Computer Science • 23(1) 2022 https://doi.org/10.7494/csci.2022.23.1.4139

Mohammadsadegh Mohagheghi
Jaber Karimpour
Ayaz Isazadeh

IMPROVING MODIFIED POLICY ITERATION FOR PROBABILISTIC MODEL CHECKING

Abstract *Along with their modified versions, value iteration and policy iteration are well-known algorithms for the probabilistic model checking of Markov decision processes. One challenge with these methods is that they are time-consuming in most cases. Several techniques have been proposed to improve the performance of iterative methods for probabilistic model checking; however, the running times of these techniques depend on the graphical structure of the utilized model. In some cases, their performance can be worse than the performance of standard methods. In this paper, we propose two new heuristics for accelerating the modified policy iteration method. We first define a criterion for the usefulness of the computations of each iteration of this method. The first contribution of our work is to develop and use a criterion to reduce the number of iterations in modified policy iteration. As the second contribution, we propose a new approach for identifying useless updates in each iteration. This method reduces the running time of the computations by avoiding the useless updates of states. The proposed heuristics have been implemented in the PRISM model checker and applied on several standard case studies. We compare the running time of our heuristics with the running times of previous standard and improved methods. Our experimental results show that our techniques yields a significant speed-up.*

Keywords probabilistic model checking, Markov decision processes, modified policy iteration, probabilistic reachability

Citation Computer Science 23(1) 2022: 63–80

1. Introduction

Model checking is an automated formal verification approach that is used to verify computer systems. In this approach, labeled transition systems are usually used to model systems, and temporal logics are used to specify a system's properties. Some computer systems have stochastic behaviors; therefore, probabilistic model checking should be used to analyze their quantitative properties. In this domain, Markov decision processes (MDPs) are used to model both probabilistic and non-deterministic behaviors [3]. Probabilistic computation tree logic (PCTL) is used to specify a wide range of system properties [3, 4]. A main class of PCTL properties is optimal (maximum or minimum) reachability probabilities [9, 12]. In most cases, numerical computations are used to calculate these probabilities. Value iteration and policy iteration are widely used for approximating optimal reachability probabilities [3, 9, 19].

Several model checker tools have been proposed over the last 20 years to perform probabilistic model checking. PRISM [6] and STORM [7] are well-known tools that are used to analyze systems with probabilistic behaviors. A wide range of examples from practical and academical works are available from the PRISM website (http://www.prismmodelchecker.org/).

Excessive time or space requirements usually limit the efficiency or feasibility of model checking in all variants. The main reason for this limitation is the state-space explosion problem [3, 4, 9, 12]. In addition, numerical computations are time-consuming in the case of probabilistic model checking [6, 9]. Several approaches have been proposed to tackle this problem; however, the performance of the proposed methods depends on the graphical structure of the underlying model. In some cases, the running time of these methods is more than the running times of standard iterative methods [5, 6, 16, 17].

1.1. Referred problems in work

The overall problem that is addressed in this paper is to accelerate the modified policy iteration (MPI) [19] for probabilistic model checking. In this method, the value updates of some iterations are useless and time-consuming. In this case, some states of the model can be used for a limited number of iterations but are useless in other iterations. A solution to this problem is to pick (and, if necessary, modify) an existing formalism or introduce a new one with the following specific characteristics:

1. Must have precise and sound syntax and semantics.
2. Must be compatible with previous work.
3. Must reduce total number of iterations while preserving accuracy of solutions.
4. Must avoid unnecessary updates.
5. Must be able to be used in parallel computing machines.

Our solution to this problem (stated clearly as our claim in Section 1.3) satisfies Items 1–4; this sets the stage for future work on the next item.

1.2. Motivation and historical perspective of problem

Several approaches have been proposed over the past 15 years to tackle the state-space explosion problem for probabilistic model checking. Symbolic model checking [13,18], compositional verification [8], statistical model checking [2,11], symmetry reduction [14], and incremental model construction [21] are the main parts of these approaches that reduce the memory that is needed to store the information of the model. Several other reduction techniques have been proposed to reduce the running time of probabilistic model checking. SCC-based approaches [1,16] identify any strongly connected components (SCCs) of an underlying model and compute the reachability values of the states of each component in the correct order. A learning-based algorithm was proposed in [6] to solve the reachability probability problems of MDPs. An extrapolation technique was proposed in [20] to approximate the optimal policies. A deep-learning method was developed in [10] to predict the optimal policies for large MDP models. Interval iteration was proposed in [5] for computing PCTL properties with the desired accuracy. The main challenge of the standard and improved iterative methods is that they perform some redundant updates in their computations [6,16].

1.3. Claim

In this paper, we focus on maximal reachability probabilities. We claim that a solution that satisfies Items 1-4 of Sect. 1.1 is possible by doing the following work.

1. We propose a heuristic to determine the number of iterations after each policy improvement. This can reduce the total number of iterations of the MPI method while preserving the accuracy of the solutions.

2. We develop an approach for detecting useless updates in order to avoid them in most iterations. Using this approach for MPI, we can reduce the number of useful states after each policy modification and improve the performance of this iterative method. Although this technique does not reduce the whole number of states, its main advantage is to avoid the useless updates of the states.

1.4. Paper structure

After the introduction in this section, we will review some related definitions and standard iterative methods for computing reachability probabilities of MDPs in Section 2. In Section 3, we describe our techniques for accelerating MPI. Section 4 presents our experimental results. Finally, Section 5 concludes the paper.

2. Background

In this section, we provide an overview of MDPs and standard algorithms for computing optimal reachability probabilities. We mainly follow the notations of [4,9]. We use $Dist(S)$ as the set of all discrete probability distributions over a finite set S; i.e., the set of all $p : S \to [0,1]$ functions for which $\sum_{s \in S} p(s) = 1$.

Definition 1 (Markov Decision Process) A Markov decision process (MDP) is a tuple $M = (S, s_0, Act, \delta)$ where S is a finite set of states, $s_0 \in S$ is an initial state, Act is a finite set of actions, and $\delta : S \times Act \rightarrow Dist(S)$ is a probabilistic transition function. The probability of a transition from s to s' by action $a \in Act(s)$ is shown by $\delta(s,a)(s')$. The size of M (which is shown by $|M|$) is defined as the number of states of M plus the number of its transitions. For each $s \in S$ state of an MDP M, one or more actions of Act are defined as enabled actions. We define this set as $Act(s) = \{a \in Act \mid \delta(s,a)$ is defined$\}$. For each $s \in S$ and $a \in Act(s)$, we use $Post(s,a)$ for the set of a successors of s [9]:

$$Post(s,a) = \{s' \in S \mid \delta(s,a)(s') > 0\}.$$

A discrete-time Markov chain (DTMC) is an MDP for which each state has exactly one enabled action [3]. A path in M is a non-empty (finite or infinite) sequence $\pi = s_0 \overset{a_0}{\rightarrow} s_1 \overset{a_1}{\rightarrow} ...$ where $s_i \in S$ and $a_i \in Act(s_i)$ and $s_{i+1} \in Post(s_i, a_i)$ for each $i \geq 0$. We use $Path_s$ to denote the set of all infinite paths of M that start in state s. We also use $\pi(i)$ to denote the (i+1)-th state in path π (i.e., $\pi(i) = s_i$). For reasoning about the probabilistic behavior of an MDP M, we use the notion of adversary (also called policy). In this paper, we use only *deterministic* and *memory-less* adversaries that are sufficient for computing reachability probabilities [3, 9].

Definition 2 (Deterministic Adversary) A deterministic adversary of an MDP M is defined as a $\sigma : FPath_s \rightarrow Act$ function where, for each finite path $\pi = s_0 \overset{a_0}{\rightarrow} s_1 \overset{a_1}{\rightarrow} ... \overset{a_{i-1}}{\rightarrow} s_i$ selects an enabled action $a_i \in Act(s_i)$. An adversary σ is called *memory-less* if it depends only on the last state of the path. We use Adv_M for the set of all deterministic adversaries of M.

Definition 3 (Quotient DTMC) The quotient DTMC for an MDP $M = (S, s_0, Act, \delta)$ and a deterministic finite-memory adversary σ is the finite state DTMC $M^\sigma = (S, s_0, P)$ where S and s_0 are the same as in M, and $P : S \times S \rightarrow [0,1]$ is a transition probability matrix that is defined as $P(s,s') = \delta(s, \sigma(s))(s')$ [9]. We use $Path_s^\sigma$ for the set of all (infinite) paths of the quotient DTMC M^σ that start in a state s.

2.1. Reachability probabilities

Temporal logics (such as PCTL [9, 12] and probabilistic LTL [4]) are commonly used for specifying those properties that should be verified against MDPs. A main class of PCTL properties is the set of extremal reachability probabilities; i.e., the maximal or minimal probability of reaching a state in some final set $F \subseteq S$ when starting in s:

$$P_s^{min}(F) = inf_{\sigma \in Adv_M} p_s^\sigma(F), P_s^{max}(F) = sup_{\sigma \in Adv_M} p_s^\sigma(F),$$

where $p_s^\sigma(F) = Prob_s^\sigma(\{\pi \in Path_s^\sigma | \exists i.\pi(i) \in F\})$. Probability space $Prob_s^\sigma$ is defined over $Path_s^\sigma$. More details about the definition of this probability space can be found in [2, 4, 8]. The reachability probabilities are calculated in two steps. The first step (which is called pre-computation) uses a graph-based analysis to partition state space S into three sets: $S^y = \{s \in S | P_s^{max}(F) = 1\}$,$S^n = \{s \in S | P_s^{max}(F) = 0\}$ and $S^? = S/(S^y \cup S^n)$. The second step (quantitative analysis) uses numerical computations to determine the reachability values of the states in $S^?$.

2.2. Quantitative reachability

The reachability probabilities can be defined as the solutions to the *Bellman equations.* If we define $x_s = P_s^{max}(F)$ for every state $s \in S$, then the maximum reachability probabilities are the least solution of the *Bellman equations*:

$$\begin{aligned} x_s &= 1 && \text{if} \quad s \in S^y \\ x_s &= 0 && \text{if} \quad s \in S^n \\ x_s &= max_{a \in Act(s)} \sum_{s' \in S} \delta(s,a).x_{s'} && Otherwise \end{aligned}$$

Numerical computations can solve these equations [4,9,16]. Most probabilistic model checkers (such as PRISM [15] and STORM [7]) use iterative methods such as value and policy iteration to solve these equations. We review the idea of policy iteration and its modified version. More details about value iteration and policy iteration and their convergence criteria are available in [9,17,19].

2.3. Policy iteration

The idea of policy iteration is to generate a sequence of memory-less adversaries and construct the quotient DTMC of each adversary. The method uses standard numerical methods such as Gauss-Seidel (GS) [9] to compute the reachability probabilities of the states in the corresponding DTMCs. After satisfying the stopping criterion for the computation of any quotient DTMC, the method improves the adversary and continues the computations in another cycle of the iteration. This terminates when it reaches a situation where there is no further change in the computed adversary [9].

2.4. Modified policy iteration

The idea of MPI [19] is to update each adversary after a fixed number of iterations (100 iterations, for example). Figure 1 describes the algorithm of this method; it initiates the values of x_s in Line 2. A cycle of policy iteration starts in Line 5 and continues through Line 18; it applies a limited number of iterations (shown by max_iters) of the Gauss-Seidel method for the related quotient DTMC (Lines 6–14) to update the values of the states of $S^?$. The policy iterations continue until satisfying both convergence criteria: having the same two subsequent adversaries, and the maximum difference of two consecutive values of the states becomes less than the threshold ϵ.

3. Our improvements for modified policy iteration

In this section, we propose two heuristics for accelerating MPI.

Algorithm 1 Modified Policy Iteration for $P_s^{Max}(F)$.

```
    input: an MDP $M = (S, s_0, Act, \delta)$, a target set $F \subseteq S$, stopping threshold $\epsilon$ and $max\_iters$.
    output: Approximation of $P_s^{max}(F)$ for all $s \in S^?$
 1: for all $s \in S$ do
 2:     $x_s \leftarrow \begin{cases} 1, & \text{if } s \in S^y \\ 0, & \text{otherwise} \end{cases}$
 3: end for
 4: Select arbitrary adversary $\sigma$.
 5: do
 6:     for i = 1 to max_iters do
 7:         diif ← 0;
 8:         for all $s \in S^?$ do
 9:             $x_{new} \leftarrow \sum_{s' \in S} \delta(s, \sigma(s))(s') \times x_{s'}$;
10:             $diff \leftarrow max(diff, (x_{new} - x_s)/x_s)$;
11:             $x_s \leftarrow x_{new}$;
12:         end for
13:         if diff $\leq \epsilon$ then
14:             break;
15:         end if
16:     end for
17:     for all $s \in S$ do
18:         $\sigma(s) \leftarrow argmax_{a \in Act(s)} \sum_{s' \in S} \delta(s, a)(s') \times x_{s'}$;
19:     end for
20: while $\sigma$ has changed or $diff > \epsilon$ ;
```

3.1. Reducing iterations of modified policy iteration

One challenge with MPI is that it is not easy to find a good fixed value for the max_iters parameter. In some cases, non-suitable values for this parameter degrade the performance of the method. We use a dynamic approach for determining the number of iterations for each quotient DTMC. For each adversary σ, we consider the total difference of the values between two subsequent iterations as a criterion for the convergence of the computations of the quotient DTMC. The benefit of this criterion is that it is not limited to the maximal difference of the values (which is the case in a standard policy iteration method). For each state $s \in S^?$ and each adversary σ, we use $x_s^{\sigma,i}$ for the value of s at the i-th iteration after selecting σ. Specially, $x_s^{\sigma,0}$ is the value of s when the method computes σ. We define $\Delta_s^{\sigma,i} = x_s^{\sigma,i} - x_s^{\sigma,i-1}$, and for set $S^?$, we define $\Delta_{S^?}^{\sigma,i} = \sum_{s \in S^?} \Delta_s^{\sigma,i}$ as the total difference of the values at iteration i. We consider $\Delta_{S^?}^{\sigma,i} / \Delta_{S^?}^{\sigma,1}$ as a criterion of the efficiency of adversary σ after the i-th iteration as compared to the first one: if $\Delta_{S^?}^{\sigma,i} / \Delta_{S^?}^{\sigma,1}$ is greater than 0.1, most updates are valuable, and the iterative method should continue with σ. On the other hand, if $\Delta_{S^?}^{\sigma,i} / \Delta_{S^?}^{\sigma,1}$ is less than 0.1, the method should improve the adversary. This criterion should be considered in Line 13 of Algorithm 1. For each $s_i \in S^?$, the method should initialize the value of $\Delta_{S^?}^{\sigma,i}$ before Line 8 and update it after the computation of x_{new} in Line 9. In this heuristic, we consider 0.1 as the threshold for the efficiency of the iterations of a selected adversary σ.

3.2. Avoiding useless updates of each iteration

Depending on the structure of the model, the reachability probability of some states may not change in some iterations of numerical iterative methods. We use a graph-based technique to identify useless updates in MPI. One can use the idea of an SCC-reduction technique for DTMCs [1] to accelerate the iterative computations. This approach decomposes each quotient DTMC into its SCCs and selects them according to their topological order; it then applies iterative methods to update the values of the states of each SCC. However, the overhead of the SCC decomposition for a series of DTMCs is high and affects the overall running time of the iterative method. Alternatively, we develop a heuristic that finds the set of states for each quotient DTMC M^σ that do not belong to any non-trivial SCC (and their values do not affect the value of any SCCs). Formally, we define $In\_trans[s]$ as the number of useful incoming transitions of M^σ to s for each state $s \in S$. We call a transition *useful* if it belongs to an SCC or is between two SCCs. Useful transitions should be used in iterative computations; on the other hand, useless transitions cannot affect the value of any state in some SCCs. A useless transition can be used once at most: after the termination of the iterative computations for M^σ. For any state $s \in S$, if $In\_trans[s]$ is zero, then the value of s does not affect the value of any other state in M^σ, and its outgoing transitions are useless. The algorithm can give up the updates of those states with $In\_trans[s] = 0$ in their iterations. Instead, it updates these states after termination and before the update of the adversary. Initially, we consider all transitions of M^σ as useful and set $In\_trans[s]$ as the number of incoming transitions to s. Next, for each $s \in S$ with $In\_trans[s] = 0$, we consider its outgoing transitions as useless and disregard them in the computation of $In\_trans[s]$ of the other states $s \in S$. Ignoring these states and their outgoing transitions, we may have $In\_trans[s'] = 0$ for some other states $s' \in S$. The computations continue until no other useless transition can be found. We define $S_{In\_trans} = 0$ for the set of states $s \in S^?$ for which $In_trans[s] = 0$ and $S_{In\_trans>0}$ for the set of states where $In\_trans[s] > 0$.

Figure 1 shows a quotient DTMC that has two SCCs: five states before the first SCC, and one state between SCC_1 and SCC_2. We have $In\_trans[s1] = In\_trans[s2] = 0$. Avoiding these two states and their outgoing transitions from the iterative computations, we have $In\_trans[s3] = In\_trans[s4] = 0$ (again avoiding s_3 and s_4 and their outgoing transition results in $In\_trans[s5] = 0$). For the other states, the In_trans values are greater than zero. As a result, $S_{In\_trans=0} = \{s_1, s_2, s_3, s_4, s_5\}$ and MPI can ignore the update of the value of these five states. Instead, it should update their values according to their topological order after terminating the iterations.

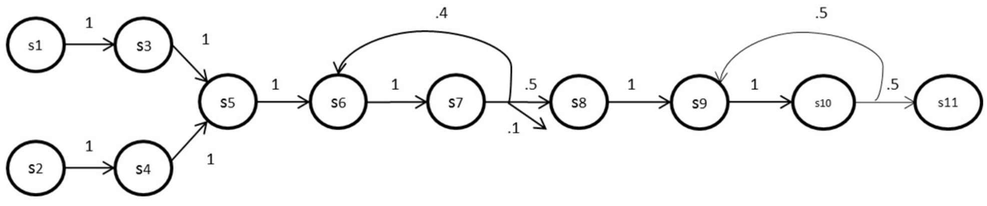

Figure 1. Quotient DTMC with 11 states and 2 SCCs ($SCC_1 = s_6, s_7$, and $SCC_2 = s_9, s_10$)

The set of useless transitions of Figure 1 is $\{(s_1, s_3), (s_2, s_4), (s_3, s_5), (s_4, s_5), (s_5, s_6)\}$. Note that we consider a transition to be useless if it cannot affect the value of any states of some SCCs. Figure 3 shows the details of our graph-based approach for improving MPI. It first computes the $In\_trans[s]$ value of each state $s \in S^?$ and adds those states s for which $In\_trans[s] = 0$ to Q (Lines 6–15). Then, it adds the state to the stack U for each state s in Q. For each state s that is removed from Q (Line 17), we have $In\_trans[s] = 0$, and we are sure that it does not belong to any SCC of M^σ. Because the values of any of these states do not affect the values of the states of any SCC, the algorithm disregards them in the iterative computations. In addition, the algorithm should not consider the outgoing transitions of these states for computing the In_trans value of the other states. To do so, the algorithm updates the $In\_trans[s']$ value of any state $s' \in Post(s, \sigma(s))$ for each state s that is removed from Q (Line 20). It next adds such s' states to Q if the $In\_trans[s_0]$ value is zero, which means that s' is not in any SCC. After this adversary-based pre-computation, the algorithm performs the iterative computations for the rest of the states that are not in U (Lines 26–36). Finally, the algorithm updates the values of the states that are in stack U (Lines 37–41). In this case, the algorithm pops states from U and updates their values. The correctness of Algorithm 2 is shown below.

Lemma 1. For each adversary σ and for each state $s \in S_{In\_trans<0}$, Algorithm 1 and Algorithm 2 approximate the same value for x_s if they perform the same number of iterations in their inner loop (100 iterations, for example) and start from the same vector of values $\bar{x}$.

Proof: Note that the values of the states in $S_{In\_trans>0}$ do not depend on the values of the states in $S_{In\_trans=0}$. For each state $s \in S_{In\_trans=0}$, Algorithm 2 pops the state from U in the correct order and updates its value according to the value of the states that have been previously updated: for each state $s' \in Post(s, \sigma(s))$ either $s' \in S_{In\_trans>0}$, or the algorithm has popped s' from U before popping s. In both cases, the value of $x_{s'}$ was updated. For each $s' \in S_{In\_trans=0}$, the value of x_s depends (directly or indirectly) on the value of some states of $s' \in S_{In\_trans>0}$.

Algorithm 2 uses the values of $s' \in S_{In\_trans>0}$ from the last iteration to update the values of the states in $S_{In\_trans=0}$. As each x_s value never decreases in the iterative methods [4], the values from the last iteration are the best values for updating the value of the states in $S_{In\_trans=0}$. Algorithm 2 uses the optimal topological order [8] for updating the values of the states in $S_{In\_trans=0}$.

Algorithm 2 Graph-based improved modified policy iteration for $P_s^{Max}(F)$.

```
    input: MDP $M = (S, s_0, Act, \delta)$ , target set $F \subseteq S$, stopping threshold $\epsilon$ and $max\_iters$.
    output: Approximation of $P_s^{max}(F)$ for all $s \in S^?$
 1: for all $s \in S$ do
 2:     $x_s \leftarrow \begin{cases} 1, & \text{if } s \in S^y \\ 0, & \text{otherwise} \end{cases}$
 3: end for
 4: Select arbitrary adversary $\sigma$.
 5: do
 6:     $U \leftarrow \phi$;            // Set stack $U$ as empty.
 7:     for all $s_i \in S$ do
 8:         $In\_trans[s_i] \leftarrow 0$;
 9:     end for
10:     for all transition $(s, \sigma(s), s', p)$ do
11:         if $s \in S^?$ then
12:             $In\_trans[s'] \leftarrow In\_trans[s'] - 1$;
13:         end if
14:     end for
15:     $Q \leftarrow \{s \in S^? | In\_trans[s] = 0\}$;
16:     while $Q$ is not empty do
17:         Remove state $s$ from $Q$;
18:         Push $(U, s)$;
19:         for all $s' \in Post(s, \sigma(s)) \cap S^?$ do
20:             $In\_trans[s'] \leftarrow In\_trans[s'] - 1$;
21:             if $In\_trans[s'] = 0$ then
22:                 Add $s'$ to $Q$;
23:             end if
24:         end for
25:     end while
26:     for i = 1 to max_iters do
27:         diif $\leftarrow 0$;
28:         for all $s' \in S^?$ where $s$ is not in $U$ do
29:             $x_{new} \leftarrow \sum_{s' \in S} \delta(s, \sigma(s))(s') \times x_{s'}$;
30:             $diff \leftarrow max(diff, (x_{new} - x_s)/x_s)$;
31:             $x_s \leftarrow x_{new}$;
32:         end for
33:         if diff $\leq \epsilon$ then
34:             break;
35:         end if
36:     end for
37:     while $U$ is not empty do
38:         $s \leftarrow top(U)$;
39:         $x_s \leftarrow \sum_{s' \in S} \delta(s, \sigma(s))(s') \times x_{s'}$;
40:         $Pop(U)$;
41:     end while
42:     for all $s \in S$ do
43:         $\sigma(s) \leftarrow argmax_{a \in Act(s)} \sum_{s' \in S} \delta(s, a)(s') \times x_{s'}$;
44:     end for
45: while $\sigma$ has changed or $diff > \epsilon$;
46: return $(x_s)_{s \in S}$;
```

As a result, the value of x_s after performing max_iters iterations of Algorithm 2 is greater than or equal to the case of performing max_iters iterations of Algorithm 1 for each $s \in S^?$ (if both algorithms use the same adversary σ) ■

In general, the inner loop of Algorithm 1 may perform more iterations than the inner loop of Algorithm 2 for a fixed adversary σ and a fixed vector of values $\vec{x}$. Algorithm 1 considers all of the states of $S^?$. Some states that are pushed to U and disregarded in Algorithm 2 may postpone the satisfaction of the convergence criterion of Algorithm 1. This can increase the precision of the computations of Algorithm 1 as compared to Algorithm 2. In practice, the precision of the computed values of Algorithm 2 is usually more than that of the computed values of Algorithm 1. The running time of the adversary-based pre-computation (Lines 6–25 of Algorithm 2) is linear at a size of M, and its overhead is negligible when compared to 100 iterations of Gauss-Seidel for the quotient DTMCs (Lines 6–16 of Algorithm 1). Note that one can use our heuristics with the SCC-based method of [9]. In this case, the MDP will be decomposed into its SCCs, and our heuristics for MPI can be used to compute the reachability probabilities of the states of each SCC.

4. Experimental results

We used several case studies to compare the performances of the standard iterative methods with our proposed heuristics. These case studies were selected from the PRISM benchmark suite and have been used in previous works [5–7, 11, 16, 17]. In addition, we generated several artificial case studies in order to compare the performances of our methods with the previous ones. To the best of our knowledge, PRISM is the only tool that supports MPI for probabilistic model checking. We used $max\_iters = 100$ as the default value, as it is used in the explicit engine of PRISM. Table 1 describes the models by including their names, parameters, total numbers of states, numbers of states in $|S^?|$, total numbers of actions, and numbers of transitions. We also present the running times and the numbers of iterations for the Gauss-Seidel value iteration (GS), MPI with $max\_iters = 100$, and MPI with our proposed dynamic method (D-MPI). All of the times are given in seconds. We excluded the running time of the pre-computations, as these are negligible in good implementations [6]. The best running times are indicated in bold. For the Consensus and *csma* case studies, we used symmetric reduction method [14] to reduce the numbers of the states of the models. The running times for constructing the models and the running times of the pre-computations for detecting the S^y and S^n sets are proposed in Table 2. PRISM uses BDD-based methods to construct models and perform pre-computations [16]. These approaches are relatively time-consuming; however, more efficient approaches are proposed in [6, 7, 21] that can reduce these running times by several orders of magnitude. We implemented our heuristics for MPI in PRISM and used its sparse engine for our implementations (which was developed in C++). We used a machine with 2.8 GHz Core i7 processor and 8GB of RAM for running our case studies. Our implementation is available at GitHub (https://github.com/mohagheghivru/PRISM_ImprovingPI).

Table 1
Information on case studies (K $= 10^3$)

Model name	Parameters	$\lvert S\rvert$	$\lvert S^?\rvert$	$\lvert Act\rvert$	$\lvert Trans\rvert$	Running time			Iterations		
						GS	MPI	D-MPI	GS	MPI	D-MPI
	N = 6, K = 15	274.5K	186K	1.1M	1.3M	293	175	**174**	75K	87K	87K
	N = 6, K = 30	554.5K	372K	2.2M	2.7M	1372	743	**730**	151K	175K	175K
cons	N = 6, K = 45	814.5K	558K	3.2M	4.1M	5137	3405	**3387**	465K	582K	581K
	N = 8, K = 5	399.5K	225K	2.1M	2.6M	127	59	**58**	18K	20K	20K
	N = 8, K = 10	778.3K	450.5K	4.15M	5.2M	579	307	**304**	42K	45K	45K
	N = 8, K = 15	1.157K	676K	6.2M	7.7M	2238	1189	**1186**	121K	143K	143K
csma	N = 3, K = 6	14.2M	3.9M	14.2M	24.7M	**15.1**	35.8	21.7	112	357	154
	N = 4, K = 4	5.87M	5.2M	5.98M	11.4M	**7.15**	17.3	7.73	126	291	135
	N = 5, ttm = 1500	3.6M	36.2K	6.3M	7.6M	0.83	2.37	**0.78**	102	342	136
wlan	N = 5, ttm = 5000	9.1M	150K	17.4M	18.8M	2.3	6.74	**1.97**	104	347	139
	N = 6, ttm = 1000	8.1M	57K	12.5M	17.7M	**.57**	2.27	.64	51	225	101
	N = 6, ttm = 2500	12.8M	36.9K	21.9M	27M	**1.35**	5.44	1.57	53	228	106
wlan_	ttm = 500, col = 5	6.5M	24.4K	9.4M	14.5M	2.35	5.9	**2.11**	182	475	218
collide	ttm = 1250, col = 6	8.8M	65.6K	14.1M	19.2M	5.95	12	**5.23**	324	778	359
	ttm = 3000, col = 6	14.3M	127K	25M	30.2M	10.1	20.7	**8.6**	327	782	362
	dl = 36, ddl = 800	530K	198K	803K	954K	2.96	1.08	**1.07**	614	615	615
	dl = 36, ddl = 2500	1.8M	994K	2.8M	3.35M	31.5	16.5	**16.4**	2326	2327	2318
firewire	dl = 108, ddl = 1500	1.6M	903K	3.1M	4M	14.4	8.8	**8.6**	1330	1331	1322
(abst)	dl = 108, ddl = 2500	2.7M	1.7M	5.4M	6.9M	63.6	28.9	**28.7**	2339	2332	2320
	dl = 108, ddl = 5000	5.6M	3.7M	10.9M	14.1M	246	**125**	126	4834	4835	4819
	dl = 165, ddl = 5000	6.9M	4.9M	14.7M	19.5M	342	159	**158**	4863	4864	4832
	K = 14, N = 20	4.4M	3.1M	8.2M	10M	14.7	26.7	**6.2**	194	1142	230
	K = 14, N = 2000	4,4M	3.1M	8.2M	10M	20.4	49.8	**12.6**	320	1591	490
	K = 16, N = 20	5M	3.6M	9.2M	11.3M	16.3	31.4	**8.1**	199	1187	261
zero-	K = 16, N = 2000	5M	3.6M	9.2M	11.3M	19	50.9	**12.6**	326	1644	510
conf	K = 18, N = 20	5.5M	4.1M	10.1M	12.4M	23.5	38.1	**10.6**	207	1270	295
	K = 18, N = 2000	5.5M	4.1M	10.1M	12.4M	30.7	52	**15.9**	332	1702	515

As compared to GS, MPI increased the number of iterations because it may have used some non-optimal policies. On the other hand, the average number of computations per state was reduced in MPI. As presented in Table 1, MPI outperformed GS for most of the cases, and GS outperformed all of the others. An interesting result of our experiments is that, for the $csma, wlan, wlan\_col$ and $zeroconf$ case studies, we had a considerable improvement in the running times of MPI when we used our dynamic method for the numbers of iterations of each quotient DTMC. For these classes of case studies, MPI had a late convergence to the optimal policy, which increased the overall running time of this method as compared to the GS method. For these cases, our dynamic approach for computing the number of iterations for each policy reduced the overall number of iterations and, as a result, proposed a considerable improvement in the iterative computations. On the other side, MPI had a fast convergence to the optimal policies for the $cons$ and $firewire$ models, and because of their average numbers of actions per state, MPI was much faster than GS for these classes of models. We compared the running time of our improved method for MPI (Algorithm 2) and the running times of prominent previous works; these results can be found in Table 3. We considered a learning-based method from [6] and an SCC-based method from [16]. For the computing reachability probabilities of the states of each SCC, we used MPI with $max\_iters = 100$.

Table 2

Running times of pre-computation, model construction, and MPI with different max_iters

Model name	Const-ruction	S^n	S^y	MPI $max\_iters = 20$	MPI $max\_iters = 50$	MPI $max\_iters = 500$	Non-optimal Actions
consensus(6,15)	0.5	2.1	471.5	192.2	180.2	170.5	3.50%
consensus(6,30)	1.1	2.4	1258	852	797	721	3.62%
consensus(6,45)	1.4	4.9	2383.3	3620	3422	3368	3.80%
consensus(8,5)	0.7	3.58	1953	65.8	63.2	56.9	3.20%
consensus(8,10)	1.1	6.3	2878	348	323	297	3.42%
consensus(8,15)	1.7	9.4	4312	1227	1213	1172	3.60%
csma(3,6)	116	225	7634	35.5	31.7	45.3	1.10%
csma(4,4)	73	154	1333	17.5	15.5	20.3	1.20%
wlan(5,1500)	127	32.43	92.43	2.21	1.97	2.45	9.40%
wlan(5,5000)	997	128	274	5.8	5.41	6.85	9.21%
wlan(6,1000)	178.6	72.6	223.6	5.18	4.78	5.58	11.50%
wlan(6,2500)	690	149	353	4.89	5.44	5.62	10.71%
wlan_col(500)	145	45.7	123.6	5.49	5.15	6.44	9.70%
wlan_col(1250)	192	77.9	352.7	11.74	10.31	14.12	9.40%
wlan_col(3000)	280	112	471	11.74	10.31	14.12	9.40%
firewire(36,800)	3.9	.9	1.5	1.1	1.08	1.06	38.3%
firewire(36,2500)	31.7	1.6	2.1	17.3	16.8	16.3	38%
firewire(108,1500)	11.7	1.5	1.2	8.98	8.84	8.4	37.7%
firewire(108,2500)	31.2	1.5	1.8	29.5	29.1	28.6	38.5%
firewire(108,5000)	125	1.6	3.2	131	128	122	38.1%
firewire(165,5000)	132	2.3	3.4	170	164	155	38%
zeroconf(14,20)	175	7.5	154	19.64	23.93	33.74	33.70%
zeroconf(14,2000)	263	9.9	175.8	35.31	42.72	61.82	35.20%
zeroconf(14,20)	261	9.8	175.3	30.74	34.19	44.56	41.20%
zeroconf(18,2000)	321	11.3	194.3	37.3	42.5	57.72	43.60%

Table 3 lists the running times and some related information on the iterative methods. We recalled the running time of the best iterative method from Table 1. For the SCC-based method, the running times include the times for SCC decomposition and the running times of the iterative methods for computing the reachability probabilities. For our improved method (called improved MPI in Table 3), we considered two alternatives. First, we present the running time of Algorithm 2 for MPI in the $max\_iters = 100$ column. Next, we consider Algorithm 2 with a dynamic approach for the number of iterations for each quotient DTMC and propose the running time in the 'D-MPI' column. For the Consensus models, we also used the SCC-decomposition approach and applied our improved method for computing the reachability probabilities of each SCC. In the last column, we show the average number of useful states per the states in $S^?$.

Table 3
Running times of proposed improved methods

Model name	Best Standard Method	Learning--based Method	SCC--based Method	Improved MPI		
				max_iters =100	D-MPI	$\frac{usefulstates}{\lvert S^?\rvert}$
consensus(6,15)	174.7	579.2	57.16	22.4	22.3	22.60%
consensus(6,30)	730	1952	255	107	106	21.17%
consensus(6,45)	3405	7842	1149	524.4	522.7	19.40%
consensus(8,5)	59.5	212.9	17.7	7.81	7.78	23.70%
consensus(8,10)	304	926	121	53	51	22.69%
consensus(8,15)	1189	4370	327.7	142.5	141.8	21.50%
csma (3,6)	15.12	27.91	14.1	0.72	0.72	< 1%
csma (4,4)	7.15	18.34	39.4	0.46	0.46	< 1%
wlan(5,1500)	0.83	0.23	1.54	0.13	0.11	< 1%
wlan(5,5000)	2.3	0.35	3.11	0.3	0.28	< 1%
wlan(6, 1000)	.57	0.28	4.42	< 0.1	< 0.1	< 1%
wlan(6, 2500)	1.35	0.46	7.19	< 0.1	< 0.1	< 1%
wlan_Col(500)	2.35	0.52	3.25	0.23	0.19	< 1%
wlan_Col(1250)	5.95	0.67	4.52	0.32	0.27	< 1%
wlan_Col(3000)	10.1	0.79	6.7	0.4	0.37	< 1%
firewire(36,800)	2.96	2.78	.25	< .1	< .1	< 1%
firewire(36,2500)	31.5	2.78	.46	0.16	0.15	< 1%
firewire(108,1500)	14.4	2.78	.35	0.15	0.15	< 1%
firewire(108,2500)	63.6	2.78	.48	0.28	0.27	< 1%
firewire(108,5000)	246	2.78	.84	0.85	0.84	< 1%
firewire(165,5000)	342	2.78	.85	0.98	0.96	< 1%
zeroconf(14,20)	14.67	4.57	31.7	8	6.1	1.93%
zeroconf(14,2000)	20.43	37.24	43.9	8.71	6.7	1.83%
zeroconf(16,20)	16.3	7.53	42.1	9.1	7.4	1.59%
zeroconf(16,2000)	19	38.1	55.7	9.43	7.8	1.54%
zeroconf(18,20)	23.53	11.92	52.2	11.63	9.07	1.37%
zeroconf(18,2000)	30.66	37.9	67.5	11.67	8.95	1.39%

In general, the running time of each method depends on the structures of the models. Each iterative method performs several updates on the value of each state in $S^?$. As a result, the number of states in $S^?$ affects the performance of each method. In addition, the number of actions for each state and the number of outgoing transitions are important in the computations. For example, GS considers all of the actions of each state to update its value, while MPI considers the best action in most iterations. According to Table 1, the average number of actions per state is close to one for the *csma* models. As a result, the difference between GS and MPI is negligible for this case. On the other hand, each state had more than three actions on average for the *consensus* models, which affected the running times of GS. In addition, the graphical structure of a model can affect the performance of the iterative methods. Our second proposed method uses the information of each quotient DTMC to avoid useless updates.

For all of the samples, our improved method outperformed all of the standard iterative methods. The last column of Table 3 shows that our method identified a large portion of the states of each quotient DTMC as useless states. In this case, a few numbers of states should have been updated in each iteration, which improved the performance of MPI. The SCC-based method was more useful for the *consensus* samples. The overhead of the SCC decomposition was high for the other cases, and their overall running times were more than the running times of our improved method. Our experiments showed that the learning-based method was more useful for *wlan* and some of the *zeroconf* samples, but it was not promising for the other ones. To compare the first proposed method with MPI, we also considered the running time of MPI for three other values of the *max_iters*; these results are reported in Columns 5–7 of Table 2. The eighth column of Table 2 shows the average number of non-optimal actions per state during the policy modifications when we applied MPI with *max_iters* = 100. To compute this value, we considered the optimal policy (after convergence) and repeated the experiments again to compare the computed policies with the optimal one. According to these results, the performance of MPI was slightly better for the Consensus models when we considered higher values for the *max_iters*. For this class of models, the method converged to the optimal policy after a few policy modifications, and MPI did not need to consider the non-optimal actions in the other iterations. On the other hand, the method showed the slow convergence to an optimal policy for the *zeroconf* models, and a large number of computations were useless due to the non-optimal policies. In this case, the lower values for the *max_iters* caused a faster convergence to the optimal policy. In general, it is not easy to determine a good value for the *max_iters*. We considered $\Delta^{\sigma,i}_{S?}/\Delta^{\sigma,1}_{S?}$ to be a criterion in our first method to determine the usefulness of each policy. We considered 0.1 to be the threshold to decide on improving the policies. To study the impact of this threshold on the performance of the proposed method, we considered different values for this parameter.

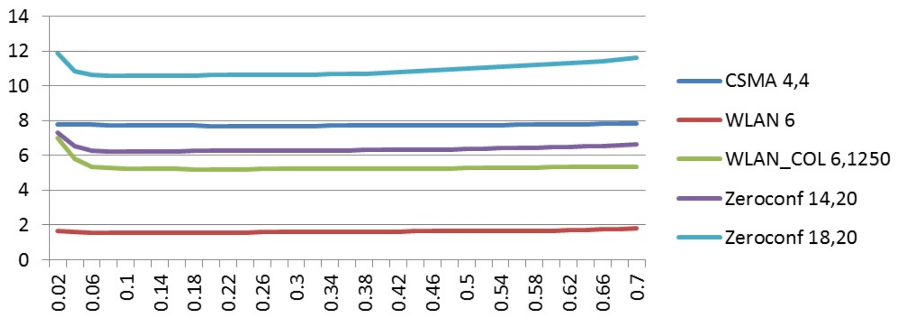

Figure 2. Impact of various thresholds on performance of D-MPI

Figure 2 proposes the running time of D-MPI with different values of this threshold. In most cases, values between 0.06 and 0.3 are good choices for this threshold. Lower values cause more iterations for each policy, which increases the number of useless computations. On the other hand, higher values cause more policy modifications and the increased use of non-optimal actions. However, the results showed that the difference of the performance of the method is negligible when the threshold is set at around 0.1.

In addition to the standard case studies, we considered eight classes of simulated models in order to compare the performances of our methods with others. Table 4 shows some details of each class (each containing ten models), and we present the average running times of the models. All of the models had 10,000 states. The difference of D1–D4 was in the number of actions per state, while there were two outgoing transitions for each action. The destination of the transitions and their probabilities were determined randomly. When the number of actions per state increased, MPI performed better than the GS and SCC-based methods. In this case, MPI avoided more actions in each update. Our D-MPI method outperformed the other methods in all cases because it converged to the optimal policy faster than MPI and reduced the number of iterations. For D5–D8, each state has two actions, but the average numbers of outgoing transitions were different. Again, D-MPI was better than the other methods. In this case, the performance of the learning-based method degraded when the average number of outgoing transitions increased.

Table 4
Running times of simulated models

Model	$\|Act\|$	$\|Trans\|$	GS	MPI	Learning based	SCC based	D-MPI	Improved MPI
D1	2	40K	11.4	7.3	10.8	11.4	5.78	5.39
D2	4	80K	24.5	8.45	25.4	24.7	6.22	5.67
D3	7	140K	45.7	9.83	41.9	46.2	7.2	6.4
D4	10	200K	63.7	11.2	57.4	64	9.5	8.31
D5	2	60K	16.7	11.4	12.5	16.8	9.6	8.46
D6	2	90K	25.7	15.7	19.5	25.9	13.3	11.7
D7	2	160K	47.9	28.1	36.6	48	25.2	22.2
D8	2	300K	98.3	57.4	71.6	98.6	53.1	49.8

5. Conclusion

Modified policy iteration is an iterative method that is widely used in probabilistic model checking to compute maximum or minimum reachability probabilities. The work presented in this paper has established a way to accelerate the MPI method.

To do this, we proposed two new heuristics in Section 3. For the first heuristic, we defined a criterion for determining the usefulness of a policy. This relies on the impact of the current policy on the values of the underlying states of a model. This criterion is more efficient than the standard criterion in MPI, and the experimental results show that it usually reduced the overall number of iterations. The second proposed heuristic performed a graphical analysis of the quotient DTMCs to identify and avoid any useless updates of the states. This considered the number of incoming transitions to a state to determine any useless states. Our experimental results showed that these two techniques outperformed the best previous methods for most case studies. For our future works, we plan to develop a parallel version of our heuristics and apply them for accelerating the interval iteration method for computing the reward-based properties that were proposed in [5]. In addition, one can extend the second heuristic by considering the probability of the incoming transitions to prioritize the states.

References

[1] Ábrahám E., Jansen N., Wimmer R., Katoen J.P., Becker B.: DTMC model checking by SCC reduction. In: *2010 Seventh International Conference on the Quantitative Evaluation of Systems*, pp. 37–46, IEEE, 2010.

[2] Ashok P., Křetínský J., Weininger M.: PAC statistical model checking for Markov decision processes and stochastic games. In: *International Conference on Computer Aided Verification*, pp. 497–519, Springer, 2019.

[3] Baier C., Hermanns H., Katoen J.P.: The 10,000 facets of MDP model checking. In: *Computing and Software Science*, pp. 420–451, Springer, 2019.

[4] Baier C., Katoen J.P.: *Principles of model checking*, MIT Press, 2008.

[5] Baier C., Klein J., Leuschner L., Parker D., Wunderlich S.: Ensuring the reliability of your model checker: Interval iteration for Markov decision processes. In: *International Conference on Computer Aided Verification*, pp. 160–180, Springer, 2017.

[6] Brázdil T., Chatterjee K., Chmelik M., Forejt V., Křetínský J., Kwiatkowska M., Parker D., Ujma M.: Verification of Markov decision processes using learning algorithms. In: *International Symposium on Automated Technology for Verification and Analysis*, pp. 98–114, Springer, 2014.

[7] Dehnert C., Junges S., Katoen J.P., Volk M.: A storm is coming: A modern probabilistic model checker. In: *International Conference on Computer Aided Verification*, pp. 592–600, Springer, 2017.

[8] Feng L., Kwiatkowska M., Parker D.: Compositional verification of probabilistic systems using learning. In: *2010 Seventh International Conference on the Quantitative Evaluation of Systems*, pp. 133–142, IEEE, 2010.

[9] Forejt V., Kwiatkowska M., Norman G., Parker D.: Automated verification techniques for probabilistic systems. In: *International school on formal methods for the design of computer, communication and software systems*, pp. 53–113, Springer, 2011.

[10] Gros T.P., Hermanns H., Hoffmann J., Klauck M., Steinmetz M.: Deep statistical model checking. In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pp. 96–114, Springer, 2020.

[11] Hartmanns A.: *On the Analysis of Stochastic Timed Systems*, Ph.D. thesis, Saarland University, 2015.

[12] Katoen J.P.: The probabilistic model checking landscape. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pp. 31–45, 2016.

[13] Klein J., Baier C., Chrszon P., Daum M., Dubslaff C., Klüppelholz S., Märcker S., Müller D.: Advances in probabilistic model checking with PRISM: variable reordering, quantiles and weak deterministic Büchi automata, *International Journal on Software Tools for Technology Transfer*, vol. 20(2), pp. 179–194, 2018.

[14] Kwiatkowska M., Norman G., Parker D.: Symmetry reduction for probabilistic model checking. In: *International Conference on Computer Aided Verification*, pp. 234–248, Springer, 2006.

[15] Kwiatkowska M., Norman G., Parker D.: PRISM 4.0: Verification of probabilistic real-time systems. In: *International Conference on Computer Aided Verification*, pp. 585–591, Springer, 2011.

[16] Kwiatkowska M., Parker D., Qu H.: Incremental quantitative verification for Markov decision processes. In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pp. 359–370, IEEE, 2011.

[17] Mohagheghi M., Karimpour J., Isazadeh A.: Prioritizing methods to accelerate probabilistic model checking of discrete-time Markov models, *The Computer Journal*, vol. 63(1), pp. 105–122, 2020.

[18] Parker D.A.: *Implementation of symbolic model checking for probabilistic systems*, Ph.D. thesis, University of Birmingham, 2003.

[19] Puterman M.L.: *Markov decision processes: discrete stochastic dynamic programming*, John Wiley & Sons, 2014.

[20] Rataj A., Woźna-Szcześniak B.: Extrapolation of an Optimal Policy using Statistical Probabilistic Model Checking, *Fundamenta Informaticae*, vol. 157(4), pp. 443–461, 2018.

[21] Ujma M.: *On verification and controller synthesis for probabilistic systems at runtime*, Ph.D. thesis, Citeseer, 2015.

Affiliations

Mohammadsadegh Mohagheghi
University of Tabriz, Department of Computer Science, Tabriz, Iran,
m_mohagheghi@tabrizu.ac.ir, ORCID ID: https://orcid.org/0000-0001-8059-3691

Jaber Karimpour
University of Tabriz, Department of Computer Science, Tabriz, Iran, karimpour@tabrizu.ac.ir

Ayaz Isazadeh
University of Tabriz, Department of Computer Science, Tabriz, Iran, isazadeh@tabrizu.ac.ir

Received: 9.03.2021
Revised: 14.06.2021
Accepted: 13.09.2021