Data Structures and Heuristics for State Elimination in Markov Models

MARK BOOM, University of Twente, The Netherlands

Model checking is a popular technique to verify (software) systems, where calculating properties of Markov Models is common practice. Many algorithms exist that achieve this, one of which is state elimination. Optimizing these algorithms provides more accurate results and more performance. In this paper we specifically address the underlying data structures for storing models in memory. We develop a novel data structure, implement it within the framework of an existing model checker and benchmark it against a more traditional object-oriented data structure. Furthermore, we evaluate heuristics regarding the elimination order of states. We show that the data structure used may have a significant impact on performance. Within the realm of Markov Decision Processes, the problem of transition blowup emerges which renders a state elimination algorithm unfit without further optimizations. For Discrete-Time Markov Chains, elimination order has impact on memory usage. However, the effects of a heuristic are highly dependent on the characteristics of the model used.

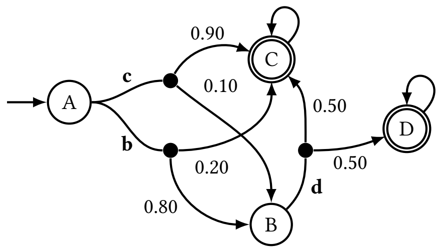

Fig. 1. Example of an MDP

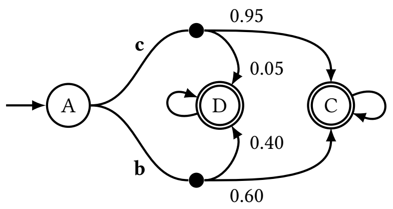

Fig. 2. MDP from Figure 1 with state B eliminated

1 INTRODUCTION

A popular technique for verifying the correct operation of (software) systems is model checking, which aims to guarantee certain properties of systems [8]. This is done by defining all different states the system can be in and the various transitions between those states. A single state of the system can, for example, be defined by taking note of the value of all variables within that system as well as the program counter. Transitions represent events, inputs or the passing of time. Then, predefined properties are asserted. Starting from a predefined initial state, is it possible to reach state X? More complicated questions also play a role: how many steps would it at best or worst take to reach state Y?

The model can be extended by augmenting it with probabilities. Each transition gets annotated with a probability. As a result, it is possible to calculate a new type of property: what is the probability to reach state Z? This type of property forms the core of probabilistic model checking and is the subject of this paper. A collection of states with corresponding transitions based on a probability is called a Markov chain. An important invariant is that for any state, the sum of all outgoing probabilities equals 1. Furthermore, Markov chains are subject to the Markov property which guarantees memorylessness: every decision solely depends on the current state only. In this paper, the concept of a Markov chain refers to a Discrete-Time Markov Chain (DTMC) [9]. Within an instance of the model, one or more goal states are present; properties (such as reachability) of these states are asserted.

A DTMC may be extended with nondeterminism to obtain a Markov Decision Process (MDP) [5]. An MDP is able to model processes where both (user) choice as well as probabilities are involved. Figure 1 shows an example of such a model. From the initial state A, all states can be reached. States C and D are marked as goal states by a double border. Transition c has a probability of 0.9 to reach state C, and a probability of 0.1 to reach state B instead. These different destinations are referred to as the different branches of a transition. Due to the possibility that a state has multiple transitions, the model is no longer deterministic. State C can be reached directly or through state B, so we now will calculate a minimum and maximum probability to reach C instead of the single probability we would obtain for a DTMC model.

This paper describes and motivates an implementation of probabilistic model checking using the state elimination technique [2] for both MDPs and DTMCs, with a focus on the latter. First, we introduce the algorithm along with research questions after which we present related work. We then provide background and definitions, followed by details about the implementation of a state elimination algorithm focusing on the areas of data structures and elimination order heuristics. This implementation is subsequently benchmarked and we present results along with a conclusion.

1.1 State elimination

This section will further elaborate upon the state elimination algorithm [2]. When given an MDP or DTMC, the initial state as well as one or more goal states are known. By eliminating non-goal states, the algorithm results in an MDP or DTMC with only the initial state along with a set of final states remaining. A state is final when it is either a designated goal state or possesses a self-loop. The latter case is considered a non-goal final state; they can not be removed from the model while preserving probabilities. The primary goal of elimination is to directly connect the initial state to the final states with just a single transition. Therefore, after elimination, the desired

TScIT 39, July 7, 2023, Enschede, The Netherlands

properties can be calculated by examining the transitions between the initial state and final states.

Elimination of individual states is achieved by first redistributing the probabilities of any self-loops, if applicable. Each incoming transition of the state to be eliminated will then be connected directly to the destinations of outgoing transitions. This process is later discussed in more detail. An example of the elimination of a single state can be found in Figure 2.

Compared to other methods like value iteration [10], state elimination has the advantage of being able to produce exact results. The Storm model checker [7] also makes use of state elimination to calculate properties of parametric Markov models.

There are also disadvantages to using state elimination. As with almost every algorithm, speed is an important factor. Depending on the shape of the MDP or DTMC that serves as input, state elimination can be a very slow process. During the execution of the algorithm, many lookups and modifications will have to be made to the in-memory structure that represents the given Markov model. When these lookups or operations are performed in an inefficient manner, this can have a significant impact on the overall performance.

Furthermore, it is important to realize that regarding the outcome of the algorithm, the order in which states are eliminated is irrelevant. As such, the question arises which elimination order yields maximum performance.

When applied to MDPs, state elimination suffers from another, more fundamental, issue. When eliminating any given state, every incoming transition causes as many transitions to be created as there are outgoing transitions for that state. Consequently, an exponential increase in the amount of transitions is observed. Without using methods to control this blowup, it is not feasible to perform state elimination on large MDPs. This paper will therefore focus mainly on DTMCs. As DTMCs are strictly limited to a single outgoing transition per state, this effect is less prevalent. Section 2 will further elaborate on this difference.

1.2 Our contributions

Knowing that state elimination inherently is subject to performance issues, but at the same time can also achieve exact results, we would like to investigate methods of implementing state elimination as efficient as possible. The two main avenues of exploration in this paper focus on the underlying data structure as well as elimination order heuristics.

This results in the following research questions:

(1) Which data structures facilitate efficient state elimination in DTMCs?
(2) Which elimination order heuristics improve the efficiency of state elimination in DTMCs?

The findings obtained while answering these questions may guide new implementations of the state elimination towards better performance. Section 2 will introduce definitions and provide context. Sections 3 and 4 will address the research questions about data structure and elimination order heuristics respectively. Section 5 will provide a conclusion.

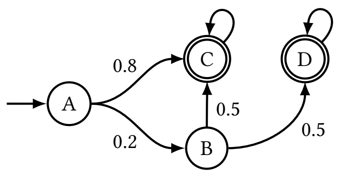

Fig. 3. Example of a DTMC

1.3 Related work

As mentioned, the Storm model checker [7] currently implements state elimination as well. The algorithm has also already been implemented within the Modest Toolset [4]: it features a regular implementation of state elimination within the *mcsta* tool [2] as well as a symblicit implementation [3]. By exploring only a small subsets of states and their transitions at a time during the elimination process, the symblicit implementation aims to keep memory usage to a minimum.

2 BACKGROUND

The most basic probabilistic model that is discussed in this paper is the Discrete-Time Markov Chain (DTMC).

Definition 2.1. A DTMC is a 3-tuple $M = (S, I, T)$ where S is a set of states, $I \in S$ is the initial state and T is a function $S \times S \to [0, 1]$, where $\forall_{s \in S} \sum_{s' \in S} T(s, s') = 1$.

Here, T denotes the transitions between states in S. Note a transition does not need to exist. An example of a DTMC can be found in Figure 3. Note that some definitions include the option to label transitions. While this paper uses labels for illustrative purposes in the examples, they serve no purpose in state elimination and are therefore left out of scope when it concerns the algorithms or data structures used.

By introducing nondeterminism to a DTMC, an MDP is obtained.

Definition 2.2. An MDP is a 4-tuple $M = (S, I, T, B)$ where S is a set of states, $I \in S$ is the initial state, T is a function $S \to \mathcal{P}(B)$, B is a set of functions $S \to [0, 1]$ where $\forall_{b \in B} \sum_{s \in S} b(s) = 1$.

Note $\mathcal{P}(B)$ denotes the powerset of B. By introducing nondeterminism we deviate only slightly from the definition of a DTMC. Instead of a transition heading towards a single destination state, a transition now may result in multiple branches, each of which have a certain probability of transitioning into a destination state. Each item of B represents a collection of branches, which are assigned as transitions to states via relation T. Multiple items out of B can be selected for a single state, as states may have any number of outgoing transitions. Naturally, it is still required that the sum of all branches of a certain transition is 1.

Note that any DTMC can be easily transformed to an MDP where each state only has a single transition with its branches corresponding to the transitions the state had in the original DTMC. Therefore, this state elimination algorithm is able to operate on both model types.

As it is relevant to this paper, we will also define the concept of the Dirac self-loop:

Definition 2.3. Within an MDP, a *Dirac self-loop* is a transition with a single branch of probability 1 that points to the originating state.

To illustrate, in Figure 1, both states C and D posses a Dirac self-loop.

Now that the models as well as relevant concepts are defined, we will define the state elimination algorithm. This algorithm is used for all experiments within this paper and consists of three parts. First, there is the main function which is included as Algorithm 1. Its goal is to loop through all available states, determine if a state is in need of elimination, and call the elimination function. Then, the final property calculation is performed by enumerating the different transitions and summing probabilities for reaching a goal state. Based on the value of the *max* parameter, probabilities to not reach a goal state may be summed instead.

Algorithm 1: Main function

```
Input: $S$, $I$, $goal$, $is\_max$
for $state \in \{s \in S | s \neq I, s \neq goal\}$ do
    eliminate_state($state$)
end
redistribute_loops($init\_state$)
if $max$ then
    $result \leftarrow 0$
else
    $result \leftarrow 1$
end
for $transition \in I.transitions$ do
    $localResult \leftarrow 0$
    for $branch \in \{b \in transition.branches | b \notin \{goal, I\}\}$ do
        $localResult \leftarrow localResult + branch.probability$
    end
    if $is\_max = localResult > result$ then
        $result \leftarrow localResult$
    end
end
return $result$
```

The elimination function is defined in Algorithm 2 and is responsible for eliminating a single state. Assumed is that Dirac self-loops are completely removed beforehand and states that have a Dirac self-loop are instead annotated with the *dirac* property.

Eliminating a state relies on a function that redistributes self-loops, which is defined in Algorithm 3. This function will reassign the probability of individual branches of all outgoing transitions of a state so that there are no branches anymore that loop back to the state the transition originates from.

3 DATA STRUCTURE

The data structure used when implementing the above state elimination algorithm is a crucial factor in performance. From reading the given pseudocode, a few characteristics and common operations are be visible. Notable, for example, is that states themselves are not

Algorithm 2: State elimination algorithm

```
Input: $state$
func eliminate_state()
    redistribute_loops($state$)
    for $t_{in} \in state.incoming\_transitions$ do
        $p \leftarrow t_{in}.get\_branch\_to(state).probability$
        for $t_{out} \in state.transitions$ do
            $t_{new} \leftarrow Transition(source = state)$
            for $b \in t_{in}.branches$ do
                if $b.target = state$ then
                    continue
                end
                $t_{new}.add\_branch(b)$
            end
            for $b \in t_{out}.branches$ do
                $b.set\_probability(p)$
                $t_{new}.add\_branch(b)$
            end
            $t_{in}.source.add\_transition(t_{new})$
        end
    end
    $state.remove\_outgoing\_transitions()$
end
```

Algorithm 3: Redistribution algorithm

```
Input: $state$
func redistribute_loops()
    for $outgoingTransition \in state.transitions$ do
        $branches \leftarrow outgoingTransition.branches$
        $P \leftarrow \sum_{b \in branches | b.destination = state} b.probability$
        for $branch \in \{b \in branches | b.destination \neq state\}$ do
            $branch.probability \leftarrow \frac{branch.probability}{1-P}$
        end
    end
end
```

modified. No states are added and upon elimination merely all transitions coming from and going towards the state will cease to exist. On the other hand, transitions are subject to a lot of modifications as well as backwards lookups (figuring out incoming transitions for a certain state).

Keeping this in mind, we designed a data structure that aims to efficiently deal with these operations. This should result in better performance characteristics than a naive object-oriented approach. First, we will explain our novel data structure (3.1) after which we analyze the time and memory complexity (3.2). Finally, the performance of the novel structure is compared to a naive object-oriented approach (3.3 and 3.4).

3.1 Design and implementation

The data structure proposed in this paper consists of a set of arrays which contain all information about the MDP at hand. These arrays have a length based on the amount of branches present, which varies throughout execution of the algorithm. Taking n as the amount of branches and s as the amount of states, the four arrays are named, typed and sized as follows:

- *branch*, type long, minimum size $4n$
- *probability*, type double, minimum size n
- *goal*, type boolean, minimum size s
- *dirac*, type boolean, minimum size s

Within this paper, we use the long data type to represent state indices to comply with practices already in use within the Modest Toolset, which is later used as a framework for implementation.

In the *branch* array, a single branch is represented by four indices. This could be effectively implemented using, for example, a struct. Transition information is encoded within these branches as well. Every branch contains the following properties:

- The index of the source state
- The index of the target state
- An identifier to match this branch to a transition
- Pointer to the next branch belonging to the state

States are never explicitly stored in the data structure. This is not required as states are not subject to any modifications and therefore an implicit definition is sufficient. Regarding states, we assume the first branch of state with index i is at position $4i$ within the *branches* array. Each state therefore has a very specific slot somewhere near the start of the array.

Furthermore, it should be noted that states are not independent objects within the data structure. Every transition has an identifier—a unique integer with the sole purpose of telling the different transitions apart. In the *branches* array, for each branch the corresponding transition identifier is stored.

The *probability* array stores for each branch the given probability. Given that a branch is a 4-tuple within the *branch* array, a branch which has its first element at index i in the *branch* array will have an assigned probability at $i/4$ in the *probability* array.

Two pieces of state-level information are still to be stored. First, there needs to be a mechanism to keep track of goal states. The *goal* array provides this mechanism, simply indicating with a boolean if a state is a goal state. The index within the *goal* array corresponds to the index of the state, meaning that index i within the goal array corresponds to a state which has the first element of the first branch in position $4i$ of the *branch* array.

Figure 4 provides a visualization of the layout of the four arrays. The corresponding MDP is drawn in Figure 5.

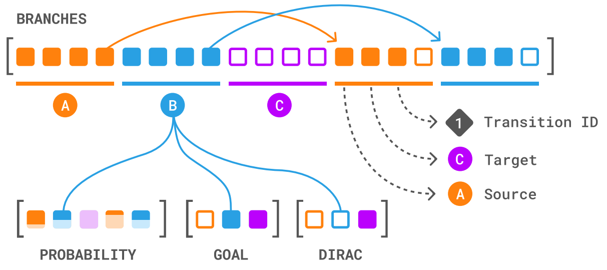

Fig. 4. Visualized layout of arrays in the novel data structure

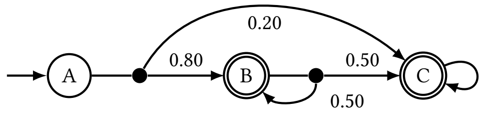

Fig. 5. MDP depicted in Figure 4

The operations the data structure is capable of performing were identified and resulted in the interface below.

```
long[] GetTransitions(long state);
Pair<double, long>[] GetBranches(long transition);
void SetBranchProbability(
    long transition,
    long targetState,
    double newProbability
);
Pair<long, long>[] GetIncomingTransitions(
    long state
);
void RemoveTransition(long transition);
void AddTransition(
    long source,
    Pair<double,long>[] branches
);
bool InitialOrGoal(long state);
bool IsDirac(long state);
```

Notable compromises that were made include the choice to repeat the source state throughout every branch. While this does result in a duplication of the information over all branches of a transition, it allows for a faster determination of the source state when working with incoming transitions.

With this structure, the *branches* array represents a linked list. Array access generally is fast and hence this structure ensures traversing branches of a transition is performant.

Two remarks should be made regarding filling and modifying the *branches* array. As demonstrated, updating branches and transitions is a very common operation when performing state elimination. During the process of elimination, the amount of branches as well as transitions may exceed far beyond the original amount of transitions or branches. At times, it is thus required to update the size of the array. The implementation from this paper multiplies the size of the arrays with 1.5 every time more space is needed. Secondly, one of the advantages of using a linked list this way is that insertions are efficient. To ensure optimal use of the available memory, it is strongly recommended that the implementation keeps track of spots that are freed up due to the deletion of branches. When inserting new branches, those free spots can then be used. Care must be taken, however, that a spot reserved for the first branch of a state due to its position in the array must naturally not be used for a branch belonging to a different state.

3.2 Complexity analysis

In this section we will provide a complexity analysis of both the novel data structure presented in Section 3.1 as an object-oriented approach. We address both time and memory complexity.

Assumed array access has a time complexity of 1 and both longs as well as doubles take 8 bytes to store.

3.2.1 *Novel data structure*

Based on the operations the data structure needs to support, we can identify four main categories of operations. These are the retrieval, update, deletion and addition operations.

Retrieving data. When a state is given, we may want to retrieve transitions and their branches or incoming transitions. Assuming the state has n transitions, then retrieving all of them including branches requires n array accesses since we can traverse the linked list starting from the index that belongs to the targeted state.

Looking up incoming transitions for a certain state is likely the most expensive operation we can perform within the data structure as it requires us to loop over the entire array once. This is unfortunate since we need to perform this operation often and is therefore likely to form a bottleneck within the algorithm.

Updating and deleting transitions. Updating a transition—or, in other words, setting its branches—as well as deleting a transition have the same time complexity. The main and only challenge is to find all branches of the transition within the array. Assuming the state the transition belongs to has n branches in total for all transitions combined, this will on average take $\frac{n}{2}$ array lookups. At worst, it will take n lookups.

Note that the performance of this operation therefore mainly depends on the average amount of branches per state, and not the amount of states.

Adding transitions. To add a transition to a state, we must locate the last branch belonging to the state within the array. Building upon the previous paragraph, this will always require n array accesses where n is the total amount of branches the state already has.

Memory complexity of the structure. Each branch uses $4 * 8 = 16$ bytes within the *branches* array and 8 more in the *probability* array, totaling 24 bytes. A computer with 16 GB of memory would therefore be able to work with up to about 666.000 branches. The amount of states or transitions has no influence on memory usage. This is an advantage to an object-oriented approach as that will suffer from the overhead of creating separate objects for states and transitions, which take up space in memory.

3.2.2 *Object-oriented approach*

When using a detailed representation of the state space as objects in memory, operations become more straightforward as more bookkeeping is being done: when, for example, keeping track of incoming transitions on state objects, it is now much easier to produce a list of incoming transitions for a certain state. However, both the bookkeeping as well as the individual operations do require extra work and therefore result in worse performance when compared to simple array accesses.

The object-oriented approach also requires more memory. The exact amount of memory is hard to determine as it will vary much depending on the runtime used, but in general we can conclude that each object has some overhead when stored in memory. We will need objects for both states, transitions and branches as well as all extra information we store such as incoming transitions; they all have a memory footprint.

3.3 Experiments

We benchmarked aforementioned data structure in order to evaluate its performance. Prior to benchmarking, we implemented the algorithm. The Modest Toolset [4] served as a good basis for implementation due to already having support for a wide range of model input formats, a built-in benchmarking tool and in general a solid framework for dealing with probabilistic models such as DTMCs and MDPs. The existing toolset is written in C#, and so was this algorithm.

Not only was the aim to benchmark the novel data structure introduced in this paper, but a comparison is also deemed desirable. Therefore, the same algorithm has been implemented using a more naive object-oriented approach.

To reconcile the two approaches and make accurate benchmarking possible, an interface was constructed first. This interface contains all methods required to manipulate the given MDP and was then implemented twice allowing for object-oriented variant alongside the novel data structure. The interface implements the methods exactly as specified in Section 3.1.

The models used for the benchmarks originate from the Quantitative Verification Benchmark Set [6]. We listed the models we used for the experiments in Table 1. For MDPs, we selected two models small enough to be completed by the algorithm in a reasonable amount of time. For DTMCs, we selected all available models with a P property from the set.

We also included the *haddad_monmege* DTMC not only because it is part of the benchmark set, but also because it is an especially interesting case due to it being specifically designed to highlight issues with convergence when using value iteration [1]. While value iteration algorithms tend to produce incorrect results for this model, state elimination algorithms are expected to produce an exact result. Our algorithm does so, though when $N > 54$, incorrect results are still produced (presumably due to floating-point rounding errors).

While benchmarking data structures, no use was made of any elimination order heuristic. This paper will later study different elimination order heuristics. The arbitrary heuristic eliminates states based on insertion order within the state space. This is a deterministic order, in each run and on both data structures, states where eliminated in exactly the same order.

We used the *mobench* tool, which is embedded into the Modest Toolset, to solve each model in Table 1 three times using both the novel data structure presented in Section 3.1 as well as a naive object-oriented data structure.

Experiments have been conducted on a 2020 MacBook Pro (macOS Ventura, Intel i5 10^{th} generation 2 GHz quad-core) with 16 GB of RAM. A timeout of 10 minutes was used.

3.4 Results

We present benchmark results in Table 2. For each model, the average amount of seconds over the three runs is listed. Not all runs did complete within the maximum amount of time.

On all DTMC instances, our novel approach performed better than or equal to the object-oriented approach. The object-oriented approach was unable to solve the larges instances within the timeout, while the novel approach was able to do so. However, the results also indicate that when working with MDP models, the novel approach can be slower, up to 22% at the *cdrive* model.

Table 2. Benchmark results for data structures

Model	Object-oriented	Novel
cdrive	415*s*	529*s*
triangle_tireworld	6.7*s*	23.2*s*
haddad_monmege	1.1*s*	1.1*s*
coupon	5.1*s*	5.0*s*
leader_sync	3.6*s*	3.5*s*
brp	3.3*s*	2.4*s*
crowds	57.8*s*	20.6*s*
egl	Timed out	317*s*
nand	Timed out	220*s*

4 ELIMINATION ORDER HEURISTICS

The given state elimination algorithm produces correct results no matter the order in which states are eliminated. The elimination order can, however, impact memory usage of the algorithm. After all, more transitions and therefore more branches require more space in memory. This can easily be demonstrated by fully randomizing the elimination order and running the algorithm a few times. Substantial differences in time to completion can then be observed, though every result will still be correct. This is illustrated by Figure 6.

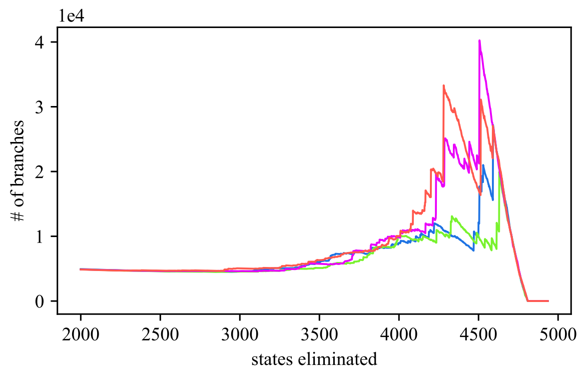

Fig. 6. Evolution of number of branches when eliminating states in a random order, 4 runs, using the *brp* model

4.1 Implementation

We will use the following heuristics for benchmarking:

- *max_trans* (for MDPs)
  max_branch (for DTMCs)
- *min_trans* (for MDPs)
  min_branch (for DTMCs)
- *in/out* (referred to as *io*)

For DTMCs, focus is placed on branches while for MDPs focus is instead placed on transitions. This has to do with the representation of the different models in memory: a DTMC is simply an MDP where every state has exactly one transition and many branches. The amount of branches affects memory usage and performance and is hence an interesting factor. With MDPs we expect potentially many transitions per state, each of them not containing a disproportionally large amount of branches. Hence, with MDPs transitions are the more interesting metric.

When using the *max* and *min* heuristics, priority is given to either states with respectively the most or least branches or transitions. The *io* heuristic calculates a value for each state. This value is based on the number of incoming as well as outgoing transitions. They are multiplied, and states with the highest value are eliminated first.

Definition 4.1. The *value* of a given state is the number of its incoming transitions multiplied by the number of its outgoing transitions.

As with the data structures, we established an interface to represent a certain heuristic. The interface is structured as follows:

```
long Next();
bool Done();
void EventInit(long stateCount);
void EventTransitionAdd(long transition, long state);
void EventTransitionRemove(
    long transition,
    long state
);
void EventTransitionUpdate(long transition);
```

Table 1. Models used, along with their types and parameters

Model	Type	Parameters	Target	# of states
cdrive	MDP	$c = 2$	*goal*	38
triangle_tireworld	MDP	$l = 9$	*goal*	80
haddad_monmege	DTMC	$N = 50, p = 0.7$	*target*	101
coupon	DTMC	$N = 5, DRAWS = 2, B = 5$	*collect_all*	4.155
leader_sync	DTMC	$N = 5, K = 4$	*eventually_elected*	4.244
brp	DTMC	$N = 64, MAX = 5$	*p1*	5.192
crowds	DTMC	$TotalRuns = 6, CrowdSize = 5$	*positive*	15.233
egl	DTMC	$N = 5, L = 8$	*unfairA*	115.123
nand	DTMC	$N = 20, K = 2$	*reliable*	154.942

The $Next()$ method provides the next state to eliminate based on the heuristic. The $Done()$ method can be used to determine if there exists a next state. Furthermore, the heuristic possesses a few methods that get called when events that mutate the model occur. This allows for the construction of a more efficient heuristic, since any changes in the desirable order can be processed immediately.

4.2 Experiments

In order to analyze aforementioned heuristics, we will run the elimination algorithm using all of them while continuously collecting the number of branches and transitions. These numbers are collected after each elimination and can therefore be visualized in a graph that plots the number of states eliminated against the total number of branches and transitions.

On top of the three heuristics introduced in this section, for completeness we also plot the arbitrary heuristic (*arb*). This is the same heuristic used in Section 3.

We will benchmark a selection of models from the benchmark set introduced in Section 3. Findings will be reported in the next section, where we focus on interesting characteristics of the graphs.

Especially when dealing with MDP models, due to a large increase in the amount of transitions, some models may not be able to finish within a reasonable amount of time. We take a 10 minute timeout into account, terminating every run that does not finish within the timeframe. The same hardware is used as listed in section 3.3.

4.3 Results

The results of the experiments are represented in graphs in Figure 7 (for MDP models) and Figure 8 (for DTMC models).

For the MDP models, not all simulations were able to finish. Only the *arb* and *io* heuristics for the *cdrive* models finished computing within the given timeframe. Other heuristics were unable to contain an exponential blowup of transitions.

For DTMCs, less of an exponential effect is observed. All models finished computing well before the timeout. In the three models shown, *max* generally was the most optimal heuristic. It is, however, very difficult to generalize that statement as the *min* heuristic illustrates: it is the optimal heuristic in two cases, but the worst heuristic in the *crowds* model.

The *io* heuristic performs poor on the *coupon* model, average on the *crowds* model and best on the *leader_sync* model.

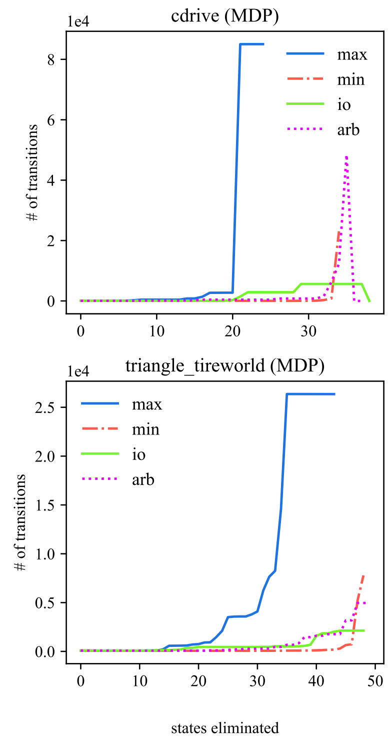

Fig. 7. Heuristic benchmark results for MDP models

The *leader_sync* model produces a remarkable graph; all four heuristics follow the same path. We observed the same effect when benchmarking the $haddad_monmege$ model.

Other models, such as the *brp* model produced graphs similar to the *coupon* and *crowds* models.

5 CONCLUSION

By making use of models from the Quantitative Verification Benchmark Set [6] we demonstrated that when implementing an algorithm for state elimination, the underlying data structure can have a significant impact on performance. We conducted experiments that compared a novel data structure against a more traditional and

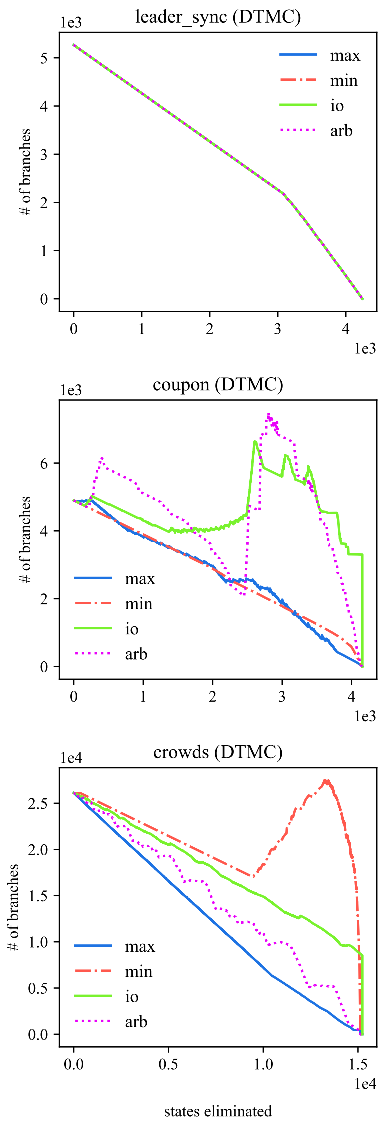

Fig. 8. Heuristic benchmark results for DTMC models

naive object-oriented implementation. The novel implementation is inspired by operations the state elimination algorithm frequently performs and tries to refrain from any unnecessary bookkeeping an object-oriented approach entails. We implemented the novel data structure within the framework of the Modest Toolset [4]. Results show a considerable improvement in performance for DTMCs when using our novel approach. For MDPs results are inconclusive, with the two models that we benchmarked performing better using an object-oriented approach.

The inability from drawing conclusions around MDP models stems from performance issues caused by the phenomenon of transition blowup when eliminating states from MDPs. We illustrated how the amount of transitions within an MDP will grow exponentially when the state elimination algorithm is applied. We must conclude that without further optimizations regarding this issue, state elimination is not a viable option for MDPs.

The second part of the paper investigated several heuristics that can be applied to the elimination order of states. Three heuristics were investigated. The heuristic orders states by the amount of transitions or branches. We developed both a minimum and maximum approach as well as an approach where a value was calculated based on the product of incoming and outgoing transitions and branches. For DTMCs, the number of branches is used as a metric instead of the number of transitions due to the fact that within a DTMC each state possesses exactly one transition with many branches. In this experiment, we show that heuristics have a significant effect on the amount of transitions and branches generated and thereby potentially influence memory usage of the algorithm. We were unable to identify a single heuristic that yields a positive effect on every model, indicating the applicability of heuristics depends on the characteristics of the model.

State elimination algorithms are one way to perform probabilistic model checking. An advantage compared to other methods like value iteration is the ability of state elimination to compute exact results. When implementing such an algorithm, care must be taken to carefully select a data structure that fosters performance. When dealing with MDPs, extra caution is advised and in general it should be assumed that state elimination is not a viable option. For DTMCs, where state elimination is a more successful strategy, heuristics regarding elimination order play a role in the amount of memory used. The effects of heuristics, however, may differ strongly based on the characteristics of the model used.

5.1 Future work

While this paper explores state elimination for both MDPs as well as DTMCs, we clearly illustrated that a naive state elimination algorithm is not able to handle MDPs of moderate sizes due to transition blowup. Further research could focus on methods to contain this blowup by combining transitions as they are being created. Additionally, the newly developed data structure could be further improved, especially regarding the currently expensive operation of looking up incoming transitions.

ACKNOWLEDGMENTS

Thanks to Arnd Hartmanns for supervising the research, introducing me to the topic and providing extensive feedback on the paper.

REFERENCES

[1] Serge Haddad and Benjamin Monmege. 2018. Interval iteration algorithm for MDPs and IMDPs. *Theoretical Computer Science* 735 (2018), 111–131. https://doi.org/10.1016/j.tcs.2016.12.003

[2] Ernst Moritz Hahn and Arnd Hartmanns. 2016. A Comparison of Time- and Reward-Bounded Probabilistic Model Checking Techniques. In *Dependable Software Engineering: Theories, Tools, and Applications - Second International Symposium, SETTA 2016, Beijing, China, November 9-11, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9984)*, Martin Fränzle, Deepak Kapur, and Naijun Zhan (Eds.). 85–100. https://doi.org/10.1007/978-3-319-47677-3_6

[3] Ernst Moritz Hahn and Arnd Hartmanns. 2021. Symblicit exploration and elimination for probabilistic model checking. In *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021*, Chih-Cheng Hung, Jiman Hong, Alessio Bechini, and Eunjee Song (Eds.). ACM, 1798–1806. https://doi.org/10.1145/3412841.3442052

[4] Arnd Hartmanns and Holger Hermanns. 2014. The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8413)*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, 593–598. https://doi.org/10.1007/978-3-642-54862-8_51

[5] Arnd Hartmanns, Sebastian Junges, Tim Quatmann, and Maximilian Weininger. 2023. A Practitioner's Guide to MDP Model Checking Algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13993)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer, 469–488. https://doi.org/10.1007/978-3-031-30823-9_24

[6] Arnd Hartmanns, Michaela Klauck, David Parker, Tim Quatmann, and Enno Ruijters. 2019. The Quantitative Verification Benchmark Set. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11427)*, Tomás Vojnar and Lijun Zhang (Eds.). Springer, 344–350. https://doi.org/10.1007/978-3-030-17462-0_20

[7] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2022. The probabilistic model checker Storm. *Int. J. Softw. Tools Technol. Transf.* 24, 4 (2022), 589–610. https://doi.org/10.1007/s10009-021-00633-z

[8] Holger Hermanns and Joost-Pieter Katoen. 2002. Guest editors' introduction: Model checking in a nutshell. *J. Log. Algebraic Methods Program.* 52-53 (2002), 1–5. https://doi.org/10.1016/S1567-8326(02)00030-9

[9] Joost-Pieter Katoen. 2013. Model Checking Meets Probability: A Gentle Introduction. In *Engineering Dependable Software Systems*, Manfred Broy, Doron A. Peled, and Georg Kalus (Eds.). NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 34. IOS Press, 177–205. https://doi.org/10.3233/978-1-61499-207-3-177

[10] Tim Quatmann and Joost-Pieter Katoen. 2018. Sound Value Iteration. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 643–661. https://doi.org/10.1007/978-3-319-96145-3_37