POWVER

Technical Report 2021-06

Title: **A Modest Approach to Dynamic Heuristic Search in Probabilistic Model Checking**

Authors: Michaela Klauck, Holger Hermanns

Report Number: 2021-06

ERC Project: Power to the People. Verified.

ERC Project ID: 695614

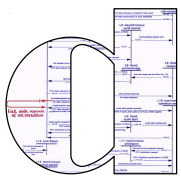
Funded Under: H2020-EU.1.1. – EXCELLENT SCIENCE

Host Institution: Universität des Saarlandes, Dependable Systems and Software
Saarland Informatics Campus

Published In: QEST 2021

Please cite this publication as follows:

Michaela Klauck, Holger Hermanns.
A Modest Approach to Dynamic Heuristic Search in Probabilistic Model Checking.
Quantitative Evaluation of Systems - 18th International Conference, QEST 2021, Paris, France, August 23-27, 2021, Proceedings. Lecture Notes in Computer Science 12846, Springer 2021, ISBN 978-3-030-85171-2: 15-38.



POWER TO THE PEOPLE.
VERIFIED.
erc

A Modest Approach to Dynamic Heuristic Search in Probabilistic Model Checking*

Michaela Klauck[1] and Holger Hermanns[1,2]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2]Institute of Intelligent Software, Guangzhou, China
{klauck,hermanns}@cs.uni-saarland.de

Abstract. This paper presents Modysh, a probabilistic model checker which harvests and extends non-exhaustive exploration methods originally developed in the AI planning context. Its core functionality is based on enhancements of the heuristic search methods *labeled real-time dynamic programming* and *find-revise-eliminate-traps* and is capable of handling efficiently maximal and minimal reachability properties, expected reward properties as well as bounded properties on general MDPs. Modysh is integrated in the infrastructure of the Modest Toolset and extends the property types supported by it. We discuss the algorithmic particularities in detail and evaluate the competitiveness of Modysh in comparison to state-of-the-art model checkers in a large case study rooted in the well-established *Quantitative Verification Benchmark Set*. This study demonstrates that Modysh is especially attractive to use on very large benchmark instances which are not solvable by any other tool.

1 Introduction

Markov decision processes (MDPs) are the base model for probabilistic model checking. A variety of probabilistic model checkers are being developed, and are supported by orchestrated initiatives like the QComp competition [18,13] and the quantitative verification benchmark set QVBS [24]. While in probabilistic model checking MDPs often reflect concurrency phenomena, they have a longer tradition in the context of *sequential decision making under uncertainty* [6,27].

Depending on the modelling context, MDPs are usually decorated with rewards or costs. The term reward is traditionally used if the goal is to maximize the earnings. In the dual context of costs, the spendings are usually to be minimized, under the assumption that decisions in the MDP are controllable. Instead, in a setting where the MDP results from concurrent interleavings it can also be natural to ask for the maximal cost lurking or the minimal reward obtainable, since here the decisions need to be assumed as being uncontrollable. Typical properties of interest in this context include (max, min) reach probabilities w.r.t. a set of goal

* This work has received support by the ERC Advanced Investigators Grant 695614 POWVER, by the DFG Grant 389792660 as part of TRR 248 CPEC, and by the Key-Area Research and Development Grant 2018B010107004 of Guangdong Province.

states as well as (max, min) expected rewards (costs) which are accumulated until reaching a goal state. These properties can also include bounds on the number of steps until reaching a goal or enforce a certain reward (cost) amount to be accumulated on the way to the goal. Iterative methods like *value iteration* are the standard solution to calculate results for these property types. In its basic form, value estimates for each state in the state space are updated synchronously based on the values of their successors until convergence is reached [6].

Heuristic search methods [4,21,8,9] try to compute such optimal values based on only a small fraction of the states, sufficient to answer the considered property. These methods exploit state-wise estimates of the optimal value, for only a subset of the state space. The order in which values are updated is made dependent on their current value estimates, in an approach called *asynchronous value iteration*.

This paper presents a probabilistic model checker that harvests modified versions of asynchronous value iteration based on heuristic search. The core components are the *labeled real-time dynamic programming* (LRTDP) [8] and *find-revise-eliminate-traps* (FRET) [31] procedures. LRTDP tries to find the optimal values by continually updating the current best solution of the state value estimates on single exploration paths. Only one state's value is updated at each step. FRET is needed to guarantee convergence of LRTDP to the optimal value in special MDP structures. It eliminates cycles to guide LRTDP to the correct solution. While contributions to this research line are manyfold (see Sect. 5), they are quite fragmented w.r.t. assumptions on property types and model characteristics. We instead take care to support most of the established property types, from reach probabilities to reward expectations (but no long-run averages), also including bounded versions, on general MDP structures efficiently.

As a result, our tool Modysh considerably enlarges the property types supported by heuristic methods. The new elements and their integration are described in detail in this paper. A large empirical evaluation shows that Modysh is competitive relative to state-of-the-art model checkers and is able to solve benchmark instances which are too large to be solved by other tools. Modysh is shipped as an extension component to the Modest Toolset [22] inside which it can be considered as an alternative to mcsta [16,19,23], which is an explicit-state probabilistic model checker based on traditional value iteration. The toolset is available for Windows, Linux and Mac OS. Integrating Modysh into it brings the benefit that the same input languages and operating systems are supported, and it opens the Modest Toolset for property types not supported thus far.

Outline. In Sect. 2 we review the theoretical background. Sect. 3 introduces heuristic search approaches and discusses how LRTDP and FRET can be extended and modified such that they are applicable to general MDP structures and properties. Sect. 4 presents a large empirical evaluation demonstrating that Modysh is competitive, outperforming state-of-the-art model checkers especially on very large state spaces with a parallel structure. We conclude with a short discussion of our achievements.

2 Theoretical Background

Before looking into the details of the heuristic search techniques implemented in MODYSH, we introduce the theoretical background. A probability distribution over a (countably in-)finite set X is a function $\mu : X \to [0,1]$ s.t. $\sum_{x \in X} \mu(x) = 1$. We denote by $\mathcal{D}(X)$ the set of all probability distributions over X.

A *Markov Decision Process (MDP)* is a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, s_0, \mathcal{S}_* \rangle$ consisting of a finite set of *states* $\mathcal{S}$, a finite set of *actions* $\mathcal{A}$, the partial *transition probability function* $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{D}(\mathcal{S})$, a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}_0^+$ assigning a reward (or cost) value to each triple of state, action, state, a single *initial state* $s_0 \in \mathcal{S}$, and a set of absorbing *goal states* $\mathcal{S}_* \subseteq \mathcal{S}$.

An action $a \in \mathcal{A}$ is *applicable* in a state $s \in \mathcal{S}$ if $\mathcal{P}(s,a)$ is defined. In this case we denote by $\mathcal{P}(s,a,t)$ the probability $\mu(t)$ of state t according to $\mathcal{P}(s,a) = \mu$. We denote by $\mathcal{A}(s) \subseteq \mathcal{A}$ the set of all actions that are applicable in s. We restrict to MDPs where for each state s, $\mathcal{A}(s)$ is nonempty, which is no restriction as per the following. A state s is called *terminal* if $|\mathcal{A}(s)| = 1$ and for this $a \in \mathcal{A}(s)$ it holds that $\mathcal{P}(s,a,s) = 1$ and $\mathcal{R}(s,a,s) = 0$. All goal states g are assumed to be terminal, which forces to stay in g forever without accumulating further reward. Terminal states not contained in $\mathcal{S}_*$ are called *dead-ends*.

For a given MDP $\mathcal{M}$, a function $\pi : \mathcal{S} \to \mathcal{A}$ with $\pi(s) \in \mathcal{A}(s)$ for each state s is called a (memoryless) *policy*, used to determine the next action to take for any given state. We later extend this when focussing on specific, bounded properties. The *accumulated reward* over an infinite sequence of states $\zeta = (s_i)_{i \in \mathbb{N}}$, called *path*, induced by a policy π through $\mathcal{M}$ is defined by $\rho(\zeta) = \sum_{i=0}^{\infty} \mathcal{R}(s_i, \pi(s_i), s_{i+1})$. For the finite prefixes τ of such a path, called finite paths, the reward summation constituting $\rho(\tau)$ is truncated accordingly. We let $\mathit{Paths}(\mathcal{M})$ denote the set of all paths through $\mathcal{M}$ rooted in its initial state s_0. Each policy π induces a probability space on the set of infinite paths through $\mathcal{M}$ in the usual way [25] and this in turn induces well-defined probability measures for each of the finite paths τ, and similarly for the accumulated reward measures $\rho(\tau)$. States from which $\mathcal{S}_*$ can not be reached with positive probability regardless of the policy π are called *sink states* and collected in $\mathcal{S}_\perp$. This set can be precomputed by a simple fixpoint computation (checking for each state the LTL property $\square\neg\texttt{goal}$ where `goal` identifies all states in $\mathcal{S}_*$) in the underlying graph. This graph G over $\mathcal{S}$ is spanned by the edge set $E = \{(s,t) \mid \exists a \in \mathcal{A} : \mathcal{P}(s,a,t) > 0\}$.

The subgraph G_π induced by policy π is obtained by restricting the edge set of G to $\{(s,t) \mid \mathcal{P}(s,\pi(s),t) > 0\}$. π is *almost-sure* if the probability of reaching $\mathcal{S}_*$ it induces is 1 regardless of the initial state. If instead that probability is guaranteed to be positive, π is called *proper*. A *cycle* is a path in G starting and ending in the same state. A *strongly connected component (SCC)* in G is a subset of states V such that $\forall (s,t) \in V \times V$ a path from s to t exists. A bottom SCC (BSCC) B is a SCC of maximal size from which only states in B are reachable.

Measures of Interest. We denote by P^π the probability measure induced by π and by E^π the expectation of the accumulated reward ρ w.r.t. measurable sets of paths starting in s_0. We define the extremal values $P_{\max}(\Pi) = \sup_\pi P^\pi(\Pi)$ and $P_{\min}(\Pi) = \inf_\pi P^\pi(\Pi)$, as well as $E_{\max}(\Pi) = \sup_\pi E^\pi(\Pi)$ and $E_{\min}(\Pi) =$

$\inf_\pi E^\pi(\Pi)$, for measurable $\Pi \in Paths(\mathcal{M})$ and $\pi \in \Pi$. We consider the following property types, echoing what JANI supports [28,12], with $opt \in \{\max, \min\}$:

- *MaxProb* and *MinProb*: $P_{opt}(S_U \ \mathcal{U} \ \mathcal{S}_*) = P^{opt}(\{\tau \in Paths(\mathcal{M}) \mid \exists s \in \mathcal{S}_* : \tau = (s_i)_{i=0}^{\infty} \wedge s = s_j \wedge \forall k < j : s_k \notin \mathcal{S}_* \wedge s_k \in S_U\})$ is the max/min probability of eventually reaching a goal state and all states visited before being in S_U. $P_{opt}(S_U \ \mathcal{U} \ \mathcal{S}_*)$ will be abbreviated as $P_{opt}(\diamond \mathcal{S}_*)$.
- maximal/minimal expected rewards: $E_{opt}(S_U \ \mathcal{U} \ \mathcal{S}_*) = E^{opt}(\{\tau \in Paths(\mathcal{M}) \mid \exists s \in \mathcal{S}_* : \tau = (s_i)_{i=0}^{\infty} \wedge s = s_j \wedge \forall k < j : s_k \notin \mathcal{S}_* \wedge s_k \in S_U\})$ is the maximal or minimal reward expectation of eventually reaching a goal state. Note that reward ∞ is accumulated for non-almost-sure policies.
- step bounded properties: $P_{opt}(S_U \ \mathcal{U}_{[l,u]} \ \mathcal{S}_*)$ is the maximal or minimal probability of reaching a goal state in $[l, u]$ steps defined as $P^{opt}(\Pi_{[l,u]})$ where $\Pi_{[l,u]}$ is the set of paths that reach a goal state in $[l, u]$ steps while only passing through S_U. Similar for step bounded expected reward properties.
- reward bounded properties: If a reward structure is defined, $P_{opt}(S_U \ \mathcal{U}_{[l,u]} \ \mathcal{S}_*)$ is the extremal probability of reaching a goal state with accumulated reward in $[l, u]$ defined as $P^{opt}(\Pi_{[l,u]})$ where $\Pi_{[l,u]}$ is the set of paths having a prefix τ with accumulated reward in $[l, u]$ containing a goal state and only passing through S_U before. Similar for reward bounded expected reward properties. Bounds with open intervals are also supported (for all bounded properties).

3 Dynamic Heuristic Search

Value Iteration. The problems discussed above are in practice often solved using *value iteration*. This is a variant of dynamic programming where a value is assigned to each state by a value function $V : \mathcal{S} \to \mathbb{R}$ which specifies the current approximation of the value of this state. The value function is placed in an iterative procedure updating the states' values depending on the values of their successors. These values are refined until convergence to the least fixpoint. In many situations this fixpoint $V^* = \lim_{n\to\infty} V_n$ corresponds to the optimal value one is looking for, from which the optimal policy can be extracted. Usually, the value function is calculated greedily via the *Bellman function* [5] (similar for maximum): $V_{i+1}(s) = \min_a \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s') \cdot (\mathcal{R}(s, a, s') + V_i(s'))$ (1) where a value of 1 is assigned to goal states and 0 to dead-ends. A value function is *admissible* if it is an optimistic estimate of the correct final value. This means, if we try to minimize, the value function V is admissible if always $V(s) \leq V^*(s), \forall s \in \mathcal{S}$. If we instead maximize, a value function with $V(s) \geq V^*(s), \forall s \in \mathcal{S}$ is admissible. A *greedy policy* is always defined w.r.t. a value function V. For each state the greedy policy always picks the action leading to the successor state with the best value according to the value function. This action may not be unique which means there can be multiple greedy policies. A *greedy graph* G_V of graph G with respect to value function V is the superposition of all G_π induced by any greedy policy π w.r.t. V, so it is the combined reachability graph of all greedy policies.
Heuristic Search. The approach we generally pursue is based on the heuristic search algorithm LRTDP [8], a heuristic search dynamic programming optimiza-

tion of standard value iteration. To find an optimal policy, up to a prespecified accuracy ε, starting in an initial state, it attempts to avoid exploring the entire state space and delivers the requested values for the initial state only, instead of for all states as in standard value iteration. It constantly keeps updating a *current best solution*, a partial value function providing the current state value estimates. In each round only a single state is selected for an update. These updates are obtained by repeatedly sampling *trials*, i. e., executions starting in the initial state, and ending once a state is reached for which an update does not change the value estimate by more than ε. While doing so, the optimal policy is constructed incrementally by extending a partial policy step by step. A partial policy π will be called *closed* for a state $s \in \mathcal{S}$, if $\pi(t)$ is defined for every state $t \notin (\mathcal{S}_\bot \cup \mathcal{S}_*)$ that is reachable (with positive probability) from s by following π.

Traps. A *trap* [30, p. 171 ff.] is a BSCC not containing a goal state. In our approach *traps are defined on the greedy graph* G_V induced on G by value function V. We distinguish *permanent traps* which are also BSCCs of G, i. e., there is no non-greedy policy which would lead out of the trap. In contrast, *transient traps* are SCCs, but not BSCCs of G, so there is a policy leading out of the trap.

Convenience MDPs. The planning literature has identified a number of model classes with convenient properties and initially arbitrary rewards in $\mathbb{R}$. A *Stochastic Shortest Path* ($\mathcal{SSP}$) MDP [6] is an MDP admitting (i) at least one almost-sure policy and (ii) inducing expected accumulated reward ∞ for each not almost-sure policy π. The latter corresponds to G_π containing no reachable cycle on which (in the MDP) the accumulated reward does not increase. Assuming the former, the latter can trivially be enforced by restricting to models with reward function confined to positive values (possibly except at goal states). As an apparent relaxation, Bertsekas [7] later introduced condition (i′) and (ii′) which replace the role of *almost-sure* policies by *proper* policies in (i), respectively (ii), but showed them to be (pairwise) equivalent. In a *Generalized Stochastic Shortest Path* ($\mathcal{GSSP}$) MDP [30] the first condition (i) is kept while the second condition is further relaxed by instead assuming that (ii″) for each policy and state the expected sum of negative rewards is bounded from below. This relaxation in particular supports zero-reward cycles, while it precludes cycles with alternations of positive and negative rewards that cancel out. Condition (ii″) can trivially be enforced by restricting to models with a reward function confined to non-negative values, as we do. Our contribution relinquishes condition (i) and (i′) of $\mathcal{SSP}$ and $\mathcal{GSSP}$, i. e., we do not rely on the existence of almost-sure or proper policies.

Algorithm Overview. We introduce our algorithmic contributions in the sequel one-by-one. All modifications, adaptions and extensions made to the original versions are marked in blue. If existing, the original version of modified lines is stated in comments of the form ▷. . . . The base algorithm expects as inputs the state s of the MDP for which to evaluate the property, the result precision ε and uses flags dependent on the property class to be evaluated. `max-rew` is *True* if a maximal expected reward property is evaluated, otherwise it is *False*, analogously for `min-rew`. We do not use explicit flags for indicating (max or min) reachability probabilities because there are no code fragments specific to these

property types. We assume that the initial and current value function, V_0 and V_i, are always globally accessible.

In fact, the original algorithmic contributions have been made without a specific focus on reachability probabilities, which as long as zero reward values are supported, can actually be cast into reward accumulations. We here make an explicit distinction between these cases for the purpose of better explanability and for the purpose of more direct and hence faster implementation in MODYSH.

3.1 Reachability Properties

For reachability properties `max-rew` and `min-rew` are set to *False*. We first concentrate on calculating *MinProb*, i. e., $P_{\min}(S_U \; \mathcal{U} \; \mathcal{S}_*)$. We detail our modifications to the original version of the algorithm in order to enable that condition (i′) and thus (i) can be dropped. Afterwards we turn to *MaxProb* and show how FRET-LRTDP can be modified to solve these kind of properties on general MDPs, too. Kolobov et al. [31] already provided a reduction to show that FRET in combination with LRTDP is applicable to general *MaxProb* properties, even if condition (i′) is violated. We will give an alternative proof, based on the proof for *MinProb*, demonstrating that our implementation is also valid for general MDP types as defined above, not only for problems having at least one proper policy.

We denote by $V^\pi : \mathcal{S} \mapsto [0,1]$ the goal-reachability probabilities induced by π. Goal states $\mathcal{S}_*$ have probability value 1 while sinks and other states enforced to be avoided have probability value 0. This corresponds to the fact that if a partial policy π is closed for s, V^π constitutes the least fixpoint of Equation (2).

$$V^\pi(s) = \begin{cases} 1 & \text{if } s \in \mathcal{S}_*, \\ 0 & \text{if } s \in \mathcal{S}_\perp \cup \overline{S_U} \setminus \mathcal{S}_*, \\ \sum_{s' \in \mathcal{S}} \mathcal{P}(s, \pi(s), s') \cdot V^\pi(s') & \text{otherwise.} \end{cases} \tag{2}$$

Minimum Reach Probability. For *MinProb* properties the objective is to find the minimal probability to reach a state in $\mathcal{S}_*$ if initialized in s_0 and while avoiding the complement of S_U. We are ultimately interested in the value

$$V^*(s_0) = \min_{\pi : \pi \text{ closed for } s_0} V^\pi(s_0). \tag{3}$$

An admissible initialization for this case is a valuation of 0, except for goal states which get a value of 1. Using a reward function defined as $\mathcal{R}(s,a,s') = 1$ if $s \notin \mathcal{S}_* \wedge s' \in \mathcal{S}_*$ and 0 otherwise and then applying the Bellman equation (1) of synchronous value iteration will iteratively fill the partial policy bottom up. Spelled out for our case, this amounts to replacing the third line of (2) by

$$\min_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s') \cdot V^\pi(s') \quad \text{otherwise.} \tag{4}$$

which echoes the greedy nature of the computation. However, giving up synchronicity in favor of a heuristic approach is the key to efficiency. The base algorithm for this case we call GLRTDP, a generalization of LRTDP [8, Alg. 4]. The pseudocode is shown in Alg. 1. The algorithm iteratively selects only a single state for a Bellman update in each round. It continually updates a *current best solution*, a partial function providing the current state value estimates and

repeatedly runs *trials* (line 4), sample executions of the MDP, starting from the initial state, and ending once a state is reached for which an update does not change the value by more than ε, i. e., ε-consistency is reached (line 13, lines 17-27 are not relevant here). To determine which successor state to follow after state s, GLRTDP considers an action $a \in \mathcal{A}(s)$ *greedy* w.r.t. the current value function (line 14), i. e., one that minimizes Equation (4) for s (cf. Alg. 2, line 2 and 4) [8, Alg. 2], and then selects a successor state (line 16). Picking the next state randomly from the set of successors of the greedy action (cf. Alg. 2, line 9) instead of taking the probability into account is an optimization leading to better performance as noted in probabilistic FAST DOWNWARD [36]. The entire exploration procedure is systematic, i. e., does not starve relevant states if the heuristic function used is admissible. A state, which has not converged so far, will not stay in the greedy graph forever without its value being revised. Therefore, it is guaranteed to converge to an optimal solution. After each trial, those states are labeled as *solved* whose values and those of their descendants have reached ε-consistency (cf. Alg. 3) [8, Alg. 3]. Trials are terminated at solved states. GLRTDP terminates the value update procedure as soon as the initial state is solved (cf. Alg. 1 line 3, 8, and 31).

Alg. 1 General Labeled Real-Time Dynamic Programming (GLRTDP)

```
1:  proc GLRTDP(s: State; ε: float)
2:   max-rew, min-rew = True, if max., resp.
       min. reward property is calculated
3:   while ¬Solved(s) do
4:    GLRTDP-trial(s, ε)

5:  proc GLRTDP-TRIAL(s: State, ε: float)
6:   visited := Empty-Stack
7:
8:   while ¬Solved(s) do
9:    visited.Push(s)
10:   v_old = V(s)
11:   Update(s)
12:   v_new = V(s)
13:   if Is-cons(v_old, v_new, ε) then break
                ▷ original condition Is-goal(s)
14:   a := Greedy-action(s)
15:   if a ≠ NULL then
16:    s := Pick-next(a, s)
17:    if max-rew
         && visited.Contains(s) then
18:     if Elim-cycle-max-rew() then
19:      V(init-node) = ∞
20:      Solved(init-node) = True
21:      return
22:    else
23:     if min-rew
          && visited.Contains(s) then
24:      if Elim-cycle-min-rew then
25:       s := Merged-node(s)
26:   else
27:    break
28:
29:   while visited ≠ Empty-Stack do
30:    s := visited.Pop()
31:    if ¬Check-solved(s, ε) then
32:     break
```

Alg. 2 Subroutines of GLRTDP and FRET

```
1:  proc GREEDY-ACTION(s: State)
2:   return argMinMax_{a∈A(s)} QValue(a, s)

3:  proc QVALUE(a: action, s: State)
4:   return
       Σ_{s'} P(s, a, s') · (R(s, a, s') + V(s'))

5:  proc UPDATE(s: State)
6:   a = Greedy-action(s)
7:   V(s) = QValue(a, s)

8:  proc PICK-NEXT(a: action, s: State)
9:   pick s' randomly from all successors with
       P(s, a, s') > 0
        ▷ originally with probability P(s, a, s')
10:  return s'

11: proc IS-CONS(s_old, s_new, ε: float)
12:  if abs(s_old − s_new) ≤ ε ||
        s_old = ∞ && s_new = ∞ then
13:   return True
14:  return False
```

This is possible because a value remains ε-consistent if its descendants' and its own value do not change by more than ε anymore (Alg. 3). This is because $V(s)$ can only change by more than ε if the greedy graph starting in s changes or the value of a descendant changes by more than ε. The graph can only change if the value of a state within the graph changes. Updating states outside the greedy graph will never make them part of it, because by the monotonicity property, updates according to the Bellman function can only make the states less attractive. Thus, a state's value can only change by more than ε if a descendant changes by more than ε but then it can not have been marked as solved before.

This algorithm converges faster than classical value iteration because not all states need to be converged or even updated. The termination criterion is similar to ε convergence in simple value iteration. If a cycle (zero-reward cycle in MDPs with rewards) occurs in a policy, it needs to be handled during the construction of trials in GLRTDP to guarantee convergence to an optimal value function. In the *MinProb* case permanent and transient traps have to be treated as dead-ends because in the worst case it is possible to always take an edge back to a state in the cycle instead of leaving the loop, i. e., $P_{\min}$ of eventually reaching the goal is 0. This is done indirectly by the termination criteria and the check before adding a new state (line 13 Alg. 1 and line 37 in Alg. 3). Because of the initialization with 0, values of trap states will lead to a cut immediately, because they never change their value in an update and stay ε-consistent, i. e., the cycle is not explored further and the algorithm concentrates on other branches.

To sum up, when calculating *MinProb* over an MDP, GLRTDP presented in Alg. 1 with an admissible initialization for this case and `Check-Solved()` as in Alg. 3 can be used. We will explain in the following why the combination of GLRTDP solves *MinProb* properties on general MDP structures correctly by converging to the optimal fixpoint. A formal proof can be found in Appendix A.

All greedy policies inspected by GLRTDP at some point end in a goal state or a dead-end state. This could be a real dead-end, i. e., a sink state with only a self-loop or a trap. Because of the initialization their value is already 0. In addition, we tag these states, do not explore them further and propagate their value back through the graph. Cycling forever is not possible because eventually all such cycles in greedy policies are eliminated. Having this, we can state that at some point no more states are left to explore in GLRTDP because all relevant traps are eliminated or a goal or a sink has been found. Then GLRTDP runs until the state values of the current greedy policy is converged up to ε. Even if the greedy policy is not the same in every iteration, at some point it will stay within a set of states which are part of finitely many policies. The values of these states converged close enough to the optimal ones such that the algorithm concentrates on these policies. The value function used in GLRTDP is initialized admissibly and therefore can only monotonically increase and approach the optimal result (fixpoint) from below. When this point is reached, the whole procedure terminates. This fixpoint has to be the only one and therefore has to be optimal because it has already been shown that the Bellman equation only has one fixpoint [7].

Alg. 3 Check-solved Procedure used in GLRTDP

```
1:  proc CHECK-SOLVED(s: State; ε: float)
2:   rv := True
3:   open := Empty-Stack
4:   closed := Empty-Stack
5:
6:   if ¬Solved(s) then open.Push(s)
7:
8:   while open ≠ Empty-Stack do
9:    s := open.Pop()
10:   closed.Push(s)
11:
12:   if Dead-end(s) || Goal(s) then con-
      tinue
13:
14:   a := Greedy-action(s)
15:   if max-rew || min-rew then
16:    check-∞-loop = False
17:    for each s' s.t. P(s, a, s') > 0 do
18:     if closed.Contains(s') then
19:      check-∞-loop = True
20:    if max-rew && check-∞-loop then
21:     if Elim-cycle-max-rew() then
22:      V(init-node) = ∞
23:      Solved(init-node) = True
24:      return True
25:    else
26:     if min-rew && check-∞-loop then
27:      if Elim-cycle-min-rew then
28:       return False
29:
30:   v_old = V(s)
31:   Update(s)
32:   v_new = V(s)
33:   if not Is-cons(v_old, v_new, ε) then
34:    rv = False
35:    continue
36:   for each s' s.t P(s, a, s') > 0 do
37:    if ¬Solved(s') &&
       ¬In(s', open ∪ closed) then
38:     open.Push(s')
39:
40:  if rv then
41:   for each s ∈ closed do
42:    Solved(s) := True
43:  else
44:   for s ∈ closed do
45:    Update(s)
46:  return rv
```

Alg. 4 Find, Revise, Eliminate Traps (FRET) (M is the graph of the MDP)

```
1:  proc FRET(M, s, V_0)
2:   V_i := V_0
3:   V'_i := GLRTDP(s, ε)
         ▷ originally Find-and-Revise(M, V_i)
4:   (V_{i+1}, elim-trap) :=
     Eliminate-Traps(M, V'_i)
5:   while elim-trap do
6:    V_i := V_{i+1}
7:    V'_i := GLRTDP(s, ε)
         ▷ originally Find-and-Revise(M, V_i)
8:    (V_{i+1}, elim-trap) :=
     Eliminate-Traps(M, V'_i)
```

Alg. 5 Eliminate-Traps (for *MaxProb*)

```
1:  proc ELIMINATE-TRAPS(M, V)
2:   elim-trap := False
3:   V_next := V
4:   G_V := {S_V, A_V} ← Vs greedy graph
5:   SCC := Tarjan(G_V)
6:   CSet := ∅
7:
8:   for each SComp C = {S_C, A_C} ∈ SCC
     do
9:    if ∄(s_i, s_j) ∈ A_G : (s_i ∈ S_C, s_j ∉ S_C)
      && (∄g ∈ G : g ∈ S_C) then
10:    CSet := CSet ∪ {C}
11:
12:  for each C = {S_C, A_C} ∈ CSet do
13:   if ∄a ∈ A, s ∈ S_C, s' ∉ S_C :
       T(s, a, s') > 0 then
14:    for each s ∈ S_C do
15:     V_next(s) := 0
16:    MergeSCC(C)
17:    elim-trap := True
18:   else
19:    A_e := {a ∈ A|∃s ∈ S_C; s' ∉ S_C :
       T(s, a, s') > 0}
20:    m := max_{s∈S_C, a∈A_e} Q^V(s, a)
21:    for each s ∈ S_C do
22:     V_next(s) = m
23:    MergeSCC(C)
24:    elim-trap := True
25:  return (V_next, elim-trap)
```

Maximum Reach Probability. For *MaxProb* properties, $P_{\max}(S_U \ \mathcal{U} \ \mathcal{S}_*)$, the objective is to find the maximal probability to reach a state in $\mathcal{S}_*$ if initialized in s_0 while avoiding $\overline{S_U}$. An admissible initialization is 1 except for states from which only dead-end states can be reached, which get a value of 0. $P_{\max}$ can be calculated by changing the initialization and replacing the occurrences of min by max in equations (3) and (4). In MODYSH, we use a combination of GLRTDP (Alg. 1, `max-rew`=`min-rew`=*False*) and a modified version of FRET (Alg. 4 and 5), adapted from the originals [31, Alg. 1] to calculate *MaxProb*. The combination is needed to guarantee convergence of GLRTDP for *MaxProb* [31,30]. In FRET iterations of GLRTDP followed by a call to `Eliminate-Traps()` to eliminate

zero-reward cycles are performed. In the original version, any Find-and-Revise algorithm is foreseen, we fix GLRTDP (Alg. 4, line 3 and 7) in our implementation. The call to `Eliminate-Traps()` (line 4 and 8) is needed if facing zero-reward cycles, because these may induce convergence of GLRTDP-trials to a non-optimal value by always choosing an action that loops on the cycle and thus the goal is never reached (line 14, 16 in Alg. 1). The trap elimination procedure changes the value function computed in the last iteration of GLRTDP and the graph it is working on, thus guaranteeing progress in its next call (Alg. 4, line 8). This is achieved by finding and eliminating *traps* (cf. Alg. 5). States which are part of a trap are merged into a single new state replacing all trap states.

In contrast to *MinProb*, where traps are handled directly during the trial construction, permanent and transient traps have to be handled differently here. All SCCs in the current greedy policy are collected using Tarjan's Algorithm [37] (Alg. 5, line 5) and it has to be checked if these SCCs are traps (line 8). First, permanent traps (line 13) are dead-ends from which the goal can never be reached. Therefore, all states' values in this SCC can be set to 0 (line 15) and the states of the SCC can be merged into one. If the SCC is a transient trap (line 19), it has to be left to reach the goal eventually. From all states in the SCC it is possible to take the exit with the highest probability value to reach the goal (line 20). Therefore, we merge these states and set the resulting state to this value (line 21). In the next GLRTDP trial this will change the greedy policy, i. e., the cycle is eliminated from the greedy graph. The algorithm terminates if the policy of the last GLRTDP run does not contain a trap anymore.

While the original version of FRET [31] considers in each trap elimination step *all* actions that are optimal according to the current value function, our implementation uses an optimization of Tarjan's algorithm (line 5), called FRET-π [36], considering in the subgraph of the state space inspected during trap elimination only those state transitions chosen into the current greedy policy.

To sum up, when calculating *MaxProb* over an MDP we call FRET, Alg. 4, with GLRTDP, Alg. 1, with an admissible initialization for this case. The trap elimination procedure in FRET is instantiated with Alg. 5. A formal proof of the correctness of this approach for general MDPs in the style of the proof for *MinProb* can be found in Appendix B.

3.2 Expected Reward Properties

Expected reward properties $E_{opt}(S_U \; \mathcal{U} \; \mathcal{S}_*)$, ask for the minimal or maximal (referred to by *opt*) expected accumulated reward when reaching a goal state. For the reachability properties considered thus far, we have been able to ignore the reward function of the MDP or more precisely, assumed it to be 0 except for actions leading to goal states. The calculation of E_{opt} proceeds very much in the same way. Iteratively a variation of the Bellman function updates is performed as presented in Eq. 1, where contrary to the P_{opt}-case (2) rewards are gained by taking a transition. The conceptual variation is that goal states initially get a value of 0 and states $s \in \mathcal{S}_\perp \cup \overline{S_U} \setminus \mathcal{S}_*$ a value of ∞.

Reward maximization. For E_{max} `max-rew` is set to *True*. A trivial admissible initialization is 0 for goal states and ∞ for all others. Because initializing non-goal states with ∞, i. e., the largest possible overapproximation, increases the runtime extremely, we approach an admissible initialization for non-dead-end states from below by starting with a smaller *maxValue*, obtained by *exponential search*. Dead-ends directly get a value of ∞. We execute full GLRTDP runs, as long as one of the final state values after termination is larger than the last *maxValue*, because if this happens, the initialization has not been admissible. In each iteration the new *maxValue* is set to the largest state value increased by 1 and multiplied by 2, which leads to the fastest solution we found in our experiments. Cycles again require a special treatment. Before adding the next state to the current trial (line 16, Alg. 1 and line 38, Alg. 3) it has to be checked if this state closes a SCC in the current greedy graph (independent of the reward accumulated in the SCC) (`Elim-cycle-max-rew()`: line 18, Alg. 1 and line 21 et seq., Alg. 3). If this is the case, the maximal expected reward for this property can directly be set to ∞ because in the worst case always this loop could be taken, i. e., the goal would never be reached.
Reward minimization. For $E_{\min}$ `min-rew` is set to *True* and the value function is initialized admissibly with ∞ for dead-ends and with 0 for all other states. Similar to the $E_{\max}$ case, when adding the next state to the current trial, it has to be checked if it closes a zero-reward SCC which has to be eliminated because it has to be left immediately to reach the goal with minimal reward (`Elim-cycle-min-rew()` in line 24, Alg. 1 and line 27, Alg. 3).

Correctness and optimality proofs for these property types are very similar to the proofs for *MaxProb* and *MinProb* spelled out in the appendices.

3.3 Bounded Properties

Reachability and expected reward properties can be extended by step or reward bounds. $P_{opt}(S_U \; \mathcal{U}_{[l,u]} \; \mathcal{S}_*)$ is the extremal probability of reaching a goal state in $[l, u]$ steps or with accumulated reward in $[l, u]$. Notably, and in contrast to the other properties considered thus far, for such bounded properties, memoryless policies can be outperformed by policies that are aware of the history regarding their past evolution, namely with respect to the number of steps taken or reward accumulated thus far. So, formally, we here work with a definition of policies that deviates form the one in Sect. 2 in that a policy can remember how many steps have already been made or what reward has been accumulated.

Let us first look at step-bounded properties. For those, in standard value iteration, updating all state values synchronously makes it possible to iterate only t times for properties with upper bound t [17] and then to extract a step-dependent policy. In heuristic search algorithms like FRET-LRTDP this is not possible because only the current greedy path is updated. In this case, a straightforward remedy is to encode a step counter in each state and consider all states for which the bounds regarding these counters are exceeded as dead-end. Formally, one works in a derived MDP where states are enriched with counters and where states differing in counter value are different and thus also the policy decision might

differ for them (implying history awareness with respect to the original MDP). States which fulfill the reachability property and whose bound-counter lies in the target interval are considered goal states. In our implementation we use the same variants of GLRTDP in combination with FRET like for the unbounded cases above and only add the bound and step counter to the state as described. For reward-bounded properties the basic strategy is the same, except that the counters are now replaced by real-valued variables. If the reward of the current policy exceeds the bound, the current state is considered as a dead-end. In either case (step or reward bounds), the derived MDP can be constructed in such a way that it is guaranteed to be finite-state (which is one of our early assumptions). Expected reward properties with bounds can be solved in a similar way. Since the overall procedures stay the same when adding bounds, the correctness and optimality proofs follow the respective same strategy.

Modysh is the only tool of the Modest Toolset which fully supports all variants of bounds w.r.t. step/reward bounds and interval types. All other tools do not treat step bounds at all and only support inclusive upper bounds.

4 Empirical Evaluation

Prototypical predecessor versions of Modysh with less functionality, implemented on a different, less performant code base and with strategies closer to the original version of FRET-LRTDP took part in QComp 2019 and 2020 [18,13], where the approach already showed promising results in comparison to other state-of-the-art model checkers. Since then, the new implementation approach presented in this work and several other optimizations implied a decrease in runtime of Modysh by a factor of nearly 1/3.

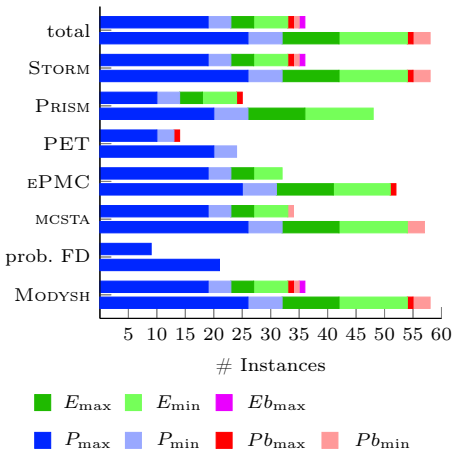


Fig. 1: Number of benchmark instances supported by tools per property type. (upper bars: QComp, lower: additional)

The benchmark set of QComp comprises, appart from other model types, 36 MDP instances. For evaluation purposes, we reran the experiments from QComp 2020 *default often ε-correct track*, i. e., with a precision of $\varepsilon = 10^{-3}$ and a timeout of 30 min, on an Intel Core i7-4790 CPU 3.60GHz with 32 GB RAM. With this setup we are able to show plots which are directly comparable to the evaluation of QComp and the performance improvements of Modysh are clearly visible. In addition, we added 58 *additional* benchmark instances from the quantitative verification benchmark set QVBS [24] to our case study to enlarge the number of MDP benchmarks and thereby also the number of minimum reach and bounded properties. Furthermore,

we wanted to test the tools on both smaller benchmarks, because many tools time out on the difficult QComp instances, as well as on considerably larger instances than in QComp, to demonstrate the capabilities and benefits of Modysh of only inspecting a fraction of the state space. Therefore, we scaled the models for the *israeli-jalfon*, *philosophers-mdp*, *pnueli-zuck*, *rabin* and *wlan* benchmarks up by parallelizing up to 100 automata for all of them except for wlan, for which 10 processes were already enough such that only Modysh was able to solve it. The QVBS contains only smaller instances of these benchmarks. For israeli-jalfon, the largest instance results in a state space size of $(2^{100}) - 1$, i. e., $1.268 \cdot 10^{30}$. For 100 dining philosophers the state space grows into the order of 10^{99} and for 100 parallel processes in pnueli-zuck and rabin it is in the order of 10^{100} and 10^{105}, respectively. 10 parallel senders in wlan result in a size of around $7 \cdot 10^8$ states. The number of benchmark instances supported by each tool per property type are listed in Figure 1. Since QComp 2020, Modysh added functionality for bounded properties and some special minimum reach cases.

Not all participating tools of QComp 2020 support MDP benchmarks. Therefore, we were not able to consider modes [11], STAMINA [34] and DFTRES [35]. But we added new results for Probabilistic Fast Downward [36], which took part in QComp 2019 but not in 2020. In addition, ePMC [20], mcsta [16,19,23] of the Modest Toolset [22], PET [10], Prism [33] and Storm [14] are part of our evaluation. We contacted the authors of all tools and asked for the newest version, i. e., improvements in other tools are also taken into account.

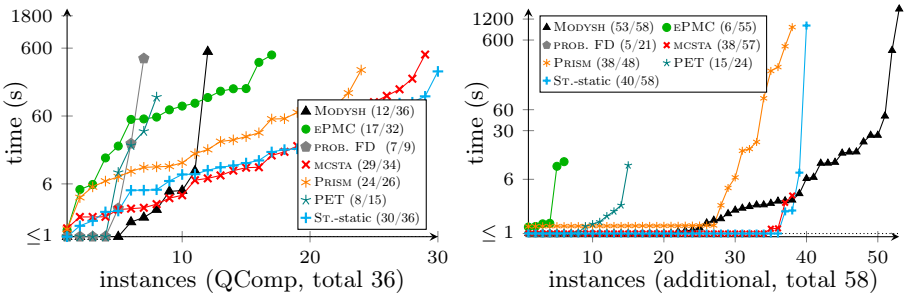


Fig. 2: Quantile plots for default tool versions in often ε-correct track.

In the quantile plots in Fig. 2 a point (x, y) indicates that the runtime of the xth fastest instance of the tool was y seconds. This allows comparing the overall performance of the tools. The benchmark instances are ordered independently for each tool depending on its runtime. The count of correctly solved benchmarks c (no timeout or error) and of supported instances s is given in the label as c/s. Modysh improved the runtime for many of the QComp instances (Fig. 2, left, contrastable with Fig. 4 bottom right in [13]) in comparison to QComp 2020 such that it is now among the best three tools for a large number of instances. The strength of Modysh is impressively demonstrated by the results on the

additional benchmark set on the right of Fig. 2. It clearly outperforms the other tools on the extremely large scaled benchmarks because only a small fraction of the state space needs to be visited. MODYSH is able to solve 7 benchmarks in less than 30s for which all other tools time out or do not have enough memory. For 5 other models only one other tool is able to solve them. For the largest instances of philosophers, pnueli-zuck, rabin and wlan only a few thousand states have to be visited in MODYSH and only $1.7 \cdot 10^3$ for israeli-jalfon. All these benchmarks have in common that they consist of the parallelization of automata of symmetric structure. The results on both benchmark sets show that MODYSH is clearly able to compete with state-of-the-art model checkers and on certain MDP structures it is even able to quickly solve instances which no other tool is able to handle.

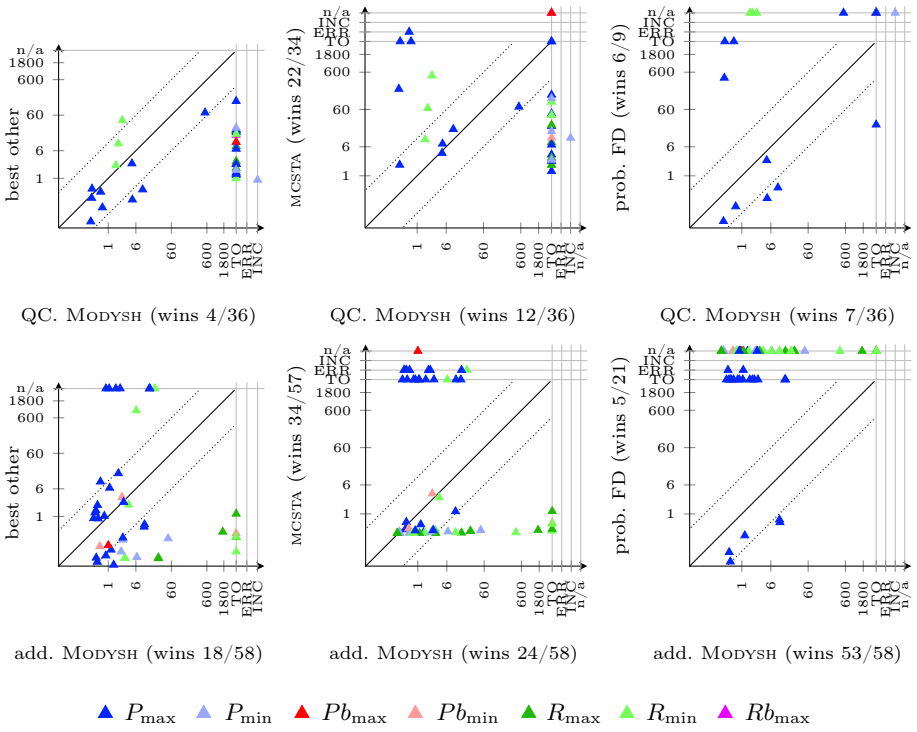


Fig. 3: Scatter plots (row 1: QComp, 2: additional benchmarks).

More detailed results can be inspected in Fig. 3 showing scatter plots comparing individual benchmark instances between two tools or a tool and the best of all other tools. A point (x, y) indicates a runtime of x seconds for the tool on the x-axis and a runtime of y seconds for the tool on the y-axis. This means, if the point lies above the diagonal line, the tool on the x-axis was

the fastest. If the point lies above the dotted line, it was more than ten times faster. "TO", "ERR" and "INC" mean timeout, error, e. g., out of memory, and incorrect result, respectively. "n/a" means that the tool is unable to handle the benchmark instance. The number of benchmark instances on which the tool on the x-axis outperformed the tool(s) on the y-axis is given in parenthesis in the label. By the evaluation setup, the upper left plot is a direct update of [13, Fig. 8, middle]. We see that Modysh is able to compete with the other tools especially on the additional benchmark set for which the results are depicted in the lower row. It solves way more instances and property types than probabilistic Fast Downward (right column), which is based on the same algorithms. It also supports more properties than mcsta (upper row, middle), i. e., improves the range of the Modest Toolset and shows better performances on many instances, especially where mcsta (lower row, middle) or various other tools (lower row, left) are not able to deliver results at all. This demonstrates the potential of the methods implemented in Modysh because first, it improves the model checking performance of the Modest Toolset in comparison to mcsta on the same code base. Second, integrating these techniques specifically in Storm looks promising. If Modysh was dominated by a competitor, e. g., on the QComp benchmarks (upper left), it was often outperformed by Storm. From QComp 2020 it is already known that Storm's code base is highly efficient and the performance is currently out of reach for other model checkers on most of the benchmarks. Implementing our approach in Storm would boost its performance even more.

Interactive result tables which enable a direct runtime comparison across benchmark instances are available online for the QComp benchmarks and for the additional QVBS benchmarks. Furthermore, an artifact enabling the reproduction of all empirical results reported in this paper is available online [29].

5 Related Work

As already described in Sect. 3, our algorithms are generalizations of well-known approaches used in the planning community for the purpose of cost-optimal planning. Of course, ideas behind heuristic search have already been used in model checking. We highlight the parallels but also the differences to our work.
Probabilistic Planning and Heuristic Search. A variant of FRET-LRTDP is available in the probabilistic version of Fast Downward [26] which is one of the classical progression planning systems based on heuristic search. It has been extended by Steinmetz et al. [36] for goal probability analysis, i. e., computing the maximal probability to reach a goal. That extension also encompasses several heuristic search algorithms like LRTDP with FRET-π.

The original LRTDP work by Bonet et al. [8] is tailored to $\mathcal{SSP}$ assuming conditions (i′) and (ii′), the second version of Bertsekas [7], with strictly positive action rewards (except at goal states). Kolobov et al. [31,30] instead uses (i) and (ii″) when discussing $\mathcal{GSSP}$ problems. They showed that several MDP problems, including *MaxProb*, can be reduced to this problem class [31,30] and that the

respective properties can be solved using FRET with LRTDP. We do not need to assume any of these, but restrict to non-negative reward structures.

Probabilistic Model Checking. This is not the first work to explore probabilistic planning and heuristic search approaches for probabilistic model checking. For instance, heuristic search dynamic programming methods have been applied to MDPs, but for generating probabilistic counterexamples [1]. Closer to our work, Kretinsky et al. developed heuristics for initializing policies in policy iteration such that the computation time to solve long-run average reward properties on MDPs is reduced [32], with specific treatments of SSCs and maximal end components similar to the approach of Modysh. The PAC tool [3] uses asynchronous bounded value iteration techniques interleaved with guided simulation phases with permanent and transient trap elimination for statistical model checking for reachability analysis on stochastic games. A combination of Bounded Real-Time Dynamic Programming (BRTDP) and Monte Carlo Tree Search has been devised with objectives similar to ours [2]. Technical differences aside, this approach has only been applied to solve *MaxProb* properties.

Machine learning techniques have been exploited [10] to verify reachability properties on MDPs using (1) BRTDP and (2) delayed Q-learning for MDPs with limited information. The techniques are also applicable to arbitrary MDP structures due to special treatments of end components and are implemented in PET (aka. Prism-TUM), which is part of our evaluation in Sect. 4. In parts, the approach is close to ours for simple reachability properties, but restricted to that, and uses BRTDP instead of FRET-LRTDP. The paper explicitly mentions that so far no attempts have been made to adapt these methods in the context of probabilistic verification. With Modysh we completely fill this gap.

As became clear in our empirical evaluation, heuristic search can be especially attractive for handling excessively large models. An entirely different approach to attack such problems is the use of external storage to slowly but exhaustively model check problem sizes that otherwise do not fit in memory [23].

6 Conclusion

We introduced a heuristic approach to probabilistic model checking all established property types, except long-run averages, on general MDP structures based on LRTDP combined with FRET. The approach is implemented in Modysh. We reported on a large empirical evaluation that has demonstrated the competitiveness of Modysh relative to other state-of-the-art model checking tools. On very large state spaces our tool outperforms its competitors, demonstrating that planning techniques can indeed be used to enhance the performance and capabilities of model checkers.

As a next step we are looking into performance optimizations by exploring the trade-offs between memory usage and runtime. In addition, other heuristics known to work well in the planning community might be worth to implement. Extending the approach to work on other automata types seems also promising.

A Proof for *MinProb*

As announced in Sect. 3.1, this appendix provides a proof that GLRTDP solves *MinProb* properties on general MDP structures correctly by converging to the optimal fixpoint.

To show convergence to the optimal value function from below in case of an admissible initialization, we can argue along the invariant

$$\forall k, \sigma : V_k(s) \leq P_s^\sigma(\diamond G), \text{ where } \sigma \text{ s.t. } P_s^\sigma(\diamond G) = V^*(s)$$

stating that the value function in every iteration is always at most the value under the optimal policy. This means that an initially admissible value function always stays admissible. This is true for the admissible initialization when $k = 0$, because then $V_0(s) = 1$ if $s \in G$ and 0 otherwise. For all other iterations it holds that $V_{k+1}(s) \coloneqq \sum_{s'} P(s, a, s') \cdot V_k(s')$ for some action a and we can derive that

$$\sum_{s'} P(s, a, s') \cdot V_k(s') \leq \sum_{s'} P(s, a, s') \cdot \min_\sigma P_{s'}^\sigma(\diamond G)$$
$$\leq \sum_{s'} \min_\sigma (P(s, a, s') \cdot P_{s'}^\sigma(\diamond G)) \leq \min_\sigma \sum_{s'} P(s, a, s') \cdot P_{s'}^\sigma(\diamond G).$$

The second inequality holds because σ_{opt} is memoryless and independent of s'. Now assume σ_{opt} is such that $P_s^{\sigma_{opt}}(\diamond G)$ is minimal for all s. Then for action $a = \mathit{greedy}(s, V_k)$ we have for any action b, and in particular for $b = \sigma_{opt}(s)$,

$$\sum_{s'} P(s, a, s') \cdot V_k(s') \leq \sum_{s'} P(s, b, s') \cdot V_k(s').$$

Moreover $V_k(s') \leq P_{s'}^{\sigma_{opt}}(\diamond G)$, which allows us to derive

$$\sum_{s'} P(s, a, s') \cdot V_k(s') \leq \sum_{s'} P(s, \sigma_{opt}(s), s') \cdot P_{s'}^{\sigma_{opt}}(\diamond G) = P_s^{\sigma_{opt}}(\diamond G).$$

Claim: If V_k is a fixpoint for $k \to \infty$ then $P^\sigma(\diamond G) = V_\infty(s_0)$ $\forall \sigma$ greedy in V. (5)
Since $V^*(s) \coloneqq \min_\sigma P_s^\sigma(\diamond G)$ this means $V^*(s_0) \leq V_\infty(s_0)$ and with the result from above ($\forall k : V_k \leq V^*$) we can conclude $V^*(s_0) = V_\infty(s_0)$.

It remains to show that (5) holds: Let $\sigma_k \coloneqq \mathit{greedy}(V_k)$, i. e., a greedy policy with respect to the value function V_k and $S_k = \{s | P_{s_0}^{\sigma_k}(\diamond s) > 0\}$, i. e., all states reachable with this greedy policy, then $\max(\mathit{residual}(S_k)) \leq \delta_k$ and for $k \to \infty$ it holds that $\delta_k \to 0$.

To show that δ_k will approach 0 it is enough to argue about the states which will be updated an infinite number of times, i. e., in the end, about the states on optimal policies. These are the states in $S_\infty = \bigcap_{i \geq 0} \bigcup_{k \geq i} S_k$.

Let K be such that $\forall k \geq K : \bigcup_{i \geq k} S_i = S_\infty$, i. e. a step from which on we only consider states which will be infinitely often visited when running GLRTDP infinitely long. Assume we are in a step $j + 1 \geq K$. Let $s \in S_\infty$. We have to distinguish two cases:

- If s has not been updated then $V_{j+1}(s) = V_j(s)$.
- If s is the updated state then $V_{j+1}(s) = \min_\alpha \sum_{s'} P(s, \alpha, s') \cdot V_j(s')$

But this is the same as for simple synchronous value iteration, for which convergence against the optimal fixpoint is proven. For our asynchronous case in GLRTDP we nevertheless have to guarantee fairness among the states in S_∞, i. e., we have to make sure that they are updated infinitely often. This is the case because each possible trial of S_∞ (there are finitely many trials) appears infinitely often, i. e. the states in this trial are updated infinitely often (by construction of GLRTDP when choosing the next greedy action). All other states not in S_∞ can be ignored because they will not influence the greedy policy and optimal values because they are already too large:

For any $s \in S \setminus S_\infty$ it holds that $V_\infty(s) = V_K(s) \leq V^*(s)$ and for any $s \in S_\infty$ by definition of S_∞ and K we know that an action leading again to a state in S_∞ will be chosen, i. e., an $a \in \sigma_\infty$: $V_\infty(s) \leq \sum_{s' \in S_\infty} P(s, \sigma_\infty, s') \cdot V_\infty(s')$ but for every action we choose the greedy one and for any $k \geq K$ it holds that $V_k(s) \leq \sum_{s' \in S} P(s, a, s') \cdot V_k(s') \leq V_\infty(s)$, i. e., the action in σ_∞ must have been the greedy action not leading to $S \setminus S_\infty$. This means that V_∞ defines an optimal strategy on S_∞ for $s_0 \in S_\infty$ which is also an optimal strategy on S because no state $s' \in S \setminus S_\infty$ is visited even with $V_\infty(s') < V^*(s')$. In addition the initial state lies in S_∞ by construction, i. e., $P_{\min}(\diamond G) = V^{\sigma_{opt}}(s_0)$ reaches the fixpoint and is updated infinitely often.

In summary, when running GLRTDP in an infinite number of iterations, the value function for states in S_∞ will approach the optimal values of the minimal probability to reach the goal from below, will never get larger than the optimal value and the difference between V and V^* always becomes strictly smaller for these states. In addition, we can at some point stop updating the value function for parts of the state space because these values will not have an influence on the correct optimal result for the initial state. In our implementation GLRTDP is designed in such a way that it stops when the values on the optimal policy only change by less than ε, which is the same convergence criterion as for simple value iteration.

B Proof for *MaxProb*

Taking up our promise from Sect. 3.1, in the following we will first give an intuition about why the presented combination of GLRTDP and FRET solves *MaxProb* properties on general MDP structures correctly, not only on problems having at least one almost-sure policy as proven in [31], by converging to the optimal fixpoint. Afterwards we sketch a more formal proof.

All greedy policies inspected by GLRTDP at some point end in a goal state or a dead-end state. This could be a real dead-end, i. e., a sink state with only a self-loop or a permanent trap which has been transformed to a dead-end by the cycle elimination of FRET. If it is a permanent trap identified by FRET, the values of all states in it are set to 0. Otherwise, when the sink state is discovered for the first time its value is also directly set to 0. This means we tag these

states, do not explore them further and propagate their value back through the graph. Cycling forever is not possible because FRET eventually eliminates all such cycles in greedy policies. With this, we can state that at some point no more states are left to explore in the current GLRTDP trial because all relevant traps are eliminated or a goal or a sink has been found. Then GLRTDP runs until the state values of the current greedy policies are converged up to ε. Even if the greedy policy is not the same in every iteration, at some point it will stay within a set of greedy states which are part of finitely many greedy policies. The values of these states will have converged close enough to the optimal ones such that the algorithm concentrates on these optimal policies. The value function used in GLRTDP is initialized admissibly and therefore can only monotonically decrease and approach the optimal fixpoint from above. When this point is reached (up to ε), the entire procedure (GLRTDP + FRET) terminates. This fixpoint must be the optimal one because the Bellman equation only admits a single fixpoint [7].

To show convergence to the optimal value function from above in case of an admissible initialization, we can argue along the invariant

$$\forall k, \sigma : V_k(s) \geq P_s^\sigma(\diamond G), \text{ where } \sigma \text{ s.t. } P_s^\sigma(\diamond G) = V^*(s)$$

stating that the value function in every iteration is always greater or equal than the optimal value under the optimal policy. This means that an initially admissible value function always stays admissible. This is true for the admissible initialization when $k = 0$, because then $V_0(s) = 0$ if $s \in \mathcal{S}_\perp$ and 1 otherwise. For all other iterations it holds that

$$V_{k+1}(s) \coloneqq \sum_{s'} P(s, a, s') \cdot V_k(s')$$

for some action a and we can derive that

$$\sum_{s'} P(s, a, s') \cdot V_k(s') \geq \sum_{s'} P(s, a, s') \cdot \max_\sigma P_{s'}^\sigma(\diamond G)$$
$$\geq \sum_{s'} \max_\sigma (P(s, a, s') \cdot P_{s'}^\sigma(\diamond G)) \geq \max_\sigma \sum_{s'} P(s, a, s') \cdot P_{s'}^\sigma(\diamond G)$$

The second inequality holds because σ_{opt} is memoryless and independent of s'.

Now assume σ_{opt} is such that $P_s^{\sigma_{opt}}(\diamond G)$ is maximal for all s. Then for action $a = \mathit{greedy}(s, V_k)$ we have for any action b, and in particular for $b = \sigma_{opt}(s)$,

$$\sum_{s'} P(s, a, s') \cdot V_k(s') \geq \sum_{s'} P(s, b, s') \cdot V_k(s').$$

Moreover $V_k(s') \geq P_{s'}^{\sigma_{opt}}(\diamond G)$ and hence

$$\sum_{s'} P(s, a, s') \cdot V_k(s') \geq \sum_{s'} P(s, \sigma_{opt}(s), s') \cdot P_{s'}^{\sigma_{opt}}(\diamond G) = P_s^{\sigma_{opt}}(\diamond G).$$

Claim: If V_k is a fixpoint for $k \to \infty$ then $P^\sigma(\diamond G) = V_\infty(s_0)\ \forall \sigma$ greedy in V. (6)
Since $V^*(s) \coloneqq \max_\sigma P_s^\sigma(\diamond G)$ this means $V^*(s_0) \geq V_\infty(s_0)$ and with the result from above ($\forall k : V_k \geq V^*$) we can conclude $V^*(s_0) = V_\infty(s_0)$.

It remains to show that (6) holds: Let $\sigma_k := greedy(V_k)$, i. e., a greedy policy with respect to the value function V_k and $S_k = \{s | P^{\sigma_k}_{s_0}(\diamond s) > 0\}$, i. e., all states reachable with this greedy policy, then $\max(residual(S_k)) \leq \delta_k$ and for $k \to \infty$ it holds that $\delta_k \to 0$.
To show that δ_k will approach 0 it is enough to argue about the states which will be updated an infinite number of times, i. e., in the end, about the states on optimal policies. These are the states in $S_\infty = \bigcap_{i \geq 0} \bigcup_{k \geq i} S_k$.
Let K be such that $\forall k \geq K : \bigcup_{i \geq k} S_i = S_\infty$, i. e. a step from which on we only consider states which will be infinitely often visited when running FRET-LRTDP infinitely long. Assume we are in a step $j + 1 \geq K$. Let $s \in S_\infty$. We have to distinguish two cases:

- If s has not been updated then $V_{j+1}(s) = V_j(s)$.
- If s is the updated state then $V_{j+1}(s) = \max_\alpha \sum_{s'} P(s, \alpha, s') \cdot V_j(s')$

This is the same as for simple synchronous value iteration, for which convergence against the optimal fixpoint is proven. For our asynchronous case in GLRTDP we are left with the duty to guarantee fairness among the states in S_∞, i. e., we have to make sure that they are updated infinitely often. This is the case because each possible trial of S_∞ (there are finitely many trials) appears infinitely often, i. e., the states in this trial are updated infinitely often (by construction of GLRTDP when choosing the next greedy action). All other states not in S_∞ can be ignored because they will not influence the greedy policy and optimal values because they are already too large:

For any $s \in S \setminus S_\infty$ it holds that $V_\infty(s) = V_K(s) \geq V^*(s)$ and for any $s \in S_\infty$ by definition of S_∞ and K we know that an action leading again to a state in S_∞ will be chosen, i. e., an $a \in \sigma_\infty$: $V_\infty(s) \geq \sum_{s' \in S_\infty} P(s, \sigma_\infty, s') \cdot V_\infty(s')$ but for every action we choose the greedy one and for any $k \geq K$ it holds that $V_k(s) \geq \sum_{s' \in S} P(s, a, s') \cdot V_k(s') \geq V_\infty(s)$, i. e., the action in σ_∞ must have been the greedy action not leading to $S \setminus S_\infty$.

This means that V_∞ defines an optimal strategy on S_∞ for $s_0 \in S_\infty$ which is also an optimal strategy on S because no state $s' \in S \setminus S_\infty$ is visited even with $V_\infty(s') > V^*(s')$.

In addition the initial state lies in S_∞ by construction, i. e., $P_{\max}(\diamond G) = V^{\sigma_{opt}}(s_0)$ reaches the fixpoint and is updated infinitely often.

Altogether, this shows that when running FRET-LRTDP over an infinite number of iterations, the value function for states in S_∞ will approach the optimal values of the maximal probability to reach the goal from above, will never get smaller than the optimal value and the difference between V and V^* always becomes strictly smaller for these states. In addition, we can at some point stop updating the value function for parts of the state space because these values will not have an influence on the correct optimal result for the initial state. In our implementation FRET-LRTDP is designed in such a way that it stops when the values on the optimal policy only change by less than ε, which is the same convergence criterion as for simple value iteration.

References

1. Aljazzar, H., Leue, S.: Generation of counterexamples for model checking of Markov decision processes. In: QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems, Budapest, Hungary, 13-16 September 2009. pp. 197–206. IEEE Computer Society (2009). https://doi.org/10.1109/QEST.2009.10, https://ieeexplore.ieee.org/xpl/conhome/5290656/proceeding
2. Ashok, P., Brázdil, T., Kretínský, J., Slámecka, O.: Monte carlo tree search for verifying reachability in Markov decision processes. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11245, pp. 322–335. Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_21
3. Ashok, P., Kretínský, J., Weininger, M.: PAC statistical model checking for Markov decision processes and stochastic games. In: Dillig and Tasiran [15], pp. 497–519. https://doi.org/10.1007/978-3-030-25540-4_29
4. Barto, A.G., Bradtke, S.J., Singh, S.P.: Learning to act using real-time dynamic programming. Artif. Intell. **72**(1-2), 81–138 (1995). https://doi.org/10.1016/0004-3702(94)00011-O
5. Bellman, R.: A Markovian decision process. Journal of mathematics and mechanics **6**(5), 679–684 (1957)
6. Bertsekas, D.P.: Dynamic Programming and Optimal Control, Vol. 1. Athena Scientific (1995)
7. Bertsekas, D.P.: Dynamic Programming and Optimal Control, Vol. 2. Athena Scientific (1995)
8. Bonet, B., Geffner, H.: Labeled RTDP: improving the convergence of real-time dynamic programming. In: Giunchiglia, E., Muscettola, N., Nau, D.S. (eds.) Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003), June 9-13, 2003, Trento, Italy. pp. 12–21. AAAI (2003), http://www.aaai.org/Library/ICAPS/2003/icaps03-002.php
9. Bonet, B., Geffner, H.: Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs. In: Long, D., Smith, S.F., Borrajo, D., McCluskey, L. (eds.) Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006. pp. 142–151. AAAI (2006), http://www.aaai.org/Library/ICAPS/2006/icaps06-015.php
10. Brázdil, T., Chatterjee, K., Chmelik, M., Forejt, V., Kretínský, J., Kwiatkowska, M.Z., Parker, D., Ujma, M.: Verification of Markov decision processes using learning algorithms. In: Cassez, F., Raskin, J. (eds.) Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8837, pp. 98–114. Springer (2014). https://doi.org/10.1007/978-3-319-11936-6_8, https://doi.org/10.1007/978-3-319-11936-6
11. Budde, C.E., D'Argenio, P.R., Hartmanns, A., Sedwards, S.: An efficient statistical model checker for nondeterminism and rare events. Int. J. Softw. Tools Technol. Transf. **22**(6), 759–780 (2020). https://doi.org/10.1007/s10009-020-00563-2
12. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: Quantitative model and tool interaction. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on

Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10206, pp. 151–168 (2017). https://doi.org/10.1007/978-3-662-54580-5_9
13. Budde, C.E., Hartmanns, A., Klauck, M., Kretinsky, J., Parker, D., Quatmann, T., Turrini, A., Zhang, Z.: On Correctness, Precision, and Performance in Quantitative Verification (QComp 2020 Competition Report). In: Proceedings of the 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Software Verification Tools. (2020)
14. Dehnert, C., Junges, S., Katoen, J., Volk, M.: A storm is coming: A modern probabilistic model checker. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10427, pp. 592–600. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_31
15. Dillig, I., Tasiran, S. (eds.): Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I, Lecture Notes in Computer Science, vol. 11561. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4
16. Hahn, E.M., Hartmanns, A.: A comparison of time- and reward-bounded probabilistic model checking techniques. In: Fränzle, M., Kapur, D., Zhan, N. (eds.) 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, pp. 85–100 (2016). https://doi.org/10.1007/978-3-319-47677-3_6
17. Hahn, E.M., Hartmanns, A.: Efficient algorithms for time- and cost-bounded probabilistic model checking. CoRR **abs/1605.05551** (2016), `http://arxiv.org/abs/1605.05551`
18. Hahn, E.M., Hartmanns, A., Hensel, C., Klauck, M., Klein, J., Kretínský, J., Parker, D., Quatmann, T., Ruijters, E., Steinmetz, M.: The 2019 comparison of tools for the analysis of quantitative formal models - (QComp 2019 competition report). In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11429, pp. 69–92. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_5
19. Hahn, E.M., Hartmanns, A., Hermanns, H.: Reachability and reward checking for stochastic timed automata. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **70** (2014). https://doi.org/10.14279/tuj.eceasst.70.968
20. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: iscasMc: A web-based probabilistic model checker. In: Jones, C.B., Pihlajasaari, P., Sun, J. (eds.) FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8442, pp. 312–317. Springer (2014). https://doi.org/10.1007/978-3-319-06410-9_22, `https://doi.org/10.1007/978-3-319-06410-9`
21. Hansen, E.A., Zilberstein, S.: Lao*: A heuristic search algorithm that finds solutions with loops. Artif. Intell. **129**(1-2), 35–62 (2001). https://doi.org/10.1016/S0004-3702(01)00106-0
22. Hartmanns, A., Hermanns, H.: The Modest Toolset: An integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) 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, pp. 593–598. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_51

23. Hartmanns, A., Hermanns, H.: Explicit model checking of very large MDP using partitioning and secondary storage. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9364, pp. 131–147. Springer (2015). https://doi.org/10.1007/978-3-319-24953-7_10
24. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The Quantitative Verification Benchmark Set. In: Vojnar, T., Zhang, L. (eds.) 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, pp. 344–350. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_20
25. Hatefi-Ardakani, H.: Finite horizon analysis of Markov automata. Ph.D. thesis, Saarland University, Germany (2017), http://scidok.sulb.uni-saarland.de/volltexte/2017/6743/
26. Helmert, M.: The fast downward planning system. CoRR **abs/1109.6051** (2011), http://arxiv.org/abs/1109.6051
27. Izadi, M.T.: Sequential decision making under uncertainty. In: Zucker, J., Saitta, L. (eds.) Abstraction, Reformulation and Approximation, 6th International Symposium, SARA 2005, Airth Castle, Scotland, UK, July 26-29, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3607, pp. 360–361. Springer (2005). https://doi.org/10.1007/11527862_33
28. The JANI specification. http://www.jani-spec.org/, accessed on 25/06/2021
29. Klauck, M., Hermanns, H.: Artifact accompanying the paper "A Modest Approach to Dynamic Heuristic Search in Probabilistic Model Checking" (2021), available at http://doi.org/10.5281/zenodo.4922360
30. Kolobov, A.: Scalable methods and expressive models for planning under uncertainty. Ph.D. thesis, University of Washington (2013)
31. Kolobov, A., Mausam, Weld, D.S., Geffner, H.: Heuristic search for generalized stochastic shortest path mdps. In: Bacchus, F., Domshlak, C., Edelkamp, S., Helmert, M. (eds.) Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011. AAAI (2011), http://aaai.org/ocs/index.php/ICAPS/ICAPS11/paper/view/2682
32. Kretínský, J., Meggendorfer, T.: Efficient strategy iteration for mean payoff in Markov decision processes. In: D'Souza, D., Kumar, K.N. (eds.) Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10482, pp. 380–399. Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_25, https://doi.org/10.1007/978-3-319-68167-2
33. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_47, https://doi.org/10.1007/978-3-642-22110-1

34. Neupane, T., Myers, C.J., Madsen, C., Zheng, H., Zhang, Z.: STAMINA: stochastic approximate model-checker for infinite-state analysis. In: Dillig and Tasiran [15], pp. 540–549. https://doi.org/10.1007/978-3-030-25540-4_31
35. Ruijters, E., Reijsbergen, D., de Boer, P., Stoelinga, M.: Rare event simulation for dynamic fault trees. Reliab. Eng. Syst. Saf. **186**, 220–231 (2019). https://doi.org/10.1016/j.ress.2019.02.004
36. Steinmetz, M., Hoffmann, J., Buffet, O.: Goal probability analysis in probabilistic planning: Exploring and enhancing the state of the art. J. Artif. Intell. Res. **57**, 229–271 (2016). https://doi.org/10.1613/jair.5153
37. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972). https://doi.org/10.1137/0201010