Improved Iterative Methods for Verifying Markov Decision Processes

Jaber Karimpour, Ayaz Isazadeh, Mohammadsadegh Mohagheghi, Khayyam Salehi

Improved Iterative Methods for Verifying Markov Decision Processes

Jaber Karimpour, Ayaz Isazadeh, MohammadSadegh Mohagheghi, and Khayyam Salehi

Department of Computer Science, University of Tabriz
karimpour@tabrizu.ac.ir, isazadeh@tabrizu.ac.ir, sadegh_rk@yahoo.com, khayyam.salehi@gmail.com

Abstract. Value and policy iteration are powerful methods for verifying quantitative properties of Markov Decision Processes (MDPs). In order to accelerate these methods many approaches have been proposed. The performance of these methods depends on the graphical structure of MDPs. Experimental results show that they don't work much better than normal value/policy iteration when the graph of the MDP is dense. In this paper we present an algorithm which tries to reduce the number of updates in dense MDPs. In this algorithm, instead of saving unnecessary updates we use graph partitioning method to have more important updates.

Keywords: Markov decision processes, probabilistic model checking, value iteration, policy iteration, graph partitioning, variable ordering

1 Introduction

Markov Decision Processes (MDPs) are transition systems that can be used for modeling both nondeterministic and stochastic behaviors of reactive systems. In this paper we mainly focus on the quantitative verification of MDPs and consider reachability probabilities, i.e., calculating the maximum (or minimum) probability of reaching some goal states.

In general there are some main classic methods to solve MDPs: value iteration [1], Gauss-Seidel, policy iteration, and linear programming approach [6]. Many researchers have tried to improve the performance of Value Iteration (VI) and Policy Iteration (PI) by reducing the number of updates of states [2, 4, 7–9]. Although the main focus of so-called papers is on dealing with some problems in learning, one can use those techniques for quantitative verification of MDPs [3,5].

Experimental results show that when the graph of an MDP is dense or in the situation where all states belong to only one Strongly Connected Component (SCC), Gauss-Seidel version of VI works better than other advanced methods. In addition, a good variable ordering can accelerate iterative methods, but finding the optimal variable ordering for cyclic MDPs is an NP-complete problem [9].

In this paper we concentrate on dense MDPs and consider maximum reachability probability problems (as defined in [1]). We present an algorithm that

works faster than VI (and also faster than other methods). The main contribution is to present a prioritized algorithm for accelerating verification of dense MDPs.

The remainder of this paper is structured as follows. Section 2 formally defines MDPs and reviews PI algorithm. In Section 3 we present heuristics for variable reordering in both sparse and dense MDPs and then present a prioritized algorithm for dense MDPs. Section 4 summarises our experimental results and Section 5 presents conclusions and ideas for future research.

2 Preliminaries

In this section, we provide an overview of MDP and policy iteration method. Detailed definitions and topics are available in [1, 5].

2.1 Markov Decision Processes (MDPs)

An MDP is a tuple $M = (S, Act, P, AP, L)$, where S is a countable set of states, Act is a finite set of actions, $P : S \times Act \times S \to [0, 1]$ is the transition probability function such that for each state $s \in S$ and each action $\alpha \in Act : \sum_{s' \in S} P(s, \alpha, s') \in \{0, 1\}$, AP is a nonempty set of atomic propositions, and $L : S \to 2^{AP}$ is a labeling function.

For the sake of simplicity, we suppose that $AP = \{goal, non - goal\}$ and there is a unique start state s_0. For a state $s \in S$ and an enabled action $\alpha \in Act$, set of successors are defined as

$$Post(s, \alpha) = \{s' \in S | P(s, \alpha, s') > 0\} \text{ and } Post(s) = \cup_{\alpha \in Act} Post(s, \alpha).$$

A path in an MDP is a non-empty (finite or infinite) sequence of the form: $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \cdots$ where $s_i \in S$ and $s_{i+1} \in Post(s_i, \alpha)$ for each $i > 0$. We define $Paths_s$ as the set of infinite paths that start in s. To resolve nondeterministic choices in an MDP we require the notion of policy. A policy π selects an enabled action in each state based on the history of choices made so far (or simply the last state in memory-less policies). It restricts the behavior of the MDP to a set of paths $paths_s^\pi \subseteq paths_s$. One can define a probability space $Prob_s^\pi$ over the paths $Paths_s^\pi$ [1]. For an MDP M we use Π_M to denote the set of all policied of M.

MDPs can be used in the verification of systems (probabilistic verification). In this area, properties that should be verified against MDPs can be expressed using temporal logics such as PCTL [5]. In this paper, we concentrate on a limited yet important class of problems: maximum reachability probabilities, i.e. the maximum probability that a path through the MDP which starts from s_0 eventually reaches a goal state.

2.2 Quantitative Verification of MDPs

Model checking of PCTL formulas can be reduced to some important questions against MDPs. The maximum (or minimum) reachability probability is one of the most important questions against them. Given a set of goal states $G \subseteq S$, the maximum and minimum reachability probability can be defined as:

$$p_s^{min}(G) = \inf_{\pi \in \Pi_M} p_s^{\pi}(G) \text{ and } p_s^{max}(G) = \sup_{\pi \in \Pi_M} p_s^{\pi}(G),$$

where $p_s^{\pi}(G) = Prob_s^{\pi}(\{\omega \in Path_s^{\pi} | \exists i.\omega_i \in G\})$. As we have mentioned, there are some methods to compute reachability probabilities in MDPs but we only consider PI (because PI has better performance for dense MDPs).

2.3 Policy Iteration

Algorithm 1 describes the policy iteration method to calculate the values of p_s^{max}. This algorithm uses an array P to save the value $p_{s_i}^{max}$ for each state s_i. It first estimates a good policy (Line 8) and computes the value of states iteratively according to this policy (Lines 15-25). After reaching the threshold if the estimated policy is not optimal it will be improved and the iterative method continues. $Act[s_i]$ saves the best estimated action for each state s_i.

Algorithm1: *Policy Iteration*

1. Set initial values: $P[s_i] = 1$ if $\{s_i \in G\}$ and o.w. $P[s_i] = 0$.
2. diff1 := 1
3. while diff1 > epsilon do
4. diff1 := 0; temp := 0; diff2 := 1
5. for i := 1 to number_of_states do
6. if $P[s_i] < 1$ then
7. temp := $\max_{\alpha \in enabled(s_i)} \sum_{s' \in S} p(s_i, \alpha, s') \times P[s']$
8. $Act[s_i]$:= $\arg\max_{\alpha \in enabled(s_i)} \sum_{s' \in S} p(s_i, \alpha, s') \times P[s']$
9. if temp $-$ $P[s_i]$ > diff1 then
10. diff1 := temp $-$ $P[s_i]$ end if
11. end if
12. end for
13. if diff1 < epsilon then
14. return $P[s_1]$ end if
15. while diff2 > epsilon do
16. diff2 := 0
17. temp := 0
18. for i:= 1 to number of states do
19. if $P[s_i] < 1$ then
20. $temp := \sum_{s' \in S} p(s_i, Act[s_i], s') \times P[s']$
21. if temp $-$ $P[s_i]$ > diff2 then

```
22.                         diff2 := temp − P[s_i] end if
23.              end if
24.         end for
25.      end while
26. end while
```

3 Accelerating VI and PI algorithms

One of the main drawbacks of VI (and also PI) is that in every iteration it updates the values of all (nontrivial) states. Researchers proposed a range of methods to avoid unnecessary updates and speed up this algorithm. Some of these methods [4, 9] try to split the MDP to some SCCs and in every iteration only compute the new value for states in some SCCs. Another approach for accelerating iterative methods is to consider a better variable ordering [4, 5, 9].

We review the idea of variable reordering and propose a heuristic algorithm for variable reordering in dense MDPs.

3.1 Variable ordering

The PI algorithm in Section 2 tries to use the update of states as soon as possible. In this case the order of updated states can influence the performance of algorithm. Let $StatesOrder[s_i]$ be the array that determines the (static) order of states for update. When the value of a state is updated the new value can be propagated to next computation if the values of next states depend on the value of the current state. There are some heuristics for variable ordering in previous works [3–5, 9], but none of them are useful for dense MDPs.

Here we propose an algorithm for variable reordering whose time complexity is linear in the size of the MDP. The idea of this algorithm is to select a state s for update where the most of states of $Post(s)$ have been updated before. The *for* loop in line 5 is used to guarantee the selection of all states.

Algorithm2: *VariableReordering*

```
1. for i := 0 to n - 1
2.     Selected[i] := false;
3. end for
4. left := right := 0;
5. for i := 0 to n - 1
6.     if Selected[i] = false then
7.         Selected[i] := true; StateOrder[left] := i;
8.         while left <= right and right < n do
9.              j := StateOrder[left++];
10.             for each s_k ∈ Post(s_j)
11.                 if Selected[k] = false then
12.                     Selected[k] := true;
```

```
13.                         StateOrder[right++] = k;
14.                     end if
15.                 end for
16.             end while
17.         end if
18.     end for
```

Policy iteration algorithm should call this function before the beginning of inner *while* loop (line 15 of Algorithm 1). In this case the VariableReordering function uses $Post(s_j, Act[s_j])$ (line 10 of algorithm 2). The *for* loop of line 18 (of algorithm 1) should be in reverse order and the state s_i could be selected according to the *StateOrder* array:

```
for i := number_of_states down to 1
```
$s_i = StateOrder[i]$;

This algorithm is useful for sparse MDPs but in order to deal with dense MDPs we modify the Post set in line 10. Given $t < 1$ as a threshold, we define $Post(s_i, \alpha, t) = \{s_j | P(s_i, \alpha, s_j) \geq t\}$. The experimental results show that the good value for t is between 0.3 and 0.5. It causes the algorithm to consider more important transitions. Algorithm 2 is used as a precomputation for every policy modification and doesn't affect the correctness of the PI algorithm.

3.2 Prioritized Algorithm for PI

Prioritized algorithms [9] in general try to focus on regions of the problem space that are more important and have the maximum effect on the whole problem. In this section we propose an algorithm that uses a simple graph partitioning method for prioritizing state updates.

Inspiring from the idea of SCC-based methods [5] we propose an algorithm which uses a good heuristic to split the state space to some partitions and updates these states according to this partitioning. Let B be a partition. Define $AverageTrans(s_i, B) = \frac{\sum_{S_j \in B} P(s_i, Act(s_i), s_j)}{sizeof(B)}$. Our method tries to make partitions where for each state s_i and each partition B the value of $AverageTrans(s_i, B)$ is high if $s_i \in B$ and this value is low otherwise. In this case the partition that contains goal states is the most important and the frequency of updates for states of this partition should be more than the other two. We use this partitioning algorithm in the PI because for dense MDPs the performance of PI is usually better than the performance of VI.

We use an $O(n^2)$ heuristic for graph partitioning of the MDP because its overhead in the case of dense MDPs is negligible for most case studies. There are some options for the size of partitions. In this paper we define 3 partitions and suppose that the size of the first partition is 25% and the second partition is 35% of all state space. Algorithm 3 describes this partitioning method and because of page limitation, we only present the code for the first partition (with 25% of states).

Algorithm3: *Graph Partitioning*

```
1.  for i := 1 to number_of_states do
2.     Distance[i] := 0;
3.  end for
4.  for i := 1 to number_of_states do
5.     if s_i is a goal state then
6.        Selected[i] := true;
7.        for j := 1 to number_of_states do
8.             Distance[j] += P(s_j, Act[s_j], s_i);
9.        end for
10.    end if
11. end for
12. for i := 1 to 0.25×number_of_states do
13.      k := the index of non-selected state for which
           Distance[k] is maximum;
14.      Selected[k] := true;
15.      for j := 1 to number_of_states do
16.         if Selected[j] = false then
17.             Distance[j] += P(s_k, Act[s_k], s_j) + P(s_j, Act[s_j], s_k);
18.         end if
19.      end for
20. end for
```

This algorithm should be called before the second *while* loop (Line 15) in Algorithm 1. In order to use the result of this algorithm we modify Algorithm 1 and add the following statements after Line 18:

```
if i % 7 = 0
     update states of partition #3
else if i % 7 = 2 or i % 7 = 5
        update states of partition #2
     else
        update states of partition #1
     end if
end if
```

4 Experiments

We implemented the proposed and original iterative algorithms in C++ using MS Visual Studio 2010 and ran it on an Intel Core i3 processor with 4GB memory. In the sections that follow we first consider the proposed algorithm of Section 3.1 and then consider the algorithm of Section 3.2.

4.1 Results for the Variable Ordering Algorithm

We tested our modified ordering algorithm on some dense MDPs. These models are randomly generated problems with 100 states and 3 actions per state and a parameter λ. We created these MDPs such that for each state s_i, the average of maximum value of $P(s_i, \alpha, s_{i+1})$ is λ with standard deviation of 0.1. Table 1 shows results of running Policy Iteration with best and worst variable ordering and also a random variable ordering. For simplicity we only present number of iterations for the execution of PI with $\epsilon = 10^{-6}$.

Table 1. Results of Variable Ordering for Some Dense MDPs

λ	0.9	0.7	0.5	0.3	0.2	0.1
Random ordering	11.6K	5.5K	3.7K	2.9K	2.6K	2.5K
Best ordering	1.7K	1.9K	2K	2.1K	2.2K	2.2K
Worst ordering	20.5K	9.2K	5.6K	3.8K	3.2K	2.9K
Best/random	0.147	0.364	0.568	0.758	0.846	0.88

The impact of variable ordering for these case studies is considerable where the λ parameter is more than 30%.

4.2 Results for partitioning-based algorithms

SysAdmin problem [9] is a good example of real problems that have dense MDPs. Because of relatively high number of actions per state, the performance of PI algorithm is better than VI and Gauss-Seidel. We used an implementation for the MDP of this problem that is developed by the authors of [9].

We also defined one other MDP: M1 is a model that the probability of reaching to 20% of its states is about 8 times more than the probability of reaching other 80% of states. This model has 400 states where average number of actions per state is 3.

It has been shown that the performance of SCC-based methods is lower than standard iterative method for dense MDPs [9]. Hence, we don't compare our algorithm with SCC-based methods. To compare it with learning based ones, we use the implementation of learning algorithms that the authors of [3] proposed.

Table 2 shows the results of running normal VI and PI methods and, our prioritized algorithm (PI with algorithm 3). We called our algorithm Prioritized PI (PPI). Because of high average number of actions per state, the running time of PI is less than VI. The results show that our prioritized algorithm reduces the running time for all models. While prioritized methods of [9] could not improve the performance of iterative methods for SysAdmin, our algorithm accelerates iterative computations for these models.

The main reason that the performance of learning based algorithms for dense MDP's is so low is that these methods doesnt usually propose good variable ordering for this class of MDPs.

Table 2. Results for our prioritized algorithm

Model	Number of States	Time in VI	Time in PI	Time in Learning-based	Time in PPI
SysAdmin6	64	< 0.1	< 0.1	2.6	< 0.1
SysAdmin8	256	0.7	0.19	11.5	0.16
SysAdmin9	512	4.5	0.76	24.3	0.63
SysAdmin10	1023	27.2	2.45	49	1.96
M1	410	1.11	0.95	8.5	0.49

5 Conclusions and Future Research

The main contribution of this paper is that we present a prioritized algorithm for dense MDPs and show that it can reduce the running time of iterative methods for solving probabilistic reachability problems of this class of MDPs.

An approach for future works is to use a good variable ordering in each partition in our prioritized algorithm. In addition, one can propose a better prioritized algorithm that outperforms other prioritized ones in both dense and sparse models.

While many of previous works focus on VI, we believe that variable reordering can improve the performance when it is used in PI. One can also improve the performance of PI for both dense and sparse MDPs and outperform the VI approach by using advanced methods for selection of good policies (like action elimination.)

References

1. Baier, C., Katoen, J.P.: Principles of model checking. MIT press Cambridge (2008)
2. Barto, A.G., Bradtke, S.J., Singh, S.P.: Learning to act using real-time dynamic programming. Artificial Intelligence 72(1), 81–138 (1995)
3. Brázdil, T., Chatterjee, K., Chmelík, M., Forejt, V., Křetínskỳ, J., Kwiatkowska, M., Parker, D., Ujma, M.: Verification of markov decision processes using learning algorithms. In: Automated Technology for Verification and Analysis, pp. 98–114. Springer (2014)
4. Dai, P., Mausam, J.G., Weld, D.S., Goldsmith, J.: Topological value iteration algorithms. J. Artif. Intell. Res.(JAIR) 42, 181–209 (2011)
5. Kwiatkowska, M., Parker, D., Qu, H.: Incremental quantitative verification for markov decision processes. In: Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on. pp. 359–370. IEEE (2011)
6. Puterman, M.L.: Markov decision processes: discrete stochastic dynamic programming. John Wiley & Sons (1994)
7. Sanner, S., Goetschalckx, R., Driessens, K., Shani, G.: Bayesian real-time dynamic programming. In: IJCAI. pp. 1784–1789. Citeseer (2009)
8. Smith, T., Simmons, R.: Focused real-time dynamic programming for mdps: Squeezing more out of a heuristic. In: AAAI. pp. 1227–1232 (2006)
9. Wingate, D., Seppi, K.D.: Prioritization methods for accelerating mdp solvers. In: Journal of Machine Learning Research. pp. 851–881 (2005)