Compiling Probabilistic Model Checking into Probabilistic Planning

Michaela Klauck, Marcel Steinmetz, Jörg Hoffmann, Holger Hermanns
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany

Abstract

It has previously been observed that the verification of safety properties in deterministic model-checking frameworks can be compiled into classical planning. A similar connection exists between goal probability analysis on either side, yet that connection has not been explored. We fill that gap with a translation from Jani, an input language for quantitative model checkers including the Modest toolset and PRISM, into PPDDL. Our experiments motivate further cross-fertilization between both research areas, specifically the exchange of algorithms. Our study also initiates the creation of new benchmarks for goal probability analysis.

Introduction

Previous works have explored connections between qualitative model-checking and classical planning (Edelkamp 2003). A similar connection exists for probabilistic models. Here, we introduce a compilation from Jani (Budde et al. 2017), an input language for quantitative model-checkers, into PPDDL (Younes et al. 2005). We complement the compilation by an empirical comparison of methods used in model-checking – variants of value iteration (VI) – with the heuristic search algorithms developed by the AI community.

Tools from probabilistic model-checking have become very popular in the robotics and motion planning communities due to their support of expressive formal modeling languages, e. g., (Johnson and Kress-Gazit 2011; Lacerda, Parker, and Hawes 2015). Recent works in decision-theoretic planning started to use temporal logics to encode history-dependent reward functions (Camacho et al. 2017; Brafman, Giacomo, and Patrizi 2018). Also they introduced compilations back to standard formalisms, allowing to use well-established planning algorithms for Markov decision processes (MDP). Teichteil-Königsbuch (2012) has presented an overarching framework connecting decision-theoretic planning with formalisms from model-checking. The result is a very general class of problems, which however falls out of scope of existing algorithms. Despite those works, a direct connection between models and algorithms in probabilistic model-checking and that in probabilistic planning has so far not been explored. We start to close that gap through our compilation and accompanied experiments.

Jani is a powerful language that can express models of distributed and concurrent systems in the form of networks of automata decorated with variables, clocks and probabilities. A large spectrum of case studies exists. In full generality, Jani models are networks of stochastic timed automata. But the core formalism are MDPs, as in PPDDL probabilistic planning. We consider a relevant fragment of Jani that allows for structure-preserving translation into PPDDL.

We focus on the verification of safety properties, which translates in PPDDL to goal probability analysis. In this class of MDPs, heuristic search algorithms like HDP (Bonet and Geffner 2003) cannot be run as-is, but require an outer loop of iterations known as the FRET framework: find, revise, eliminate traps (Kolobov et al. 2011). Traps are sets of states that are closed under probabilistic branching in the subgraph of the state space induced by the current value function approximation v. While FRET as per Kolobov et al. considers in this subgraph *all* actions that are optimal according to v, Steinmetz et al. (2016) devised a variant which considers only the actions chosen by the greedy policy π. We will refer to these FRET variants as FRET-v and FRET-π respectively. I-Dual is a recently introduced heuristic search algorithm which does not require the FRET outer loop (Trevizan, Teichteil-Königsbuch, and Thiébaux 2017). In essence, this algorithm interleaves linear program (LP) evaluations with state space exploration. The LPs solved are encodings of goal probability MDPs (Altman 1999), restricted to the part of the state space explored so far. By considering in the exploration step only states touched by the last LP solution, one can often find a solution of the MDP while considering only a small fraction of the state space.

In our experiments, we compare PRISM (Kwiatkowska, Norman, and Parker 2011), Modest (Hartmanns and Hermanns 2014), and Fast Downward (Helmert 2006) based on two landmark problems in quantitative model-checking: the dining cryptographers protocol (DCP) and the randomized consensus shared coin protocol (RCSCP). It turns out that FRET-π excels in DCP, being surpassed only by PRISM's symbolic engine. In RCSCP heuristic search is inferior to VI. Nevertheless, with the help of a heuristic identifying 0 goal probability states, FD's VI achieves better scaling. Beyond this case study, our research motivates further cross-fertilization, pertaining to the exchange of algorithms. Our study also initiates the creation of new benchmarks for goal probability analysis. A TR provides more details on the compilation and the experiments (Klauck et al. 2018).

Background

Probabilistic PDDL (PPDDL) and FD. PPDDL extends PDDL with the possibility to define probabilistic action effects, i.e., a probability distribution over multiple possible outcomes. Fast Downward (FD) is a wide-spread code base in classical planning. It has been extended by Steinmetz et al. (2016) for goal probability analysis. That extension encompasses topological value iteration (TVI) (Dai et al. 2011), along with several heuristic search algorithms of which here we consider HDP along with FRET-v and FRET-π. We have extended their code by an implementation of the I-Dual algorithm (Trevizan, Teichteil-Königsbuch, and Thiébaux 2017). There are no inherently probabilistic heuristic functions in FD yet. As suggested by Bonet and Geffner (2005), we use the all-outcomes determinization with classical-planning heuristic functions. In our context, this comes down to goal probability estimate 0 for detected dead-ends, and estimate 1 elsewhere. We use $h^{\max}$ (Bonet and Geffner 2001) which provides strong dead-end detection capabilities at a comparatively small computational cost.

The Modest toolset (MT) and PRISM. Probably the most widespread probabilistic model checker is PRISM (Kwiatkowska, Norman, and Parker 2011). MT's core MDP-algorithms are known to be competitive PRISM (Hahn and Hartmanns 2016). Both support the analysis of hybrid, real-time, distributed and stochastic systems. PRISM offers multiple model-checking engines, including variants of VI based on an *explicit* state space representation as well as a *symbolic* variant. In a nutshell, the symbolic version represents the current value function estimation and the transition probabilities as multi-terminal binary decision diagrams (MTBDD). This data structure has builtin support of arithmetic operations, allowing to directly express value updates as MTBDD operations. Furthermore, PRISM optionally identifies states with goal probability 0, respectively 1 in a preprocessing step, which can then be filtered out inside VI. MT provides several components to solve various forms of quantitative model-checking problems. For our MDP setting, we consider one such component, called Mcsta. Mcsta provides MT's variants of value iteration, featuring a preprocessing step similar to PRISM. An external-memory VI (EVI) variant caters for MDPs whose state space is too large to fit into main memory (Hartmanns and Hermanns 2015).

Jani. Jani is an overarching language conceived to foster verification tool interoperation and comparability. It allows to model a rich variety of quantitative automata networks with variable decorations. Properties to be checked are temporal formulas based on computation tree logic (CTL).

Each automaton has locations connected by directed edges. The edges may be taken with a given probability provided the edge's *guard* is satisfied. Then an effect may modify the variable values, and a new location is occupied.

Jani allows variables of various types (e.g. *int* and *bool*). They can be restricted to a finite range. Expressions over these variables support many standard arithmetic operations,

```
"automata": [ {"name":"aut1",
 "locations": [ {"name":"loc0"}, {"name":"loc1"} ],
 "initial−location": ["loc0"],
 "edges": [ {  "location":"loc0",
    "guard": { "exp": {"op":"=", "left":"coin1", "right":0} },
    "destinations": [
     { "probability": {"exp":0.5},  "location":"loc1",
       "assignments": [ { "ref":"coin1", "value":1 } ] },
     { "probability": {"exp":0.5},  "location":"loc0",
       "assignments": [ { "ref":"coin1", "value":0 } ] } ]}]}]
```

Figure 1: A Jani specification example.

as well as conjunction and disjunction. The possible initial variable values are specified via a *restrict-initial* expression.

A Jani file lists the occurring constants (objects), and the variable definitions followed by the *restrict-initial* block. This preamble is followed by a list of automata along with their locations and edges.

An example snippet of Jani is given in Figure 1. It specifies an automaton called *aut1*, with two locations *loc0* and *loc1*. From the initial location *loc0*, there are two edges that can be taken provided the variable *coin1* has value 0, as expressed by the guard. In this case, the edges lead to different destinations, each weighted by a probability and involving an assignment changing the value of variable *coin1*.

Compilation from Jani to PPDDL

We outline the main design decisions in our compilation. The source code of our compiler, along with all Jani and corresponding PPDDL benchmarks, is available online.[1]

Fragment of Jani considered. To match PPDDL planning, we consider probabilistic safety properties, i.e., reachability queries of the form $\Diamond$ $goal$ (eventually $goal$) where the maximum probability is sought. We do not consider temporal goals, as our design rationale in this work is to stick to a Jani fragment that allows for a largely structure-preserving translation into PPDDL. Hence, we avoid compiling temporal formulas into propositional goals (Edelkamp 2006; Baier, Bacchus, and McIlraith 2009), and can focus on the translation of Jani's state space representation into PDDL.

We consider the finite-state model-checking fragment of Jani, where the definition of each integer variable specifies a finite range. Moreover, as we cannot, in PPDDL, compactly handle sets of possible initial states, we consider only *restrict-initial* blocks specifying exactly one value for each variable. The latter two restrictions are not limiting in the sense that we still cover most existing Jani models.

A more limiting restriction we make regards synchronization, where we only consider shared-variable synchronization, via guards referring to variables written on by other automata. Jani also supports handshake synchronization over multiple automata. This can, in principle, be realized in PDDL by adding additional small protocols, similarly to Edelkamp's (2003) approach. But restricting focus on shared-variable synchronization allows a much more

[1] `http://fai.cs.uni-saarland.de/downloads/jani-ppddl.zip`

direct, more structure-preserving, compilation, and is still practically relevant. Unfortunately, many Jani benchmarks make use of handshake synchronization, hence restricting the benchmark set in the empirical evaluation.

Predicates, Types, Objects. Variable types in Jani directly translate into (P)PDDL types, with an additional type *loc* for automata locations. The values associated with a type in Jani are encoded as PDDL objects of that type. Jani variables are also encoded as PDDL objects, variable-value assignments are represented by the PDDL predicate $(value\ var\ val)$. We list all Jani constants, variables and locations as PDDL constants, so we can use them in action descriptions. Current automata locations are encoded through an $(at\_X\ loc)$ predicate for each automaton X. To encode the goal, an additional Boolean Jani variable *goal_condition* is introduced.

We handle finite-range arithmetics in PDDL via finite enumeration of arithmetic-operation outcomes, similarly as in previous works encoding finite-range integer variables into PDDL, e.g., (Nakhost, Hoffmann, and Müller 2012). Namely, we define one PDDL type *int_[lower]_[upper]* for each variable range, with corresponding constants *n0, n1, . . . , nk* representing the values within the range. Arithmetic operations are then hard-coded into the initial state, enumerating e.g. all triples *x*, *y*, and $z = x * y$ within a variable's range, via the list of corresponding static ground facts of the form $(multiply\ x\ y\ z)$. To compactly encode nested expressions, we split these into the recursive application of arithmetic operations. The outcome of each operation is stored in an auxiliary PDDL object. For example, the outcome value z of the expression $(x_1 + x_2) * y$ is encoded through the conjunction of $(add\ x_1\ x_2\ z')$ with $(multiply\ z'\ y\ z)$.

Actions. Jani edge descriptions translate into PPDDL actions. The action parameters are chosen to represent variables affected by the edge's guards or assignments. In particular, this pertains to numeric variables whose value may change: the respective values before and after the action application become parameters constrained by the precondition to match the necessary value computation. For example, an edge guard $z = (x_1 + x_2) * y$ is encoded into parameters $?x_1$, $?x_2$, $?y$, $?z'$, $?z$, along with the preconditions $(add\ ?x_1\ ?x_2\ ?z')$ and $(multiply\ ?z'\ ?y\ ?z)$.

Preconditions and effects can now be directly compiled from Jani guards and assignments. In addition, a precondition and effect of the form $(at\_X\ \ start)$ and $(at\_X\ \ destination)$ encodes the automaton location before and after the action application. Jani permits disjunctive edge guards, which we compile away using standard techniques for DNF transformation (Gazen and Knoblock 1997), followed by splitting the action into one copy per disjunct.

The compilation of effects is the only place where we require the modeling power of probabilistic PDDL, to encode probabilistic edge outcomes. For example, the Jani model from Figure 1 results in the following PPDDL fragment:

```
(:action aut1_loc0_loc1or0
:parameters ()
:precondition (and (at_aut1 loc0) (value coin1 n0))
:effect (probabilistic
  0.5 (and (at_aut1 loc1) (not (value coin1 n0)) (value coin1 n1))
  0.5 (and)))
```

Initial State and Goal. In Jani, the model initialization is spread over the *restrict-initial* expression, the definition of constants, and for each automaton its *initial-location*. These fragments can directly be translated into the *init* part of PDDL (along with arithmetic operations as outlined above).

We translate the property to be checked into a goal, through an action whose precondition is the property expression, and whose effect is *(value goal_condition true)*. The latter ground fact becomes the PDDL goal condition.

Compilation Size. Obviously, the size of our compiled PPDDL encoding relates linearly to that of the input Jani model, except for (a) our encoding of finite-range arithmetic operations, and (b) DNF transformation of edge guards/action preconditions. Both can be expected to be uncritical in practice. Regarding (a), our encoding is exponential in the arity of arithmetic operations, which is harmless as that arity typically is 2. Regarding (b), the exponential blow-up in DNF transformation would be relevant only on highly complex transition guards.

That said, a potentially problematic aspect is the size of the *grounded* encoding resulting from our PPDDL, as the number of ground actions is exponential in the number of action parameters required to capture all variable values relevant to the action. Any one action may contain, in principle, arbitrarily many arithmetic expressions. Again though, in practice, Jani edges – individual steps in the execution of a concurrent system – involve few arithmetic operations. Note also that the parameter-value combinations for ground actions are constrained by the static predicates giving the semantics of the arithmetic operations. Planning systems like FD exploit this property to not even generate obviously unreachable ground actions in the first place.

Experiments

We compare runtime and search space statistics of various configurations of FD, MT, and PRISM. We ran FD using TVI, I-Dual, FRET-v and FRET-π with HDP, all with and without $h^{\max}$. For MT, we ran VI and EVI with and without the preprocessing option. For PRISM, we ran both explicit (Exp) and symbolic engines (Sym) with and without the preprocessing step. All experiments were conducted on an Intel Xeon E5-2660 machine with time (memory) cut-offs of 1h (6GB). We used Cplex as LP solver. In what follows we give a short summary of the results. More details are available in an online TR (Klauck et al. 2018). The preprocessing step of MT and PRISM turned out to be detrimental, we will keep it disabled throughout. We will not report results for PRISM-Exp, which was consistently outperformed by MT-VI.

Most Jani models make use of handshake synchronization, which is currently not supported by our translation, limiting the selection of domains. We consider two case studies, which belong to the most popular benchmarks in probabilistic model-checking. All results are shown in Figure 2. We introduce the case studies, and discuss the respective results.

Dining Cryptographers Protocol (DCP). In DCP (Chaum 1988) $n \geq 3$ cryptographers want to check if the NSA is paying for their dinner while respecting each others privacy,

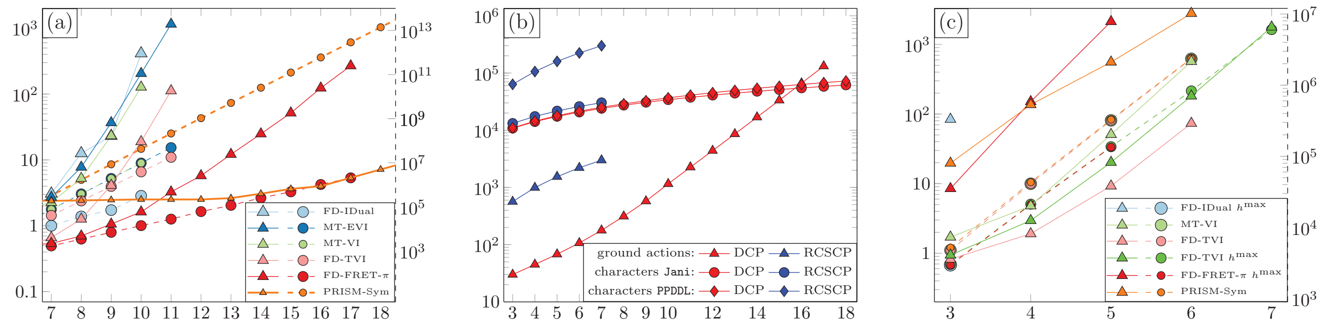

Figure 2: Empirical results on our case studies, as a function of the number n of automata: (a) runtime in seconds (triangles) and number of visited states (circles) in DCP; (b) compilation size measures in both protocols; (c) runtime/visited states in RCSCP. In (a) and (c), the left y-axis shows runtime in seconds while the right y-axis shows the number of states visited.

i.e., without advertising who has paid exactly. For that, everyone throws a coin, the outcomes being encoded as 0 and 1. Then each cryptographer C_i looks at the outcome o and the outcome o_N of his left-hand neighbor. If C_i does not pay for dinner, then C_i announces $o \oplus o_N$ to the table. If C_i does pay for dinner, then C_i announces $\overline{o \oplus o_N}$. The intended guarantee is that the xor over all announced values is 0 if the NSA paid, and is 1 if one of the cryptographers paid. What we wish to check is that the protocol is correct. This is encoded as reaching a target condition with probability 1.

A specific property of this domain is the absence of 0 goal-probability states. In such cases, $h^{\max}$ merely constitutes an additional overhead. In larger DCP instances this overhead was significant. In the following, we disabled $h^{\max}$ in all FD configurations. On the other hand, this property particularly qualifies the use of contingent planners. We ran the state-of-the-art contingent planner PRP (Muise, McIlraith, and Beck 2012), but which was outperformed by FD-FRET-π. Detailed results are available in the TR.

Consider Figure 2(a) from top to bottom. Having to solve increasingly large LPs, renders FD-IDual uncompetitive – despite offering small advantages in search space size. The additional overhead induced by MT-EVI compared to MT-VI shows in runtime, but its smaller memory demand improves scaling to n=11. FD-TVI is significantly faster than MT-VI, presumably due to more efficient internal data structures. It scales up to n=11, and similarly to MT's VI versions, runs out of memory afterwards. FD-FRET-π benefits from the ability of the underlying heuristic search algorithm HDP to find a solution while visiting only a small fraction of the state space. The resulting reduction in the number of visited states grows exponentially with n. In FRET-v, not shown in the figure, that benefit is lost in trap removal. Interestingly, the most limiting factor in FD-FRET-π lies in the grounding process. Figure 2(b) illuminates this with compilation-size data. For the models themselves, i.e. Jani vs. PPDDL, the difference is small. But at the grounded level, where FD enumerates action parameter instantiations, matters are different. The number of ground actions increases steeply in n. The clear winner in this case study is PRISM-Sym, which is able to compactly represent extremely large numbers of states. Additionally to benefits in memory usage, this representation leads to tremendous runtime improvements, boosting scalability up to n=26.

Randomized Consensus Shared Coin Protocol (RCSCP). In this protocol (Aspnes and Herlihy 1990) n tourists wish to reach consensus of which place to visit next. In a nutshell, each tourist keeps tossing a coin; depending on the outcome of this toss, a global counter is either incremented or decremented; when the counter is within a certain desired range, the tourist stops. What we want to check is the probability that all tourists stopped at the same counter value. In difference to the DCP, that probability is < 1.

Figure 2(c) shows the performance results. FD-IDual again suffers from having to solve many LPs. Contrary to DCP, HDP does not provide any reduction in search space size. Moreover, it requires significantly more value updates until convergence than, e.g., TVI, rendering the respective FRET configurations uncompetitive. PRISM-Sym cannot compactly represent the state space, while numerical operations on the symbolic data structures are more expensive. In RCSCP it is not possible to give a good state-partitioning function organizing MT-EVI's external-memory access. FD-TVI is again more efficient than MT-VI. Although the heuristic evaluations of FD-TVI-$h^{\max}$ negatively affect runtime, compared to FD-TVI, the number of visited states can be reduced considerably. This allows to scale to n=7, turning FD-TVI-$h^{\max}$ here to the overall best configuration. Consider finally the part of Figure 2(b) pertaining to RCSCP. In difference to DCP, the Jani models are much smaller than the PPDDL ones. This is due to the size of the initial state, which needs to encode several arithmetic operations. In terms of the number of ground actions though, RCSCP is harmless.

Conclusion and Future Work

We have established a new connection between probabilistic model-checking and probabilistic planning. The empirical results show advantages for techniques from both areas, depending as always on the domain. To foster compilability of model-checking models into probabilistic planning, support for numeric state variables, and for more general goals, would be desirable. Further directions suggested by our experiments include the development of hybrid methods between grounding and lifted methods in planning, combining the benefits of both sides. Porting probabilistic heuristic search algorithms to model-checking tools directly, would give an opportunity to tackle all the syntactic elements that cannot be easily compiled. Moreover, this would allow to profit from techniques developed in this context already, notably the wealth of abstraction techniques that can be used to obtain better goal probability bounds.

Acknowledgments. This work was partially supported by the ERC Advanced Investigators Grant 695614 (POWVER) and by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA, grant no. 16KIS0656).

References

Altman, E. 1999. *Constrained Markov Decision Processes*. Chapman and Hall.

Aspnes, J., and Herlihy, M. 1990. Fast randomized consensus using shared memory. *J. Algorithms* 11(3):441–461.

Baier, J. A.; Bacchus, F.; and McIlraith, S. A. 2009. A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence* 173(5-6):593–618.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Bonet, B., and Geffner, H. Faster heuristic search algorithms for planning with uncertainty and full feedback. *Proc. IJCAI'03*, 1233–1238.

Bonet, B., and Geffner, H. 2005. mgpt: A probabilistic planner based on heuristic search. *Journal of Artificial Intelligence Research* 24:933–944.

Brafman, R. I.; Giacomo, G. D.; and Patrizi, F. LTL$_f$ / LDL$_f$ non-markovian rewards. *Proc. AAAI'18*.

Budde, C. E.; Dehnert, C.; Hahn, E. M.; Hartmanns, A.; Junges, S.; and Turrini, A. JANI: quantitative model and tool interaction. *TACAS'17*, 151–168.

Camacho, A.; Chen, O.; Sanner, S.; and McIlraith, S. A. Decision-making with non-markovian rewards: From LTL to automata-based reward shaping. *Proc. RLDM'17*, 279–283.

Chaum, D. 1988. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptology* 1(1):65–75.

Dai, P.; Mausam; Weld, D. S.; and Goldsmith, J. 2011. Topological value iteration algorithms. *Journal of Artificial Intelligence Research* 42:181–209.

Edelkamp, S. Promela planning. *Proc. SPIN'03*, 197–212.

Edelkamp, S. On the compilation of plan constraints and preferences. *Proc. ICAPS'06*, 374–377.

Gazen, B. C., and Knoblock, C. Combining the expressiveness of UCPOP with the efficiency of Graphplan. *Proc. ECP'97*, 221–233.

Hahn, E. M., and Hartmanns, A. A comparison of time- and reward-bounded probabilistic model checking techniques. *Proc. SETTA'16*, 85–100.

Hartmanns, A., and Hermanns, H. The modest toolset: An integrated environment for quantitative modelling and verification. *TACAS'14*, 593–598.

Hartmanns, A., and Hermanns, H. Explicit model checking of very large MDP using partitioning and secondary storage. *Proc. ATVA'15*, 131–147.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Johnson, B., and Kress-Gazit, H. 2011. Probabilistic analysis of correctness of high-level robot behavior with sensor error. In *Robotics: Science and Systems*.

Klauck, M.; Steinmetz, M.; Hoffmann, J.; and Hermanns, H. 2018. Compiling probabilistic model checking into probabilistic planning (technical report). Technical report, Saarland University. Available at http://fai.cs.uni-saarland.de/hoffmann/papers/icaps18a-tr.pdf.

Kolobov, A.; Mausam; Weld, D. S.; and Geffner, H. Heuristic search for generalized stochastic shortest path MDPs. *Proc. ICAPS'11*.

Kwiatkowska, M. Z.; Norman, G.; and Parker, D. PRISM 4.0: Verification of probabilistic real-time systems. *Proc. CAV'11*, 585–591.

Lacerda, B.; Parker, D.; and Hawes, N. Optimal policy generation for partially satisfiable co-safe LTL specifications. *Proc. IJCAI'15*.

Muise, C. J.; McIlraith, S. A.; and Beck, J. C. Improved non-deterministic planning by exploiting state relevance. *Proc. ICAPS'12*.

Nakhost, H.; Hoffmann, J.; and Müller, M. Resource-constrained planning: A Monte Carlo random walk approach. *Proc. ICAPS'12*, 181–189.

Steinmetz, M.; Hoffmann, J.; and Buffet, O. 2016. Goal probability analysis in mdp probabilistic planning: Exploring and enhancing the state of the art. *Journal of Artificial Intelligence Research* 57:229–271.

Teichteil-Königsbuch, F. Path-constrained markov decision processes: bridging the gap between probabilistic model-checking and decision-theoretic planning. *Proc. ECAI'12*, 744–749.

Trevizan, F. W.; Teichteil-Königsbuch, F.; and Thiébaux, S. Efficient solutions for stochastic shortest path problems with dead ends. *UAI'17*.

Younes, H. L. S.; Littman, M. L.; Weissman, D.; and Asmuth, J. 2005. The first probabilistic track of the international planning competition. *Journal of Artificial Intelligence Research* 24:851–887.