`INTERLEAVE` : An Empirically Faster Symbolic Algorithm for Maximal End Component Decomposition of MDPs

Thesis submitted by

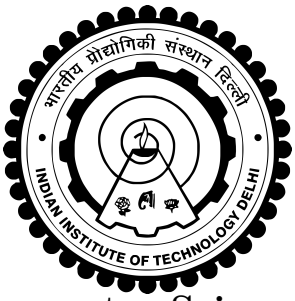
Ramneet Singh
2019CS50445

under the guidance of

Prof. Suguman Bansal, Georgia Institute of Technology
Prof. Subodh Sharma, Indian Institute of Technology Delhi

in partial fulfilment of the requirements
for the award of the degree of

Bachelor and Master of Technology



Department Of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY DELHI

June 2024

THESIS CERTIFICATE

This is to certify that the thesis titled **`INTERLEAVE` : An Empirically Faster Symbolic Algorithm for Maximal End Component Decomposition of MDPs**, submitted by **Ramneet Singh (2019CS50445)**, to the Indian Institute of Technology, Delhi, for the award of the degree of **Bachelor and Master of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Subodh Sharma
Associate Professor
Dept. of CSE
IIT-Delhi, 110 016

Place: New Delhi

Date: 26th June 2024

ACKNOWLEDGEMENTS

There are many people I need to thank for their kindness and support while I have been working on this thesis. First, I thank my family, who always stood by me and tolerated me through hard times. Mom – thank you for loving me unconditionally and being my rock through whatever problems I face. Dad – thank you for showing me how to lead a life of dignity, and for all the good food, sports and music we enjoy together. Avi – 5 minutes with you can brighten my darkest days. Thank you for matching my crazy with your calm (and sometimes your crazy). Mom, Dad and Avi – I love you.

In the last 2.5 years, my paternal grandfather, maternal grandfather and great grandmother passed away, two of them while I was working on my thesis. I am the person I am because of them, because of all the love they gave me and how they lived their lives. My grandmothers continue to give me that love today, and I am extremely grateful for it.

I thank my advisor, Prof. Suguman Bansal, for going out of her way to support me and giving me the opportunity to visit Georgia Tech and work on the thesis. I have learnt a great deal from her, about work-ethic, how to frame a research problem, how to break it down into small achievable goals, and much more. I am also grateful to my co-advisor at IIT Delhi, Prof. Subodh Sharma, who ensured that I could visit Georgia Tech without any unforeseen problems. I have been working with him since the end of my first year, on a variety of things including research projects, apps, and course projects. He has always been helpful and kind, and dispensed valuable advice whenever I asked for it. As any college student would attest, this support means a lot. I am also grateful to Prof. S. Arun-Kumar, Prof. Sorav Bansal and Prof. Kumar Madhukar at IITD for inspiring me to work in the area of Programming Languages.

I thank the GT-PLSE group for making me feel welcome during my visit and showing me lots of cool work. I thank David Parker for his gracious help in understanding how symbolic algorithms are implemented in probabilistic model checkers. It would have been significantly harder to implement my algorithm without his help.

Many friends were there for me and helped, probably much more than they might know. Shaokai Lin was very kind and helped me deal with the lows that inevitably come during research. Ruhanshi Barad gave me invaluable support in dealing with the new environment and made me feel at home. Some others who I turned to for critical support include Mrunmayi, Siddhesh, Devanshu, Ribhav, Niket and Mudit. To them and the many, many others whose name I cannot fit here: thank you.

Dedicated to my *nanu, Maj. Gen. Gurmeet Singh Narula (Retd.) (02 May, 1952 – 02 February, 2023)*, and *dadu, Sardar Jagjot Singh Kumar (06 March, 1942 – 17 December, 2021)*, who will always live in me.

ABSTRACT

KEYWORDS: Probabilistic Model Checking ; Symbolic Algorithms ; Maximal End Components ; Markov Decision Processes ; Binary Decision Diagrams

The model checking problem takes as input a *model* and a *specification*, and outputs whether the model *satisfies* the specification. Probabilistic model checking deals with input models that are probabilistic, e.g., Markov Chains (MCs) and Markov Decision Processes (MDPs). A fundamental algorithmic problem arising in probabilstic model checking is the Maximal End Component (MEC) decomposition of an MDP, with applications including LTL verification and learning-based verification of MDPs. To deal with the state-space explosion problem, modern probabilistic model checkers often use a symbolic representation of MDPs, leading to a need for symbolic algorithms.

We present a novel symbolic algorithm, called `INTERLEAVE`, for the MEC Decomposition of an MDP. For an input MDP with n vertices and m edges, the theoretically-fastest algorithm takes $\widetilde{O}(n^{1.5})$ symbolic operations and $\widetilde{O}(\sqrt{n})$ symbolic space. `INTERLEAVE` is theoretically slower, taking $O(n^2)$ symbolic operations and $O(\log n)$ symbolic space. However, a previous empirical evaluation has shown that the theoretical improvements in worst-case number of symbolic operations do not translate to empirical improvements in running time. We implement `INTERLEAVE` in the `STORM` probabilistic model checker and perform an experimental evaluation on the Quantitative Verification Benchmark Set. We compare it to previous symbolic MEC Decomposition algorithms (excluding one which doesn't work on the same representation, but was shown to be no faster than another previously), and show that it solves 19 **more benchmarks** (out of 379) than the closest previous algorithm given the same timeout. Among the ones that both can solve, we demonstrate a **3.81× average speedup** in decomposition time.

Contents

ACKNOWLEDGEMENTS i

ABSTRACT iii

LIST OF TABLES vi

LIST OF FIGURES vii

ABBREVIATIONS viii

NOTATION ix

1 Introduction 1
- 1.1 Related Work . . . 2
- 1.2 Contributions of this Thesis . . . 4
- 1.3 Structure of the Thesis . . . 4

2 Preliminaries 5
- 2.1 MDPs and MECs . . . 5
  - 2.1.1 Basic Definitions . . . 5
  - 2.1.2 Algorithmic Concepts . . . 9
  - 2.1.3 Graph Representations of MDPs . . . 10
- 2.2 Binary Decision Diagrams . . . 13
  - 2.2.1 Definition . . . 13
  - 2.2.2 Reduced and Ordered BDDs . . . 13
  - 2.2.3 Operations on ROBDDs . . . 14

3 Sparse and Symbolic MDP Storage 16
- 3.1 Sparse Storage . . . 16
- 3.2 Symbolic Storage . . . 17

4 Symbolic MEC Decomposition Algorithms 20
- 4.1 Symbolic Algorithms Formalism . . . 20
  - 4.1.1 Definition . . . 20
  - 4.1.2 Symbolic Time and Space Complexity . . . 20
- 4.2 Omitted Functions . . . 21
- 4.3 `NAIVE` Symbolic MEC Decomposition . . . 21
- 4.4 `SKELETON` Symbolic SCC Decomposition . . . 23
- 4.5 `INTERLEAVE` Symbolic MEC Decomposition . . . 26
  - 4.5.1 Informal Development . . . 26
  - 4.5.2 Pseudocode . . . 27
  - 4.5.3 Complexity Analysis . . . 31

4.5.4 Correctness Proof 32

5 Evaluation **40**
5.1 Setup 41
5.2 MEC Decomposition Performance 41
5.2.1 Quantile Plot for Runtime 42
5.2.2 Pairwise Runtime Comparison on Individual Benchmarks 43
5.2.3 Quantile Plot for Number of Symbolic Operations 45
5.2.4 Pairwise Symbolic Operations Comparison on Individual Benchmarks 47
5.2.5 Algorithmic Analysis of `INTERLEAVE`'s Performance 49

6 Conclusion and Future Work **51**

List of Tables

1.1 Symbolic Algorithms for MEC Decomposition : Theoretical Complexity . . 3

5.1 Summary of the Number of QVBS Benchmarks Solved by Different Algorithms Within Timeout 42

List of Figures

2.1 An Example MDP 6
2.2 End Components in an Example MDP 7
2.3 Maximal End Components in an Example MDP 8
2.4 Different representations (2.4b : Edge-Based Actions and 2.4c : Vertex-Based Actions) of the same original MDP 2.4a 11
2.5 Example (unreduced) BDD for $f(x_1, x_2, x_3) = ((\neg x_2) \vee x_3)$ 13
2.6 Example Reduced BDD for $f(x_1, x_2, x_3) = ((\neg x_2) \vee x_3)$ 14
2.7 Summary of Basic Operations on ROBDDs and their Time Complexities (Source : [HR04]) 14

3.1 Transition Matrices for (3.1a : Vertex-Based Actions and 3.1b : Edge-Based Actions) (Source: Faber's Thesis [Fab23b]) 17

4.1 Example Execution of `NAIVE` (Source : Faber's Thesis [Fab23b]) 23
4.2 Execution of `INTERLEAVE` on an Example MDP 30

5.1 Quantile Plot of the Runtimes of Different Symbolic MEC Decomposition Algorithms on the QVBS 42
5.2 Scatter Plot of the Runtimes of `NAIVE` and `LOCKSTEP` on Individual QVBS Benchmarks 43
5.3 Scatter Plot of the Runtimes of `LOCKSTEP` and `INTERLEAVE` on Individual QVBS Benchmarks 44
5.4 Scatter Plot of the Runtimes of `NAIVE` and `INTERLEAVE` on Individual QVBS Benchmarks 45
5.5 Quantile Plot of the Number of Symbolic Operations Performed by Different Symbolic MEC Decomposition Algorithms on the QVBS 46
5.6 Scatter Plot of the Number of Symbolic Operations Performed by `NAIVE` and `LOCKSTEP` on Individual QVBS Benchmarks 47
5.7 Scatter Plot of the Number of Symbolic Operations Performed by `LOCKSTEP` and `INTERLEAVE` on Individual QVBS Benchmarks 48
5.8 Scatter Plot of the Number of Symbolic Operations Performed by `NAIVE` and `INTERLEAVE` on Individual QVBS Benchmarks 49

ABBREVIATIONS

MDP	Markov Decision Process
EC	End Component
MEC	Maximal End Component
SCC	Strongly Connected Component
BDD	Binary Decision Diagram
MTBDD	Multi Terminal Binary Decision Diagram
ADD	Algebraic Decision Diagram
ROBDD	Reduced Ordered Binary Decision Diagram

NOTATION

$O(.)$	Big-Oh Complexity Notation
$\widetilde{O}(.)$	Big-Oh Complexity Notation, hiding polylogarithmic factors
f_S	BDD representing set S
G_{VBA}	Vertex-Based Actions Graph Representation
G_{EBA}	Edge-Based Actions Graph Representation
ROut	Random-out
Attr	Attractor

Chapter 1

Introduction

Model checking is used to verify (algorithmically) if a *model*, representing a system, satisfies a *specification*. Models and specifications can be of various types. For instance, *probabilistic* model checking handles probabilistic models, including Discrete Time Markov Chains (DTMCs) and Markov Decision Processes (MDPs), among others. Specifications can include, for example, safety, reachability or liveness properties. In probabilistic systems, the notion of satisfaction may be quantitative instead of qualitative, e.g., the probability that an execution is safe, or reaches a certain state. In this work, we will focus on a fundamental task in MDPs, namely Maximal End Component (MEC) Decomposition, which finds application in the verification of multiple types of specifications. The book, Principles of Model Checking by Baier and Katoen [BK08], provides a comprehensive overview of model checking, including a chapter (chapter 10) on probabilistic model checking.

MDPs consist of states with non-deterministically chosen actions and next states chosen from a probability distribution [Put14]. This makes them a popular choice for modelling systems with both non-determinism and stochasticity. MDPs are widely used to model and solve control problems in stochastic systems [FV96] and planning problems in artificial intelligence [Put14]. In the verification world, they are used to represent randomised distributed algorithms (e.g. leader election, with a coin-toss used to break symmetry), model systems with presence of hardware failure or message loss, and to evaluate their reliability and performance using these models [BK08].

Maximal End Components (MECs) of an MDP are, informally, portions of the MDP which are strongly connected, and where it is possible (if you choose the right actions) to loop infinitely often. The MEC Decomposition problem is a central algorithmic problem in probabilistic model checking [BK08]. It is a part of the core of two leading tools in probabilistic verification - `STORM` [HJK+22] and `PRISM` [KNP11], and has proven useful for a wide variety of problems. Some of them are:

- Almost-sure (with probability 1) reachability sets can be computed in linear time given the MEC decomposition [CDHL16].
- MEC Decomposition is a required pre-processing step in the verification of MDPs with respect to ω-regular properties [BK08]. The maximum or minimum satisfaction probabilities are equal to the max/min reachability probabilities of accepting MECs in a product MDP.
- MEC Decomposition is required to ensure convergence in interval iteration for computing maximum reachability probabilities [HM18, BKL+17]. Interval iteration ensures that the convergence criterion gives a bound on the approximation, and allows an analysis of the convergence rate, both of which are key drawbacks of the popular value iteration

technique.

- applying learning algorithms to verification requires MEC decomposition computation [KPR18].

As the systems being represented become more complex, the number of states and actions in the MDP quickly explodes and places more demands on both memory and time. *Explicit* algorithms, which store and process each state individually, become infeasible when working with, e.g., billions of states, simply because of the requirement to construct the state space. *Symbolic* model checking aims to represent large systems compactly by exploiting structure and regularity, often using Binary Decision Diagrams (BDDs) [Lee59, Ake78, Bry85, Bry92]. In probabilistic systems, Multi-Terminal BDDs (MTBDDs) are used to exploit the structure of MDPs and get an efficient compact representation. After BDD-based model checking had been successful in non-probabilistic systems, the paper, "Symbolic Model Checking of Probabilistic Processes using MTBDDs and the Kronecker Representation" (2000) [KNP02] proposed symbolic techniques for probabilistic model checking and implemented them in an early version of `PRISM`. They continue to be an important part of probabilistic model checkers like `STORM` and `PRISM`.

The use of MTBDDs to represent MDPs created a need for *symbolic* algorithms, which did not have explicit access to the model, but could only access it through the MTBDDs. As the above mentioned applications of MEC Decomposition show, it is a fundamental algorithm for probabilistic model checking, and thus, developing symbolic algorithms for it is an important endeavour. In this thesis, we add an algorithm of our own to this category. We will now briefly survey related work and previous algorithms, followed by a summary of our contributions and the structure of this thesis.

1.1 Related Work

We will broadly mention related work on explicit MEC decomposition algorithms, symbolic MEC and SCC decomposition algorithms, and experimental evaluation of symbolic MEC decomposition algorithms. Here, and for the rest of the thesis, when talking about the complexity of algorithms for an MDP, we will use n to be the number of states plus the number of actions (sometimes called the number of vertices of an MDP), and m to be the number of possible state-action-next state triples plus the number of state-enabled action pairs (sometimes called the number of edges of an MDP). These correspond to the number of edges and vertices in the G_{VBA} representation of an MDP (which we will introduce in subsection 2.1.3), and mention complexity using these to maintain consistency with previous work.

Explicit MEC Decomposition: The classical algorithm for this was given by deAlfaro in his PhD Thesis [Alf98] and takes $O(n.m)$ operations in the worst-case [CH14]. This

bound was improved to $O(m\sqrt{m})$ in [CH11] by introducing a lockstep search based on the depth-first search algorithm for SCC decomposition given by Tarjan [Tar72]. [CH14] gives an $O(n^2)$ operations algorithm, while [CDHS19] gives an $\widetilde{O}(m)$ expected-operations randomised algorithm. In [WKB14], Wijs et al show that an implementation of the basic $O(n^2)$ algorithm on a GPU can empirically show significant speedups when compared to a CPU-based implementation.

Symbolic SCC Decomposition: The `SKELETON` [GPP03] algorithm (symbolically) performs the SCC Decomposition of a graph with n vertices in $O(n)$ number of symbolic steps. This is also provably optimal [CDHL17]. `SKELETON` is used as a component of previous symbolic MEC decomposition algorithms, and it served as the inspiration for our algorithm. The `CHAIN` [LSS+23] algorithm is a simplified version of `SKELETON`, taking $O(n)$ symbolic steps and $O(\log n)$ symbolic space.

Symbolic MEC Decomposition: There are three previous symbolic MEC decomposition algorithms.

1. The first is a symbolic implementation of the classical explicit algorithm in [Alf98]. We will refer to it as the `NAIVE` algorithm. `NAIVE` requires $O(n^2)$ symbolic operations and $O(\log n)$ symbolic space, given an $O(n)$ symbolic operations (and $O(\log n)$ symbolic space) SCC decomposition algorithm (like `SKELETON`) [CHL+18].

2. Based on the explicit algorithm from [CH11], Chatterjee et al introduce a symbolic version of lockstep search, and, using it, a symbolic MEC decomposition algorithm in [CHL+18] that requires $O(n\sqrt{m})$ symbolic operations and $O(\sqrt{m})$ symbolic space. We will refer to this algorithm as `LOCKSTEP`.

3. In [CDHS21], Chatterjee et al break the worst-case $O(n^2)$ symbolic operations bound by providing a parameterised algorithm that focuses on detecting and collapsing ECs quickly. Given an $0 < \epsilon \leq 1/2$, the algorithm requires $\widetilde{O}(n^{2-\epsilon})$ symbolic operations and $\widetilde{O}(n^{\epsilon})$ symbolic space. Setting $\epsilon = 0.5$, they give an $\widetilde{O}(n\sqrt{n})$ symbolic operations and $\widetilde{O}(\sqrt{n})$ symbolic space algorithm. We will refer to this algorithm as `COLLAPSING`.

Table 1.1 below summarises these results, and also includes the algorithm we present in this thesis (called `INTERLEAVE`).

Algorithm	Symbolic Operations	Symbolic Space
`NAIVE`	$O(n^2)$	$O(\log n)$
`LOCKSTEP`	$O(n\sqrt{m})$	$O(\sqrt{m})$
`COLLAPSING`	$\widetilde{O}(n\sqrt{n})$	$\widetilde{O}(\sqrt{n})$
`INTERLEAVE`	$O(n^2)$	$O(\log n)$

Table 1.1: Symbolic Algorithms for MEC Decomposition : Theoretical Complexity

Experimental Evaluation of Symbolic MEC Decomposition: To the best of our

knowledge, Faber's Bachelor Thesis [Fab23b] is the only previous empirical comparison of the different symbolic MEC decomposition algorithms. We have used their implementations (with minor bug-fixes) from [Fab23a] and added an implementation of `INTERLEAVE` for performing our experimental evaluation.

1.2 Contributions of this Thesis

In summary, we make the following contributions:

1. We present a novel symbolic algorithm, called `INTERLEAVE`, for the MEC Decomposition of an MDP. While all previous algorithms perform multiple SCC Decomposition calls during their execution, the key idea in our algorithm is to interleave the MEC computation with the work that a single SCC Decomposition call would do. The algorithm requires $O(n^2)$ symbolic operations and $O(\log n)$ symbolic space for an MDP with n vertices and m edges.

2. We provide an implementation of the `INTERLEAVE` algorithm in the `STORM` probabilistic model checker. The implementation follows the format of [Fab23a], and is available at https://github.com/Ramneet-Singh/storm-masters-thesis/tree/stable.

3. We perform an experimental evaluation of `INTERLEAVE`, comparing it to the `NAIVE` and `LOCKSTEP` algorithms on 379 benchmarks from the Quantitative Verification Benchmark Set (QVBS) [HKP+19]. We do not compare it to `COLLAPSING` since it works on a different graph-based representation than the one `STORM` uses [Fab23b]. However, [Fab23b] showed that even after converting the representations, `NAIVE` performed better than `COLLAPSING` on QVBS. Hence, our experimental evaluation shows that `INTERLEAVE` is the empirically fastest symbolic MEC Decomposition algorithm on QVBS, solving 19 more benchmarks than the closest other algorithm (`NAIVE`) given the same timeout (240 seconds) and achieving an average speedup of 3.81× on the ones that both were able to solve.

1.3 Structure of the Thesis

The first chapter introduces the MEC Decomposition problem along with its applications, surveys related work and existing algorithms, and summarises the contributions of the thesis. We then formally define a few preliminaries, including MDPs, MECs, the graph structures of an MDP and BDDs in the second chapter. Chapter 3 outlines how an MDP is stored – using sparse or symbolic data structures, in contemporary probabilistic model checkers like `STORM`. Chapter 4 introduces symbolic algorithms, explains the key idea behind the `NAIVE`, followed by a detour into the symbolic SCC Decomposition algorithm `SKELETON` and how it led to the `INTERLEAVE` algorithm. The chapter ends with a correctness proof and complexity analysis of `INTERLEAVE`. In Chapter 5, we present our experimental evaluation of `INTERLEAVE` on the QVBS and analyse its runtimes and number of symbolic operations in comparison to the `NAIVE` and `LOCKSTEP` algorithms. Finally, Chapter 6 concludes the thesis, highlighting opportunities for future work.

Chapter 2

Preliminaries

We begin with defining MDPs and stating our assumptions, then define MECs (with other required notions), followed by concepts like attractor, which are used in MEC Decomposition algorithms. We then define two graph-based representations of MDPs, and end with a note on BDDs. In subsection 2.1.1, we also show that the MEC of a state in an MDP, if it exists, is unique (lemma 2.1), ensuring that the problem we are tackling is well-defined.

2.1 MDPs and MECs

2.1.1 Basic Definitions

Definition 2.1 (MDP)**.** A (finite) MDP $\mathcal{M}$ is given by a four-tuple $(S, A, d_{\text{init}}, \delta)$ where S is a finite set of states, A is a finite set of actions, $d_{\text{init}} : S \to [0, 1]$ is an initial probability distribution over states (so $\Sigma_{s \in S} d_{\text{init}}(s) = 1$), and $\delta : S \times A \times S \to [0, 1]$ specifies the next-state distributions for each state and each action (so $\Sigma_{s' \in S} \delta(s, \alpha, s') \in \{0, 1\}$ for all $s \in S, \alpha \in A$).

Figure 2.1 shows an example MDP. The example has been borrowed from Christel Baier's excellent slides on probabilistic model checking for MDPs [Bai17]. The probabilities are omitted where there is only one possible next state for a state and action. For this MDP, $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$ and $A = \{\alpha, \beta, \gamma, \delta, \sigma\}$. The d_{init} is omitted (since MECs do not depend on it), and δ, for e.g., maps (s_0, α, s_1) to $\frac{1}{2}$ and (s_5, β, s_6) to $\frac{1}{4}$.

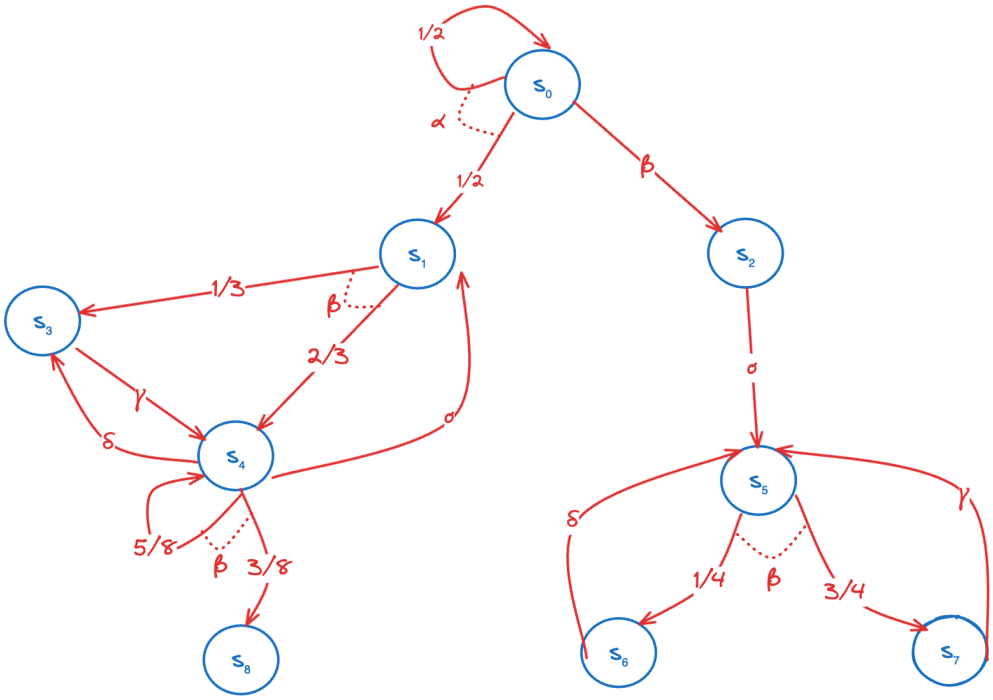


Figure 2.1: An Example MDP

For the remainder of this section, let $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$ be an MDP.

Definition 2.2 (Enabled Actions)**.** We say that an action $\alpha \in A$ is *enabled* in state $s \in S$ if $\Sigma_{s' \in S}\delta(s, \alpha, s') = 1$ (or, equivalently, if $\exists s' \in S \,.\, \delta(s, \alpha, s') > 0$). For an action set $A' \subseteq A$ and state $s \in S$, the set of actions in A' which are enabled in s is denoted by $A'[s]$.

We make the following two assumptions about the MDPs we consider.

Assumption. Every state has at least one enabled action, i.e., $A[s] \neq \emptyset$ for all $s \in S$.

Assumption. Every action is enabled in some state, i.e., $\bigcup_{s \in S} A[s] = A$.

Now we will define maximal end components, the core of the problem we are trying to solve, i.e., the MEC Decomposition of an MDP.

Definition 2.3 (sub-MDP)**.** A sub-MDP of $\mathcal{M}$ is a pair (T, π) where $\emptyset \neq T \subseteq S$ and $\pi : T \to 2^A$ such that:

- $\emptyset \neq \pi(s) \subseteq A[s]$ for all $s \in T$.
- For all $s \in T, \alpha \in \pi(s)$ and $s' \in S$, if $\delta(s, \alpha, s') > 0$, then $s' \in T$.

Definition 2.4 (State-Action Pair Set of a sub-MDP)**.** Let (T, π) be a sub-MDP. We define its state-action pair set, denoted $\text{sa}(T, \pi)$ as

$$\text{sa}(T, \pi) = \{(s, \alpha) \in T \times A \mid \alpha \in \pi(s)\}$$

Definition 2.5 (Inclusion of sub-MDPs)**.** A sub-MDP (T_1, π_1) of $\mathcal{M}$ is said to be *included in*

another sub-MDP (T_2, π_2) of $\mathcal{M}$, denoted $(T_1, \pi_1) \subseteq (T_2, \pi_2)$ if $T_1 \subseteq T_2$, and, for each $s \in T_1$, $\pi_1(s) \subseteq \pi_2(s)$.

Definition 2.6 (Reachability in an MDP)**.** A path of length $n \in \mathbb{N}$ from a state $s \in S$ to a state $s' \in S$ in an MDP $\mathcal{M}$ is given by states $s_0, s_1, \ldots, s_n \in S$ and actions $\alpha_0, \ldots, \alpha_{n-1} \in A$ such that $s_0 = s$, $s_n = s'$, and $\delta(s_i, \alpha_i, s_{i+1}) > 0$ for all $i \in \{0, \ldots, n-1\}$. We say that $s \in S$ reaches $s' \in S$ in $\mathcal{M}$ if there exists a path in $\mathcal{M}$ from s to s'.

Definition 2.7 (Reachability in a sub-MDP)**.** Let (T, π) be a sub-MDP of $\mathcal{M}$. We say that a state $s \in T$ can reach a state $s' \in T$ in (T, π) if there is a path $s_0\alpha_0 s_1\alpha_1 \ldots s_{n-1}\alpha_{n-1}s_n$ in $\mathcal{M}$ from s to s' such that $s_i \in T$ for all $i \in \{0, \ldots, n\}$ and $\alpha_i \in \pi(s_i)$ for all $i \in \{0, \ldots, n-1\}$.

Definition 2.8 (End Component (EC))**.** A sub-MDP (T, π) is called an *end component* of $\mathcal{M}$ if for all $s, t \in T$, s can reach t in (T, π).

As an illustration, figure 2.2 shows the end components in the example MDP from figure 2.1. All the states inside each box form an end-component. For each state, the set of actions which cannot take it outside the end component is included.

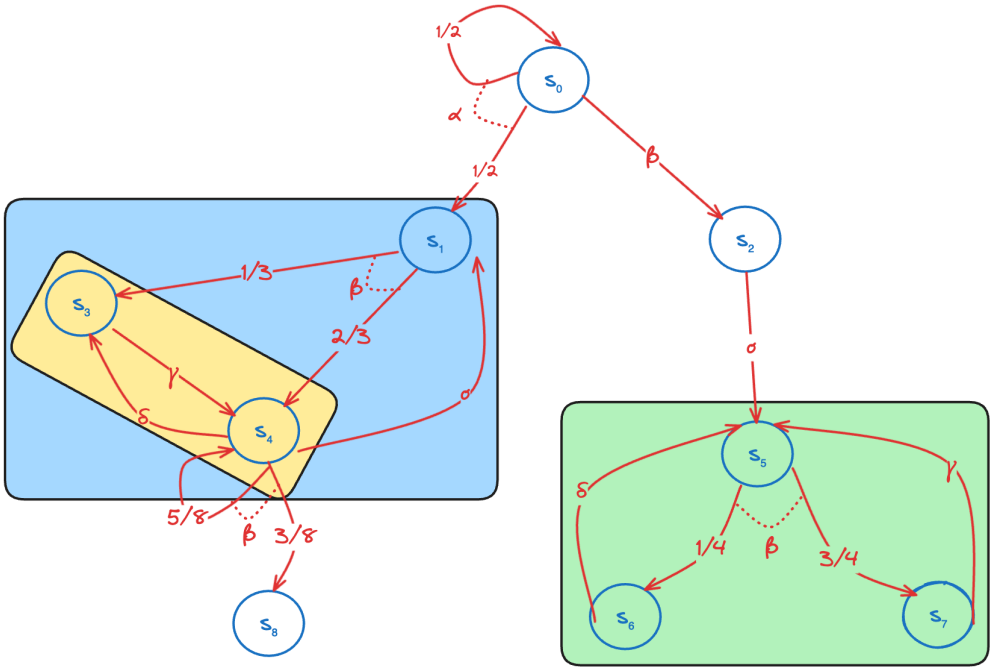


Figure 2.2: End Components in an Example MDP

The yellow end component is included in the blue one. In other words, it is not a *maximal* end component. We define it formally below.

Definition 2.9 (Maximal End Component (MEC))**.** An end component (T, π) of an MDP $\mathcal{M}$ is called a *maximal end component* of $\mathcal{M}$ if it is maximal with respect to sub-MDP inclusion, i.e., there is no end component (T', π') of $\mathcal{M}$ such that $(T, \pi) \subseteq (T', \pi')$ and $(T, \pi) \neq (T', \pi')$.

Figure 2.3 shows the maximal end components of the example MDP from figure 2.1.

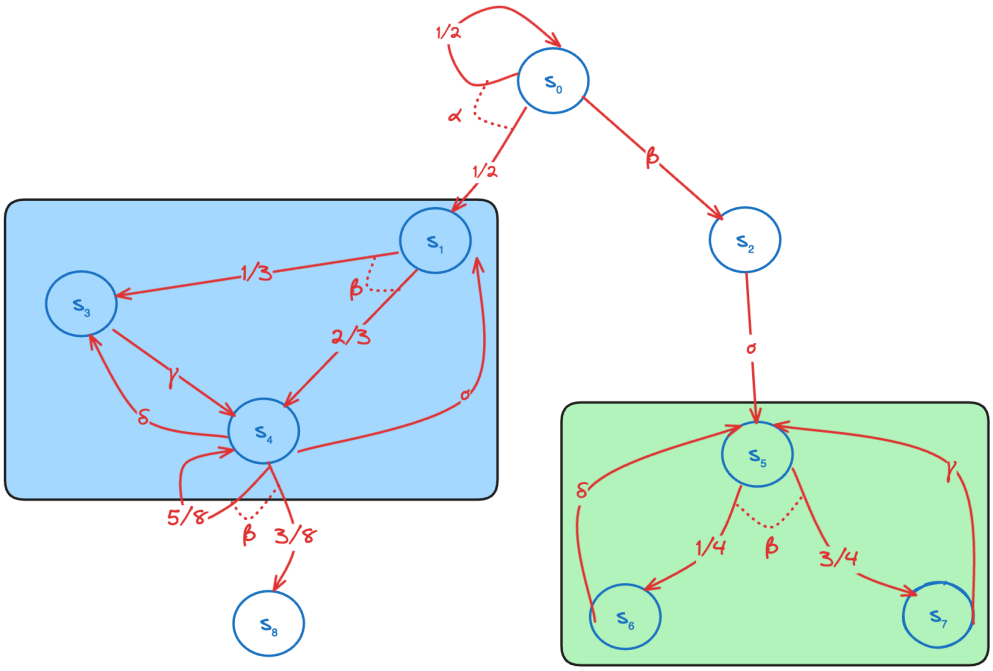


Figure 2.3: Maximal End Components in an Example MDP

We will now show that the MEC each state (and thus, each state-action pair) is in, if it exists, is unique. This ensures that the MEC Decomposition of an MDP is well-defined and unique.

Lemma 2.1 (Uniqueness of MEC)**.** *Every state (and thus, every state-action pair) belongs to at most one MEC.*

Proof. Proof is by contradiction. Assume, if possible, that there is a state $s \in S$ and distinct maximal end components (T_1, π_1) and (T_2, π_2) such that $s \in T_1$ and $s \in T_2$. Consider the pair (T, π), defined as $T = T_1 \cup T_2$ and

$$\pi(s') = \begin{cases} \pi_1(s') & \text{if } s' \in T_1 \setminus T_2 \\ \pi_2(s') & \text{if } s' \in T_2 \setminus T_1 \\ \pi_1(s') \cup \pi_2(s') & \text{otherwise} \end{cases}$$

We will show that (T, π) is a sub-MDP, and, in fact, an end component of $\mathcal{M}$. To see why it is a sub-MDP, note the following:

- Since (T_1, π_1) and (T_2, π_2) are ECs (and therefore sub-MDPs), we have $\emptyset \neq \pi_1(s') \subseteq A[s']$ for all $s' \in T_1$ and $\emptyset \neq \pi_2(s') \subseteq A[s']$ for all $s' \in T_2$. Now, for all $s' \in T = T_1 \cup T_2$, $\pi(s')$ is either $\pi_1(s')$ ($s' \in T_1$ in this case), $\pi_2(s')$ ($s' \in T_2$ in this case) or $\pi_1(s') \cup \pi_2(s')$ ($s' \in T_1 \cap T_2$ in this case). Thus we have $\emptyset \neq \pi(s') \subseteq A[s']$.
- Consider any $s' \in T, \alpha \in \pi(s'), t \in S$ such that $\delta(s', \alpha, t) > 0$. If $s' \in T_1 \setminus T_2$ (resp. $T_2 \setminus T_1$), then since (T_1, π_1) (resp. (T_2, π_2)) is a sub-MDP, $t \in T_1$ (resp. T_2), and thus

in $T_1 \cup T_2$. Otherwise, if $s' \in T_1 \cap T_2$, then $\alpha \in \pi_1(s')$ or $\alpha \in \pi_2(s')$. In either case, because $s' \in T_1$, $s' \in T_2$ and $(T_1, \pi_1), (T_2, \pi_2)$ are sub-MDPs, $t \in T_1 \cup T_2$.

Take any $s_1, s_2 \in T_1 \cup T_2$. We have the following three cases:

- If $s_1, s_2 \in T_1$, then since (T_1, π_1) is an EC, s_1 can reach s_2 in (T_1, π_1). Since $T_1 \subseteq T_1 \cup T_2$ and $\pi_1(s') \subseteq \pi(s') \forall\, s' \in T_1$, this implies s_1 can also reach s_2 in $(T_1 \cup T_2, \pi)$.
- Similarly, if $s_1, s_2 \in T_2$, then since (T_2, π_2) is an EC, s_1 can reach s_2 in (T_2, π_2). Since $T_2 \subseteq T_1 \cup T_2$ and $\pi_2(s') \subseteq \pi(s') \forall\, s' \in T_2$, this implies s_1 can also reach s_2 in $(T_1 \cup T_2, \pi)$.
- Wlog, suppose $s_1 \in T_1$ and $s_2 \in T_2$. Then, since $s \in T_1$ and (T_1, π_1) is an EC, there is a path from s_1 to s using only states from T_1 and actions from π_1. Similarly, since $s \in T_2$ and (T_2, π_2) is an EC, there is a path from s to s_2 using only states from T_2 and actions from π_2. The concatenation of these is a path from s_1 to s_2 using only states from $T_1 \cup T_2$ and actions from π. Thus, s_1 can reach s_2 in $(T_1 \cup T_2, \pi)$.

Since (T_1, π_1) and (T_2, π_2) are distinct, (T, π) must strictly include one of them. But we have shown that (T, π) is an end component of $\mathcal{M}$, which contradicts the fact that (T_1, π_1) and (T_2, π_2) are maximal end components. □

Definition 2.10 (MEC of a state and a state set)**.** For any state $s \in S$, if it's part of an MEC, denote the MEC of s (unique by lemma 2.1) by $\text{MEC}_{\mathcal{M}}(s)$. For a set of states $S' \subseteq S$, let $\text{MECs}_{\mathcal{M}}(S')$ be the set of MECs of all states in S' which have an MEC.

2.1.2 Algorithmic Concepts

We define the notion of an MEC-closed sub-MDP below. Informally, these are pieces of the original MDP for which MEC computation can be done independently. This will be useful for the correctness proof of our algorithm.

Definition 2.11 (MEC-closed sub-MDP)**.** A sub-MDP (T, π) is called MEC-closed if for every state $s \in T$, $\text{MEC}_{\mathcal{M}}(s)$, if it is defined, satisfies $\text{MEC}_{\mathcal{M}}(s) \subseteq (T, \pi)$.

We now define some concepts which will be used by our algorithm. The ROut of a state set has been used for multiple previous symbolic MEC Decomposition algorithms [CHL+18, CDHS21]. It is usually computed for a state set that you know is strongly connected, and gives the state-action pairs that can take you out of the state set. As we will show in the correctness proof for our algorithm, these state-action pairs cannot be part of any MEC, and thus can be discarded.

Definition 2.12 (ROut of a state set in a sub-MDP)**.** Let (T, π) be a sub-MDP and $S' \subseteq T$. $\text{ROut}_{(T,\pi)}(S')$ ("random-out" of S' in (T, π)) is defined as the set of state-action pairs in (T, π) which can go outside S'. Formally,

$$\text{ROut}_{(T,\pi)}(S') = \{(s, \alpha) \in S' \times A \mid \alpha \in \pi(s) \wedge \exists s' \in (T \setminus S')\,.\,(\delta(s, \alpha, s') > 0)\}$$

Next, we define the attractor of a state-action pair set in a sub-MDP. This has also been used in previous algorithms [CHL+18, CDHS21]. It is meant to be called on the ROut of a strongly-connected state set. The idea is that if for a state, its pairs with all its enabled actions can't be part of an MEC, then the state can't be either. Similarly, if a state-action pair leads to a state that can't be part of an MEC, then that state-action pair can't be part of the MEC either. Thus, as an optimisation, not only can the ROut be removed, but its attractor can be removed when considering a strongly-connected state set. We will state and prove all of this formally in the correctness proof for our algorithm.

Definition 2.13 (Attractor of a state-action pair set in a sub-MDP)**.** Let (T, π) be a sub-MDP and $X \subseteq \mathrm{sa}(T, \pi)$ be a state-action pair set. The attractor of X in (T, π) is defined as, $\mathrm{Attr}_{(T,\pi)}(X) = (S', X') = (\bigcup_{i\in\mathbb{N}} S_i, \bigcup_{i\in\mathbb{N}} X_i)$, where

- $(S_0, X_0) = (\emptyset, X)$
- For $i \geq 0$,
  - $S_{i+1} = S_i \cup \{s \in T \mid \forall\, \alpha \in \pi(s)\,.\,((s, \alpha) \in X_i)\}$.
  - $X_{i+1} = X_i \cup \{(s, \alpha) \in \mathrm{sa}(T, \pi) \mid \exists\, s' \in S_i\,.\,(\delta(s, \alpha, s') > 0)\}$.

2.1.3 Graph Representations of MDPs

While writing both algorithms and code that deals with MDPs, it is often helpful to think of them as graphs. There are two popular graph-based representations of an MDP, one primarily used in the symbolic MEC decomposition algorithms literature, and one used in probabilistic model checkers like `STORM`. In both these representations, we will see that the transition probability value is abstracted away, and it only matters whether the probability is 0 or > 0. Such an abstraction is possible for a few MDP tasks, like reachability analysis and MEC decomposition. In probabilistic model checkers, the transition probabilities are stored, but there is a way to access this abstract representation for tasks that can make do with it. We will elaborate on how MDPs are stored in the next chapter.

In the G_{EBA} representation (*Edge-Based Actions*), the vertices are the states of the MDP. For each triple (s, α, t) such that $\delta(s, \alpha, t) > 0$, there is a *labelled, directed* edge from s to t, with label α.

Definition 2.14 ($G_{\mathrm{EBA}}(\mathcal{M})$ representation for $\mathcal{M}$)**.** The edge-based actions graphical representation for the MDP $\mathcal{M} = (S, A, d_{\mathrm{init}}, \delta)$ is given by $G_{\mathrm{EBA}}(\mathcal{M}) = (V, E)$ where $V = S$ and $E = \{(s, \alpha, s') \in S \times A \times S \mid \delta(s, \alpha, s') > 0\}$.

In fact, we can (and will) represent sub-MDPs as graphs too.

Definition 2.15 ($G_{\mathrm{EBA}}(T, \pi)$ representation for sub-MDP (T, π))**.** The edge-based actions graphical representation for the sub-MDP (T, π) is given by $G_{\mathrm{EBA}}(T, \pi) = (V, E)$ where $V = T$ and $E = \{(s, \alpha, s') \in T \times A \times T \mid (\alpha \in \pi(s) \wedge \delta(s, \alpha, s') > 0)\}$.

In the G_{VBA} representation (*Vertex-Based Actions*), each state and each action of

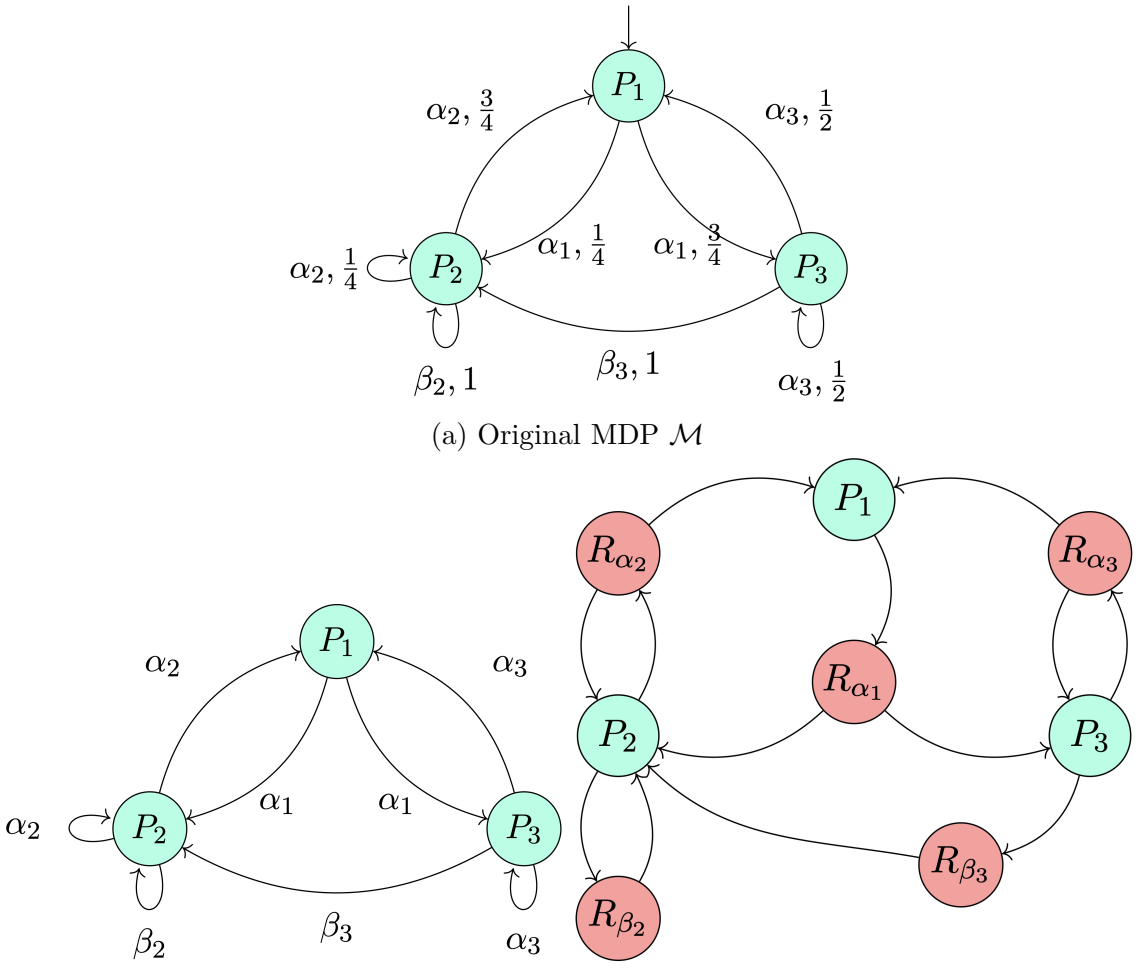
the MDP has a corresponding vertex. For each pair of state s and enabled action $\alpha \in A[s]$,

- there is a *directed* edge from s to α, and
- for each state t such that $\delta(s, \alpha, t) > 0$, there is a *directed* edge from α to t.

Definition 2.16 ($G_{\text{VBA}}(\mathcal{M})$ representation for $\mathcal{M}$)**.** Let $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$ be an MDP. Assume, without loss of generality, that for all distinct states $s, t \in S$ $(s \neq t)$, their enabled actions are distinct, i.e., $A[s] \cap A[t] = \emptyset$. The vertex-based actions graphical representation for the MDP $\mathcal{M}$ is given by $G_{\text{VBA}}(\mathcal{M}) = (V, E)$ where:

- $V = V_P \cup V_R$, where $V_P = S$ are called the player-one vertices, and $V_R = A$ are called the random vertices.
- $E = \{(s, \alpha) \in S \times A \mid \alpha \in A[s]\} \bigcup \{(\alpha, t) \in S \times A \mid \exists\, s \in S\,.\,(\delta(s, \alpha, t) > 0)\}$.

Figure 2.4 shows both the representations for an MDP, to illustrate their differences. The example has been borrowed from Faber's (also excellent) thesis [Fab23b].



(a) Original MDP $\mathcal{M}$

(b) Edge-Based Actions Graph Representation $G_{\text{EBA}}(\mathcal{M})$

(c) Vertex-Based Actions Graph Representation $G_{\text{VBA}}(\mathcal{M})$

Figure 2.4: Different representations (2.4b : Edge-Based Actions and 2.4c : Vertex-Based Actions) of the same original MDP 2.4a

In this thesis, we will use the G_{EBA} graph representation for an MDP, for the following two reasons:

1. The two leading probabilistic model checkers (`STORM` and `PRISM`) use the edge-based actions representation for efficiency reasons. Creating a separate vertex (with its own separate edges too) for each action in an MDP is infeasible. Therefore, by using this representation for our algorithm, it becomes easier to implement inside a real model checker (as we have done).

2. The vertex-based actions representation requires an assumption that no two states in an MDP share an action. In practice, this assumption doesn't hold. It is infeasible to make all actions distinct, and so they are reused whenever possible. While the assumption simplifies algorithm design and analysis, we believe it isn't needed for our algorithm, and thus, we don't make it.

We will often be interested in the predecessors and successors in the graph (V, E) for a set $V' \subseteq V$. Formally, these are defined as:

Definition 2.17 ($\text{Pre}_{(V,E)}(.)$ and $\text{Post}_{(V,E)}(.)$)**.** Let $(V, E) = G_{\text{EBA}}(T, \pi)$ for some sub-MDP (T, π). Then, for any set $V' \subseteq V$, define:

$$\text{Post}_{(V,E)}(V') = \{v_2 \in V \mid \exists v_1 \in V', \alpha \in A \,.\, ((v_1, \alpha, v_2) \in E)\}$$
$$\text{Pre}_{(V,E)}(V') = \{v_1 \in V \mid \exists v_2 \in V', \alpha \in A \,.\, ((v_1, \alpha, v_2) \in E)\}$$

In practice, we may pre-convert the labelled edge relation E to an unlabelled one (say, E') and then use that for each call $\text{Pre}_{(V,E)}(.)$ or $\text{Post}_{(V,E)}$.

Remark. **Note on Algorithm Complexity Notation:** When we talk about the complexity of different symbolic algorithms, we will say things like, "for an MDP with n vertices and m edges, this algorithm requires $O(n^2)$ symbolic operations". Here, the vertices and edges refer to the G_{VBA} representation. In other words, $n = |S| + |A|$ and $m = |\{(s, \alpha, t) \in S \times A \times S \mid \delta(s, \alpha, t) > 0\}|$ (not exactly the number of edges in G_{VBA} but $\Theta(\cdot)$ of that). The prior literature ([CHL+18, CDHS21]) uses this notation and we stay with it to avoid confusion.

Having seen how MDPs are viewed as graphs, the next chapter will explain how they are stored in model checkers. Before we do that, we will take a detour and provide a brief explanation of Binary Decision Diagrams – the tools typically used for symbolically storing MDPs.

2.2 Binary Decision Diagrams

2.2.1 Definition

Binary Decision Diagrams are data structures that can represent arbitrary boolean-valued[1] functions whose input is a sequence of boolean variables. They were first developed for logic circuit modelling [Lee59, Ake78], but have been critical to the success of symbolic model checking, so critical, in fact, that the Model Checking Handbook has a chapter dedicated to them [CG18].

A BDD on n boolean variables $x_1, \ldots, x_n$ represents a function $f : \{0,1\}^n \to \{0,1\}$. Formally, a BDD is a Directed Acyclic Graph such that:

- There is a unique root node.
- Each non-terminal node is labelled with a variable from $\{x_1, \ldots, x_n\}$.
- Each terminal node is labelled with a 0 or 1.
- Each non-terminal node has only two outgoing edges, labelled 0 and 1.

In order to get a value of f at a point $(b_1, \ldots, b_n)$, you start at the root node, and, until you reach a terminal node, repeat:

- Suppose the non-terminal node is labelled with variable x_i. If $b_i = 0$, take the edge labelled 0, otherwise take the edge labelled 1.

The value of the terminal node you reach gives the function's value at that point.

2.2.2 Reduced and Ordered BDDs

It is trivial to come up with a binary decision tree for $f(x_1, \ldots, x_n)$. For example, with $n = 3$ and $f(x_1, x_2, x_3) = ((\neg x_2) \vee x_3)$, it might look like figure 2.5.

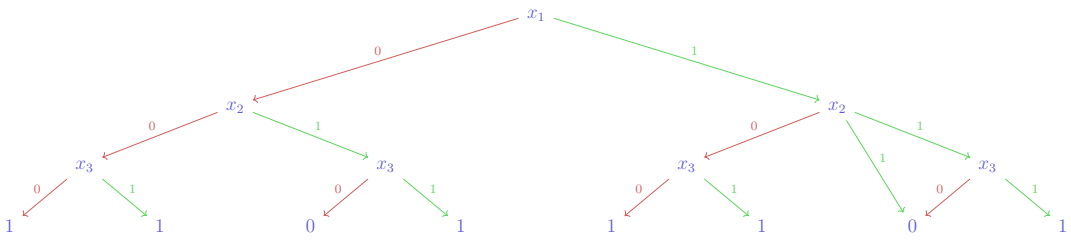


Figure 2.5: Example (unreduced) BDD for $f(x_1, x_2, x_3) = ((\neg x_2) \vee x_3)$

This is, however, not the most efficient representation. *Reduced* binary decision diagrams aim to improve this. To reduce a BDD, you take the following steps:

1. **Remove duplicate terminals:** Merge all 0 terminal nodes into a single node, and all 1 terminal nodes into a single node.

[1] and, with slight changes, integer or real-valued too

2. **Remove redundant tests:** If the 0 and 1 outgoing edges for a node go to the same subgraph, then remove that node.
3. **Remove duplicate nodes:** If the subgraphs headed at two different nodes are identical, then merge them into a single node.

Efficient algorithms exist in practice to perform `reduce`(B) in $O(|B|.\log|B|)$ time. After reducing the earlier BDD for $f(x_1, x_2, x_3) = ((\neg x_2) \vee x_3)$, we would get the reduced BDD in figure 2.6.

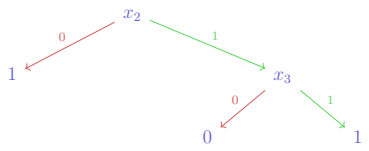


Figure 2.6: Example Reduced BDD for $f(x_1, x_2, x_3) = ((\neg x_2) \vee x_3)$

Reduced *Ordered* BDDs (ROBDDs) further impose an ordering on the variables. Given an ordering $x_1, \ldots, x_n$, there is no path in a Reduced Ordered BDD where a node labelled with x_i is followed by a node labelled with x_j and $j <= i$. Given an ordering, there is a unique ROBDD for a boolean function, and the choice of ordering can have significant impacts on the size of the ROBDD [HR04].

2.2.3 Operations on ROBDDs

Some basic operations on ROBDDs are:

- Binary logical operations on the functions reprsented by them. For example, if B_f and B_g are the ROBDDs representing functions f and g, then `apply`$(\circ, B_f, B_g)$ outputs the reduced ordered BDD representing $f \circ g$ (here $\circ$ could be any binary operation on $\{0, 1\}$ like $\vee, \wedge$ etc.).
- `restrict`(b, x, B_f) computes the ROBDD for $f[b/x]$, where $b \in \{0, 1\}$.
- `exists`(x, B_f) computes the ROBDD for $\exists\, x\,.\, f$.

The following table (figure 2.7) from Huth and Ryan's book on Logic In Computer Science [HR04] summarises their complexities.

Algorithm	Input OBDDs	Output OBDD	Time complexity
`reduce`	B	reduced B	$O(\lvert B\rvert \cdot \log \lvert B\rvert)$
`apply`	B_f, B_g (reduced)	$B_{f \square g}$ (reduced)	$O(\lvert B_f\rvert \cdot \lvert B_g\rvert)$
`restrict`	B_f (reduced)	$B_{f[0/x]}$ or $B_{f[1/x]}$ (red'd)	$O(\lvert B_f\rvert \cdot \log \lvert B_f\rvert)$
$\exists$	B_f (reduced)	$B_{\exists x_1 \ldots x_n . f}$ (reduced)	NP-complete

Figure 2.7: Summary of Basic Operations on ROBDDs and their Time Complexities (Source : [HR04])

Apart from this, we will use a few more operations on ROBDDs, which include:

- (Inverse) Relational product: Let $S \subseteq U$ be a subset of some set U and $R \subseteq U \times U$ be a binary relation on U. Suppose f_S and f_R are the boolean functions representing S and R (we will elaborate on how sets and relations are represented as boolean functions in the next chapter). Then, the relational product of S with R yields the set $S' = \{s' \in U \mid \exists\, s \in S\,.\,(s, s') \in R\}$, and the inverse relational product of S with R is the set $S'' = \{s' \in U \mid \exists\, s \in S\,.\,(s', s) \in R\}$. The (resp. inverse) relational product operation on ROBDDs takes ROBDDs representing f_S and f_R and returns the ROBDD representing $f_{S'}$ (resp. $f_{S''}$).
- Cardinality: When representing a set S using a boolean function f_S and representing that through an ROBDD B_{f_S}, we sometimes want to know the size of S. For the ROBDD, this is equivalent to counting the number of satisfying assignments to its variables.

Finally, we would like to mention that with some changes, BDDs need not be restricted to boolean-valued functions. *Multi-Terminal* BDDs, also called *Algebraic* DDs (ADDs) are DDs where the terminal nodes may have integer or real values [BFG+97, FMY97]. MTBDDs support additional operations like addition, multiplication etc.

Chapter 3

Sparse and Symbolic MDP Storage

We have seen how MDPs can be represented as graphs, and how binary decision diagrams can represent boolean functions. In this chapter, we will combine the two and explain how MDPs are stored symbolically using BDDs in probabilistic model checkers. For the sake of completeness, we will also cover how they are stored explicitly, using sparse data structures. Since our algorithm will use relatively higher-level operations (like Pre, Post), we hope that this chapter gives a peek into how they are implemented in practice.

We assume an MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$. We will show how both sparse and symbolic structures support storing and modifying vertices, edges and actions of the G_{VBA} and G_{EBA} representations. We will also explain how $\text{Pre}_{(V,E)}(V')$ and $\text{Post}_{(V,E)}(V')$ can be implemented. As we remarked earlier, the transition probability values do not matter for the task of MEC decomposition, it only matters if they're non-zero or not. However, we will mention how they are stored too, for the sake of completeness. The descriptions are based on the `STORM` model checker [HJK+22] and on chapter 4 of Faber's thesis [Fab23b].

3.1 Sparse Storage

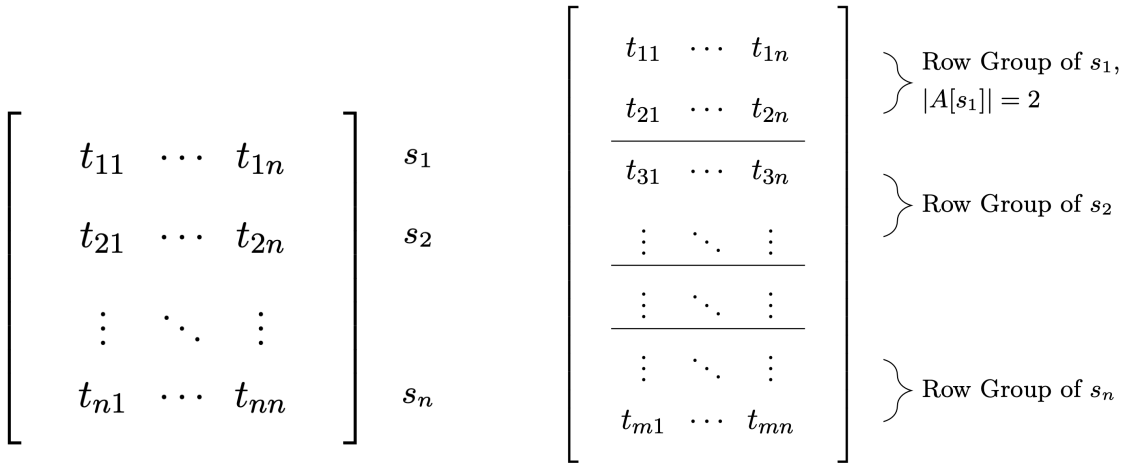
In the sparse style of storage, each state and transition is stored explicitly. For representing a set of vertices of a graph (whether G_{VBA} or G_{EBA}) with n vertices, a bitvector of size n can be stored, with the i-th bit indicating whether the i-th vertex is in the set or not. Basic set operations like intersection, union and complementation can be done using the corresponding logical operations of and, or, not on these bitvectors. Additionally, in the case of G_{VBA}, each vertex needs to have one bit that indicates whether it is a player-1 vertex or a random vertex.

For G_{VBA}, the edges can be stored using an $n \times n$ transition matrix T, where $(T)_{ij} \neq 0$ indicates that s_i has an edge to s_j. If s_i is a player-1 vertex, then $(T)_{ij} \in \{0, 1\}$ indicates whether the action represented by s_j can be chosen from state s_i. Otherwise, s_i is a random vertex (representing an action, say, α), and $(T)_{ij} \in [0, 1]$ represents a transition probability. Note that since G_{VBA} assumes that no two states share an action, there is a unique state s for which $(T)_{ij} = \delta(s, \alpha, s_j)$.

For G_{EBA}, two vertices may have multiple edges between them (labelled with different actions). So it isn't enough to store one value for each pair of vertices. Instead, the transition matrix T here has n "row groups", one for each vertex (equivallent to state in G_{EBA}). The row group for state $s \in S$ has $|A[s]|$ rows, i.e., one row for each enabled action in s. Supposing s is numbered i, α is numbered a (note that action numbering starts from 0 for each state, so

in G_{EBA}, actions are shared by multiple states, which we had mentioned when discussing the reasons for choosing this representation), and t is numbered j, to get $\delta(s, \alpha, t)$ for $s, t \in S, \alpha \in A[s]$, you look at the a-th row in the i-th row group, and get its j-th element.

For both graph-like structures, the transition matrices of most MDPs are sparsely populated, but their size grows quadratically with the number of states of the MDP. Due to this, most implementations employ sparse matrices, in which only non-zero elements are stored.



(a) Transition Matrix for $G_{\text{VBA}}(\mathcal{M})$ (b) Transition Matrix for $G_{\text{EBA}}(\mathcal{M})$

Figure 3.1: Transition Matrices for (3.1a : Vertex-Based Actions and 3.1b : Edge-Based Actions) (Source: Faber's Thesis [Fab23b])

Given a graph (V, E) and a vertex set $V' \subseteq V$, to implement $\text{Post}_{(V,E)}(V')$ for G_{VBA} (resp. G_{EBA}), we process each vertex $v' \in V'$ individually, and store all non-zero columns of the corresponding row (resp. row group) of T. Similarly, to implement $\text{Pre}_{(V,E)}(V')$ for G_{VBA} (resp. G_{EBA}), we store all the non-zero rows (resp. row groups) of the corresponding column.

For both Pre and Post, and both graph structures G_{VBA} and G_{EBA}, storing vertices and edges individually is not scalable. The computation required scales (at least) linearly with the size of V' since each vertex is processed individually. Large MDPs can consist of billions of states with thousands of SCCs, and simply storing them can become infeasible due to memory requirements. These problems lead us to the idea of symbolic storage.

3.2 Symbolic Storage

Symbolic storage provides an alternative to handling vertices and edges individually – you deal with entire sets of vertices or edges at once. So the interface is different, and more restrictive (you can't access a single vertex's edges directly). But in exchange, we get scalability. This is done through BDDs.

Assume that the number of vertices in the graph structure we are dealing with (whether

G_{VBA} or G_{EBA}) is n. Then, each vertex can be encoded into a $t = \lceil \log_2 n \rceil$ bit-vector (plus one bit in the case of G_{VBA} to indicate if it's in V_P or V_R, but we ignore that for simplicity), and a set $V' \subseteq V$ can be represented as a boolean function on t boolean variables $x_1, \ldots, x_t$:

$$f_{V'} : \{0,1\}^t \to \{0,1\} \qquad f_{V'}(\underbrace{x_1, \ldots, x_t}_{\text{encodes } s \in V}) = \begin{cases} 1 & \text{if } s \in V' \\ 0 & \text{otherwise} \end{cases}$$

$f_{V'}$ can be represented by a BDD, and that is how we store sets of vertices symbolically.

For G_{VBA}, a set of edges can be encoded into an ADD, representing the transition matrix. The ADD takes a set of Boolean arguments for the source vertex $s_i \in V$ (row) and another set of Boolean arguments for the destination vertex $s_j \in V$ (column) of an edge. If the ADD evaluates to a non-zero value, then an edge from s_i to s_j exists, and if $s_i \in V_R$, $s_j \in V_P$, then the ADD evaluates to a probability given by δ. For the computation of MECs, all that matters is whether the value is non-zero or not, hence this ADD can be converted to an (often smaller) BDD, representing the following boolean function:

$$t_{\text{VBA}}(\underbrace{x_1, \ldots, x_t}_{s \in V \text{ (Row)}}, \underbrace{x'_1, \ldots, x'_t}_{s' \in V \text{ (Column)}}) = \begin{cases} 1 & \text{if } (s, s') \in E \\ 0 & \text{otherwise} \end{cases}$$

Note that t_{VBA} can be represented by a BDD on $2t$ variables, and that it represents a binary relation E on V. Hence, we can compute $\text{Post}_{(V,E)}(V')$ (resp. $\text{Pre}_{(V,E)}(V')$) by taking the (resp. inverse) product of V' (or the ROBDD representing $f_{V'}$) with E (or the ROBDD representing t_{VBA}).

G_{EBA} is not just a directed graph, it is a directed labelled graph (with the labels being actions of the MDP). Therefore, we will need more boolean variables $y_1, \ldots, y_u$ to encode actions. Since we will never really deal with just actions alone, only with state-action pairs, we do not need to encode all $|A|$ actions, so it suffices to have $u = \lceil \log_2 \max_{s \in S} |A[s]| \rceil$. Note that this way of encoding also allows (and encourages) different states to share actions, so that the BDD size is smaller.

$$t_{\text{EBA}}(\underbrace{x_1, \ldots, x_t}_{s \in V \text{ (Row)}}, \underbrace{y_1, \ldots, y_u}_{\alpha \in A[s] \text{ (Action)}}, \underbrace{x'_1, \ldots, x'_t}_{s' \in V \text{ (Column)}}) = \begin{cases} 1 & \text{if } (s, \alpha, s') \in E \\ 0 & \text{otherwise} \end{cases}$$

To compute $\text{Post}_{(V,E)}(V')$ or $\text{Pre}_{(V,E)}(V')$, we convert the labelled directed edge relation to a directed edge relation E' and compute its relational or inverse relational product with V'. In BDD-land, this can be done using an exists operation, i.e.,

$$t'_{\text{EBA}}(x_1, \ldots, x_t, x'_1, \ldots, x'_t) = \exists \{y_1, \ldots, y_u\} \,.\, t_{\text{EBA}}(x_1, \ldots, y_1, \ldots, y_u, x_t, x'_1, \ldots, x'_t)$$

As we mentioned before, at the cost of a more complex interface and implementation, the major benefit of symbolic storage is its scalability: as $\text{Pre}_{(V,E)}(V')$ and $\text{Post}_{(V,E)}(V')$ operations are performed on BDDs, all elements of $V' \subseteq V$ are processed at once. Further, it has been shown that space efficient representations of structured probabilistic models can be constructed using MTBDDs [dAKN+00]. So, BDDs can often represent larger sets of vertices and edges than would be feasible with explicit storage.

However, BDDs are not without their drawbacks. Firstly, as shown by [BK08], no single data structure can compactly represent all boolean functions, so there are classes of MDPs for which the sizes of the BDDs are very large (as large as the memory required by explicit storage). Secondly, as we mentioned in chapter 2, the size of each BDD is also heavily influenced by the chosen boolean variable ordering. While finding the optimal ordering (and even improving a given ordering) for a given Boolean function is NP-hard (see [BW96]), various heuristics exist, depending on the problem being modelled. For transition functions, an interleaved ordering $x_1, x'_1, [y_1,] \ldots, x_t, x'_t, [y_t, y_{t+1}, \ldots, y_u]$ tends to yield good results [EFT91].

Chapter 4

Symbolic MEC Decomposition Algorithms

In this chapter, we aim to build the `INTERLEAVE` algorithm from the `NAIVE` symbolic MEC decomposition algorithm and the `SKELETON` symbolic SCC decomposition algorithm. We will begin by formalising what "symbolic algorithms" mean, then explain the `NAIVE` algorithm, followed by a detour to the `SKELETON` algorithm. Since the `SKELETON` algorithm is what inspired the creation of `INTERLEAVE`, we will see how it leads naturally to the `INTERLEAVE` algorithm. We will then formally prove its correctness and do a complexity analysis.

4.1 Symbolic Algorithms Formalism

4.1.1 Definition

We have seen that in symbolic storage, MDPs are not represented explicitly during their analysis. Instead, they are implicitly represented using data structures like BDDs. "Symbolic algorithms" is a theoretical model for algorithms that work on this implicit representation, that abstracts away the specifics of the representation and implementation [CDHS21]. 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 an adjacency list or an adjacency matrix, but can only access the input graph 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 with an edge to (resp. 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.

4.1.2 Symbolic Time and Space Complexity

Since symbolic operations are more expensive than non-symbolic operations, symbolic time is defined as the number of symbolic operations in a symbolic algorithm. As in previous literature [CHL+18, CDHS21], we focus only on the number of Pre/Post operations in a symbolic algorithm. This is due to two reasons. First, the basic set operations are computationally less expensive (as they typically deal with only one set of state and action variables) compared to the Pre/Post operations (since they have to deal with two sets of state variables - current and next). Second, in all previous algorithms and in `INTERLEAVE`, the number of basic set operations is asymptotically at most the number of Pre/Post operations.

Symbolic space is defined as the maximum number of sets (regardless of the size of the sets) stored by a symbolic algorithm at any point of time. This is because large sets may have a compact BDD representation (e.g., the set $\{0, \ldots, 2^n - 1\}$ has a BDD with a single node). Moreover, since the symbolic model is motivated by memory restrictions and compact storage, symbolic algorithms typically aim for sub-linear symbolic space (otherwise, why would you even use a symbolic algorithm?).

4.2 Omitted Functions

We omit the pseudocode for the following functions, since it follows in a straightforward manner from the definitions and the operations on BDDs presented in chapter 2.

- `ROut`(V', V, E) takes a vertex set $V' \subseteq V$ and a graph $(V, E) = G_{\text{EBA}}(T, \pi)$ for some sub-MDP (T, π), and returns $\text{ROut}_{(T,\pi)}(V')$ (see definition 2.12).
- `Attr`(X, V, E) takes a state-action-pair set $X \subseteq \text{sa}(T, \pi)$ and a graph $G_{\text{EBA}}(T, \pi) = (V, E)$ for some sub-MDP (T, π), and returns $\text{Attr}_{(T,\pi)}(X)$ (see definition 2.13).
- `Pre`(V', E) (resp. `Post`(V', E)) takes a vertex set $V' \subseteq V$ and a labelled (for MEC algorithms)/unlabelled (for SCC algorithms) edge relation E for a graph $G = (V, E)$, and returns $\text{Pre}_{(V,E)}(V')$ (resp. $\text{Post}_{(V,E)}(V')$) (see definition 2.17). In practice, one could abstract labels away and pass an unlabelled edge relation for MECs too.

4.3 `NAIVE` Symbolic MEC Decomposition

In this section, we will present a naive symbolic version of the explicit MEC decomposition algorithm given by deAlfaro in his PhD Thesis in 1998 [Alf98]. It follows closely from the definition of MECs (definition 2.9). Though it is simple, Faber's thesis [Fab23b] found that it had the fastest runtime on the Quantitative Verification Benchmark Set, and our experimental evaluation (chapter 5) found it to be the second-fastest after `INTERLEAVE`.

The input to the algorithm is the graph $G_{\text{EBA}}(\mathcal{M}) = (V, E)$ for an MDP $\mathcal{M}$, and its output is the set $\text{MECs}(\mathcal{M})$. The algorithm maintains a worklist χ of vertex sets, which are "candidates" for being MECs (note that the edges for an MEC can be constructed from its vertex set later, assuming unwanted state-action pairs have been deleted). It also maintains a set result that stores all the MECs found so far. result is initially empty, and χ is initialised with all the SCCs of (V, E).

In each iteration, the algorithm picks a candidate V' from χ. It is either:

1. identified as a maximal end component and added to result, or
2. removed because the induced sub-MDP doesn't contain an edge, or
3. it contains vertices with outgoing actions. In this case, the ROut of the vertex set gives these state-action pairs, and its Attr gives the states and state-action pairs which we know can't be part of any MEC. We remove the state-action pairs from E and

remove the states from V'. After this step, the induced subgraph may not be strongly connected anymore. Thus, the SCCs of this subgraph are determined and added to χ as new candidates for maximal end components.

Algorithm 1 shows the pseudocode for the NAIVE algorithm. It assumes a symbolic SCC decomposition algorithm called AllSCCs that takes an (unlabelled) directed graph and returns the list of SCCs of the graph.

Algorithm 1 MEC-Decomp-Naive($G = (V, E)$)

Input: Graph $G = (V, E) = G_{\text{EBA}}(\mathcal{M})$ for some MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$.

Output: The set of edge-based graph representations of MECs($\mathcal{M}$), i.e., $\{G_{\text{EBA}}(T, \pi) \mid (T, \pi) \in \text{MECs}(\mathcal{M})\}$.

```
1:  result ← ∅
2:  χ ← AllSCCs(V, ∃ α ∈ A . E)                ▷ Abstract actions for the AllSCCs call.
3:  while χ ≠ ∅ do
4:      Remove some V' ∈ χ from χ
5:      rout ← ROut(V', V, E)
6:      (U_1, X_1) ← Attr(rout, V, E)
7:      if rout = ∅ then
8:          if Post(V', E) ∩ V' ≠ ∅ then        ▷ G[V'] has at least one edge.
9:              result ← result ∪ {(V', E ∩ (V' × A × V'))}
10:     else
11:         E ← E \ (X_1 × V)                  ▷ Remove the state-action pairs in X_1 from E.
12:         V' ← V' \ U_1                      ▷ Remove the states in U_1 from V'.
13:         χ ← χ ∪ AllSCCs(V', (∃ α ∈ A . E) ∩ (V' × V'))   ▷ Abstract actions and restrict
                                                edges for the AllSCCs call.
14: return result
```

Figure 4.1 shows the execution of the NAIVE algorithm on an example MDP. It has been

borrowed from Faber's thesis [Fab23b].

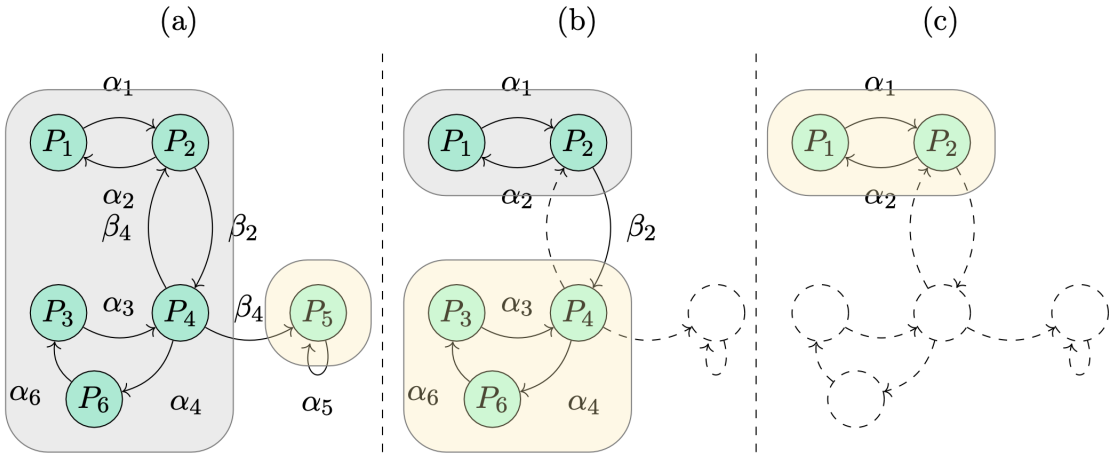


Figure 4.1: Example Execution of NAIVE (Source : Faber's Thesis [Fab23b])

The figures show the following things:

(a) The graph is decomposed into SCCs. The SCC $\{P_5\}$ with the action α_5 is identified as an MEC (yellow) since $\mathrm{ROut}(P_5) = \emptyset$. The other SCC C is not an MEC (gray) due to the outgoing action β_4.

(b) After the removal of β_4, another SCC decomposition on C is performed. The SCC $\{P_3, P_4, P_6\}$ has no outgoing actions and is thus identified as an MEC with its actions $\{\alpha_3, \alpha_4, \alpha_6\}$. The other SCC $\{P_1, P_2\}$ is not an MEC due to the outgoing action β_2.

(c) After the removal of β_2, the final SCC decomposition on $\{P_1, P_2\}$ is performed. The resulting SCC has no outgoing edges and is identified as an MEC with its actions $\{\alpha_1, \alpha_2\}$.

We will not prove the correctness of the algorithm here, though the correctness proof for INTERLEAVE has some of the required pieces. The analysis from [CHL+18] shows that this algorithm requires $O(n^2)$ symbolic operations, and $O(\log n)$ symbolic space, assuming an $O(n)$ symbolic operations (and $O(\log n)$ symbolic space) symbolic SCC decomposition algorithm. The specific algorithm they mention is the SKELETON algorithm from [GPP03]. We will take a look at that algorithm next, and then present the INTERLEAVE algorithm.

4.4 SKELETON Symbolic SCC Decomposition

The SKELETON algorithm came out in 2003, and was the only $O(n)$ symbolic operations SCC decomposition algorithm until 2023, when the CHAIN algorithm [LSS+23] was published. It is also optimal, as proved in [CDHL17]. The CHAIN algorithm essentially does the same thing as SKELETON, but uses less space. In this thesis, we will focus on SKELETON since it is easier to analyse.

A primitive version of the SKELETON algorithm would look like the following. The input is a graph (V, E) with $V \neq \emptyset$.

1. Pick a vertex $v \in V$ arbitrarily.
2. Compute the set of vertices reachable from v ($F_v = \text{Fwd}(v, E)$) by repeated Post operations on $\{v\}$.
3. Compute the SCC of v (SCC_v) by repeated Pre operations on $\{v\}$ and intersecting with F_v.
4. Output SCC_v.
5. Recurse on the subgraphs induced by the two remaining partitions of V (if they are non-empty) : $(F_v \setminus \text{SCC}_v)$ and $V \setminus F_v$.

What drives the efficiency of this algorithm is that a vertex $v \in V$ is not picked *arbitrarily*, a spine set $< S, v >$ is given as input and v is chosen. Initially, $< S, v >$ is empty and so v is chosen arbitrarily, but everytime $\text{Fwd}(v, E)$ is computed, a new spine set is computed for recursing on the $(F_v \setminus \text{SCC}_v)$ partition. So, the actual SKELETON algorithm looks like this:

Algorithm 2 SCC-Decomp-Skeleton$(V, E, S, \{v\})$

Input: A directed graph $G = (V, E)$ and a spine set $< S, v >$ in G.
Output: The set of SCCs of G.

1: **if** $V = \emptyset$ **then return**

2: **if** $S = \emptyset$ **then** $\{v\} \leftarrow \text{Pick}(V)$

3: ▷ *Compute the forward set of v and a new spine set $< S', v' >$.*

4: $F_v, S', \{v'\} \leftarrow$ Skeleton-Fwd$(V, E, \{v\})$

5: $\text{SCC}_v \leftarrow \{v\}$ ▷ *Compute and output the SCC of v.*

6: **while** $((\text{Pre}(\text{SCC}_v) \cap F_v) \setminus \text{SCC}_v) \neq \emptyset$ **do**

7: $\quad \text{SCC}_v \leftarrow \text{SCC}_v \cup (\text{Pre}(\text{SCC}_v) \cap F_v)$

8: **Output** SCC_v as an SCC.

9: ▷ *Recurse on the partition $F_v \setminus \text{SCC}_v$. Use the new spine set.*

10: $V' \leftarrow F_v \setminus \text{SCC}_v \quad , E' \leftarrow E \cap (V' \times V')$

11: $S' \leftarrow S' \setminus \text{SCC}_v \quad , \{v'\} \leftarrow \{v'\} \setminus \text{SCC}_v$

12: SCC-Decomp-Skeleton$(V', E', S', \{v'\})$.

13: ▷ *Recurse on the partition $V \setminus F_v$. Use the old spine set.*

14: $V' \leftarrow V \setminus F_v \quad , E' \leftarrow E \cap (V' \times V')$

15: $S' \leftarrow S \setminus \text{SCC}_v \quad , \{v'\} \leftarrow (\text{Pre}(S \cap \text{SCC}_v)) \cap (S \setminus \text{SCC}_v)$

16: SCC-Decomp-Skeleton$(V', E', S', \{v'\})$.

The exact definition and calculation of a spine set are not important to our algorithm, so we will not go into them. But there are a couple of important properties:

1. A spine set $< S, v >$ implicitly represents a sequence $v_0, v_1, \ldots, v_n$ of vertices, with $v = v_n$. By implicitly, we mean that having access only to the *set* (not the *sequence*) S and the last vertex $v = v_n$, we can get the second last vertex v_{n-1} and $S' = \{v_0, \ldots, v_{n-1}\}$ using only basic set operations, and $< S', v_{n-1} >$ is also a spine set.

2. Given a spine set $< S, v >$, the intersection of SCC(v) and S is a contiguous chunk of the sequence, i.e., $\{v_j, \ldots, v_n\}$ for some $j \in \{0, \ldots, n\}$.

The point of computing a spine set while calculating the forward set and then passing it to the first call is, that on the chain of second calls after that, the SCCs of vertices of the chain set are taken out in reverse order.

The reason for this is to ensure efficiency of the algorithm, which we will not go into. But how this is done is important for us. The two properties ensure that we know what to pass as the spine set for the second recursive call (we remove the contiguous chunk that forms SCC(v), then take the last vertex before that in the spine set).

We initially sought to translate a similar algorithm for MEC decomposition, but as we will now see, a few complications arise.

4.5 INTERLEAVE Symbolic MEC Decomposition

4.5.1 Informal Development

We saw a primitive version of the SKELETON SCC decomposition algorithm. The key idea behind the INTERLEAVE algorithm is that NAIVE performs an SCC decomposition, then performs some MEC computation, performs more SCC decompositions and so on. Is it possible to interleave the MEC computation *while* performing the SCC decomposition instead of waiting for an entire SCC decomposition to finish and then performing it? Along those lines, a similar primitive version of the INTERLEAVE MEC decomposition algorithm would look like the following (the differences have been coloured blue). The input is a graph $G = (V, E) = G_{\text{EBA}}(T, \pi)$ for some sub-MDP (T, π) of some MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$.

1. Pick a vertex $v \in V$ arbitrarily.
2. Compute the set of vertices reachable from v ($F_v = \text{Fwd}(v, E)$) by repeated Post operations on $\{v\}$.
3. Compute the SCC of v (SCC_v) by repeated Pre operations on $\{v\}$ and intersecting with F_v.
4. If rout = $\texttt{ROut}(\text{SCC}_v, V, E) = \emptyset$ then output the sub-MDP induced by SCC_v.
5. Otherwise, compute $\texttt{Attr}(\text{rout}, V, E)$, set $V_1 = (\text{SCC}_v \setminus \text{states})$ and remove the state-action pairs from E.
6. Compute $\texttt{Attr}(\texttt{ROut}(V \setminus F_v, V, E), V, E)$, set $V_3 = ((V \setminus F_v) \setminus \text{states})$ and remove the state-action pairs from E.
7. Recurse on the sub-MDPs induced by the following three vertex-sets (if they are non-empty) : V_1, $(F_v \setminus \text{SCC}_v)$ and V_3.

Note that the ROut for the second vertex set $(F_v \setminus \text{SCC}_v)$ is actually always $\emptyset$ (this is proven in our correctness proof), so we can omit that. Now, this algorithm is correct, as the primitive

SKELETON algorithm was. But we would like to get efficiency. So, we can try to use the spine set idea in this algorithm. However, there are two problems.

1. In this algorithm, we cannot output the SCC of v directly, since it may contain some state-action pairs that can go outside the SCC. In that case, they will be identified using ROut (and more states and state-action pairs will be added using Attr), and then we need to recurse on the remaining subgraph again. We do not know what to pass as the spine set for this recursive call.

2. For the third recursive call (the $V \setminus F_v$ partition), in the SKELETON algorithm, we could chop off a contiguous chunk of the spine set from the end and pass the remaining spine. However, here, since we are removing ROut (and its Attr), we may remove random nodes and edges on the spine, and it will no longer implicitly represent a sequence. In other words, the spine may not remain a spine, and thus cannot be passed as a spine set.

It is only the second recursive call (on $F_v \setminus \mathrm{SCC}_v$) from which we don't have to remove any vertices or edges. So we can pass it the same next vertex to start from, however the rest of the spine isn't useful since it would have been used in the chain of (V F_v) recursive calls after that (which we do not know how to make). That is essentially what INTERLEAVE does. For the second recursive call, it picks any vertex at maximum distance from v, and passes that as the vertex to start from (for SKELETON, v' in the returned spine set $< S', v' >$ is always a vertex at maximum distance from v).

The benefit we get from doing that is, suppose v' is a vertex at maximum distance from v, and suppose $v_0 = v, \ldots, v_n = v'$ is a shortest path from v to v'. Note that computing v's SCC requires $2n$ symbolic operations, so we can charge $O(1)$ operations to each v_i. Then, in the recursive call on $F_v \setminus \mathrm{SCC}_v$, we will compute $F_{v'}$, and we are guaranteed that only those v_is will be in $F_{v'}$ which are also in $\mathrm{SCC}_{v'}$. So we will not charge the same vertices again, except when they are in the SCC we compute. This is not strong enough to change the complexity of the algorithm (as it only gives a guarantee on the immediate next call), as we will see. But it does seem to lead to empirical improvements in running time.

4.5.2 Pseudocode

Algorithm 3 shows the pseudocode for the INTERLEAVE algorithm. It is the same as the primitive version mentioned earlier, with the difference that the recursive call on $F_v \setminus \mathrm{SCC}_v$ now gets a specified vertex to start from, some v' that is at maximum distance from v. SCC-Fwd-NewStart$(\{v\}, E)$ computes the SCC of v, the forward set of v, and such a v'.

Algorithm 3 `MEC-Decomp-Interleave`$(V, E, \{v\} = \emptyset)$

Input: $(V, E) = G_{\text{EBA}}(T, \pi)$ for some sub-MDP (T, π) of some MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$ and (optionally), a start vertex $v \in V$. For the initial call, $\{v\} = \emptyset$.

Output: The set of graph representations of $\text{MECs}_{\mathcal{M}}(T)$, i.e., $\{G_{\text{EBA}}(T', \pi') \mid (T', \pi') \in \text{MECs}_{\mathcal{M}}(T)\}$.

1: **if** $\{v\} = \emptyset$ **then** $\{v\} \leftarrow \text{Pick}(V)$

2: $C_v, F_v, \{v'\} \leftarrow$ `SCC-Fwd-NewStart`$(\{v\}, E)$

3: $(U_1, X_1) \leftarrow$ `Attr`$($`ROut`$(C_v, V, E), V, E)$

4: **if** $X_1 = \emptyset$ **then**

5: ▷ *Every vertex has an outgoing edge from definitions of sub-MDP and G_{EBA}. So $E \cap (C_s \times A \times C_s) \neq \emptyset$.* **Output** $(C_v, E \cap (C_v \times A \times C_v))$ as an MEC

6: **else**

7: $E_1 \leftarrow E \setminus (X_1 \times V)$ ▷ *Remove the state-action pairs in X_1 from E.*

8: $V_1 \leftarrow C_v \setminus U_1$ ▷ *Remove the states in U_1 from C_v.*

9: **if** $V_1 \neq \emptyset$ **then** `MEC-Decomp-Interleave`$(V_1, E_1 \cap (V_1 \times A \times V_1), \emptyset)$

10: $V_2 \leftarrow (F_v \setminus C_v)$

11: **if** $V_2 \neq \emptyset$ **then** `MEC-Decomp-Interleave`$(V_2, E \cap (V_2 \times A \times V_2), \{v'\})$

12: $(U_3, X_3) \leftarrow$ `Attr`$($`ROut`$(V \setminus F_v, V, E), V, E)$

13: $E_3 \leftarrow E \setminus (X_3 \times V)$ ▷ *Remove the state-action pairs in X_3 from E.*

14: $V_3 \leftarrow (V \setminus F_v) \setminus U_3$ ▷ *Remove the states in U_3 from $V \setminus F_v$.*

15: **if** $V_3 \neq \emptyset$ **then** `MEC-Decomp-Interleave`$(V_3, E_3 \cap (V_3 \times A \times V_3), \emptyset)$

Algorithm 4 SCC-Fwd-NewStart$(\{s\}, E)$

Input: A singleton vertex set $\{s\}$ ($s \in V$) and a labelled edge relation E for a graph $G = (V, E)$.

Output: The SCC of s in G, the set of vertices reachable from s, and a vertex at maximum distance from s.

1: $F_s, \{s'\} \leftarrow$ Fwd-NewVertex$(\{s\}, E)$

2: $C_s \leftarrow \{s\}$.

3: **while** $(\mathtt{Pre}(C_s, E) \cap F_s) \not\subseteq C_s$ **do**

4: $\quad C_s \leftarrow C_s \cup (\mathtt{Pre}(C_s, E) \cap F_s)$

5: **return** $C_s, F_s, \{s'\}$

Algorithm 5 Fwd-NewVertex$(\{v\}, E)$

Input: A vertex set $v \in V$ and a labelled edge relation E for a graph $G = (V, E)$.

Output: The set of vertices in V reachable from v and a vertex at maximum distance from v (in the graph G).

1: $F \leftarrow \emptyset$

2: $L \leftarrow \{v\}$

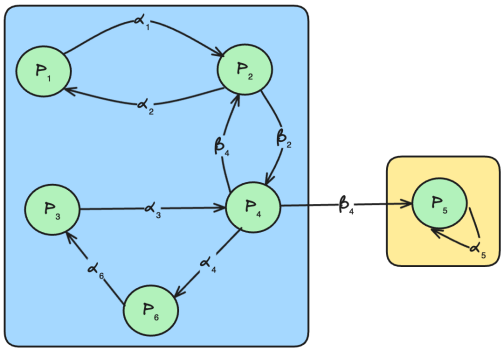
3: **while** $L \neq \emptyset$ **do**

4: $\quad F \leftarrow F \cup L$

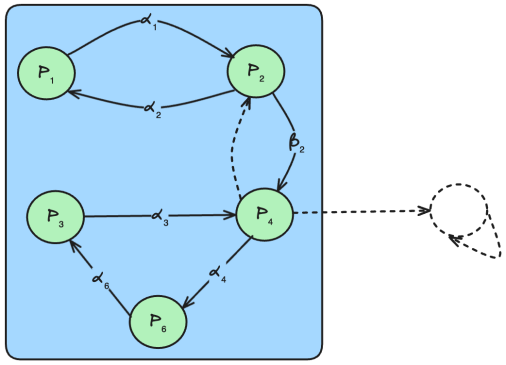
5: $\quad L \leftarrow \mathtt{Post}(L, E) \setminus F$

6: **return** F

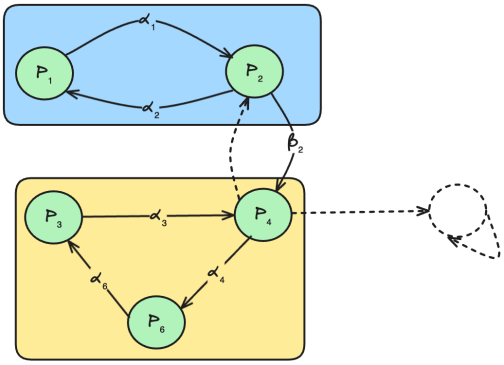
Figure 4.2 shows an example execution of the INTERLEAVE algorithm.



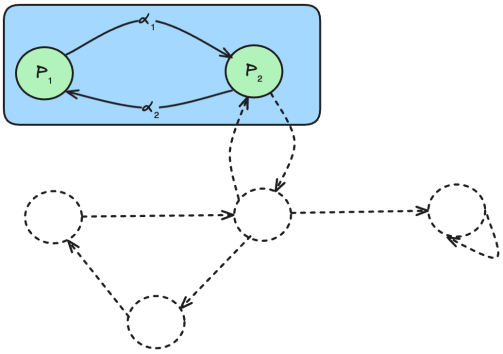
(a) Call 1 : $v = P_5$, SCC_v in yellow and $V \setminus F_v$ in blue



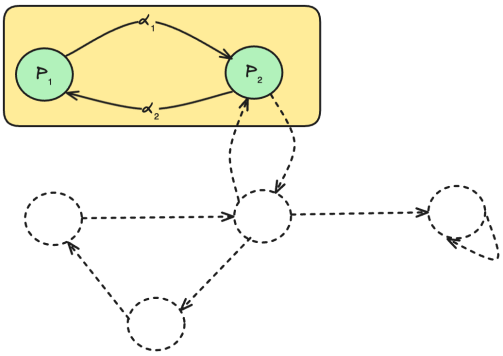
(b) Call 2 : Initial graph



(c) Call 2 : $v = P_3$, SCC_v in yellow and $V \setminus F_v$ in blue



(d) Call 3 : Initial graph



(e) Call 3 : $v = P_1$, SCC_v in yellow

Figure 4.2: Execution of INTERLEAVE on an Example MDP

The figures show the following things:

(a) $v = P_5$ is picked as the vertex to start from. $\mathrm{SCC}_v = \{P_5\}$ is computed (yellow), $F_v \setminus \mathrm{SCC}_v = \emptyset$ and $V \setminus \mathrm{SCC}_v = \{P_1, P_2, P_3, P_4, P_6\}$ (blue). Since $\texttt{ROut}(\mathrm{SCC}_v, V, E) = \emptyset$, it is output as an MEC (with edges $\{(P_5, \alpha_5, P_5)\}$).

(b) $\texttt{ROut}(V \setminus \mathrm{SCC}_v, V, E) = \{(P_4, \beta_4)\}$ (and its Attr is the same) is removed from the graph

before the recursive call is made on $V \setminus \mathrm{SCC}_v$. The subgraph passed to the recursive call is highlighted in blue.

(c) $v = P_3$ is picked as the vertex to start from. $\mathrm{SCC}_v = \{P_3, P_4, P_6\}$ is computed (yellow), $F_v \setminus \mathrm{SCC}_v = \emptyset$ and $V \setminus \mathrm{SCC}_v = \{P_1, P_2\}$ (blue). Since $\texttt{ROut}(\mathrm{SCC}_v, V, E) = \emptyset$, it is output as an MEC (with edges $\{(P_3, \alpha_3, P_4), (P_4, \alpha_4, P_6), (P_6, \alpha_6, P_3)\}$).

(d) $\texttt{ROut}(V \setminus \mathrm{SCC}_v, V, E) = \{(P_2, \beta_2)\}$ (and its Attr is the same) is removed from the graph before the recursive call is made on $V \setminus \mathrm{SCC}_v$. The subgraph passed to the recursive call is highlighted in blue.

(e) $v = P_1$ is picked as the vertex to start from. $\mathrm{SCC}_v = \{P_1, P_2\}$ is computed (yellow), $F_v \setminus \mathrm{SCC}_v = \emptyset$ and $V \setminus \mathrm{SCC}_v = \emptyset$. Since $\texttt{ROut}(\mathrm{SCC}_v, V, E) = \emptyset$, it is output as an MEC (with edges $\{(P_1, \alpha_1, P_2), (P_2, \alpha_2, P_1)\}$).

If we ignore figures 4.2b and 4.2d which are just showing the initial arguments, the steps taken by INTERLEAVE seem to be similar to the steps taken by NAIVE (see figure 4.1). However, the first step of NAIVE is actually two steps (assuming its SCC decomposition call starts from the same $v = P_5$), because it needs to figure out the SCC decomposition. **One key difference is that when the SCC decomposition algorithm for NAIVE (let's assume it is SKELETON) looks at the blue subgraph, it doesn't know that β_4 is removed.** We don't need to perform the SCC decomposition of the blue subgraph with β_4 included, so NAIVE performs one additional step that is redundant.

4.5.3 Complexity Analysis

Number of Symbolic Operations: As mentioned before, we will be focusing on the number of Pre/Post operations. Suppose the input graph MDP is $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$, $(V, E) = G_{\text{EBA}}(\mathcal{M})$, and $\texttt{MEC-Decomp-Interleave}(V, E, \emptyset)$ is called. Let $|S| = p$ and $|\{(s, \alpha) \in S \times A \mid \alpha \in A[s]\}| = q$. Then, there are two kinds of Pre/Post operations:

1. Those in the ROut and Attr computations on lines 3 and 12. Each such operation discovers at least one new state or state-action pair that is thereafter removed from the graph and never seen again. Thus, over the entire algorithm, the cost of these is $O(p + q)$ symbolic operations.

2. Those in the SCC-Fwd-NewStart computation. When given an input graph with n vertices, these can be at most $2n$ operations. Now, if you look at the tree of recursive calls, the cost incurred at the top level is $c_1 = 2p$. At the second level, the cost incurred is $c_2 = 2(|V_1| + |V_2| + |V_3|)$. Since $V_1 \subseteq \mathrm{SCC}_v$, $V_2 = F_v \setminus \mathrm{SCC}_v$ and $V_3 \subseteq V \setminus F_v$, we have $c_2 \leq 2(|\mathrm{SCC}_v| + |F_v \setminus \mathrm{SCC}_v| + |V \setminus F_v|) = 2|V| = 2p$. So the cost for each level is at most $2p$. Now, how many levels can there be? In every recursive call, the number of states and/or the number of state-action pairs decreases by at least one. Therefore, there can be at most $(p+q)$ levels. This means that the number of symbolic operations done by SCC-Fwd-NewStart over the entire algorithm is $O(p(p + q))$.

The total cost for the algorithm then becomes $O(p(p + q))$ symbolic operations. As we had mentioned earlier, to maintain consistency with previous work, we will mention the complexity assuming n is the number of vertices and m is the number of edges in the G_{VBA}

representation for $\mathcal{M}$. Note that this means $n = p + q$. **Thus, the symbolic operations complexity of this algorithm is $O(n^2)$.**

Symbolic Space Complexity: This is the maximal number of sets that the algorithm stores simultaneously at any point of time. Note that each recursive call, excluding the space used by its children, only stores $O(1)$ number of sets simultaneously. Thus, symbolic space is just the maximum depth of recursive calls reached throughout the algorithm.

To achieve efficiency here, we will use the fact that our algorithm is tail-recursive. After the first two recursive calls have returned, the memory stored by the current call can be deleted before making the third recursive call. So we just need to look at the maximum depth reached through the first two recursive calls. Now, the three recursive calls should be made in increasing order of the sizes of the vertex sets $|V_1|, |V_2|, |V_3|$. The sizes of vertex sets for both of the first two recursive calls must be $< 2p/3$. On both branches, the vertex set sizes decreases by at least 2/3. Therefore, the maximum depth that can be reached is $\log_{\frac{3}{2}} p$. Again, since $p = O(n)$ where n is the number of vertices in the G_{VBA} representation, **the symbolic space complexity of this algorithm is $O(\log n)$.**

4.5.4 Correctness Proof

To prove the algorithm correct, we will show (1) Soundness (whenever it outputs an MEC, it is actually an MEC) and (2) Completeness (it outputs all the MECs of the MDP it is called on). Before we show these, we will need to prove that every time we remove a state or state-action pair, we remove one that's not in any MEC. That is what the following series of lemmas is for.

Lemma 4.1. *Let (T, π) be an MEC-closed sub-MDP of some MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$ and $X \subseteq \text{sa}(T, \pi)$ be a state-action pair set such that for all $(T', \pi') \in \text{MECs}(\mathcal{M})$, $\text{sa}(T', \pi') \cap X = \emptyset$. Suppose $\text{Attr}_{(T,\pi)}(X) = (S', X')$. Then, for all $(T', \pi') \in \text{MECs}(\mathcal{M})$, $T' \cap S' = \emptyset$ and $\text{sa}(T', \pi') \cap X' = \emptyset$.*

Proof. Let $(T', \pi') \in \text{MECs}(\mathcal{M})$. From definition 2.13, we have $(S', X') = \text{Attr}_{(T,\pi)}(X) = (\bigcup_{i \in \mathbb{N}} S_i, \bigcup_{i \in \mathbb{N}} X_i)$. We will show by induction on i that for all $i \in \mathbb{N}$, $T' \cap S_i = \emptyset$ and $\text{sa}(T', \pi') \cap X_i = \emptyset$.

Base Case ($i = 0$): By definition 2.13, $S_0 = \emptyset$ and $X_0 = X$. The first implies $T' \cap S_0 = \emptyset$. The second, along with our assumption, implies $\text{sa}(T', \pi') \cap X_0 = \emptyset$.

Induction Step: We assume that $T' \cap S_i = \emptyset$ and $\text{sa}(T', \pi') \cap X_i = \emptyset$.

- Assume, for the sake of contradiction, that $T' \cap S_{i+1} \neq \emptyset$. Pick $s \in T' \cap S_{i+1}$. Since $T' \cap S_i = \emptyset$ from the induction hypothesis, we must have $s \in T' \cap (S_{i+1} \setminus S_i)$. Note that we have $(T', \pi') = \text{MEC}_{\mathcal{M}}(s)$. We have:

$$s \in T \qquad \because s \in (S_{i+1} \setminus S_i) \text{ and definition 2.13} \tag{4.1}$$

$$\forall\, \alpha \in \pi(s)\,.\,(s,\alpha) \in X_i \qquad \because s \in (S_{i+1} \setminus S_i) \text{ and definition 2.13} \tag{4.2}$$

$$(T',\pi') \subseteq (T,\pi) \qquad \because \text{ equation 4.1, } (T',\pi') = \mathrm{MEC}_{\mathcal{M}}(s) \text{ and } (T,\pi) \text{ is MEC-closed} \tag{4.3}$$

Pick $\alpha \in \pi'(s)$ (exists from definition 2.3 since (T',π') is a sub-MDP). Then, $\alpha \in \pi(s)$ from equation 4.3. This implies $(s,\alpha) \in X_i$ ($\because$ equation 4.2). Note that since $s \in T'$ and $\alpha \in \pi'(s)$, we also have $(s,\alpha) \in \mathrm{sa}(T',\pi')$. But from the induction hypothesis, we have that $\mathrm{sa}(T',\pi') \cap X_i = \emptyset$, which is a contradiction. Therefore, we must have $T' \cap S_{i+1} = \emptyset$.

- Assume, for the sake of contradiction, that $\mathrm{sa}(T',\pi') \cap X_{i+1} \neq \emptyset$. Pick $(s,\alpha) \in \mathrm{sa}(T',\pi') \cap X_{i+1}$. Since $\mathrm{sa}(T',\pi') \cap X_i = \emptyset$ from the induction hypothesis, we must have $(s,\alpha) \in \mathrm{sa}(T',\pi') \cap (X_{i+1} \setminus X_i)$. Since $(s,\alpha) \in (X_{i+1} \setminus X_i)$, definition 2.13 tells us that $\exists\, s' \in S_i\,.\,(\delta(s,\alpha,s') > 0)$. Since (T',π') is a sub-MDP, $s \in T'$ and $\alpha \in \pi'(s)$, this implies $s' \in T'$. But from the induction hypothesis, $T' \cap S_i = \emptyset$, which is a contradiction. Therefore, we must have $\mathrm{sa}(T',\pi') \cap X_{i+1} = \emptyset$.

This completes the induction step, and the proof. □

Lemma 4.2. *Let (T,π) be an MEC-closed sub-MDP of some MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$ and $(V,E) = G_{\text{EBA}}(T,\pi)$. Consider the execution of* `MEC-Decomp-Interleave`*$(V,E,\{v\})$ where $\{v\} = \emptyset$ or $v \in V$. Then, for any $(T',\pi') \in \mathrm{MECs}(\mathcal{M})$, $T' \cap U_1 = \emptyset$ and $\mathrm{sa}(T',\pi') \cap X_1 = \emptyset$.*

Proof. We know, from the algorithm, that $(U_1, X_1) = \mathrm{Attr}_{(T,\pi)}(\mathrm{ROut}_{(T,\pi)}(C_s))$, where C_s is the SCC in (V,E) of some $s \in T$ (which is v if the third argument is not $\emptyset$). First, we will show that for any $(T',\pi') \in \mathrm{MECs}(\mathcal{M})$, $\mathrm{sa}(T',\pi') \cap \mathrm{ROut}_{(T,\pi)}(C_s) = \emptyset$. Then, applying lemma 4.1 proves the final result (note that definition 2.12 guarantees the other precondition of lemma 4.1, i.e., $\mathrm{ROut}_{(T,\pi)}(C_s) \subseteq \mathrm{sa}(T,\pi)$). So, assume, for the sake of contradiction, that there is some $(T',\pi') \in \mathrm{MECs}(\mathcal{M})$ such that $\mathrm{sa}(T',\pi') \cap \mathrm{ROut}_{(T,\pi)}(C_s) \neq \emptyset$.

Pick $(s',\alpha) \in \mathrm{sa}(T',\pi') \cap \mathrm{ROut}_{(T,\pi)}(C_s)$. Since $(s',\alpha) \in \mathrm{ROut}_{(T,\pi)}(C_s)$, from definition 2.12, we have $t \in (T \setminus C_s)$ such that $\delta(s',\alpha,t) > 0$. As (T',π') is a sub-MDP, $s' \in T'$ and $\alpha \in \pi'(s')$, this implies $t \in T'$. Now, (T',π') is an end-component and $s', t \in T'$. So, we have that s' and t can reach each other in (T',π'). Since $s' \in T$ and (T,π) is MEC-closed, we have $(T',\pi') = \mathrm{MEC}_{\mathcal{M}}(s') \subseteq (T,\pi)$. Thus s' and t can also reach each other in (T,π). This implies that $t \in \mathrm{SCC}_{(V,E)}(s') = C_s$, which is a contradiction to the fact that $t \in T \setminus C_s$. Therefore, we must have that for all $(T',\pi') \in \mathrm{MECs}(\mathcal{M})$, $\mathrm{sa}(T',\pi') \cap \mathrm{ROut}_{(T,\pi)}(C_s) = \emptyset$. As argued before, applying lemma 4.1 now proves the final result. □

Lemma 4.3 is the reason why we don't need to compute ROut before the second recursive call.

Lemma 4.3. *Let (T, π) be an MEC-closed sub-MDP of some MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$ and $(V, E) = G_{\text{EBA}}(T, \pi)$. Consider the execution of* `MEC-Decomp-Interleave`$(V, E\{v\})$ *where $\{v\} = \emptyset$ or $v \in V$. If $\{v\} = \emptyset$, let $s \in V$ be the vertex picked on line 1. Otherwise, let $s = v$. Then,* $\text{ROut}_{(T,\pi)}(F_s \setminus C_s) = \emptyset$.

Proof. Assume, for the sake of contradiction, that there is some $(s_1, \alpha) \in \text{ROut}_{(T,\pi)}(F_s \setminus C_s)$. From the definition of ROut, there is some $s_2 \in (T \setminus (F_s \setminus C_s)) = ((T \setminus F_s) \uplus C_s)$ such that $\delta(s_1, \alpha, s_2) > 0$. First note that $s_2 \in F_s$ since $s_1 \in F_s$ and there is an edge (labelled α) from s_1 to s_2. So we must have $s_2 \in C_s$. But then, there is a path from s_2 to s, and thus from s_1 to s. Combined with $s_1 \in F_s$, this means $s_1 \in C_s$, contradicting the fact that $(s_1, \alpha) \in \text{ROut}_{(T,\pi)}(F_s \setminus C_s)$. Therefore, we must have $\text{ROut}_{(T,\pi)}(F_s \setminus C_s) = \emptyset$. □

Lemma 4.4. *Let (T, π) be an MEC-closed sub-MDP of some MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$ and $(V, E) = G_{\text{EBA}}(T, \pi)$. Consider the execution of* `MEC-Decomp-Interleave`$(V, E\{v\})$ *where $\{v\} = \emptyset$ or $v \in V$. Then, for any $(T', \pi') \in \text{MECs}(\mathcal{M})$, $T' \cap U_3 = \emptyset$ and $\text{sa}(T', \pi') \cap X_3 = \emptyset$.*

Proof. We know, from the algorithm, that $(U_3, X_3) = \text{Attr}_{(T,\pi)}(\text{ROut}_{(T,\pi)}(T \setminus F_s))$, where F_s is the forward reachable set in (V, E) of some $s \in T$ (which is v if the third argument is not $\emptyset$). First, we will show that for any $(T', \pi') \in \text{MECs}(\mathcal{M})$, $\text{sa}(T', \pi') \cap \text{ROut}_{(T,\pi)}(T \setminus F_s) = \emptyset$. Then, applying lemma 4.1 proves the final result (note that definition 2.12 guarantees the other precondition of lemma 4.1, i.e., $\text{ROut}_{(T,\pi)}(T \setminus F_s) \subseteq \text{sa}(T, \pi)$).

So, assume, for the sake of contradiction, that there is some $(T', \pi') \in \text{MECs}(\mathcal{M})$ such that $\text{sa}(T', \pi') \cap \text{ROut}_{(T,\pi)}(T \setminus F_s) \neq \emptyset$. Pick (s_1, α) in the intersection. From the definition of ROut, we have $s_2 \in (T \setminus (T \setminus F_s)) = F_s$ such that $\delta(s_1, \alpha, s_2) > 0$. As (T', π') is a sub-MDP, $s_1 \in T'$ and $\alpha \in \pi'(s_1)$, this implies $s_2 \in T'$. Now, (T', π') is an end-component and $s_1, s_2 \in T'$. So, we have that s_2 can reach s_1 in (T', π'). Since $s_1 \in T$ and (T, π) is MEC-closed, we have $(T', \pi') = \text{MEC}_{\mathcal{M}}(s_1) \subseteq (T, \pi)$. Thus s_2 can also reach s_1 in (T, π). Since $s_2 \in F_s$, this implies that $s_1 \in F_s$, which is a contradiction to the fact that $(s_1, \alpha) \in \text{ROut}_{(T,\pi)}(T \setminus F_s)$. Therefore, we must have that for all $(T', \pi') \in \text{MECs}(\mathcal{M})$, $\text{sa}(T', \pi') \cap \text{ROut}_{(T,\pi)}(T \setminus F_s) = \emptyset$. As argued before, applying lemma 4.1 now proves the final result. □

Now that we have shown we never remove a "wrong" state or state-action pair, we are ready to prove soundness – if we say something is an MEC, then it is an MEC.

Theorem 4.5 (Soundness)**.** *Let (T, π) be an MEC-closed sub-MDP of some MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$ and $(V, E) = G_{\text{EBA}}(T, \pi)$. Consider the execution of* `MEC-Decomp-Interleave`$(V, E\{v\})$ *where $\{v\} = \emptyset$ or $v \in V$. If $\{v\} = \emptyset$, let $s \in V$ be the vertex picked on line 1. Otherwise, let $s = v$. If $X_1 = \emptyset$, then $(C_s, E \cap (C_s \times A \times C_s)) = G_{\text{EBA}}(\text{MEC}_{\mathcal{M}}(s))$.*

Proof. From the algorithm, we know $(U_1, X_1) = \text{Attr}_{(T,\pi)}(\text{ROut}_{(T,\pi)}(C_s))$. From the definition of Attr, if $X_1 = \emptyset$, then we must have $\text{ROut}_{(T,\pi)}(C_s) = \emptyset$. Define $\pi' = \pi \mid_{C_s}$, i.e., the function

π restricted to the domain C_s.

Claim 1: (C_s, π') is a sub-MDP of $\mathcal{M}$. **Proof:** This is true because $C_s \neq \emptyset$ (as $s \in C_s$) and for each $s' \in C_s \subseteq T$, $\pi'(s') = \pi(s')$. Since (T, π) is a sub-MDP, this implies that $\emptyset \neq \pi'(s') \subseteq A[s']$. Lastly, assume there are $s_1 \in C_s, \alpha \in \pi'(s_1), s_2 \in S$ such that $\delta(s_1, \alpha, s_2) > 0$. Then, since $s_1 \in T, \pi'(s_1) = \pi(s_1)$ and (T, π) is a sub-MDP, $s_2 \in T$. Now, if $s_2 \notin C_s$, then we must have $(s_1, \alpha) \in \mathrm{ROut}_{(T,\pi)}(C_s)$, but since we don't, $s_2 \in C_s$.

Claim 2: (C_s, π') is an end-component. **Proof:** Since C_s is an SCC, every $s_1 \in C_s$ can reach every $s_2 \in C_s$ using only vertices in C_s and actions from their π sets. Since the π' sets for each vertex in C_s are equal to their π sets, all pairs of vertices in C_s can reach other in (C_s, π'). The claim now follows from Claim 1.

Claim 3: $(C_s, \pi') = \mathrm{MEC}_{\mathcal{M}}(s)$. **Proof:** Since $s \in T$ and (T, π) is MEC-closed, we know $\mathrm{MEC}_{\mathcal{M}}(s) \subseteq (T, \pi)$. Suppose it is (T_0, π_0). We first show that $(T_0, \pi_0) \subseteq (C_s, \pi')$. Take any $s_0 \in T_0$, there is a path from s_0 to s (and vice versa) using states from T_0 and actions from their π_0 sets. Since $T_0 \subseteq T$ and $\pi_0(s') \subseteq \pi(s') \,\forall\, s' \in T_0$, s_0 and s can reach each other in (T, π). This implies that $s_0 \in C_s$. So $T_0 \subseteq C_s$. Next, for any $s' \in T_0$, we have $\pi_0(s') \subseteq \pi(s') = \pi'(s')$ since $s' \in C_s$. Thus, $(T_0, \pi_0) \subseteq (C_s, \pi')$. But both are end components and (T_0, π_0) is maximal, therefore (C_s, π') must be $\mathrm{MEC}_{\mathcal{M}}(s)$.

Claim 4: $(C_s, E \cap (C_s \times A \times C_s)) = G_{\mathrm{EBA}}(C_s, \pi')$. **Proof:** We will show that $E \cap (C_s \times A \times C_s) = \{(s_1, \alpha, s_2) \in C_s \times A \times C_s \mid \alpha \in \pi'(s_1) \wedge (\delta(s_1, \alpha, s_2) > 0)\}$. If $s_1 \in C_s, \alpha \in \pi'(s_1), s_2 \in C_s$ such that $\delta(s_1, \alpha, s_2) > 0$, then since $C_s \subseteq T$, $\pi'(s_1) = \pi(s_1)$, and $(V, E) = G_{\mathrm{EBA}}(T, \pi)$, we must have $(s_1, \alpha, s_2) \in E$, and thus, $(s_1, \alpha, s_2) \in E \cap (C_s \times A \times C_s)$. On the other hand, if $(s_1, \alpha, s_2) \in E \cap (C_s \times A \times C_s)$, then since $(V, E) = G_{\mathrm{EBA}}(T, \pi)$, we must have $\alpha \in \pi(s_1) = \pi'(s_1)$ and $\delta(s_1, \alpha, s_2) > 0$. □

We will next prove completeness by induction on the size of the sub-MDP. However, before we do that, we need to prove that all our recursive calls satisfy the preconditions that we require from our inputs.

Lemma 4.6. *Let (T, π) be an MEC-closed sub-MDP of some MDP $\mathcal{M} = (S, A, d_{\mathrm{init}}, \delta)$ and $(V, E) = G_{\mathrm{EBA}}(T, \pi)$. Consider the execution of* `MEC-Decomp-Interleave`*$(V, E\{v\})$ where $\{v\} = \emptyset$ or $v \in V$. If $\{v\} = \emptyset$, let $s \in V$ be the vertex picked on line 1. Otherwise, let $s = v$. If $X_1 \neq \emptyset$ and $V_1 \neq \emptyset$, then $(V_1, E_1 \cap (V_1 \times A \times V_1)) = G_{\mathrm{EBA}}(T_1, \pi_1)$ for some MEC-closed sub-MDP (T_1, π_1).*

Proof. Define, for all $s' \in V_1$, $\pi_1(s') = \{\alpha \in A \mid \exists t \in V_1 \,.\, (s', \alpha, t) \in E_1\}$. It is clear from this definition that $(V_1, E_1 \cap (V_1 \times A \times V_1)) = G_{\mathrm{EBA}}(V_1, \pi_1)$.

Claim 1: (V_1, π_1) is a sub-MDP. **Proof:** $V_1 \neq \emptyset$ from assumption.

- For any state $s' \in V_1$, there must be some $\alpha \in \pi(s')$ such that $(s', \alpha) \notin X_1$. This is because, if not, then, from the definition of Attr, we will have $s' \in U_1$, contradicting

the fact that $s' \in V_1 = C_s \setminus U_1$. Now, since $\alpha \in \pi(s')$, there is some $t \in T$ such that $(s', \alpha, t) \in E$. We have $(s', \alpha, t) \in E_1$ since $(s', \alpha) \notin X_1$. If $t \in U_1$, then $(s', \alpha) \in X_1$ (definition of Attr). If $t \in T \setminus C_s$, then $(s', \alpha) \in \text{ROut}_{(T,\pi)}(C_s)$, and thus $(s', \alpha) \in X_1$. These contradict the fact that $(s', \alpha) \notin X_1$. Thus, we must have $t \in V_1$. So $(s', \alpha, t) \in E_1 \cap (V_1 \times A \times V_1)$, and thus $\alpha \in \pi_1(s') \neq \emptyset$. We also have $\pi_1(s') \subseteq \pi(s') \subseteq A[s']$.

- Suppose $s' \in V_1, \alpha \in \pi_1(s'), t \in S$ such that $\delta(s', \alpha, t) > 0$. Since $\pi_1(s') \subseteq \pi(s')$, $t \in T$. If $t \in T \setminus C_s$ or $t \in U_1$, then $(s', \alpha) \in X_1$, which contradicts the fact that $\alpha \in \pi_1(s')$. Thus $t \in V_1$.

Claim 2: (V_1, π_1) is MEC-closed. **Proof:** Take any $s_1 \in V_1$. We will show that $\text{MEC}_{\mathcal{M}}(s_1) \subseteq (V_1, \pi_1)$. Suppose $\text{MEC}_{\mathcal{M}}(s_1) = (T_{s_1}, \pi_{s_1})$. Since (T, π) is MEC-closed and $s_1 \in V_1 \subseteq T$, $(T_{s_1}, \pi_{s_1}) \subseteq (T, \pi)$.

- Now, for any $s_2 \in T_{s_1}$, from the definition of an MEC, s_1 and s_2 can reach each other using only state-action pairs from $\text{sa}(T_{s_1}, \pi_{s_1})$. Since $\text{sa}(T_{s_1}, \pi_{s_1}) \subseteq \text{sa}(T, \pi)$, it follows that s_1, s_2 are in the same SCC (i.e., C_s). Now, if $s_2 \notin V_1$, then $s_2 \in U_1$, but then, from lemma 4.2, it cannot be in any MEC. Thus $s_2 \in V_1$. This shows that $T_{s_1} \subseteq V_1$.

- Take any $s_2 \in T_{s_1}$ and $\alpha \in \pi_{s_1}(s_2)$. Since (T_{s_1}, π_{s_1}) is a sub-MDP, there exists $t \in T_{s_1} \subseteq V_1$ such that $\delta(s_2, \alpha, t) > 0$. Further, $(s_2, \alpha, t) \in E_1$ because if not, then $(s_2, \alpha) \in X_1$ which is not possible due to lemma 4.2. The previous two assertions imply that $\alpha \in \pi_1(s_2)$. Thus, $\pi_{s_1}(s_2) \subseteq \pi_1(s_2)$ for all $s_2 \in T_{s_1}$.

□

Lemma 4.7. *Let (T, π) be an MEC-closed sub-MDP of some MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$ and $(V, E) = G_{\text{EBA}}(T, \pi)$. Consider the execution of* `MEC-Decomp-Interleave`*$(V, E\{v\})$ where $\{v\} = \emptyset$ or $v \in V$. If $\{v\} = \emptyset$, let $s \in V$ be the vertex picked on line 1. Otherwise, let $s = v$. If $V_2 \neq \emptyset$, then $(V_2, E_2 \cap (V_2 \times A \times V_2)) = G_{\text{EBA}}(T_2, \pi_2)$ for some MEC-closed sub-MDP (T_2, π_2).*

Proof. We know $V_2 = F_s \setminus C_s$. Define, for all $s' \in V_2$, $\pi_2(s') = \{\alpha \in A \mid \exists t \in V_2 \,.\, ((s', \alpha, t) \in E)\}$. It is clear from this definition that $(V_2, \pi_2) = G_{\text{EBA}}(V_2, E \cap (V_2 \times A \times V_2))$.

Claim 1: (V_2, π_2) is a sub-MDP. **Proof:** $V_2 \neq \emptyset$ by assumption.

- For any $s' \in V_2$, take any $\alpha \in \pi(s')$ (exists because (T, π) is a sub-MDP). From the definition of a sub-MDP, there is $t \in T$ such that $\delta(s', \alpha, t) > 0$. If $t \notin V_2$, then $(s', \alpha) \in \text{ROut}_{(T,\pi)}(F_s \setminus C_s)$, but we know that it is empty (lemma 4.3). Thus $t \in V_2$, implying $\alpha \in \pi_2(s') \neq \emptyset$. Also, $\pi_2(s') \subseteq \pi(s') \subseteq A[s']$.

- Let $s' \in V_2, \alpha \in \pi_2(s'), t \in S$ such that $\delta(s', \alpha, t) > 0$. Since $\pi_2(s') \subseteq \pi(s')$ and (T, π) is a sub-MDP, $t \in T$. If $t \notin V_2 = F_s \setminus C_s$, then $(s', \alpha) \in \text{ROut}_{(T,\pi)}(F_s \setminus C_s)$, but we know that it is empty (lemma 4.3). Thus, $t \in V_2$.

Claim 2: (V_2, π_2) is MEC-closed. **Proof:** Take any $s_1 \in V_2$. Suppose $\text{MEC}_{\mathcal{M}}(s_1) = (T_{s_1}, \pi_{s_1})$. We will show that $(T_{s_1}, \pi_{s_1}) \subseteq (V_2, \pi_2)$. Since (T, π) is MEC-closed and $s_1 \in V_2 \subseteq T$, $(T_{s_1}, \pi_{s_1}) \subseteq (T, \pi)$.

- Let $s_2 \in T_{s_1}$. From the definition of an MEC, s_1 can reach s_2 in (T_{s_1}, π_{s_1}). Since $(T_{s_1}, \pi_{s_1}) \subseteq (T, \pi)$, s_1 can reach s_2 in (T, π). Since $s_1 \in F_s$, this implies $s_2 \in F_s$. Further, if $s_2 \in C_s$, then s_2 can reach s, so s_1 can reach s, contradicting the fact that $s_1 \in F_s \setminus C_s$. Thus, $s_2 \in F_s \setminus C_s$. Since s_2 was arbitrary, we get $T_{s_1} \subseteq V_2$.
- Let $s_2 \in T_{s_1}, \alpha \in \pi_{s_1}(s_2)$. Since (T_{s_1}, π_{s_1}) is a sub-MDP, there is $t \in T_{s_1}$ such that $\delta(s_2, \alpha, t) > 0$. As $T_{s_1} \subseteq V_2$, we have $t \in V_2$. Then, by definition, $\alpha \in \pi_2(s_2)$. Thus, $\pi_{s_1}(s_2) \subseteq \pi_2(s_2)$ for all $s_2 \in T_{s_1}$.

□

Lemma 4.8. *Let (T, π) be an MEC-closed sub-MDP of some MDP $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$ and $(V, E) = G_{\text{EBA}}(T, \pi)$. Consider the execution of* `MEC-Decomp-Interleave`*$(V, E\{v\})$ where $\{v\} = \emptyset$ or $v \in V$. If $\{v\} = \emptyset$, let $s \in V$ be the vertex picked on line 1. Otherwise, let $s = v$. If $V_3 \neq \emptyset$, then $(V_3, E_3 \cap (V_3 \times A \times V_3)) = G_{\text{EBA}}(T_3, \pi_3)$ for some MEC-closed sub-MDP (T_3, π_3).*

Proof. Define, for all $s' \in V_3$, $\pi_3(s') = \{\alpha \in A \mid \exists t \in V_3 \,.\, (s', \alpha, t) \in E_3\}$. It is clear from this definition that $(V_3, E_3 \cap (V_3 \times A \times V_3)) = G_{\text{EBA}}(V_3, \pi_3)$.

Claim 1: (V_3, π_3) is a sub-MDP. **Proof:** $V_3 \neq \emptyset$ from assumption.

- For any state $s' \in V_3$, there must be some $\alpha \in \pi(s')$ such that $(s', \alpha) \notin X_3$. This is because, if not, then, from the definition of Attr, we will have $s' \in U_3$, contradicting the fact that $s' \in V_3 = (V \setminus F_s) \setminus U_3$. Now, since $\alpha \in \pi(s')$, there is some $t \in T$ such that $(s', \alpha, t) \in E$. We have $(s', \alpha, t) \in E_3$ since $(s', \alpha) \notin X_3$. If $t \in U_3$, then $(s', \alpha) \in X_3$ (definition of Attr). If $t \in F_s$, then $(s', \alpha) \in \text{ROut}_{(T,\pi)}(V \setminus F_s)$, and thus $(s', \alpha) \in X_3$. These contradict the fact that $(s', \alpha) \notin X_3$. Thus, we must have $t \in V_3$. So $(s', \alpha, t) \in E_3 \cap (V_3 \times A \times V_3)$, and thus $\alpha \in \pi_3(s') \neq \emptyset$. We also have $\pi_3(s') \subseteq \pi(s') \subseteq A[s']$.
- Suppose $s' \in V_3, \alpha \in \pi_3(s'), t \in S$ such that $\delta(s', \alpha, t) > 0$. Since $\pi_3(s') \subseteq \pi(s')$, $t \in T$. If $t \in F_s$ or $t \in U_3$, then $(s', \alpha) \in X_3$, which contradicts the fact that $\alpha \in \pi_3(s')$. Thus $t \in V_3$.

Claim 2: (V_3, π_3) is MEC-closed. **Proof:** Take any $s_1 \in V_3$. Suppose $\text{MEC}_{\mathcal{M}}(s_1) = (T_{s_1}, \pi_{s_1})$. We will show that $(T_{s_1}, \pi_{s_1}) \subseteq (V_3, \pi_3)$. Since (T, π) is MEC-closed and $s_1 \in V_3 \subseteq T$, $(T_{s_1}, \pi_{s_1}) \subseteq (T, \pi)$.

- Now, for any $s_2 \in T_{s_1}$, from the definition of an MEC, s_2 can reach s_1 in (T_{s_1}, π_{s_1}). Since $(T_{s_1}, \pi_{s_1}) \subseteq (T, \pi)$, s_2 can also reach s_1 in (T, π). So, since $s_1 \in V \setminus F_s$, we have $s_2 \in V \setminus F_s$. Now, if $s_2 \in U_3$, then, from lemma 4.4, it cannot be in any MEC. Thus $s_2 \in (V \setminus F_s) \setminus U_3 = V_3$. This shows that $T_{s_1} \subseteq V_3$.
- Take any $s_2 \in T_{s_1}$ and $\alpha \in \pi_{s_1}(s_2)$. Since (T_{s_1}, π_{s_1}) is a sub-MDP, there exists $t \in T_{s_1} \subseteq V_3$ such that $\delta(s_2, \alpha, t) > 0$. Further, $(s_2, \alpha, t) \in E_3$ because if not, then $(s_2, \alpha) \in X_3$ which is not possible due to lemma 4.4. The previous two assertions imply that $\alpha \in \pi_3(s_2)$. Thus, $\pi_{s_1}(s_2) \subseteq \pi_3(s_2)$ for all $s_2 \in T_{s_1}$.

□

Finally, we are ready to prove the correctness of our algorithm.

Theorem 4.9. *Let $\mathcal{M} = (S, A, d_{\text{init}}, \delta)$ be an MDP, (T, π) be an MEC-closed sub-MDP of $\mathcal{M}$ and $G_{\text{EBA}}(T, \pi) = (V, E)$. Then, `MEC-Decomp-Interleave`$(V, E\{v\})$ where $\{v\} = \emptyset$ or $v \in V$ outputs the graph representations of all MECs in* $\text{MECs}_{\mathcal{M}}(T)$.

Proof. Proof is by strong induction on the number of edges in $G_{\text{EBA}}(T, \pi) = (V, E)$. Note that from definitions 2.3 and 2.15, V must have at least one vertex and each vertex must have at least one outgoing edge.

Base Case ($|E| = 1$): If $\{v\} = \emptyset$, let $s \in V$ be the vertex picked on line 1. In this case, since every vertex has at least one outgoing edge, V must have just one vertex. So $(V, E) = (\{s\}, \{(s, \alpha, s)\})$ and $(T, \pi) = (\{s\}, s \mapsto \{\alpha\})$.

We have $F_s = C_s = \{s\}$, so $\text{ROut}_{(T,\pi)}(C_s)$, from the definition of ROut, is equal to $\emptyset$ (note that we must have $\delta(s, \alpha, s) = 1$ since (T, π) is a sub-MDP). Then, $\text{Attr}_{(T,\pi)}(\emptyset)$ is equal to $\emptyset$ (definition of Attr). So our algorithm will output $(C_s, E \cap (C_s \times A \times C_s)) = (\{s\}, \{(s, \alpha, s)\})$ as an MEC. This is indeed the MEC of s (also the only MEC in (T, π)) because (T, π) is MEC-closed, i.e., $\text{MEC}_{\mathcal{M}}(s) \subseteq (T, \pi)$, and an MEC must have at least one state and state-action pair. Note also that since $V \setminus F_s = \emptyset$ and $F_s \setminus C_s = \emptyset$, there will be no recursive calls.

Induction Hypothesis: For some $k \in \mathbb{N}$, for all sub-MDPs (T, π) with $G_{\text{EBA}}(T, \pi) = (V_{(T,\pi)}, E_{(T,\pi)})$ and $|E_{(T,\pi)}| \leq k$, `MEC-Decomp-Interleave`$(V_{(T,\pi)}, E_{(T,\pi)}, \{v\})$ with $v \in V_{(T,\pi)}$ or $\{v\} = \emptyset$ outputs the graph representations of all MECs of (T, π).

Induction Step ($|E| = k + 1$): Let $(V, E) = G_{\text{EBA}}(T, \pi)$ and $|E| = k + 1$. Consider the call `MEC-Decomp-Interleave`$(V, E, \{v\})$. If $\{v\} = \emptyset$, let $s \in V$ be the vertex picked on line 1. Otherwise, let $s = v$. To simplify this proof, assume that a call `MEC-Decomp-Interleave`$(V', E', \{v'\})$ with $V' = \emptyset$ doesn't output anything and simply returns (this preserves the behaviour of the algorithm since such a call is never made).

1. If $X_1 = \emptyset$, then the algorithm outputs $(C_s, E \cap (C_s \times A \times C_s))$, which from theorem 4.5, is $G_{\text{EBA}}(\text{MEC}_{\mathcal{M}}(s))$. If $X_1 \neq \emptyset$, then the recursive call `MEC-Decomp-Interleave`$(V_1, E_1 \cap (V_1 \times A \times V_1), \emptyset)$ is made. From the induction hypothesis (can apply due to lemma 4.6), it outputs $\{G_{\text{EBA}}(\text{MEC}_{\mathcal{M}}(s')) \mid s' \in V_1\}$. Since $V_1 = C_s \setminus U_1$ and no state in U_1 has an MEC (lemma 4.2), this is equal to $\{G_{\text{EBA}}(\text{MEC}_{\mathcal{M}}(s')) \mid s' \in C_s\}$. Thus the algorithm outputs $\{G_{\text{EBA}}(\text{MEC}_{\mathcal{M}}(s')) \mid s' \in C_s\}$ in both cases.

2. From the induction hypothesis (can apply due to lemma 4.7), the recursive call `MEC-Decomp-Interleave`$(V_2, E_2 \cap (V_2 \times A \times V_2), \{v'\})$ outputs $\{G_{\text{EBA}}(\text{MEC}_{\mathcal{M}}(s')) \mid s' \in V_2\}$. We know $V_2 = F_s \setminus C_s$. Thus the algorithm outputs $\{G_{\text{EBA}}(\text{MEC}_{\mathcal{M}}(s')) \mid s' \in F_s \setminus C_s\}$.

3. From the induction hypothesis (can apply due to lemma 4.8), the recursive call `MEC-Decomp-Interleave`$(V_3, E_3 \cap (V_3 \times A \times V_3), \emptyset)$ outputs $\{G_{\text{EBA}}(\text{MEC}_{\mathcal{M}}(s')) \mid s' \in V_3\}$. From lemma 4.4, no state in U_3 is in an MEC, thus since $V_3 = (V \setminus F_s) \setminus U_3$,

this is equal to $\{G_{\text{EBA}}(\text{MEC}_{\mathcal{M}}(s')) \mid s' \in (V \setminus F_s)\}$. So, the algorithm outputs $\{G_{\text{EBA}}(\text{MEC}_{\mathcal{M}}(s')) \mid s' \in V \setminus F_s\}$.

Putting the above three points together, the algorithm outputs $\{G_{\text{EBA}}(\text{MEC}_{\mathcal{M}}(s')) \mid s' \in V\}$. Since $T = V$, this completes the proof of the theorem. □

Chapter 5

Evaluation

The two MEC decomposition algorithms `LOCKSTEP` and `COLLAPSING` by Chatterjee et al. [CHL+18, CDHS21] are "better" than the `NAIVE` algorithm in terms of the asymptotic number of symbolic operations in the worst-case. However, this doesn't necessarily mean that they will be faster on a given MDP. In fact, [Fab23b] found "that the worst-case instances in which `LOCKSTEP` and `COLLAPSING` improve upon `NAIVE` do not occur often enough in order to yield a lower amount of symbolic operations.". There are a couple of subtleties in interpreting the worst-case asymptotic symbolic operations bound:

1. These are "worst-case" bounds. The MDPs for which the worst case occurs may not occur in most applications. So it is important to measure the performance on benchmarks derived from applications of probabilistic model checkers.

2. They provide a bound on the *number* of symbolic operations. However, the time taken for a symbolic operation depends on the size of the BDDs involved. Many operations on small BDDs may be faster than few operations on large BDDs.

Both of these stress the importance of performance evaluation on real-world benchmarks. In this chapter, we will measure the runtime and number of symbolic operations for the `NAIVE`, `LOCKSTEP`, and `INTERLEAVE` algorithms on the Quantitative Verification Benchmark Set (QVBS) [HKP+19].

Note: We do not evaluate the `COLLAPSING` algorithm in this thesis since it works on the G_{VBA} representation and model checkers natively use the G_{EBA} representation. However, [Fab23b] gave an algorithm to convert from G_{EBA} to G_{VBA} and then run the `COLLAPSING` algorithm. They found that `COLLAPSING` on G_{VBA} was competitive with `NAIVE` on G_{VBA}, but `NAIVE` on G_{EBA} was the fastest across all algorithms for all benchmarks.

`NAIVE` on G_{EBA} is thus the state-of-the-art in terms of runtime performance on the Quantitative Verification Benchmark Set. Since we compare `INTERLEAVE` against `NAIVE` on G_{EBA} and show that `INTERLEAVE` is faster on almost all (except 7) benchmarks, it is reasonable to conclude that `INTERLEAVE` is faster on the QVBS than the `COLLAPSING` algorithm with the converted G_{VBA} representation, although, since their conversion algorithm doesn't give the "best" G_{VBA} representation (see [Fab23b] for details), this evaluation of `COLLAPSING` might not be representative. It remains to be seen how `COLLAPSING` with a "native" G_{VBA} representation would perform.

5.1 Setup

Implementation. The implementations of `NAIVE` and `LOCKSTEP` were taken from Faber's implementations [Fab23a] in a custom build of the model checker `STORM` [HJK+22]. We adapted them to a newer version of `STORM` and added an implementation of the `INTERLEAVE` algorithm in the same format as the others. All three used the same implementation of the `SKELETON` SCC decomposition algorithm [GPP03]. We converted the recursive `INTERLEAVE` algorithm specified in the thesis to an iterative form for implementation. All the code can be found at this GitHub repository: https://github.com/Ramneet-Singh/storm-masters-thesis/tree/stable.

Benchmarks. The benchmarks consist of the Quantitative Verification Benchmark Set (QVBS) [HKP+19], of which all MDP models and the underlying MDPs of the Markov automata models were considered. Storm constructs the MDPs in their G_{EBA} representation. For the BDD operations, `STORM` supports either using the `CUDD` libary [Som05] or the multi-threaded library `Sylvan` [vD15]. We used the `CUDD` library.

For each of the 379 benchmarks, Storm constructed the G_{EBA} representation and computed an MEC decomposition with a total time limit of four minutes. There were 168 benchmarks on which at least one algorithm finished in time, and 128 benchmarks on which all three algorithms finished in time. We will focus on the 168 benchmarks in which at least one algorithm finished in time for measuring the runtime performance. On the 128 benchmarks on which all algorithms terminated in time, we measured the number of symbolic operations performed by each algorithm. We counted each non-basic set operation (like `exists`) on the (bigger) transition BDDs (complexity analyses usually focus on the number of Pre/Post operations, which amount to essentially the same thing).

Hardware. All benchmarks were performed on machines equipped with Intel(R) Xeon(R) CPU E5-2695 v2 processors running at 2.40GHz. The machines had 8 cores and 8 GB RAM, and the maximum memory used by `CUDD` was set to 4 GB for each benchmark.

5.2 MEC Decomposition Performance

In this section, we will look at the runtimes and number of symbolic operations of the `INTERLEAVE`, `NAIVE`, and `LOCKSTEP` algorithms on the QVBS. It has 379 benchmarks containing MDPs and Markov automata in total, out of which 11 are not supported by the `STORM` model checker (so the MDP model can't be built), leaving 368. Out of these, these are the number of benchmarks solved without timing out by each algorithm. `NAIVE` solved all 128 that `LOCKSTEP` did, and `INTERLEAVE` solved all 149 that `NAIVE` did. On the 149 benchmarks that both solved, `INTERLEAVE` had an average speedup of $3.81x$ over `NAIVE`.

Algorithm	Number of Benchmarks Solved
NAIVE	149
LOCKSTEP	128
INTERLEAVE	168

Table 5.1: Summary of the Number of QVBS Benchmarks Solved by Different Algorithms Within Timeout

5.2.1 Quantile Plot for Runtime

Figure 5.1 is a quantile plot of the runtimes of the algorithms on the 168 benchmarks that at least one algorithm solved within the timeout. A point (x, y) on the line for an algorithm means that it was able to solve x benchmarks within y seconds of runtime.

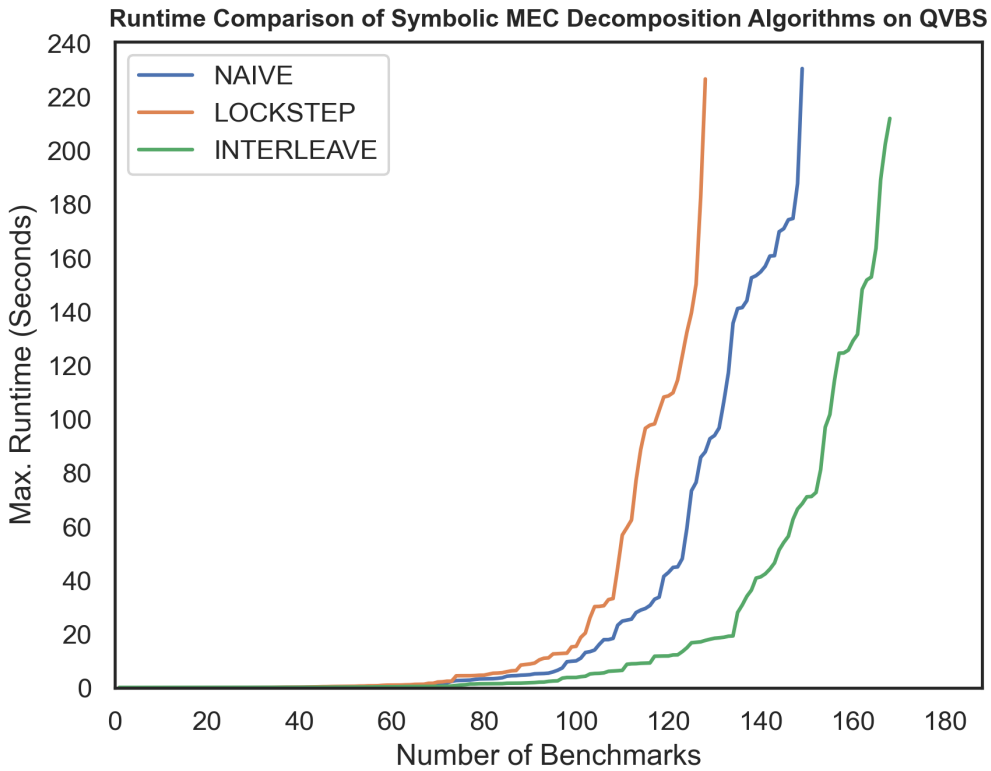


Figure 5.1: Quantile Plot of the Runtimes of Different Symbolic MEC Decomposition Algorithms on the QVBS

It is clear from the plot that given the same time-limit, `NAIVE` is able to solve more benchmarks than `LOCKSTEP`, and `INTERLEAVE` is able to solve more benchmarks than `NAIVE`. However, this doesn't necessarily mean that the runtime for each benchmark follows this order. To explore this, we will now look at pairwise scatter plots of the runtimes of each pair of algorithms for

individual benchmarks.

5.2.2 Pairwise Runtime Comparison on Individual Benchmarks

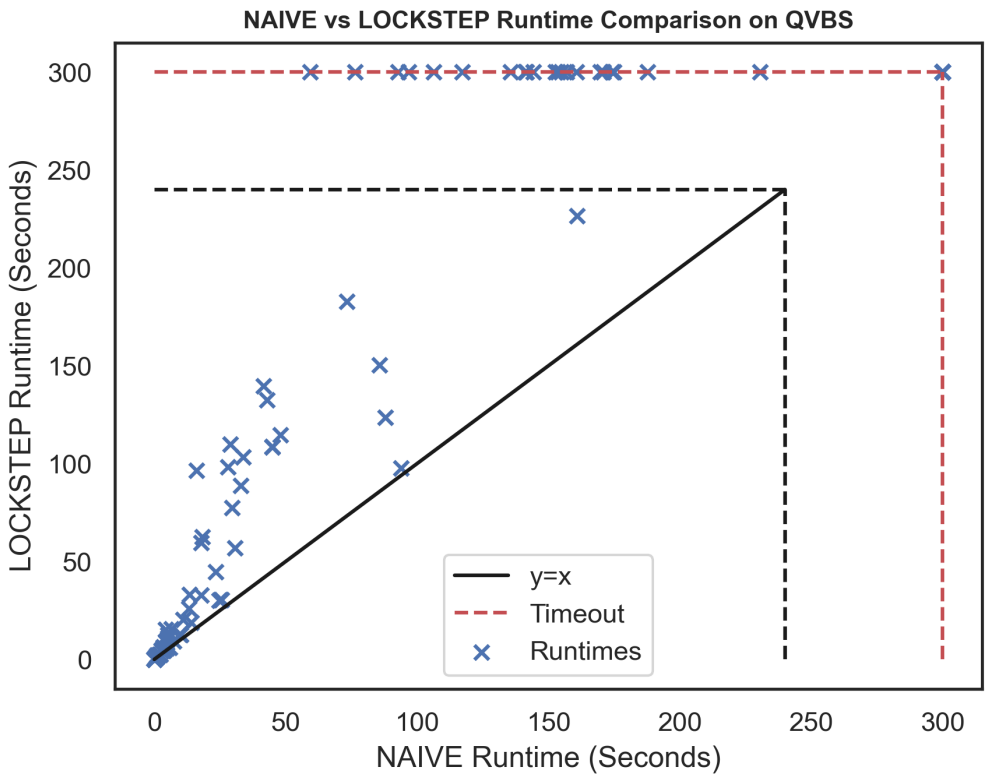


Figure 5.2: Scatter Plot of the Runtimes of `NAIVE` and `LOCKSTEP` on Individual QVBS Benchmarks

As figure 5.2 shows, the runtime of the `NAIVE` algorithm is less than or equal to the runtime of the `LOCKSTEP` algorithm for all (except 3) benchmarks. The points on the red horizontal line with $x < 250$ represent the 21 benchmarks that `NAIVE` solved and `LOCKSTEP` timed out on, while the farthest point on the red horizontal line represents the 19 benchmarks that both of these timed out on.

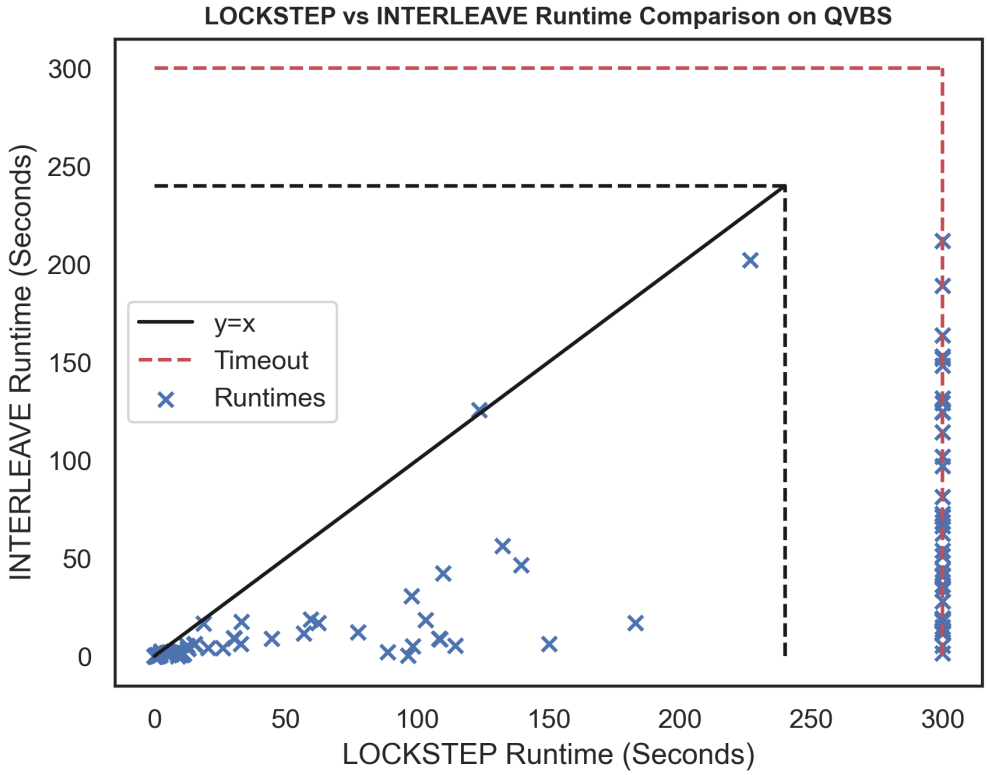


Figure 5.3: Scatter Plot of the Runtimes of `LOCKSTEP` and `INTERLEAVE` on Individual QVBS Benchmarks

Figure 5.3 shows that the runtime of the `INTERLEAVE` algorithm is less than or equal to the runtime of the `LOCKSTEP` algorithm for almost all benchmarks. `INTERLEAVE` is slower than `LOCKSTEP` on only 2 benchmarks. The points on the red vertical line represent the 40 benchmarks that `INTERLEAVE` solved and `LOCKSTEP` timed out on.

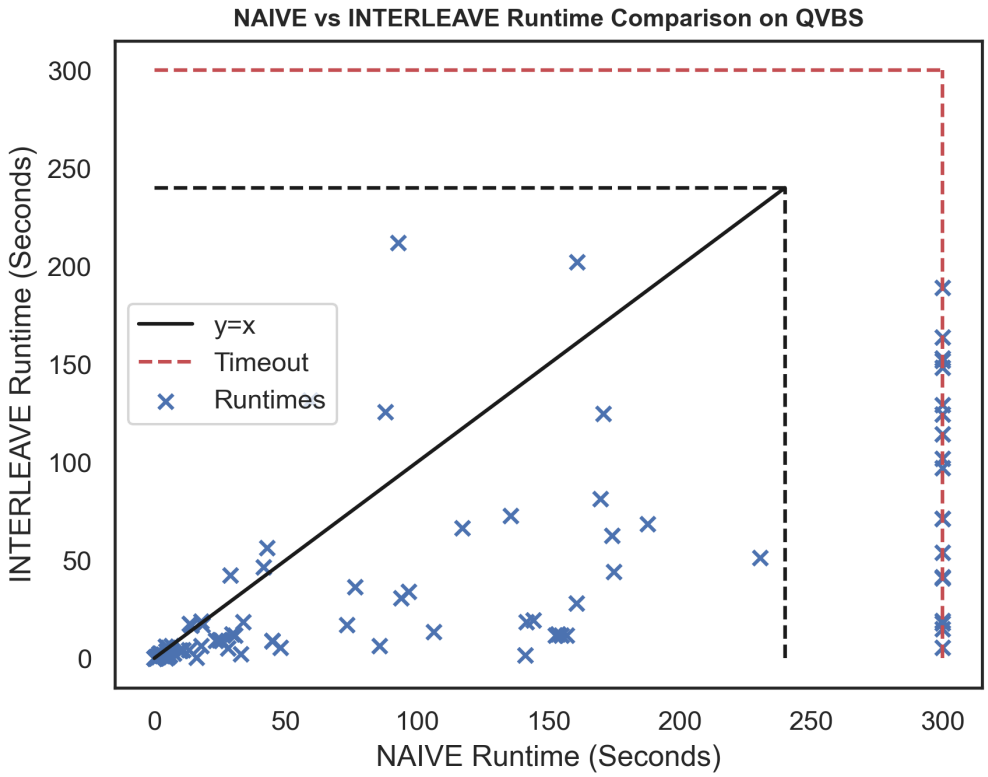


Figure 5.4: Scatter Plot of the Runtimes of `NAIVE` and `INTERLEAVE` on Individual QVBS Benchmarks

There are 24 benchmarks for which `INTERLEAVE` is slower than `NAIVE`. For the rest 142 that they both solved, Figure 5.4 shows that `INTERLEAVE` is faster or matches the runtime of `NAIVE`. For small running times, their runtimes are similar, but performance diverges as running time increases. Most benchmarks with large running time have `INTERLEAVE` as faster, while a few have `NAIVE` as faster. The points on the red vertical line represent the 19 benchmarks that `INTERLEAVE` solved and `NAIVE` timed out on.

5.2.3 Quantile Plot for Number of Symbolic Operations

Figure 5.5 is a quantile plot of the number of symbolic operations performed by the algorithms on the 128 benchmarks that all algorithms solved within the timeout. A point (x, y) on the line for an algorithm means that it was able to solve x benchmarks with the number of symbolic operations being less than y.

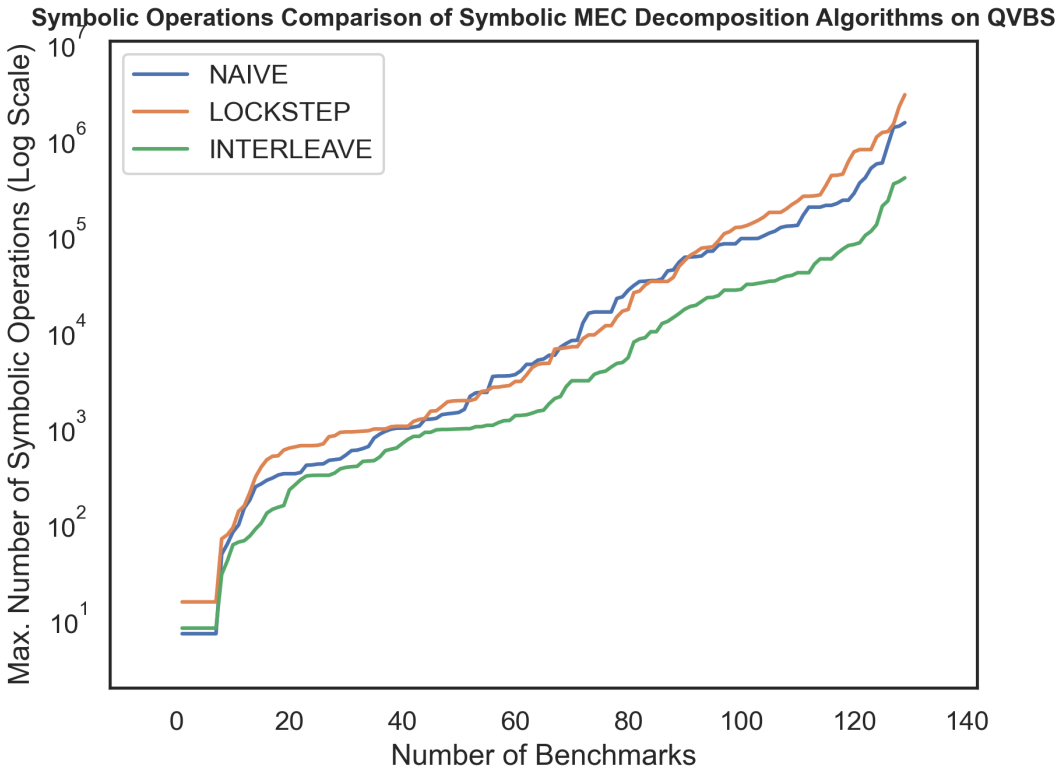


Figure 5.5: Quantile Plot of the Number of Symbolic Operations Performed by Different Symbolic MEC Decomposition Algorithms on the QVBS

It is clear from the plot that excepting very low limits, given the same limit on the number of symbolic operations, `INTERLEAVE` is able to solve the most benchmarks. Between `NAIVE` and `LOCKSTEP`, however, there is no clear winner. For some limits, `LOCKSTEP` solves more benchmarks while `NAIVE` solves more for other limits. However, this only gives an idea of the distributions of the number of symbolic operations, and doesn't tell us how they compare for individual benchmarks. To explore this, we will now look at pairwise scatter plots of the number of symbolic operations performed by each pair of algorithms for individual benchmarks.

5.2.4 Pairwise Symbolic Operations Comparison on Individual Benchmarks

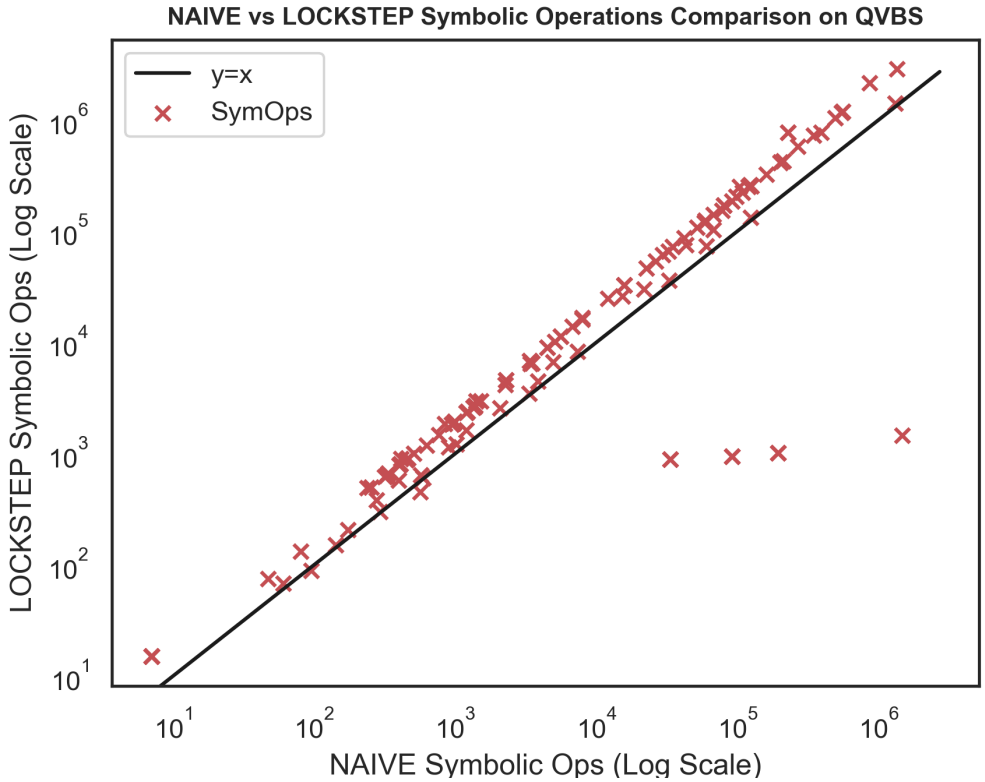


Figure 5.6: Scatter Plot of the Number of Symbolic Operations Performed by `NAIVE` and `LOCKSTEP` on Individual QVBS Benchmarks

From figure 5.6, it is clear that `NAIVE` performs fewer or equal number of symbolic operations than `LOCKSTEP` on almost all benchmarks.

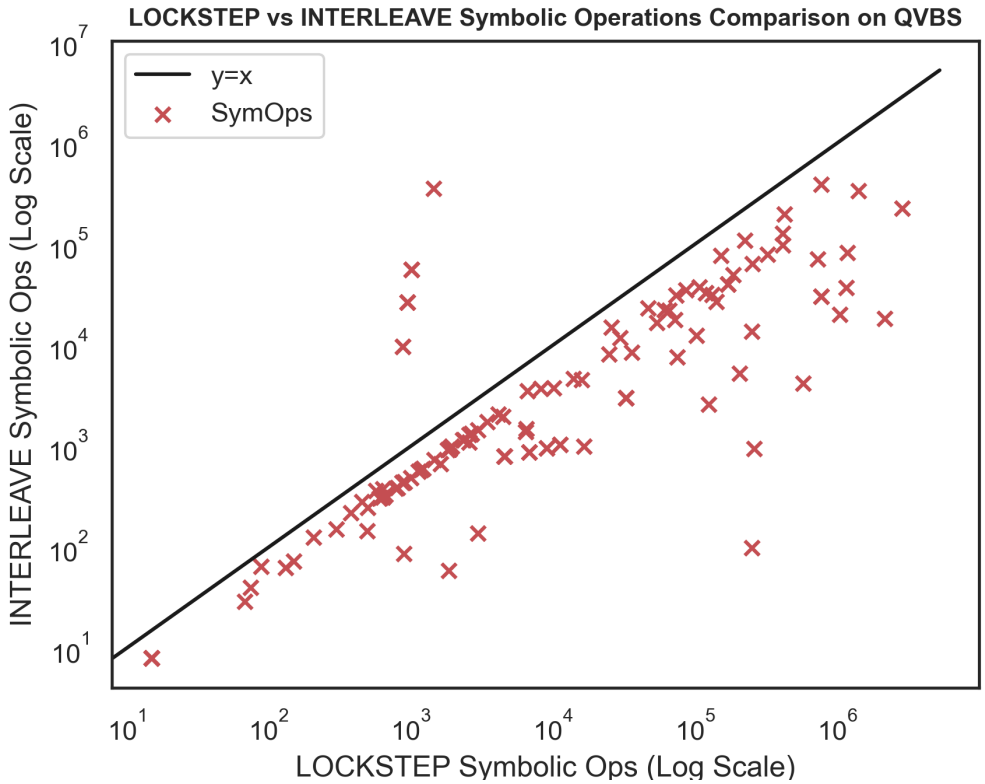


Figure 5.7: Scatter Plot of the Number of Symbolic Operations Performed by `LOCKSTEP` and `INTERLEAVE` on Individual QVBS Benchmarks

Figure 5.7 shows that the `INTERLEAVE` algorithm performs fewer symbolic operations than the `LOCKSTEP` algorithm for almost all benchmarks.

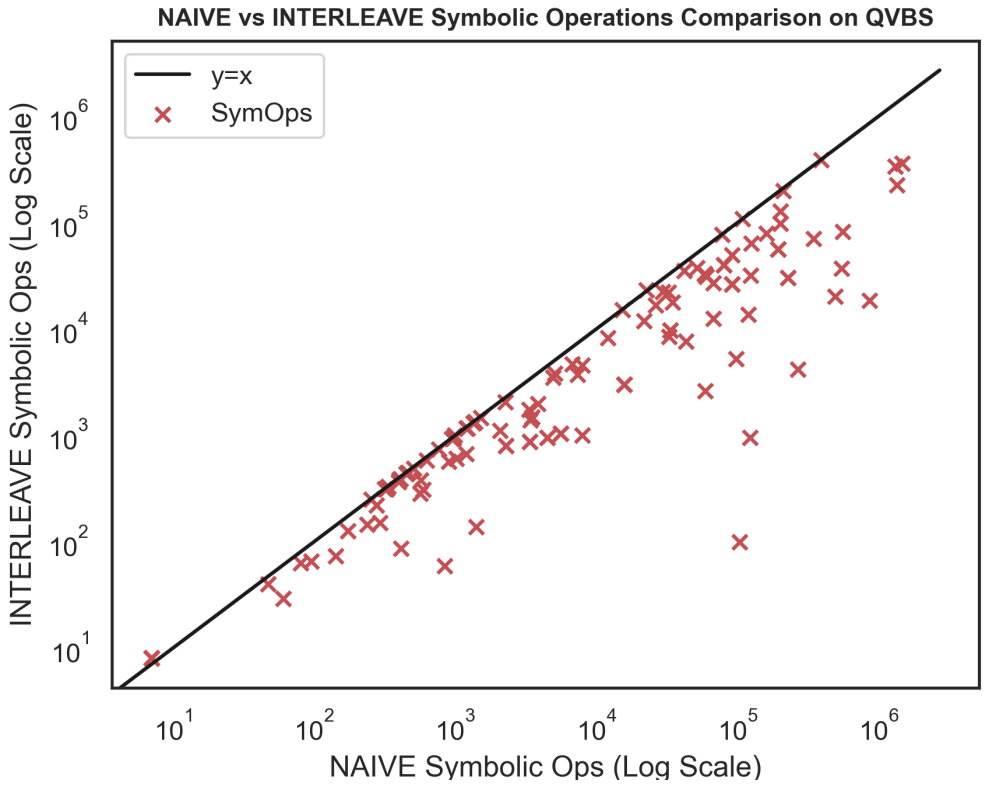


Figure 5.8: Scatter Plot of the Number of Symbolic Operations Performed by `NAIVE` and `INTERLEAVE` on Individual QVBS Benchmarks

From figure 5.8, it is clear that `INTERLEAVE` performs fewer or equal number of symbolic operations than `NAIVE` on all benchmarks. This seems to correlate with the runtimes, though the difference in runtimes is more stark.

5.2.5 Algorithmic Analysis of `INTERLEAVE`'s Performance

In this subsection, we will try to analyse the reasons for `INTERLEAVE`'s performance. To do this, we will compare `INTERLEAVE` with the `NAIVE` algorithm, and see the difference in the steps performed by both.

Since `NAIVE` first calls the `SKELETON` SCC decomposition function, `INTERLEAVE` and `NAIVE` perform the exact same steps until they have found the first SCC. Then, `NAIVE` adds the SCC to a queue for processing, and moves on to find other SCCs of the original graph, calling `SKELETON` on $F_v \setminus \mathrm{SCC}_v$ (with a new spine set) and $V \setminus F_v$. `INTERLEAVE` does the following things differently:

1. Before adding the SCC to a queue for processing (i.e., making the recursive call on SCC_v), it removes the ROut and its Attr. But this would have been the first thing `NAIVE` did when it picked up the SCC for processing, so it shouldn't make a difference in the runtime.

2. In the second recursive call, it doesn't pass a spine set, but only a single vertex to start

from. This also means that there is no spine to pass in the third recursive call after one second call.

3. Before making the third recursive call on $V \setminus F_v$, it removes the ROut and its Attr. `NAIVE` would have done this after computing the SCCs of $V \setminus F_v$ (when it would process each SCC separately and remove its ROut and Attr), while `INTERLEAVE` does it before it even starts computing the SCCs of $V \setminus F_v$. The difference is `NAIVE` would then have to remove more states and state-action pairs than `INTERLEAVE` has to remove (since it's the ROut of different SCCs rather than the ROut of the entire rest of the graph).

The second point suggests that `INTERLEAVE` won't be able to find SCCs as quickly as `NAIVE` does. However, the third point means that `INTERLEAVE` removes useless states and state-action pairs more prematurely from the rest of the graph instead of waiting for the SCCs to be computed and then removing for each of them. Removing a few states and state-action pairs from a large set may be preferrable to removing many states and state-action pairs from many small sets. Further, removing them prematurely also change the SCCs computed, but that would have to happen anyway, so `INTERLEAVE` manages to avoid some unnecessary computation. This also agrees with the observation that `INTERLEAVE` generally performs fewer symbolic operations, and so is our hypothesis for why `INTERLEAVE` performs well empirically.

Chapter 6

Conclusion and Future Work

In summary, this thesis presents a novel symbolic algorithm for the MEC Decomposition of MDPs. The main insight behind the algorithm is to interleave the computation of MECs with the computation of SCCs (which is done anyway by existing algorithms) instead of doing them in two separate phases. We believe that this idea is a promising one, and provides a simple basic algorithm upon which to build. There are various possible avenues for future work:

1. As we have seen, the worst-case complexity of `INTERLEAVE` is $O(n^2)$ symbolic operations and $O(\log n)$ symbolic space for an MDP with n vertices and m edges. It would be exciting to see a symbolic algorithm using a similar idea and also achieving better worst-case complexity. In particular, we tried to use the ideas of a skeleton and spine set from the `SKELETON` algorithm for this, but failed. We have included the problems we faced and how we ended up with `INTERLEAVE` in this thesis (see section 4.5), in the hope that future work may solve them.

2. Algorithms like `LOCKSTEP` and `COLLAPSING` have been built to work with the G_{VBA} representation instead of the G_{EBA} representation that model checkers use. While it is possible to write `LOCKSTEP` to deal with G_{EBA} without changing the algorithm much, this is not possible for `COLLAPSING`. Faber's thesis [Fab23b] got around this problem by converting the G_{EBA} to (not necessarily the best possible) G_{VBA} before passing it to `COLLAPSING`. But it remains to be seen how `COLLAPSING` would perform with a "native" G_{VBA} representation. We believe that the construction of these representations, and the difference in performance of algorithms on them is an important area for future work.

3. Chapter 5 showed that there is a chasm between the empirical performance of algorithms (at least on the QVBS) and their worst-case symbolic complexity. This implies a gap in our understanding of the classes of MDPs for which each algorithm is well-suited. For example, it may be the case that the MDPs which occur in benchmarks do not trigger the $O(n^2)$ worst-case for `NAIVE` and `INTERLEAVE`. We can't analyse the algorithms well-enough (yet) to reason about their empirical performance on different MDPs. This is another promising direction for future work.

Bibliography

[Ake78] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27:509–516, 1978.

[Alf98] Luca Alfaro. Formal verification of probabilistic systems. Technical report, Stanford University, Stanford, CA, USA, 1998.

[Bai17] Christel Baier. Tutorial: Probabilistic model checking, summer school 2017: Verification technology, systems & applications. https://resources.mpi-inf.mpg.de/departments/rg1/conferences/vtsa17/slides/baier-vsta17-MDP-final.pdf, July 2017. [Accessed 21-06-2024].

[BFG+97] Ruth Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and F. Somenzi. Algebric decision diagrams and their applications. *Formal Methods in System Design*, 10:171–206, 1997.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BKL+17] Christel Baier, Joachim Klein, Linda Leuschner, D. Parker, and Sascha Wunderlich. Ensuring the reliability of your model checker: Interval iteration for markov decision processes. In *International Conference on Computer Aided Verification*, 2017.

[Bry85] Randal E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. *22nd ACM/IEEE Design Automation Conference*, pages 688–694, 1985.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24:293–318, 1992.

[BW96] Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Computers*, 45:993–1002, 1996.

[CDHL16] Krishnendu Chatterjee, Wolfgang Dvořák, Monika Henzinger, and Veronika Loitzenbauer. Model and objective separation with conditional lower bounds: Disjunction is harder than conjunction. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, page 197–206, New York, NY, USA, 2016. Association for Computing Machinery.

[CDHL17] Krishnendu Chatterjee, Wolfgang Dvořák, Monika Henzinger, and Veronika Loitzenbauer. Lower bounds for symbolic computation on graphs: Strongly connected components, liveness, safety, and diameter. In *ACM-SIAM Symposium on Discrete Algorithms*, 2017.

[CDHS19] Krishnendu Chatterjee, Wolfgang Dvořák, Monika Henzinger, and Alexander Svozil. Near-linear time algorithms for streett objectives in graphs and mdps. *ArXiv*, abs/1909.05539, 2019.

[CDHS21] Krishnendu Chatterjee, Wolfgang Dvořák, Monika Henzinger, and Alexander Svozil. Symbolic time and space tradeoffs for probabilistic verification. *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2021.

[CG18] Sagar Chaki and Arie Gurfinkel. *BDD-Based Symbolic Model Checking*, pages 219–245. Springer Publishing Company, Incorporated, 1st edition, 05 2018.

[CH11] Krishnendu Chatterjee and Monika Henzinger. Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In *ACM-SIAM Symposium on Discrete Algorithms*, 2011.

[CH14] Krishnendu Chatterjee and Monika Henzinger. Efficient and dynamic algorithms for alternating büchi games and maximal end-component decomposition. *Journal of the ACM (JACM)*, 61:1 – 40, 2014.

[CHL+18] Krishnendu Chatterjee, Monika Henzinger, Veronika Loitzenbauer, Simin Oraee, and Viktor Toman. Symbolic algorithms for graphs and markov decision processes with fairness objectives. *ArXiv*, abs/1804.00206, 2018.

[dAKN+00] Luca de Alfaro, M. Kwiatkowska, Gethin Norman, D. Parker, and Roberto Segala. Symbolic model checking of probabilistic processes using mtbdds and the kronecker representation. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2000.

[EFT91] Reinhard Enders, Thomas Filkorn, and Dirk Taubner. Generating bdds for symbolic model checking in ccs. *Distributed Computing*, 6:155–164, 1991.

[Fab23a] Felix Faber. Comparison of Maximal End Component Decomposition Algorithms: Data and Code, September 2023.

[Fab23b] Felix Faber. Comparison of Symbolic Maximal End Component Decomposition Algorithms. https://felixfaber.dev/thesis.pdf, September 2023. [Accessed 21-06-2024].

[FMY97] Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10:149–169, 1997.

[FV96] Jerzy Filar and Koos Vrieze. *Competitive Markov decision processes*. Springer-Verlag, Berlin, Heidelberg, 1996.

[GPP03] Raffaella Gentilini, Carla Piazza, and Alberto Policriti. Computing strongly connected components in a linear number of symbolic steps. In *ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[HJK+22] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker storm. *International Journal on Software Tools for Technology Transfer*, 24(4):589–610, Aug 2022.

[HKP+19] Arnd Hartmanns, Michaela Klauck, David Parker, Tim Quatmann, and Enno Ruijters. The quantitative verification benchmark set. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 344–350, Cham, 2019. Springer International Publishing.

[HM18] Serge Haddad and Benjamin Monmege. Interval Iteration Algorithm for MDPs and IMDPs. *Theoretical Computer Science*, 735:111 – 131, July 2018.

[HR04] Michael Huth and Mark Ryan. *Logic in computer science : modelling and reasoning about systems*. Cambridge University Press, Cambridge [U.K.]; New York, 2004.

[KNP02] M. Kwiatkowska, Gethin Norman, and D. Parker. Prism: Probabilistic symbolic model checker. In *Computer Performance Evaluation / TOOLS*, 2002.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[KPR18] Jan Křetínský, Guillermo A. Pérez, and Jean-François Raskin. Learning-based mean-payoff optimization in an unknown mdp under omega-regular constraints. In *International Conference on Concurrency Theory*, 2018.

[Lee59] Choong Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[LSS+23] Casper Abild Larsen, Simon Schmidt, Jesper Steensgaard, Anna Blume Jakobsen, Jaco van de Pol, and Andreas Pavlogiannis. A truly symbolic linear-time algorithm for scc decomposition. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2023.

[Put14] M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 2014.

[Som05] F. Somenzi. Cudd: Cu decision diagram package release 2.4. 1. *University of Colorado at Boulder*, 01 2005.

[Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[vD15] Tom van Dijk. Sylvan: multi-core decision diagrams. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2015.

[WKB14] Anton Wijs, Joost-Pieter Katoen, and Dragan Bosnacki. Gpu-based graph decomposition into strongly connected and maximal end components. In *International Conference on Computer Aided Verification*, 2014.