an arbitrary iteration of the outer while loop at Line 4. Let $D(S)$ be the data structure pulled from the list Q in that iteration. Edges are only removed in Line 9 of the algorithm where bad vertices and incident edges are removed from S. As bad vertices cannot be part of any good component in S the algorithm does not delete any edge in a good component. That is, every good component of the initial graph stays strongly connected in the modified graph during the whole algorithm and the claim follows from Lemma 4.3.3. □

Proposition 4.3.5. *Algorithm 4.1 outputs a good component if one exists, otherwise the algorithm reports that no such component exists.*

Proof. First consider the case where Algorithm 4.1 outputs a subgraph $G[S]$. We show that $G[S]$ is a good component: Line 10 ensures only non-trivial SCSs are considered. After the removal of bad vertices from S in Lines 6-9, we know that for all $1 \leq j \leq k$ that $U_j \cap S \neq \emptyset$ if $S \cap L_j \neq \emptyset$. Due to Line 13 there is only one SCC in $G[S]$ and thus $G[S]$ is a good component. Second, if Algorithm 4.1 terminates without a good component, by Lemma 4.3.4, we have that the initial graph has no good component and thus the result is correct as well. □

The running time bounds for the decremental SCC algorithm of [31] (cf. Corollary 4.2.2) only hold against an oblivious adversary. Thus we have to show that in our algorithm the sequence of edge deletions does not depend on the random choices of the decremental SCC algorithm. The key observation is that only the order of the computed SCCs depends on the random choices of the decremental SCC and we eliminate this effect by sorting the SCCs.

Proposition 4.3.6. *The sequence of deleted edges does not depend on the random choices of the decremental SCC Algorithm but only on the given instance.*

Proof. Note that we only delete edges at Line 9. We prove that the claim holds for every iteration of the while loop: Initially, there is only one element in Q at Line 4. Thus the claim holds initially. Assume that the claim holds before the while-loop. Thus, the S we pull from Q does not depend on the random choices of the decremental SCC algorithm $\mathscr{A}$. Note, that when we remove vertices at Line 9 the newly

created SCCs are returned to K are determined by the input instance and do not depend on the random choices of the algorithm. Only *the order* in which the SCCs are returned depends on the random choices of the decremental SCC algorithm $\mathscr{A}$. When we look at the set S of vertices which contains all of the new SCCs at Line 12-21 we add all SCCs with at most $|S|/2$ vertices to a list. We sort this list by the minimum vertex id of each SCC and add them according to this order to our list data structure Q (if there is an SCC with more than $|S|/2$ vertices it is at the end of Q). Thus, again the order in which SCCs are dequeued from Q is fixed and does not depend on the random choices of $\mathscr{A}$. □

Due to Lemma 4.3.2, Lemma 4.3.5 and Proposition 4.3.6 we obtain the following result.

Theorem 4.3.7. *In a graph, the winning set for a k-pair Streett objective can be computed in $\widetilde{O}(m+b)$ expected time.*

4.4 Algorithms for MDPs

In this section, we present expected near-linear time algorithms for computing a MEC decomposition, deciding almost-sure reachability and maintaining a MEC decomposition in a decremental setting. In the last section, we present an algorithm for *MDPs* with Streett objectives by using the new algorithm for the decremental MEC decomposition.

4.4.1 Maximal End-Component Decomposition

In this section, we present an expected near linear time algorithm for MEC decomposition. Our algorithm is an efficient implementation of the static algorithm presented in [91, p. 29]: The difference is that the bottom SCCs are computed with a dynamic SCC algorithm instead of recomputing the static SCC algorithm. A similar algorithm was independently proposed in an unpublished extended version of [98].

Algorithm Description. The MEC algorithm described in Algorithm 4.2 repeatedly removes bottom SCCs and the corresponding random attractor. After removing bottom SCCs the new SCC decomposition with its bottom SCCs is computed using a dynamic SCC algorithm.

Correctness follows because our algorithm just removes attractors of bottom SCCs and marks bottom SCCs as MECs. This is precisely the second static algorithm presented in [91, p. 29] except that the bottom SCCs are computed using a dynamic data structure. By using the decremental SCC algorithm described in Subsection 4.2 we obtain the following lemma.

Lemma 4.4.1. *Algorithm 4.2 returns the MEC-decomposition of an MDP $\boldsymbol{P}$ in expected time $\widetilde{O}(m)$.*

Algorithm 4.2: MEC Algorithm

Input: MDP $P = (V, E, \langle V_1, V_R \rangle, \delta)$, decremental SCC algorithm $\mathscr{A}$

1 Invoke an instance $\mathscr{A}$ of the decremental SCC algorithm
2 Compute the SCC-decomposition of $G = (V, E)$: $C = \{C_1, \ldots, C_\ell\}$
3 Let $M = \emptyset$; $Q \leftarrow \{C_i \in C \mid C_i \text{ has no outgoing edges}\}$
4 **while** Q *is not empty* **do**
5 $\quad C \leftarrow \emptyset$
6 $\quad$ **for** $C_k \in Q$ **do** $C \leftarrow C \cup C_k$; $M \leftarrow M \cup \{C_k\}$
7 $\quad A \leftarrow attr_R(C, P)$
8 $\quad Q \leftarrow \mathscr{A}$.DELETE-ANNOUNCE-NO-OUTGOING($E(A)$)// `remove A from P`
9
10 **return** M

Proof. The running time of algorithm $\mathscr{A}$ is in total time $O(T_d) = \widetilde{O}(m)$ by Theorem 4.2.1 and Corollary 4.2.2. Initially, computing the SCC decomposition and determining the SCCs with no outgoing edges takes time $O(m + n)$ by using [216]. Each time we compute the attractor of a bottom SCC C_k at Line 7 we remove it from the graph by deleting all its edges and never process these edges and vertices again. Since we can compute the attractor A at Line 7 in time $O(\sum_{v \in A} In(A))$, we need $O(m + n)$ total time for computing the attractors of all bottom SCCs. Hence, the running time is dominated by the decremental SCC algorithm $\mathscr{A}$, which is $O(T_d) = \widetilde{O}(m)$. □

The algorithm uses $O(m + n)$ space because the decremental SCC algorithm $\mathscr{A}$ uses $O(m + n)$ space and Q only contains vertices.

Theorem 4.4.2. *Given an MDP the MEC-decomposition can be computed in $\widetilde{O}(m)$ expected time. The algorithm uses $O(m + n)$ space.*

Note that we can use the decremental SCC Algorithm $\mathscr{A}$ of [31] even though this algorithm only works against an oblivious adversary as the sequence of deleted edges does not depend on the random choices of the decremental SCC Algorithm.

4.4.2 Almost-Sure Reachability

In this section, we present an expected near linear-time algorithm for the almost-sure reachability problem. In the almost-sure reachability problem, we are given an MDP P and a target set T and we ask for which vertices player 1 has a strategy to reach T almost surely, i.e., $\langle\langle 1 \rangle\rangle_{a.s.}(\text{Reach}(T))$. Due to [75, Theorem 4.1] we can determine the set $\langle\langle 1 \rangle\rangle_{a.s.}(\text{Reach}(T))$ in time $O(m + \mathsf{MEC})$ where MEC is the running time of the fastest MEC algorithm. We use Theorem 4.4.2 to compute the MEC decomposition and obtain the following theorem.

Theorem 4.4.3. *We can compute $\langle\langle 1 \rangle\rangle_{a.s.}(Reach(T))$ in $\widetilde{O}(m)$ expected time.*

4.4.3 Decremental Maximal End-Component Decomposition

We present an expected near-linear time algorithm for the MEC-decomposition which supports player-1 edge deletions and a query that answers if two vertices are in the same MEC. We need the following lemma from [78] to prove the correctness of our algorithm. Given an SCC C we consider the set U of the random vertices in C with edges leaving C. The lemma states that for all non-trivial MECs X in P the intersection with U is empty, i.e., $attr_R(U, P) \cap X = \emptyset$.

Lemma 4.4.4 (Lemma 2.1(1), [78]). *Let C be an SCC in P. Let $U = \{v \in C \cap V_R \mid E(v) \cap (V \setminus C) \neq \emptyset\}$ be the random vertices in C with edges leaving C. Let $Z = attr_R(U, P) \cap C$. Then, for all non-trivial MECs X in P we have $Z \cap X = \emptyset$ and for any edge (u, v) with $u \in X$ and $v \in Z$, u must belong to V_1.*

The *pure MDP graph* P^P of an MDP $P = (V, E, \langle V_1, V_R \rangle, \delta)$ is the graph which contains only edges in non-trivial MECs of P. More formally, the pure MDP graph P^P is defined as follows: Let $M_1, \dots M_k$ be the set of MECs of P. Then we define the MDP $P^P = (V^P, E^P, \langle V_1^P, V_R^P \rangle, \delta^P)$ where $V^P = V, V_1^P = V_1, V_R^P = V_R$, $E^P = \bigcup_{i=1}^{k} \{(u, v) \in E \cap (M_i \times M_i)\}$ and for each $v \in V_R$: $\delta^P(v)$ the uniform distribution over vertices u with $(v, u) \in E^P$.

Throughout the algorithm, we maintain the pure MDP graph P^P for an input MDP P. Note that every non-trivial SCC in P^P is also a MEC due to the fact that there are only edges inside of MECs. Moreover, a trivial SCC $\{v\}$ is a MEC iff $v \in V_1$. Note furthermore that when a player-1 edge of an MDP P is deleted, existing MECs might split up into several MECs but no new vertices are added to existing MECs.

Initially, we compute the MEC-decomposition in $\widetilde{O}(m)$ expected time using the algorithm described in Section 4.4.1. Then we remove every edge that is not in a MEC. The resulting graph is the pure MDP graph P^P. Additionally, we invoke a decremental SCC algorithm $\mathscr{A}$ which can (1) announce new SCCs under edge deletions and return a list of their vertices and (2) can answer queries that ask whether two vertices v, u belong to the same SCC. When an edge (u, v) is deleted, we know that (i) the MEC-decomposition stays the same or (ii) one MEC splits up into new MECs and the rest of the decomposition stays the same. We first check if u and v are in the same MEC, i.e., if it exists in P^P. If not, we are done. Otherwise, u and v are in the same MEC C and either (1) the MEC C does not split or (2) the MEC C splits. In the case of (1) the SCCs of the pure MDP graph P^P remain intact and nothing needs to be done. In the case of (2) we need to identify the new SCCs $C_1, \dots, C_k$ in P^P using the decremental SCC algorithm $\mathscr{A}$. Let, w.l.o.g., C_1 be the SCC with the most vertices. We iterate through every edge of the vertices in the SCCs $C_2, \dots, C_k$. By considering all the edges, we identify all SCCs (including C_1) which are also MECs. We remove all edges (y, z) where y and z are not in the same SCC to maintain the pure MDP graph P^P. For the SCCs that are not MECs let U be the set of random vertices with edges leaving its SCC. We compute and remove $A = attr_R(U, P^P)$ (these vertices belong to no MEC due to Lemma 4.4.4) and recursively start the pro-

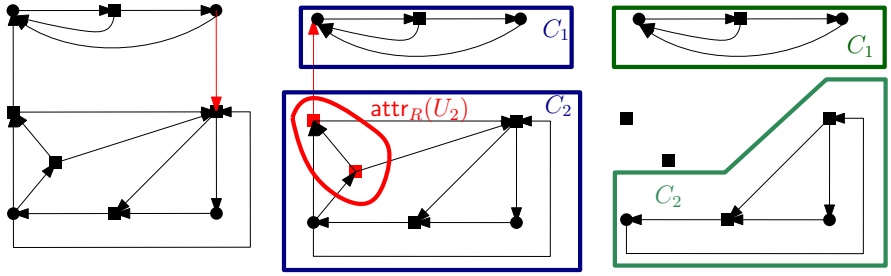


Figure 4.2: We delete an edge which splits the MEC into two new SCCs C_1 and C_2. The SCC C_2 is not a MEC. We thus compute and remove the attractor of U_2 and the resulting SCC is a MEC.

Algorithm 4.3: Decremental MEC-update

Input: Player-1 Edge $e = (u, v)$

1 **if** $\mathscr{A}$.QUERY$(u, v) =$ **true then**
2 List $K \leftarrow \{\mathscr{A}$.DELETE-ANNOUNCE$((u, v))\}$
3 **while** $K \neq \emptyset$ **do**
4 pull a list J of SCCs from K and let C_1 be the largest SCC
5 $\{C_1, \dots C_k\} \leftarrow$ Sort all SCCs in J except C_1 by the smallest vertex id.
6 $\mathsf{MEC}^{C_1} =$ **true**, $U_1 \leftarrow \emptyset$
7 **for** $i = 2;\ i \leq k;\ i{+}{+}$ **do**
8 $\mathsf{MEC}^{C_i} =$ **true**, $U_i \leftarrow \emptyset$
9 **for** $e = (s, t)$ *where* $e \in E(C_i)$ **do**
10 **if** $(s \notin C_i) \vee (t \notin C_i)$ **then** $\mathscr{A}$.DELETE(e)
11 **if** $(s \in V_R \wedge t \notin C_i)$ **then** $\mathsf{MEC}^{C_i} =$ **false**; $U_i \leftarrow U_i \cup \{s\}$
12 **if** $(s \in V_R \wedge s \in C_1)$ **then** $\mathsf{MEC}^{C_1} =$ **false**; $U_1 \leftarrow U_1 \cup \{s\}$
13 **if** $\mathsf{MEC}^{C_i} =$ **false then**
14 $A \leftarrow attr_R(U_i, P^P) \cap C_i$
15 $J \leftarrow \mathscr{A}$.DELETE-ANNOUNCE$(E(A)) \setminus \left(\bigcup_{a \in A} \mathscr{A}.\mathrm{REP}(a)\right)$
16 **if** $J \neq \emptyset$ **then** $K \leftarrow K \cup \{J\}$
17 **if** $\mathsf{MEC}^{C_1} =$ **false then**
18 $A \leftarrow attr_R(U_1, P^P) \cap C_1$
19 $J \leftarrow \mathscr{A}$.DELETE-ANNOUNCE$(E(A)) \setminus \left(\bigcup_{a \in A} \mathscr{A}.\mathrm{REP}(a)\right)$
20 **if** $J \neq \emptyset$ **then** $K \leftarrow K \cup \{J\}$

cedure on the new SCCs generated by the deletion of the attractor. The algorithm is illustrated in Figure 4.2.

Lemma 4.4.5 describes the key invariants of the while-loop at Line 3. We prove it with a straightforward induction on the number of iterations of the while-loop and apply Lemma 4.4.4.

Lemma 4.4.5. *Assume that $\mathscr{A}$ maintains the pure MDP graph P^P before the deletion*

of $e = (u, v)$ then the while-loop at Line 3 maintains the following invariants:

1. *For the graph stored in $\mathscr{A}$ and all lists of SCCs $\{C_1, \ldots, C_k\}$ in K there are only edges inside the SCCs or between the SCCs in the list, i.e., for each $(x, y) \in \bigcup_{j=0}^{k} E[C_j]$ we have $x, y \in \bigcup_{j=0}^{k} C_j$.*

2. *If a non-trivial SCC of the graph in $\mathscr{A}$ is not a MEC of the current MDP it is in K.*

3. *If M is a MEC of the current MDP then we do not delete an edge of M in the while-loop.*

Proof. Initially, $\mathscr{A}$ contains the SCCs of the pure MDP graph P^P. If the deletion of e does not change the SCCs the while loop is not executed and the three invariants hold. Thus in the remainder of the proof we consider the case where deletion of e splits a SCC and the newly created SCCs $C_1, \ldots C_k$ are added to K.

1. Initially, the invariant holds because we assume that we maintain the pure MDP graph in $\mathscr{A}$. Assume the invariant holds before an iteration of the the while-loop. Let C_i be an arbitrary SCC in the list $\{C_1, \ldots, C_k\}$. Due to the for-loop at Lines 9-12 we remove all edges going into or out of C_i (Line 10. Thus all outgoing edges of C_i are removed. Note that because we delete edges between SCCs no new SCCs are created due to this removal. If $i \neq 1$ then, we remove edges inside of C_i at Line 15 which might split up C_i into new SCCs which are added to K. But notice that C_i does not have any outgoing edges and thus the newly created SCCs have only edges inside the SCCs or between them which proves our claim. The same argument holds for $i = 1$ but now edges are removed at Line 19.

2. Initially the invariant holds since we put any potential new SCCs after deleting the input edge e into K at Line 2. Assume the invariant holds before an iteration of the while-loop. We prove that the invariant also holds after this iteration of the while-loop. We only create potential new SCCs when we remove edges in $E(attr_R(U_i, \cap)C_i)$ where $U_i = \{v \in C_i \cap V_R \mid E(v) \cap (V \setminus C_i)\}$ for all $1 \leq i \leq k$. The other edge deletions between SCCs cannot create new SCCs. Consider the removal of any $E(attr_R(U_i, \cap)C_i)$: If the removal creates SCCs we put them into the set J and add them to K (Lines 13-16 and Lines 17-20). On the other hand, if the removal of $E(attr_R(U_i, \cap)C_i)$ does not create new SCCs, U_i is empty and thus C_i has no outgoing random edge. Thus C_i is a MEC. If K is now empty each SCC is a MEC due to the fact that the invariant is true at the end of the last iteration of the while-loop. As we do not remove elements from K, the condition holds for the remaining SCCs in K due to the assumption that the invariant was true before the while-loop.

3. The while loop deletes edges between SCCs at Line 10 and if we delete edges between SCCs we do not remove edges inside any MEC because a MEC is

contained in the SCCs of the graph. If we delete attractors of random vertices with edges leaving their corresponding SCCs (Line 15 and 19), we do not remove edges of any nontrivial MEC due to Lemma 4.4.4.

□

Proposition 4.4.6. *Algorithm 4.3 maintains the pure MDP graph P^P in the data structure $\mathscr{A}$ under player-1 edge deletions.*

Proof. We show that after deleting an edge using Algorithm 4.3 (i) every non-trivial SCC is a MEC and vice-versa, and (ii) there are no edges going from one MEC to another. Initially, we compute the pure MDP graph and both conditions are fulfilled.

When we delete an edge and the while-loop at Line 3 terminates (i) is true due to Lemma 4.4.5(2,3). That is, as we never delete edges within MECs they are still strongly connected and when the while-loop terminates, $K = \emptyset$ which means that all SCCs are MECs.

For (ii) notice that each SCC is once processed as a List J. Consider an arbitrary SCC C_i and the corresponding list of SCCs $J = \{C_1, \dots, C_k\}$ of the iteration in which C_i was identified as a MEC. By Lemma 4.4.5(1) there are no edges to SCCs not in the list. Additionally, due to Line 10 we remove all edges from C_i to other SCCs in J. □

Now that we maintain the pure MDP graph P^P in $\mathscr{A}$, we can answer MEC queries of the form: QUERY(u, v): Returns whether u and v are in the same MEC in P, by an SCC query $\mathscr{A}$.QUERY(u, v) on the pure MDP graph P^P.

The key idea for the running time of Algorithm 4.3 is that we do not look at edges of the largest SCCs but the new SCC decomposition by inspecting the edges of the smaller SCCs. Note that we identify the largest SCC by processing the SCCs in a lockstep manner. This can only happen $\lceil \log n \rceil$ times for each edge. Additionally, when we sort the SCCs, we only look at the vertex ids of the smaller SCCs and when we charge this cost to the vertices we need $O(n \log^2 n)$ additional time.

Proposition 4.4.7. *Algorithm 4.3 maintains the MEC-decomposition of P under player-1 edge deletions in expected total time $\widetilde{O}(m)$. Algorithm 4.3 answers queries that ask whether two vertices v, u belong to the same MEC in $O(1)$. The algorithm uses $O(m + n)$ space.*

Proof. Initialization, i.e., the initial MEC-decomposition can be done in $\widetilde{O}(m)$ expected time as shown in Section 4.4.1. We will prove that the while loop at Line 3 can also be computed in total time $O(T_d + m \log^2 n)$. When the algorithm detects new SCCs, say $C_1, C_2, \dots, C_k$, we consider all vertices of the new SCCs except the vertices in the largest SCC, i.e., w.l.o.g., C_1. Considering all the edges of the vertices in C_i for $i \in [2, k]$ we identify all the SCCs which are also MECs and the set U, i.e., the set of random vertices with edges leaving C_i. This can be achieved with one pass through the edges in C_i. When an SCC C falls apart, an edge can be in the part with at most $|C|/2$ vertices only $\lceil \log n \rceil$ times. As this event can happen only $\lceil \log n \rceil$

times for each edge and we only perform a constant amount of work for each such edge, we obtain a running time of $O(m \log n)$. Also, we sort the newly creates SCCs at Line 5. As we do this only for the smaller SCCs, we can again bound the total running time by $n \log^2 n$: Every vertex can only be in the smaller SCC $\lceil \log n \rceil$ times and sorting the SCCs costs $O(1 + \log n)$ (we need to go through the SCCs except for the largest and check for the smallest vertex id) for each vertex which yields a total running time of $O(n \log^2 n)$.

Computing the attractor at Line 14 or Line 18 only costs total $O(m)$ time as we either delete the edges of the vertices in the attractor or, for the edges we do not delete, we charge the constant amount of work to the edges in the smaller half of the $C/2$ vertices. This results in a running time of $O(T_d + m)$ or expected running time $\widetilde{O}(m)$. Finally, the query operation can be implemented by $\mathscr{A}.\textsc{query}(u, v)$ and takes time $O(1)$. □

Due to the fact that the decremental SCC algorithm we use in Corollary 4.2.2 only works for an oblivious adversary, we prove the following proposition. The key idea is that we sort SCCs returned by the decremental SCC Algorithm. Thus, the order in which new SCCs are returned does only depend on the given instance.

Proposition 4.4.8. *The sequence of deleted edges does not depend on the random choices of the decremental SCC Algorithm but only on the given instance.*

Proof. Note that we only delete edges at Lines 2, 10, 15 and 19. The initial edge deletion at Line 2 does not depend on the decremental SCC algorithm.

We will prove that the claim holds for every iteration of the while-loop at Line 3: Initially, there is only one element in K. Thus the claim holds initially.

Assume that the claim holds before the while-loop. Thus, the list of SCCs we pull from K does not depend on the random choices of the decremental SCC algorithm $\mathscr{A}$. The edges we remove at Line 10 does not create new SCCs because they are not in any SCCs. Note, that when we remove vertices at Lines 15 and 19 the newly created SCCs are returned to J in a order depending on the random choices of the decremental SCC algorithm $\mathscr{A}$. Thus, we sort the SCCs in J by the minimum vertex id and add them according to this order to our list data structure K (the largest is put at the end of the list). Thus, again, the order in which SCCs are processed from Q is fixed and does not depend on the random choices of $\mathscr{A}$. □

The algorithm presented in [31] fulfills all the conditions of Proposition 4.4.7 due to Corollary 4.2.2. Therefore we obtain the following theorem due to Proposition 4.4.6 and Proposition 4.4.7.

Theorem 4.4.9. *Given an MDP with n vertices and m edges, the MEC-decomposition can be maintained under the deletion of $O(m)$ player-1 edges in total expected time $\widetilde{O}(m)$ and we can answer queries that ask whether two vertices v, u belong to the same MEC in $O(1)$ time. The algorithm uses $O(m + n)$ space. The bound holds against an oblivious adversary.*

4.4.4 MDPs with Streett Objectives

Similar to graphs we compute the winning region of Streett objectives with k pairs (L_i, U_i) (for $1 \leq i \leq k$) for an MDP P as follows:

1. We compute the MEC-decomposition of P.
2. For each MEC, we find good end-components, i.e., end-components where $L_i \cap X = \emptyset$ or $U_i \cap X \neq \emptyset$ for all $1 \leq i \leq k$ and label the MEC as satisfying.
3. We output the set of vertices that can almost-surely reach a satisfying MECs.

For 2., we find good *end-components* similar to how we find good components as in Section 4.3. The key idea is to use the decremental MEC-Algorithm described in Section 4.4.3 instead of the decremental SCC Algorithm. We modify the Algorithm presented in Section 4.3 as follows to detect good end-components: First, we use the decremental MEC-algorithm instead of the decremental SCC Algorithm. Towards this goal, we augment the decremental MEC-algorithm with a function to return a list of references to the new MECs when we delete a set of edges. Second, the decremental MEC-algorithm does not allow the deletion of arbitrary edges, but only player-1 edges. To overcome this obstacle, we create an equivalent instance where we remove player-1 edges when we remove 'bad' vertices.

Lemma 4.4.10. *Given an MDP $P = (V, E, \langle V_1, V_R \rangle, \delta)$ with m edges and n vertices, we can maintain a data structure that supports the operation*

- DELETE-ANNOUNCE(E): *Deletes the set of E of player-1 edges (u, v) from the MDP P. If the edge deletion creates new MECs $C_1, \ldots, C_k$ the operation returns a list $Q = \{C_1, \ldots, C_k\}$ of references to the new non-trivial MECs.*

in total expected update time $\widetilde{O}(m)$. The bound holds against an oblivious adaptive adversary.

Proof. The data structure $\mathscr{A}$ maintained in Section 4.4.3 can be extended to support the modified delete operation DELETE-ANNOUNCE(E). To this end, we maintain a list $\mathscr{L}$ of references to the SCCs in P^P that correspond to the new MECs in the current MDP. Whenever we identify a new MEC in Line 13 or Line 17 in Algorithm 4.3. we test whether it is non-trivial and add a reference to that MEC to the list $\mathscr{L}$. Whenever a MEC in the List splits due to another edge update we remove the old identifier from the list and add the new non-trivial MECs (if any). □

Deleting bad vertices. As the decremental MEC-algorithm only allows deletion of player-1 edges, we first modify the original instance $P = (V, E, \langle V_1, V_R \rangle, \delta)$ to a new instance $P' = (V', E', \langle V_1', V_R' \rangle, \delta')$ such that we can remove bad vertices by deleting player-1 edges only. In P' each vertex $v \in V_x$ for $x \in \{1, R\}$ is split into two vertices $v_{in} \in V_1'$ and $v_{out} \in V_x'$ such that $E' = \{(u_{out}, v_{in}) \mid (u, v) \in E\} \cup \{(v_{in}, v_{out}) \mid v \in V\}$ and $L_i' = \{v_{in} \in V' \mid v \in L_i\}$ and $U_i' = \{v_{out} \in V' \mid v \in U_i\}$

for all $1 \leq i \leq k$. The new probability distribution is $\delta'(v_{out})[w_{in}] = \delta(v)[w]$ for $v \in V_R$ and $w \in Out(v)$. Note that for each $v \in V_R$ the corresponding vertex $v_{out} \in V'_R$ has the same probabilities to reach the representation v_{out} of a vertex as v. The described reduction allows us to remove bad vertices from MECs by removing the player-1 edge (v_{in}, v_{out}).

The key idea for the following lemma is that for each original vertex $v \in V$ either both v_{in} and v_{out} are part of a good end-component or none of them. Note that the only way that v_{in} and v_{out} are strongly connected is when the other vertex is also in the strongly connected component because v_{in} (v_{out}) has only one outgoing (incoming) edges to v_{out} (from v_{in}).

Lemma 4.4.11. *There is a good end-component in the modified instance P' iff there is a good component in the original instance P.*

Proof. We show the "if" and the "only if" parts separately.

- (if) Assume there is a good end-component X' in the modified instance P'. Thus, $L'_i \cap X = \emptyset$ or $U'_i \cap X' \neq \emptyset$ for all $1 \leq i \leq k$ and X' is strongly connected. Note that for each vertex v_{in} in X', v_{out} must also be in X' (and vice versa) as otherwise X' is not strongly connected.

  Let $X = \{v \in V \mid v_{in} \in X'\}$. We prove it is a good end-component in P. First we prove that X is strongly connected. Let $v, u \in X$ be arbitrary. Note that $v_{in} \in X'$. There is a path from v_{in} to u_{in} because X' is strongly connected. Thus there is a path from v to u because v posesses all incoming and outgoing edges of v_{in} and v_{out} respectively (the same holds for u and u_{in}, u_{out} and the respective vertices on the path). Moreover, there is no random edge out of X as the random edges in X' only go to vertices v_{in} in X' and we include each corresponding v in X. Thus X is a end-component in P.

  It remains to show that X is also a *good* end-component. Let $i \in [1, k]$. In the first case, $L'_i \cap X' = \emptyset$. But then, by the definition of L'_i, we know that $L_i \cap X = \emptyset$. In the second case, $U'_i \cap X' \neq \emptyset$, i.e., there is some $v_{out} \in U'_i$ in X'. But then the corresponding vertex $v \in U_i$ is in X because $v_{in} \in X'$ due to the fact that X' is otherwise not strongly connected (v_{out} has only one incoming edge: v_{in}). Hence, X is a good end-component in P.

- (only if) Assume there is a good component X in the original instance P. Thus, $L_i \cap X = \emptyset$ or $U_i \cap X' \neq \emptyset$ for all $1 \leq i \leq k$. Let $X' = \{v_{in} \in V' : v \in X\} \cup \{v_{out} \in V' \mid v \in X\}$. We first prove that X' is strongly connected: Let $v_j, u_\ell \in X'$ for $j, \ell \in \{in, out\}$ be arbitrary. By the definition of X', $v, u \in X$ and there is a path from v to u in X. But then there is a path from v_{in} to u_{out} due to the fact that for each z on the path from v to u each z_{in} (z_{out}) on this path contains all incoming edges of z (outgoing edges of z) and $z_{in}, z_{out} \in E'$ which proves the claim. Moreover, there is no random edge out of X' the random edges in X only go to vertices v in X and we include v_{in} in

X'. Thus X' is a end-component in P'. We prove it is a good end-component in P'. Let $i \in [1, k]$. In the first case, $L_i \cap X = \emptyset$. But then, by the definition of L'_i, we know that $L'_i \cap X = \emptyset$. In the second case, $U_i \cap X \neq \emptyset$, i.e., there is some $v \in U_i$ in X. Then the corresponding vertex $v_{out} \in U'_i$ is in X' by the definition of X'. □

On the modified instance P' the algorithm for MDPs is identical to Algorithm 4.1 except that we use a dynamic MEC algorithm instead of a dynamic SCC algorithm.

Theorem 4.4.12. *In an MDP the winning set for a k-pair Street objectives can be computed in $\widetilde{O}(m + b)$ expected time.*

CHAPTER 5

Faster Algorithms for Bounded Liveness in Graphs and Game Graphs

In this chapter, we consider algorithms for computing the winning set of bounded Büchi objectives in graphs and MDPs.

5.1 Introduction

Graphs and games on graphs. Graphs and two-player games played on graphs provide a general mathematical framework for a wide range of problems in computer science: in particular, for the analysis of reactive systems, where the vertices of the graph represent the states of a reactive system and the edges represent the transitions between the states. The classical synthesis problem (the problem of Church) asks for the construction of a winning strategy in a game played on the graph [100, 193, 197] and the fundamental model-checking problem is an algorithmic graph problem [110].

Omega-regular specifications: strength and weakness. In the analysis of reactive systems, the desired temporal properties that the system should satisfy constitute the specification. The class of ω-regular languages provides a robust specification formalism [181, 193]. Every ω-regular objective can be decomposed into a safety part and a liveness part [14]. The safety part ensures that the system will not do anything "bad" (such as violating an invariant) within any finite number of transitions. The liveness part ensures that the system will do something "good" (such as proceed or respond) in the long-run. Liveness can be violated only in the limit, by infinite sequences of transitions, as no bound is specified on when a "good" event must happen. This infinitary formulation has several strengths, such as robustness and

simplicity [181, 219]. However, there is also a weakness of the classical definition of liveness: it can be satisfied by systems that are unsatisfactory because no bound can be put between the occurrence of desired events.

Stronger notion of liveness. For the weakness of the infinitary formulation of liveness, alternative and stronger formulations of liveness have been proposed. The first formulation is *bounded liveness* which ensures, given a bound d, that eventually, good events happen within d transitions. The second formulation is *finitary liveness* which requires the existence of a bound such that, eventually, good events happen within the bound. Finitary liveness was proposed in [18] and has been widely studied; e.g., games on graphs with finitary ω-regular objectives [96], and logics such as PromptLTL based on finitary liveness [166]. The notion of bounded liveness has also been investigated in many contexts, such as MSO with bounding quantifiers [38], bounded model-checking [33], and "bounded until" in logics such as RTCTL [124].

Algorithmic questions for bounded liveness. In this work, we consider graphs and games on graphs with bounded liveness objectives. Consider a graph with n vertices, m edges, and a bounded liveness objective with bound d. A basic algorithmic approach is to reduce the bounded liveness objective to a liveness objective on a larger graph (that we call the auxiliary graph) that explicitly keeps track of the number of transitions since the last good event. This basic approach yields the following bounds: (a) an $O(dm)$-time algorithm for graphs (applying the linear-time algorithm for liveness objectives on graphs), and (b) an $O(n^2d^2)$-time algorithm for games on graphs (applying the current best-known $O(n^2)$-time algorithm for games on graphs with liveness objectives [91]). A fundamental algorithmic question is whether the above bounds can be improved.

Our contributions. In this work, our main contributions are improved algorithmic bounds for bounded liveness on graphs and games on graphs.

- In graphs, there are two relevant semantics: (a) an existential semantic that asks whether there exists a path to satisfy the objective, and (b) a universal semantic that asks whether all paths satisfy the objective. The answer to the universal semantics with bounded liveness is "Yes" if and only if the answer is "No" for existential semantics with the complementary bounded coliveness objective. We consider graphs with the existential semantics and bounded liveness and bounded coliveness objectives. For bounded liveness objectives, all previous algorithmic approaches yield an $O(n^3)$ worst-case time-bound (where $d = O(n)$) and we present a randomized algorithm with one-sided error whose worst-case time-bound is $O(n^{2.5} \log n)$. For bounded coliveness objectives, we present a deterministic linear-time algorithm.

- For games on graphs with bounded liveness objectives, we present an $O(n^2d)$-time algorithm that improves the previous $O(n^2d^2)$-time algorithm.

Significance of the contributions. On the technical front, it is threefold.

1. To break the $O(n^3)$-time barrier for graphs, we exploit randomization to estimate for all pairs of good events how far they are from each other. Using this information along with a suitably modified auxiliary graph results in the faster $O(n^{2.5} \log n)$-time algorithm.

To get the improved time bound of $O(n^2 d)$ for game graphs:

2. we construct an auxiliary game graph (similar to the graph case) and make a crucial observation that this game graph after each iteration has a lot of structure, a property we call *induced symmetry*;

3. we strategically introduce as many "layover" vertices as there are good events; in combination with induced symmetry, this enables us to prove that a significant chunk of the auxiliary game graph is deleted after each iteration.

Furthermore, there are several important implications of our contributions. First, for graphs with bounded liveness objectives, the previous worst-case time-bound is $O(n^3)$. In recent years, many such algorithmic problems with $O(n^3)$ bound are conditionally optimal with a reduction from classical problems such as BMM (boolean matrix multiplication) [3, 6, 72, 75, 87, 222]. Our new algorithm breaks the $O(n^3)$ barrier and shows that such conditional lower bound approaches do not apply for bounded liveness in graphs. Second, for graphs with bounded coliveness objectives our linear-time bound shows that there is a very efficient algorithm for the complement of the bounded liveness objectives. Finally, we show that the basic algorithmic approach for games on graphs can also be improved. Given our results improve the bounds for graphs and games on graphs with bounded liveness objectives, there are several interesting questions for future work. Whether the bounds can be further improved or a deterministic sub-cubic time algorithm can be obtained for graphs with bounded liveness objectives are the most interesting algorithmic open questions.

5.2 Algorithms for Graphs

Graphs are a special case of game graphs with $V_2 = \emptyset$. Hereon, we will call this "the graph case" as opposed to "the game graph case" (where $V_1 \neq \emptyset$ and $V_2 \neq \emptyset$). The objectives we consider are *prefix independent*, i.e., if $\omega \in \Omega$, then any play obtained by adding or removing a finite prefix to or from ω is also in Ω. Hence, with respect to computing winning vertices, it is enough to focus on strongly connected graphs. The reasoning is as follows.

In the input graph, we call a strongly connected component (SCC) S *good* if the graph restricted to S has a winning vertex. Due to prefix independence, all vertices in a good SCC and those from which you can reach a good SCC are winning. We will prove that such vertices are exactly the winning vertices, and that this set can be computed by the following procedure:

- Compute the SCCs of the input graph (can be done in linear time [216]).
- Determine for each SCC if it is good (this step depends on the objective).
- Consider the set of all vertices belonging to a good SCC. Perform reachability to this set. (This can also be done in linear time.)

Lemma 5.2.1. *A vertex v is a winning vertex if and only if it there is path from v to some vertex in a good SCC.*

Proof. As mentioned before, due to prefix independence, if v has a path to some vertex in a good SCC, then it is winning. Next, we show the converse.

If v is winning, then there is a winning play ω starting at v. Since SCCs themselves form a directed acyclic graph (DAG), ω must eventually enter an SCC S and stay there. Again, due to prefix independence, the vertices visited by ω in S are also winning, i.e., S is a good SCC. □

By Lemma 5.2.1 and the procedure described above it, the problem of computing the winning vertices is reduced to determining, given a strongly-connected graph, whether there is a winning vertex or not. More formally, we get the following lemma.

Lemma 5.2.2. *Let $S_1, S_2, \ldots$ be SCCs of the graph $G = (V, E)$. When $V_2 = \emptyset$, i.e., in the graph case, for a prefix independent objective, the set of winning vertices can be computed in time $O(m + \sum_i t(S_i))$ time, where $m = |E|$ and $t(S_i)$ is the time required to compute whether S_i is a good SCC or not.*

In this chapter, we consider bounded Büchi and bounded coBüchi objectives.

5.2.1 The Bounded Büchi Objective

We are given a graph $G = (V, E)$, a set B of Büchi vertices, and a positive integer d. A cyclic-walk in G is a walk $(v_1, v_2, \ldots, v_\ell)$ such that $v_1 = v_\ell$. We say that a cyclic-walk C is *feasible* if it has at least one Büchi vertex and the number of edges in C between any two consecutive Büchi vertices is at most d. We assume that G is strongly connected, and our goal is to determine if there is a winning vertex in G. Then, using Lemma 5.2.2, we generalize the result to a graph that might not be strongly connected. The following lemma reduces this problem to finding a feasible cyclic-walk in G.

Lemma 5.2.3. *The strongly-connected input graph G has a winning vertex with respect to the bounded Büchi objective if and only if it has a feasible cyclic-walk.*

Proof. If G has a winning vertex, say v, then there is a winning play ω that starts at v. Let $\omega = \langle v_0 = v, v_1, v_2, \ldots \rangle$; so by the definition of winning play, $\exists i \geq 1$ such that $\forall j \geq i : \{v_j, v_{j+1}, \ldots, v_{j+d-1}\} \cap B \neq \emptyset$. Consider the set $Inf(\omega)$ of vertices that appear infinitely often in ω. Since ω is winning, $Inf(\omega) \cap B \neq \emptyset$. Thus, we can choose

a $j' \geq i$ such that $v_{j'} \in Inf(\omega) \cap B$. Since $v_{j'}$ appears infinitely often, for some $j'' > j'$, we have that $v_{j''} = v_{j'}$. Thus $(v_{j'}, v_{j'+1}, \ldots, v_{j''} = v_{j'})$ is a feasible cyclic-walk because $j' \geq i$ and, as mentioned earlier, $\forall j \geq i : \{v_j, v_{j+1}, \ldots, v_{j+d-1}\} \cap B \neq \emptyset$.

In the other direction, if G has a feasible cyclic-walk, then we can keep traversing it to construct a winning play, which means G has a winning vertex. □

An $O(dm)$-time algorithm for bounded Büchi

Next, we recall the basic $O(dm)$-time algorithm to determine if there is a feasible cyclic-walk. This algorithm tries to trace a feasible cycle by maintaining a counter with each possible non-Büchi vertex denoting how far away we are from the last visit to a Büchi vertex. We construct a $(d+1)$-layered auxiliary graph $G^* = (V^*, E^*)$, where $V^* = (B \times \{0\}) \cup ((V \setminus B) \times \{1, \ldots, d\})$. We define a more general graph here that we also use in Section 5.3. We illustrate an example in Figure 5.1. So, for $(v, \ell) \in V^*$, the integer ℓ corresponds to the aforementioned counter. We call the vertices in $B \times \{0\}$ Büchi vertices and the vertices in $(V \setminus B) \times \{1, \ldots, d\}$ non-Büchi vertices. The edge set E^* is constructed by Algorithm 5.1. The last layer of the auxiliary graph is actually not needed for the graph case but is needed for the game graph case later. Observe that the auxiliary graph is also a game graph. (The ownership of the vertices will be defined later in a natural way.)

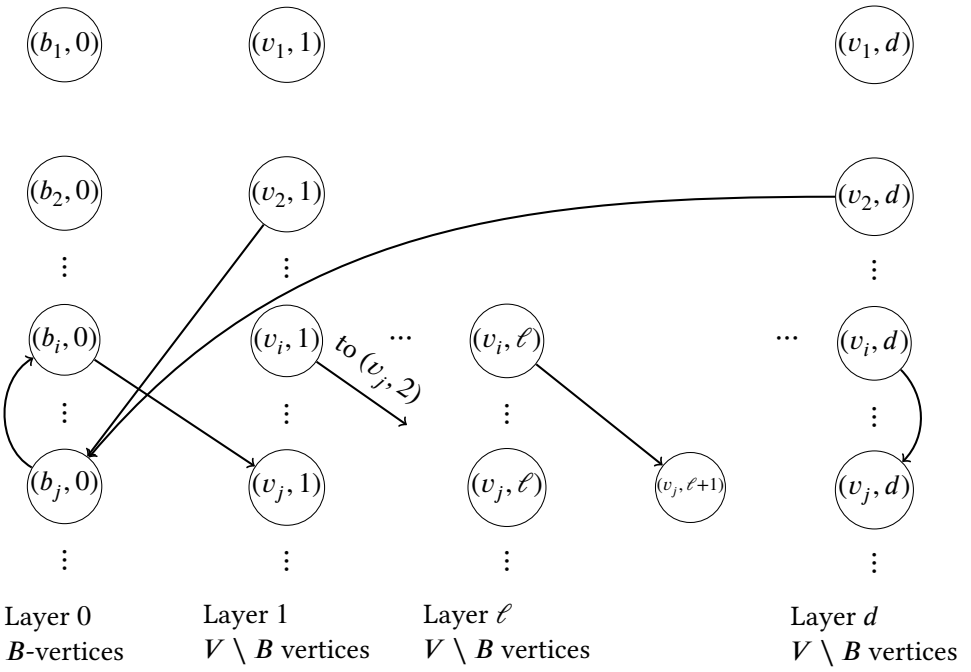


Figure 5.1: An illustration of how the auxiliary layered graph is constructed. If G contains the edges (b_j, b_i), (b_i, v_j), (v_2, b_j), and (v_i, v_j), then the auxiliary layered graph G^* will have shown edges.

Algorithm 5.1: Construction of the auxiliary graph G^* from G, B, and d. It is easy to see that the running time of this algorithm is $O(dm)$ and G^* has at most dm edges.

```
1  Procedure ConstructAuxiliaryGraph(G = (V, E), B ⊆ V, d)
2      V* ← (B × {0}) ∪ ((V \ B) × {1, ..., d}) and E* ← ∅
3      for (u, v) ∈ E such that v ∉ B (add counter-incrementing edges) do
4          if u ∉ B then
5              for i ∈ {1, ..., d−1} do
6                  Add ((u, i), (v, i+1)) to E*.
7              Add ((u, d), (v, d)) to E* (edges in the last layer to V \ B stay in the last
                layer).
8          else
9              Add ((u, 0), (v, 1)) to E*.
10     for (u, v) ∈ E such that v ∈ B (add counter-resetting edges) do
11         if u ∉ B then
12             for i ∈ {1, ..., d} do
13                 Add ((u, i), (v, 0)) to E*.
14         else
15             Add ((u, 0), (v, 0)) to E*.
16     return G* = (V*, E*)
17 Procedure AuxiliaryGraph-d-Layers(G = (V, E), B ⊆ V, d)
18     G* ← ConstructAuxiliaryGraph(G = (V, E), B ⊆ V, d)
19     Return the graph resulted by removing layer-d from G*, called G' = (V', E').
```

Lemma 5.2.4. *The running time of the procedures* ConstructAuxiliaryGraph(·) *and* AuxiliaryGraph-d-Layers(·) *in Algorithm 5.1 is* $O(dm)$.

Proof. In ConstructAuxiliaryGraph(·), each of the outer for loop runs for at most m iterations, and each of the inner for loops runs for at most d iterations. AuxiliaryGraph-d-Layers(·) just calls ConstructAuxiliaryGraph(·) and removes the last layer, which takes $O(dm)$ time. □

For the graph case, we are interested in G^* induced on layers-$\{0, 1, \ldots, d-1\}$. Let G' denote this graph.

Lemma 5.2.5. *The strongly-connected input graph G has a feasible cyclic-walk if and only if G' has a cycle.*

Proof. Let $C = (b_1, v_{1,1}, \ldots, v_{1,\ell_1}, b_2, v_{2,1}, \ldots, v_{2,\ell_2}, b_3, \ldots, b_1)$, where each $b_i \in B$, each $v_{i,j} \in V \setminus B$, and each $\ell_i \leq d-1$, be a feasible cyclic-walk in G. There is a corresponding cyclic-walk C' in G':

- for each $(b_i, v_{i,1}) \in C$, the edge $((b_i, 0), (v_{i,1}, 1)) \in E'$,

- for each $(v_{i,j}, v_{i,j+1}) \in C$, the edge $((v_{i,j}, j), (v_{i,j+1}, j+1)) \in E'$,
- for each $(v_{i,\ell_j}, b_{i+1}) \in C$, the edge $((v_{i,\ell_j}, \ell_j), (b_{i+1}, 0)) \in E'$, and
- for the final edge $(v_{i,\ell_j}, b_1) \in C$, the edge $((v_{i,\ell_j}, \ell_j), (b_1, 0)) \in E'$.

If C' consists of union of cycles can be short-cut to get a cycle in G'.

In the other direction, consider a cycle in G'. A projection of this cycle on the first coordinate of the vertices, by construction, gives a feasible cyclic-walk in G, because the number of edges between consecutive Büchi vertices is at most d. □

Thus, by Lemmas 5.2.3 and 5.2.5, we get Algorithm 5.2.

Algorithm 5.2: This algorithm determines if the strongly-connected input graph has a winning vertex with respect to the bounded Büchi objective.

```
1 Procedure BoundedBüchi(G = (V, E), B ⊆ V, d)
2     G' ← AuxiliaryGraph-d-Layers(G, B, d)
3     Run depth-first search on G' to determine if it has a cycle.
4     if G' has a cycle then
5         return "G has a winning vertex."
6     else
7         return "G does not have a winning vertex."
```

Lemma 5.2.6. *Algorithm 5.2 determines if the strongly-connected input graph G has a winning vertex with respect to the bounded Büchi objective in $O(dm)$ time.*

Proof. By Lemmas 5.2.3 and 5.2.5, G has a winning vertex if and only if G' has a cycle. Since a depth-first search finds if there is a cycle in G', the correctness of the algorithm is established. By Lemma 5.2.4, AuxiliaryGraph-d-Layers$(\cdot)$ takes $O(dm)$ time, and a depth-first search on G' takes time $O(dm)$, because the number of edges in G' is $O(dm)$. □

Thus, by Lemma 5.2.2, we get the following theorem.

Theorem 5.2.7. *The set of winning vertices for the bounded Büchi objective in the graph case can be computed in time $O(dm)$.*

Proof. Let $S_1, S_2, \ldots$ be SCCs of the input graph $G = (V, E)$. Let $m_1, m_2, \ldots$ be the number of edges in the SCCs $S_1, S_2, \ldots$. Then, by Lemma 5.2.6, for $i = 1, 2, \ldots$, we can determine in time $O(dm_i)$ whether S_i is good. Since $m \geq \sum_i m_i$, the proof is complete by Lemma 5.2.2. □

An $O(|B|m)$-time algorithm for bounded Büchi

Now, we briefly discuss an $O(|B|m)$-time algorithm for bounded Büchi. Given $G = (V, E)$ and B, consider the graph $G' = (B, E')$ such that $(b, b') \in E'$ if the distance from b to b' in G is at most d. We allow self loops in G'. It is easy to see that G has a feasible-cyclic walk if and only if G' has a cycle. To construct G', we perform $|B|$ breadth-first searches, one starting from each vertex in B. This takes time $O(|B|m)$. Then, by a similar argument as in the proof of Theorem 5.2.7, we get the following theorem.

Theorem 5.2.8. *The set of winning vertices for the bounded Büchi objective in the graph case can be computed in time $O(|B|m)$.*

Remark 5.2.9. Note that both algorithms that we have seen so far can take $\Theta(n^3)$ time if $m = \Theta(n^2)$ and B and d are $\Theta(n)$. The next algorithm we see is combinatorial and has running time $O(n^{2.5} \log n)$ for the worst setting of the parameters and breaks the cubic barrier. This also rules out any conditional lower bound approaches to get an $\Omega(n^3)$ lower bound for combinatorial algorithms.

An $O((m + |B|^2)\sqrt{n} \log n)$-time algorithm for bounded Büchi

In this section, we present an $O((m + |B|^2)\sqrt{n} \log n)$-time algorithm for bounded Büchi in the graph case. This is one of our main contributions. Here, we give a procedure that computes distances between all pairs of Büchi vertices if the distance is at least $\sqrt{N}$, where $N \geq |V|$ is a parameter that we will fix later. This information can be used to reduce the number of layers in the auxiliary graph to $\sqrt{N}$. By dist, we denote the distance function for G. For any $u, v \in V$, if $u \neq v$, then $\text{dist}(u, v)$ denotes the length of a shortest path from u to v, and for any $u \in V$, $\text{dist}(u, u)$ denotes the length of a shortest cycle through u.

Lemma 5.2.10. *Let $N \geq |V|$. Algorithm 5.3 determines with probability at least $1 - 1/N^2$ if the strongly-connected input graph G has a winning vertex with respect to the bounded Büchi objective in $O((m + |B|^2)\sqrt{N} \log N)$ time. It never returns a false positive, i.e., if it outputs that G has a winning vertex, then it is correct with probability 1. Its running time is $O((m + |B|^2)\sqrt{N} \log N)$.*

Proof. If $d < \sqrt{N}$, then we are done by Lemma 5.2.6. Thus, we assume for the rest of the proof that $d \geq \sqrt{N}$.

For any $b, b' \in B$, by $T(b, b')$, we denote a fixed shortest cycle through b if $b = b'$ or a fixed shortest path from b to b' otherwise. Let the event that a vertex $v(b, b') \in T(b, b')$ is sampled into S be denoted by $\mathscr{E}(b, b')$. Since $v(b, b') \in T(b, b')$, we have that $\text{dist}(b, b') = \text{dist}(b, v(b, b')) + \text{dist}(v(b, b'), b')$. This implies that if $\mathscr{E}(b, b')$ occurs, then $\text{dist}(b, v(b, b'))$ and $\text{dist}(v(b, b'), b')$ are computed by the algorithm using the incoming and outgoing BFS at $v(b, b')$, and hence $\text{dist}^S(b, b') = \text{dist}(b, b')$. Let $\mathscr{E}^c(b, b')$ be the complement of $\mathscr{E}(b, b')$. Now, $\Pr[\mathscr{E}^c(b, b')] = (1 -$

Algorithm 5.3: This algorithm determines if the strongly-connected input graph has a winning vertex with respect to the bounded Büchi objective.

```
1  Procedure RandBoundedBüchi(G = (V, E), B ⊆ V, d, N)
2      if d < √N then
3          return BoundedBüchi(G = (V, E), B ⊆ V, d)
4      Sample 4√N ln N vertices uniformly at random, independently, and with
         replacement.
5      S ← the set of sampled vertices.
6      for s ∈ S do
7          Perform incoming and outgoing breadth-first search (BFS) to and from s.
             Compute distances dist(b, s) and dist(s, b) for each b ∈ B during the BFSs.
8      G′ ← AuxiliaryGraph-d-Layers(G, B, √N − 1) for b ∈ B do
9          for b′ ∈ B do
10             dist^S(b, b′) ← ∞
11             for s ∈ S do
12                 dist^S(b, b′) ← min{dist^S(b, b′), dist(b, s) + dist(s, b′)}
13             if dist^S(b, b′) ≤ d then
14                 Add ((b, 0), (b′, 0)) to E′ (this would be a self-loop if b = b′).
15     Run depth-first search on G′ to determine if it has a cycle.
16     if G′ has a cycle then
17         return "G has a winning vertex."
18     else
19         return "G does not have a winning vertex."
```

$\text{dist}(b,b')/|V|)^{4\sqrt{N}\ln N}$, because $1 - \text{dist}(b,b')/|V|$ is the probability that a fixed sample does not contain a vertex of $T(b,b')$ and we draw $4\sqrt{N}\ln N$ independent samples.

For any $b, b' \in B$, where $\text{dist}(b,b') \geq \sqrt{N}$, we denote the event that $\text{dist}^S(b,b') = \text{dist}(b,b')$ by $\mathscr{E}'(b,b')$. As noted earlier, $\text{dist}^S(b,b') = \text{dist}(b,b')$ if $\mathscr{E}(b,b')$ occurs, hence:

$$
\begin{aligned}
\Pr[\mathscr{E}'(b,b')] &\geq \Pr[\mathscr{E}(b,b')] && \mathscr{E}(b,b') \text{ is a subevent of } \mathscr{E}'(b,b'),\\
&= 1 - \Pr[\mathscr{E}^c(b,b')] \\
&= 1 - \left(1 - \frac{\text{dist}(b,b')}{|V|}\right)^{4\sqrt{N}\ln N} && \text{by the argument earlier,}\\
&\geq 1 - \left(1 - \frac{1}{\sqrt{N}}\right)^{4\sqrt{N}\ln N} && \text{because } \text{dist}(b,b')/|V| \geq 1/\sqrt{N},\\
&\geq 1 - \frac{1}{N^4} && \text{by well-known fact } (1-1/x)^x \leq 1/e.
\end{aligned}
$$

Since $N \geq |B|$, by the union bound, we have $\Pr[\forall (b, b') \in B \times B : \mathscr{E}'(b, b')] \geq 1 - 1/N^2$. Let us condition on the event that for all $(b, b') \in B \times B : \mathscr{E}'(b, b')$, and let G' be the auxiliary graph constructed by the algorithm.

Suppose G has a winning vertex. By Lemma 5.2.3, there is a feasible cyclic-walk C in G. Then for any consecutive Büchi vertices b and b' in C, either $\text{dist}(b, b') \geq \sqrt{N}$, in which case there is an edge $((b, 0), (b', 0))$ or $\text{dist}(b, b') < \sqrt{N}$, in which case there exists a cycle $((b, 0), (u_1, 1), (u_2, 2), \ldots, (u_\ell, \ell), (b', 0))$ in G', where $\ell < \sqrt{N} - 1$. Thus, C induces a cycle in G'.

On the other hand, if there is a cycle C' in G', then a projection of C' on the first coordinate of the vertices, by construction of G', gives a feasible cyclic-walk in G after replacing all edges in C' of the form $((b, 0), (b', 0))$ by corresponding paths of length at most d that certify $\text{dist}^S(b, b')$. By Lemma 5.2.3, G has a winning vertex.

Also, if the algorithm does return that G has a winning vertex, then G' has a cycle, and the existence of a feasible cyclic-walk in G can be shown in the same way as above. This shows that the algorithm never returns a false positive.

Running time Incoming and outgoing BFSs from the vertices in S take time $O(m\sqrt{N} \log N)$. AuxiliaryGraph-d-Layers$(\cdot)$ takes $O(m\sqrt{N})$ time. Computing dist^S takes time $O(|B|^2 \sqrt{N} \log N)$. DFS on G' takes time $O(|B|^2 + m\sqrt{N})$. In total, Algorithm 5.3 has running time $O((m + |B|^2)\sqrt{N} \log N)$. □

Finally, we use Lemma 5.2.2 to generalize the above to a graph that may not be strongly connected. Fix N to be n in Algorithm 5.3 when running it for each SCC. Then, by a similar argument as in the proof of Theorem 5.2.7, we get the following theorem.

Theorem 5.2.11. *The set of winning vertices for the bounded Büchi objective can be computed with probability at least* $1 - 1/n$ *in time* $O((m + |B|^2)\sqrt{n} \log n)$ *which is* $O(n^{2.5} \log n)$. *Moreover, the algorithm never returns a false positive, i.e., each vertex in the set it outputs is a winning vertex with probability* 1.

Proof. Let $S_1, S_2, \ldots$ be SCCs of the input graph $G = (V, E)$. Let $m_1, m_2, \ldots$ be the number of edges and by $\beta_1, \beta_2, \ldots$, be the number of Büchi vertices in the SCCs $S_1, S_2, \ldots$, respectively. Then, by Lemma 5.2.10, for $i = 1, 2, \ldots$, the algorithm outputs in time $O((m_i + \beta_i^2)\sqrt{n} \log n)$ whether S_i is good. Since $m \geq \sum_i m_i$ and $|B|^2 = (\sum_i \beta_i)^2 \geq \sum_i \beta_i^2$, the running time bound is proved.

The probability bound is obtained by a union bound over at most n SCCs. Moreover, the algorithm never returns a *false positive* by Lemma 5.2.10. □

5.2.2 The Bounded coBüchi Objective

Given a graph $G = (V, E)$, a set C of vertices, and a positive integer d, a walk W is called a *feasible* walk if $W \subseteq C$ and the number of vertices in W is at least d.

We assume that G is strongly connected, and our goal is to determine if there is a winning vertex in G for the bounded coBüchi objective. The following lemma reduces this problem to finding a feasible walk in G.

Lemma 5.2.12. *The strongly-connected input graph G with a set C of vertices has a winning vertex with for the bounded coBüchi objective if and only if it has a feasible walk.*

Proof. If G has a winning vertex, say v, then there is a winning play ω that starts at v. Let $\omega = \langle v_0 = v, v_1, v_2, \ldots \rangle$; so by the definition of winning play, $\forall i \geq 0, \exists j \geq i : \{v_j, v_{j+1}, \ldots, v_{j+d-1}\} \subseteq C$. Any such walk $v_j, v_{j+1}, \ldots, v_{j+d-1}$ is a feasible walk.

In the other direction, say G has a feasible walk $v'_1, v'_2, \ldots, v'_\ell$, where $\ell \geq d$ and $v'_i \in C$ for $i \in \{1, \ldots, \ell\}$. Now, consider the cyclic walk $v'_1, v'_2, \ldots, v'_{d-1}, v'_d, v'_{d-1}, \ldots, v'_1$. We keep traversing it to construct a winning play. The existence of a winning play implies that G has a winning vertex. □

Now we give an $O(m)$-time algorithm to determine if the strongly-connected input graph G has a feasible walk.

Algorithm 5.4: This algorithm determines if the strongly-connected input graph has a winning vertex with respect to the bounded coBüchi objective.

```
1 Procedure BoundedcoBüchi(G = (V, E), C ⊆ V, d)
2     G' ← G induced on C
3     Perform DFS on G'; if there is a cycle, return "G has a winning vertex."
4     if LongestPath(G') ≥ d then
5         return "G has a winning vertex."
6     else
7         return "G does not have a winning vertex."
```

Algorithm 5.5: An algorithm to compute the length of a longest path in a directed acyclic graph.

```
1 Procedure LongestPath(G = (V, E))
2     Compute the topological ordering T of G.
3     L ← integer array of size |V|, initialized to all zeros.
4     M ← 0
5     for vertex v in order of T do
6         for incoming edges (u, v) do
7             L[v] ← max{L[v], L[u] + 1}
8         M ← max{M, L[v]}
9     return M
```

Lemma 5.2.13. *Algorithm 5.4 determines if the strongly-connected input graph G has a winning vertex with respect to the bounded coBüchi objective in $O(m)$ time.*

Proof. If G' has a cycle, then this cycle can be repeated to get a feasible walk in G, which, by Lemma 5.2.12, means that G has a winning vertex.

If G' is acyclic and has the length of longest path at least d, then there is a feasible walk in G, which, by Lemma 5.2.12, means that G has a winning vertex. Conversely, if G' is acyclic and the length of longest path in G' is less than d then G cannot contain a feasible walk, which, by Lemma 5.2.12, means that G does not have a winning vertex.

The calls to DFS on G' and LongestPath(G') take time $O(m)$; hence, the running time of Algorithm 5.4 is $O(m)$. □

By a similar argument as in the proof of Theorem 5.2.7, we get the following theorem.

Theorem 5.2.14. *The set of winning vertices for the bounded coBüchi objective in the graph case can be computed in time $O(m)$.*

5.3 Algorithms for Game Graphs

In this section, we present algorithms for the bounded Büchi objective in game graphs. We first introduce the auxiliary *game graph* similar to the auxiliary graph defined earlier. We then show that we can compute in $O(n^2d^2)$ time the winning set of a given bounded Büchi objective on game graphs by computing the winning set of a coBüchi objective on the auxiliary game graph. Finally, we show how to improve the running time to $O(n^2d)$ by using structural properties of the auxiliary game graph and adapting a known technique for solving Büchi Games [91].

The Auxiliary Game Graph. Given a game graph $\Gamma = (V, E, \langle V_1, V_2\rangle)$ with n vertices, m edges and a bounded Büchi objective boundedBüchi(B, d), we first construct the auxiliary graph by calling ConstructAuxiliaryGraph($(V, E), B, d$) in Algorithm 5.1 and additionally partition the vertices of the auxiliary graph V^* into player-1 vertices V_1^* and player-2 vertices V_2^*, i.e., for each $(v, \ell) \in V^*$ we get $(v, \ell) \in V_1^*$ if $v \in V_1$ and $(v, \ell) \in V_2^*$ if $v \in V_2$. The auxiliary game graph has $O(nd) = O(n^2)$ vertices and $O(md) = O(mn)$ edges. We say that a vertex $(v, \ell) \in V^*$ is a *layer-ℓ vertex* and v is its *first component*.

For any play λ, we denote by λ_k the kth vertex of the play. If a play has a superscript, it denotes the starting vertex of the play, e.g λ^v means that the play λ starts at v. By λ_k^v we refer to the kth vertex of the play λ^v which starts at v. Given a finite feasible play $\lambda^{(w,\ell)}$ in Γ^* starting at (w, ℓ), we define $\mathrm{Proj}(\lambda^{(w,\ell)})$ to be the projection of $\lambda^{(w,\ell)}$ on the first component of the vertices in it; by definition, this finite play starts at w and is feasible in Γ. Analogously, given a finite feasible play λ^w in Γ, we define $\mathrm{Lift}(\lambda^w, \ell)$ to be the unique finite feasible play in Γ^* starting at (w, ℓ) such that the first component of $\mathrm{Lift}(\lambda^w, \ell)_k$ is the same as λ_k^w. For (u, v) in

E such that $(u, j) \in V^*$ define (the appropriate next layer number if you followed the copy of (u, v) starting in layer j)

$$\mathrm{NxtLyr}(u, v, j) = \begin{cases} j + 1 \text{ if } j < d \text{ and } v \notin B \\ d \text{ if } j = d \text{ and } v \notin B \\ 0 \text{ if } v \in B\,. \end{cases}$$

Now, define $\mathrm{Lift}(\lambda^w, \ell)_1 = (w, \ell)$, and for $k > 1$, given $\mathrm{Lift}(\lambda^w, \ell)_{k-1} = (\lambda^w_{k-1}, j)$ define $\mathrm{Lift}(\lambda^w, \ell)_k = (\lambda^w_k, \mathrm{NxtLyr}(\lambda^w_{k-1}, \lambda^w_k, j))$. Similarly, given the finite feasible play $\lambda^{(w,\ell)}$ in Γ^*, we define $\mathrm{Shift}(\lambda^{(w,\ell)}, \ell')$ to be the finite play that starts at (w, ℓ') in Γ^* such that, for any k, the first components of $\lambda^{(w,\ell)}_k$ and $\mathrm{Shift}(\lambda^{(w,\ell)}, \ell')_k$ are the same. By construction of Γ^* the finite play $\mathrm{Shift}(\lambda^{(w,\ell)}, \ell')$ is well-defined because (1) edges going from layer-i vertices to layer-$(i + 1)$ vertices ($1 \le i \le d - 1$) exist in all layers with the same respective first components except in layer-d where these edges go again to layer-d, (2) edges going to layer-0 vertices exist in all layers ($1 \le i \le d$) and (3) because edges originating from layer-0 vertices implies that both plays are currently visiting the same layer-0 vertex.

In comparison, the goal of the two operations $\mathrm{Proj}(\cdot)$ and $\mathrm{Lift}(\cdot)$ is to map finite plays between Γ^* and Γ such that the finite play in Γ^* has, for all vertices, the same first component as the corresponding finite play in Γ and vice versa. In contrast, $\mathrm{Shift}(\lambda^{(w,\ell)}, \ell')$ maps a finite play in Γ^* to a finite play also in Γ^* which has the same first component but a "shifted" starting vertex.

5.3.1 An $O(n^2d^2)$-time Algorithm for Bounded Büchi in Games

In this section, we show that we can compute the winning set of a given *bounded Büchi objective* on game graphs by computing the winning set of a *coBüchi objective* on the auxiliary game graph. Then we apply the best-known algorithm for computing the winning set of a Büchi objective on the auxiliary game graph to get the desired result. In the following lemma, we prove that computing $W_1(\mathrm{boundedBüchi}(B, d, \Gamma))$ is the same as computing $W_1(\mathrm{coBüchi}(C^*, \Gamma^*))$ where C^* are the vertices in layers-$\{0, 1, \ldots, d-1\}$. Intuitively, when a play ϕ in $\mathrm{coBüchi}(C^*, \Gamma^*)$ stays in layers-$\{0, 1, \ldots, d-1\}$, it reaches a vertex in layer 0 every at most d steps by construction of Γ^*. The layer-0 vertices correspond to the vertices in B which means that a play ϕ' in Γ defined as the projection on the first component of the vertices in ϕ visits a vertex in B every at most d steps which implies that $\phi' \in \mathrm{boundedBüchi}(B, d, \Gamma)$. On the other hand, when player 1 has a strategy in Γ to visit a vertex in B every at most d steps, a similar strategy which visits the same vertices in the first component in Γ^* allows player 1 to stay in the first d layers of the auxiliary graph.

Lemma 5.3.1. *Let $\Gamma = (V, E, \langle V_1, V_2\rangle)$ be a game graph with bounded Büchi objective $\mathrm{boundedBüchi}(B, d)$, let $\Gamma^* = (V^*, E^*, \langle V^*_1, V^*_2\rangle)$ be the corresponding auxiliary game graph, and let C^* be the vertices in the first d layers of the auxiliary*

graph, i.e., $C^* = \{(v,i) \in V^* \mid 0 \le i \le d-1\}$. *Then* $\{w \mid (w,i) \in W_1(\textit{coBüchi}(C^*, \Gamma^*)),$ *for some* $0 \le i \le d\} = W_1(\textit{boundedBüchi}(B, d, \Gamma))$.

Proof. We first prove that $\{w \mid (w,i) \in W_1(\text{coBüchi}(C^*, \Gamma^*)),$ for some $0 \le i \le d\} \subseteq W_1(\text{boundedBüchi}(B, d, \Gamma))$. Let $(w,i) \in W_1(\text{coBüchi}(C^*, \Gamma^*))$. Then player 1 has a winning strategy σ^* in Γ^* such that for all player-2 strategies π^*, we have that $\omega((w,i), \sigma^*, \pi^*) \in \text{coBüchi}(C^*, \Gamma^*)$.

Whenever player 1 makes a move in Γ^*, we define the corresponding move in Γ as follows: For any finite play λ^w in Γ that ends in a player-1 vertex, define $\sigma(\lambda^w)$ to be the first component of $\sigma^*(\text{Lift}(\lambda^w, i))$. (It does not matter how we define σ for plays that do not start at w.)

Next, we argue why σ is a winning player-1 strategy for $\text{boundedBüchi}(B, d, \Gamma)$ starting at w. Let π be an arbitrary player-2 strategy in Γ. We define a corresponding player-2 strategy π^* in Γ^*: for $\lambda^{(w,i)}$ that ends in a player-2 vertex (u,j), let $v = \pi(\text{Proj}(\lambda^{(w,i)}))$ and define $\pi^*(\lambda^{(w,i)}) = (v, \text{NxtLyr}(u, v, j))$.

Now, it is straightforward to show that the first component of $\omega((w,i), \sigma^*, \pi^*)_k$ is equal to $\omega(w, \sigma, \pi)_k$ by induction on k.

Since the play $\omega((w,i), \sigma^*, \pi^*) \in \text{coBüchi}(C^*, \Gamma^*)$, it stays in C^* after a finite number of steps. Note that to stay in C^* means to visit a layer-0 vertex after every at most d steps because there are only d layers in C^* and each step that does not go to a layer-0 vertex increases the layer counter. Since the first component of each layer-0 vertex is in B, the play $\omega(w, \sigma, \pi)$ visits a vertex in B every at most d steps after a finite number of steps and is in $\text{boundedBüchi}(B, d, \Gamma)$.

The other direction, i.e., $W_1(\text{boundedBüchi}(B, d, \Gamma)) \subseteq \{w \mid (w,i) \in W_1(\text{coBüchi}(C^*, \Gamma^*))$, for some $0 \le i \le d\}$ can be shown with a similar argument.

Let $w \in W_1(\text{boundedBüchi}(B, d, \Gamma))$. Then player 1 has a winning strategy σ in Γ such that for all player-2 strategies π, we have that $\omega(w, \sigma, \pi) \in \text{boundedBüchi}(B, d, \Gamma)$.

Whenever player 1 makes a move in Γ, we define the corresponding move in Γ^* as follows: Let $0 \le i \le d$ be arbitrary such that $(w,i) \in V^*$. For any finite play $\lambda^{(w,i)}$ in Γ that ends in a player-1 vertex (u,j), let $v = \sigma(\text{Proj}(\lambda^{(w,i)}))$ and define $\sigma^*(\lambda^{(w,i)}) = (v, \text{NxtLyr}(u, v, j))$.

Next, we argue why σ^* is a winning player-1 strategy for $\text{coBüchi}(C^*, \Gamma^*)$ starting at (w,i). Let π^* be an arbitrary player-2 strategy in Γ^*. We define a corresponding player-2 strategy π in Γ: For the finite play λ^w that ends in a player-2 vertex u, let $\pi(\lambda^w)$ be the first component of $\pi^*(\text{Lift}(\lambda^w, i))$ (it does not matter how we define π for plays that do not start at w).

Now it is straightforward to show by induction on k that the first component of $\omega((w,i), \sigma^*, \pi^*)_k$ is equal to $\omega(w, \sigma, \pi)_k$.

Since the play $\omega(w, \sigma, \pi) \in \text{boundedBüchi}(B, d, \Gamma)$, it visits a vertex in B every at most d steps after a finite number of steps. Thus, the play $\omega((w,i), \sigma^*, \pi^*)$ visits a layer-0 vertex every at most d steps after a finite number of steps. Hence, $\omega((w,i), \sigma^*, \pi^*)$ stays in C^* as the play increments the layer counter to at most $d-1$ and is in $\text{coBüchi}(C^*, \Gamma^*)$. □

To compute $W_1(\text{coBüchi}(C^*))$ in Γ^*, we observe that, by the duality of Büchi objectives, $W_1(\text{coBüchi}(C^*)) = V^* \setminus W_2(\text{Büchi}(V^* \setminus C^*)) = V^* \setminus W_2(\text{Büchi}(\{(v,d) \in V^*\}))$. Since, traditionally, we always compute the player-1 winning set of a given objective, we swap player-1 and player-2 vertices in Γ^*. Then we compute $W = W_1(\text{Büchi}(\{(v,d) \in V^*\}))$ using the algorithm of Chatterjee and Henzinger [91], which is the fastest algorithm for Büchi games known, and project $V^* \setminus W$ on the first coordinate. We illustrate the details in Algorithm 5.6.

Algorithm 5.6: Determine $W_1(\text{boundedBüchi}(B,d))$, given a game graph Γ

Procedure BoundedBüchiGames($\Gamma = (V, E, \langle V_1, V_2\rangle), B, d$)
  $(V^*, E^*) \leftarrow$ ConstructAuxiliaryGraph((V, E))
  $V_1^* \leftarrow \{(v,i) \in V^* \mid v \in V_1\}, V_2^* \leftarrow \{(v,i) \in V^* \mid v \in V_2\}$
  $\Gamma^* \leftarrow (V^*, E^*, V_1^*, V_2^*)$; $B^* \leftarrow \{(v,d) \in V^* \mid v \in V \setminus B\}$
  $W \leftarrow$ BüchiGamesFast($\Gamma^* = (V^*, E^*, \langle V_2^*, V_1^*\rangle), B^*$) ([91], Algorithm 5.7)
  return $\{x \mid (x,i) \in V^* \setminus W$ *for some* $0 \le i \le d\}$

The correctness of Algorithm 5.6 is due to the correctness of the fast Büchi games algorithm [91, Theorem 2.14], the argument above, and Lemma 5.3.1. The argument for the running time of Algorithm 5.6 is as follows. We first construct Γ^* in $O(md)$ time and then compute the winning set of coBüchi(C^*) in time $O(|V^*|^2)$ [91, Theorem 2.14]. As $|V^*| = O(nd)$ and $d = O(n)$, we get the following theorem.

Theorem 5.3.2. *The set of winning vertices for the bounded Büchi objectives in games can be computed in time* $O(n^2d^2) = O(n^4)$.

5.3.2 An $O(n^2d)$-time Algorithm for Bounded Büchi in Games

In this section, we give a refined running time analysis of Algorithm 5.6 giving us an $O(n^2d)$-time algorithm for bounded Büchi games. We first describe the fastest algorithm for Büchi Games [91] for completeness. Then, we identify key ideas of the refined running time analysis when the input is an auxiliary game graph and prove the improved running time formally.

The Büchi Games Algorithm of [91]

Given a game graph $\Gamma = (V, E, \langle V_1, V_2\rangle)$ and a set B of Büchi vertices[1], we fix an order on the edges. In this fixed order, the edges (u,v) where u is a non-Büchi player-2 vertex, i.e., $u \in (V_2 \setminus B)$, come before all other edges. We call them priority-1 edges. All the other edges are priority-0 edges.

Definition 5.3.3. *Given a game graph* $\Gamma = (V, E, \langle V_1, V_2\rangle)$, *let* $\Gamma_i = (V, E_i, \langle V_1, V_2\rangle)$ *for* $1 \le i \le \log n$ *be a subgraph of* Γ *which we define as follows: For all* $u \in V$, *the set* E_i *contains the following edges:*

[1] not to be confused with the input for the bounded Büchi problem in the previous and later sections

1. *If the outdegree of u in E is at most 2^i, E_i contains all edges of the form (u, v), i.e., if $|Out(u)| \leq 2^i$ then the set $\{(u, v) \mid v \in Out(u)\} \subseteq E_i$.*

2. *If the edge (v, u) belongs to the first 2^i inedges of vertex u in E, we have $(v, u) \in E_i$ ("first" means with respect to to the fixed order we specified above).*

Note that $E_{i-1} \subseteq E_i$ since the order of the edges is fixed. We form a partition of V in Γ_i by giving each vertex a color:

- *Blue: A player-1 vertex v in Γ_i is blue if the outdegree of v is greater than 2^i.*
- *Red: A player-2 vertex u in Γ_i is red if it has no outedge in E_i.*[2]
- *All other vertices are white.*

Thus, if a player-1 vertex is white then all its outedges are in E_i, and if a player-2 vertex is white then it has at least one outgoing edge in E_i.

Algorithm description. The input of Algorithm 5.7 is a game graph Γ and a set of Büchi vertices B. Recall that every vertex in a player-1 closed set S without Büchi vertices cannot be in the player-1 winning set of the given Büchi objective $W_1(\text{Büchi}(B))$ (Proposition 2.5.2 (2)). We repeatedly find such a set S by removing from V the player-1 attractor of the set B (Proposition 2.5.3) and forming S from all the remaining vertices. Then we remove the player-2 attractor of S. In the algorithm, we identify such a set S_j at Line 10 and remove the attractor at Line 14. Note that a naive algorithm would take $O(nm)$ time, as the attractor of S could always be of size 1 and computing the attractor is in $O(m)$ time. To obtain a quadratic-time (in the number of vertices) algorithm, the improved algorithm of Chatterjee and Henzinger constructs, for $i = 1, \ldots, \log n$, the graph Γ_i which has at most 2^i edges. Due to the properties of Γ_i, it can be shown that the set S_j has size of at least 2^{i-1}. In this way, the attractor computation takes time proportional to the removed vertices. Since player-1 vertices with missing outgoing edges or player-2 vertices with no outgoing edge in Γ^i, i.e., non-white vertices might still be able to reach a vertex in B, we compute the player-1 attractor of the non-white vertices combined with the vertices in B. We illustrate the details in Algorithm 5.7.

The removal of player-1 closed sets in Γ_i now includes vertices which are blue, red, and white. The definition of a *separating cut* further refines the definition of the winning regions for player 2 in this regard.

Separating cut. A set S of vertices induces a separating cut in a game graph Γ_i or Γ_i^j in Algorithm 5.7 if

1. the only edges from S to $V \setminus S$ come from player-2 vertices in S
2. every player-2 vertex in S has an edge to another vertex in S

[2] In the algorithm of Chatterjee and Henzinger [91] red vertices are player-2 vertices where an edge of E is missing. We change this definition slightly, i.e., without changing their algorithm or correctness argument, by saying that player-2 vertices are red if they do not have any outedges in E_i.

Algorithm 5.7: Determine $W_1(\text{Büchi}(B))$, given a game graph Γ [91]

```
1  Procedure BüchiGamesFast(Γ = (V, E, ⟨V_1, V_2⟩), B)
2      Let j ← 0; U ← ∅; Y_0 ← attr_1(B, Γ); S_0 ← V \ Y_0; D_0 ← attr_2(S_0, Γ); Γ^j ← Γ
3      j ← j + 1; while D_{j−1} ≠ ∅ do
4          Remove the vertices in D_{j−1} from Γ^{j−1} to obtain Γ^j; and U ← U ∪ D_{j−1}
5          i ← 1
6          repeat
7              Construct Γ_i^j from Γ^j as described in Definition 5.3.3.
8              Let Z_i^j be the vertices of V^j that are either red or blue
9              Y_i^j ← attr_1(B^j ∪ Z_i^j, Γ_i^j)
10             S_j ← V^j \ Y_i^j
11             i ← i + 1
12         until S_j is nonempty or i ≥ 1 + log n
13         if S_j ≠ ∅ then
14             D_j ← attr_2(S_j, Γ^j)
15         else
16             return V \ U
17         j ← j + 1
```

3. every player-1 vertex in S is white and

4. $B \cap S = \emptyset$.

Thus, a separating cut S is a player-1 closed set where (i) player-1 vertices are white and which (ii) does not contain a vertex in B.

The following lemmas are needed to establish the improved running time guarantees in the next section. Detailed proofs can be found in the paper by Chatterjee and Henzinger [91].

Lemma 5.3.4 below says that the set S_j is indeed a separating cut in Γ^j (not only in Γ_i^j) and that due to the careful construction of Γ_i^j from the game graph Γ^j in iteration j, S_j does not include a vertex of the player-1 attractor of the Büchi vertices in Γ^j.

Lemma 5.3.4 ([91], Lemma 2.9)**.** *Let S_j be the non-empty set computed by Algorithm 5.7 in iteration j. Then, (1) S_j is a separating cut in Γ^j; and (2) $S_j \cap attr_1(B^j, \Gamma^j) = \emptyset$.*

Lemma 5.3.5 establishes that the separating cut found in Γ_i^j is indeed the maximum separating cut in Γ_i^j. Also, if Γ_i^j contains a separating cut, Algorithm 5.7 finds it.

Lemma 5.3.5 ([91], Lemma 2.11)**.** *Let Γ_i^j be the game graph in iteration j of the outer loop and iteration i of the inner loop. If S induces a separating cut in Γ_i^j, then $S \subseteq S_j$.*

Lemma 5.3.6 says that the set S_j is a separating cut in Γ_i^j. This does not follow from Lemma 5.3.4(1) because Γ_i^j might have less edges than Γ^j and separating cuts are not preserved if we only consider a subset of edges in Γ^j (property 2 might be violated).

Lemma 5.3.6 ([91], Lemma 2.12)**.** *Consider an iteration j of the outer loop of Algorithm 5.7 such that the algorithm stops the inner loop at value i and identifies a non-empty set S_j. Then, S_j is a separating cut in Γ_i^j.*

Faster Algorithm for Bounded Büchi Games

In this section, we give the refined running time analysis of Algorithm 5.6. We note that Γ^* gets redefined to be $(V^*, E^*, \langle V_2^*, V_1^* \rangle)$ in Algorithm 5.6 on Line 5. Therefore, from here on, when we say player 1 (respectively player 2), we mean the player controlling the vertices in V_2^* (respectively, those in V_1^*).

Distinct vertices. We call a set of vertices S in Γ^* *distinct* if, for each pair of vertices $(v, \ell), (v', \ell') \in S$, we have $v \neq v'$.

Copies of a vertex. Let $Copies(v)$ denote the set of "copies" of a vertex $v \in V^*$, i.e., for a layer-0 vertex $(v, 0)$ we have that $Copies((v, 0)) = \{(v, 0)\}$ and for a vertex (v, ℓ), where $\ell > 0$, we have $Copies((v, \ell)) = \{(v, 1), \ldots, (v, d)\}$.

The improved running time guarantee is due to two key ideas.

Key idea 1. When *there is a* vertex (v, ℓ) in D_j then $Copies((v, \ell)) \subseteq D_j$, i.e., *all* its copies are in D_j.

On a very high level, the argument is that if there is a player-2 strategy to go from a vertex to S_j, then there exists a player-2 strategy from all copies of that vertex to S_j. While the idea is simple to state, complicated machinery is needed to prove it formally. We prove the key idea in Claim 5.3.12 building on Definition 5.3.10 and Claim 5.3.11.

Now, if we follow the original running-time argument [91], then we can only claim that we remove 2^{i-1} vertices in *total* if the inner loop at Line 12 stops at iteration i, but the second key idea states something stronger.

Key idea 2. If the inner loop at Line 12 stops at iteration i^*, we remove 2^{i^*-1} *distinct* vertices.

Combining the key ideas, we remove from the game graph in iteration j all copies of those distinct vertices. The ith iteration of the loop at Lines 12–12 takes time $O(2^i nd)$ for constructing the auxiliary version of $(\Gamma^*)_i$ and performing the attractor computations. The iterations of the loop in Lines 12–12 before $i' < i$ amount to a total running time of $O(2^i nd)$. Thus, we charge the 2^{i-1} removed distinct vertices the cost of the iteration and the iterations before, i.e., each such removed original vertex is charged $O(nd)$. As we can remove only n distinct vertices since they correspond to the vertices in the game graph Γ, we have a total cost of $O(n^2 d)$.

For the second key idea to work, we must modify the original bounded Büchi instance (Γ, B, d) carefully. For every vertex in $v \in B$ we add a player-2 vertex v'

which is not in B and an edge (v', v). Then we redirect all edges which go to v in the original instance and make them go to v' instead, i.e., for all $v \in B$ we have $V_2 \leftarrow V_2 \cup \{v'\}$ and $E \leftarrow (E \cup \{(v', v)\} \cup \{(u, v') \mid (u, v) \in E\}) \setminus \{(u, v) \in E\}$. Also, we increase d by one, as we increase the distance to all vertices in B by one. Note that this simple modification allows us to assume, without loss of generality, that all vertices in B have incoming edges from player-2 vertices only. Since we swap the player-1 vertices with player-2 vertices in Algorithm 5.6 we can assume that all incoming edges to a layer-0 vertex are from player-1 vertices. This adds at most n vertices and edges to Γ.

Observation 5.3.7. *We can assume, without loss of generality, that all layer-0 vertices $v \in V^*$ of the auxiliary game graph Γ^* created at Line 2 in Algorithm 5.6 have no incoming edges from player-2 vertices, i.e., if $(v, 0) \in V^*$ then $In((v, 0)) \cap V_2^* = \emptyset$.*

With the above observation, we can prove the following proposition which is the crux of this section.

Proposition 5.3.8. *Algorithm 5.6 runs in time $O(n^2 d) = O(n^3)$.*

Proof. In this proof we denote by (Γ^*, B^*) the input of Algorithm 5.7 at Line 5 of Algorithm 5.6. The input to Algorithm 5.6 is (Γ, B, d). If we can show that the running time of the call to Algorithm 5.7 at Line 5 is in $O(n^2 d) = O(n^3)$ we are done, as the rest of the operations of Algorithm 5.6 are in $O(md)$. This entails constructing (Γ^*, B^*) and going through W. We therefore prove the following lemma.

Lemma 5.3.9. *The total time Algorithm 5.6 spends in Algorithm 5.7 is $O(n^2 d) = O(n^3)$.*

Every vertex v in Γ^* has only $O(n)$ out-edges by the definition of the auxiliary game graph. Thus, when we consider the graphs $(\Gamma^*)_i$ of Definition 5.3.3 for $1 \leq i \leq \log n$, we have $(\Gamma^*)_{\log n} = \Gamma^*$. The construction of $(\Gamma^*)_i$ $(1 \leq i \leq \log n)$ takes time $O(nd \cdot 2^i)$.

We split the running time argument into two parts. In the first part, we bound the running time of all except the last iteration of the while loop at Line 3. In the second part of the analysis, we bound the running time of the last iteration of the same loop.

Running time bound for all iterations of the while loop except the last. Consider iteration j, and assume that Algorithm 5.7 stops the repeat-until loop at Line 12 with value i^* and it is not the last iteration of the while loop at Line 3. Thus, S_j is not empty. By Lemma 5.3.6, the set S_j is a separating cut in $(\Gamma^*)^j_{i^*}$. We make a detour to set up some claims.

We need the following definition because it helps us translate plays and strategies from a vertex to its copies.

Definition 5.3.10. *If Γ^*_s is an induced subgraph of Γ^* such that for all (u, ℓ_s) in Γ^*_s we have that $Copies((u, \ell_s))$ are also in Γ^*_s, then we say that Γ^*_s has the induced-symmetry property or that it is symmetrically induced.*

The following claim is about the translation of a strategy from a vertex to its copy.

Claim 5.3.11. *Suppose Γ_s^* is symmetrically induced. Then, in Γ_s^*, if a player has a strategy to reach a copy of w from a copy of u, then from all copies of u, she has a strategy to reach some copy of w. More formally, in Γ_s^*, if player ρ has a strategy π to reach (w, ℓ_d) from (u, ℓ_s), then for all copies (u, ℓ'_s), she also has a strategy π' to reach (w, ℓ'_d) for some ℓ'_d.*

Proof. We define π'. Consider a finite feasible play $\lambda^{(u,\ell'_s)}$ that ends in a player-ρ vertex (v, j). Let $\pi(\text{Shift}(\lambda^{(u,\ell'_s)}, \ell_s)) = (y, p)$. Define $\pi'(\lambda^{(v,\ell'_s)}) = (y, \text{NxtLyr}(v, y, j))$. Now, the play $\text{Shift}(\lambda^{(u,\ell'_s)}, \ell_s)$ is feasible and the strategy π' is well defined because Γ_s^* is symmetrically induced.

We argue why player ρ can reach a copy of w using π'. Let σ' be an arbitrary strategy for the other player, i.e., player $(3 - \rho)$. For any finite feasible play $\lambda^{(u,\ell_s)}$ that ends in a player-$(3 - \rho)$ vertex (v, j), let $\sigma'(\text{Shift}(\lambda^{(u,\ell_s)}, \ell'_s)) = (y, p)$. Define $\sigma(\lambda^{(u,\ell_s)}) = (y, \text{NxtLyr}(v, y, j))$. Again, $\text{Shift}(\lambda^{(u,\ell_s)}, \ell'_s)$ is feasible and σ is well defined because Γ_s^* is symmetrically induced.

Now, it is straightforward to show by induction on k that the first components of $\omega((u, \ell_s), \sigma, \pi)_k$ and $\omega((u, \ell'_s), \sigma', \pi')_k$ are the same. This means that if $\omega((u, \ell_s), \sigma, \pi)_k$ reaches (w, ℓ_d), then $\omega((u, \ell'_s), \sigma', \pi')_k$ reaches (w, ℓ'_d) for some ℓ'_d.

□

The following claim is a formal version of the first key idea.

Claim 5.3.12. *If a vertex (v, ℓ) is in D_j, then $Copies((v, \ell)) \subseteq D_j$; and, $(\Gamma^*)^j$ has induced symmetry.*

Proof. We prove the claim by induction on j.

Base case, $j = 0$. If $(v, \ell) \in D_0$, then there is a player-2 strategy π_1 to reach $(w, p) \in S_0$. The set $S_0 = V \setminus attr_1(B^*, \Gamma^*)$ is a player-1 closed set by Observation 2.5.3: This means that there is a player-2 strategy π_2 to stay inside S_0. By construction of Γ^*, any edge from a non-layer-d vertex goes to the next layer or to layer-0. Then, since $S_0 \cap B^* = \emptyset$, that is, since S_0 does not contain any layer-d vertices, any (infinite) play that stays inside S_0 must eventually return to layer-0. Thus, player 2 can first use π_1 to reach $(w, p) \in S_0$ from (v, ℓ), then use π_2 to reach $(x, 0) \in S_0$ from (w, p); effectively, this gives a player-2 strategy to go to $(x, 0) \in S_0$ from (v, ℓ). Then, by Claim 5.3.11, player 2 has a strategy to reach a copy of $(x, 0)$ from (v, ℓ') for any ℓ' because Γ^* itself has induced symmetry. Now, $(x, 0)$ does not have any other copy, this means player 2 has a strategy to reach $(x, 0) \in S_0$ from (v, ℓ'). By induced symmetry of Γ^* again, we have that all copies of (v, ℓ), i.e., $Copies((v, \ell))$ are in Γ^*; moreover, by the above argument, for each of these copies, there is a player-2 strategy to reach S_0, which implies that $Copies((v, \ell)) \subseteq D_0$. Noting that $(\Gamma^*)^0 = \Gamma^*$ has induced symmetry finishes the base case.

Induction step, $j \geq 1$. By induction hypothesis, $(\Gamma^*)^{j-1}$ has induced symmetry, and if a vertex (v, ℓ) is in D_{j-1}, then $Copies((v, \ell)) \subseteq D_{j-1}$. This implies that deleting D_{j-1} from $(\Gamma^*)^{j-1}$ to get $(\Gamma^*)^j$ means deleting all copies of a vertex being deleted. Therefore, since $(\Gamma^*)^{j-1}$ has induced symmetry, $(\Gamma^*)^j$ also has induced symmetry.

Since S_j is a separating cut (by Lemma 5.3.4), it is a player-1 closed set. Thus, by the same argument as in the base case that uses the induced symmetry of $(\Gamma^*)^j$, if (v, ℓ) is in D_j, then $Copies((v, \ell)) \subseteq D_j$. This completes the induction step and the proof. □

The following claim is the formal proof of the second key idea.

Claim 5.3.13. *The set S_j contains at least 2^{i^*-1} distinct vertices.*

Proof. The proof is similar to the proof of [91, Lemma 2.13] except that we must now argue that all of the 2^{i^*-1} vertices are distinct. Consider the set S_j in the game graph of the iteration before, i.e., we argue about S_j in $(\Gamma^*)^j_{i^*-1}$. Note that we have the following two cases.

- In the first case, S_j contains a player-1 vertex (x, ℓ) for $1 \leq \ell \leq d$ that is blue in $(\Gamma^*)^j_{i^*-1}$. Thus, (x, ℓ) has outdegree at least 2^{i^*-1} in $(\Gamma^*)^j_{i^*}$ and none of these edges go to vertices in $V^j \setminus S_j$ in $(\Gamma^*)^j_{i^*}$. Thus, S_j contains at least 2^{i^*-1} vertices. Note that vertex (x, ℓ) can only have edges to vertices which are distinct to (x, ℓ), i.e., for all $((x, \ell), (y, \ell')) \in E^*$ we have $x \neq y$ because the game graph Γ does not have self loops.

- In the second case, all player-1 vertices in S_j are white in $(\Gamma^*)^j_{i^*-1}$. Thus, their outedges in $(\Gamma^*)^j_{i^*}$ and $(\Gamma^*)^j_{i^*-1}$ are identical. We now argue, why a player-2 vertex in S_j exists: Assume for contradiction that no player-2 vertex in S_j exists. Hence, S_j is a separating cut only consisting of player-1 vertices. As S_j is a separating cut in $(\Gamma^*)^j_{i^*}$ we have $S_j \cap B = \emptyset$. Thus, S_j is also a separating cut in $(\Gamma^*)^j_{i^*-1}$. But then, by Lemma 5.3.5, the algorithm would have terminated in iteration $i^* - 1$ which is a contradiction because it terminated in iteration i^*.

  Note that repeat-until loop at Lines 12–12 would have stopped in iteration $i^* - 1$ in $(\Gamma^*)^j_{i^*-1}$ as all player-1 vertices in S_j are white.

  Consider a player-2 vertex u in S_j. Note that u must have an edge $(u, v) \in (E^*)^j_i$ with $v \in S_j$ because S_j is a separating cut in $(\Gamma^*)^j_{i^*}$ (Lemma 5.3.6). Again, there are two possibilities:

  - For all player-2 vertices $u \in S_j$ there exists a vertex $v \in S_j$ with $(u, v) \in (E^*)^j_{i^*-1}$. But then S_j would be a separating cut in $(\Gamma^*)^j_{i^*-1}$ as the outedges of player 1 are identical in $(\Gamma^*)^j_{i^*}$ and $(\Gamma^*)^j_{i^*-1}$. By Lemma 5.3.5, the separating cut would have been found in iteration i^*-1 of the repeat-until loop at Line 12, which is a contradiction.

– Therefore, there exists a player-2 vertex $u \in S_j$ that has an edge $(u, v) \in (E^*)^j_{i^*}$ to a vertex $v \in S_j$ but this edge is not contained in $(E^*)^j_{i^*-1}$. This can only happen if v has at least 2^{i^*-1} other inedges in $(E^*)^j_{i^*-1}$. Note that u is a player-2 vertex not in $(B^*)^j$ (because all vertices of $(B^*)^j$ belong to Y^j), and hence the edge (u, v) has priority 1 and recall that by the fixed inorder of edges priority-1 edges come before all priority-0 edges. Thus, it follows that since the edge (u, v) is not in $(\Gamma^*)^j_{i^*-1}$, all inedges of v that are in $(\Gamma^*)^j_{i^*-1}$ must have priority 1 by the fixed order of inedges, that is, all the inedges of v in $(\Gamma^*)^j_{i^*-1}$ are from non-Büchi player-2 vertices. Note that $v \in S_j$ and since S_j is a separating cut and, thus, a closed set, all player-2 vertices which are not in B^* with an edge to v are also in S_j. Since v has at least 2^{i^*-1} inedges from player-2 vertices which are not in B^*, the set S_j must contain at least 2^{i^*-1} vertices.

  Furthermore, all incoming edges are from distinct vertices: Note that v cannot be a layer 0 vertex of Γ^*, because by Observation 5.3.7 all vertices in B of the given bounded Büchi objective have no incoming edges from a player-2 vertex. Also, layer-d vertices cannot be in S_j as they are in B^* and would be in the player-1 attractor $Y^j_{i^*}$ computed at Line 9. All other vertices in Γ^* have incoming edges only from distinct vertices. Thus, all 2^{i^*-1} such vertices are distinct. □

Due to Claim 5.3.13, S_j contains at least 2^{i^*-1} distinct vertices, and since $S_j \subseteq D_j$, the set D_j also contains all copies of all vertices in S_j due to Claim 5.3.12. All of D_j is deleted. We resume from the detour. The time spent in all graphs $(\Gamma^*)^j_1, \ldots, (\Gamma^*)^j_{i^*}$, i.e., the time spent in the repeat-until loop at Line 12 for the graph construction and the attractor computations, sums up to $O(2^{i^*} \cdot nd)$. We charge $O(nd)$ work to each distinct vertex. This accounts for all the running time except for the last iteration of the outer loop. Since we always remove all copies of a vertex $v \in S_j$, the algorithm deletes at most n distinct vertices throughout a run of the algorithm. Thus, the total time spent over the whole algorithm other than the last iteration is $O(n^2 d)$.

The last iteration of the outer loop. In the last iteration j^* of the outer loop, when no vertex is deleted, the algorithm works on all $\log n$ game graphs, spending time $O(n \cdot 2^i)$ on game graph $(\Gamma^*)^{j^*}_i$. Since each graph $(\Gamma^*)^{j^*}_i$ has at most $nd \cdot 2^{i+1}$ edges and there are $\log n$ graphs, the total number of edges worked in the last iteration is

$$\sum_{i=1}^{\log n} nd \cdot 2^{i+1} = 4nd \sum_{i=1}^{\log n} 2^{i-1} = 4nd(2^{\log n} - 1) = 4nd(n-1) = O(n^2 d).$$

□

Theorem 5.3.14. *The set of winning vertices for the bounded Büchi objective and bounded coBüchi objectives in game graphs can be computed in time* $O(n^2 d) = O(n^3)$.

CHAPTER 6

Algorithms and Conditional Lower Bounds for Planning Problems

In this chapter, we consider queries of reachability objectives and sequential reachability objectives in graphs, MDPs, and game graphs.

6.1 Introduction

One of the basic and fundamental algorithmic problems in artificial intelligence is the *planning problem* [171, 200]. The most basic planning problem is the *discrete feasible planning problem* [171]. The problem has a finite *state space* and a finite amount of *actions* for each state. Starting from an *initial state*, the planner repeatedly chooses an available action at the current state which, as a result, produces a new current state as described by a *state transition function*. The question is if the planner can produce a state which is in a certain subset of the state space called *goal* or *target*[1].

Planning models. We study this problem in the following classical models:

- *Graphs.* Discrete Feasible Planning can be directly translated into a graph search problem: The vertices in the graph describe the state space and for every action in a state, there is an edge to the vertex which corresponds to the new state given by the state transition function [171, 200].

[1]This chapter includes results originally intended for the planning community and thus we use planning-specific language and motivate the problems from a "planning perspective".

- *MDPs.* In the presence of interaction with nature, the graph model is extended with probabilities or stochastic transitions, which gives rise to Markov Decision Processes (MDPs) [129, 134, 146, 191, 195].

- *Games on graphs.* In the presence of interaction with an adversarial environment, the graph model is extended to game graphs (or AND-OR graphs) [136, 179].

Planning problems. The planner tries to solve a planning problem given one of the above described planning models. The starting position is not restricted to the vertices controlled by the planner but can be any kind of vertex in the considered model. We consider the following basic planning problems:

- *Reachability.* Given a set T of target vertices the goal is to determine if some target vertex from the starting position is reachable.

- *Coverage.* In the coverage problem we are given k different target sets, namely, $T_1, \ldots, T_k$, and a starting vertex. The coverage problem asks whether we can achieve reachability for all target sets T_i where $1 \leq i \leq k$. Coverage models the following scenarios: Consider a robot stationed in an outpost and k different locations of interest. If an event or an attack happens in one of the locations, then that location must be reached. However, the location of the event or the attack is not known in advance and the robot must be prepared that the target set could be any of the k target sets.

- *AllCoverage.* In the AllCoverage problem there are again k different target sets $T_1, \ldots, T_k$ but in contrast to Coverage we want to determine *all starting positions* where Coverage with $T_1, \ldots, T_k$ holds. This corresponds to finding a viable outpost for the above described robot.

- *Sequential reachability.* In the *sequential reachability* problem we are given k different target sets, namely, $T_1, T_2, \ldots, T_k$ and a starting position. The goal is to output whether we can first reach T_1, then T_2 and so on up to T_k from the starting position. This represents the scenario that the tasks must be achieved in a sequence by the planner.

The above are natural planning problems and have been studied widely in the literature, e.g., in robot planning [99, 163, 170].

Basic Planning Questions. For the above problems the *basic* planning questions are as follows: (a) for graphs, the question is whether there exists a plan (or a path) such that the planning problem is solved; (b) for MDPs, the basic question is whether there exists a strategy such that the planning problems is satisfied almost-surely (i.e., with probability 1); and (c) for games on graphs, the basic question is whether there exists a strategy that solves the planning problem irrespective of the choices of the adversary. The almost-sure satisfaction for MDPs is also known as the strong

	Graphs		MDPs		Games	
Objectives	Upper B.	Lower B.	Upper B.	Lower B.	Upper B.	Lower B.
Reachability	$O(m)$		$\widetilde{O}(m)$		$O(m)$	
Coverage	$O(m + \sum_{i=1}^{k} \lvert T_i \rvert)$		$\widetilde{O}(k \cdot m)$	$\tilde{\Omega}(k \cdot m)$ (Thm. 6.2.1)	$O(k \cdot m)$	$\tilde{\Omega}(k \cdot m)$ (Thm. 6.2.8)
AllCoverage	$O(k \cdot m)$	$\tilde{\Omega}(k \cdot m)$ (Thm. 6.3.1)	$O(k \cdot m)$	$\tilde{\Omega}(k \cdot m)$ (Thm. 6.3.1)	$O(k \cdot m)$	$\tilde{\Omega}(k \cdot m)$ (Thm. 6.3.1)
Sequential	$O(m + \sum_{i=1}^{k} \lvert T_i \rvert)$ (Thm. 6.4.6)		$\widetilde{O}(m + \sum_{i=1}^{k} \lvert T_i \rvert)$ (Thm. 6.4.12)		$O(k \cdot m)$	$\tilde{\Omega}(k \cdot m)$ (Thm. 6.4.13)

Table 6.1: Algorithmic bounds where n and m are the number of vertices and edges of the underlying model, and k denotes the number of different target sets. The $\tilde{\Omega}(\cdot)$ bounds are conditional lower bounds (CLBs) under the BMM conjecture and SETH. They establish that polynomial improvements over the given bound are not possible, however, polylogarithmic improvements are not excluded. Note that CLBs are quadratic for $k = \Theta(n)$. The new results are highlighted in boldface.

cyclic planning in the planning literature [1], and games on graphs question represent planning in the presence of a worst-case adversary [136, 179] (aka adversarial planning, strong planning [201], or conformant/contingent planning [40, 144, 190]).

Algorithmic study. In this chapter, we study the planning problems for graphs, MDPs, and games on graphs algorithmically. For all the above questions, polynomial-time algorithms exist. When polynomial-time algorithms exist, proving an unconditional lower bound is extremely rare. A new approach in complexity theory aims to establish a conditional lower bound (CLB) based on a well-known conjecture. Two standard conjectures for CLBs are as follows: The (a) *Boolean matrix multiplication (BMM)* conjecture states that there is no sub-cubic combinatorial algorithm for boolean matrix multiplication; and the (b) *Strong exponential-time hypothesis (SETH)* states that there is no sub-exponential time algorithm for the k-SAT problem when k grows to infinity. Many CLBs have been established based on the above conjectures, e.g., for dynamic graph algorithms and string matching [5, 46].

Previous results and our contributions. We denote by n and m the number of vertices and edges of the underlying model, and k denotes the number of different target sets. The $\widetilde{O}$ notation hides poly-log factors, e.g. $O(m(\log n)^4) = \widetilde{O}(m)$. We call a running time *near-linear* if it is linear in the input but has some additional polylogarithmic factor, e.g. $O(m(\log n)^4)$. For the reachability problem, while the graphs and games on graphs problem can be solved in linear time [28, 147], the current best-known bound for MDPs is $\widetilde{O}(m)$ [89, Theorem 12]. For the coverage and sequential reachability, an $O(k \cdot m)$ upper bound follows for graphs and games on graphs, and an $\widetilde{O}(k \cdot m)$ upper bound follows for MDPs. Our contributions are as follows:

1. *Coverage problem:* First, we present an $O(m + \sum_{i=1}^{k} \lvert T_i \rvert)$ time algorithm for

graphs; second, we present an $\Omega(k \cdot m)$ lower bound for MDPs and games on graphs, both under the BMM conjecture and the SETH. Note that for graphs our upper bound is in linear time, however, if each $|T_i|$ is constant and $k = \theta(n)$, for MDPs and games on graphs the CLB is quadratic.

2. *Sequential reachability problem:* First, we present an $O(m + \sum_{i=1}^{k} |T_i|)$ time algorithm for graphs; second, we present an $\widetilde{O}(m + \sum_{i=1}^{k} |T_i|)$ time algorithm for MDPs; and third, we present an $\Omega(k \cdot m)$ lower bound for games on graphs, both under the BMM conjecture and the SETH.

The summary of the results is presented in Table 6.1. The most interesting results are the conditional lower bounds for MDPs and game graphs for the coverage problem, the sub-quadratic algorithm for MDPs with sequential reachability objectives, and the conditional lower bound for game graphs with sequential reachability objectives.

Practical Significance. The sequential reachability and coverage problems we consider are the tasks defined in [163], where the problems have been studied for games on graphs and mentioned as future work for MDPs. The applications of these problems have been demonstrated in robotics applications. We present a complete algorithmic picture for games on graphs and MDPs, settling open questions related to games and future work mentioned in [163].

Theoretical Significance. Our results present a very interesting algorithmic picture for the natural planning questions in the fundamental models.

1. First, we establish results showing that some models are harder than others. More precisely,

   - for the reachability problem, the MDP model seems harder than graphs and games on graphs (linear-time algorithm for graphs and games on graphs, and only near-linear time algorithms are known for MDPs);
   - for the coverage problem, MDPs, and games on graphs are harder than graphs (linear-time algorithm for graphs and quadratic CLBs for MDPs and games on graphs);
   - for the sequential reachability problem, games on graphs are harder than MDPs and graphs (linear-time upper bound for graphs and sub-quadratic upper bound for MDPs, whereas quadratic CLB for games on graphs).

   In summary, we establish model-separation results with CLBs: For the coverage problem, MDPs and games on graphs are algorithmically harder than graphs; and for the sequential reachability problem, games on graphs are algorithmically harder than MDPs and graphs.

2. Second, we also establish problem-separation results. For the model of MDPs consider the different problems: Both for reachability and sequential reachability the upper bound is sub-quadratic and in contrast to the coverage problem we establish a quadratic CLB.

Further Related Work In this chapter, our focus lies on the algorithmic complexity of fundamental planning problems and we consider *explicit state-space* graphs, MDPs, and game graphs, where the complexities are polynomial. The explicit model and algorithms for it are widely considered: For example, in LTL Synthesis [60–62, 163], Probabilistic Planning [63, 157, 159, 218], Nondeterministic Planning [13, 64, 130, 183, 187], Contingent Planning [39, 186] and Verification [91]. In factored models such as STRIPS and SAS+ the complexities are higher (PSPACE-complete and NP-complete [20, 58]), and then heuristics are the focus (e.g., [136]) rather than the exact algorithmic complexity. Notable exceptions are

1. the work on parameterized complexity of planning problems (e.g., [165]),

2. conditional lower bounds based on the ETH [148] showing that certain general propositional planning problems (e.g., propositional STRIPS with negative goals (PSN)) do not admit algorithms with running times of the form $2^{|P|^c}$ for instance size $|P|$ and concrete constants $c > 0$ [7, 19],

3. conditional lower bounds based on the SETH of the form $2^{(1+\epsilon)v} \cdot poly(|P|)$ where v is the number of variables and $\epsilon > 0$ for very large subclasses PSN [19],

4. conditional lower bounds based on the graph colourability problem of the form $2^{v/2} \cdot poly(v)$,

5. conditional lower bounds based on the ETH showing that the minimum constraint removal problem, a well-studied problem in both robotic motion planning, does not admit algorithms with running times of the form $2^{o(n)}$ [122].

6.2 Coverage Problem

In this section, we consider the coverage query problem in graphs, MDPs and game graphs. We are given a starting vertex v and a coverage query. Our goal is to check if a set of player-1 strategies exist such that the resulting plays achieve the given coverage query when starting at v.

First, we present a linear-time algorithm for graphs and quadratic algorithms for MDPs and game graphs. Then we focus on the conditional lower bounds for MDPs and game graphs, which establish that there is no subquadratic algorithm for the coverage problem when one assumes the STC and OV conjectures.

6.2.1 Algorithms

The results below present the upper bound for graphs, MDPs and game graphs of the second row of Table 6.1.

Coverage Problem in Graphs. For the coverage problem in graphs we are given a graph $G = (V, E)$, a coverage query $Coverage(T_1, \ldots, T_k)$ and a start vertex $s \in V$. The algorithmic problem is to find out if starting from an initial vertex v the reachability, i.e., Reach(T_i), can be achieved for all $1 \leq i \leq k$. The algorithmic solution is as follows: Initially, mark each $v \in T_i$ for $1 \leq i \leq k$ with i. Compute the BFS tree starting from s and check if all the targets are contained in the resulting BFS tree. This instantly gives an algorithm with running time $O(m + \sum_{i=0}^{k} |T_i|)$. Note that the running time is linear and thus we cannot hope to find any quadratic lower bounds.

Coverage Problem in MDPs and game graphs. We determine in MDPs and game graphs whether there exists a set of strategies for a given coverage query with k reachability objectives and start vertex v, by applying the reachability algorithm of the respective model k times, i.e., once for each of the target sets. This yields a solution in $\widetilde{O}(km)$ time for MDPs and $O(km)$ time for game graphs respectively. Notice that for $k \in \Theta(n)$ the running time is quadratic in the input size.

6.2.2 Conditional Lower Bounds

We present conditional lower bounds for the coverage problem in MDPs and game graphs (i.e., the CLBs of the second row of Table 6.1). For MDPs and game graphs the conditional lower bounds complement the quadratic algorithms from the previous subsection. Notice that we cannot provide a quadratic lower bound for graphs as a linear-time algorithm exists. The conditional lower bounds are due to reductions from *OV* and *triangle detection*.

MDPs.

We present the following conditional lower bounds for MDPs:

Theorem 6.2.1. *For all $\epsilon > 0$, checking if a vertex has a set of a.s. winning strategies for the coverage problem in MDPs does not admit:*

1. *an $O(m^{2-\epsilon})$ algorithm under Conjecture 2.7.5,*
2. *an $O((k \cdot m)^{1-\epsilon})$ algorithm under Conjecture 2.7.5,*
3. *a combinatorial $O(n^{3-\epsilon})$ algorithm under Conjecture 2.7.3 and*
4. *a combinatorial $O((k \cdot n^2)^{1-\epsilon})$ algorithm under Conjecture 2.7.3.*

Using the OV-Conjecture. Below we prove the results 1–2 of Theorem 6.2.1. We reduce the OV problem to Coverage in MDPs. By applying Conjecture 2.7.5 we infer the result.

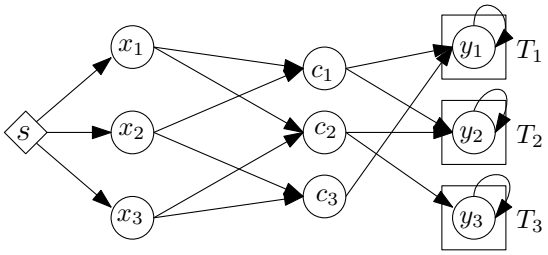


Figure 6.1: An example reduction from OV to Coverage in MDPs.

Reduction 6.2.2. *Given two sets S_1, S_2 of d-dimensional vectors, we build the MDP P as follows.*

- *The vertices V of the MDP are given by a start vertex s, sets of vertices S_1 and S_2 representing the sets of vectors and vertices $C = \{c_i \mid 1 \leq i \leq d\}$ representing the coordinates of the vectors in the OVC instance.*
- *The edges E of P are defined as follows: The start vertex s has an edge to every vertex of S_1. Furthermore for each $x_i \in S_1$ there is an edge to $c_j \in C$ iff $x_i[j] = 1$ and for each $y_i \in S_2$ there is an edge from $c_j \in S_2$ to y_i iff $y_i[j] = 1$. Also, the y_i have self-loops so that every vertex has an outgoing edge.*
- *The set of vertices is partitioned into player-1 vertices $V_1 = S_1 \cup C \cup S_2$ and random vertices $V_R = \{s\}$.*

Example 6.2.3 (Example: Reduction from OV to Coverage)**.** *Let the OV instance be $S_1 = \{(1,1,0),(1,0,1),(0,1,1)\}, S_2 = \{(1,0,1),(1,1,0),(0,1,0)\}$. Notice that the second vector in S_1 and the third vector in S_2 are orthogonal. Due to the fact that s is a random vertex, there is a nonzero probability that x_2 is the successor. There is no path from x_2 to y_3. As $T_3 = \{y_3\}$, there is no a.s. winning strategy from s for the given instance of coverage. We illustrate the example of the reduction in Figure 6.1.*

Notice that for orthogonal vectors x_i and y_j we have that for each $c_\ell \in C$ either x_i is not connected to c_ℓ or y_j is not connected to c_ℓ. Thus there is no path from x_i to y_i. Starting from s there is a non-zero probability to end in x_i and thus also a non-zero probability to fail reaching the target set $T_j = \{y_j\}$ no matter of player 1's strategy σ_j.

Lemma 6.2.4. *Let $P = (V, E, \langle V_1, V_R\rangle, \delta)$ be the MDP given by Reduction 6.2.2 and $T_i = \{y_i\}$ for $1 \leq i \leq N$. There exist orthogonal vectors $x \in S_1$, $y \in S_2$ iff s is not winning for $Coverage(\{T_i \mid 1 \leq i \leq N\})$.*

Proof. The MDP P is constructed in such a way that there is no path between vertex x_i and y_j iff the corresponding vectors are orthogonal in the OV instance: If x_i is orthogonal to y_j, the outgoing edges lead to no vertex which has an incoming edge to y_j as either $x_i[k] = 0$ or $y_j[k] = 0$. On the other hand, if there is no path

from x_i to y_j we again have by the construction of the underlying graph that for all $1 \leq k \leq d$: $x_i[k] = 0$ or $y_j[k] = 0$. This is the definition of orthogonality for x_i and y_j. When starting from s the token is randomly moved to one of the vertices x_i and thus player 1 can reach each y_j almost surely from s iff it can reach each y_j from each x_i. Thus, we have that there is an a.s. winning player 1 strategy for Reach(T_i) iff y_i has. Hence, S_2 has no orthogonal vector in S_1 iff each Reach(T_i) has an a.s. winning player 1 strategy. □

The MDP P has only $O(N)$ many vertices and Reduction 6.2.2 can be performed in $O(N \cdot d)$ time (recall that $d = \omega(\log N)$). The number of edges m is $O(N \cdot d)$ and the number of target sets $k \in \theta(N)$. Thus the points 1–2 of Theorem 6.2.1 follow.

Using the ST-conjecture. Towards the results 3–4 in Theorem 6.2.1 we reduce the triangle detection problem to Coverage problem in MDPs. By applying Conjecture 2.7.3 we infer the result.

Reduction 6.2.5. *Given an instance of triangle detection, i.e., a graph $G = (V, E)$, we build the following MDP $P = (V', E', \langle V_1', V_R' \rangle, \delta)$.*

- *The vertices V' are given as four copies V_1, V_2, V_3, V_4 of V and a start vertex s.*
- *The edges E' of P are defined as follows: There is an edge from s to every $v_{1i} \in V_1$ for $i = 1 \dots n$. In addition for $1 \leq j \leq 4$ there is an edge from v_{ji} to $v_{(j+1)k}$ iff $(v_i, v_k) \in E$. Finally, v_{4i} for $i = 1 \dots n$ has a self-loop.*
- *The set of vertices V' is partitioned into player-1 vertices $V_1' = \emptyset$ and random vertices $V_R' = \{s\} \cup V_1 \cup V_2 \cup V_3 \cup V_4$.*

Notice that all the vertices of the constructed MDP are random vertices.

Example 6.2.6 (Reducing triangle detection to Coverage.)**.** *Let G be the graph given in Figure 6.2. We construct the MDP P as in Reduction 6.2.5. Notice that G has the triangle (v_1, v_2, v_3) and the constructed MDP P has a nonzero chance to take the path marked by the fat edges that correspond to this triangle, i.e., player-1 does not have a winning strategy from s for the coverage objective given in the reduction because he cannot satisfy T_1. The example is illustrated in Figure 6.2.*

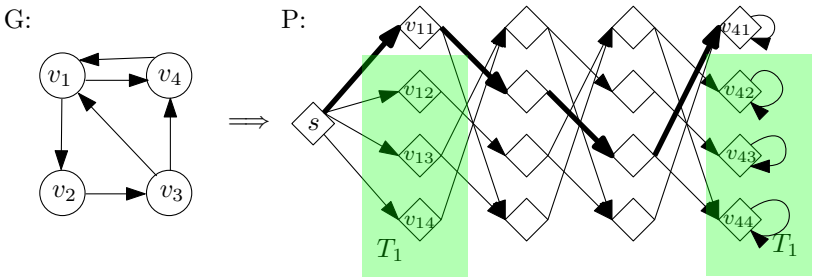


Figure 6.2: Reduction from Triangle to Coverage

Lemma 6.2.7. *Let P be the MDP given by Reduction 6.2.5 when applied to a graph G and let $T_i = V_1 \setminus \{v_{1i}\} \cup V_4 \setminus \{v_{4i}\}$ for $i = 1 \dots n$ be target sets. The graph G has a triangle iff s is not winning for $Coverage(\{T_i \mid 1 \leq i \leq N\})$ in P.*

Proof. First, s is not winning for $Coverage(T_1, \dots, T_N)$ iff there is a T_i such that player-1 has no of a.s. winning strategies from s for Reach(T_i). Second, there is a triangle in the graph G iff there is a path from some vertex v_{1i} in the first copy of G to the same vertex in the fourth copy of G, v_{4i}. Finally, notice that player 1 does not control any vertex and thus the strategy of player 1 does not matter and each possible path is played with non-zero probability. If G has a triangle containing vertex v_i then the corresponding play from v_{1i} to v_{4i} has non-zero probability and is not in Reach(T_i). That is, s is not winning for and thus not winning for $Coverage(\{T_i \mid 1 \leq i \leq N\})$. Now assume that s is not winning for $Coverage(\{T_i \mid 1 \leq i \leq N\})$ and thus not winning for Reach(T_i). Then there is a path from v_{1i} to v_{4i} and thus a triangle in G. □

Moreover, the size and the construction time of the MDP P are linear in the size of the original graph G and we have $k = \theta(n)$ target sets. Thus 3–4 of Theorem 6.2.1 follow.

Game Graphs.

Next, we describe how the results for MDPs can be extended to game graphs. We prove the following theorem which states multiple specific lower bounds for checking if a vertex has a set of winning strategies for a coverage query.

Theorem 6.2.8. *For all $\epsilon > 0$, checking if a vertex has a set of winning strategies for a coverage query in game graphs does not admit:*

1. *an $O(m^{2-\epsilon})$ algorithm under Conjecture 2.7.5,*
2. *an $O((k \cdot m)^{1-\epsilon})$ algorithm under Conjecture 2.7.5,*
3. *a combinatorial $O(n^{3-\epsilon})$ algorithm under Conjecture 2.7.3 and*
4. *a combinatorial $O((k \cdot n^2)^{1-\epsilon})$ algorithm under Conjecture 2.7.3.*

Using the OV-Conjecture. Below we prove the results 1–2 of Theorem 6.2.8. We reduce the OV problem to Coverage in game graphs. By applying Conjecture 2.7.5 we infer the result. In Reduction 6.2.9 we change the random starting vertex of Reduction 6.2.2 to a player-2 vertex. The rest of the reduction stays the same. The proof then proceeds as before with the adversary now overtaking the role of the random choices.

Reduction 6.2.9. *Given two sets S_1, S_2 of d-dimensional vectors, we build the following game graph $\Gamma = (V, E, \langle V_1, V_2 \rangle)$.*

- *The vertices V and edges E are defined as before in Reduction 6.2.2*
- *The set of vertices is now partitioned into player-1 vertices $V_1 = S_1 \cup C \cup S_2$ and player-2 vertices $V_2 = \{s\}$.*

Lemma 6.2.10. *Let Γ be the game graph given by Reduction 6.2.9 with a coverage query $Coverage(\{T_i \mid 1 \leq i \leq n\})$ where $T_i = \{y_i\}$ for $i = 1 \dots N$. There exist orthogonal vectors $x \in S_1$, $y \in S_2$ iff there is no set of winning strategies from start vertex s for the coverage query.*

The game graph Γ has only $O(N)$ many vertices and Reduction 6.2.9 can be performed in $O(N \cdot d)$ time (recall that $d = \omega(\log N)$). The number of edges m is $O(N \cdot d)$ and the number of target sets $k \in \theta(N)$. Thus the points 1–2 in Theorem 6.2.8 follow.

Using the STC conjecture. Below we prove the results 3–4 in Theorem 6.2.8. We reduce the triangle detection problem to Coverage in game graphs. By applying Conjecture 2.7.3 we infer the result. In Reduction 6.2.11 we change the random vertices of Reduction 6.2.5 to player-2 vertices. Notice that the resulting game graph consists of only player-2 vertices. Again, if there is a path starting from s which violates a reachability objectives in the given coverage query then player 2 wins. As the reachability objectives are defined such that they rule out the triangles of the reduction, player 1 only wins iff there is no such path, i.e., there is no triangle in the original graph.

Reduction 6.2.11. *Given an instance of triangle detection, i.e., a graph $G = (V, E)$, we build the following game graph $\Gamma = (V', E', \langle V_1', V_2' \rangle)$.*

- *The vertices V' and Edges E' are the same as in Reduction 6.2.5.*
- *The set of vertices V' is partitioned into player-1 vertices $V_1' = \emptyset$ and player-2 vertices $V_2' = \{s\} \cup V_1 \cup V_2 \cup V_3 \cup V_4$.*

Lemma 6.2.12. *Let Γ be the game graph given by Reduction 6.2.11 when applied to a graph G and let $T_i = V_1 \setminus \{v_{1i}\} \cup V_4 \setminus \{v_{4i}\}$ for $i = 1 \dots n$. The graph G has a triangle iff s is winning for $Coverage(\{T_i \mid 1 \leq i \leq n\})$ in Γ.*

Moreover, the size and the construction time of game graph Γ are linear in the size of the original graph G and we have $k = \theta(n)$ target sets. Thus 3–4 in Theorem 6.2.8 follow.

6.3 AllCoverage Problem

In this section, we consider the AllCoverage problem. First, we present simple algorithms for all models based on the standard reachability problems of the models. Notice that the respective algorithms can also be used to solve the corresponding Coverage Problem. Then we present a conditional lower bound for graphs which

establishes that the existing algorithm cannot be polynomially improved under the STC and OV conjectures.

6.3.1 Algorithms

We present quadratic algorithms for MDPs, games and graphs. The results present the upper bounds for graphs, MDPs and Games third row of Table 6.1.

Given the query $Coverage(\{T_i \mid 1 \leq i \leq k\})$ for graphs, MDPs and game graphs, we propose an algorithm which first solves the k reachability objectives using the basic results detailed in Section 2.5. Notice that the result of the algorithms for solving the reachability objective is a set of vertices that have a strategy to achieve the objective. Then we take the intersection of the resulting sets. (1) For graphs using BFS which is in $O(m)$ time we obtain an $O(k \cdot m)$ time algorithm. (2) For game graphs, using the $O(m)$-time attractor computation, we have an $O(k \cdot m)$ time algorithm. (3) For MDPs, the MEC-decomposition followed by k many $O(m)$-time almost-sure reachability computation, gives an $O(k \cdot m + \text{MEC})$ time algorithm.

6.3.2 Conditional Lower Bounds

In this section, we present conditional lower bounds for the AllCoverage problem in graphs (i.e., the CLBs of the third row of Table 6.1). For MDPs and game graphs the conditional lower bounds follow from Section 6.2 because the Coverage problem can be trivially reduced to the AllCoverage problem, i.e., once we have computed all the vertices that can reach a target it is easy to check whether a specific vertex can reach that target. The conditional lower bounds are due to reductions from OV and the triangle detection problem.

Theorem 6.3.1. *For all $\epsilon > 0$, computing the solution of the AllCoverage problem in graphs does not admit*

1. *an $O(m^{2-\epsilon})$ algorithm under Conjecture 2.7.5,*
2. *an $O((k \cdot m)^{1-\epsilon})$ algorithm under Conjecture 2.7.5,*
3. *a combinatorial $O(n^{3-\epsilon})$ algorithm under Conjecture 2.7.3 and*
4. *a combinatorial $O((k \cdot n^2)^{1-\epsilon})$ algorithm under Conjecture 2.7.3.*

Using the OV-Conjecture. In this section we prove the results 1–2 in Theorem 6.3.1. We reduce the OV problem to the AllCoverage problem in graphs. By applying Conjecture 2.7.5 we infer the result.

Reduction 6.3.2. *Given two sets S_1, S_2 of d-dimensional vectors, we build the graph G as follows.*

- *The construction of the graph is the same as in Reduction 6.2.2 except the we do not have a vertex s.*

Example 6.3.3 (Reducing AllCoverage to OV)**.** *Let the instance of OV be given by* $S_1 = \{(1,1,0),(1,0,1),(0,1,1)\}, S_2 = \{(1,0,1),(1,1,0),(0,1,0)\}$*. Notice that the second vector in* S_1 *and the third vector in* S_2 *are orthogonal. We construct* G *with Reduction 6.3.2. There is no path from* x_2 *to* y_3*. As* $T_3 = \{y_3\}$*,* x_2 *is not in the winning set of* $Coverage(\{T_1, T_2, T_3\})$ *The reduction is illustrated in Figure 6.3.*

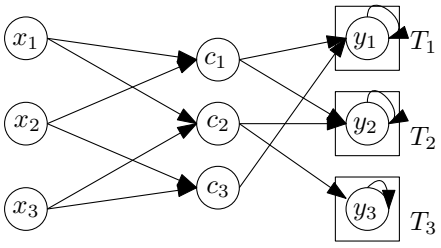


Figure 6.3: Reduction from OV to AllCoverage

Lemma 6.3.4. *Let* $G = (V, E)$ *be the graph given by Reduction 6.3.2 with target sets* $\mathcal{T} = \{T_i \mid T_i = \{y_i\}$ *for* $i = 1 \dots N\}$*. A vector* $x_i \in S_1$ *is orthogonal to some vector in* S_2 *iff the vertex* x_i *is not in the winning set of* $Coverage(\mathcal{T})$*.*

Proof. The graph P is constructed in such a way that there is no path between vertex x_i and y_j iff the corresponding vectors are orthogonal in the OV instance: If x_i is orthogonal to y_j, the outgoing edges lead to no vertex which has an incoming edge to y_j as either $x_i[k] = 0$ or $y_j[k] = 0$. On the other hand, if there is no path from x_i to y_j we again have by the construction of the underlying graph that for all $1 \leq k \leq d$: $x_i[k] = 0$ or $y_j[k] = 0$. This is the definition of orthogonality for x_i and y_j. Thus, x_i is in the winning set of $Coverage(\mathcal{T})$ iff x_i is orthogonal to some vector in S_2. □

Notice that we solve the given instance of *OV* with our reduction as we compute all vectors in S_1 which are orthogonal to some vector in S_2. The Graph G has only $O(N)$ many vertices and Reduction 6.2.2 can be performed in $O(N \cdot d)$ time (recall that $d = \omega(\log N)$). The number of edges m is $O(N \cdot d)$ and the number of target sets $k \in \theta(N)$. Thus the points 1–2 in Theorem 6.3.1 follow.

Using the ST-Conjecture. Below we prove 3–4 in Theorem 6.3.1. We reduce the triangle detection problem to the AllCoverage problem in graphs. By applying Conjecture 2.7.3 we infer the result.

Reduction 6.3.5. *Given an instance of triangle detection, i.e., a graph* $G = (V, E)$*, we build the following graph* $G = (V', E')$*. The vertices and edges are the same as in Reduction 6.2.5 except that we have player-1 vertices instead of random vertices and there is no start vertex* s*.*

Example 6.3.6 (Reducing Triangle to AllCoverage)**.** *Consider* G *in Figure 6.4. Notice that* G *has the triangle* (v_1, v_2, v_3) *and there is a path from* v_{11} *to* v_{41} *in the constructed*

MDP P illustrated by the strong edges corresponding to this triangle. The winning set of $Coverage(T_1, T_2, T_3, T_4)$ contains v_{11} by using the strategy which uses the path from v_{11} to v_{41}: First we achieve trivially, Reach(T_2), Reach(T_3), Reach(T_4) with this strategy by starting from v_{11}. Then Reach(T_1) is achieved by arriving at v_{41}.

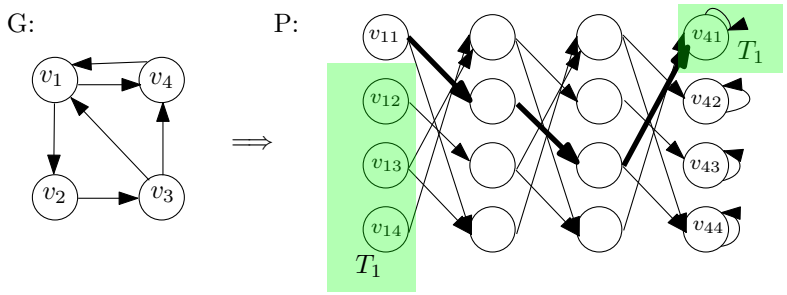


Figure 6.4: Reduction from Triangle to AllCoverage

Lemma 6.3.7. *Let G be the graph given by Reduction 6.3.5 with n target set $T_i = V_1 \setminus \{v_{1i}\} \cup \{v_{4i}\}$ for $i = 1 \dots n$. A graph G has a triangle with vertex v_i iff the vertex v_{1i} is in the winning set of the query $Coverage(T_1, \dots, T_n)$.*

Proof. Notice that there is a triangle in the graph G iff there is a path from some vertex v_{1i} in the first copy of G to the same vertex in the fourth copy of G, v_{4i}. Also, a path σ starting in v_{1i} for $1 \leq i \leq n$ is a viable strategy in the Coverage query for all Reach(T_i) $1 \leq i \leq n$ objectives iff it is able to visit v_{4i}: By definition, T_j includes v_{1i} for $1 \leq j \leq n$ and $j \neq i$. Thus σ achieves all Reach(T_j) for $j \neq i$. To achieve Reach(T_i), there must be a path from v_{1i} to v_{4i}. Thus, there is a triangle with vertex v_i iff v_{1i} is in the winning set of the query $Coverage(\mathcal{T})$. □

Note that we solve the given instance of the *triangle detection* problem if we know all vertices which are in triangles. Moreover, the size and the construction time of the MDP P are linear in the size of the original graph G and we have $k = \theta(n)$ target sets. Thus 3–4 in Theorem 6.3.1 follow.

6.4 Sequential Reachability Problem

We consider the sequential reachability problem in all models. In contrast to the quadratic CLB for the coverage problem, quite surprisingly there is a subquadratic algorithm for MDPs. We first present an algorithm for graphs and then build upon that to present the algorithm for MDPs. For games, we present a quadratic algorithm and a quadratic CLB.

6.4.1 Algorithms

The results below present the upper bounds of the fourth row of Table 6.1.

Algorithm for Graphs.

Given a graph $G = (V, E)$ and the sequential reachability objective $Seq(T_1, \dots, T_k)$, we compute the strongly connected components, contract each strongly connected component to a single vertex and remove multi edges. This results in a *directed acyclic graph (DAG)*. Additionally, the vertex v' which represents an SCC C in the resulting DAG D, is in all target sets of its members, i.e., $v' \in T_i$ if there exists a vertex $u \in C$ such that $u \in T_i$ for all $1 \leq i \leq k$. Notice that this step does not change the reachability conditions of the resulting acyclic graph: Every vertex in an SCC can be reached starting from every other vertex in the same SCC. Thus, it suffices to give an algorithm for DAGs. Given a DAG $D = (V, E)$, we maintain (a) a set of unprocessed vertices S, which is initialized with V and (b) a queue Q containing the vertices which are not processed but where all successors are processed, initialized with all vertices with no outgoing edges. Notice that the queue is initially non-empty because the bottom SCCs of G are now vertices without outgoing edges in D. Additionally, for each vertex v, we maintain the values $count_v$, ℓ_v and $best_v$. The variable $count_v$ counts the number of vertices in $Out(v)$ which are not processed yet. The label ℓ_v is such that vertex v has a winning strategy for the objective $Seq(\mathcal{T}_{\ell_v})$ where $\mathcal{T}_{\ell_v} = (T_{\ell_v}, \dots, T_k)$. In other words, there is a strategy to win from v if we already visited the targets sets $\mathcal{T}_1, \dots \mathcal{T}_{\ell_v - 1}$. The variable $best_v$ is used to store the minimum label of the already processed successors of v. The algorithm proceeds as follows. While the queue Q is not empty, we take a vertex v from the queue and call ProcessVertex$(\cdot)$. The function computes the label ℓ_v of the vertex v using $best_v$ and the target sets where v is in. Then, it removes v from S and updates the variables $best_w$ and $count_w$ of its predecessors w. In particular, we set $best_w = \min(best_w, \ell_v)$ and decrement $count_w$ by one. When the queue is empty, all vertices are processed and the algorithm terminates. We show that the described algorithm for DAGs has a linear running time, i.e., $O(m + \sum_{i=1}^{n} |T_i|)$ and the details are presented in Algorithm 6.1.

Proposition 6.4.1 (Correctness)**.** *Given a DAG $D = (V, E)$ and a sequential reachability objective $Seq(\mathcal{T})$ with target sets $\mathcal{T} = \{T_1, \dots, T_k\}$, Algorithm 6.1 returns the set of all start vertices with a path for the objective $Seq(\mathcal{T})$.*

Observation 6.4.2. *The input graph has one or more vertices v with $Out(v) = \emptyset$ and thus Q is non-empty after the initialization.*

Proof. Note that there is always a vertex $v \in V$ where $Out(v) = \emptyset$ because we assumed that D is a DAG. □

The invariants below state that (a) the variables ($best_v$, $count_v$, ℓ_v) have the intended meaning, (b) Q contains all the unprocessed vertices whose successors are already processed and (c) that the queue contains vertices as long as S is not empty.

Lemma 6.4.3. *The following statements are invariants of the while loop at Line 7.*

1. $count_v = |Out(v) \cap S|$
2. $v \in Q$ *iff* $v \in S$ *and all* $Out(v) \cap S = \emptyset$.
3. *If* S *is not empty then the queue* Q *is not empty.*
4. $best_v = k + 1$ *for all* $v \in V$ *with* $Out(v) = \emptyset$.
5. *If* $v \in Q$ *then* $best_v \neq null$.
6. *If* $v \in V \setminus S$ *then* $\ell_v \neq null$.
7. $best_v = \min_{w \in Out(v) \setminus S} \ell_w$, *for all* $v \in V$ *with* $Out(v) \setminus S \neq \emptyset$.

Algorithm 6.1: Sequential Reachability in Graphs

Input: DAG $D = (V, E)$, targets $\mathcal{T} = (T_1, \ldots, T_k)$

1 $S \leftarrow V$
2 $L_v \leftarrow \{i \mid v \in T_i : i = 1 \ldots k\}$ for $v \in V$
3 $count_v \leftarrow |Out(v)|$ for $v \in V$
4 $best_v \leftarrow \begin{cases} k+1 & \text{if } Out(v) = \emptyset \\ null & \text{otherwise} \end{cases}$ for $v \in V$
5 $\ell_v \leftarrow null$ for $v \in V$
6 $Q \leftarrow \{v \mid Out(v) = \emptyset\}$
7 **while** $S \neq \emptyset$ **do**
8 $v = Q$.pop()
9 ProcessVertex(v)
10 **return** $\{v \in V \mid \ell_v = 1\}$
11 **function** *ProcessVertex*(*Vertex* v)
12 $\ell_v \leftarrow best_v$
13 **while** $\ell_v - 1 \in L_v$ **do**
14 $\ell_v \leftarrow \ell_v - 1$
15 $S \leftarrow S \setminus \{v\}$
16 **for** $w \in In(v)$ **do**
17 $best_w \leftarrow \min(best_w, \ell_v)$
18 $count_w \leftarrow count_w - 1$
19 **if** $count_w = 0$ **then**
20 Q.push(w)

Proof. 1. The counters $count_v$ are initialized as $|Out(v)|$ and S is initialized as V. Thus the claim holds when first entering the while loop.

Assume the claim holds at the beginning of the iteration where vertex u is processed. The set S is only changed in Line 15. There u is removed from the set. The counters are only changed in Line 18: All counters of vertices w with $u \in Out(w)$ are decreased by one. Consequently $count_v = |Out(v) \cap S|$ holds for all $v \in V$ also after this iteration of the loop and the claim follows.

2. In the initial phase S is set to V and Q is set to $\{v \in V \mid Out(v) = \emptyset\}$. Thus the claim holds when first entering the while loop.

Assume the claim holds at the beginning of the iteration where vertex v is processed. The set S is only changed in Line 15 where v is removed.

First consider a vertex $w \in Q \setminus \{v\}$. As w is not removed from the set S and no vertex is added to S the claim is still true for w. Now consider a vertex w that might be added during the iteration of the loop. This can only happen in Line 20 and the if conditions ensure that $w \in S$ and $Out(v) \cap S = \emptyset$ (by the previous invariant) and thus the claim also holds for the newly added vertices.

3. Due to Observation 6.4.2 the claim holds when first entering the while loop.

Assume the claim holds at the beginning of the iteration, where vertex v is processed. The vertex v is removed from S in Line 15 and if the set S is empty now, the claim follows trivially. On the other hand, if S is non-empty and Q is also non-empty the claim follows again. In the third case S is non-empty and Q is empty. Assume for contradiction that no vertex is added at line 20. By invariant (2), every vertex $v \in S$ has a successor in S as otherwise, v would be in Q. That implies that there exists a cycle which is a contradiction with D being a DAG.

4. For $v \in V$ with $Out(v) = \emptyset$ the variables $best_v$ are initialized with $k + 1$ (Line 4) and $best_v$ is only changed in Line 17 when a successor of the vertex is processed. As v has no successor, $best_v$ is not changed during the algorithm.

5. If $v \in Q$ initially, it must be due to the initialization and we have $best_v = k+1$. The claim holds when first entering the while loop. Assume the claim holds at the beginning of the iteration where v is processed. The only time we add a vertex w to Q is at Line 20. Notice that we set $best_w$ before at Line 17.

6. Initially, every vertex is in S, thus the claim holds before the first iteration of the while loop. Assume the claim holds at the beginning of the iteration where v is processed. The only time we remove a vertex from S is at Line 15, i.e., in ProcessVertex(v). Notice that we set ℓ_v in Line 12 to $best_v$ which cannot be *null* due to Lemma 6.4.3 (5).

7. Initially, V is S, and for all $v \in V$ we set ℓ_v, $best_v$ to *null*. Notice that $best_v$ with $Out(v) \neq \emptyset$ are not changed at Line 4. Thus the claim holds when the algorithm enters the loop.

   Now consider the iteration of vertex v and assume the claim is true at the beginning. The set S is only changed in Line 15 where v is removed. Let S_{old} be the set at the beginning of the iteration and $S_{new} = S_{old} \setminus \{v\}$ the updated set. Due to Lemma 6.4.3 (6) $\ell_v \neq$ *null*. For a vertex $w \in In(v)$, the value $best_w$ is updated to $\min(best_w, \ell_v)$ (Line 17) which by assumption is equal to $\min_{x \in (Out(w) \setminus S_{old}) \cup \{v\}} \ell_x = \min_{x \in (Out(w) \setminus S_{new})} \ell_x$, i.e., the equation holds. For vertices $w \notin In(v)$ both $best_w$ as well as the right hand side of the equation are unchanged. Hence, the claim holds also after the iteration.

□

From the following invariants, we obtain the correctness of our algorithm.

Lemma 6.4.4. *The following statements are invariants of the while loop at Line 7 for all $v \in V \setminus S$:*

1. *There exists a path $p_v \in Seq(\mathcal{T}_{\ell_v})$.*
2. *There exists no path $p_v \in Seq(\mathcal{T}_{\ell_v - 1})$.*

where $\mathcal{T}_{\ell_v} = \{T_{\ell_v}, \ldots, T_k\}$ or $\ell_v > k$.

Proof. As S is initialized with the set of vertices V the two statements trivially hold after the initialization.

Now consider the iteration where vertex v is processed and assume the invariants hold at the beginning of the iteration. Let $be(v) = \min_{w \in Out(v)} \ell_w$. By Lemma 6.4.3 (2) we have $Out(v) \cap S = \emptyset$ and by Lemma 6.4.3 (6) also $\ell_w \neq$ *null* for all $w \in Out(v)$. Thus by Lemma 6.4.3 (7) we have $be(v) = best_v$ and ℓ_v can be computed. The while loop in Line 13 decrements ℓ_v which is initialized to $best_v - 1$ as long has $best_v - 1 \in L_v$. L_v contains $\ell_v, \ldots, best_v - 1$ but does not contain $\ell_v - 1$.

1. We next show that there is a path p_v in $Seq(\mathcal{T}_{\ell_v})$: Let $w = be(v)$. A path for vertex w where $p_w \in Seq(\mathcal{T}_{\ell_w})$, exists by induction hypothesis. The targets $\{T_{\ell_v}, \ldots, T_{\ell_{w-1}}\}$ are visited by starting from v. The path is obtained as follows: $p_v = v, p_w$, which proves the claim.

2. We next show that there is no path p_v in $Seq(\mathcal{T}_{\ell_v - 1})$. The current vertex v is not in the set $T_{\ell_v - 1}$ and no successor w has a path p_w with $p_w \in Seq(\mathcal{T}_{\ell_v - 1})$ because $\ell_v - 1 < \ell_v \leq \ell_w$ (Lines 12–14). Thus there is also no path $p_v \in Seq(\mathcal{T}_{\ell_v - 1})$ which concludes the proof.

□

Proposition 6.4.5. *Algorithm 6.1 has running time $O(m + \sum_{i=1}^{k} |T_i|)$.*

Proof. We first argue that the initialization takes $O(m + \sum_{i=1}^{k} |T_i|)$ time. Initializing the sets L_v can be done by first initializing the set L_v as $\emptyset$ (in $O(n)$) and then iterate over all set T_i and for each $v \in T_i$ add i to L_v (in $O(\sum_{i=1}^{k} |T_i|)$). The other variables can be initialized by iterating over all vertices and for each vertex consider all outgoing edges. That is in $O(n + m) = O(m)$ time. Now consider the main part of the algorithm. In the while loop we process each vertex $v \in V$ once (recall that D is a DAG) and call the function ProcessVertex(v) at Line 9 In the function call ProcessVertex(v), we iterate over the set L_v (Lines 13–14) and all incoming edges of v (Lines 16–20). If we sum over all the vertices we obtain a running time of $O(m + \sum_{v \in V} |L_v|) = O(m + \sum_{i=1}^{n} |T_i|)$. □

Theorem 6.4.6. *Given a graph $G = (V, E)$, a sequential reachability objective and a vertex $s \in V$ we can decide whether s is winning for $Seq(\mathcal{T})$ in $O(m + \sum_{i=1}^{k} |T_i|)$ time.*

Algorithm for MDPs.

The algorithm for MDPs builds on to of the algorithm for graphs. The first difference is that instead of computing an SCC decomposition and contracting SCCs for MDPs ws compute a MEC-decomposition and contract MECs into player-1 vertices: Given an MDP $P = (V, E, \langle V_1, V_R \rangle, \delta)$ with the sequential reachability objective $Seq(T_1, \ldots, T_k)$, we compute the MEC-decomposition of the MDP. Then, each MEC M is contracted into a player-1 vertex v' without self-loops. The resulting MDP is P'. The target sets of P' are as follows: The vertex v' is in all the target sets of the corresponding vertices in M, i.e., $v' \in T_i$ if there exists a vertex $u \in M$ such that $u \in T_i$ for all $1 \leq i \leq k$. Notice that this step does not change the reachability conditions of the resulting MDP: Every vertex in the MEC can be reached almost surely starting from every other vertex in the same MEC, regardless of their type (player-1, random). Thus it suffices to give an algorithm for MEC-free MDPs.

Key Challenge. When computing a MEC-decomposition and contracting the MECs we get an MDP that may still contain cycles. Thus our MDP algorithm has to deal with cycles which are in contrast to the graph setting where we only had to deal with DAGs, i.e., in Algorithm 6.1 we maintained a Queue Q which contained all unprocessed vertices where all successors are processed. In each iteration, we process one such vertex. Notice that when the queue is the only mechanism to process the vertices, we need the fact (which we also show in Lemma 6.4.3 (3)) that there either exists a vertex where all successors have been processed or all vertices have been processed and the algorithm can terminate. When running the algorithm on MEC-free MDPs there might be a situation where the Queue Q is empty and some vertices have not been processed yet. We illustrate such a situation in Example 6.5. Thus, we need an additional mechanism to process vertices in this situation.

Example 6.4.7 (Queue empty but Graph not processed)**.** *Consider the MDP P given in Figure 6.5: Contracting the MECs of P into player-1 vertices, we obtain P'. Notice that P' still contains a cycle. Using only the queue to obtain the next vertex to process*

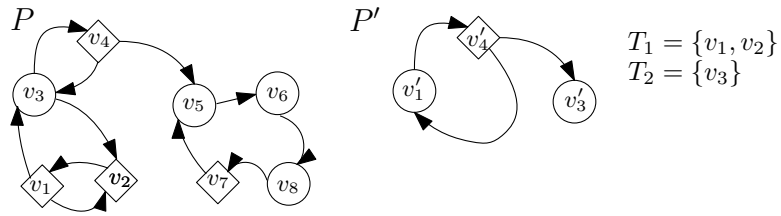


Figure 6.5: Key difficulty when computing Sequential Reachability in MDPs.

we have the following problem: After v'_3 is processed, the queue Q is empty because v'_2 has still has an unprocessed succesor, namely v'_1. Notice that v'_2 and v'_1 have not been processed yet.

Algorithm Description. The algorithm for MEC-free MDPs maintains the set of unprocessed vertices S and a queue Q, the values $count_v$, ℓ_v, and $best_v$ for each vertex v. The value $count_v$, as in Algorithm 6.1, stores the number of vertices in $Out(v)$ which are not processed yet. The label ℓ_v for v is now such that v has an *almost-sure winning strategy* for the objective $Seq(\mathcal{T}_{\ell_v})$ where $\mathcal{T}_{\ell_v} = (T_{\ell_v}, \dots, T_k)$. Random vertices might choose the worst possible successor with nonzero probability, i.e., the vertex with highest ℓ_v, whereas player-1 vertices always choose the vertex with the lowest ℓ_v. We reflect this fact as follows in the variable $best_v$: The variable $best_v$ stores the maximum (for $v \in V_R$) / minimum (for $v \in V_1$) label of the already processed successors of v. The set S is initialized with V, and, initially, all vertices with no outgoing edges are added to the queue Q. Notice that the bottom MECs of P are now vertices without outgoing edges in P' and thus Q is initially non-empty. If the queue Q is non-empty, a vertex from the queue is processed as in Algorithm 6.1. When Q is empty, the algorithm has to process a vertex where some successors are not processed yet. In that case, we consider all the random vertices for which at least one successor is processed and choose the random vertex with maximum $best_v$ to process next. We show that, as the graph has no MECs, whenever Q is empty (and S is not) there exists such a random vertex. Moreover, whenever Q is empty, all vertices in the set of unprocessed vertices S have a strategy that satisfies $Seq(\mathcal{T}_m)$ for $m = \max_{v \in V_R \cap S} best_v$: Intuitively, this is due to the fact that all vertices in S can reach a vertex v' (which is possibly different to $v = \operatorname{argmax}_{v \in V_R \cap S} best_v$) in the set of already processed vertices and in the worst case $\ell_{v'} = m$, i.e., they can satisfy $Seq(\mathcal{T}_m)$. For the vertex $v = \arg\max_{v \in V_R \cap S} best_v$ all successors w without a label are in S and are going to obtain a label ℓ_w of at most m. The current value of $best_v$ is m, i.e., v has a successor w with $\ell_w = m$. As v is in V_R the final value of $best_v$ must be m. Hence, one can process v without knowing the exact label of all the successors. We present the details in Algorithm 6.2 and prove a running time which is in $O(m \log n + \sum_{i=0}^{k} |T_i|)$. Notice that the running is near-linear time, linear up to a $\log n$ factor.

Algorithm 6.2: Sequential Reachability for MEC-free MDPs.

Input: MEC-free MDP $P = (V, E, \langle V_1, V_R \rangle, \delta)$, targets $\mathcal{T} = (T_1, \dots, T_k)$
Output: All vertices with a strategy for $Seq(\mathcal{T})$.

$S \leftarrow V$
$L_v \leftarrow \{i \mid v \in T_i : i = 1 \dots k\}$ for $v \in V$
$count_v \leftarrow |Out(v)|$ for $v \in V$
$best_v \leftarrow \begin{cases} k+1 & \text{if } Out(v) = \emptyset \\ null & \text{otherwise} \end{cases}$ for $v \in V$
$\ell_v \leftarrow null$ for $v \in V$
$Q \leftarrow \{v \mid Out(v) = \emptyset\}$
while $S \neq \emptyset$ **do**
  if $Q \neq \emptyset$ **then**
    v = $Q.pop()$
    ProcessVertex(v)
  else
    $v \leftarrow \text{argmax}_{v \in V_R \cap S}\ best_v$
    ProcessVertex(v)
return $\{v \in V \mid \ell_v = 1\}$;
function *ProcessVertex*(*Vertex* v)
  $\ell_v \leftarrow best_v$
  while $\ell_v - 1 \in L_v$ **do**
    $\ell_v \leftarrow \ell_v - 1$
  $S \leftarrow S \setminus \{v\}$
  for $w \in \{w : (w, v) \in E\}$ **do**
    if $w \in V_1$ **then**
      $best_w \leftarrow \min(best_w, \ell_v)$
    else
      $best_w \leftarrow \max(best_w, \ell_v)$
    $count_w \leftarrow count_w - 1$
    if $count_w = 0 \wedge w \in S$ **then**
      $Q.push(w)$

Proposition 6.4.8 (Correctness)**.** *Given an MDP $\boldsymbol{P}$ and a sequential reachability objective $Seq(\mathcal{T})$ with targets $\mathcal{T} = (T_1, \ldots, T_k)$, Algorithm 6.2 returns the set of all start vertices with a player-1 strategy for the objective $Seq(\mathcal{T})$.*

We next state invariants of the while loop (see Line 7) that will enable us to show the correctness of the algorithm. The invariants state that (a) the variables $best_v$ and $count_v$ have the meaning as described in the algorithm description for all $v \in V$, (b) Q contains all the unprocessed vertices whose successors are already processed, and (c) that the function argmax is well-defined whenever called, i.e., there is a random vertex where $best_v$ is not *null*.

Lemma 6.4.9. *The following statements are invariants of the while loop in Line 7.*

1. $count_v = |Out(v) \cap S|$;
2. $v \in Q$ *iff* $v \in S$ *and* $Out(v) \cap S = \emptyset$;
3. $best_v = k + 1$, *for all* $v \in V$ *with* $Out(v) = \emptyset$.
4. *If* $v \in Q$ *we have* $best_v \neq$ *null*.
5. *If* $v \in V \setminus S$ *we have* $\ell_v \neq$ *null*.
6. *For all* $v \in V$ *with* $Out(v) \setminus S \neq \emptyset$: $best_v = \begin{cases} \min_{w \in Out(v) \setminus S} \ell_w & v \in V_1 \\ \max_{w \in Out(v) \setminus S} \ell_w & v \in V_R \end{cases}$
7. *If* $S \neq \emptyset$ *and* $Q = \emptyset$ *there is a* $v \in S \cap V_R$ *such that* $best_v \neq$ *null*.

Proof. The proofs of (1) — (5) proceed as the proofs of the corresponding statements in the proof of Lemma 6.4.3.

6. Initially $S = V$ and for all $v \in V$ we set $\ell_v, best_v$ to *null*. Also, $best_v$ with $Out(v) \neq \emptyset$ are not changed at Line 4 and the claim holds when the algorithm enters the loop.

   Now consider the iteration of vertex v and assume the claim is true at the beginning. The set S is only changed in Line 19 where v is removed. Let S_{old} be the set at the beginning of the iteration and $S_{new} = S_{old} \setminus \{v\}$ the updated set. First notice that $best_v \neq$ *null* as v is either chosen by (a) as element of Q or (b) by argmax. In the former case we apply Lemma 6.4.9 (4) and in the latter case $best_v \neq$ *null* by the definition of argmax. For a vertex $w \in In(v) \cap V_1$ the value $best_w$ is updated to $\min(best_w, \ell_v)$ (Line 22) which by assumption is equal to $\min_{x \in (Out(w) \setminus S_{old}) \cup \{v\}} \ell_x = \min_{x \in (Out(w) \setminus S_{new})} \ell_x$, i.e., the equation holds. For a vertex $w \in In(v) \cap V_R$ the value $best_w$ is updated to $\max(best_w, \ell_v)$ (Line 24) which by assumption is equal to $\max_{x \in (Out(w) \setminus S_{old}) \cup \{v\}} \ell_x = \max_{x \in (Out(w) \setminus S_{new})} \ell_x$, i.e., the equation holds. For vertices $w \notin In(v)$ $best_w$ remains unchanged. Hence, the claim holds for w by the assumption that the invariant is true before the iteration.

7. Initially the statement is true as each MEC-free MDP has a vertex v with $Out(v) = \emptyset$ and thus Q is non-empty (otherwise there would be an SCC with no outgoing edge which thus would be a MEC).

   Now consider the iteration processing vertex v and assume the claim is true at the beginning and $Q = \emptyset$. Notice that $best_w$ is set for vertices as soon as one vertex in $Out(w)$ was processed. Towards a contradiction assume that all vertices $w \in S \cap V_R$ have $best_w = null$, i.e., no vertex $w \in S \cap V_R$ has a successor in $V \setminus S$. Note that S contains only the vertices which are not processed yet. Each $w \in S$ has at least one successor in S as otherwise, w would be in Q. Thus S is either empty which would make the statement trivially true or has again a bottom SCC (on the induced subgraph G_P) with more than one vertex that has no random outgoing edges. Again such an SCC would be a MEC and we obtain our desired contradiction. □

From the following invariant, we obtain the correctness of our algorithm.

Lemma 6.4.10. *The following statements are invariants of the while loop in Line 7 for all $v \in V \setminus S$:*

1. *there exists a player 1 strategy σ s.t. $\Pr_v^\sigma(Seq(\mathcal{T}_{\ell_v})) = 1$; and*
2. *there is no player 1 strategy σ s.t. $\Pr_v^\sigma(Seq(\mathcal{T}_{\ell_v - 1})) = 1$.*

where $\mathcal{T}_{\ell_v} = \{T_{\ell_v}, \dots, T_k\}$ or $\ell_v > k$.

Proof. As S is initialized as set V the two statements hold after the initialization.

Now consider the iteration where vertex v is processed and assume the invariants hold at the beginning of the iteration. Notice that we do not change ℓ'_v for any other vertex $v' \neq v$ and the invariant holds trivially for v'. We first introduce the following notation

$$be(v) = \begin{cases} \min_{w \in Out(v)} \ell_w & v \in V_1 \\ \max_{w \in Out(v)} \ell_w & v \in V_R \end{cases}$$

We distinguish the case where Q is non-empty and the case where Q is empty.

- *Case $Q \neq \emptyset$*: By Lemma 6.4.9 (2) we have $Out(v) \cap S = \emptyset$. Because we only remove vertices from S if we process them, all $w \in Out(v)$ are processed and thus $\ell_w \neq null$. Thus by Lemma 6.4.9 (6) we have $be(v) = best_v$. By the while-loop in Line 17 we have $L_v \supseteq \{\ell_v, \dots, best_v - 1\}$ but does not contain $\ell_v - 1$, i.e., $\ell_v - 1 \notin L_v$.

  (1) Thus we can easily obtain a strategy σ with $\Pr_v^\sigma(Seq(\mathcal{T}_{\ell_v})) = 1$ as follows.

  If $v \in V_1$ pick the vertex w that corresponds to $be(v)$ and then player 1 can follow the existing strategy σ' for vertex w. Because the invariant holds for w, there exists a strategy σ' such that $\Pr_w^{\sigma'}(Seq(\mathcal{T}_{be(v)})) = 1$.

If $v \in V_R$ let vertex $w \in Out(v)$ be the randomly chosen vertex. By the invariant which holds during the iteration, w has a strategy σ such that $\Pr_w^\sigma(Seq(\mathcal{T}_{be(v)})) = 1$. Combined with L_v, this is the desired strategy, i.e., $\Pr_v^\sigma(Seq(\mathcal{T}_{\ell_v})) = 1$

(2) We next show that there is no strategy for $Seq(\mathcal{T}_{\ell_v - 1})$. By Line 17 we have $v \notin T_{\ell_v - 1}$. If $v \in V_1$ no successor $w \in Out(v)$ has a strategy σ with $\Pr_w^\sigma(Seq(\mathcal{T}_{\ell_v - 1})) = 1$ as the invariant holds also for w. Thus there is also no strategy σ for v such that $\Pr_v^\sigma(Seq(\mathcal{T}_{\ell_v - 1})) = 1$. If $v \in V_R$ there is at least one successor w (because the invariant holds also for w) which has no strategy σ such that $\Pr_w^\sigma(Seq(\mathcal{T}_{\ell_v - 1})) = 1$. Consequently there is no strategy σ for v with $\Pr_v^\sigma(Seq(\mathcal{T}_{\ell_v - 1})) = 1$ as there is a non-zero chance that a vertex w is picked that, by the fact that the invariant holds at the current iteration, cannot reach a node in $T_{l_v - 1}$.

- *Case* $Q = \emptyset$: Due to Lemma 6.4.9 (7) there is at least one vertex in $V_R \cap S$ such that $best_v \neq null$. Let $best_{\max} = \max_{v \in V_R \cap S} best_v$.

  (1) As we have no MEC (in S), there is a strategy σ, so that the play almost surely leaves S by using one of the outgoing edges of a random node: Note that for all random nodes between S and $V \setminus S$ we have a strategy which achieves at least $Seq(\mathcal{T}_{best_{\max}})$. The strategy σ can be arbitrary, except that for a player-1 vertex $x \in S$ with an edge (x, y) where $y \in V \setminus S$ we choose $\sigma(x) \in S$ (which must exist as x would be in Q otherwise). As there are no MECs (in S) the strategy σ_1 will eventually lead to a vertex in $V \setminus S$ using a random node. This implies that from each vertex in S player 1 has a strategy to reach a vertex in $V \setminus S$ coming from a random vertex. Because the invariant holds at the current iteration each successor of a random vertex v' where $best_{v'} \neq null$ has a strategy to satisfy $Seq(\mathcal{T}_{best_{\max}})$. Thus it follows that from each vertex in S player 1 has a strategy to satisfy $Seq(\mathcal{T}_{best_{\max}})$. Now consider the random vertex v that was chosen by the algorithm as $\operatorname{argmax}_{v \in V_R \cap S} best_v$. Because v' is a random vertex, all successors have a strategy to satisfy $Seq(\mathcal{T}_{best_{\max}})$ almost-surely. As L_v contains $\ell_v, \ldots, best_v - 1$ but does not contain $\ell_v - 1$ we obtain a strategy σ with $\Pr_v^\sigma(Seq(\mathcal{T}_{\ell_v})) = 1$.

  (2) By the choice of v there is also a successor (that is chosen with non-zero probability) that, by assumption, has no strategy for $Seq(\mathcal{T}_{best_{\max} - 1})$ and, moreover, L_v does not contain $\ell_v - 1$. Thus, when starting in v each strategy will fail to satisfy $Seq(\mathcal{T}_{best_{\max} - 1})$ with non-zero probability, i.e., there is no strategy σ for $\Pr_v^\sigma(Seq(\mathcal{T}_{\ell_v - 1})) = 1$.

□

Proposition 6.4.11 (Running Time). *Algorithm 6.2 runs in* $O(m \log n + \sum_{i=0}^{k} |T_i|)$ *time.*

Proof. Initializing the algorithm takes $O(m + \sum_{i=0}^{k} |T_i|)$ time and calling the function ProcessVertex(v) takes time $O(|In(v)| + |L_v|)$ (cf. proof of Proposition 6.4.5).

Consider the main while-loop at Line 7 where every vertex is processed once (recall that either Q is nonempty or there exists a $v \in S \cap V_R$ such that $best_v \neq null$ due to Lemma 6.4.9). The costly operations are the calls to the ProcessVertex(·) function and the evaluation of the argmax function. Summing up over all vertices we obtain a $O(m + \sum_{i=0}^{k} |T_i|)$ bound for the calls to ProcessVertex(·). To compute argmax efficiently we have to maintain a priority queue containing all not yet processed random vertices. As we have $O(m)$ updates this costs only $O(m \log n)$ for one of the standard implementations of priority queues. Summing up this yields a $O(m \log n + \sum_{i=0}^{k} |T_i|)$ running time for Algorithm 6.2. □

By considering also the time MEC for the MEC decomposition we obtain the desired bound and the following theorem.

Theorem 6.4.12. *Given an MDP $\boldsymbol{P}$, a start vertex s and a sequential reachability objective $Seq(\mathcal{T})$, we can compute whether there is a player-1 strategy σ_1 at s for $Seq(\mathcal{T})$ in $O(\text{MEC} + m \log n + \sum_{i=0}^{k} |T_i|)$ time.*

Algorithm for Games.

Given a game graph Γ with sequential reachability objectives $Seq(\mathcal{T})$ where $\mathcal{T} = (T_1, \ldots, T_k)$, the basic algorithm (stated as Algorithm 6.3) performs k player-1 attractor computations. It starts with computing the attractor $S_k = attr_1(T_k, \Gamma)$ of T_k, and then iteratively computes the sets $S_\ell = attr_1(S_{\ell+1} \cap T_\ell, \Gamma)$ for $1 \leq \ell < k$, and finally returns the set S_1 as the start vertices from which player 1 can reach all the target sets in the given order. This gives an $O(k \cdot m)$-time algorithm. Note that for $k = \Theta(n)$ the running time is quadratic in the input size.

Algorithm 6.3: Sequential Reachability for Games.

Input: Game graph $\Gamma = ((V, E), \langle V_1, V_2 \rangle)$ and target sets $\mathcal{T} = (T_1, \ldots, T_k)$

1 $S_{k+1} \leftarrow V$
2 $\ell \leftarrow k$
3 **while** $\ell > 0$ **do**
4 $\quad S_\ell \leftarrow attr_1(T_\ell \cap S_{\ell+1}, \Gamma)$
5 $\quad \ell \leftarrow \ell - 1$
6 **return** S_1

6.4.2 Conditional Lower Bounds

We present CLBs for game graphs based on the conjectures STC and OVC which establish the CLBs for the fourth row of Table 6.1. Notice that we cannot provide conditional lower bounds for graphs and MDPs as linear time algorithms for these two models exist.

Theorem 6.4.13. *For all $\epsilon > 0$, checking if a vertex has a winning strategy for the sequential reachability problem in game graphs does not admit*

1. *an $O(m^{2-\epsilon})$ algorithm under Conjecture 2.7.5,*
2. *an $O((k \cdot m)^{1-\epsilon})$ algorithm under Conjecture 2.7.5,*
3. *a combinatorial $O(n^{3-\epsilon})$ algorithm under Conjecture 2.7.3 and*
4. *a combinatorial $O((k \cdot n^2)^{1-\epsilon})$ algorithm under Conjecture 2.7.3.*

Using the OV-Conjecture. Below we prove (1–2) of Theorem 6.4.13 by reducing the OV problem to the sequential reachability problem in game graphs. The reduction is an extension of Reduction 6.2.2, where we (a) produce a player-2 vertex instead of a random vertex and (b) also every vertex of S_2 has an edge back to s.

Example 6.4.14. *Let the OV instange be given by $S_1 = \{(1,1,0),(1,0,1),(0,1,1)\}$, $S_2 = \{(1,0,1),(1,1,0),(0,1,0)\}$. Notice that the second vector in S_1 and the third vector in S_2 are orthogonal. Due to the fact that s is a player-2 vertex, it can choose x_2 as the successor. There is no path from x_2 to y_3. As $T_3 = \{y_3\}$, there is no winning strategy for player 1 from s for the given sequential reachability objective. We illustrate the reduction in Figure 6.6.*

Reduction 6.4.15. *Given two sets S_1, S_2 of d-dimensional vectors, we build the following game graph Γ.*

- *The vertices V of the game graph are given by a start vertex s, sets of vertices S_1 and S_2 representing the sets of vectors and vertices $C = \{c_i \mid 1 \leq i \leq d\}$ representing the coordinates of the vectors in the OVC instance.*
- *The edges E of Γ are defined as follows: the start vertex s has an edge to every vertex of S_1 and every vertex of S_2 has an edge back to s; furthermore for each $x_i \in S_1$ there is an edge to $c_j \in C$ iff $x_i[j] = 1$ and for each $y_i \in S_2$ there is an edge from $c_j \in S_2$ to y iff $y_i[j] = 1$.*
- *The set of vertices is partitioned into player-1 vertices $V_1 = S_1 \cup C \cup S_2$ and player-2 vertices $V_2 = \{s\}$.*

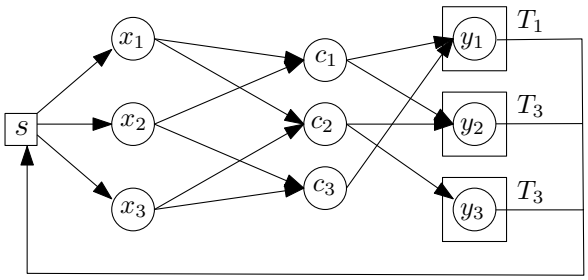


Figure 6.6: Reduction from OV to Sequential Reachability.

Lemma 6.4.16. *Let Γ be the game graph given by Reduction 6.4.15 with a sequential objective $Seq(\mathcal{T})$ where $\mathcal{T} = (T_1, \dots, T_k)$ and $T_i = \{y_i\}$ for $i = 1 \dots N$. There exist orthogonal vectors $x_i \in S_1, y_j \in S_2$ iff s has no player-1 strategy σ_1 to ensure winning for the objective $Seq(\mathcal{T})$.*

Proof. Notice that the game graph Γ is constructed in such a way that there is no path between x_i and y_j iff they are orthogonal in the OV instance. Notice that each play starting at s revisits s every four steps and if there is no path between x_i and y_j then player 2 can disrupt player 1 from visiting a target T_j by moving the token to x_i whenever the token is in s. However, if there is no such x_i and y_j, player 2 cannot disrupt player 1 from s because no matter which vertex x_i player 2 chooses, player 1 has a strategy to reach the next target set. If s has no player-1 strategy σ_1 to ensure winning for the objective $Seq(\mathcal{T})$ there must be a target player 1 cannot reach. This must be due to the fact that there is no path between some x_i and y_j and player 2 always chooses x_i. □

The number of vertices in Γ, constructed by Reduction 6.2.2 is $O(N)$ and the construction can be performed in $O(N \log N)$ time (recall that $(d = \omega(\log N))$). The number of edges m is $O(N \log N)$ and the number of target sets $k \in \theta(N) = \theta(m/\log N)$. Thus (1–2) in Theorem 6.4.13 follow.

Using the ST-Conjecture. In this section we prove the results 3–4 in Theorem 6.4.13. We reduce the triangle detection problem to the sequential reachability problem in game graphs. The reduction extends Reduction 6.2.5, where we (a) produce player-2 vertices instead of random vertices and (b) every vertex in the fourth copy has an edge back to s.

Reduction 6.4.17. *Given an instance of triangle detection, i.e., a graph $G = (V, E)$, we build the following game graph $\Gamma = (V', E', \langle V_1', V_2' \rangle)$.*

- *The vertices V' are given as four copies V_1, V_2, V_3, V_4 of V and a start vertex s.*
- *The edges E' are defined as follows: There is an edge from s to every $v_{1i} \in V_1$ where $i = 1 \dots n$. In addition for $1 \leq j \leq 3$ there is an edge from v_{ji} to $v_{(j+1)k}$ iff $(v_i, v_k) \in E$. Furthermore there are edges from every $v_{4i} \in V_4$ to the start vertex s.*
- *The set of vertices V' is partitioned into player-1 vertices $V_1' = \emptyset$ and player-2 vertices $V_2' = \{s\} \cup V_1 \cup V_2 \cup V_3 \cup V_4$.*

Example 6.4.18 (Reduction triangle detection to sequential reach. in games)**.** *Consider the graph G given in Figure 6.7. The vertices of G are player-2 vertices in Γ and the graph is copied four times. The edges of Γ go to the same target but to next copy of the graph. Notice that G has the triangle (v_1, v_2, v_3) and the constructed game graph Γ enables player-2 to take the path marked by the fat edges, i.e., player-1 does not have a winning strategy from s for the sequential reachability objective given in the reduction because he cannot satisfy T_1. We illustrate the example of the reduction in Figure 6.7.*

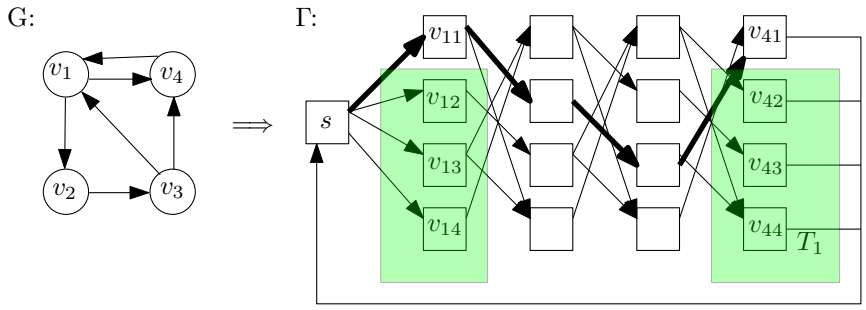


Figure 6.7: Reduction from Triangle to Sequential Reachability

Lemma 6.4.19. *Let Γ' be the game graphs given by Reduction 6.4.17 with $Seq(\mathcal{T})$ as follows: $\mathcal{T} = (T_1, T_2, \dots, T_k)$ where $T_i = V_1 \setminus \{v_{1i}\} \cup V_4 \setminus \{v_{4i}\}$ for $i = 1 \dots k$. The graph G has a triangle iff there is no strategy σ_1 to ensure winning for the objective $Seq(\mathcal{T})$ from start vertex s.*

Proof. For the correctness of the reduction notice that there is a triangle in the graph G iff there is a path from some vertex v_{1i} in the first copy of G to the same vertex in the fourth copy of G, v_{4i} in P. Player 2 then has a strategy to always visit only v_{1i} from the first copy and only v_{4i} from the fourth copy which prevents player 1 from visiting target T_i. □

The size and the construction time of graph Γ, given by Reduction 6.2.5, are linear in the size of the original graph G and we have $k = \Theta(n)$ target sets. Thus (3–4) in Theorem 6.4.13 follow.

6.5 Discussion and Conclusion

In this chapter, we presented lower bound results for planning objectives in an explicit state space. We next discuss implications from these results for the same planning objectives in factored models and then end this chapter with concluding remarks.

6.5.1 Implications for factored models

Here we relate our results for the explicit state space to factored models, like STRIPS [200]. For the reachability problem different conditional lower bounds were established in [7, 19]. In the following, we discuss to which extent our conditional lower bounds provide lower bounds for the corresponding problems in the factored models. We use the AllCoverage problem on graphs as an example but similar arguments apply to the other planning problems as well. A planning instance of the AllCoverage problem for graphs in a factored model is given by variables V, the domain of the variables D, the actions A and a set of conditions defining the target

sets $s_{G_1}, \ldots, s_{G_k}$. A state is a function that specifies a value in D to every variable in V. The planner can go from one state to another by applying the actions defined in the function A. A is a mapping from an input state to an output state, i.e., the possible transitions between states are defined by the actions. The goal of the planner is to output all states which can reach all sets s_{G_i} $(1 \leq i \leq k)$ using the actions described in A. We next investigate how our graph based lower bounds can be interpreted in the factored model. To obtain lower bounds similar to Theorem 6.3.1 we aim to encode the graphs computed by our reductions (see e.g. Reduction 6.3.2 and Reduction 6.3.5) in the factored model. A naive encoding that simply numbers the vertices and then uses a binary encoding of these numbers can represent n states with $v = \log_2 n$ variables. This encoding would give lower bounds w.r.t. the number of variables v (and thus the state space) that exclude $O(k \cdot 2^{(1-\epsilon)v} \cdot poly(v))$ algorithms for the AllCoverage problem in the factored planning model. However, these lower bounds are only w.r.t. the number of variables and not w.r.t. the total size of the problem instance which, in particular, also includes the number of actions. The naive encoding requires as many actions as there are edges in the graph, i.e., the size of the problem instance is dominated by the number of actions. Thus, using this naive encoding we do not get interesting lower bounds w.r.t. the instance size. To obtain lower bounds w.r.t. the instance size we have to encode the vertices of the graph in a way that also allows us to encode the edges of the graphs compactly. For our reductions from triangle detection, i.e., Reduction 6.3.5, the edge relation can be rather arbitrary and, thus, the reduction is unlikely to allow for a compact representation in the factored model (without making additional assumptions). In contrast, our reductions from the OV-problem, i.e., Reduction 6.3.2, are well-suited for such a compact encoding as the resulting graphs are sparse and the edge relation is defined systematically based on the content of the vertices. Thus, if one uses the binary vectors as a basis for the encoding of the vertices in the factored model then the transitions can be represented with a relatively low number of actions. However, the details of such an encoding and the corresponding lower bounds depend on the actual factored model. Investigating such encodings for concrete factored models is beyond the scope of this chapter and we thus leave it open as an interesting direction for future research.

6.5.2 Concluding Remarks

In this chapter, we study several natural planning problems in graphs, MDPs, and game graphs, which are basic algorithmic problems in artificial intelligence. Our main contributions are a sub-quadratic algorithm for sequential reachability in MDPs, and quadratic conditional lower bounds. Note that graphs are a special case of both MDPs and game graphs, and the algorithmic problems are simplest for graphs, and in all cases except for AllCoverage, we have linear-time upper bounds. The key highlight of our results is an interesting separation of MDPs and game graphs: for reachability, MDPs are harder than game graphs; for the coverage problem, both MDPs and game graphs are hard (quadratic CLBs); for sequential reachability, game

graphs are harder than MDPs.

In this chapter, we clarified the algorithmic landscape of basic planning problems with CLBs and better algorithms. An interesting direction of future work would be to consider CLBs for other polynomial-time problems in planning and AI in general. For MDPs with sequential reachability objectives, we establish subquadratic upper bounds, and hence the techniques of the chapter that establish quadratic CLBs are not applicable. Other CLB techniques for this problem are an interesting topic to investigate as future work.

CHAPTER 7

Quasipolynomial Set-Based Symbolic Algorithms for Parity Games

In this chapter, we present the first quasi-polynomial symbolic algorithm for parity games.

7.1 Introduction

We present new contributions related to algorithms for parity games in the set-based symbolic model of computation.

Parity games. Games on graphs are central in many applications in computer science, especially, in the formal analysis of reactive systems. The vertices of the graph represent states of the system, the edges represent transitions of the system, the infinite paths of the graph represent traces of the system and the players represent the interacting agents. The reactive synthesis problem (Church's problem [100]) is equivalent to constructing a *winning strategy* in a graph game [54, 193, 198]. Besides reactive synthesis, the game graph problem has been used in many other applications, such as (1) verification of branching-time properties [125], (2) verification of open systems [15], (3) simulation and refinement between reactive systems [17, 142, 185]; (4) compatibility checking [11], (5) program repair [150], (6) synthesis of programs [66]; to name a few. Game graphs with *parity winning* conditions are particularly important since all ω-regular winning conditions (such as safety, reachability, liveness, fairness) as well as all Linear-time Temporal Logic (LTL) winning conditions can be translated into parity conditions [202, 203]. In a parity game, every vertex of the game graph is assigned a non-negative integer priority from $\{0, 1, \ldots, d-1\}$, and a play is winning if the minimum priority visited infinitely of-

ten is even. Game graphs with parity conditions can model all the applications mentioned above and are also equivalent to the modal μ-calculus [162] model-checking problem [125]. Thus the parity games problem is a core algorithmic problem in formal methods, and has received wide attention over the decades [47, 59, 125, 151, 152, 154, 208, 210, 224].

Models of computation: Explicit and symbolic algorithms. For the algorithmic analysis of parity games, two models of computation are relevant. First, the standard model of *explicit* algorithms, where the algorithms operate on the explicit representation of the game graph. Second, the model of *implicit or symbolic* algorithms, where the algorithms do not explicitly access the game graph but operate with a set of predefined operations. For parity games, the most relevant class of symbolic algorithms are called *set-based symbolic algorithms*, where the allowed symbolic operations are: (a) basic set operations such as union, intersection, complement, and inclusion; and (b) one step predecessor (Pre) operations (see [12, 105, 143]).

Significance of set-based symbolic algorithms. We describe the two most significant aspects of set-based symbolic algorithms.

1. Consider large-scale finite-state systems, e.g., hardware circuits, or programs with many Boolean variables or bounded-domain integer variables. While the underlying game graph is described implicitly (such as program code), the explicit game graph representation is huge (e.g., exponential in the number of variables). The implicit representation and symbolic algorithms often do not incur the exponential blow-up, which is inevitable for algorithms that require the explicit representation of the game graph. Data-structures such as Binary Decision Diagrams (BDDs) [51] (with well-established tools e.g. CuDD [212]) support symbolic algorithms that are used in verification tools such as NuSMV [104].
2. In several domains of formal analysis of infinite-state systems, such as games of hybrid automata or timed automata, the underlying state space is infinite, but there is a finite quotient. Symbolic algorithms provide a practical and scalable approach for the analysis of such systems: For many applications, the winning set is characterized by μ-calculus formulas with one-step predecessor operations which immediately give the desired set-based symbolic algorithms [10, 12]. Thus, the set-based symbolic model of computation is an equally important theoretical model of computation to be studied as the explicit model.

Symbolic resources. In the explicit model of computation, the two important resources are time and space. Similarly, in the symbolic model of computation, the two important resources are the number of symbolic operations and the symbolic space.

- *Symbolic operations:* Since a symbolic algorithm uses a set of predefined operations, instead of time complexity, the first efficiency measure for a symbolic algorithm is the number of symbolic operations required. Note that basic set operations (that only involve variables of the current state) are less resource

intensive compared to the predecessor operations (that involve both variables of the current and of the next state). Thus, in our analysis, we will distinguish between the number of basic set operations and the number of predecessor operations.

- *Symbolic space:* We refer to the number of sets stored by a set-based symbolic algorithm as the symbolic space for the following reason: A set that contains all vertices, or a set that contains all vertices where the first variable is true, represent each $\Theta(n)$ vertices, but can be represented as BDD of constant size. While the size of a set and the size of its symbolic representation is notoriously hard to characterize (e.g., for BDDs it can depend on the variable reordering), the symbolic model of computation counts every set as unit symbolic space, and the symbolic space requirement is thus the maximum number of sets required by a symbolic algorithm.

The goal is to find algorithms that minimize the symbolic space (ideally poly-logarithmic) and the symbolic operations.

Previous results. We summarize the main previous results for parity games on graphs with n vertices, m edges, and d priorities. To be concise in the following discussion, we ignore denominators in d in the bounds.

- *Explicit algorithms.* The classical algorithm for parity games requires $O(n^{d-1} \cdot m)$ time and linear space [184, 229], which was then improved by the small progress measure algorithm that requires $O(n^{d/2} \cdot m)$ time and $O(d \cdot n)$ space [151]. Many improvements have been achieved since then, such as the big-step algorithm [208], the sub-exponential time algorithm [154], an improved algorithm for dense graphs [80], and the strategy-improvement algorithm [224], but the most important breakthrough was achieved last year where a quasi-polynomial time $O(n^{\lceil \log d \rceil + 6})$ algorithm was obtained [59]. While the original algorithm of [59] required quasi-polynomial time and space, a succinct small progress measure based algorithm [152] and value-iteration based approach [128] achieve the quasi-polynomial time bound with quasi-linear space. However, all of the above algorithms are inherently explicit algorithms.
- *Set-based symbolic algorithms.* The basic set-based symbolic algorithm (based on the direct evaluation of the nested fixed point of the μ-calculus formula) requires $O(n^d)$ symbolic operations and $O(d)$ space [123]. In a breakthrough result [47] presented a set-based symbolic algorithm that requires $O(n^{d/2+1})$ symbolic operations and $O(n^{d/2+1})$ symbolic space (for a simplified exposition see [210]). In recent work [73], a new set-based symbolic algorithm was presented that requires $O(n^{d/3+1})$ symbolic operations and $O(n)$ symbolic space, where the symbolic space requirement is $O(n)$ even with a constant number of priorities.

Open questions. Despite the wealth of results for parity games, many fundamental algorithmic questions are still open. Besides the major and long-standing open question of the existence of a polynomial-time algorithm for parity games, two im-

Table 7.1: Set-Based Symbolic Algorithms for Parity Games.

reference	symbolic operations	symbolic space
[123, 229]	$O(n^d)$	$O(d)$
[47, 210]	$O(n^{d/2+1})$	$O(n^{d/2+1})$
[73]	$O(n^{d/3+1})$	$O(n)$
Thm. 7.4.2, 7.5.4	$n^{O(\log d)}$	$O(d \log n)$

portant open questions about set-based symbolic algorithms are as follows:

- *Question 1.* Does there exist a set-based symbolic algorithm that requires only quasi-polynomially many symbolic operations?
- *Question 2.* Given the $O(d)$ symbolic space requirement of the basic algorithm, whereas all other algorithms require at least $O(n)$ space (even for a constant number of priorities) an important question is: Does there exist a set-based symbolic algorithm that requires $\widetilde{O}(d)$ symbolic space (note that $\widetilde{O}$ hides poly-logarithmic factors), but beats the number of symbolic operations of the basic algorithm? This question is especially relevant since in many applications the number of priorities is small, e.g., in determinization of ω-automata, the number of priorities is logarithmic in the size of the automata [203].

Our contributions. In this work, we not only answer the above open questions (Question 1 and Question 2) in the affirmative but also show that both can be achieved by the same algorithm:

- First, we present a black-box set-based symbolic algorithm based on explicit progress measure algorithm for parity games that use $O(n)$ symbolic space and $n^{O(\log d)}$ symbolic operations. There are two important consequences of our algorithm: (a) First, given the ordered progress measure algorithm (which is an explicit algorithm), as a consequence of our black-box algorithm, we obtain a set-based symbolic algorithm for parity games that requires quasi-polynomially many symbolic operations and $O(n)$ symbolic space. (b) Second, any future improvement in progress measure based explicit algorithm (such as polynomial-time progress measure algorithm) would immediately imply the same improvement for set-based symbolic algorithms. Thus we answer Question 1 in the affirmative and also show that improvements in explicit progress measure algorithms carry over to symbolic algorithms.
- Second, we present a set-based symbolic algorithm that requires quasi-polynomially many symbolic operations and $O(d \cdot \log n) = \widetilde{O}(d)$ symbolic space. Thus we not only answer Question 2 in affirmative, we also match the number of symbolic operations with the current best-known bounds for explicit algorithms. Moreover, for the important case of $d \leq \log n$, our algorithm requires polynomially many symbolic operations and poly-logarithmic symbolic space.

We compare our main results with previous set-based symbolic algorithms in Table 7.1.

Symbolic Implementations. Recently, symbolic algorithms for parity games received attention from a practical perspective: First, three explicit algorithms (Zielonka's recursive algorithm, Priority Promotion [30] and Fixpoint-Iteration [48]) were converted to symbolic implementations [204]. The symbolic solvers had a huge performance gain compared to the corresponding explicit solvers on several of practical instances. Second, four symbolic algorithms to solve parity games were compared to their explicit versions (Zielonka's recursive algorithm, small progress measure, and an automata-based algorithm [167, 213]) [214]. For the symbolic versions of the small progress measure, two implementations were considered: (i) Symbolic Small Progress Measure using Algebraic Decision Diagrams [57] and (ii) the Set-Based Symbolic Small Progress Measure [73]. The symbolic algorithms were shown to perform better in several structured instances.

Other related works. Besides the discussed theoretical results on parity games, there are several practical approaches for parity games, such as, (a) accelerated progress measure [9], (b) quasi-dominion [30], (c) identifying winning cores [223], (d) BDD-based approaches [155, 156], and (e) an extensive comparison of various solvers [118]. A straightforward symbolic implementation (not set-based) of small progress measure was done in [57] using *Algebraic Decision Diagrams (ADDs)* and BDDs. Unfortunately, the running time is not comparable with our results as using ADDs breaks the boundaries of the Set-Based Symbolic Model: ADDs can be seen as BDDs which allow the use of a finite domain at the leaves [21]. Recently, a novel approach for solving parity games in quasi-polynomial time which uses the register-index was introduced [173]. Moreover, [173] presents a μ-calculus formula describing the winning regions of the parity game with alternation depth based on the register-index. The existence of such a μ-calculus formula does not immediately imply a quasi-polynomial set-based symbolic algorithm due to constructing the formula using the register-index.

Our work considers the theoretical model of symbolic computation and presents a black-box algorithm as well as a quasi-polynomial algorithm, matching the best-known bounds of explicit algorithms. Thus our work makes a significant contribution towards the theoretical understanding of symbolic computation for parity games.

7.2 The Progress Measure Algorithm

High-level intuition. Let $\mathcal{P} = (V, E, \langle V_1, V_2\rangle, p)$ be a parity game and let $(\mathscr{W}, \prec)$ be a finite total order with a maximal element $\top$ and a minimal element min. Let $C = \{1, \ldots, d-1\}$ be the set of all priorities. A *ranking function* is a function f which maps every vertex in V to a value in $\mathscr{W}$. The value of a vertex v concerning the ranking function f is called *rank* of v. The rank $f(v)$ of a vertex v determines how "close" the vertex is to being in W_z, the winning set of a fixed player z. Initially,

the rank of every vertex is the minimal value of $\mathscr{W}$. The *progress measure algorithm* iteratively increases the rank of a vertex v with an operator called $Lift$ with respect to the successors of v and another function called "lift". The algorithm terminates when the rank of no vertex can be increased any further, i.e., the least fixed point of $Lift$ is reached. We call the least simultaneous fixed point of all $Lift$-operators *progress measure*. When the rank of a vertex is the maximal element of the total order it is declared winning for player z. The rest of the vertices are declared winning for the adversarial player $\bar{z}$.

Ranking Function. Let $\mathscr{W}$ be a total order with a minimal element min and maximal element $\top$. A ranking function is a function $f : V \mapsto \mathscr{W}$.

The best function. The $best$ function represents the ability of player z, to choose the vertex in $Out(v)$ with the minimal ranking function. Analogously, it constitutes the ability of player $\bar{z}$ to choose the vertex in $Out(v)$ with the maximal ranking function. Formally, the function $best$ is defined for a vertex v and a ranking function f as follows:

$$best(f, v) = \begin{cases} \max\{f(w) \mid w \in Out(v)\} & \text{if } v \in V_{\bar{z}} \\ \min\{f(w) \mid w \in Out(v)\} & \text{if } v \in V_z \end{cases}$$

The lift*-function.* The function $\mathrm{lift} : \mathscr{W} \times C \mapsto \mathscr{W}$ defines how the rank of a vertex v is increased according to the rank r of a successor vertex, and the priority $p(v)$ of v. The lift function needs to be monotonic in the first argument. Notice that we do not need information about the graph to compute the lift function. In all known progress measures, the lift function is computable in constant time.

The $Lift$-operation. The $Lift$-operation potentially increases the rank of a vertex v according to its priority $p(v)$ and the rank of all its successors in the graph.[1]

$$Lift(f, v)(u) = \begin{cases} \mathrm{lift}(best(f, v), p(v)) & \text{if } u = v \\ f(u) & \text{otherwise} \end{cases}$$

A ranking function is a *progress measure* if it is the least simultaneous fixed point of all $Lift(\cdot, v)$-operators.

The Progress Measure Algorithm. The progress measure algorithm initializes the ranking function f with the minimum element of $\mathscr{W}$. Then, the $Lift(\cdot, v)$-operator is computed in an arbitrary order regarding the vertices. The winning set of player z can be obtained from a progress measure by selecting those vertices whose rank is $\top$. Notice that we need to define the total order $(\mathscr{W}, \prec)$ and a function lift to initialize the algorithm.

For example, the following instantiations of the progress measure algorithm determine the winning set of a parity game: (i) Small Progress Measure [151], (ii) Succinct Progress Measure [152] and the (iii) The Ordered Approach [128]. The

[1] Notice that in the original definition [151] the lift function is applied to all successors and the best of them is chosen subsequently. As the lift is monotone in the first argument the two definitions are equivalent.

running time is dominated by the size of $\mathscr{W}$. For a discussion on the state-of-the-art size of $\mathscr{W}$ we refer the reader to Remark 7.4.3.

7.3 Set-Based Symbolic Black Box Progress Measure Algorithm

In the symbolic setting, a parity game (Γ, p) is given by a game graph Γ and $V_i = \{v \in V \mid p(v) = i\}$ for $i \in C = \{1, \dots, d-1\}$. In this section, we briefly present a basic version of a set-based symbolic progress measure algorithm. The key idea is to compute the Lift-operation with a set-based symbolic algorithm. Then, we improve the basic version to obtain our black box set-based symbolic progress measure algorithm. Finally, we prove its correctness and analyze the symbolic resources.

Basic black box Algorithm. Throughout the algorithm, we maintain the family $\mathbf{S}$ of sets of vertices, which contains a set for every element in $\mathscr{W}$, i.e., $\mathbf{S} = \{S_r \mid r \in \mathscr{W}\}$. Intuitively, a vertex $v \in S_r$ has rank $f(v) = r$. Initially, we put each vertex into the set with the minimal value of $\mathscr{W}$, i.e., S_{min}. In each iteration, we consider all non-empty sets $S_r \in \mathbf{S}$: The algorithm checks if the ranking function of the predecessors of the vertices in S_r must be increased, i.e. $Lift(f, v)(v) \succ f(v)$ where $v \in Pre(S_r)$, and if so, performs the $Lift$-operator for the predecessors. We repeat this step until the algorithm arrives at a fixed point.

Performing a Lift operation. To compute the $Lift$-operation in the set-based symbolic setting we need to compute two functions for the predecessors of S_r: (1) the lift-function and (2) the $best$-function. By definition, the lift-function does not access the game graph or vertices thereof. Thus, we can compute the lift-function without the use of symbolic operations. To compute the $best$-function we need access to the game graph. It turns out it is simpler to compute the vertices with $best(f, v) \geq r$ rather than the vertices with $best(f, v) = r$. Thus, we lift all vertices v with $best(f, v) \geq r$ to the rank $\mathrm{lift}(r, p(v))$. To this end, we first compute the set $S_{\geq r} = \bigcup_{l \geq r} S_l$ of vertices with rank $\geq r$. Then, we compute $P = CPre_z(S_{\geq r})$ and, hence, the set P comprises the vertices $v \in P$ with $best(f, v) \geq r$. Finally, to compute $Lift(f, v)$, for each $c \in C$, we consider the vertices of P with priority c, i.e., the set $(P \cap V_c)$, and add them to the set $S_{\mathrm{lift}(r,c)}$. Notice that we lift each vertex v to $\mathrm{lift}(r, p(v))$ where $r = best(f, v)$ as we consider all non-empty sets $S_r \in \mathbf{S}$. No vertex v will be lifted to a set higher than $\mathrm{lift}(r, p(v))$ where $r = best(f, v)$ due to the monotonicity of the lift function in the first argument. If after an iteration of the algorithm a vertex appears in several sets of $\mathbf{S}$ we only keep it in the set corresponding to the largest rank and remove it from all the other sets.

7.3.1 Improving the Basic Algorithm

In this section, we improve the basic Algorithm by (a) reducing the symbolic space from $O(|\mathscr{W}|)$ to $O(n)$ and (b) by reducing the number of symbolic operations required to compute the fixed point.

Key Idea. The naive algorithm considers each non-empty set S_r in iteration i+1 again no matter if S_r has been changed in iteration i or not. Notice that we only need to consider the predecessors of the set S_r again when the set $S_{\geq r}$ in iteration i+1 contains additional vertices compared to the set $S_{\geq r}$ in iteration i. To overcome this weakness, we propose Algorithm 7.1. In this algorithm, we introduce a data structure called D. In the data structure D we keep track of the sets $S_{\geq r}$ instead of the sets S_r. The set $S_{\geq r}$ contains all vertices with a rank greater than or equal to r. Furthermore, we separately keep track of the elements $r \in \mathscr{W}$ where the set $S_{\geq r}$ changed since the last time $S_{\geq r}$ was selected to be processed. These elements of $\mathscr{W}$ are called *active*. Moreover, if we have two sets $S_{\geq r} = S_{\geq r'}$ with $r \prec r'$ there is no need to process the set $S_{\geq r}$ because $\mathrm{lift}(r', c) \geq \mathrm{lift}(r, c)$ holds due to the monotonicity of the lift function. To summarize, we precisely store a set $S_{\geq r}$ if there is no r' with $r \prec r'$ and $S_{\geq r} = S_{\geq r'}$. Notice, that this instantly gives us a bound on the symbolic space of $O(n)$.

Algorithm Description. In Algorithm 7.1 we use the data structure D to manage the active $r \in \mathscr{W}$ and in each iteration of the outer while-loop we process the corresponding set $S_{\geq r}$ of such an $r \in \mathscr{W}$. We first compute $P = CPre_z(S_{\geq r})$, then, for each $c \in C$ we compute $r' = \mathrm{lift}(r, c)$ and update the set $S_{\geq r'}$ by adding $P \cap V_c$. The inner while-loop ensures that $P \cap V_c$ is also added to all the sets $S_{\geq r'}$ with $r' \prec r$ and the properties of the data structure are maintained, i.e., (a) all active elements are in the active list, and (b) exactly those $r \in \mathscr{W}$ with $S_{\geq r} \supset S_{\geq r'}$, for $r \prec r'$ are stored in D.

Active Elements. Intuitively, an element $r \in \mathscr{W}$ is *active* if $S_{\geq r} \supset S_{\geq r'}$, for all r' where $r \prec r'$ and $S_{\geq r}$ has been changed since the last time $S_{\geq r}$ was selected at Line 3 of Algorithm 7.1. We define active elements more formally later.

Datastructure 7.3.1. *Our algorithm relies on a data structure D, which supports the following operations:*

- *D.popActiveSet() returns an element $r \in \mathscr{W}$ marked as active and makes it inactive. If all elements are inactive, returns false.*
- *D.getSet(r) returns the set $S_{\geq r}$.*
- *D.getNext(r) returns the smallest r' with $S_{\geq r} \supset S_{\geq r'}$.*
- *D.getPrevious(r) returns the largest r' where $S_{\geq r} \subset S_{\geq r'}$.*
- *D.removeSet(r) marks r as inactive.*
- *D.activate(r) marks r as active.*
- *D.update(r, S) updates the set $S_{\geq r}$ to S, i.e., D.getSet(r) returns S. Moreover, all sets $S_{\geq r'}$ with $r' \prec r$ and $S_{\geq r'} = S_{\geq r}$ beforehand are updated to S as well.*

We initialize D with D.update(min, V) and D.update($\top, \emptyset$).

We can define *active* elements formally now as the definition depends on D.

Definition 7.3.2. *Let $S^0_{\geq r} = D.getSet(r)$ be the set stored in D for $S_{\geq r}$ after the initialization of D, and let $S^i_{\geq r} = D.getSet(r)$ be the set stored in D for $S_{\geq r}$ after the i-th iteration of the while-loop at Line 3. An element $r \in \mathscr{W}$ is active after the i-th*

Algorithm 7.1: Black Box Set-Based Symbolic Progress Measure

```
   input :Parity Game 𝒫
1  Initialize data structure D
2  D.activate(min)
3  while r ← D.popActiveSet() do
4      S_{≥r} ← D.getSet(r)
5      P ← CPre_z(S_{≥r})
6      for c ∈ C do
7          r' ← lift(r, c)
8          S_{≥r'} ← D.getSet(r')
9          while P ∩ V_c ⊈ S_{≥r'} do
10             S_{≥r'} ← S_{≥r'} ∪ (P ∩ V_c)
11             S_{≥next(r')} ← D.getSet(D.getNext(r'))
12             if r' = ⊤ or S_{≥r'} ⊃ S_{≥next(r')} then
                   // S_{≥r'} is a superset of S_{≥next(r')} and we save it
13                 D.update(r', S_{≥r'}); D.activate(r')
14             S_{≥prev(r')} ← D.getSet(D.getPrevious(r'))
15             if S_{≥r'} = S_{≥prev(r')} ∪ (P ∩ V_c) then
                   // We only keep sets which are different
16                 D.removeSet(D.getPrevious(r'))
17             r' ← D.getPrevious(r')
18             S_{≥r'} ← D.getSet(r')
19 return S_⊤
```

iteration of the while-loop if (i) for all $r' \in \mathscr{W}$ where $r \prec r'$ we have $S^i_{\geq r} \supset S^i_{\geq r'}$ and (ii) there is a $j < i$ such that $S^i_{\geq r} \supset S^j_{\geq r}$ and for all $j < j' \leq i$ the set $S_{\geq r}$ is not selected in Line 3 in the j'-th iteration. Additionally, we consider $r = \min$ as active before the first iteration. An element $r \in \mathscr{W}$ is inactive if it is not active.

Notice that, in Algorithm 7.1 an $r \in \mathscr{W}$ is active iff r is marked as active in D. The algorithm ensures this in a very direct way. At the beginning, only $\min \in W$ is active, which is also set active in D in the initial phase of the algorithm. Whenever some vertices are added to a set $S_{\geq r}$, it is tested whether $S_{\geq r}$ is larger than its successor and if so r is activated (Lines 12-13). On the other hand, if something is added to the successor of $S_{\geq r}$ in the data structure D then the algorithm tests whether the two sets are equal and if so r is rendered inactive (Lines 15-16).

Implementation of the data structure D. The data structure uses an AVL-tree and a doubly linked list called "active list" that keeps track of the active elements. The nodes of the tree contain a pointer to the corresponding set $S_{\geq r}$ and to the corresponding element in the active list.

- Initialization of the data structure D: Create the AVL tree with the elements *min* and $\top$. The former points to the set of all vertices and the latter to the empty set. Create the doubly linked list called "active list" as an empty list.

- $D.popActiveSet()$: Return the first element from the active list and remove it from the active list. If the list is empty, return false.
- $D.getSet(r)$: Searches the AVL tree for r or for the next greater element (w.r.t. $\succeq$). Then we return the set by using the pointer we stored at the node.
- $D.getNext(r)$: First performs $D.getSet(r)$ and then computes the inorder successor in the AVL-tree. This corresponds to the next greater node w.r.t. $\succeq$.
- $D.getPrevious(r)$: First performs $D.getSet(r)$ and then computes the inorder predecessor in the AVL-tree. This corresponds to the next smaller node w.r.t. $\succeq$.
- $D.removeSet(r)$: This operation needs the element r to be stored in the AVL tree. Search the AVL tree for r. Remove the corresponding element from the active list and the AVL Tree.
- $D.activate(r)$: This operation needs the element r to be stored in the AVL tree. Add r to the active list and add pointers to the AVL-tree. The element in the active list contains a pointer to the tree element and vice versa.
- $D.update(r, S)$: Perform $S_{\succeq r} \leftarrow D.getSet(r)$: If r is contained in the AVL tree then update $S_{\succeq r}$ to S. Otherwise, insert r as a new element and let the element point to S.

We initialize the data structure D with $min \in \mathscr{W}$ and $\top \in \mathscr{W}$. Thus, whenever we query D for a value $r \in \mathscr{W}$ we find it or there exists an $r' \succ r$ which is in D.

Analysis of the data structure D.

The data structure can be implemented with an AVL-tree and a doubly linked list called "active list" that keeps track of the active elements such that all of the operations can be performed in $O(\log n)$: when the algorithm computes $D.update(r, S)$ we store r and a pointer to the set S as a node in the AVL tree. By construction, the algorithm only stores pointers to different sets and when we additionally preserve anti-monotonicity among the sets we only store $\leq n$ sets. Therefore, the AVL tree has only $\leq n$ nodes with pointers to the corresponding sets and searching for a set with the operation $D.getSet(r)$ only adds a factor of $\log n$ to the non-symbolic operations when we store r as key with a pointer to $S_{\succeq r}$ in the AVL-tree. Moreover, we maintain pointers between the elements of the active list and the corresponding vertices in the AVL tree.

Remark 7.3.3. The described algorithm is based on a data structure D which keeps track of the sets that will be processed at some point later in time. Note that this data structure does not access the game graph but only stores pointers to sets that the Algorithm 7.1 maintains. The size of the AVL tree implementing D is proportional to the symbolic space of the algorithm.

7.3.2 Correctness

To prove the correctness of Algorithm 7.1 we tacitly assume that the algorithm terminates. An upper bound on the running time is then shown in Proposition 7.3.10.

Proposition 7.3.4 (Correctness.). *Let $\mathscr{P}$ be a parity game. Given a finite total order $(\mathscr{W}, \prec)$ with minimum element min, a maximum element $\top$ and a monotonic function* lift $: \mathscr{W} \times C \mapsto \mathscr{W}$ *Algorithm 7.1 computes the least simultaneous fixed point of all $Lift(\cdot, v)$-operators.*

To prove the correctness of Algorithm 7.1, we prove that when Algorithm 7.1 terminates, the function $\rho(v) = \max\{r \in \mathscr{W} \mid v \in S_{\geq r}\}$ is equal to the least simultaneous fixed point of all $Lift(\cdot, v)$-operators. We show that when the properties described in Invariant 7.3.5 hold, the function ρ is equal to the least fixed point at the termination of the algorithm. Then, we prove that we maintain the properties of Invariant 7.3.5.

Invariant 7.3.5. *Let $\tilde{\rho}$ be the least simultaneous fixed point of $Lift(\cdot, v)$ and $\rho(v) = \max\{r \in \mathscr{W} \mid v \in S_{\geq r}\}$ be the ranking function w.r.t. the sets $S_{\geq r}$ that are maintained by the algorithm.*

1. *Before each iteration of the while-loop at Line 3 we have $S_{\geq r_2} \subseteq S_{\geq r_1}$ for all $r_1 \preceq r_2$ (anti-monotonicity).*
2. *Throughout Algorithm 7.1 we have $\tilde{\rho}(v) \succeq \rho(v)$ for all $v \in V$.*
3. *For all $r \in \mathscr{W}$: (a) r is active or (b) for all $v \in CPre_z(S_{\geq r})$: if $best(\rho, v) = r$, then $\rho(v) = Lift(\rho, v)(v)$.*

In the following paragraph, we describe the intuition of Invariant 7.3.5. Then, we show that the properties of Invariant 7.3.5 are sufficient to obtain the correctness of Algorithm 7.1. Finally, we prove that each property holds during the while-loop at Line 3.

Intuitive Description. The intuitive description is as follows:

1. Ensures that the sets $S_{\geq r}$ contain the correct elements. Having the sets $S_{\geq r}$ allows computing $best(f, v) \succeq r$ as discussed at the beginning of the section.
2. Guarantees that ρ is a lower bound on $\tilde{\rho}$ throughout the algorithm.
3. When an $r \in \mathscr{W}$ is not active, the rank of no vertex can be increased by applying lift to the vertices which have $best(\rho, v) = r$.

When the algorithm terminates, all $r \in \mathscr{W}$ are inactive and ρ is a fixed point of all $Lift(\rho, v)$ by condition (3b). The next lemma proves that Algorithm 7.1 computes the least simultaneous fixed point of all $Lift(\cdot, v)$ operators for a parity game.

Lemma 7.3.6 (The Invariant is sufficient). *Let the* lift *function be monotonic in the first argument and $(\mathscr{W}, \prec)$ be a total order. The ranking function ρ at termination of Algorithm 7.1 is equal to the least simultaneous fixed point of all $Lift(\cdot, v)$-operators for the given parity game $\mathscr{P}$.*

Proof. Consider the ranking function $\rho(v) = \max\{r \in \mathscr{W} \mid v \in S_{\geq r}\}$ computed by Algorithm 7.1. By Invariant 7.3.5(2) we have $\tilde{\rho}(v) \succeq \rho(v)$ for all $v \in V$. We next show that $\rho(v)$ is a fixed point of $Lift(\rho, v)$ for all $v \in V$. When the algorithm terminates, no $r \in \mathscr{W}$ is active. Consider an arbitrary v and let $r = best(\rho, v)$. Now, as the set r is not active, by Invariant 7.3.5(3b), we have $\rho(v) = Lift(\rho, v)(v)$. Thus $\rho(v)$ is a fixed

point of $Lift(\rho, v)$ for all vertices in V. Therefore, as ρ is a simultaneous fixed point of all $Lift(\cdot, v)$-operators and $\tilde{\rho}$ is the least such fixed point, we obtain $\rho(v) \succeq \tilde{\rho}(v)$ for all $v \in V$. Hence we have $\rho(v) = \tilde{\rho}(v)$ for all $v \in V$. □

The following lemmas prove each part of the invariant separately. The first part of the invariant describes the anti-monotonicity property which is needed to compute the $best$ function with the $CPre_z$ operator.

Lemma 7.3.7. *Invariant 7.3.5(1) holds: Let $r_1, r_2 \in \mathscr{W}$ and $r_1 \preceq r_2$. Before each iteration of the while-loop at Line 3 we have that if a vertex v is in a set $S_{\geq r_2}$ then it is also in $S_{\geq r_1}$ (anti-monotonicity).*

Proof. We prove the claim by induction over the iterations of the while-loop. Initially, the claim is satisfied as the only non-empty set is S_{min}. It remains to show that when the claim is valid at the beginning of an iteration, then the claim also holds in the next iteration. By the induction hypothesis, the claim holds for the sets at the beginning of the while-loop. In the trivial case, the algorithm terminates and the claim holds by the induction hypothesis. Otherwise, the sets are only modified at Line 10 and stored at Line 13. First, the vertices $P \cap V_c$ are added into the set $S_{\geq r'}$. Let $r'' = D.getPrevious(r')$. Notice that after activating r' all r with $r'' \prec r \prec r'$ refer to the same set as r' and thus we add $P \cap V_c$ implicitly to all r. In the next iteration the while-loop then adds $P \cap V_c$ also to the set $S_{\geq r''}$. As this done iteratively until a set $S_{\geq r^*}$ with $P \cap V_c \subseteq S_{\geq r^*}$ is reached (Lines 9-17), the algorithm ensures that $P \cap V_c$ is contained in all set $S_{\geq r''}$ with $r \prec r'$. By induction hypothesis we know that the invariant holds for all $r_2 \succ r'$ ($S_{\geq r_2}$ is unchanged), and as the algorithm added $P \cap V_c$ to all set $S_{\geq r''}$ with $r'' \preceq r'$ the claim holds for all $r_1, r_2 \preceq r'$. □

The second part of the invariant shows that the fixed point Algorithm 7.1 computes is always smaller or equal to the least fixed point. In particular, the fixed point computed by the algorithm is defined as $\rho(v) = \max\{r \in \mathscr{W} \mid v \in S_{\geq r}\}$ and we denote the least fixed point with $\tilde{\rho}$. The proof is by induction: In the beginning, every vertex is initialized with the minimum element which suffices for the claim. When we apply the lift function to vertices, we observe that by the induction hypothesis the current value of a vertex is below or equal to the fixed point. Additionally, we obtain a rank that is also smaller or equal to the lifted value of $\tilde{\rho}$ for every vertex as lift is a monotonic function.

Lemma 7.3.8. *Invariant 7.3.5(2) holds: Throughout Algorithm 7.1 we have $\tilde{\rho}(v) \geq \rho(v)$ for all $v \in V$.*

Proof. Before the while-loop at Line 3 the claim is obviously satisfied as $\tilde{\rho}(v) \succeq \min$ for all $v \in V$. We prove the claim by induction over the iterations of the while-loop: Assume we have $\rho(v) \preceq \tilde{\rho}(v)$ for all $v \in V$ before an iteration of the while-loop. The function $\rho(\cdot)$ is only changed at Line 10 and stored at Line 13 where the set $(P \cap V_c)$ is added to $S_{\geq r'}$. For $v \in P \cap V_c$ we have that v is a priority c vertex and either v is a player-z vertex with a successor in $S_{\geq r}$ or a player-$\bar{z}$ vertex with all successors

in $S_{\geq r}$. Thus, $r \preceq best(\rho, v)$ for $v \in P$. At Line 7 we compute the lift-operation for ranking r with priority c which results in the ranking r' for the first iteration of the while-loop. By the monotonicity of the lift operation and the induction hypothesis we have that $r' = \text{lift}(r, c)(v) \preceq \text{lift}(best(\rho, v), c)(v) \preceq \text{lift}(best(\tilde{\rho}, v), c) = \tilde{\rho}(v)$ for $v \in P \cap V_c$ and thus adding v to $S_{\geq r'}$ maintains the invariant (if $\rho(v) \succ r'$ beforehand it is not changed and otherwise it is lifted to $r' \prec \tilde{\rho}(v)$). In the later iterations of the while-loop $P \cap V_c$ is added to sets with smaller r', which does not affect ρ, as these vertices already appear in sets with larger rank. □

The following lemma proves the third part of Invariant 7.3.5: Either there is an active $r \in \mathscr{W}$, i.e., the set $S_{\geq r}$ needs to be processed, or $\rho(v)$ is a fixed point. We prove the property again by induction: Initially, the set $min \in \mathscr{W}$ is active and every other set is empty which trivially fulfills the property. Then, in every iteration when we change a set with value r we either activate it, or there is a set with a value $r' \geq r$ where $S_{\geq r'}$ subsumes $S_{\geq r}$. In the former case, the condition is instantly fulfilled. In the latter case, there is no vertex v where $best(\rho, v) = r$ which renders $S_{\geq r}$ irrelevant by definition of ρ.

Lemma 7.3.9. *Invariant 7.3.5(3) holds: For all $r \in \mathscr{W}$:*

1. *$S_{\geq r}$ is active or,*
2. *$\forall v \in CPre_z(S_{\geq r})$: if $best(\rho, v) = r$, then $\rho(v) = Lift(\rho, v)(v)$*

Proof. We prove this invariant by induction over the iterations of the while-loop: Before the while-loop at Line 3 the claim is obviously satisfied as we activate min which contains all vertices; for all other $r \in \mathscr{W}$ the set $S_{\geq r}$ is empty and thus condition (2) is trivially satisfied.

Assume the condition holds at the beginning of the loop. We can, therefore, assume by the induction hypothesis that the condition holds for all the sets. If there is no active $r \in \mathscr{W}$, the algorithm terminates and the condition holds by the induction hypothesis. The condition for a set $S_{\geq r}$ can be violated only if either the set $S_{\geq r}$ is changed or the set $S_{\geq r}$ is deactivated. That is either at Line 3, Line 10 or Line 16 of the algorithm.

Let us first consider the changes made in the while-loop. If a set $S_{\geq r}$ is changed in Line 10, then the algorithm either activates r (Line 13) and thus satisfies (1) or $S_{\geq next(r)} = S_{\geq r}$ which implies that $best(\rho, v) \neq r$ and thus (2) is fulfilled trivially. At Line 16 there is no vertex v with $best(\rho, v) = r$ (as there is no vertex w with $\rho(w) = r$) and thus (2) is satisfied (and it is safe to remove/deactivate the set in Line 16).

Now consider the case where we remove the set $S_{\geq r}$ and make r inactive at Line 3. If the set $S_{\geq r}$ is unchanged during the iteration of the outer while-loop then $S_{\geq r}$ satisfies condition (2) after the iteration. This is because for all v with $best(\rho, v) = r$ and $p(v) = c$ we have that if v is not already contained in $S_{\geq \text{lift}(r,c)}$ the algorithm adds it to the set $S_{\geq \text{lift}(r,c)}$ in Line 10 in the first iteration of the while-loop when processing c. This is equivalent to applying $Lift(\rho, v)(v) = \text{lift}(r, c)$. If the set $S_{\geq r}$ is changed during the iteration then this happens in the inner while-loop.

As argued above, then either r is activated and thus satisfies (1) or $S_{\geq next(r)} = S_{\geq r}$ holds. Thus, there is no vertex v with $best(\rho, v) = r$, i.e., (2) is satisfied. □

Symbolic Resources

In the following, we discuss the symbolic resources Algorithm 7.1 needs. We determine the number of symbolic one-step operations, the number of basic set operations, and the symbolic space consumption.

Proposition 7.3.10. *The number of symbolic one-step operations in Algorithm 7.1 is in* $O(n \cdot |\mathscr{W}|)$.

Proof. Each iteration of the while-loop at Line 3 processes an active r. That means, that the set $S_{\geq r}$ was changed in a prior iteration. We use a symbolic one-step operation at Line 5 for each active $S_{\geq r}$. It, therefore, suffices to count the number of possibly active sets throughout the execution of the algorithm. Initially only $S_{\geq \min}$ is active. After extracting an active set out of the data structure D, it is deactivated at Line 3. We only activate a set $S_{\geq x}$ when a new vertex is added to it at Line 13. Because there can only be n vertices with ranking $\geq x$ for all $x \in \mathscr{W}$ the size of each set $|S_{\geq x}|$ is smaller or equal to n. In the worst case, we eventually put every vertex into every set $S_{\geq x}$ where $x \in \mathscr{W}$. Thus we activate $n \cdot |\mathscr{W}|$ sets which is equal to the number of symbolic one-step operations. □

A similar argument works for analysing the number of basic set operations.

Proposition 7.3.11. *The number of basic set operations in Algorithm 7.1 is in* $O(d \cdot n \cdot |\mathscr{W}|)$.

Proof of Proposition 7.3.11. As proven in Proposition 7.3.10, there are $O(n \cdot |\mathscr{W}|)$ iterations of the outer while-loop and thus $O(n \cdot |\mathscr{W}|)$ iterations of the for-loop. Thus the inner while-loop is started $O(dn \cdot |\mathscr{W}|)$ times. The test whether the while-loop is started only requires two basic set operations and the overall costs are bound by $O(dn \cdot |\mathscr{W}|)$. We bound the overall costs for the iterations of the inner while-loop by an amortized analysis. First, notice that each iteration just requires 8 basic set operations (including testing the while condition afterward). In each iteration for a value $r' \in \mathscr{W}$ we charge the r' for the involved basic set operations. Notice, that in each such iteration new vertices are added to the set $S_{\geq r'}$ and thus r' is processed at most n times. Thus each $r' \in \mathscr{W}$ is charged for at most $8n$ basic set operations Therefore, the number of basic set operations is $O(d \cdot n \cdot |\mathscr{W}|) + O(n \cdot |\mathscr{W}|) = O(d \cdot n \cdot |\mathscr{W}|)$. □

Due to Proposition 7.3.4, Proposition 7.3.10, Proposition 7.3.11 and the fact that we use $\leq n$ sets in the data structure D, we obtain Theorem 7.3.12.

Theorem 7.3.12. *Given a parity game, a finite total order* $(\mathscr{W}, \succ)$ *and a monotonic function* lift *we can compute the least fixed point of all* $Lift(\cdot, v)$ *operators with* $O(n \cdot |\mathscr{W}|)$ *symbolic one-step operations,* $O(d \cdot n \cdot |\mathscr{W}|)$ *basic set operations, and* $O(n)$ *symbolic space.*

7.4 Implementing the Ordered Progress Measure

In this section, we plug the ordered approach to progress measure (OPM) described by Fearnley et al. [128] into Algorithm 7.1. To do this, we recall the witnesses they use in their algorithm and encode it with a specially-tailored technique to obtain an algorithm with a sublinear amount of symbolic space. Finally, we argue that the function $\mathrm{lift} : \mathscr{W} \times C \mapsto \mathscr{W}$ and the total order $(\mathscr{W}, \preceq)$ described in [128] can be used to fully implement Algorithm 7.1.

The Ordered Progress Measure. To implement the ordered progress measure algorithm we argue that the lift-operation is monotonic in the first argument and the order $(\mathscr{W}, \preceq)$ is a total finite order to fulfill the conditions of Algorithm 7.1. Let $\mathscr{P}$ be a parity game and $C = \{0, \ldots, d-1\}$ be the set of priorities in $\mathscr{P}$. The set $\mathscr{W}$ in the ordered progress measure consists of tuples of priorities of length k, where $k \in O(\log n)$. Each element in the tuple is an element of $C_\_ = C \cup \{\_\}$, i.e., it is either a priority or "_". The set $C_\_$ has a total order $(C_\_, \preceq)$ such that _ is the smallest element, odd priorities are ordered descending and are considered smaller than even priorities which are ordered ascending. The order $(\mathscr{W}, \preceq)$ is then obtained by extending the order $(C_\_, \preceq)$ lexicographically to the tuples $r \in \mathscr{W}$.

For the details of the lift function, we refer the reader to the work of Fearnley et al. [128]. An implementation of the lift operation can be found at the GitHub repository of the Oink system [118].

By the results in [128] the order $(\mathscr{W}, \preceq)$ and lift meet the requirements of our algorithms.

Lemma 7.4.1. *The following holds: (1) The function* $\mathrm{lift} : \mathscr{W} \times C \mapsto \mathscr{W}$ *is monotonic in the first parameter [128, p.6]. (2) The order* $(\mathscr{W}, \preceq)$ *is a total finite order [128, p.3]. (3) Let* ρ *be the least simultaneous fixed point of all* $Lift(\cdot, v)$ *operators. Then* $\rho(v) = \top$ *iff player* $\mathscr{E}$ *has a strategy to win the parity game* $\mathscr{P}$ *when starting from* v *[128, Lemma 7.3, Lemma 7.4].*

Theorem 7.3.12 together with Lemma 7.4.1 imply the following theorem.

Theorem 7.4.2. *Algorithm 7.1 implemented with the OPM computes the winning set of a parity game with* $O(n \cdot |\mathscr{W}|)$ *symbolic one-step operations,* $O(d \cdot n \cdot |\mathscr{W}|)$ *basic set operations, and* $O(n)$ *symbolic space.*

Remark 7.4.3. (Bounds for $|\mathscr{W}|$). We now discuss the bounds on $|\mathscr{W}|$. The breakthrough result of [59] shows that $|\mathscr{W}|$ is quasi-polynomial ($n^{O(\log d)}$) in general and polynomial when $d \leq \log n$. Using the refined analysis of [128], we obtain the following bound on $|\mathscr{W}|$: in general, $\min(n \cdot \log(n)^{d-1}, h \cdot n^{c_{1.45}+\log_2(h)})$, where $c_{1.45} = \log_2(e) < 1.45$ and $h = \lceil 1 + d/\log(n) \rceil$; and if $d \leq \log n$, then $|\mathscr{W}|$ is polynomial due to [59, Theorem 2.8] and [128, Corollary 8.8]. Note that $O\left(n^{2.45+\log_2(d)}\right)$ gives a naive upper bound on $|\mathscr{W}|$ in general. Plugging the bounds in Theorem 7.4.2 we obtain a set-based symbolic algorithm that requires quasi-polynomially many symbolic one-step and basic set operations and $O(n)$ symbolic space. The algorithm

requires only polynomially many symbolic one-step and basic set operations when $d \leq \log n$.

7.5 Reducing the Number of Sets for the OPM

In this section, we tailor a data structure for the OPM to only use $O(d \cdot \log n)$ sets. While each progress measure can be encoded by $\log(|\mathscr{W}|)$ many sets, the challenge is to provide a representation that also allows to efficiently compute the sets $S_{\geq r}$. Such a representation has been provided for the small progress measure [73] and in the following, we adapt their techniques for the OPM.

Key Idea. The key idea of the symbolic space reduction is that we *encode* the value of each coordinate of the rank r separately. A set no longer just stores the vertices with specific rank $r = b_1 \dots b_k$ but instead stores all vertices where, say, the first coordinate b_1 is equal to a specific value in $C_\_$. This encoding enables us to use only a polylogarithmic amount of symbolic space under the assumption that the number of priorities in the game graph is polylogarithmic in the number of vertices.

Symbolic Space Reduction. Let the rank of v be $r = b_1 \dots b_k$. Vertex v is in the set C^i_x iff the ith coordinate of the rank of v is x and a vertex v is in the set $C_\top$ iff the rank of v is $\top$. Thus $O(\log(n) \cdot d)$ sets suffice to encode all $r \in \mathscr{W}$. We demonstrate this encoding of the sets in Example 7.5.1.

Example 7.5.1. *Let $\mathscr{P}$ be a parity game containing the vertices v_1, v_2, v_3. Assume the following ranking function: $f(v_1) = 65433$, $f(v_2) = 75422$, $f(v_3) = \_\_\_32$. Using the definition of our encoding, we have that: $\{v_3\} \subseteq C^1_\_, \{v_1\} \subseteq C^1_6, \{v_2\} \subseteq C^1_7, \{v_3\} \subseteq C^2_\_, \{v_1, v_2\} \subseteq C^2_5, \{v_3\} \subseteq C^2_\_, \{v_1, v_2\} \subseteq C^3_4, \{v_2\} \subseteq C^4_2, \{v_1, v_3\} \subseteq C^4_3, \{v_1\} \subseteq C^5_3, \{v_2, v_3\} \subseteq C^5_2$.*

Computing the set $S_{\geq r}$ from C^i_x. We obtain the set S_r for rank $r = b_1 \dots b_k$ with an intersection of the sets $\bigcap_{i=1}^{k} C^i_{b_i} = S_r$. To acquire the set $S_{\geq r}$ we first consider sets where the first i elements are the equal to $b_1, \dots b_i$ but the $i{+}1$th element x is $\succ b_{i+1}$.

$$S^i_{\geq r} = \bigcap_{1 \leq j \leq i} C^j_{b_j} \cap \bigcup_{x \succ b_{i+1}} C^{i+1}_x \tag{7.1}$$

To *construct* the set $S_{\geq r}$ we apply the following union operations:

$$S_{\geq r} = \bigcup_{i=1}^{k-1} S^i_{\geq r} \cup S_r \cup C_\top \tag{7.2}$$

That is, we can compute the set $S_{\geq r}$ with $O(d \cdot \log n)$ set operations and four additional sets. Notice that there is no need to store all sets $S^i_{\geq r}$ as we can immediately add them to the final set when we have computed them. The number of $\cup$-operations is immediately bounded by $O(d \cdot k) = O(d \cdot \log n)$ by the above definitions. In order

to bound the number of $\cap$-operations by $O(\log n)$, we do the following. To compute the sets $S^i_{\geq r}$ we introduce an additional set $T^i = \bigcap_{1 \leq j \leq i} C^j_{b_j}$. We have that $S^i_{\geq r} = T^i \cap \bigcup_{x > b_{i+1}} C^{i+1}_x$ and $T^{i+1} = T^i \cap C^{i+1}_{b_{i+1}}$, i.e., we just need two $\cap$ operation to compute the next set $S^{i+1}_{\geq r}$. Moreover, we have that $S_r = T^k$ and thus can be computed with just one $\cap$ operation. In total, this amounts to $2k - 2 = O(\log n)$ many $\cap$-operations.

Updating the set $S_{\geq r}$ to $S'_{\geq r}$. Assume that the set $S_{\geq r}$ is the old set that is saved within the sets C^i_x. The new set, $S'_{\geq r}$ is an updated set, which is also a superset. First, compute the difference $S_\Delta = S'_{\geq r} \setminus S_{\geq r}$. Intuitively, the algorithm increased the rank of the vertices in S_Δ. We delete their old values by updating $C^i_x = C^i_x \setminus S_\Delta$ for all $i = 0 \dots k$ and each $x \in \{0, \dots d-1\}$. Then we add the vertices to the set $C^i_{r_i} = C^i_{r_i} \cup S_\Delta$ for all $i = 0 \dots k$. In total there are $O(d \log n)$ many $\setminus$-operations and $O(k) = O(\log n)$ many $\cup$-operations.

7.5.1 Algorithmic Details for the Reduced Symbolic Space Algorithm 1

In this section we present the algorithmic details for the reduced symbolic space algorithm.

Data Structure. The new data structure D supports all functions of Data Structure 7.3.1 but the nodes of the AVL tree no longer store a pointer to a set (but only the value r). The data structure D stores $d \log n + 1$ sets as described in Section 7.5, and overrides the $D.update(r, S)$-operation.

- D.update(r,S): Updates the set $S_{\geq r}$ to be S. Every set $S_{\geq r'}$ with $r' \prec r$ is updated to $S_{\geq r'} \cup S$.

Implementation of the data structure D.

- D.getSet(r): This function computes the set $S_{\geq r}$ as described in Section 7.5 and returns it.
- D.update(r,S): Computes the update of the set $S_{\geq r}$ with S as described in Section 7.5. Precondition: $S_{\geq r} \subseteq S$.

Notice that Algorithm 7.2 only differs from Algorithm 7.1 in (a) the way the `D.getSet(r)` method is implemented and (b) and how we update the sets, i.e., the overridden `D.update(r,S)` method. Hence, to establish the correctness of Algorithm 7.2, it suffices to show that these two methods do the same as the corresponding operations in Algorithm 7.1. The correctness then directly follows from Proposition 7.3.4.

Proposition 7.5.2 (Correctness)**.** *Given a parity game $\mathcal{P}$ Algorithm 7.2 computes the corresponding winning set.*

Algorithm 7.2: Parity Algorithm with reduced sets

```
   input : Parity Game 𝒫
1  Initialize C^i ← V for 0 ≤ i ≤ k
2  Initialize C_c^i ← ∅ for 0 ≤ i ≤ k, c ∈ C
3  D.activate(min)
4  while r ← D.popActiveSet() do
5      S_≥r ← D.getSet(r)
6      D.deactivate(r)
7      P ← CPre_z(S_≥r)
8      for c ∈ C do
9          r' ← lift(r, c)
10         S_≥r' ← D.getSet(r')
11         rold ← r'
12         S_rold ← S_≥r' ∪ (P ∩ V_c)
13         while P ∩ V_c ⊈ S_≥r' do
14             S_≥r' ← S_≥r' ∪ (P ∩ V_c)
15             S_≥next(r') ← D.getSet(D.next(r'))
16             if r' = ⊤ or S_≥r' ⊃ S_≥next(r') then
17                 D.activate(r', S_≥r')
18             S_≥prev(r') ← D.getSet(D.getPrevious(r'))
19             if S_≥r' = S_≥prev(r') ∪ (P ∩ V_c) then
20                 D.removeSet(D.getPrevious(r'))
21             r' ← D.getPrevious(r')
22             S_≥r' ← D.getSet(r')
23         D.update(rold, S_rold)
24 return S_⊤
```

Proof. We show that the `D.getSet(r)` method (in the interplay with the `D.update(r,S)` method) in Algorithm 7.2 returns the sames set as the `D.getSet(r)` method in Algorithm 7.1. The correctness then directly follows from Proposition 7.3.4.

The proof is by induction. Consider the base case after the initialization of the data structure D and its sets C_c^i. In Algorithm 7.1 we have that $D.getSet(\min)$ would return V and $D.getSet(r)$ would return the empty set for $\min \prec r$. In Algorithm 7.2, by the initialization in Line 1, we have that $D.getSet(\min)$ would return V and by the initialization in Line 2 we have $D.getSet(r)$ would return the empty set for $\min \prec r$. Therefore, the base case is satisfied.

Notice that the data structures both in Algorithm 7.1 and Algorithm 7.2 are only changed in the for-loop. Assume the claim holds before an iteration of the for-loop. Let $\bar{r}$ be the element of $\mathscr{W}$ currently processed by the outer while-loop, and let $\bar{r}'$ the r' currently processed by the for-loop. Consider some $r \in \mathscr{W}$. By induction hypothesis, $D.getSet(r)$ coincides in both algorithms beforehand. If $r \succ \bar{r}'$ then $D.getSet(r)$ is not affected by the changes in both algorithms and thus $D.getSet(r)$

coincides in both algorithms after the iteration of the for loop. If $r \preceq \bar{r}'$ then Algorithm 7.1 updates the data structure such that $P \cup V_c$ is added to the set $S_{\geq r}$ (the set returned by $D.getSet(r)$). Now consider Algorithm 7.2. Here the algorithm adds the set $P \cup V_c$ to the set C_x^i that correspond to $\bar{r}'$. As $r \preceq \bar{r}'$ there is an $i \geq 0$ such that r and $\bar{r}'$ coincide on the first i elements and the set $P \cup V_c$ is then contained in the set $S^i_{\geq r}$. That means that the set returned by $D.getSet(r)$ contains the set $P \cup V_c$. Moreover, as only vertices in $P \cup V_c$ are affected by the update, all the vertices that were previously contained in $D.getSet(r)$ are still contained in the set. In other words, Algorithm 7.2 adds $P \cup V_c$ to the set $S_{\geq r}$. That is, the two $D.getSet(r)$ methods coincide also after the iteration of the for-loop for all $r \in \mathscr{W}$. Thus, we have that $D.getSet(r)$ coincides in the two algorithms and the correctness of Algorithm 7.1 extends to Algorithm 7.2. □

Proposition 7.5.3 (Symbolic operations)**.** *Algorithm 7.2 uses $O(d \log n)$ sets with $O(n|\mathscr{W}|)$ symbolic one-step operations and $O(d^2 n|\mathscr{W}| \log n)$ basic set operations.*

Proof. There are $O(n \cdot |\mathscr{W}|)$ iterations if the while-loop at Line 4 (cf. Proposition 7.3.10). Therefore, the number of symbolic one-step operations is $O(n \cdot |\mathscr{W}|)$. In each iteration of the while-loop, the for-loop at Line 8 has d iterations. The basic set operations at Line 10, Line 13, Line 12 and at Line 23 occur $O(d \cdot n \cdot |\mathscr{W}|)$ times. This sums up to a total of $O(d^2 \cdot n \cdot |\mathscr{W}| \log n)$ basic set operations (as each `getSet(r)` requires $O(d \log n)$ basic set operations). By the same amortized argument as in the proof of Proposition 7.3.10, we obtain that the total number of basic set-operations in executions of the inner while-loop is in $O(n|\mathscr{W}|d \log n)$. The number of basic set operations is, thus, in $O(d^2 n|\mathscr{W}| \log n) + O(dn|\mathscr{W}| \log n) = O(d^2 n \cdot |\mathscr{W}| \log n)$. □

Due to Proposition 7.5.2 and Proposition 7.5.3 and the fact that we use only $O(d \log n)$ symbolic space in the modified data structure D, we obtain Theorem 7.5.4.

Theorem 7.5.4. *The winning set of a parity game can be computed in $O(n \cdot |\mathscr{W}|)$ symbolic one-step operations, $O(d^2 n \cdot |\mathscr{W}| \cdot \log n)$ basic set operations, and $O(d \cdot \log n)$ symbolic space.*

Remark 7.5.5. Note that Theorem 7.5.4 achieves bounds similar to Theorem 7.3.12 with a factor $d \cdot \log n$ increase in basic set operations, however, the symbolic space requirement decreases from $O(n)$ to $O(d \cdot \log n)$. In particular, using the bounds as mentioned in Remark 7.4.3, we obtain a set-based symbolic algorithm that requires quasi-polynomially many symbolic one-step and basic set operations, and $O(d \cdot \log n)$ symbolic space, and moreover, when $d \leq \log n$, then the algorithm requires polynomially many symbolic one-step and basic set operations and only poly-logarithmic $O(\log^2 n)$ symbolic space.

Remark 7.5.6. Recall that our AVL-tree data structure potentially requires $O(n)$ non-symbolic space (cf. Remark 7.3.3) in the worst case. The algorithm above reduces the symbolic space requirement to $O(d \cdot \log n)$. We now argue how to reduce the non-symbolic space to the same bound. The main purpose of our AVL-tree data structure

is to (a) avoid storing all sets explicitly and (b) efficiently maintain pointers to the active sets. As we now have a succinct representation of all sets we are only left with (b). For the price of a higher number of basic set operations, we can rid our black box algorithm of the AVL-tree in the implementation of the data structure: We can explicitly compute the active sets in each iteration of the while loop. This increases the number of basic set operations to $O(d \cdot n \cdot |\mathscr{W}|^2 \log n)$ while the other symbolic resource consumption stays the same.

We give the algorithmic details of Remark 7.5.6 in the next section.

7.6 Algorithmic Details for the Reduced Symbolic Space Algorithm 2

In this section, we present a black box parity algorithm that only uses $O(d \cdot \log n)$ symbolic space. In return, the algorithm needs a factor $|\mathscr{W}|$ more basic set operations.

Definition 7.6.1. *An element $r \in \mathscr{W}$ is active if (i) $S_{\geq r} \supset S_{\geq r'}$, for $r \prec r'$ and (ii) if the set $S_{\geq r}$ has changed since the last time r was selected at Line 4 of Algorithm 7.3.*

Key Intuition. Algorithm 7.2 uses the AVL tree to keep track of the $r \in \mathscr{W}$ that need to be potentially processed in the future, i.e., the *active* r in $\mathscr{W}$. Algorithm 7.3 initializes $d \log n + 1$ additional sets, to determine if the set $S_{\geq r}$ was changed since the last time we chose r at Line 4. We call the additional $d \log n + 1$ sets *the data structure O*. In each iteration of Line 4 we update O by copying the content the current set $S_{\geq r}$ to the corresponding set $T_{\geq r}$ in O. Then we do our usual lifting like in Algorithm 7.2 with the set $S_{\geq r}$. To determine the next active r, we iterate from the minimum element in $\mathscr{W}$ to the maximum element. To check condition (ii) in the iteration r, we determine the set $S_{\geq r}$ and $T_{\geq r}$ using the data structures C for $S_{\geq r}$ and O for $T_{\geq r}$ respectively. Notice that when $S_{\geq r} \neq T_{\geq r}$ we changed the set $S_{\geq r}$ since the last time r was picked in the while-loop at Line 4. To check condition (i) we need to check whether $S_r \neq \emptyset$ which would mean that there is no set with higher rank $r' > r$ which already contains all elements of $S_{\geq r}$, i.e., $S_{\geq r} = S_{\geq r'}$. If both conditions are satisfied we return the active set.

In Subsection 7.6.1 we detail the data structure which is used for C and O in Algorithm 7.3. The proofs of Subsection 7.6.2 and Subsection 7.6.3 yield the theorem detailed below.

Theorem 7.6.2. *The winning set of a parity game can be computed without the usage of nonsymbolic space with $O(d \log n)$ symbolic space, $O(dn|\mathscr{W}|^2 \log n)$ basic set operations and $O(n|\mathscr{W}|)$ symbolic one-step operations.*

7.6.1 Data Structure.

The new data structure which is used by D and O supports the following functions

Algorithm 7.3: Parity Algorithm with reduced sets

```
input : Parity Game 𝒫, finite total order (𝒲, ⪯), monotone function lift(·), a
        player z ∈ {𝒞, Õ}
Initialize C with C^i ← V for 0 ≤ i ≤ k
Initialize C with C_c^i ← ∅ for 0 ≤ i ≤ k, c ∈ C
Initialize O with O_c^i ← ∅ for 0 ≤ i ≤ k, c ∈ C ∪ {_}
while r ← getMinActiveSet() do
    S_≥r ← D.getSet(r)
    O.update(r, S_≥r)
    P ← CPre_z(S_≥r)
    for c ∈ C do
        r' ← lift(r, c)
        S_≥r' ← C.getSet(r')
        C.update(r', S_≥r' ∪ (P ∩ V_c))
return S_⊤
function getMinActiveSet()
    S_≥r ← V
    T_≥r ← O.getSet(min)
    for r = min, min+1, ..., ⊤ do
        S_r ← C.getSet2(r)
        if S_r ≠ ∅ and S_≥r ⊃ T_≥r then
            return r;
        T_r ← O.getSet2(r)
        S_≥r ← S_≥r \ S_r
        T_≥r ← T_≥r \ T_r
    return false
```

- getSet(r): returns the set $S_{\geq r}$.
- getSet2(r): returns the set S_r.
- update(r,S): Updates the set $S_{\geq r}$ to be S. Every set $S_{\geq r'}$ with $r' \preceq r$ is updated to $S_{\geq r'} \cup S$.

Implementation of the data structure.

- getSet(r): This function computes the set $S_{\geq r}$ as described in Section 7.5 and returns it.
- getSet2(r,S): This function computes the set S_r as described in Section 7.5 and returns it.
- update(r,S): Computes the update of the set $S_{\geq r}$ with $S \subseteq V$ as described in Section 7.5. Precondition: $S_{\geq r}$ is a subset of S.

7.6.2 Correctness

In this section, we prove the correctness of Algorithm 7.3.

Proposition 7.6.3. *Given a parity game $\mathcal{P}$, Algorithm 7.3 computes the winning set.*

The only difference in Algorithm 7.2 and Algorithm 7.3 is how active $r \in \mathscr{W}$ are obtained. That is, in order to prove the correctness of Algorithm 7.3 it suffices to show that the function getMinActiveSet$(\cdot)$ returns the set of an active $r \in \mathscr{W}$ whenever there exist one, and false otherwise. The function getMinActiveSet$(\cdot)$ returns the minimal[2] $r \in \mathscr{W}$ that satisfies (i) $S_{\geq r} \supset S_{\geq r'}$, for $r \prec r'$ (by the anti-monotonicity of the set it suffices to test this for the direct successor) and (ii) $T_r \subset S_r$. The next lemma shows that these sets are actually the active $r \in \mathscr{W}$.

Lemma 7.6.4. *An element $r \in \mathscr{W}$ is active iff (i) $S_{\geq r} \supset S_{\geq r'}$, for $r \prec r'$ and (ii) $T_{\geq r} \subset S_{\geq r}$*

Proof. We show the claim by induction. C is initialized such that only min is active and all the other $r \in \mathscr{W}$ correspond to the empty set and thus all, but min and $\top$ violate (i). Now as all the sets of O are empty (ii) holds for min but does not hold for $\top$. Hence, the statement is true when first entering the loop.

Now consider an iteration of the loop where $\bar{r}$ is processed and assume the claim is true beforehand. By induction hypothesis $\bar{r}$ is the minimal active $r \in \mathscr{W}$. Notice that the claim is only affected by update operations on O and C. First, we argue that the claim is preserved in Line 6. This operation only affects $r' \preceq \bar{r}$. Notice that $\bar{r}$ is no longer active: We process $\bar{r}$ and violate condition (ii) as we set $T_{\geq \bar{r}} = S_{\geq \bar{r}}$ at Line 6. Let $r' \prec \bar{r}$. As $\bar{r}$ is the minimal active $r \in \mathscr{W}$ we have that r' is inactive. Then, by the induction hypothesis, we have $T_{\geq r'} = S_{\geq r'}$ beforehand and by the anti-monotonicity of the sets in C also after the execution of Line 6. That is, r' is inactive and (ii) is still violated. Second, we show that also the update in Line 11 preserves the claim. Let $\bar{r} \in \mathscr{W}$ be the value computed by the lift function at Line 9. Again, this operation only affects $r' \preceq \bar{r}$. If the update violates condition (i) for a r', i.e. $S_{\geq r'} = S_{\geq \hat{r}}$ for $r' \prec \hat{r}$ then this r' also becomes inactive by the definition of active. Let us assume (i) is not violated and r' was active beforehand. Then, by the induction hypothesis, we have $T_{\geq r'} \subset S_{\geq r'}$ beforehand, and as only vertices are added to $S_{\geq r'}$ the condition (ii) still holds. Let us assume (i) is not violated and r' was inactive beforehand. Then by the induction hypothesis $T_{\geq r'} = S_{\geq r'}$ beforehand, and as (ii) is satisfied afterwards iff $(P \cap V_c) \notin S_{\geq r'}$. Notice that if $(P \cap V_c) \notin S_{\geq r'}$ then r' is changed and r' is active because condition (i) is satisfied by assumption. If $(P \cap V_c) \subset S_{\geq r'}$ then we do not change the set and it stays inactive. Hence, we have that (2) is satisfied iff r' becomes active. □

[2]Notice that the fact that getMinActiveSet$(\cdot)$ returns the minimal active r is only used to consistently maintain the data structure O.

Thus, we have that GETMINACTIVESET(·) returns an active set whenever there exist one, and false otherwise and consequently Algorithm 7.3 computes the winning set of $\mathscr{P}$.

7.6.3 Symbolic Resource consumption

In this subsection, we present the symbolic resource consumption of Algorithm 7.3. First, we present the bounds for basic set operations, symbolic one-step operations, and finally, symbolic space.

Proposition 7.6.5. *Algorithm 7.3 needs $O(dn|\mathscr{W}|^2 \log n)$ basic set operations, $O(n|\mathscr{W}|)$ symbolic one-step operations and $O(d \log n)$ symbolic space.*

Proof. For the $n|\mathscr{W}|$ symbolic one-step operations bound, notice that we increase the rank of at least one vertex in each iteration of Algorithm 7.3 by adding the vertex to a set with a higher rank. We can only do this until every vertex has a rank $|\mathscr{W}|$. Therefore the while-loop of Algorithm 7.3 has $O(n \cdot |\mathscr{W}|)$ iterations. In each iteration, we perform exactly one $CPre_z$ operation.

For the $O(dn|\mathscr{W}|^2 \log n)$ basic set operations bound, recall that the while loop at Line 4 is called $O(n\mathscr{W})$ times. First, the call to the `getActiveSet()` function requires $O(|\mathscr{W}| \log n)$ basic set operations. There is one `update` operation in Line 6 and d `update` operations in the for-loop at Line 11, i.e., each iteration requires $O(d^2 \log n)$ basic set operations for updates. Therefore, we get the bound $O(n|\mathscr{W}| \cdot (|\mathscr{W}| \log n + d^2 \log n) = O(dn|\mathscr{W}|^2 \log n)$ (using $|\mathscr{W}| > d$).

For the $O(d \log n)$ symbolic space bound, notice that we use the data structures O and C, both of them use $d \log n + 1$ symbolic space, and a constant number of additional sets. □

7.7 Conclusion

In this work, we present improved set-based symbolic algorithms for parity games. There are several interesting directions for future work. On the practical side, practical implementation and experiments with case studies, especially for the algorithm presented in Section 7.3 instantiated with either the ordered approach or the succinct progress measure, is an interesting direction. On the theoretical side, recent work [74] has established lower bounds for symbolic algorithms for graphs, and whether lower bounds can be established for symbolic algorithms for parity games is another interesting direction for future work.

CHAPTER 8

Symbolic Time and Space Tradeoffs for Probabilistic Verification

In this chapter, we present improved symbolic algorithms for the MEC decomposition of an MDP and parity objectives in MDPs.

8.1 Introduction

The verification of probabilistic systems, e.g., randomized protocols, or agents in uncertain environments like robot planning is a fundamental problem in formal methods. We study a classical graph algorithmic problem that arises in the verification of probabilistic systems and present a faster symbolic algorithm for it. We start with the description of the graph problem and its applications, then describe the symbolic model of computation, then previous results, and finally our contributions.

Applications. In verification of probabilistic systems, the classical model is called *Markov decision processes (MDPs)* [146], where there are two types of vertices. The vertices in V_1 are the regular vertices in a graph algorithmic setting and the vertices in V_R represent random vertices. MDPs are used to model and solve control problems in systems such as stochastic systems [129], concurrent probabilistic systems [113], planning problems in artificial intelligence [195], and many problems in the verification of probabilistic systems [22]. The MEC decomposition problem is a central algorithmic problem in the verification of probabilistic systems [22, 113] and it is a core component in all leading tools of probabilistic verification [117, 168]. Some key applications are as follows: (a) the almost-sure reachability problem can be solved in linear time given the MEC decomposition [75]; (b) verification of MDPs

wrt ω-regular properties requires MEC decomposition [22, 78, 112, 113]; (c) algorithmic analysis of MDPs with quantitative objectives as well as the combination of ω-regular and quantitative objectives requires MEC decomposition [43, 82, 97]; and (d) applying learning algorithms to verification requires MEC decomposition computation [114, 164].

Symbolic model and algorithms. In verification, a system consists of variables, and a state of the system corresponds to a set of valuations, one for each variable. This naturally induces a directed graph: vertices represent states and the directed edges represent state transitions. However, as the transition systems are huge they are usually not explicitly represented during their analysis. Instead they are *implicitly represented* using e.g., binary-decision diagrams (BDDs) [51, 52]. An elegant theoretical model for algorithms that works on this implicit representation, without considering the specifics of the representation and implementation, has been developed, called *symbolic algorithms* (see e.g. [55, 79, 105, 107, 108, 132, 211]). A symbolic algorithm is allowed to use the same mathematical, logical, and memory access operations as a regular RAM algorithm, except for the access to the input graph: It is not given access to the input graph through an adjacency list or adjacency matrix representation but instead *only* through two types of *symbolic operations*:

1. *One-step operations Pre and Post:* Each *predecessor* PRE (resp., *successor* POST) operation is given a set X of vertices and returns the set of vertices Y with an edge to (resp., edge from) some vertex of X.
2. *Basic set operations:* Each basic set operation is given one or two sets of vertices or edges and performs a union, intersection, or complement on these sets.

Symbolic operations are more expensive than the non-symbolic operations and thus *symbolic time* is defined as the *number of symbolic operations* of a symbolic algorithm. One unit of space is defined as one set (not the size of the set) due to the implicit representation as a BDD. We define *symbolic space* of a symbolic algorithm as the maximal *number of sets* stored simultaneously. Moreover, as the symbolic model is motivated by the compact representation of huge graphs, we aim for symbolic algorithms that require sub-linear space.

Previous results and main open question. We summarize the previous results and the main open question. We denote by $|V| = n$ and $|E| = m$ the number of vertices and edges, respectively.

- *Standard RAM model algorithms.* The computation of the MEC (aka controllable recurrent set in early works) decomposition problem has been a central problem since the work of [8, 112, 113]. The classical algorithm for this problem requires $O(n)$ SCC decomposition calls, and the running time is $O(nm)$. The above bound was improved to (a) $O(m\sqrt{m})$ in [78] and (b) $O(n^2)$ in [91]. While the above algorithms are deterministic, a randomized algorithm with expected almost-linear $\widetilde{O}(m)$ running time has been presented in [89].
- *Symbolic algorithms.* The symbolic version of the classical algorithm for MEC decomposition requires $O(n)$ symbolic SCC computation. Given the $O(n)$

symbolic operations SCC computation algorithm from [133], we obtain an $O(n^2)$ symbolic operations MEC decomposition algorithm, which requires $O(\log n)$ symbolic space. A symbolic algorithm, based on the algorithm of [78], was presented in [79], which requires $O(n\sqrt{m})$ symbolic operations and $O(\sqrt{m})$ symbolic space.

The classical algorithm from the 1990s with the linear symbolic-operations SCC decomposition algorithm from 2003 gives the $O(n^2)$ symbolic operations bound, and since then the main open question for the MEC decomposition problem has been whether the worst-case $O(n^2)$ symbolic operations bound can be beaten.

Our contributions.

1. In this work we answer the open question in the affirmative. Our main result presents a symbolic operation and symbolic space trade-off algorithm that for any $0 < \epsilon \leq 1/2$ requires $\widetilde{O}(n^{2-\epsilon})$ symbolic operations and $\widetilde{O}(n^{\epsilon})$ symbolic space. In particular, our algorithm for $\epsilon = 1/2$ requires $\widetilde{O}(n^{1.5})$ symbolic operations and $\widetilde{O}(\sqrt{n})$ symbolic space, which improves both the symbolic operations and symbolic space of [79].

2. We also show that our techniques extend beyond MEC computation and is also applicable to almost-sure winning (probability-1 winning) of ω-regular objectives for MDPs. We consider parity objectives which are cannonical form to express ω-regular objectives. For parity objectives with d priorities the previous symbolic algorithms require $O(d)$ calls to MEC decomposition; thus leading to bounds such as (a) $O(n^2 \cdot d)$ symbolic operations and $O(\log n)$ symbolic space; or (b) $O(n\sqrt{m} \cdot d)$ symbolic operations and $O(\sqrt{m})$ symbolic space. In contrast we present an approach that requires $O(\log d)$ calls to MEC decomposition, and thus our algorithm requires $\widetilde{O}(n^{2-\epsilon})$ symbolic operations and $\widetilde{O}(n^{\epsilon})$ symbolic space, for all $0 < \epsilon \leq 1/2$. Thus we improve the time-space product from $\widetilde{O}(n^2 d)$ to $\widetilde{O}(n^2)$.

Technical contribution. Our main technical contributions are as follows: (1) We use a separator technique for the decremental SCC algorithm from [98]. However, while previous MEC decomposition algorithms for the standard RAM model (e.g. [89]) use ideas from decremental SCC algorithms, data-structures used in decremental SCC algorithms of [31, 98] such as Even-Shiloach trees [127] have no symbolic representation. A key novelty of our algorithm is that instead of basing our algorithm on a decremental algorithm we use the incremental MEC decomposition algorithm of [78] along with the separator technique. Moreover, the algorithms for decremental SCC of [31, 98] are randomized algorithms, in contrast, our symbolic algorithm is deterministic. (2) Since our algorithm is based on an incremental algorithm approach, we need to support an operation of collapsing ECs even though we do not have access to the graph directly (e.g. through an adjacency list representation), but only have access to the graph through symbolic operations. (3) All MEC algorithms in the classic model first decompose the graph into its SCCs and then run on each

SCC. However, to achieve sub-linear space we cannot store all SCCs, and we show that our algorithm has a tail-recursive property that can be utilized to achieve sub-linear space. With the combination of the above ideas, we beat the long-standing $O(n^2)$ symbolic operations barrier for the MEC decomposition problem, along with sub-linear symbolic space.

Implications. Given that MEC decomposition is a central algorithmic problem for MDPs, our result has several implications. The two most notable examples in probabilistic verification are: (a) almost-sure reachability objectives in MDPs can be solved with $\widetilde{O}(n^{1.5})$ symbolic operations, improving the previous known $O(n\sqrt{m})$ symbolic operations bound; and (b) almost-sure winning sets for canonical ω-regular objectives such as parity and Rabin objectives with d-colors can be solved with $O(d)$ calls to the MEC decomposition followed by a call to almost-sure reachability [8, 69], and hence our result gives an $\widetilde{O}(dn^{1.5})$ symbolic operations bound improving the previous known $O(dn\sqrt{m})$ bound.

8.2 Algorithmic Tools

In this section, we present various algorithmic tools that we use in our algorithms.

8.2.1 Symbolic SCCs Algorithm

In [133] and [74] symbolic algorithms and lower bounds for computing the SCC-decomposition are presented. Let D be the diameter of G and let D_C be the diameter of the SCC C. The set $\texttt{SCCs}(G)$ is a family of sets, where each set contains one SCC of G. The upper bounds are summarized in Theorem 8.2.1.

Theorem 8.2.1 ([74, 133]). *The SCCs can be computed in* $\Theta(\min(n, D|\texttt{SCCs}(G)|, \sum_{C\in\texttt{SCCs}(G)}(D_C+1)))$ *symbolic operations and* $\widetilde{O}(1)$ *symbolic space.*

Note that the SCC algorithm of [133] can be easily adapted to accept a starting vertex which specifies the SCC computed first. Also, SCCs are output when they are detected by the algorithm and can be processed before the remaining SCCs of the graph are computed.

We write $\text{SCCFind}(V', s, P)$ when we refer to the above described algorithm for computing the SCCs of the subgraph with vertices $V' \subseteq V$ and using $s \in V$ as the starting vertex for the algorithm in the MDP $P = (V, E, \langle V_1, V_R\rangle, \delta)$.

8.2.2 Symbolic Random Attractors

Given a set of vertices T in an MDP P, the random attractor $\mathsf{Attr}^P_R(T)$ is a set of vertices consisting of (1) T, (2) random vertices with an edge to some vertex in $\mathsf{Attr}^P_R(T)$, (3) player-1 vertices with all outgoing edges in $\mathsf{Attr}^P_R(T)$. Formally, given an MDP $P = (V, E, \langle V_1, V_R\rangle, \delta)$, let T be a set of vertices. The random attractor $\mathsf{Attr}^P_R(T)$ of T is defined as: $\mathsf{Attr}^P_R(T) = \bigcup_{i\geq 0} A_i$ where $A_0 = T$ and $A_{i+1} = A_i \cup$

$(Pre_E(A_i) \setminus (V_1 \cap Pre_E(V \setminus A_i)))$ for all $i > 0$. We sometimes refer to A_i as the i-th level of the attractor.

Lemma 8.2.2 ([93]). *The random attractor* $\mathsf{Attr}_R^P(T)$ *can be computed with at most* $O(|\mathsf{Attr}_R^P(T) \setminus T| + 1)$ *many symbolic operations.*

The lemma below establishes that the random attractor of random vertices with edges out of a strongly connected set is not included in any end-component and that it can be removed without affecting the ECs of the remaining graph. Hence, we use the lemma to identify vertices that do not belong to any EC. The proof is analogous to the proof in Lemma 2.1 of [78].

Lemma 8.2.3 ([78]). *Let* $P = (V, E, \langle V_1, V_R \rangle, \delta)$ *be an MDP. Let* C *be a strongly connected subset of* V*. Let* $U = \{v \in C \cap V_R \mid Out(v) \cap (V \setminus C) \neq \emptyset\}$ *be the random vertices in* C *with edges out of* C*. Let* $Z = \mathsf{Attr}_R^P(U) \cap C$*. Then, for all non-trivial EC's,* $X \subseteq C$ *in* P *we have* $Z \cap X = \emptyset$*.*

8.2.3 Separators

Given a strongly connected set of vertices X with $|X| = n$, a separator is a non-empty set $T \subseteq X$ such that the size of the SCC in the graph induced by $X \setminus T$ is small. More formally, we call $T \subseteq X$ a q-separator if each SCC in the subgraph induced by $X \setminus T$ has at most $n - q \cdot |T|$ vertices. For example, if we compute a $q = \sqrt{n}$-separator, the SCCs in the subgraph induced by $X \setminus T$ have size $\leq \sqrt{n}$ as $|T|$ is non-empty. In [98], the authors present an algorithm that computes a q-separator when the diameter of X is large. We briefly sketch the symbolic version of this algorithm.

The procedure Separator(X, γ) computes a $q = \lfloor \gamma/(2 \log n) \rfloor$-separator T when X has diameter at least γ:

1. Let $q \leftarrow \lfloor \gamma/(2 \log n) \rfloor$.
2. Try to compute a BFS tree K of either X or the reversed graph of X of depth at least γ with an arbitrary vertex $v \in X$ as root. In the symbolic algorithm, we build the BFS trees with $Pre(\cdot)$ and $Post(\cdot)$ operations.
3. If the BFS trees of both X and the reversed graph X with root v have less than γ levels then return $\emptyset$. Note that the diameter of X is then $\leq 2\gamma$.
4. Let layer L_i of K be the set of vertices $L_i \subseteq X$ with distance i from v. Due to [98, Lemma 6], there is a certain layer L_i of the BFS tree K which is a q-separator. Intuitively, they argue that removing the layer L_i of K separates X into two parts: (a) $\bigcup_{j<i} L_j$ and (b) $\bigcup_{j>i} L_j$ which cannot be strongly connected anymore due to the fact that K is a BFS tree. They show that one can always efficiently find a layer L_i such that both part (a) and part (b) are small.
5. We efficiently find the layer L_i while building K.

The detailed symbolic implementation of SEPARATOR(X, γ) is illustrated in Algorithm 17. The following lemmas summarize useful properties of SEPARATOR(X, γ).

Algorithm 8.1: Computing the separator

```
1  Procedure SEPARATOR(X, γ)
2      q ← ⌊γ/(2 log n)⌋, v ← Pick(X), i ← 0, K ← {v}, c ← 0, L ← ∅, R ← ∅
3      while (Post_E(K) ∩ X) ⊈ K do
4          if q ≤ i ≤ γ/2 then
5              Z ← Post_E(K) \ K ∩ X
6              if L = ∅ and |Z| ≤ 2^{i/q−1} then L ← Z
7          if γ/2 ≤ i ≤ γ − q then
8              Z ← Post_E(K) \ K ∩ X
9              if |Z| ≤ 2^{(γ−i)/q−1} then R ← Z
10         if i ≤ γ/2 then c ← c + |Post_E(K) \ K ∩ X|
11         K ← K ∪ (Post_E(K) ∩ X)
12         i ← i + 1
13     if i < γ then
14         i ← 0, K ← {v}, c ← 0, L ← ∅, R ← ∅
15         while i ≤ γ and (Pre_E(K) ∩ X) ⊈ K do
16             if q ≤ i ≤ γ/2 then
17                 Z ← Pre_E(K) \ K ∩ X
18                 if L = ∅ and |Z| ≤ 2^{i/q−1} then L ← Z
19             if γ/2 ≤ i ≤ γ − q then
20                 Z ← Pre_E(K) \ K ∩ X
21                 if |Z| ≤ 2^{(γ−i)/q−1} then R ← Z
22             if i ≤ γ/2 then c ← c + |Pre_E(K) \ K ∩ X|
23             K ← K ∪ (Pre_E(K) ∩ X)
24             i ← i + 1
25         if i < γ then
26             return ∅; // diam(X) < 2γ
27     if c < |X|/2 then return L
28     else return R
```

Observation 8.2.4 ([98]). *A q-separator S of a graph G with $|V| = n$ vertices contains at most $\frac{n}{q}$ vertices, i.e., $|S| < \frac{n}{q}$.*

Lemma 8.2.5 ([98]). *Let X be a strongly connected set of vertices with $|X| = k$, let $r \in X$, and let γ be an integer such that $q = \lfloor \gamma/(2 \log k) \rfloor \geq 1$. Then* SEPARATOR$(X, \gamma)$ *computes a $\lfloor \gamma/(2 \log k) \rfloor$-separator if there exists a vertex $v \in X$ where the distance between r and v is at least γ. If no such vertex v exists, then* SEPARATOR(X, γ) *returns the empty set.*

The following lemma bounds the symbolic resources of the algorithm.

Lemma 8.2.6. *Algorithm 17 runs in $O(|X|)$ symbolic operations and uses $O(1)$ symbolic space.*

Proof. The bound on the symbolic operations of the while loop at Line 3 is clearly in $O(|X|)$ because we perform $Post_E(K) \cap X$ operations until the set K fully contains X or $Post_E(K) \cap X$ is fully contained in K. Note that we only perform a constant amount of symbolic work in the body of the while-loop. As each $Post_E(K) \cap X$ operation adds at least one vertex to X until termination, we perform $O(|X|)$ many operations in total. A similar argument holds for the while-loop at Line 15. Note that we use a constant amount of sets in Algorithm 17 which implies $O(1)$ symbolic space. □

8.3 Symbolic MEC decomposition

In this section, we first define how we collapse end-components. Then we present the algorithm for the symbolic MEC decomposition.

8.3.1 Collapsing End-components

A key concept in our algorithm is to collapse a detected EC $X \subseteq V$ of an MDP $P = (V, E, \langle V_1, V_R \rangle, \delta)$ to a single player-1 vertex $v \in X$ in order to speed up the computation of end-components that contain X. Notice that we do not have access to the graph directly, but only have access to the graph through symbolic operations.

We define the collapsing (see Algorithm 18) of an EC X as picking a player-1 vertex $v \in X$, directing all the incoming edges X to v, directing the outgoing edges of X from v and removing all edges to and from vertices in $X \setminus v$. Also, we do not include edges going from X to v. For vertices not in X, we have that they are in a non-trivial MEC in the modified MDP iff they are in a non-trivial MEC in the original MDP. We denote with $\mathsf{EC}(P)$ the set of sets with all end-components in P. The following lemma summarizes the property as observed in [78].

Algorithm 8.2: Collapse an EC X into single vertex

```
1 Procedure CollapseEC(X, P)
2     if X ∩ V_1 ≠ ∅ then
3         v ← Pick(X ∩ V_1)
4         E ← E ∪ (((Pre_E(X) \ X) × {v}) ∪ ({v} × (Post_E(X) \ X))); // v gets
              edges of X
5         E ← E \ ((V × (X \ v)) ∪ ((X \ v) × V)) ;// Remove all other edges
              of X
6         return {v}
7     else E ← E \ (X × X) // Remove all edges of X
```

Lemma 8.3.1. *For MDP $P = (V, E, \langle V_1, V_R \rangle, \delta)$ with $X \in \mathsf{EC}(P)$ and the MDP P' that results from collapsing X to $v \in X$ with* CollapseEC(X, P) *we have: (a) for $D \subseteq V \setminus X$ we have $D \in \mathsf{EC}(P)$ iff $D \in \mathsf{EC}(P')$; (b) for $D \in \mathsf{EC}(P)$ with $D \cap X \neq \emptyset$ we have $(D \setminus X) \cup \{v\} \in \mathsf{EC}(P')$; and (c) for $D \in \mathsf{EC}(P')$ with $v \in D$ we have $D \cup X \in \mathsf{EC}(P)$.*

Algorithm 8.3: Computing the MEC decomposition of an MDP

```
1 Procedure SymbolicMEC((P' = (V', E', ⟨V'_1, V'_R⟩, δ'), γ))
2     M ← ∅, MECs ← ∅, P ← P'
3     while C ← SCCFind(V, ∅, P') do // compute vertices in
        non-trivial MECs of C
4         M ← M ∪ SymMEC(C, γ, P)// uses P instead of P'
5     while C ← SCCFind(M, ∅, P') do // compute non-trivial MECs
6         MECs ← MECs ∪ {C}
7     return MECs
```

Remark 8.3.2. Note that we modify the set of edges E of P using basic set operations only and that such operations are supported by all standard symbolic tools like BDDs. After the update, we then use the new set E for all subsequent $Pre(.)$ and $Post(.)$ operations.

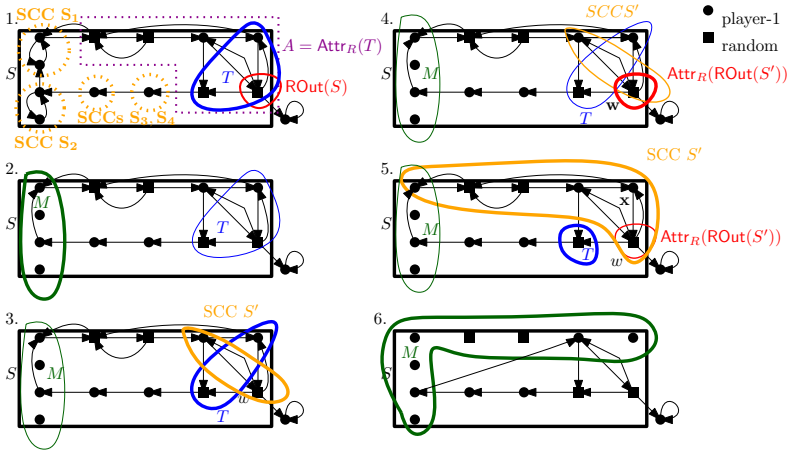
8.3.2 Algorithm Description

The input to our algorithm is an MDP $P' = (V', E', \langle V'_1, V'_R \rangle, \delta')$. In the first stage, we compute the set $M \subseteq V$ of all vertices that are in a non-trivial MEC, and then, in the second stage, we use this set M to compute the MECs with an SCC algorithm (see Algorithm 8.3).

In the first stage, we iteratively compute the SCCs $C_1, \ldots, C_\ell$ of P' and immediately apply SymMEC(·) (cf. Algorithm 8.4) to C_i to compute the vertices M_i of C_i that are in a non-trivial MEC. The set M is the union over these sets, i.e., $M = \bigcup_{1 \leq i \leq \ell} M_i$. SymMEC(·) applies collapsing operations and thus modifies the edge set of the MDP. We thus hand a copy P of the original MDP P' to SymMEC(·). Finally, to obtain all non-trivial MECs in P' we restrict the graph to the vertices in M and compute the SCCs which correspond to the MECs. Note that trivial MECs are player-1 vertices that are not contained in any non-trivial MEC and thus can be simply computed by iterating over the vertices of $V_1 \setminus M$.

In the following we focus on the function SymMEC(·) which is the core of our algorithm. To this end, we introduce the operation $ROut(S) = Pre_E(V \setminus S) \cap (S \cap V_R)$, that computes the set of random vertices in S with edges to $V \setminus S$.

The function SymMEC(·). SymMEC(·) works recursively: The input is a set S of strongly connected vertices and a parameter γ for separator computations that is fixed over all recursive calls. The main idea is to compute a separator to divide



the original graph into smaller SCCs, recursively compute the vertices which are in MECs of the smaller SCCs, and then compute the MECs of the original graph by incrementally adding the vertices of the separator back into the recursively computed MEC decomposition. In each recursive call, we first check whether the given set S is larger than one (if not it cannot be a non-trivial MEC) and then if S is a non-trivial EC by checking if $ROut(S) = \emptyset$. If S is a non-trivial EC we collapse S by calling Algorithm 18 and the algorithm then returns $M = S$. If $ROut(S) \neq \emptyset$, S is not a non-trivial EC but it may contain nontrivial ECs of C. We then try to compute a balanced separator T of S which is nonempty only if the diameter of S is large enough ($\geq 2\gamma$). We further distinguish between the two cases: In the first case, we succeed to compute the balanced separator and we recurse on the strongly connected components in $\mathtt{SCCs}(S \setminus \mathsf{Attr}_R^P(T))$. After computing and collapsing the ECs in $\mathtt{SCCs}(S \setminus \mathsf{Attr}_R^P(T))$ we incrementally add vertices of T to S and compute the ECs of $S \setminus T$ until T is empty. In each incremental step, we add one vertex v to $S \setminus T$ and find the SCC of v in $S \setminus T$. If the random attractor of $ROut(S')$ (note that this computation now considers all vertices in P) does not contain the whole SCC S' we are able to prove that we can identify a new EC of C.

Otherwise, the vertex v does not create a new EC in $S \setminus T$. In the second case where we fail to compute the balanced separator we know that the diameter of S is small ($< 2\gamma$). We remove the random attractor X of $ROut(S)$ (this set cannot contain ECs due to Lemma 8.2.3), recompute the strongly connected components in the set $S \setminus X$ and recurse on each of them one after the other.

Figure 8.3.2 illustrates one recursive call of the SymMEC(S, γ, P) function: In the first step, we compute that $ROut(S)$ is nonempty, i.e., S cannot be a MEC. We then successfully compute a separator T with parameter γ of the strongly connected graph S and recurse on the SCCs of $S \setminus \mathsf{Attr}_R^P(T)$ (orange dottet circles). In the second step, we recursively identify two nontrivial MECs and add them to M. Next, we perform the incremental iterations for all vertices in T to identify the MECs containing the vertices in $\mathsf{Attr}_R^P(T)$, (i.e., also in T). In step three, we pick $w \in T$,

remove it from T and compute its SCC $S' = S \setminus T$. In step four, we compute the random attractor of $ROut(S')$, i.e., $\mathsf{Attr}_R^P(ROut(S')) = \{w\}$. Because it contains w, there is no new MEC containing w. In step five, we remove x from T and compute its SCC S'. This time, the random attractor of $ROut(S')$, i.e., $\mathsf{Attr}_R^P(ROut(S'))$, does not contain x and the SCC S' of x without $\mathsf{Attr}_R^P(ROut(S'))$ has size greater one. We add the vertices in $S' \setminus \mathsf{Attr}_R^P(ROut(S'))$ to M. In step six, we do the last incremental iteration, identify no new nontrivial MECs and return M.

Algorithm 8.4 illustrates the pseudocode of SymMEC(S, γ, P).

Remark 8.3.3. Note that our symbolic algorithm does not require randomization which is in contrast to the best known MEC decomposition algorithms for the standard RAM model [31]. The latter algorithms rely on decremental SCCs algorithms that are randomized as they maintain ES-trees [127] from randomly chosen centers. Instead, our symbolic algorithm relies on a deterministic incremental approach.

8.3.3 Correctness

We first consider the correctness of Algorithm 8.4 which will then imply the correctness of Algorithm 8.3. To this end, consider an MDP $P = (V, E, \langle V_1, V_R \rangle, \delta)$ and an SCC C of P. The non-trivial end-components of a subset S of V are given by $\mathsf{EC}(S)$ and the set of vertices that appear in non-trivial (maximal) ECs of C are denoted by $M_C = \bigcup_{Q \in \mathsf{EC}(C)} Q$.

We first observe that in all calls to SymMEC(S, γ, P) (see Algorithm 8.4.) the set S is strongly connected.

Lemma 8.3.4. *For a strongly connected set $S \subseteq C$ we have that in the computation of* SymMEC(S, γ, P) *for all the calls* SymMEC(S_j, γ, P) *the set S_j is strongly connected and $S_j \subseteq S$.*

Proof. If $|S| = 1$ or $ROut(S) = \emptyset$ there are no recursive calls the statement is true. Now consider the case where $T \neq \emptyset$. At Line 11, we consider S_j which is an SCC in the graph $S \setminus (T \cup A)$, obviously a subset of S and strongly connected. Now consider the case where $T = \emptyset$. At Line 27 we consider S_j which is an SCC of $S \setminus X$, obviously a subset of S and strongly connected. □

Let M be the set returned by SymMEC(S, γ, P). Note that the ultimate goal of Algorithm 8.4 is to compute M_C for a given SCC C of P'. The next lemma shows that every call of Algorithm 8.4 with a strongly connected set S returns the set $M_S = \bigcup_{Q \in \mathsf{EC}(S)} Q$ and collapses all ECs $Q \in \mathsf{EC}(S)$ in P.

Lemma 8.3.5. *Let S be a strongly connected set of vertices and M be the set returned by* SymMEC(S, γ, P)*, then for all non-trivial ECs $Q \in \mathsf{EC}(S)$ in P' we have $Q \subseteq M$ and Q is collapsed in P.*

Proof. Let $Q \in \mathsf{EC}(C)$ with $Q \subseteq S$. We prove the statement by induction on the size of S. For the base case, i.e., if S is empty or contains only one vertex, Algorithm 8.4 returns the empty-set at Line 3 and thus the condition holds.

Algorithm 8.4: Compute the ECs of an SCS recursively

```
1  Procedure SymMEC(S, γ, P)
2      M ← ∅
3      if |S| ≤ 1 then return M
4      if ROut(S) = ∅ then // check if S is an EC
5          CollapseEC(S, P)
6          return M ∪ S
7      (T = {v_1, ..., v_t}) ← Separator(S, γ)
8      if T ≠ ∅ then // T ≠ ∅ if diam(S) ≥ 2γ
9          A ← Attr_R^P(T) ∩ S
10         while S_j ← SCCFind(S \ A, ∅, P) do
11             M ← M ∪ SymMEC(S_j, γ, P)
12         while T ≠ ∅ do // incremental EC detection for v ∈ T
13             v ← Pick(T)
14             T ← T \ {v}
15             S' ← SCCFind(S \ T, {v}, P)
16             if |S'| = 1 then continue // EC is trivial
17             Z ← Attr_R^P(ROut(S')); // consider vertices in P
18             S' ← S' \ Z
19             if S' ≠ ∅ then // S' contains a non-trivial EC
20                 U ← SCCFind(S', {v}, P)
21                 CollapseEC(U, P)
22                 M ← M ∪ U
23         return M
24     else // T = ∅ and thus diam(S) < 2γ
25         X ← Attr_R^P(ROut(S))
26         while S_j ← SCCFind(S \ X, ∅, P) do
27             M ← M ∪ SymMEC(S_j, γ, P)
28         return M
```

For the inductive step, let $|S| > 1$. If $Q = S$, we detect it at Line 4 and return S at Line 6. Then we collapse the set of vertices at Line 5. Thus, M contains Q and the claim holds. We distinguish two cases concerning T at Line 7: First if we have non-empty T also the random attractor A of T at Line 9 is non-empty. Thus the SCCs of $S \setminus A$ are strictly smaller than S. Thus, we can use the induction hypothesis to stipulate that if Q is contained in one of the SCCs computed at Line 10 we have $Q \subseteq M$ after the while-loop at Lines 10-11. Additionally, by the induction hypothesis, they are collapsed into a player-1 vertex. Also, if Q contains a subset $Q' \subset Q$ that is a non-trivial EC and contained in one of the SCCs of $S \setminus A$, we have that $Q' \subseteq M$ after the while-loop at Lines 10-11. Again, by induction hypothesis, Q' is collapsed into a player-1 vertex. We proceed by proving that the separator T must contain one of the vertices of Q.

Claim 8.3.6. *For any non-trivial EC $Q \in \mathsf{EC}(S)$ which is not fully contained in one of the SCCs computed at Line 10 we have $T \cap Q \neq \emptyset$.*

Proof. Let Q be an arbitrary EC in $\mathsf{EC}(S)$. Observe that if Q is not fully contained in an SCC of $S \setminus A$ (computed at Line 10), we have $Q \cap A \neq \emptyset$. We now show that $Q \cap T \neq \emptyset$: Assume the contrary, i.e., $Q \cap T = \emptyset$. Consequently, Q is a strongly connected set in $S \setminus T$ and not fully contained in an SCC of $S \setminus A$. Also, Q contains a vertex of $A \setminus T$. Let $v \in Q \cap (A \setminus T)$ such that there is no $v' \in Q \cap (A \setminus T)$ which is on a lower level of the attractor A. If $v \in V_1$, by the definition of the attractor and the above assumption, all its outing edges leave the set Q, which is in contradiction to Q being strongly connected. Thus we have $v \in V_R$ and, by the definition of the attractor, it must have an edge to a smaller level and by the above assumption the target vertex is not in Q. That is Q has an outgoing random edge which contradicts the assumption that Q is an EC. □

The following claim shows an invariant for the while-loop at Line 12.

Claim 8.3.7. *For iteration $j \geq 0$ of the while-loop at Line 12 holds: There are no non-trivial ECs $E \subseteq S \setminus T$ in P.*

Proof. The induction base $j = 0$ holds for the following reasons: $A \setminus T$ cannot include Q due to Claim 8.3.6. $S \setminus A$ cannot contain Q due to the fact that Q must contain a vertex in T and that we collapsed the ECs contained in SCCs computed at Line 10. For the induction step, assume that there are no non-trivial ECs in $S \setminus T$ before iteration $j = \ell$. When we include one vertex v of T into S at Line 13 there can only be one new non-trivial EC Q_v in S, i.e., the one including v, as the remaining graph is unchanged. We argue that if such an EC Q_v exists it is removed before iteration $\ell + 1$ of the while-loop at Line 12: The EC Q_v must be strongly connected, and thus it is contained in the SCC S' of v at Line 15. Note, that if $|S'| = 1$, the new EC is trivial, which concludes the proof. Otherwise, we compute Z at Line 17, which does not contain a vertex of Q_v by Lemma 8.2.3 and remove it from S'. If S' is non-empty, it contains exactly one non-trivial SCC as we argued above. That is the SCC Q_v and we thus add Q_v to M and collapse it. As a consequence there is no non-trivial EC in $S \setminus T$ left. Thus the claim holds. □

It remains to show that Algorithm 8.4 finds the EC Q which contains vertices in $S \setminus M$. Consider the first iteration where $Q \subseteq S \setminus T$. We show that $Q \setminus M = \emptyset$ after this iteration: Because Q might contain non-trivial ECs Q'_i that were already collapsed due to the recursive call, or due to prior iteration of the while-loop at Line 12, it follows from Lemma 8.3.1 that there exists an EC $Q' \subseteq Q$ in P such that $Q' \supset (Q \setminus M)$ and Q' contains one or more vertices corresponding to the collapsed sub-ECs of Q. Due to Claim 8.3.7 we have that Q' is collapsed in the current iteration and thus Q' is added to M, i.e., we have that $Q \subseteq M$ and that Q is collapsed.

Now consider the case $T = \emptyset$ (i.e., $\mathsf{diam}(S) < 2\gamma$). As $ROut(S) \neq \emptyset$ we have that the set X computed at Line 25 is non-empty. Note that Q cannot contain a

vertex in X by Lemma 8.2.3 and the fact that S is strongly connected by Lemma 8.3.4. Consequently, Q is contained in one of the SCCs of $S \setminus X$. Note that each such SCC has size strictly smaller than S due to the fact that X is non-empty. The claim then holds by the induction hypothesis. This completes the proof of Lemma 8.3.5. □

By the above lemma, we have that the algorithm finds all ECs. We next show that all vertices added to M are contained in some EC.

Lemma 8.3.8. *The set M is a subset of M_S, i.e., $M \subseteq \bigcup_{Q \in \mathsf{EC}(S)} Q$.*

Proof. We show the claim by induction over the size of S. For the base case, i.e., $|S| \leq 1$ the claim holds trivially because we return the empty set at Line 3 and any set of size less than two only contains trivial ECs. For the inductive step, let $|S| > 1$. For the return statement at Line 6 we argue as follows: Because S has no random vertices with edges out of S (considering P', Line 4), S is a non-trivial EC. Thus, in Line 6 we correctly return S.

For the case $T \neq \emptyset$ we argue as follows: Let T, A be the separator and its attractor as computed in Line 7 and Line 9. Note that for all SCCs S_j in the graph $S \setminus A$ we have $|S_j| < |S|$ because $|A| > 0$. Thus, by induction hypothesis, all vertices added in the recursive call at Line 11 are in M_S. We claim that for each iteration of the while loop in Line 12 we add only non-trivial ECs to M: Note that by Claim 8.3.7 there is no non-trivial EC at the start of the while loop in Line 12. Let v be the vertex we choose to remove from T at Line 13. Let S_v be the SCC of v computed at Line 15. If $|S_v| = 1$ we continue to the next iteration without adding anything to M and the claim holds. Otherwise, there are two cases based on the computation of the attractor Z of random vertices with edges out of S_v at Line 17: If Z contains S_v, we do not add vertices to M and the claim holds. If $S_v \setminus Z \neq \emptyset$ we add the SCC S'_v of v in $S_v \setminus Z$ to M. It remains to show that the non-trivial SCC S'_v as computed at Line 18 is a non-trivial EC: Note that for all $v \in S'_v \cap V_1$ we have $Out(v) \cap S'_v \neq \emptyset$, otherwise $v \in Z$. Also, for all $u \in S'_v \cap V_R$ we must have $Out(u) \subseteq S'_v$, otherwise $u \in Z$. Thus, S_v is an EC.

For the case $T = \emptyset$ note that each SCC found at Line 26 must be of size strictly less than $|S|$ because $|X| \geq 1$. Thus, by induction hypothesis, the vertices added at Line 27 are in M_S. □

Lemma 8.3.5 and Lemma 8.3.8 imply the correctness of Algorithm 8.4 and Algorithm 8.3.

Proposition 8.3.9 (Correctness)**.** *Given an SCC S of an MDP, Algorithm 8.4 returns the set $M_C = \bigcup_{Q \in \mathsf{EC}(C)} Q$, i.e., the set of vertices that are contained in a non-trivial MEC of S.*

Proposition 8.3.10 (Correctness)**.** *Given an MDP P', Algorithm 8.3 returns the set of non-trivial MECs of P'.*

Proof. Due to Proposition 8.3.9 M contains all vertices in nontrivial ECs. It remains to show that the nontrivial MECs are the SCCs of $V \cap M$. By definition, each nontrivial EC is strongly connected. Towards a contradiction assume that a nontrivial MEC Q is not an SCC but part of some larger SCC C of $V \cap M$ (there is an SCC which is not a MEC). But then, by the definition of M, we have that C is the union of several MECs, i.e., it is strongly connected and has no random outgoing edges and thus C is an EC. This is in contradiction to Q being a MEC. Thus, each nontrivial MEC is an SCC and, by the definition of M, the union of the MECs covers M. Thus there are no further SCCs. □

8.3.4 Symbolic Operations Analysis

We first bound the total number of symbolic operations for computing the separator T at Line 7, recursing upon the SCCs $S \setminus T$ at Line 11 and adding the vertices in T back to compute the rest of the ECs of S at Lines 12–23 during all calls to Algorithm 8.4.

Lemma 8.3.11. *The total number of symbolic operations of Lines 7–23 in all calls to Algorithm 8.4 is in* $O\left(\frac{n^2}{\lfloor \gamma/(2 \log n) \rfloor}\right)$.

Proof. In Lemma 8.2.6 we proved that computing the separator at Line 7 takes $O(|S|)$ symbolic operations. The same holds for computing the attractor at Line 9 due to Lemma 8.2.2 and computing the SCCs at Line 10. Each iteration of the while-loop at Line 12 takes $O(|S|)$ symbolic operations: Computing the SCC of v twice can be done in $O(|S|)$ symbolic operations due to Theorem 8.2.1. Similarly, computing the random attractor at Line 17 is in $O(|S|)$ symbolic operations due to Lemma 8.2.2. The remaining lines can be done in a constant amount of symbolic operations. It remains to bound the symbolic operations of the while-loop at Line 10 where we call Algorithm 8.4 recursively at Line 11 for each SCC in $S \setminus A$. Note that the size of T determines the number of iterations the while-loop at Line 12 has, and how big the SCCs in $S \setminus A$ are. We obtain that $|T|$ is of size at most $\frac{|S|}{\lfloor \gamma/(2 \log |S|) \rfloor}$ combining Observation 8.2.4 and Lemma 8.2.5. Due to the argument above the following equation bounds the running time of Lines 7–23 for some constant c which is greater than the number of constant symbolic operations in SymMEC(C, γ, P) if $|S| \geq \gamma$.

$$F(S) \leq |T| \cdot |S| \cdot c + \max_{|T|=1 \dots \frac{|S|}{\lfloor (\gamma/2 \log n) \rfloor}} \sum_{S_i \in \mathsf{SCCs}(S \setminus T)} F(S_i)$$

If $|S| < \gamma$ we only have the costs for computing the separator and thus $F(S) \leq c \cdot |S|$. Next, we prove the bound in Claim 8.3.12.

Claim 8.3.12. $F(S) \in O\left(\frac{|S|^2}{\lfloor \gamma/(2 \log |S|) \rfloor} + |S|\right)$.

Proof. We prove the inequality by induction on the size of S. That is we show $F(S) \leq \frac{|S|^2}{Z} c'$ where $Z = \lfloor \gamma/(2 \log |S|) \rfloor$ for some $c' > c$. Obviously the inequality

is true for $|S| < \gamma$, i.e., the base case is true. For the inductive step, consider $F(S) \leq \frac{|S|^2}{Z}c'$ for $|S| \geq \gamma$. Note that

$$\sum_{S_i \in \mathsf{SCCs}(S \setminus T)} F(S_i) \leq \sum_{S_i \in \mathsf{SCCs}(S \setminus T)} c'(|S_i|^2/(\lfloor \gamma/(2 \log |S_i|) \rfloor) + |S_i|) \leq$$
$$c'|S| + \sum_{S_i \in \mathsf{SCCs}(S \setminus T)} c'|S_i|^2/Z \leq c'|S| + \sum_{S_i \in \mathsf{SCCs}(S \setminus T)} c'|S_i| \cdot (|S| - |T|Z)/Z \leq$$
$$c'|S| + c'(|S| - |T|)(|S| - |T|Z)/Z$$

The first inequality is due to the induction hypothesis and the third inequality is due to the fact that we have a Z-separator and Lemma 8.2.5. It remains to add $|S||T|c$:

$$\begin{aligned} F(S) \leq & c'|S| + c'(|S| - |T|)(|S| - |T|Z)/Z + |S||T|c \leq \\ & c'|S| + c'/Z(|S|^2 - |S||T| - |T||S|Z + |T||S|) + |S||T|c = \\ & c'|S| + |S|^2/Zc' - |S||T|c' + |S||T|c \leq c'|S| + \frac{|S|^2}{Z}c'. \end{aligned}$$

The first inequality is due to the fact that $|T| \leq |S|/Z$ (Observation 8.2.4). This concludes our proof by induction of $F(S) \leq c' \cdot (\frac{|S|^2}{Z} + |S|)$. □

Now using that claim and $|S| \leq n$ we obtain a $O\left(n^2/\lfloor \gamma/(2 \log n) \rfloor + n\right)$ bound which can further be simplified to $O\left(n^2/\lfloor \gamma/(2 \log n) \rfloor\right)$ by using $\gamma \leq n$. □

The second part of our analysis bounds the symbolic operations of the case when $\mathsf{diam}(S) < 2\gamma$ at Lines 25–27 and the work done from Line 2 to Line 6.

Lemma 8.3.13. *The total number of symbolic operations of Lines 25–27 in all calls to Algorithm 8.4 is in* $O(n \cdot \gamma + \frac{n^2}{\lfloor \gamma/(2 \log n) \rfloor})$.

Proof. If a vertex is in the set X at Line 25 it is not recursed upon or ever looked at again, thus we charge the symbolic operations of all attractor computations to the vertices in the attractor. This adds up to a total of $O(n)$ symbolic operations by Lemma 8.2.2. Additionally, note that $|X|$ is non-empty, as otherwise $|S|$ is declared as EC in the if-condition at Line 4. Thus, Lines 25–27 occur at most n times. The number of symbolic operations for computing SCCs is in time $O(\sum_{C \in \mathsf{SCCs}(G)}(D_C + 1))$ due to Theorem 8.2.1. We can distribute the costs the SCCs according to the diameters of the SCCs. We provide separate arguments for counting the costs for SCCs S_j with $\mathsf{diam}(S_j) < 2\gamma$ and SCCs S_j with $\mathsf{diam}(S_j) \geq 2\gamma$. First, we consider the costs for SCCs S_j with $\mathsf{diam}(S_j) < 2\gamma$. Computing the SCC S_j costs $O(\gamma)$ operations and the algorithm either terminates in the next step or at least one vertex is removed from the SCC via a separator or attractor. That is we have at most n such SCCs computations and thus an overall cost of $O(n\gamma)$. Now we consider the costs for SCCs S_j with $\mathsf{diam}(S_j) \geq 2\gamma$. Whenever computing such an SCC, we simply charge all its vertices for the costs of computing the SCC, i.e., $O(1)$ for each vertex.

For such an SCC the algorithm either terminates in the next step or a separator is computed and removed. We thus have that each vertex is charged again after at least $\lfloor \gamma/(2 \log |S|) \rfloor$ many nodes are removed from its SCC. In total, each vertex is charged at most $\frac{|S|}{\lfloor \gamma/(2 \log |S|) \rfloor}$ many times. We get an $O(|S| \frac{|S|}{\lfloor \gamma/(2 \log |S|) \rfloor})$ upper bound for the SCCs with large diameter, □

Putting Lemma 8.3.11 and Lemma 8.3.13 together, we obtain the $O(n \cdot \gamma + \frac{n^2}{\lfloor \gamma/(2 \log n) \rfloor})$ bound for Algorithm 8.4 which also applies to Algorithm 8.3 as the SCC-computations only require $O(n)$ operations.

Proposition 8.3.14. *Algorithm 8.4 and Algorithm 8.3 both have* $O(n \cdot \gamma + \frac{n^2}{\lfloor \gamma/(2 \log n) \rfloor})$ *symbolic operations.*

8.3.5 Symbolic Space

Symbolic space usage counted as the maximum number of sets (and not their size) at any point in time is a crucial metric and limiting factor of symbolic computation in practice [105]. In this section, we consider the symbolic space usage of Algorithm 8.4, highlight a key issue, and present a solution to the issue.

Key Issue. Even though Algorithm 8.3 beats the current best *symbolic* Algorithm for computing the MEC decomposition in the number of symbolic operations (current best: $O(n\sqrt{m})$, space: $O(\sqrt{n})$ [93]), without further improvements, Algorithm 8.4 requires $O(n)$ symbolic space as we discuss in the following. First, note that each call of SymMEC(S, γ, P), when excluding the sets stored by recursive calls, only stores a constant number of sets and requires a logarithmic number of sets to execute SCCFind$(\cdot)$. That is, the recursion depth is the crucial factor here. As we show below, by the $\lfloor \gamma/(2 \log n) \rfloor$-separator property, the recursion depth due to the case $T \neq \emptyset$ is $O(n/\lfloor \gamma/(2 \log n) \rfloor)$. However, the case $T = \emptyset$ might lead to a recursion depth of $O(n)$ when in each iteration only a constant number of vertices is removed and the diameter of the resulting SCC is still smaller than 2γ. As Algorithm 8.4 uses a constant amount of sets for each recursive call, it uses $\widetilde{O}(n)$ space in total.

Reducing the symbolic space. We resolve the above space issue by modifying Algorithm 8.4 for the case $T = \emptyset$ (see Algorithm 8.5.): At the while loop at Line 26 we first consider the SCCs S_j with less than $|S|/2$ vertices and recurse on them. If there is an SCC S' with more than $|S|/2$ vertices we process it at the end, i.e., we use one additional set to store that SCC until the SCC algorithm terminates. As now all the computations of the current call to SymMEC(S, γ, P) are done we can simply reuse the sets of the current calls to start the computation for S'. We do so by setting S to S' and continuing in the Line 1 of Algorithm 8.4. Using that we only recurse on sets which are of size $\leq |S|/2$ and thus get a recursion depth of $O(\log n)$ for this case. Moreover, the modified algorithm has the same computation operations as the original one and thus the bounds for the number of symbolic operations apply as well.

Algorithm 8.5: Reduced Space Version of Algorithm 8.4

```
23 else // T = ∅ and thus diam(S) < 2γ
24     X ← Attr_R^P(ROut(S)); S_tmp = ∅
25     while S_j ← SCCFind(S \ X, ∅, P) do
26         if S_j ≥ |S|/2 and |S_j| > 1 then S_tmp ← S_j; continue
27         M ← M ∪ SymMEC(S_j, M, γ, P)
28     if S_tmp ≠ ∅ then S ← S_tmp; goto Line 3
29     return M
```

Lemma 8.3.15. *The maximum recursion depth of the modified algorithm is in* $O(\frac{n}{\lfloor \gamma/(2\log n)\rfloor} + \log n)$.

Proof. Consider the recursion occurring due to Line 11. Because T is a $\lfloor \gamma/(2\log n)\rfloor$-separator, an SCC in $S \setminus T$ contains at most $n - \lfloor \gamma/(2\log n)\rfloor \cdot |T| \leq n - \lfloor \gamma/(2\log n)\rfloor$ vertices as $|T| \geq 1$. We determine how often we can remove $\lfloor \gamma/(2\log n)\rfloor$ from n until there are no vertices left which gives the recursion depth k. It follows that $k = \frac{n}{\lfloor \gamma/(2\log n)\rfloor}$. Now consider the recursion occurring due to Line 27. In the modified version we have that $|S_j| < |S|/2$ and thus this kind of recursion is bounded by $O(\log n)$. □

Algorithm 8.4 has $O(n\gamma + n^2/\lfloor \gamma/(2\log n)\rfloor)$ many symbolic operations due to Lemma 8.3.11 and Lemma 8.3.13 and the number of sets is in $O(n/\lfloor \gamma/(2\log n)\rfloor + \log n)$ due to Lemma 8.3.15. In symbolic algorithms it is of particular interest to optimize symbolic space resources. Note that we obtain a space-time trade-off when setting the parameter γ such that $(2\sqrt{n}+2)\log n \leq \gamma \leq (2n+1)\log n$.

For the symbolic space of Algorithm 8.3 notice that the SCC algorithms are in logarithmic symbolic space and the algorithm itself only needs to stores the set M and the current SCC. Thus, when we immediately output the computed MECs it only requires $O(\log n)$ additional space.

Theorem 8.3.16. *The MEC decomposition of an MDP can be computed in* $O\left(n^{2-\epsilon}\log n\right)$ *symbolic operations and with symbolic space* $O\left(n^{\epsilon}\log n\right)$ *for* $0 < \epsilon \leq 0.5$.

By setting $\epsilon = 0.5$ we obtain that the MEC decomposition of an MDP can be computed in $\widetilde{O}(n\sqrt{n})$ symbolic operations and with symbolic space $\widetilde{O}(\sqrt{n})$.

8.4 Symbolic Qualitative Analysis of Parity Objectives

In this section, we present symbolic algorithms for the qualitative analysis of parity objectives.

Theorem 8.4.1 ([8, 113])**.** *For all MDPs* $\boldsymbol{P}$*, and all parity objectives Parity*(p)*, there exists a memoryless strategy* σ *such that for all* $v \in \langle\langle 1\rangle\rangle_{a.s.}(\mathit{Parity}(p))$ *we have* $\Pr_v^{\sigma}(\mathit{Parity}(p)) = 1$.

8.4.1 Almost-sure Reachability.

In this section, we present a symbolic algorithm that computes reachability objectives Reach(T) in an MDP. The algorithm is a symbolic version of [75, Theorem 4.1].

Symbolic Graph Reachability. For a graph $G = (V, E)$ and a set of vertices S, the set $GraphReach(S, G)$ is the set of vertices V that can reach a vertex of S within G. We compute it by repeatedly calling $S \leftarrow Pre_E(S)$ until we reach a fixed point. In the worst case, we add one vertex in each such call and need $|GraphReach(S, G) \setminus S| + 1 = O(n)$ many $Pre_E(\cdot)$ operations to reach a fixed point.

Algorithm Description. Given an MDP P we compute the set $\langle\langle 1 \rangle\rangle_{a.s.}(\text{Reach}(T, P))$ as follows: First, if player 1 can reach one vertex of a MEC he can reach all the vertices of a MEC and thus we can collapse each MEC M to a player-1 vertex. If M contains a vertex of T, we include the collapsed vertex into T. $P' = (V', E', \langle V_1', V_2' \rangle, \delta')$ is the MDP where the MECs of P are collapsed as described above. Next, we compute the set of vertices S which can reach T in the graph induced by P'. A vertex in $V' \setminus S$ cannot reach T almost-surely because there is no path to T. Note that a play starting from a vertex in the random attractor A of $V' \setminus S$ might also end up in $V' \setminus S$. We thus remove A from V' to obtain the set R, where vertices can almost-surely reach T in P'. Finally, to transfer the result back to P we include all MECs with a vertex in R.

We implement Algorithm 8.3 to compute the MEC decomposition of P but note that we could use any symbolic MEC algorithm. To minimize the extra space usage we also assume that the algorithm that computes the MEC decomposition outputs one MECs after the other instead of all MECs at once. Note that we can easily modify Algorithm 8.3 to output one MEC after the other by iteratively returning each SCC found at Line 5. Moreover, we only require logarithmic space to maintain the state of the SCC algorithm [74].

Algorithm 8.6: Symbolic Almost-Sure Reachability

Input: An MDP $P = (V, E, \langle V_1, V_R \rangle, \delta)$ and a target set $T \subseteq V$
Output: $\langle\langle 1 \rangle\rangle_{a.s.}(\text{Reach}(T, P))$

1 **Procedure** SymASReach(T, γ, P)
2 $V' \leftarrow V, E' \leftarrow E, V_1' \leftarrow V_1, V_R' \leftarrow V_R, \delta' \leftarrow \delta$
3 $P' = (V', E', \langle V_1', V_R' \rangle, \delta')$
4 **for** $M \leftarrow$ ComputeMECs(P) **do**
5 $C \leftarrow$ CollapseEC(M, P')
6 **if** $M \cap T \neq \emptyset$ **then** $T \leftarrow T \cup C$
7 $S \leftarrow GraphReach(T, P')$; $A \leftarrow \text{Attr}_R^{P'}(V' \setminus S)$; $R \leftarrow V' \setminus A$
8 **for** $M \leftarrow$ ComputeMECs(P) **do**
9 **if** $M \cap R \neq \emptyset$ **then** $R \leftarrow R \cup M$ // If $v \in M$ can reach T, M can reach T.
10 **return** R

We prove the following two propositions for Algorithm 8.6. Let P be an MDP,

T a set of vertices and $\mathbf{MEC}$ be the number of symbolic operations we need to compute the MEC decomposition. Let $space(\mathbf{MEC})$ denote the space of computing the MEC decomposition.

Proposition 8.4.2 (Correctness [74])**.** *Algorithm 8.6 computes the set* $\langle\langle 1\rangle\rangle_{a.s.}(Reach(T))$.

Proposition 8.4.3 (Running time and Space)**.** *The total number of symbolic operations of Algorithm 8.6 is* $O(\mathbf{MEC}+n)$. *Algorithm 8.6 uses* $\widetilde{O}(space(\mathbf{MEC}))$ *symbolic space.*

Proposition 8.4.3 and Proposition 8.4.2 together with Theorem 8.3.16 yield the following theorem.

Theorem 8.4.4. *The set* $\langle\langle 1\rangle\rangle_{a.s.}(Reach(T))$ *of an MDP can be computed with* $\widetilde{O}(n^{2-\epsilon})$ *many symbolic operations and* $\widetilde{O}(n^{\epsilon})$ *symbolic space for* $0 < \epsilon \leq 0.5$.

8.4.2 Parity Objectives.

In this section, we consider the qualitative analysis of MDPs with parity objectives. We present an algorithm for computing the winning region which is based on the algorithms we present in the previous sections and the algorithm presented in [78, Section 5]. The algorithm presented in [78, Section 5] draws ideas from a hierarchical clustering technique [158, 217]. Without loss of generality, we consider the parity objectives Parity(p) where $p : V \to \{0, 1, \ldots, 2d\}$. In the symbolic setting, instead of p, we get the sets $\mathscr{P}_{\geq i} = \{v \in V \mid p(v) \geq i\}$ where $(1 \leq i \leq 2d)$ as part of the input. We abbreviate the family $\{\mathscr{P}_{\geq i} \mid 1 \leq i \leq 2d\}$ as $(\mathscr{P}_{\geq k})_{1\leq k\leq 2d}$. Let $\mathscr{P}_{\leq m} = V \setminus \mathscr{P}_{\geq m+1}$ and $\mathscr{P}_m = \mathscr{P}_{\geq m} \setminus \mathscr{P}_{\geq m+1}$. Given an MDP $\boldsymbol{P}$, let $\boldsymbol{P}_i$ denote the MDP obtained by removing $\mathsf{Attr}^{\boldsymbol{P}}_R(\mathscr{P}_{\leq i-1})$, the set of vertices with priority less than i and its random attractor. A MEC $\boldsymbol{M}$ is a *winning MEC* in $\boldsymbol{P}_i$ if there exists a vertex $u \in \boldsymbol{M}$ such that $p(u) = i$ and i is even, i.e., the smallest priority in the MEC is even. Let WE_i be the union of vertices of winning maximal end-components in $\boldsymbol{P}_i$, and let $\mathsf{WE} = \bigcup_{0\leq i\leq 2d} \mathsf{WE}_i$. Lemma 8.4.5 says that computing $\langle\langle 1\rangle\rangle_{a.s.}(\text{Parity}(p))$ is equivalent to computing almost-sure reachability of WE. Intuitively, player 1 can infinitely often satisfy the parity condition after reaching an end-component which satisfies the parity condition.

Lemma 8.4.5 ([78])**.** *We have* $\langle\langle 1\rangle\rangle_{a.s.}(Parity(p)) = \langle\langle 1\rangle\rangle_{a.s.}(Reach(\mathsf{WE}))$.

Thus, we describe in Algorithm 8.7 how to compute WE symbolically.

Algorithm Description.

The algorithm uses a key idea which we describe first. Recall that $\boldsymbol{P}_i$ denotes the MDP obtained by removing $\mathsf{Attr}^{\boldsymbol{P}}_R(\mathscr{P}_{\leq i-1})$, i.e., the set of vertices with priority less than i and its random attractor.

Key Idea. If u, v are in a MEC in P_i, then they are in the same MEC in P_{i-1}. The key idea implies that if a vertex is in a winning MEC of P_i, it is also in a winning MEC of P_{i-1}. Intuitively, this holds due to the following two facts: (1) Because P_{i-1} contains all edges and vertices of P_i the MECs P_i are still strongly connected in P_{i-1}. (2) Because $\mathsf{Attr}^P_R(\mathscr{P}_{\leq i-1})$ makes sure that no MEC M in P_i has a random vertex with an edge leaving P_i in M, the same is true for the set M in P_{i-1}.

We next present the recursive algorithm WinParityEC$(P, (\mathscr{P}_{\geq k})_{1\leq k\leq 2d}, i, j)$ which, for a MDP P, computes the set $\bigcup_{i\leq \ell\leq j} \mathsf{WE}_i$ of winning MECs for priorities between i and j.

1. *Base Case:* If $j < i$, return $\emptyset$.
2. Compute $m \leftarrow \lceil (i+j)/2 \rceil$.
3. Compute the MECs of $P_m = V \setminus \mathsf{Attr}^P_R(\mathscr{P}_{\leq m-1})$ and for each MEC $M \in P_m$ compute the minimal priority *min* among all vertices in that MEC.
4. For each MEC M:
   - If *min* is even then add M to the set W of vertices in winning MECs.
   - If *min* is odd we recursively call WinParityEC$(P^u, (\mathscr{P}_{\geq k})_{1\leq k\leq 2d}, min+1, j)$ where P^u is the sub-MDP containing only vertices and edges inside M. This call applies the key idea and refines the MECs of P_m and computes the set $\bigcup_{min+1\leq \ell\leq j} \mathsf{WE}_\ell$.
5. Call WinParityEC$(P^\ell, (\mathscr{P}_{\geq k})_{1\leq k\leq 2d}, i, m-1)$ where P^ℓ is the MDP where all MECs in P_m are collapsed into a single vertex and thus only the edges outside the MECs of P_m are considered. This call computes the set $\bigcup_{i\leq k\leq m-1} \mathsf{WE}_k$.

We initialize WinParityEC$(P, (\mathscr{P}_{\geq k})_{1\leq k\leq 2d}, i, j)$ with WinParityEC$(P, (\mathscr{P}_{\geq k})_{1\leq k\leq 2d}, 0, 2d)$.

Algorithm 8.7 is the formal version of the sketched algorithm.

Correctness and Number of Symbolic Steps.

In this section, we argue that Algorithm 8.7 is correct and bound the number of symbolic steps and the symbolic space usage. A key difference in the analysis of Algorithm 8.7 and [78] is that we aim for a symbolic step bound that is independent of the number of edges in P and, thus, we cannot use the argument from [78] which charges the cost of each recursive call to the edges of P. The key argument in [78] is that the sets of edges in the different branches of the recursions do not overlap. For vertices, it is not that simple, as we do not entirely remove vertices that appear in a MEC but merge the MEC and represent it by a single vertex. That is, a vertex can appear in both P^ℓ and in P^u corresponding to the MEC. To accomplish our symbolic step bound we adjusted the algorithm. At Line 13 we *always* remove the minimum priority vertices instead of removing the vertices with priority m to ensure that we

Algorithm 8.7: Compute WE of an MDP P

Procedure WinParityEC$(P, (\mathscr{P}_{\geq k})_{1\leq k\leq 2d}, i, j)$
  Input: $P = (V, E, \langle V_1, V_R\rangle, \delta), (\mathscr{P}_{\geq k})_{1\leq k\leq 2d}, i, j$
  $W \leftarrow \emptyset$
  if $j < i$ **then return** W
  $m \leftarrow \lceil (i+j)/2 \rceil$
  $X_m \leftarrow \mathsf{Attr}_R^P(\mathscr{P}_{\leq m-1})$
  $Z_m \leftarrow V \setminus X_m$; $E_m \leftarrow E \cap (Z_m \times Z_m)$
  $P' \leftarrow (Z_m, E_m, \langle V_1 \cap Z_m, V_R \cap Z_m\rangle, \delta)$
  for $M \leftarrow \mathsf{ComputeMECs}(P')$ **do**
    $min \leftarrow minPriority(M)$
    if *min is even* **then**
      $W \leftarrow W \cup M$
    else
      $V^u \leftarrow M \setminus \mathsf{Attr}_R^P(\mathscr{P}_{min})$
      $P^u \leftarrow (V^u, (V^u \times V^u) \cap E, \langle V_1 \cap V^u, V_R \cap V^u\rangle, \delta)$
      $W \leftarrow W \cup$ WinParityEC$(P^u, (\mathscr{P}_{\geq k})_{1\leq k\leq 2d}, min+1, j)$
  /* MDP with MECs collapsed is P^ℓ in the text */
  for $M \leftarrow \mathsf{ComputeMECs}(P')$ **do** CollapseEC(M, P)
  $W \leftarrow W \cup$ WinParityEC$(P, (\mathscr{P}_{\geq k})_{1\leq k\leq 2d}, i, m-1)$
  return W

remove at least one vertex. Intuitively, by always removing at least one vertex from a MEC we ensure that the total number of vertices processed at each recursion level does not grow. Note that these changes of the algorithm do not affect the correctness argument of [78] as we always compute the same sets WE_m but avoid calls to WinParityEC$(\cdot)$ with no progress on some MECs.

Proposition 8.4.6 (Correctness)**.** *Algorithm 8.7 returns the set of winning end-components* WE.

Proof. The correctness of the algorithm is by induction on $j - i$ for the induction hypothesis $\bigcup_{i\leq \ell\leq j} \mathsf{WE}_\ell \subseteq$ WinParityEC$(P, (\mathscr{P}_{\geq k})_{1\leq k\leq 2d}, i, j) \subseteq \bigcup_{1\leq \ell\leq 2d} \mathsf{WE}_\ell$.

First consider the induction base cases: If $j > i$, the algorithm correctly returns the empty set. Next, consider the induction step. Assume that the results hold for $j - i \leq k$, and we consider $j - i = k + 1$. If m is even, then

$$\bigcup_{i\leq k\leq j} \mathsf{WE}_k = \mathsf{WE}_m \cup \bigcup_{i\leq k\leq m-1} \mathsf{WE}_k \cup \bigcup_{m+1\leq k\leq j} \mathsf{WE}_k,$$

otherwise (m is odd), then

$$\bigcup_{i\leq k\leq j} \mathsf{WE}_k = \bigcup_{i\leq k\leq m-1} \mathsf{WE}_k \cup \bigcup_{m+1\leq k\leq j} \mathsf{WE}_k,$$

Consider an arbitrary winning MEC M_k in P_k, i.e., the lowest even priority is k. We consider the following cases.

1. For all $k \geq m$ we have that M_k is contained in a MEC M_m of P_m. Additionally, no random vertex in M_m can have an edge leaving M_m and thus no random vertex in M_k can have a random edge leaving M_m. Moreover, for the minimum priority *min* of M_m we have $k \geq min \geq m$. If *min* is even then M_m is itself winning and thus $M_k \subseteq \text{WinParityEC}(P, (\mathscr{P}_{\geq k})_{1 \leq k \leq 2d}, i, j)$ and $M_m \subseteq \bigcup_{1 \leq \ell \leq 2d} \mathsf{WE}_\ell$ (note that it might be that $min > j$).

   If *min* is odd M_k is a winning MEC of P iff it is a winning MEC of P^u and thus by the induction hypothesis $M_k \subseteq \text{WinParityEC}(P^u, (\mathscr{P}_{\geq k})_{1 \leq k \leq 2d}, min + 1, j)$. It follows that also $M_k \subseteq \text{WinParityEC}(P, (\mathscr{P}_{\geq k})_{1 \leq k \leq 2d}, i, j)$.

2. For $k < m$ consider a MEC M_k in P_k. If M_k contains a vertex v that belongs to a MEC M_m of P_m, then $M_m \subset M_k$ (i.e., all vertices of the MEC in P_m of v also belong to M_k and M_k has at least one additional vertex with priority $< m$). We thus have that for $k \leq m$ the winning MECs M_k in P_k are in one-to-one correspondence with the winning MECs M'_k of the modified MDP where all MECs of P_m are collapsed. From the induction hypothesis it follows that $\bigcup_{i \leq k \leq m-1} \mathsf{WE}_k = \text{WinParityEC}(P, (\mathscr{P}_{\geq k})_{1 \leq k \leq 2d}, i, m-1)$

Hence, $\bigcup_{i \leq k \leq j} \mathsf{WE}_k \subseteq \text{WinParityEC}(P, (\mathscr{P}_{\geq k})_{1 \leq k \leq 2d}, i, j) \subseteq \bigcup_{1 \leq \ell \leq 2d} \mathsf{WE}_\ell$. The statements follows from setting $i = 0$ and $j = 2d$. □

Proposition 8.4.7 (Symbolic Steps)**.** *The total number of symbolic operations for Algorithm 8.7 is* $O(\mathbf{MEC} \cdot \log d)$ *for* $0 < \epsilon \leq 0.5$.

Proof. Given an MDP P with n vertices and d priorities, let us denote by $T(n, x)$ the number of symbolic steps of Algorithm 8.7 at recursion depth x and with $T_M(n)$ the number of symbolic steps incurred by the symbolic MEC Algorithm. As shown in [78], note that the recursion depth of Algorithm 8.7 is in $O(\log d)$ because we recursively consider either $(min + 1, j)$ or $(i, m - 1)$ where $min \geq m = \lceil (i + j/2) \rceil$ until $j > i$, where $j = 2d$ initially. First, we argue why there exists $c > 0$ such that

$$T(n, x) \leq c \cdot T_M(n) + \left(\sum_{i=1,\ldots,t} T\left(n_i, x - 1\right) \right)$$
$$+ T\left(n - \left(\sum_{i=1,\ldots,t} n_i \right) + t, x - 1 \right) \text{ if } x > 1$$
$$T(n, 0) \leq c.$$

The attractors computed at Line 5 and Line 13 can be done in $O(n)$ symbolic steps as the set of vertices in the attractors are all disjunct. Clearly, this is cheaper than computing the MEC decomposition. To extract the minimum priority of a set of nodes $X \subseteq V$ we apply a binary search procedure which takes $O(\log d)$ symbolic steps at Line 9. Note that when $\log d > n$ we cannot charge the cost to computing the MEC decomposition. Thus, we argue in Claim 8.4.8 that the total number of symbolic steps for Line 9 in Algorithm 8.7 is less than $O(n \log d)$. The rest of the

symbolic steps in Algorithm 8.7, (except the recursive calls and computing the MEC decomposition) in Algorithm 8.7 can be done in a constant amount of symbolic steps.

Note that when $x = 0$, i.e., in the case $j < i$, we only need a constant amount of symbolic steps.

Let t be the number of MECS in P'. When $x > 0$, consider the following argumentation for the number of symbolic steps of the recursive calls:

- WinParityEC(P^u, $(\mathscr{P}_{\geq k})_{1\leq k\leq 2d}$, min, j): We perform the recursive call for each MEC $M_i \in P'$ $(1 \leq i \leq t)$ where the vertex with minimum priority is odd. The total cost incurred by all such recursive calls is $\sum_{i=1,\ldots,t} T(n_i, x-1)$ where $n_i \leq |M_i| - 1$ because we always remove the vertices with minimum priority at Line 13.

- WinParityEC(P^ℓ, $(\mathscr{P}_{\geq k})_{1\leq k\leq 2d}$, i, $m-1$): P^ℓ consists of the vertices representing the collapsed MECs, the vertices not in P' and the vertices which are not in a MEC of P'. The number of vertices in P^ℓ is thus $n_\ell = n - \sum_{i=1,\ldots,t} |M_i| + t$ and we obtain $T(n_\ell, x-1)$.

Note that $\sum_{i=1,\ldots,t} n_i + n_\ell \leq n$. We choose c such that $cT_M(n)$ is greater than the number of symbolic steps for computing the MECs twice and the rest of the work in the current iteration of Algorithm 8.7. It is straightforward to show that $T(n,d) = O(T_M(n)\log d)$.

The following claim shows that the total number of symbolic steps incurred by Line 9 for all calls to WinParityEC($\cdot$) is only $O(n \log d)$.

Claim 8.4.8. *The total amount of symbolic steps used by Line 9 is in $O(n \log d)$.*

Proof. To obtain the set of vertices with minimum priority from a set of vertices $X \subseteq V$ the function $\mathsf{MinPriority}_(X)$ performs a binary search using the sets $(\mathscr{P}_{\geq k})_{1\leq k\leq 2d}$. This can be done in $O(\log d)$ many symbolic steps. To prove that the number of symbolic steps used by Line 9 in total is in $O(n \log d)$ note that each time the function is performed we either: (i) Remove all vertices in M, and we never perform the function on the vertices in M again. We charge the cost to an arbitrary vertex in M. (ii) Remove at least one vertex at Line 13 and we never perform the function on a MEC containing this vertex again. We charge the cost to this vertex. As there are only n vertices we obtain that the total amount of symbolic steps used by Line 9 is in $O(n \log d)$. □

Using Claim 8.4.8 we conclude the proof for the symbolic step bound of Algorithm 8.7. □

Proposition 8.4.9. *Algorithm 8.7 uses $O(space(\mathbf{MEC}) + \log n \log d)$ space.*

Proof. Let P be an MDP n vertices and a parity objective with d priorities. We denote with $space(\mathbf{MEC})$ the symbolic space used by the algorithm that computes the MEC decomposition. Observe that all computation steps in Algorithm 8.7 need

constant space except for the recursions and computing the MEC decomposition. Both at Line 8 and Line 16 we first compute the MEC decomposition and then, to minimize extra space, we output one MEC after the other by returning each SCC found given the set of vertices in nontrivial MECs. Note that we only require logarithmic space to maintain the state of the SCC algorithm [74]. As argued in [78] the recursion depth of Algorithm 8.7 is $O(\log d)$. Thus, we need $O(\log n \log d)$ space for maintaining the state of the SCC algorithm at Line 5 until we reach a leaf of the recursion tree. At each recursive call, we need additive $O(space(\mathbf{MEC}))$ space to compute the MEC decomposition of $\boldsymbol{P}$. This yields the claimed space bound. □

Given an MDP, we first compute the set WE with Algorithm 8.7 which is correct due to Proposition 8.4.6. We instantiate $\mathbf{MEC}$ and $space(\mathbf{MEC})$ in Proposition 8.4.7 and Proposition 8.4.9 respectively with Theorem 8.3.16 and thus need $O(n^{2-\epsilon} \log n \log d)$ symbolic steps and $O(n^{\epsilon} \log n + \log n \log d)$ (where $0 < \epsilon \leq 0.5$) symbolic space for computing WE. Then, we compute almost-sure reachability of WE with Theorem 8.4.4. Finally, using Lemma 8.4.5 we obtain the following theorem.

Theorem 8.4.10. *The set $\langle\langle 1 \rangle\rangle_{a.s.}(Parity(p))$ of an MDP $\boldsymbol{P}$ can be computed with $\widetilde{O}(n^{2-\epsilon})$ many symbolic operations and $\widetilde{O}(n^{\epsilon})$ symbolic space for $0 < \epsilon \leq 0.5$.*

8.5 Conclusion

We present a faster symbolic algorithm for the MEC decomposition. Furthermore, we improve the fastest symbolic algorithm for verifying MDPs with ω-regular properties. There are several interesting directions for future work. On the practical side, implementations and experiments with case studies is an interesting direction. On the theoretical side, improving upon the $\widetilde{O}(n^{1.5})$ bound for MECs is an interesting open question which would also, using our work, improve the presented algorithm for verifying ω-regular properties of MDPs.

CHAPTER 9

Conclusion

In the thesis, we examine instances of central problems in model-checking and reactive synthesis. Chapters 3–6 provide *explicit algorithms* for problems with widely-considered objectives like mean-payoff parity objectives, Streett objectives, bounded liveness objectives and variants of reachability objectives. Chapters 7–8 provide *symbolic algorithms* for problems with parity objectives in game graphs and MDPs. The careful transfer of sophisticated modern graph algorithmic techniques to instances of these central problems provides the new improved algorithms.

We conclude with concrete ideas for follow-up work.

Implementation and Experiments. Even though the discovery of improved theoretical algorithms is an important problem-solving challenge we must also implement them: The implementation of a theoretic algorithm removes semantic gaps between code and pseudocode and meaningful experiments offer valuable insights into how an algorithm performs in the real world [206].

- For Streett objectives in graphs and MDPs and parity objectives in games implementations of explicit algorithms [50, 118] and symbolic algorithms [93, 176, 205] exist. An interesting direction for future work is to implement the algorithms for parity objectives in MDPs introduced in Chapter 8.
- For problems with mean-payoff parity objectives in games our theoretical result in Chapter 3 is improved to a pseudo-quasi-polynomial running time [115] and it is important future work to determines which algorithms perform best in practice in both the explicit model and the symbolic model of computation.
- For bounded liveness objectives, implementations of the algorithms is future work.
- For the MEC-decomposition in MDPs implementations for explicit algorithm exist [225] but for symbolic algorithms it is important future work.

Theoretical follow-up question. Recently, a breakthrough for deterministic dynamic algorithms [32, 102, 207] yielded faster deterministic dynamic algorithms for many

central graph-theoretic problems and it is an important open question if the techniques are useful for improved *deterministic* algorithms for MEC decomposition in MDPs or Streett objectives in graphs. For symbolic algorithms improving upon the $\widetilde{O}(n^{1.5})$ bound for MEC decomposition is an interesting open question. For explicit algorithms any improvement upon the $\widetilde{O}(n^{2.5})$ time algorithm in graphs and the $O(n^2 d)$ time algorithms in games for bounded Büchi objectives is interesting. Sub-cubic time *deterministic* algorithm for bounded Büchi objectives in graphs discussed in Chapter 5 are interesting future work. Additionally, any conditional lower bounds for bounded Büchi objectives in games would separate the objectives from Büchi objectives. Any improvement upon the long-standing *deterministic* $O(mn^{2/3})$ time algorithm for MEC decomposition [91] is interesting. Finally, providing a *polynomial time* algorithm for computing the winning set of parity or mean-payoff objectives in games is a major theoretical open problem and a formidable task for future work.

Bibliography

[1] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. "Weak, strong, and strong cyclic planning via symbolic model checking". In: *Artificial Intelligence* 147.1-2 (2003), pp. 35–84.

[2] M. Abadi, L. Lamport, and P. Wolper. "Realizable and Unrealizable Specifications of Reactive Systems". In: *ICALP*. Vol. 372. Lecture Notes in Computer Science. Springer, 1989, pp. 1–17.

[3] A. Abboud and V. Vassilevska Williams. "Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems". In: *FOCS*. 2014, pp. 434–443.

[4] A. Abboud, A. Backurs, and V. V. Williams. "If the Current Clique Algorithms Are Optimal, so Is Valiant's Parser". In: *SIAM Journal Computing* 47.6 (2018), pp. 2527–2555.

[5] A. Abboud and V. V. Williams. "Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems". In: *FOCS*. 2014, pp. 434–443.

[6] A. Abboud, V. V. Williams, and H. Yu. "Matching Triangles and Basing Hardness on an Extremely Popular Conjecture". In: *SIAM J. Comput.* 47.3 (2018), pp. 1098–1122.

[7] M. Aghighi, C. Bäckström, P. Jonsson, and S. Ståhlberg. "Refining complexity analyses in planning by exploiting the exponential time hypothesis". In: *Annals of Mathematics and Artificial Intelligence* 78.2 (2016), pp. 157–175.

[8] L. de Alfaro. "Formal Verification of Probabilistic Systems". PhD thesis. Stanford University, 1997.

[9] L. de Alfaro and M. Faella. "An Accelerated Algorithm for 3-Color Parity Games with an Application to Timed Games". In: *CAV*. 2007, pp. 108–120.

[10] L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga. "The Element of Surprise in Timed Games". In: *CONCUR*. 2003, pp. 142–156.

[11] L. de Alfaro and T. A. Henzinger. "Interface theories for component-based design". In: *EMSOFT*. 2001, pp. 148–165.

[12] L. de Alfaro, T. A. Henzinger, and R. Majumdar. "Symbolic Algorithms for Infinite-State Games". In: *CONCUR*. 2001, pp. 536–550.

[13] R. Alford, U. Kuter, D. S. Nau, and R. P. Goldman. "Plan Aggregation for Strong Cyclic Planning in Nondeterministic Domains". In: *Artificial Intelligence* 216 (2014), pp. 206–232.

[14] B. Alpern and F. B. Schneider. "Defining Liveness". In: *Inf. Process. Lett.* 21.4 (1985), pp. 181–185.

[15] R. Alur, T. A. Henzinger, and O. Kupferman. "Alternating-time temporal logic". In: *JACM* 49 (2002), pp. 672–713.

[16] R. Alur and T. Henzinger. "Computer-aided verification". unpublished. 2004.

[17] R. Alur, T. Henzinger, O. Kupferman, and M. Vardi. "Alternating refinement relations". In: *CONCUR*. LNCS 1466. Springer, 1998, pp. 163–178.

[18] R. Alur and T. A. Henzinger. "Finitary Fairness". In: *ACM Trans. Program. Lang. Syst.* 20.6 (1998), pp. 1171–1194.

[19] C. Bäckström and P. Jonsson. "Time and Space Bounds for Planning". In: *Journal of Artificial Intelligence Research* 60 (2017), pp. 595–638.

[20] C. Bäckström and B. Nebel. "Complexity Results for SAS+ Planning". In: *Computational Intelligence* 11 (1995), pp. 625–656.

[21] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. "Algebraic Decision Diagrams and Their Applications". In: *Formal Methods in System Design* 10.2/3 (1997), pp. 171–206.

[22] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.

[23] C. Baier, L. de Alfaro, V. Forejt, and M. Kwiatkowska. "Model Checking Probabilistic Systems". In: *Handbook of Model Checking*. Springer, 2018, pp. 963–999.

[24] C. Baier, E. M. Clarke, V. Hartonas-Garmhausen, M. Z. Kwiatkowska, and M. Ryan. "Symbolic Model Checking for Probabilistic Processes". In: *ICALP*. Vol. 1256. Lecture Notes in Computer Science. Springer, 1997, pp. 430–440.

[25] C. Baier, M. Größer, M. Leucker, B. Bollig, and F. Ciesinski. "Controller Synthesis for Probabilistic Systems". In: *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France*. Vol. 155. IFIP. Kluwer/Springer, 2004, pp. 493–506.

[26] C. Baier, H. Hermanns, and J. Katoen. "The 10, 000 Facets of MDP Model Checking". In: *Computing and Software Science - State of the Art and Perspectives*. Ed. by B. Steffen and G. J. Woeginger. Vol. 10000. Lecture Notes in Computer Science. Springer, 2019, pp. 420–451.

[27] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. "Graph Expansion and Communication Costs of Fast Matrix Multiplication". In: *J. ACM* 59.6 (2012), 32:1–32:23.

[28] C. Beeri. "On the Membership Problem for Functional and Multivalued Dependencies in Relational Databases". In: *ACM Trans. Database Syst.* 5.3 (1980), pp. 241–259.

[29] M. Ben-Ari. "The bug that destroyed a rocket". In: *ACM SIGCSE Bull.* 33.2 (2001), pp. 58–59.

[30] M. Benerecetti, D. Dell'Erba, and F. Mogavero. "Solving Parity Games via Priority Promotion". In: *CAV*. 2016, pp. 270–290.

[31] A. Bernstein, M. Probst, and C. Wulff-Nilsen. "Decremental Strongly-Connected Components and Single-Source Reachability in Near-Linear Time". In: *STOC*. 2019, pp. 365–376.

[32] A. Bernstein, M. P. Gutenberg, and T. Saranurak. "Deterministic Decremental Reachability, SCC, and Shortest Paths via Directed Expanders and Congestion Balancing". In: *FOCS*. IEEE, 2020, pp. 1123–1134.

[33] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. "Bounded model checking". In: *Adv. Comput.* 58 (2003), pp. 117–148.

[34] R. Bloem, K. Chatterjee, T. A. Henzinger, and B. Jobstmann. "Better Quality in Synthesis through Quantitative Objectives". In: *CAV*. LNCS 5643. Springer, 2009, pp. 140–156.

[35] R. Bloem, K. Chatterjee, K. Greimel, T. A. Henzinger, G. Hofferek, B. Jobstmann, B. Könighofer, and R. Könighofer. "Synthesizing robust systems". In: *Acta Inf.* 51.3-4 (2014), pp. 193–220.

[36] A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J. Raskin. "Acacia+, a Tool for LTL Synthesis". In: *CAV*. Ed. by P. Madhusudan and S. A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 652–657.

[37] A. Bohy, V. Bruyère, E. Filiot, and J. Raskin. "Synthesis from LTL Specifications with Mean-Payoff Objectives". In: *TACAS*. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 169–184.

[38] M. Bojanczyk and T. Colcombet. "Bounds in ω-Regularity". In: *LICS*. IEEE Computer Society, 2006, pp. 285–296.

[39] B. Bonet and H. Geffner. "Planning under Partial Observability by Classical Replanning: Theory and Experiments". In: *IJCAI*. Ed. by T. Walsh. 2011, pp. 1936–1941.

[40] B. Bonet and H. Geffner. "Planning with Incomplete Information as Heuristic Search in Belief Space". In: *AIPS*. 2000, pp. 52–61.

[41] P. Bouyer, U. Fahrenberg, K. G. Larsen, N. Markey, and J. Srba. "Infinite Runs in Weighted Timed Automata with Energy Constraints". In: *FORMATS*. LNCS 5215. Springer, 2008, pp. 33–47.

[42] P. Bouyer, N. Markey, J. Olschewski, and M. Ummels. "Measuring Permissiveness in Parity Games: Mean-Payoff Parity Games Revisited". In: *ATVA*. LNCS 6996. Springer, 2011, pp. 135–149.

[43] T. Brázdil, V. Brozek, K. Chatterjee, V. Forejt, and A. Kucera. "Two Views on Multiple Mean-Payoff Objectives in Markov Decision Processes". In: *LICS 2011*. 2011, pp. 33–42.

[44] L. Brim, J. Chaloupka, L. Doyen, R. Gentilini, and J. F. Raskin. "Faster Algorithms for Mean-payoff Games". In: *Formal Methods System Design* 38.2 (Apr. 2011), pp. 97–118. ISSN: 0925-9856.

[45] K. Bringmann, N. Fischer, and M. Künnemann. "A Fine-Grained Analogue of Schaefer's Theorem in P: Dichotomy of Existsk-Forall-Quantified First-Order Graph Properties". In: *CCC*. 2019, 31:1–31:27.

[46] K. Bringmann and M. Künnemann. "Quadratic conditional lower bounds for string problems and dynamic time warping". In: *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*. IEEE. 2015, pp. 79–97.

[47] A. Browne, E. M. Clarke, S. Jha, D. E. Long, and W. R. Marrero. "An Improved Algorithm for the Evaluation of Fixpoint Expressions". In: *Theoretical Computer Science* 178.1-2 (1997), pp. 237–255. Announced at CAV'94.

[48] F. Bruse, M. Falk, and M. Lange. "The Fixpoint-Iteration Algorithm for Parity Games". In: *GandALF*. 2014, pp. 116–130.

[49] V. Bruyère. "Computer Aided Synthesis: A Game-Theoretic Approach". In: *Developments in Language Theory - 21st International Conference, DLT 2017, Liège, Belgium, August 7-11, 2017, Proceedings*. Ed. by É. Charlier, J. Leroy, and M. Rigo. Vol. 10396. Lecture Notes in Computer Science. Springer, 2017, pp. 3–35.

[50] V. Bruyère, G. A. Pérez, J. Raskin, and C. Tamines. "Partial Solvers for Generalized Parity Games". In: *RP*. Vol. 11674. Lecture Notes in Computer Science. Springer, 2019, pp. 63–78.

[51] R. E. Bryant. "Graph-based algorithms for boolean function manipulation". In: *IEEE Transactions on Computers* 100.8 (1986), pp. 677–691.

[52] R. E. Bryant. "Symbolic Boolean Manipulation with Ordered Binary-decision Diagrams". In: *ACM Comput. Surv.* 24.3 (Sept. 1992), pp. 293–318. ISSN: 0360-0300.

[53] J. R. Büchi. "On a decision method in restricted second-order arithmetic". In: *Proceedings of the First International Congress on Logic, Methodology, and Philosophy of Science 1960*. 1962, pp. 1–11.

[54] J. R. Büchi and L. H. Landweber. "Solving sequential conditions by finite-state strategies". In: *Trans. AMS* 138 (1969), pp. 295–311.

[55] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. "Symbolic Model Checking: 10^20 States and Beyond". In: *LICS*. 1990, pp. 428–439.

[56] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. "Sequential Circuit Verification Using Symbolic Model Checking". In: *DAC*. IEEE Computer Society Press, 1990, pp. 46–51.

[57] D. Bustan, O. Kupferman, and M. Y. Vardi. "A Measured Collapse of the Modal μ-Calculus Alternation Hierarchy". In: *STACS*. 2004, pp. 522–533.

[58] T. Bylander. "The Computational Complexity of Propositional STRIPS Planning". In: *Artificial Intelligence* 69.1-2 (1994), pp. 165–204.

[59] C. S. Calude, S. Jain, B. Khoussainov, W. Li, and F. Stephan. "Deciding parity games in quasipolynomial time". In: *STOC*. 2017, pp. 252–263.

[60] A. Camacho, J. A. Baier, C. Muise, and S. A. McIlraith. "Finite LTL Synthesis as Planning". In: *ICAPS*. 2018, pp. 29–38.

[61] A. Camacho, M. Bienvenu, and S. A. McIlraith. "Finite LTL Synthesis with Environment Assumptions and Quality Measures". In: *KR*. 2018, pp. 454–463.

[62] A. Camacho, C. J. Muise, J. A. Baier, and S. A. McIlraith. "LTL Realizability via Safety and Reachability Games." In: *IJCAI*. 2018, pp. 4683–4691.

[63] A. Camacho, C. J. Muise, and S. A. McIlraith. "From FOND to Robust Probabilistic Planning: Computing Compact Policies that Bypass Avoidable Deadends". In: *ICAPS*. 2016, pp. 65–69.

[64] A. Camacho, E. Triantafillou, C. J. Muise, J. A. Baier, and S. A. McIlraith. "Non-Deterministic Planning with Temporally Extended Goals: LTL over Finite and Infinite Traces". In: *AAAI*. 2017, pp. 3716–3724.

[65] *Case studies using the PRISM model checker*. `https://www.prismmodelchecker.org/casestudies/index.php`, Accessed on 17.2.2021.

[66] P. Cerný, K. Chatterjee, T. A. Henzinger, A. Radhakrishna, and R. Singh. "Quantitative Synthesis for Concurrent Programs". In: *CAV*. LNCS 6806. Springer, 2011, pp. 243–259.

[67] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. "Resource interfaces". In: *EMSOFT*. LNCS 2855. Springer, 2003, pp. 117–133.

[68] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. "Resource Interfaces". In: *EMSOFT*. Vol. 2855. Lecture Notes in Computer Science. Springer, 2003, pp. 117–133.

[69] K. Chatterjee. "Stochastic ω-Regular Games". PhD thesis. UC Berkeley, 2007.

[70] K. Chatterjee and L. Doyen. "Energy and Mean-Payoff Parity Markov Decision Processes". In: *MFCS*. LNCS 6907. Springer, 2011, pp. 206–218.

[71] K. Chatterjee and L. Doyen. "Energy Parity Games". In: *ICALP (B)*. LNCS 6199. Springer, 2010, pp. 599–610. ISBN: 978-3-642-14161-4.

[72] K. Chatterjee, W. Dvořák, M. Henzinger, and V. Loitzenbauer. "Conditionally Optimal Algorithms for Generalized Büchi Games". In: *MFCS*. 2016, 25:1–25:15.

[73] K. Chatterjee, W. Dvořák, M. Henzinger, and V. Loitzenbauer. "Improved Set-Based Symbolic Algorithms for Parity Games". In: *CSL*. 2017, 18:1–18:21.

[74] K. Chatterjee, W. Dvořák, M. Henzinger, and V. Loitzenbauer. "Lower Bounds for Symbolic Computation on Graphs: Strongly Connected Components, Liveness, Safety, and Diameter". In: *SODA*. 2018, pp. 2341–2356.

[75] K. Chatterjee, W. Dvořák, M. Henzinger, and V. Loitzenbauer. "Model and Objective Separation with Conditional Lower Bounds: Disjunction is Harder than Conjunction". In: *LICS*. 2016, pp. 197–206.

[76] K. Chatterjee, A. Gaiser, and J. Kretínský. "Automata with Generalized Rabin Pairs for Probabilistic Model Checking and LTL Synthesis". In: *CAV*. 2013.

[77] K. Chatterjee and M. Henzinger. "An $O(n^2)$ Time Algorithm for Alternating Büchi Games". In: *SODA*. 2012, pp. 1386–1399.

[78] K. Chatterjee and M. Henzinger. "Faster and Dynamic Algorithms for Maximal End-Component Decomposition and Related Graph Problems in Probabilistic Verification". In: *SODA*. 2011, pp. 1318–1336.

[79] K. Chatterjee, M. Henzinger, M. Joglekar, and N. Shah. "Symbolic algorithms for qualitative analysis of Markov decision processes with Büchi objectives". In: *Form. Methods Syst. Des.* 42.3 (2013), pp. 301–327.

[80] K. Chatterjee, M. Henzinger, and V. Loitzenbauer. "Improved Algorithms for One-Pair and k-Pair Streett Objectives". In: *LICS*. 2015, pp. 269–280.

[81] K. Chatterjee, M. Henzinger, and V. Loitzenbauer. "Improved Algorithms for Parity and Streett objectives". In: *Logical Methods in Computer Science* 13.3 (2017).

[82] K. Chatterjee and T. A. Henzinger. "Probabilistic Systems with LimSup and LimInf Objectives". In: *ILC*. 2007, pp. 32–45.

[83] K. Chatterjee, T. A. Henzinger, and M. Jurdziński. "Mean-Payoff Parity Games". In: *LICS*. IEEE Computer Society, 2005, pp. 178–187.

[84] K. Chatterjee and L. Doyen. "Games and Markov Decision Processes with Mean-Payoff Parity and Energy Parity Objectives". In: *MEMICS*. 2011, pp. 37–46.

[85] K. Chatterjee, L. Doyen, H. Gimbert, and Y. Oualhadj. "Perfect-Information Stochastic Mean-Payoff Parity Games". In: *FOSSACS*. 2014, pp. 210–225.

[86] K. Chatterjee, W. Dvořák, M. Henzinger, and V. Loitzenbauer. "Model and Objective Separation with Conditional Lower Bounds: Disjunction is Harder Than Conjunction". In: *LICS*. ACM, 2016, pp. 197–206. ISBN: 978-1-4503-4391-6.

[87] K. Chatterjee, W. Dvorák, M. Henzinger, and A. Svozil. "Algorithms and Conditional Lower Bounds for Planning Problems". In: *ICAPS*. AAAI Press, 2018, pp. 56–64.

[88] K. Chatterjee, W. Dvorák, M. Henzinger, and A. Svozil. "Quasipolynomial Set-Based Symbolic Algorithms for Parity Games". In: *LPAR*. Vol. 57. EPiC Series in Computing. EasyChair, 2018, pp. 233–253.

[89] K. Chatterjee, W. Dvořák, M. Henzinger, and A. Svozil. "Near-Linear Time Algorithms for Streett Objectives in Graphs and MDPs". In: *CONCUR*. 2019, 7:1–7:16.

[90] K. Chatterjee, W. Dvořák, M. Henzinger, and A. Svozil. "Symbolic Time and Space Tradeoffs for Probabilistic Verification". In: *unpublished*. 2021.

[91] K. Chatterjee and M. Henzinger. "Efficient and Dynamic Algorithms for Alternating Büchi Games and Maximal End-Component Decomposition". In: *J. ACM* 61.3 (2014), 15:1–15:40.

[92] K. Chatterjee, M. Henzinger, S. Kale, and A. Svozil. "Faster Algorithms for Bounded Liveness in Graphs and Game Graphs". In: *unpublished*. 2021.

[93] K. Chatterjee, M. Henzinger, V. Loitzenbauer, S. Oraee, and V. Toman. "Symbolic Algorithms for Graphs and Markov Decision Processes with Fairness Objectives". In: *CAV*. 2018, pp. 178–197.

[94] K. Chatterjee, M. Henzinger, V. Loitzenbauer, S. Oraee, and V. Toman. "Symbolic Algorithms for Graphs and Markov Decision Processes with Fairness Objectives". In: *CAV*. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 178–197.

[95] K. Chatterjee, M. Henzinger, and A. Svozil. "Faster Algorithms for Mean-Payoff Parity Games". In: *MFCS*. Vol. 83. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 39:1–39:14.

[96] K. Chatterjee, T. A. Henzinger, and F. Horn. "Finitary winning in omega-regular games". In: *ACM Trans. Comput. Log.* 11.1 (2009), 1:1–1:27.

[97] K. Chatterjee, T. A. Henzinger, B. Jobstmann, and R. Singh. "Measuring and Synthesizing Systems in Probabilistic Environments". In: *J. ACM* 62.1 (2015), 9:1–9:34.

[98] S. Chechik, T. D. Hansen, G. F. Italiano, J. Lacki, and N. Parotsidis. "Decremental Single-Source Reachability and Strongly Connected Components in $\widetilde{O}(m\sqrt{n})$ Total Update Time". In: *FOCS*. 2016, pp. 315–324.

[99] H. M. Choset. *Principles of Robot Motion: Theory, Algorithms, and Implementation*. MIT Press, 2005.

[100] A. Church. "Logic, arithmetic, and automata". In: *ICM*. 1962, pp. 23–35.

[101] A. Church. "An unsolvable problem of elementary number theory". In: *American journal of mathematics* 58.2 (1936), pp. 345–363.

[102] J. Chuzhoy, Y. Gao, J. Li, D. Nanongkai, R. Peng, and T. Saranurak. "A Deterministic Algorithm for Balanced Cut with Applications to Dynamic Connectivity, Flows, and Beyond". In: *FOCS*. IEEE, 2020, pp. 1158–1167.

[103] F. Ciesinski and C. Baier. "LiQuor: A tool for Qualitative and Quantitative Linear Time analysis of Reactive Systems". In: *QEST*. 2006, pp. 131–132.

[104] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. "NUSMV: a new symbolic model checker". In: *International Journal on Software Tools for Technology Transfer* 2.4 (Mar. 2000), pp. 410–425. ISSN: 1433-2779.

[105] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-03270-8.

[106] E. M. Clarke and E. A. Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic". In: *Logic of Programs*. 1981, pp. 52–71.

[107] E. M. Clarke, K. L. McMillan, S. V. A. Campos, and V. Hartonas-Garmhausen. "Symbolic Model Checking". In: *CAV*. 1996, pp. 419–427.

[108] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-guided Abstraction Refinement for Symbolic Model Checking". In: *J. ACM* 50.5 (2003), pp. 752–794.

[109] E. M. Clarke, E. A. Emerson, and J. Sifakis. "Model checking: algorithmic verification and debugging". In: *Commun. ACM* 52.11 (2009), pp. 74–84.

[110] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds. *Handbook of Model Checking*. Springer, 2018. ISBN: 978-3-319-10574-1.

[111] C. Comin and R. Rizzi. "Improved Pseudo-polynomial Bound for the Value Problem and Optimal Strategy Synthesis in Mean Payoff Games". In: *Algorithmica* 77.4 (Apr. 2017), pp. 995–1021.

[112] C. Courcoubetis and M. Yannakakis. "Markov Decision Processes and Regular Events". In: *ICALP*. 1990, pp. 336–349.

[113] C. Courcoubetis and M. Yannakakis. "The Complexity of Probabilistic Verification". In: *J. ACM* 42.4 (1995), pp. 857–907.

[114] P. Daca, T. A. Henzinger, J. Kretínský, and T. Petrov. "Faster Statistical Model Checking for Unbounded Temporal Properties". In: *ACM Trans. Comput. Log.* 18.2 (2017), 12:1–12:25.

[115] L. Daviaud, M. Jurdzinski, and R. Lazic. "A pseudo-quasi-polynomial algorithm for mean-payoff parity games". In: *LICS*. Ed. by A. Dawar and E. Grädel. ACM, 2018, pp. 325–334.

[116] L. Daviaud, M. Jurdzinski, and K. S. Thejaswini. "The Strahler Number of a Parity Game". In: *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*. Ed. by A. Czumaj, A. Dawar, and E. Merelli. Vol. 168. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 123:1–123:19.

[117] C. Dehnert, S. Junges, J. Katoen, and M. Volk. "A Storm is Coming: A Modern Probabilistic Model Checker". In: *CAV*. 2017, pp. 592–600.

[118] T. van Dijk. "Oink: An Implementation and Evaluation of Modern Parity Game Solvers". In: *TACAS*. 2018, pp. 291–308.

[119] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. The MIT Press, 1989.

[120] D. Dorfman, H. Kaplan, and U. Zwick. "A Faster Deterministic Exponential Time Algorithm for Energy Games and Mean Payoff Games". In: *ICALP*. Vol. 132. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 114:1–114:14.

[121] A. Ehrenfeucht and J. Mycielski. "Positional strategies for mean payoff games". In: *International Journal of Game Theory* 8.2 (1979), pp. 109–113. ISSN: 1432-1270.

[122] E. Eiben, J. Gemmell, I. A. Kanj, and A. Youngdahl. "Improved Results for Minimum Constraint Removal". In: *AAAI*. 2018, pp. 6477–6484.

[123] E. A. Emerson and C. Lei. "Efficient Model Checking in Fragments of the Propositional Mu-Calculus". In: *LICS*. 1986, pp. 267–278.

[124] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. "Quantitative Temporal Reasoning". In: *Real Time Syst.* 4.4 (1992), pp. 331–352.

[125] E. Emerson and C. Jutla. "Tree automata, mu-calculus and determinacy". In: *FOCS*. 1991, pp. 368–377.

[126] J. Esparza and J. Kretínský. "From LTL to Deterministic Automata: A Safraless Compositional Approach". In: *CAV*. 2014, pp. 192–208.

[127] S. Even and Y. Shiloach. "An On-Line Edge-Deletion Problem". In: *Journal of the ACM* 28.1 (1981), pp. 1–4.

[128] J. Fearnley, S. Jain, S. Schewe, F. Stephan, and D. Wojtczak. "An ordered approach to solving parity games in quasi polynomial time and quasi linear space". In: *SPIN*. ACM, 2017, pp. 112–121. ISBN: 978-1-4503-5077-8.

[129] J. Filar and K. Vrieze. *Competitive Markov Decision Processes*. Springer-Verlag, 1997.

[130] J. Fu, V. Ng, F. B. Bastani, and I. Yen. "Simple and Fast Strong Cyclic Planning for Fully-Observable Nondeterministic Planning Problems". In: *IJCAI*. Ed. by T. Walsh. IJCAI/AAAI, 2011, pp. 1949–1954.

[131] R. Gentilini, C. Piazza, and A. Policriti. "Computing strongly connected components in a linear number of symbolic steps". In: *SODA*. 2003, pp. 573–582.

[132] R. Gentilini, C. Piazza, and A. Policriti. "Symbolic Graphs: Linear Solutions to Connectivity Related Problems". In: *Algorithmica* 50.1 (2008), pp. 120–158.

[133] R. Gentilini, C. Piazza, and A. Policriti. "Computing strongly connected components in a linear number of symbolic steps". In: *SODA*. 2003, pp. 573–582.

[134] C. Guestrin, D. Koller, R. Parr, and S. Venkataraman. "Efficient Solution Algorithms for Factored MDPs". In: *Journal Artificial Intelligence Research* 19 (2003), pp. 399–468.

[135] K. Hanauer, M. Henzinger, and C. Schulz. *Recent Advances in Fully Dynamic Graph Algorithms*. 2021. arXiv: 2102.11169 [cs.DS].

[136] E. A. Hansen and S. Zilberstein. "Heuristic Search in Cyclic AND/OR Graphs". In: *AAAI*. 1998, pp. 412–418.

[137] J. Heath, M. Z. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn. "Probabilistic model checking of complex biological pathways". In: *Theor. Comput. Sci.* 391.3 (2008), pp. 239–257.

[138] M. R. Henzinger and J. A. Telle. "Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning". In: *SWAT*. 1996.

[139] M. R. Henzinger, V. King, and T. J. Warnow. "Constructing a Tree from Homeomorphic Subtrees, with Applications to Computational Evolutionary Biology". In: *Algorithmica* 24.1 (1999), pp. 1–13.

[140] M. Henzinger, S. Krinninger, and V. Loitzenbauer. "Finding 2-Edge and 2-Vertex Strongly Connected Components in Quadratic Time". In: *ICALP*. Vol. 9134. Lecture Notes in Computer Science. Springer, 2015, pp. 713–724.

[141] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak. "Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture". In: *STOC*. 2015, pp. 21–30.

[142] T. A. Henzinger, O. Kupferman, and S. K. Rajamani. "Fair Simulation". In: *Information and Computation* 173.1 (2002), pp. 64–81.

[143] T. A. Henzinger, R. Majumdar, and J. Raskin. "A classification of symbolic transition systems". In: *ACM Trans. Comput. Log.* 6.1 (2005), pp. 1–32.

[144] J. Hoffmann and R. Brafman. "Contingent Planning via Heuristic Forward Search with Implicit Belief States". In: *ICAPS*. 2005, pp. 71–88.

[145] G. J. Holzmann. "The Model Checker SPIN". In: *IEEE Trans. Softw. Eng.* 23.5 (1997), pp. 279–295.

[146] H. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.

[147] N. Immerman. "Number of Quantifiers is Better Than Number of Tape Cells". In: *J. Comput. Syst. Sci.* (1981).

[148] R. Impagliazzo and R. Paturi. "Complexity of k-SAT". In: *CCC*. 1999, pp. 237–240.

[149] R. Impagliazzo, R. Paturi, and F. Zane. "Which problems have strongly exponential complexity?" In: *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*. IEEE. 1998, pp. 653–662.

[150] B. Jobstmann, A. Griesmayer, and R. Bloem. "Program Repair as a Game". In: *CAV*. 2005, pp. 226–238.

[151] M. Jurdziński. "Small Progress Measures for Solving Parity Games". In: *STACS*. 2000, pp. 290–301.

[152] M. Jurdziński and R. Lazic. "Succinct progress measures for solving parity games". In: *LICS*. 2017, pp. 1–9.

[153] M. Jurdzinski and R. Morvan. "A Universal Attractor Decomposition Algorithm for Parity Games". In: *CoRR* abs/2001.04333 (2020). arXiv: 2001.04333.

[154] M. Jurdzinski, M. Paterson, and U. Zwick. "A Deterministic Subexponential Algorithm for Solving Parity Games". In: *SIAM J. Comput.* 38.4 (2008), pp. 1519–1532.

[155] G. Kant and J. van de Pol. "Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games". In: *GRAPHITE 2012*. 2012, pp. 50–65.

[156] G. Kant and J. van de Pol. "Generating and Solving Symbolic Parity Games". In: *GRAPHITE 2014*. 2014, pp. 2–14.

[157] T. Keller and P. Eyerich. "PROST: Probabilistic Planning Based on UCT". In: *ICAPS*. Ed. by L. McCluskey, B. C. Williams, J. R. Silva, and B. Bonet. 2012, pp. 119–127.

[158] V. King, O. Kupferman, and M. Y. Vardi. "On the Complexity of Parity Word Automata". In: *FOSSCS*. Ed. by F. Honsell and M. Miculan. Vol. 2030. Lecture Notes in Computer Science. Springer, 2001, pp. 276–286.

[159] A. Kolobov, Mausam, D. S. Weld, and H. Geffner. "Heuristic Search for Generalized Stochastic Shortest Path MDPs". In: *ICAPS*. 2011, pp. 130–137.

[160] Z. Komárková and J. Kretínský. "Rabinizer 3: Safraless Translation of LTL to Small Deterministic Automata". In: *ATVA*. 2014, pp. 235–241.

[161] A. Kozachinskiy. "Polyhedral Value Iteration for Discounted Games and Energy Games". In: *SODA*, pp. 600–616.

[162] D. Kozen. "Results on the propositional μ-calculus". In: *Theoretical Computer Science* 27.3 (1983), pp. 333–354.

[163] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. "Temporal-Logic-Based Reactive Mission and Motion Planning". In: *IEEE Transactions on Robotics* 25.6 (2009), pp. 1370–1381.

[164] J. Kretínský, G. A. Pérez, and J. Raskin. "Learning-Based Mean-Payoff Optimization in an Unknown MDP under Omega-Regular Constraints". In: *CONCUR*. 2018, 8:1–8:18.

[165] M. Kronegger, A. Pfandler, and R. Pichler. "Parameterized Complexity of Optimal Planning: A Detailed Map". In: *IJCAI*. 2013, pp. 954–961. ISBN: 978-1-57735-633-2.

[166] O. Kupferman, N. Piterman, and M. Y. Vardi. "From liveness to promptness". In: *Formal Methods Syst. Des.* 34.2 (2009), pp. 83–103.

[167] O. Kupferman and M. Y. Vardi. "Weak Alternating Automata and Tree Automata Emptiness". In: *STOC*. 1998, pp. 224–233.

[168] M. Z. Kwiatkowska, G. Norman, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems". In: *CAV*. 2011, pp. 585–591.

[169] M. Z. Kwiatkowska, G. Norman, and D. Parker. "Probabilistic verification of Herman's self-stabilisation algorithm". In: *Formal Aspects Comput.* 24.4-6 (2012), pp. 661–670.

[170] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. "Planning and acting in partially observable stochastic domains". In: *Artificial Intelligence* 101.1 (1998), pp. 99–134.

[171] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[172] F. Le Gall. "Powers of Tensors and Fast Matrix Multiplication". In: *ISSAC*. 2014, pp. 296–303.

[173] K. Lehtinen. "A modal μ perspective on solving parity games in quasi-polynomial time". In: *LICS*. 2018, pp. 639–648.

[174] K. Lehtinen and U. Boker. "Register Games". In: *Log. Methods Comput. Sci.* 16.2 (2020).

[175] Y. M. Lifshits and D. S. Pavlov. "Potential theory for mean payoff games". In: *Journal of Mathematical Sciences* 145.3 (2007), pp. 4967–4974. ISSN: 1573-8795.

[176] O. Lijzenga and T. van Dijk. "Symbolic Parity Game Solvers that Yield Winning Strategies". In: *Proceedings 11th International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2020, Brussels, Belgium, September 21-22, 2020*. Vol. 326. EPTCS. 2020, pp. 18–32.

[177] A. Lincoln, V. V. Williams, and R. R. Williams. "Tight Hardness for Shortest Cycles and Paths in Sparse Graphs". In: *SODA*. 2018, pp. 1236–1252.

[178] V. Loitzenbauer. "Improved Algorithms and Conditional Lower Bounds for Problems in Formal Verification and Reactive Synthesis". PhD thesis. University of Vienna, 2016.

[179] A. Mahanti and A. Bagchi. "AND/OR Graph Heuristic Search Methods". In: *J. ACM* 32.1 (1985), pp. 28–51.

[180] Z. Manna and A. Pnueli. "Temporal Verification of Reactive Systems: Progress (Draft)". `http://theory.stanford.edu/~zm/tvors3.html`. 1996.

[181] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. New York: Springer-Verlag, 1992.

[182] D. A. Martin. "Borel determinacy". In: *Annals of Mathematics* 102(2) (1975), pp. 363–371.

[183] R. Mattmüller, M. Ortlieb, M. Helmert, and P. Bercher. "Pattern Database Heuristics for Fully Observable Nondeterministic Planning". In: *ICAPS*. Ed. by R. I. Brafman, H. Geffner, J. Hoffmann, and H. A. Kautz. AAAI, 2010, pp. 105–112.

[184] R. McNaughton. "Infinite games played on finite graphs". In: *Annals of Pure and Applied Logic* 65.2 (1993), pp. 149–184.

[185] R. Milner. "An algebraic definition of simulation between programs". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 1971, pp. 481–489.

[186] C. J. Muise, V. Belle, and S. A. McIlraith. "Computing Contingent Plans via Fully Observable Non-Deterministic Planning". In: *AAAI*. 2014, pp. 2322–2329.

[187] C. J. Muise, S. A. McIlraith, and J. C. Beck. "Improved Non-Deterministic Planning by Exploiting State Relevance". In: *ICAPS*. Ed. by L. McCluskey, B. C. Williams, J. R. Silva, and B. Bonet. AAAI, 2012, pp. 172–180.

[188] G. Norman, D. Parker, M. Z. Kwiatkowska, S. K. Shukla, and R. Gupta. "Using probabilistic model checking for dynamic power management". In: *Formal Aspects Comput.* 17.2 (2005), pp. 160–176.

[189] G. Norman and V. Shmatikov. "Analysis of probabilistic contract signing". In: *J. Comput. Secur.* 14.6 (2006), pp. 561–589.

[190] H. Palacios and H. Geffner. "From Conformant into Classical Planning: Efficient Translations that May Be Complete Too." In: *ICAPS*. 2007, pp. 264–271.

[191] C. H. Papadimitriou and J. N. Tsitsiklis. "The Complexity of Markov Decision Processes". In: *Mathematics of Operations Research* 12.3 (1987), pp. 441–450.

[192] P. Parys. "Parity Games: Zielonka's Algorithm in Quasi-Polynomial Time". In: *MFCS*. Vol. 138. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 10:1–10:13.

[193] A. Pnueli and R. Rosner. "On the synthesis of a reactive module". In: *POPL*. 1989, pp. 179–190.

[194] A. Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57.

[195] M. Puterman. *Markov Decision Processes*. John Wiley and Sons, 1994.

[196] J. Queille and J. Sifakis. "Specification and verification of concurrent systems in CESAR". In: *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*. Ed. by M. Dezani-Ciancaglini and U. Montanari. Vol. 137. Lecture Notes in Computer Science. Springer, 1982, pp. 337–351.

[197] M. O. Rabin. *Automata on Infinite Objects and Church's Problem*. USA: American Mathematical Society, 1972. ISBN: 0821816632.

[198] P. Ramadge and W. M. Wonham. "Supervisory control of a class of discrete-event processes". In: *SIAM J. Control Optim.* 25.1 (1987), pp. 206–230.

[199] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366. ISSN: 00029947.

[200] S. J. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education, 2010.

[201] S. Maliah, R. Brafman, E. Karpas, and G. Shani. "Partially Observable Online Contingent Planning Using Landmark Heuristics". In: *ICAPS*. 2014, pp. 163–171.

[202] S. Safra. "Complexity of automata on infinite objects". PhD thesis. Weizmann Institute of Science, 1989.

[203] S. Safra. "On the complexity of ω-automata". In: *FOCS*. 1988, pp. 319–327.

[204] L. Sanchez, J. Wesselink, and T. Willemse. *BDD-based parity game solving: a comparison of Zielonka's recursive algorithm, priority promotion and fixpoint iteration*. Computer science reports. Technische Universiteit Eindhoven, 2018.

[205] L. Sanchez, W. Wesselink, and T. A. C. Willemse. "A Comparison of BDD-Based Parity Game Solvers". In: *GanALF*. Vol. 277. EPTCS. 2018, pp. 103–117.

[206] P. Sanders. "Algorithm Engineering - An Attempt at a Definition". In: *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*. Ed. by S. Albers, H. Alt, and S. Näher. Vol. 5760. Lecture Notes in Computer Science. Springer, 2009, pp. 321–340.

[207] T. Saranurak and D. Wang. "Expander Decomposition and Pruning: Faster, Stronger, and Simpler". In: *SODA*. Ed. by T. M. Chan. SIAM, 2019, pp. 2616–2635.

[208] S. Schewe. "Solving Parity Games in Big Steps". In: *Journal of Computer and Systems Science* 84 (2017). Announced at FSTTCS'07, pp. 243–262.

[209] R. Segala. "Modeling and Verification of Randomized Distributed Real-Time Systems". PhD thesis. MIT, 1995. Technical Report MIT/LCS/TR-676.

[210] H. Seidl. "Fast and Simple Nested Fixpoints". In: *Information Processing Letters* 59.6 (1996), pp. 303–308.

[211] F. Somenzi. "Binary Decision Diagrams". In: *Calculational System Design*. 1999, pp. 303–366.

[212] F. Somenzi. "Colorado University decision diagram package". `http://vlsi.colorado.edu/pub/`. 1998.

[213] A. D. Stasio, A. Murano, G. Perelli, and M. Y. Vardi. "Solving Parity Games Using an Automata-Based Algorithm". In: *CIAA*. 2016, pp. 64–76.

[214] A. D. Stasio, A. Murano, and M. Y. Vardi. "Solving Parity Games: Explicit vs Symbolic". In: *CIAA*. 2018, pp. 159–172.

[215] G. Tan. *A Collection of Well-Known Software Failures*. `http://www.cse.psu.edu/~gxt29/bug/softwarebug.html`. Accessed: 2021-2-15.

[216] R. E. Tarjan. "Depth First Search and Linear Graph Algorithms". In: *SIAM Journal of Computing* 1.2 (1972), pp. 146–160.

[217] R. E. Tarjan. "A Hierarchical Clustering Algorithm Using Strong Components". In: *Inf. Process. Lett.* 14.1 (1982), pp. 26–29.

[218] F. Teichteil-Königsbuch. "Stochastic Safest and Shortest Path Problems". In: *AAAI*. Ed. by J. Hoffmann and B. Selman. 2012, pp. 1826–1831.

[219] W. Thomas. *Languages, Automata, and Logic*. Springer, 1997, pp. 389–455.

[220] A. M. Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *Proceedings of the London mathematical society* 2.1 (1937), pp. 230–265.

[221] M. Y. Vardi. "Automatic Verification of Probabilistic Concurrent Finite-State Programs". In: *FOCS*. 1985, pp. 327–338.

[222] V. Vassilevska-Williams. "On some fine-grained questions in algorithms and complexity". In: *ICM*. 2018.

[223] S. Vester. "Winning Cores in Parity Games". In: *LICS*. 2016, pp. 662–671.

[224] J. Vöge and M. Jurdziński. "A Discrete Strategy Improvement Algorithm for Solving Parity Games". In: *CAV*. 2000, pp. 202–215.

[225] A. Wijs, J. Katoen, and D. Bosnacki. "Efficient GPU algorithms for parallel decomposition of graphs into strongly connected and maximal end components". In: *Formal Methods Syst. Des.* 48.3 (2016), pp. 274–300.

[226] R. Williams. "A new Algorithm for optimal 2-Constraint Satisfaction and its Implications". In: *Theoretical Computer Science* 348.2-3 (2005), pp. 357–365.

[227] V. V. Williams. "Multiplying Matrices Faster than Coppersmith-Winograd". In: *STOC*. 2012, pp. 887–898.

[228] V. V. Williams and R. R. Williams. "Subcubic Equivalences Between Path, Matrix, and Triangle Problems". In: *J. ACM* 65.5 (2018), 27:1–27:38.

[229] W. Zielonka. "Infinite games on finitely coloured graphs with applications to automata on infinite trees". In: *Theoretical Computer Science* 200.1–2 (1998), pp. 135–183.