Abstractions in probabilistic planning

Max Grüner
University of Basel
Max.Gruener@unibas.ch

February 11, 2016

Abstract

In planning what we want to do is to get from an initial state into a goal state. A state can be described by a finite number of boolean valued variables. If we want to transition from one state to the other we have to apply an action and this, at least in probabilistic planning, leads to a probability distribution over a set of possible successor states. From each transition the agent gains a reward dependent on the current state and his action. In this setting the growth of the number of possible states is exponential with the number of variables. We assume that the value of these variables is determined for each variable independently in a probabilistic fashion. So these variables influence the number of possible successor states in the same way as they did the state space. In consequence it is almost impossible to obtain an optimal amount of reward approaching this problem with a brute force technique. One way to get past this problem is to abstract the problem and then solve a simplified version of the aforementioned. That's in general the idea proposed by Boutilier and Dearden [1]. They have introduced a method to create an abstraction which depends on the reward formula and the dependencies contained in the problem. With this idea as a basis we'll create a heuristic for a trial-based heuristic tree search (THTS) algorithm [5] and a standalone planner using the framework PROST (Keller and Eyerich,2012). These will then be tested on all the domains of the International Probabilistic Planning Competition (IPPC).

Acknowledgements

I would like to thank Thomas Keller for all his help. The weekly meetings and the close contact via mail really have helped to push this thesis forward. I would also like to thank Professor Malte Helmert for being able to write this thesis in his research group.

Contents

Chapter 1

Introduction

To solve planning problems has always been a popular challenge in the field of computer science. One well known planning problem is the travelling salesman problem. Here a salesman has to visit all cities on his list. He has to take the shortest possible route which passes through all cities. To go about this naively by trying out all possibilities is not a smart tactic since there are too many possible routes. With just 10 cities there are already 3628800 possible routes when we assume that the salesman can only pass through a city once. There being no brute force solution available is the critical part of almost all planning problems. Therefore algorithms which are used to solve these problems work with all kinds of workarounds to approximate a solution.
A way to approximate a solution for this problem is to sum up cities which are close to each other. So we only optimize the way to the crowded areas. Putting this idea more technical we could say that we abstract the state space, which is the space of all reachable states.
To abstract the state space is a part of the idea proposed by Boutilier and Dearden [1] in their paper from 1994. They recognized the computational problems that accompany the calculation of an optimal policy when going about it with value iteration [4]. They developed a new technique to sample the state space as opposed to the envelope method [2]. The envelope technique samples the state space and then computes a solution for this envelope. If the agent falls out of the envelope (there's no policy for the state in which he is in now) the envelope is extended or altered. They propose to sample for variables rather than for states. Starting with an initial set, variables are sampled according to their dependency to the variables already contained in the set. The initial set is obtained from the reward formula of the problem. A reward formula rewards the agent depending on the state he's in and the action he applies.
Reward formulas are not an outdated concept. They're as well in use in current competitions. In 2011 and 2014 the problems at the IPPC at the International Conference on Automated Planning and Scheduling (ICAPS) were predominantly driven by reward formulas. This gives us the option to re-evaluate the method proposed by Boutilier and Dearden. The challenge will be to adapt and to implement the concepts proposed in their paper for our purposes.
As always some theoretical concepts need to be introduced first in order to build the basis for this thesis. These concepts will illustrate how to properly construct a probabilistic planning task with which we'll work later on.

Chapter 2

Background

An MDP (Markov-Decision-Process) is a deviation of a Markov-Process. In the **simplified** version (which is sufficient for our problems) a Markov-Decision-Process is defined as in Definition 1.

Definition 1 (Markov-Decision-Process)**.** A factored, finite-horizon Markov-Decision-Process consists of a 6-tuple $\langle \mathcal{V}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{H}, s_0 \rangle$ where

- $\mathcal{V} = \{v_0, \ldots, v_n\}$ is a finite set of boolean variables. $\mathcal{V}$ induces a set of states $S = \Pi(\mathcal{V})$.
- $\mathcal{A} = \{a_0, \ldots, a_m\}$ is a finite set of actions.
- $\mathcal{P} = \{p_{a_0}(\cdot,\cdot), \ldots, p_{a_m}(\cdot,\cdot)\}$ is a set of $p_{a_i} : S \times S \mapsto \mathbb{R}$ which gives the probability that state s transitions into state s' given we are in state s and action a_i is applied.
- $\Re = \{re_{a_0}(\cdot), \ldots, re_{a_m}(\cdot)\}$ where each $\mathrm{re}_{a_i} : S \mapsto \mathbb{R}$. This is the immediate reward if action a_i is applied to state s.
- $\mathcal{H} \in \mathbb{N}$ is the finite horizon.
- $s_0 \in S$ is the initial state.

With the concept given we can simulate a system where a transition can be probabilistic. This means that, given a state s, there can be more than one possible successor state. The set of states which can be created are limited by the transition functions contained in $\mathcal{P}$. We can apply one action at a time. Each action, applied to a certain state, rewards the agent and therefore implicitly defines an optimal action for each state.
Let us consider an example where we have 2 switches. There are 4 combinations in which both switches can be on, only one can be on and both can be off. We suppose that we can influence the way the switches behave by the way we pull them. We can pull them gently or with force. The probabilities for the switch-combination change depending on which action we choose. The reward does change as well. Here a matrix was chosen to represent the transition probabilities and a list for the reward to be obtained. Note that these representations can be adapted to the problem.

The MDP is defined as follows:

- $\mathcal{V} = \{switch_1, switch_2\}$
- $\mathcal{A} = \{pull_{gently}, pull_{violently}\}$
- $\mathcal{P} = \{p_{pull_{gently}}, p_{pull_{violently}}\}$

$$p_{pull_{gently}} = \begin{array}{c|cccc} & \{switch_1\} & \{switch_2\} & \{switch_1, switch_2\} & \{\} \\ \hline \{switch_1\} & 0 & 1 & 0 & 0 \\ \{switch_2\} & 0 & 0.7 & 0 & 0.3 \\ \{switch_1, switch_2\} & 0.18 & 0.28 & 0.48 & 0.12 \\ \{\} & 0 & 1 & 0 & 0 \end{array}$$

$$p_{pull_{violently}} = \begin{array}{c|cccc} & \{switch_1\} & \{switch_2\} & \{switch_1, switch_2\} & \{\} \\ \hline \{switch_1\} & 0.4 & 0 & 0 & 0.6 \\ \{switch_2\} & 0.28 & 0.18 & 0.12 & 0.48 \\ \{switch_1, switch_2\} & 0.28 & 0.18 & 0.12 & 0.48 \\ \{\} & 1 & 0 & 0 & 0 \end{array}$$

- $\Re = \{\mathrm{re}_{pull_{gently}}, \mathrm{re}_{pull_{violently}}\}$

$$\begin{array}{c|cccc} & \{switch_1\} & \{switch_2\} & \{switch_1, switch_2\} & \{\} \\ \hline re_{pull_{gently}} = & 26 & 33 & 29 & 30 \\ re_{pull_{violently}} = & 38 & 15 & 23 & 30 \end{array}$$

- $\mathcal{H} = 1$
- $s_0 = \{switch_1\}$

The following graphs represent the graphical representation for one state each. The numbers at the outgoing edges represent rewards while the numbers at the incoming edges represent the probability of transition. The Circle enclose the states while the rectangle enclose the actions.

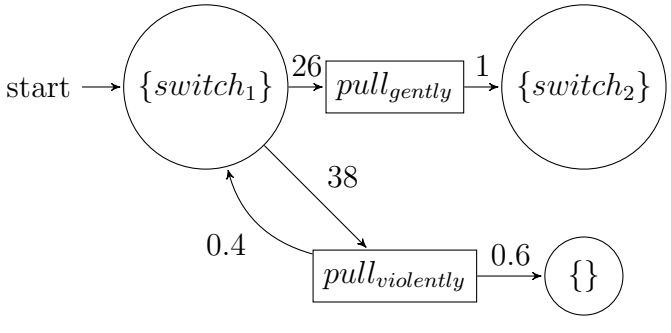

Figure 2.1: Graphical representation of the possible transitions for $\{switch_1\}$

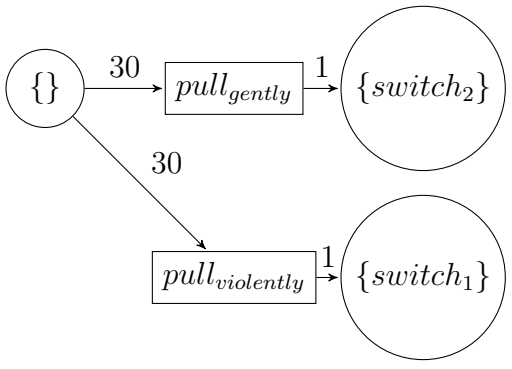

Figure 2.2: Graphical representation of the possible transitions for {}

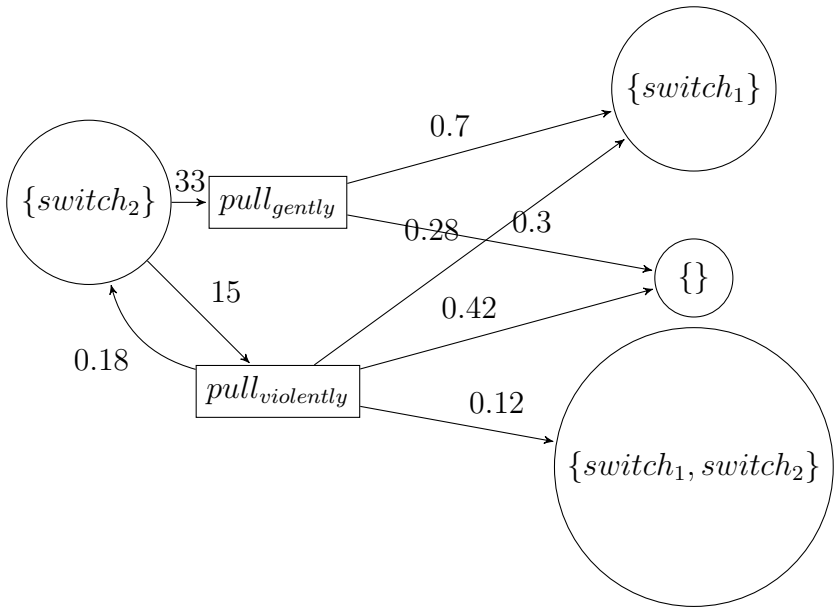

Figure 2.3: Graphical representation of the possible transitions for $\{switch_2\}$

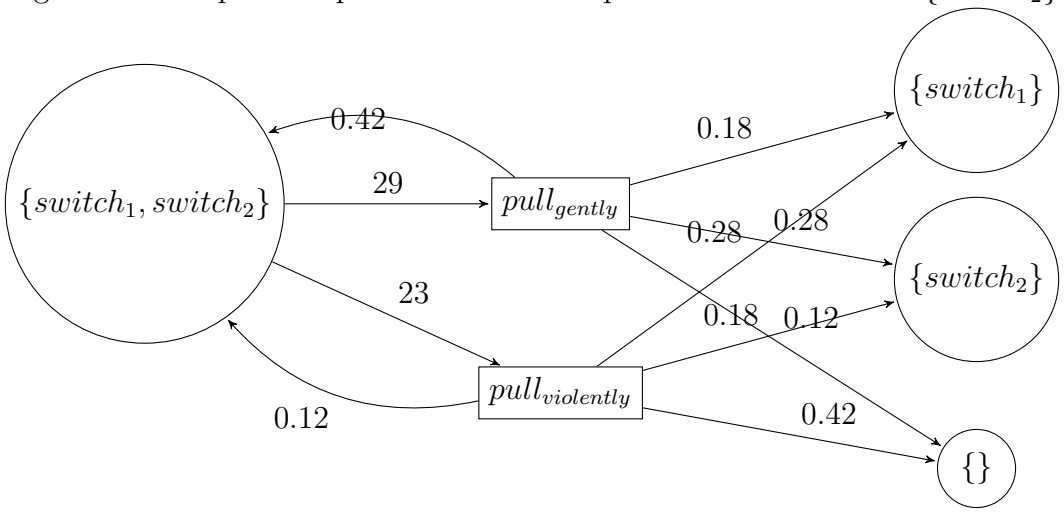

Figure 2.4: Graphical representation of the possible transitions for $\{switch_1, switch_2\}$

Now that we have a model for our system we want to find an optimal way through this system. This is to say that we need to find an optimal policy as defined in Definition 2.

Definition 2 (Optimal Policy)**.** Given an MDP $\langle \mathcal{V}, \mathcal{A}, \mathcal{P}, \Re, \mathcal{H}, s_0 \rangle$ the functions

$$V^{\pi}(s,d) = \max_a Q^{\pi}(s,d,a) \text{ and}$$
$$Q^{\pi}(s,d,a) = \begin{cases} re_a(s) + \sum_{s'} p_a(s,s') \cdot V^{\pi}(s', d-1) & d > 0 \\ re_a(s) & \text{otherwise} \end{cases}$$

induce an optimal policy π as the policy that maximizes $V^{\pi}(s_0, \mathcal{H})$. If the value $V^{\pi}(s,d)$ coincides with the real value it also can be described as $V^*(s,d) = max_a Q^*(s,d,a)$.

So a state is evaluated by the reward to be gained instantly and the possible reward to be gained in the future. The recursion depth of this formula is limited by the horizon.
Going back to our example we can illustrate this by solving the MDP defined in the last example. Since the horizon is 1 we only have to solve the equation for the start state.

$$\begin{aligned} Q(\{switch_1\}, 1, pull_{gently}) &= 26 + V^{\pi}(\{switch_2\}, 0) \\ &= 26 + 33 = 59 \\ Q(\{switch_1\}, 1, pull_{violenty}) &= 35 + 0.4 \cdot V^{\pi}(\{switch_1\}, 0) + 0.6 \cdot V^{\pi}(\{\}, 0) \\ &= 35 + 0.6 \cdot 35 + 0.6 \cdot 20 = 69 \\ V^{\pi}(\{itRains\}, 1) &= \max_a(Q(\{switch_1\}, 1, pull_{gently}), Q(\{switch_1\}, 1, pull_{violenty})) \\ &= \max(59, 69) = 69 \end{aligned}$$

This shows that choosing the action $pull_{violently}$ is the best option in the given example. Now we continue by introducing the representation of the MDPs chosen in our problem setting. To be able to describe a probabilistic planning task we need to define logical expressions and Iverson brackets first.

Definition 3 (Logical Expression)**.** Given an MDP $\langle \mathcal{V}, \mathcal{A}, \mathcal{P}, \Re, \mathcal{H}, s_0 \rangle$ a logical expression is a boolean expression over a set of variables $\widehat{\mathcal{V}} \subseteq \mathcal{V}$.

Definition 4 (Iverson Brackets)**.** Given a boolean value we can evaluate it with Iverson brackets. This notation allows us to evaluate boolean expressions to the integer values 0 and 1. This translates to

$$\begin{aligned} [v_y]^I &= 1 \quad \text{iff } v_y \text{ holds} \\ [v_y]^I &= 0 \quad \text{otherwise} \end{aligned}$$

With these definitions we're now able to define a probabilistic planning task as in Definition 5.

Definition 5 (Probabilistic Planning Task)**.** A probabilistic planning task is a 6-Tuple $< \mathcal{V}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{H}, s_0 >$ where

- $\mathcal{V} = \{v_0, \ldots, v_n\}$ is a finite set of boolean variables. $\mathcal{V}$ induces a set of states $S = \Pi(\mathcal{V})$.
- $\mathcal{A} = \{a_0, \ldots, a_m\}$ is a finite set of actions.
- $\mathcal{T} = \{t_{a_0}, \ldots, t_{a_m}\}$ where $t_{a_i} = \{t_{a_i}^{v_0}, \ldots, t_{a_i}^{v_n}\}$. Each of these elements defines as $t_{a_i}^{v_j} : S \mapsto [0, 1]$. $t_{a_i}^{v_j}$ is an ordered sequence of condition-value pairs $\{(c_0, p_0), \ldots, (c_q, p_q)\}_{a_i}^{v_j}$ where $c_q = true$. A condition c_k which is a logical expression induces an effect p_k if c_k holds. An effect $p_k \in [0, 1]$ is the truth-probability of a Bernoulli distribution. So $t_{a_i}^{v_j}(s)$ is the probability that v_j, given we're in state s and action a_i is applied, is true. Given a function $p_{c_m} : S \mapsto \{true, false\}$ which evaluates a condition c_m under the given state. This results in

  $$p(s, v_j, a_i) = p_0 \cdot [p_{c_0}(s)]^I + \sum_{r=1}^{q} p_r \cdot [p_{c_r}(s)]^I \cdot \prod_{l=0}^{r-1} (1 - [p_{c_l}(s)]^I) \mid \forall c, \exists (c, p) \in t_{a_i}^{v_j}$$

  This represents the probability that, given a state s and an action a_i, v_j is true.

- $\mathcal{R} = \{r_{a_0}(\cdot), \ldots, r_{a_m}(\cdot)\}$ where each $r_{a_i} : S \mapsto \mathbb{R}$. This results in the immediate reward if action a_i is applied to state s. This structure may contain Iverson brackets.
- $\mathcal{H} \in \mathbb{N}$ is the finite horizon.
- $s_0 \in S$ is the initial state.

t_{a_i} are transition formulas and can be referred to as conditional probability functions (cpf). A probabilistic planning task induces the mathematical structure of a MDP as described in Definition 6.

Definition 6 (Induction of an MDP)**.** A probabilistic planning task $< \mathcal{V}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{H}, s_0 >$ induces a factored,finite-horizon MDP $< \mathcal{V}, \mathcal{A}, \mathcal{P}, \Re, \mathcal{H}, s_0 >$ in the following way

$$p_{a_i}(s, s') = \prod_{v \in s'} p(s, v, a_i) \cdot \prod_{v \notin s'} (1 - p(s, v, a_i))$$
$$re_a(s) = r_a(s)$$

Therefore the state transitions are now defined implicitly rather than explicitly. We can apply this to our former example by adapting the transition probability formulas. Now we arrive at the following representation for the following probabilistic planning task which induces the MDP defined before.

- $\mathcal{T} = \{t_{pull_{gently}}, t_{pull_{violenty}}\}$
- $t_{pull_{gently}} = \{t^{switch_1}_{pull_{gently}}, t^{switch_2}_{pull_{gently}}\}$
  - $t^{switch_1}_{pull_{gently}} = \{(switch_1 \wedge switch_2, 0.6), (true, 0)\}$
  - $t^{switch_2}_{pull_{gently}} = \{(switch_2, 0.7), (true, 1)\}$
- $t_{pull_{violenty}} = \{t^{switch_1}_{pull_{violenty}}, t^{switch_2}_{pull_{violenty}}\}$
  - $t^{switch_1}_{pull_{violenty}} = \{(switch_1 \vee switch_2, 0.4), (true, 1)\}$
  - $t^{switch_2}_{pull_{violenty}} = \{(switch_2, 0.3), (true, 0)\}$
- $\mathcal{R} = \{r_{pull_{gently}}, r_{pull_{violently}}\}$
  - $r^P_{pull_{gently}} = [switch_1]^I \cdot 16 + [switch_2]^I \cdot 13 + [\backsim switch_1]^I \cdot 20 + [\backsim switch_2]^I \cdot 10$
  - $r^P_{pull_{violently}} = [switch_1]^I \cdot 18 + [switch_2]^I \cdot 5 + [\backsim switch_1]^I \cdot 10 + [\backsim switch_2]^I \cdot 20$

Having defined a theoretical concept of how to define and solve probabilistic planning tasks we're well equipped to advance.We now can begin introducing, implementing, evaluating and eventually improving the Idea by Boutilier and Dearden.

Chapter 3

Abstractions

Boutlier and Dearden introduce their idea with a simple example. They imagine a situation where a robot has to get coffee. It has to traverse a road in order to do so. Additionally it has to get an umbrella if it rains. It obtains the highest reward if it arrives dry with coffee and lowest if it arrives wet and without coffee. The most important part of the task is to get the coffee to the user, though. So the problem consists of 7 variables ($location_1, location_2, robot_{\text{coffee}}, robot_{\text{umbrella}}, rain, robot_{\text{wet}}, user_{\text{coffee}}$) and therefore a theoretical state space of the size 128 ($= 2^7$).
There are also 7 actions ($GoL_1, GoL_1WRain, GoL_2, GoL_2WRain, BuyC, GetU, DelC$) which illustrate the transitions. Starting out from this example they propose to just let the robot get coffee and ignore the fact whether it rains or not. This leaves them with only 4 variables ($location_1, location_2, robot_{\text{coffee}}, user_{\text{coffee}}$) which reduces the state space to the size 16 ($= 2^4$) and still produces a good solution. But how do they create this pattern?
First of all they take the most influential reward variable as a starting point ($user_{\text{coffee}}$). Later in the paper they define as a measure of quality for this initial pattern its span. The span is the maximum degree in which the reward of the reduced pattern differs from the real reward. Then they recursively add all variables which appear as preconditions for the actions which influence the variables which are already in the pattern. In this example they do this until they can without ambiguity evaluate the preconditions for all actions which influence the variables contained in the pattern. Consequently their technique can be described in 3 explicit and 1 implicit step.

1. (Implicit) Define a limit for the pattern. This has to be done for bigger tasks since we can't add an unlimited number of variables to a pattern.

2. Use the span the dete rmine the best subset of the variables contained in the reward formulas.

3. Recursively add variables which appear in the conditions of the actions associated with the variables contained in the pattern.

4. Create an abstract probabilistic planning task.

We start evaluating this technique with a bottom up approach. So we define the concept of abstract probabilistic planning tasks first.

3.1 Abstract Tasks

For the first section we'll assume that we already have a pattern to reduce the probabilistic planning task to. Later we'll introduce some methods to create patterns given an initial probabilistic planning task

Definition 7 (Abstract Probabilistic Planning Task)**.** An abstract probabilistic planning task $\langle \mathcal{V}^P, \mathcal{A}, \mathcal{T}^P, \mathcal{R}^P, \mathcal{H}, s_0^P \rangle$ for a probabilistic planning task $\langle \mathcal{V}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{H}, s_0 \rangle$ is defined in the following way

- $\mathcal{V}^P \subseteq \mathcal{V}$
- $f_p^{\mathcal{T}} : \mathcal{T} \mapsto \mathcal{T}^P$ where
  - $p(s, v_j, a) = p(s \setminus v_i, v_j, a) \mid \forall v_j \in \mathcal{V}^P, v_i \notin \mathcal{V}^P, v_i \in s$ where
  - $p(s, v_j, a)$ is induced by $t_a^{v_j} \in t_a^P \mid t_a^P \in \mathcal{T}^P$ and
  - $\nexists t_a^{v_i} \in t_a^P$
- $f_p^{\mathcal{R}} : \mathcal{R} \mapsto \mathcal{R}^P$ where
  - $r_a^P(s) = r_a^P(s \setminus v_i) \mid \forall r_a^P \in \mathcal{R}^P, v_i \notin \mathcal{V}^P, v_i \in s$ where
  - $r_a^P(s)$ is induced by $\mathcal{R}^P$.
- $s_0^P = s_0 \cap \mathcal{V}^P$

So we use two different functions to reduce the given structures to their reduced form. For $f_p^{\mathcal{R}}$ we use several implementations while for $f_p^{\mathcal{T}}$ we in general only use one. This implementation can also be used for the reward formulas. In the following subsection we'll introduce some formulas which given a pattern can create an abstract probabilistic planning task.

3.1.1 Reduction

To create a abstract probabilistic planning task we need two reductive functions. We need a function for the transition formula and a function for the reward formula. Since we only use one formula for the transition formulas it is showcased first and then the implementations for the reward formulas are introduced.

Ignore

The idea of this implementation is to ignore the variables which are not part of the pattern. We're given a

- probabilistic planning task $\langle \mathcal{V}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{H}, s_0 \rangle$ and
- a pattern $\mathcal{V}^P$.

The respective formula is only modified if $v \notin \mathcal{V}^P$ holds for the variable v.
In order to modify $p(s, v_j, a)$ for the transition formula, as described in the Definition 7, we need to modify for all $t_a \in \mathcal{T}$. This we do by modifying the conditions c contained in the condition-effect pairs $(c, p) \in t_a^v$ for all $t_a^v \in t_a$. So we modify every c according to the following principle which depends on the operation used on the variable. One of the following actions is applied if the preconditions are met:

- If the operator is a conjunction and $\widehat{v} =\sim v \mid \widehat{v} = v$ then $\widehat{v} \mapsto true$
- If the operator is a disjunction and $\widehat{v} =\sim v \mid \widehat{v} = v$ then $\widehat{v} \mapsto false$

In order to meet the condition described in Definition 7 for $\mathcal{R}^P$, each r_a is modified in the following way if the given preconditions are met:

- If the operator is an addition or a subtraction and $\widehat{v} =\sim v \mid \widehat{v} = v$ then $\widehat{v} \mapsto false$
- If the operator is a multiplication or a division and $\widehat{v} =\sim v \mid \widehat{v} = v$ then $\widehat{v} \mapsto true$

This technique just ignores the variables which are not contained in our pattern. The evaluation is not perturbed by these variables. If we apply this technique on the initial example of the Background chapter we arrive at the following representation.
We use the pattern $P = \{switch_1\}$. Now there are 2 less states to include in the computation representation of this probabilistic planning task. This results in the representation below.

- $\mathcal{V}^P = \{switch_1\}$
- $\mathcal{T} = \{t_{pull_{gently}}, t_{pull_{violenty}}\}$
- $t_{pull_{gently}} = \{t^{switch_1}_{pull_{gently}}\}$
  - $t^{switch_1}_{pull_{gently}} = \{(switch_1, 0.6), (true, 0)\}$
- $t_{pull_{violenty}} = \{t^{switch_1}_{pull_{violenty}}\}$
  - $t^{switch_1}_{pull_{violenty}} = \{(switch_1, 0.4), (true, 1)\}$
- $\mathcal{R} = \{r_{pull_{gently}}, r_{pull_{violently}}\}$

$$r_{pull_{gently}}(s) = [switch_1]^I \cdot 16 + [\backsim switch_1]^I \cdot 20$$
$$r_{pull_{violently}}(s) = [switch_1]^I \cdot 18 + [\backsim switch_1]^I \cdot 10$$

Now the graphical representation reduces from the former ones (2.1,2.2,2.3 and 2.4) to the ones below

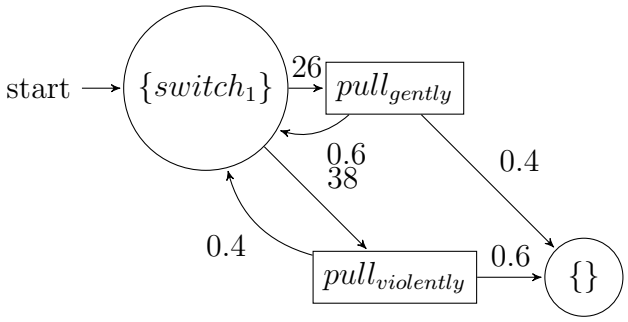

Figure 3.1: reduced graphical representation of the possible transition for $\{switch_1\}$

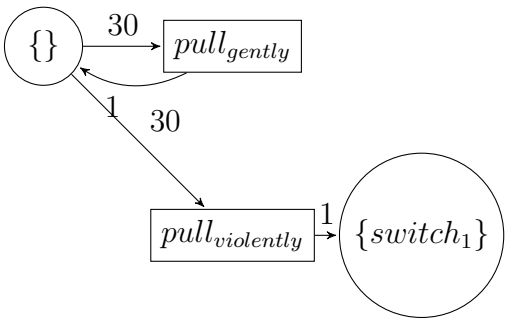

Figure 3.2: reduced graphical representation of the possible transition for {}

Now we continue exploring the possibilities of how to reduce reward formulas.

Overestimate/Underestimate

Here we replace every occurrence of all variables $v \notin \mathcal{V}^P$ in all $r_a \in \mathcal{R}$ with the range $[0, 1]$. We then use the technique of the interval arithmetic to evaluate this formula. Interval arithmetic is a kind of evaluation which doesn't evaluate single number but ranges of numbers. The rules for the evaluation go as follows:

- $[x_{min}, x_{max}] \wedge [y_{min}, y_{max}] = [x_{min} \wedge y_{min}, x_{max} \wedge y_{max}]$
- $[x_{min}, x_{max}] \vee [y_{min}, y_{max}] = [x_{min} \vee y_{min}, x_{max} \vee y_{max}]$
- $[x_{min}, x_{max}] + [y_{min}, y_{max}] = [x_{min} + y_{min}, x_{max} + y_{max}]$
- $[x_{min}, x_{max}] - [y_{min}, y_{max}] = [x_{min} - y_{min}, x_{max} - y_{max}]$
- $[x_{min}, x_{max}] \cdot [y_{min}, y_{max}] = [min(x_{min} \cdot y_{min}, x_{min} \cdot y_{max}, x_{max} \cdot y_{min}, x_{max} \cdot y_{max}), max(x_{min} \cdot y_{max}, x_{min} \cdot y_{max}, x_{max} \cdot y_{min}, x_{max} \cdot y_{max})]$
- $\frac{[x_{min}, x_{max}]}{[y_{min}, y_{max}]} = [x_{min}, x_{max}] \cdot \frac{1}{[y_{min}, y_{max}]}$ where
  - $\frac{1}{[y_{min}, 0]} = [-\inf, \frac{1}{y_{min}}]$ and
  - $\frac{1}{[0, y_{max}]} = \frac{1}{[y_{max}, \inf]}$

This technique guarantees an upper respectively a lower bound. Let's define a function *count* : $\mathcal{R} \times \mathcal{V} \mapsto \mathbb{N}$ which returns for every variable the number of its occurrences in the given reward formula. The bound for a reward formula r_a is only accurate if $\forall v \in \mathcal{V}, count(r_a, v) \leq 1$ holds. If this does not hold the formula is still bounded but no statement about the accuracy can be given. With a combinatorial method this would be possible but as 2^n grows exponentially this is not feasible. The overestimate $(\mathrm{ov}^P(r_a)(s))$ approach returns the upper limit of the interval evaluation and the underestimate $(\mathrm{ud}^P(r_a)(s))$ approach returns the lower limit of the interval evaluation. When we apply the general approach to an example this looks like this

$$\begin{aligned} r^P_{pull_{gently}} &= [0,1] \cdot 13 + [switch_1]^I \cdot 16 + [\backsim switch_1]^I \cdot 2 + [0,1] \cdot 10 \\ &= [0, 23] + [switch_1]^I \cdot 16 + [\backsim switch_1]^I \cdot 20 \\ r^P_{pull_{violently}} &= [0, 25] + [switch_1]^I \cdot 18 + [\backsim switch_1]^I \cdot 10 \end{aligned}$$

Here the inaccuracy of the method becomes obvious as the real maximum of the already evaluated interval is 13 and not 23 for the action $pull_{gently}$. This is because of the fact that the interval arithmetic evaluates each interval independently.

Boutilier and Dearden

This technique is taken from the paper by Boutilier and Dearden. Here we evidently do almost the same as in the overestimation respectively underestimation technique. The only difference is that we combine both estimates.

$$r^P_a = \frac{\mathrm{ov}^P(r_a)(s) + \mathrm{ud}^P(r_a)(s)}{2}$$

When applied to an example it looks like this

$$
\begin{aligned}
r^P_{pull_{gently}} &= \frac{\mathrm{ov}(pull_{gently}) + \mathrm{ud}(pull_{gently})}{2} \\
&= \frac{23 + [switch_1]^I \cdot 16 + [\backsim switch_1]^I \cdot 20 + [switch_1]^I \cdot 16 + [\backsim switch_1]^I \cdot 20}{2} \\
r^P_{pull_{violently}} &= \frac{\mathrm{ov}(r_{pull_{violently}}) + \mathrm{ud}(r_{pull_{violently}})}{2} \\
&= \frac{25 + [switch_1]^I \cdot 16 + [\backsim switch_1]^I \cdot 20 + [switch_1]^I \cdot 16 + [\backsim switch_1]^I \cdot 20}{2}
\end{aligned}
$$

Now we have a concept of how to reduce a probabilistic planning task with a reductive function if we have a pattern. The thing we have to do next is to investigate how to create a pattern.

3.2 Pattern Creation

In the section we'll showcase 3 methods to create patterns. The first two will rely on an initial set of variables. This set will then by enlarged by the respective algorithm proposed by the method. The third method creates this initial set from the reward formulas.
We first start with the method introduced by Boutilier and Dearden which required the aforementioned initial set.

3.2.1 Boutilier and Dearden

Boutilier and Dearden create the pattern exhibited in their paper by recursively adding dependent variables to the pattern. So the first thing we need to do is to define dependency in this context.

Definition 8 (Dependency)**.** Given an probabilistic planning task a variable v_i is dependent of a variable v_j ($dep(v_i, v_j)$) with $i \neq j$ if $\exists s \in S, a \in \mathcal{A}$

$$p(s, v_i, a) \neq p(s \setminus \{v_j\}, v_i, a)$$

These dependent variables are then put into a causal graph.

Definition 9 (Causal Graph)**.** A causal graph for a probabilistic planning task PPT is defined by a 2-Tuple $\langle \mathcal{N}, \mathcal{E} \rangle$ where

- $\mathcal{N}$ are nodes which represent the variables contained in $\mathcal{V}$.
- $\mathcal{E}$ are undirected edges between nodes which are dependant as defined in Definition 8.

Given a causal graph G, a container structure C for the pattern and a start variable v_0 we can now sample the graph for variables. Since Boutilier and Dearden didn't clearly define a stop criteria for their algorithm we have to define one ourselves. So given a constant $size_{\max}$ we limit the maximal output of a given function size which takes as argument the container structure C. This function assigns a value to the magnitude of the pattern. The concrete implementation of this function will be discussed later. The depth of the recursion is also limited. That way we get the dependent variables layer for layer. We define the following pseudo code for the general implementation of this algorithm:

Algorithm 1 Adding variables according to their dependency

```
function EXPLORE(G,C,v_0, depth, depth_max)
    if depth == depth_max then
        return
    end if
    for (v_0, v) ∈ E do
        if size(C) < size_max && v ∉ C then
            add v to C
        end if
    end for
    for (v_0, v) ∈ E do
        if size(C) < size_max && !visited(v) then
            EXPLORE(G, C, v, depth + 1, depth_max)
        end if
    end for
end function
```

We start the evaluation at the given starting nodes and then add all nodes which are dependent on the current node. This is done until the output of the function size given the structure C is bigger than the given limit or the maximal depth is reached. An illustration of this would be the following.
Image there's an instance of a domain where the this is the given causal graph.

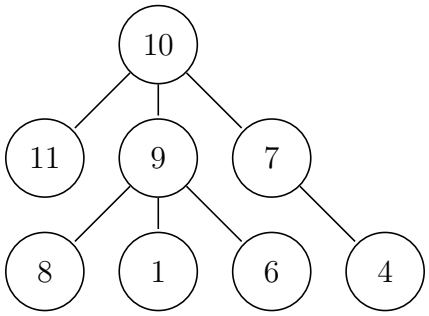

Figure 3.3: Dependency graph modelled as a tree

For this example the size of the pattern is not limited. The initial set consist of $\{v_{10}\}$. So we start out with $v_0 = v_{10}$. After the first iteration of our algorithm our container C holds $\{v_{10}, v_{11}, v_9, v_7\}$. Then we restart the evaluation with $v_0 = v_{11}$ and then with $v_0 = v_9$ and so on. When the algorithm stops all variables are contained in C.
Another way to look at the structure of the transition formulas is through the ordering of the conditions they contain. This is done in the next subsection.

3.2.2 Sequential Addition

Since the conditions in a transition formula are ordered, the evaluation is aborted once a condition holds. So c_n can be evaluated iff $\sim (c_{n-1} \wedge c_{n-2} \wedge \dots c_0)$. Adding the variables $v \in c_0$ is more important than adding the variables $v \in c_1$. So given a initial set of variables one can add dependent variables according to the condition in which they appear. This is then recursively done for all appearing variables until no more variables can be added or the given limit was reached.

Here the focus is on the ordering of the conditions rather then on just the dependency. We can't use the initial structure of the dependency graph since given an edge one can't say to which condition it belongs. This is only a technicality be has to be kept in mind when translating this principle into an algorithm.

Algorithm 2 Sequential addition

```
function SEQUENTIAL ADDITION(C)
    x ← true
    i ← 0
    while x do
        x ← false
        for v_0 ∈ C do
            j ← 0
            while j < i do
                for a ∈ A do
                    if ∃(c_i, p_i) ∈ t_a^{v_0} then
                        x ← true
                        for v ∈ c_i do
                            add (v_0, v) to E
                            if size(C) < size_max && v ∉ C then
                                add v to C
                                if size(C) ≥ size_max then
                                    return C
                                end if
                            end if
                        end for
                    end if
                end for
                j ← j + 1
            end while
        end for
        i ← i + 1
    end while
    return C
end function
```

This algorithm just sequentially searches the conditions of the transition formulas for variables to add to the pattern. The algorithm includes the same number of conditions for all variables.
We now have to methods with which we can, given an initial set of variables, create a pattern. We're still stuck with the problem that this approach needs an initial set of variables contained in the reward formulas in order to work. If this set is not given and we don't want to randomly sample the reward formulas we need to define a method which samples a good abstraction for the set of all variables contained in the reward formulas.
"Among competing hypotheses, the one with the fewest assumptions should be selected." This principle proposed by William of Ockham (1287–1347) suggest that the fewer assumptions you make, the better it depicts the reality. This idea carries on to the an idea proposed by Boutilier and Dearden in their paper. They propose to use spans to evaluate the usefulness of an initial pattern. A span is the difference between the maximal and the minimal value a reward formula can return if we have to guess the state of at minimum one variable. So a higher span indicates

more ambiguity when a smaller span indicates less ambiguity. Therefore this can be seen as the translation of this philosophical principle to planning.

3.2.3 Estimate through the Span

In this context we want to define the span of a pattern as follows.

Definition 10 (Span)**.** We're given

- a probabilistic planning task $\langle \mathcal{V}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{H}, s_0 \rangle$,
- a pattern $\mathcal{V}^P \subseteq \mathcal{V}$ for which $\forall v \in \mathcal{V}^P, \exists s \in S, r_a(s) \neq r_a(s \setminus \{v\})$,
- another pattern $L \subseteq \mathcal{V}$ for which holds $\nexists v, v \in \mathcal{V} \setminus (\mathcal{V}^P \cup L), \exists s \in S, r_a(s) \neq r_a(s \setminus \{v\})$ and
- a set of patterns $PS = \{ps_0, \ldots, ps_n\} = \Pi(L)$.

From this we can calculate the span of a pattern:

$$\text{span}(\mathcal{V}^P, PS) = \frac{1}{\mid \Pi(\mathcal{V}^P) \mid} \sum_{P \in \Pi(\mathcal{V}^P)} \max_{ps \in PS} r_a(P \cup ps) - \min_{ps \in PS} r_a(P \cup ps)$$

So we evaluate all possible values r_a with a focus on a fixed set of variables. For each combination of these variables we can calculate the difference between the maximal and minimal return value. The larger this is the larger is the possible mistake we introduce when using this pattern. Since there is more than one assignment for a variable we have to summarize the information we gain for every possible assignment. Taking the mean is an idea to generalize the obtained information for a pattern. It is important to say that this is not exactly the same technique as proposed by Boutilier and Dearden. They suggest to use the maximal span for the evaluation. This would translate into the formula

$$\text{span}(\mathcal{V}^P, PS)_{BD} = \max_{P \in \Pi(\mathcal{V}^P)} \max_{ps \in PS} r_a(P \cup ps) - \min_{ps \in PS} r_a(P \cup ps)$$

The results obtained with this formula are not as fine grained as with our formula. This can be seen in the evaluation of the example later on. First we want to illustrate the general technique with an algorithm. Given a set $\mathcal{V}^{\mathcal{R}}$ which contains all variables which appear in the reward formulas, and a limitation for the pattern size we can deduct the following algorithm:

Algorithm 3 Evaluation of the span

```
function EVALUATE SPAN(V^R)
    combination ← Π(V^R)
    for c ∈ combinations do
        eval ← Π(V^R \ c)
        if size(c) < size_max then
            if c_best is initialised then
                if span(c_best, Π(eval)) <span(c, Π(eval)) then
                    c_best ← c
                end if
            else
                c_best ← c
            end if
        end if
    end for
    return c_best
end function
```

As for an example, image the following reward formula:

$$r_a(s) = ([v_1]^I + [v_2]^I) \cdot [v_3]^I$$

Here we got 8 possible combinations for the variables.

Pattern	$\text{span}(\mathcal{V}^P, PS)$	$\text{span}(\mathcal{V}^P, PS)_{BD}$
$\{v_1, v_2, v_3\}$	0	0
$\{v_1, v_2\}$	1	2
$\{v_1\}$	2	2
$\{v_2, v_3\}$	0.5	1
$\{v_3\}$	1	2
$\{v_1, v_3\}$	0.5	1
$\{v_2\}$	2	2
$\{\}$	2	2

Figure 3.4: Tabular depicting the span calculated with the given method for the given pattern

The algorithm by Boutilier and Dearden returns only a upper boundary. So some information gets lost and therefore the quality of the very small pattern is not discernible with this approach.
In order to calculate the best pattern with the current algorithm all combinations have to be created. Because of that this is not a good approach for most instances. A more goal driven approach needs an optimization algorithm. We choose to use a hill-climbing approach. Starting from an initial point the algorithm chooses to advance in the direction where the value to optimize changes the most into the right direction. The span evaluation should be done with interval arithmetic to make it feasible. We also don't need to use the full power set. A subset of the aforementioned should suffice. Given $\mathcal{V}_\pi^P \subset \Pi(\mathcal{V}^P)$ we arrive at the following representation for the formula.

$$\text{span}_{\text{interval}}(\mathcal{V}^P, \mathcal{V}_\pi^P) = \frac{1}{\mid \mathcal{V}_\pi^P \mid} \sum_{P \in \mathcal{V}_\pi^P} \text{ov}^{\mathcal{V}^P}(r_a(P)) - \text{ud}^{\mathcal{V}^P}(r_a(P))$$

So the algorithm changes to

Algorithm 4 Improved evaluation of the span

function EVALUATE SPAN IMPROVED($\mathcal{V}^{\mathcal{R}}, c_{\text{current}}$)
  for $c_{\text{new}} \in \mathcal{V}^{\mathcal{R}}$ && $c_{\text{new}} \notin c_{\text{current}}$ **do**
    add $c_{\text{new}} \cap c_{\text{current}}$ to combinations
  end for
  for $c \in$ combinations **do**
    if size(c) $< size_{\max}$ **then**
      $\mathcal{V}^P_{\pi_c} \leftarrow$ subset of $\Pi(c)$
      if c_{best} is initialised **then**
        $\mathcal{V}^P_{\pi_{c_{\text{best}}}} \leftarrow$ subset of $\Pi(c_{\text{best}})$
        if $\text{span}_{\text{interval}}(c_{\text{best}}, \mathcal{V}^P_{\pi_{c_{\text{best}}}}) < \text{span}_{\text{interval}}(c, \mathcal{V}^P_{\pi_c})$ **then**
          $c_{\text{best}} \leftarrow c$
        end if
      else
        $\mathcal{V}^P_{\pi_{c_{\text{current}}}} \leftarrow$ subset of $\Pi(c_{\text{current}})$
        if $\text{span}_{\text{interval}}(c_{\text{current}}, \mathcal{V}^P_{\pi_{c_{\text{current}}}}) > \text{span}_{\text{interval}}(c, \mathcal{V}^P_{\pi_c})$ **then**
          $c_{\text{best}} \leftarrow c$
        end if
      end if
    end if
  end for
  if c_{best} is not initialised **then**
    return c_{current}
  else
    return EVALUATE SPAN IMPROVED($\mathcal{V}^{\mathcal{R}}, c_{\text{best}}$)
  end if
end function

In this Section we have explored a technique to create an initial pattern given the reward formulas.
Together with the former Section we're now able to create a pattern for any given probabilistic planning task. This pattern now can be used by a reductive function to create an abstract probabilistic planning task. The only question that remains is how to properly limit the size of the pattern. This question is addressed in the next section.

3.3 Limit the Pattern

As the computational expenses grow with the number of variables included in our pattern we need to limit the size of the aforementioned somehow. We'd like to introduce two kind measurements which can be used to limit the size of the pattern.
The first measurement restricts the pattern by its concrete size. This is to say that given a limit n we can only add n variables to the pattern. This is quite restrictive since it doesn't account for the fact that the growth of the state space doesn't have to be directly linked to the number of variables added.
In some domains, as for example crossing traffic, the theoretical state space is larger than the real state space. In crossing traffic this is due to the fact that there's one variable for each position of the agent. In consequence we should rather limit our pattern by number of states it implicates than the raw number of variables contained in it.
We search for this restriction by reducing the start state to our pattern and exploring the

abstract state space until there are no new states to be visited any more. This doesn't lead to a big overhead since we in any case need to create all states which can be visited for the calculation of the optimal policy. That way we can always try to add one variable if the limit for the state space is not reached.
The Algorithm 5 illustrates this technique. We define a limit n, a container structure P and a probabilistic planning task PPT.

Algorithm 5 Evaluation of the size

```
function SIZE(n, P, PPT)
    PPT^P ← reduce PPT with P
    size ←EXPLORE STATE SPACE(PPT^P, 2^n)
    return size
end function
```

Given this algorithm we now can better control the computational expenses a pattern implicates.
At the end of this chapter we would like to do a quick recap of how the technique by Boutilier and Dearden, which was introduced in the beginning of this chapter, has been implemented here.

1. Define a limit for the pattern
   We've approached this implicit requirement by translating the number of variables in a pattern into the number of maximal reachable states. This then proved to be an accurate estimate of the computational expense to be expected from an abstract probabilistic planning task which was to be created with this pattern.

2. Use the span to determine the best subset of the variables contained in the reward formulas
   We've approached this problem by the use of an hillclimbing algorithm and interval arithmetic. Through the loss some accuracy we've made this method applicable for automated planning by almost removing the combinatorial part.

3. Recursively add variables which appear in the conditions of the actions associated with the variables contained in the pattern
   Here we've illustrated two techniques. The one by Boutilier and Dearden only focuses on dependency of the variables while the other also takes into account their ordering.

4. Create an abstract probabilistic planning task
   We've proposed three different reductive formulas. The first one tries to ignore the effect of variables which are not contained in the pattern. This approach is usable for the transition formulas and the reward formulas. The other two are just usable for reward formulas. The first one of these tries to overestimate respectively underestimate the reward to be obtained in the real function as opposed to the reduced function. The next one is introduced by Boutilier and Dearden in their paper. It combines the overestimate and underestimate approach of the first method. Both estimates are combined to obtain a better result.

Having implemented methods which create abstractions we do have to ask ourselves how to use these abstractions. Evidently if we had the MDP equivalent of the abstract probabilistic planning task we would want to calculate an optimal policy for it. So the next thing to do is to describe how the idea of the optimal policy translates into algorithms for the abstract probabilistic planning task.

Chapter 4

Heuristic Search

Going back to the Definition 2 and given an MDP $< \mathcal{V}, \mathcal{A}, \mathcal{P}, \Re, \mathcal{H}, s_0 >$, we can define a value $V^\pi(s, d)$ for every state s when there are still d steps to go in the game. Since we induced the MDP with the probabilistic planning task, we basically can calculate the same estimate for the probabilistic planning tasks. When it comes to abstract probabilistic planning tasks we first have to change the nomenclature, since everything we calculate on the abstract is only a guess. Therefore we talk about heuristics. The next section will talk about theses heuristics in combination with the calculation of the optimal policy.

4.1 Heuristics

When talking about optimal policy calculation in connection with heuristics it is important to keep the following thin in mind: We have to discern between $V_P^\pi(s, d)$ which is the heuristic value obtained in an abstract probabilistic planning task defined by pattern P and $V^*(s, d)$ which is the heuristic value one would obtain in the original form of this task for the same input. Speaking more general we can say the following about a Heuristic.

Definition 11 (Heuristic)**.** Given a probabilistic planning task $< \mathcal{V}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{H}, s_0 >$ a heuristic $h(s, d)$ is a function $h : S \times \mathbb{N} \mapsto \mathbb{R}$ which estimates the reward to be obtained in s with d steps to go. The real value to be obtained is given by $h^*(s, d)$.

So in general a heuristic is a guess for the reward to be obtained. It can be the value obtained by the optimal policy, but it doesn't have to be exactly that value. For the time being we assume that $h(s, d) = V_P^\pi(s, d)$ and $h^*(s, d) = V^*(s, d)$. The following will rely on the definition of the optimal policy.
When we apply certain techniques to calculate $V_P^\pi(s, d)$ we can guarantee that the upper respectively lower bound of this value is $V^*(s, d)$.

Lemma 1. Given an abstract probabilistic planning task $\langle \mathcal{V}^P, \mathcal{A}, \mathcal{T}^P, \mathcal{R}^P, \mathcal{H}, s_0 \rangle$ we can say that

- $\forall s, d\ V_P^\pi(s, d) \geq V^*(s, d)$ if $\mathcal{R}^P$ was reduced with the overestimation approach and
- $\forall s, d\ V_P^\pi(s, d) \leq V^*(s, d)$ if $\mathcal{R}^P$ was reduced with the underestimation approach.

Proof. **This only represents a proof sketch**.
For this sketch we assume that $re_a(s) = r_a(s)$ and that $p_a(s, s')$ is induced as described earlier in Definition 6 by t_a. This lemma can be proven over induction. For the basis we assume that $d = 0$ so for any state s

$$\begin{aligned} V^*(s, 0) &= \max_a r_a(s) \\ V_P^\pi(s, 0) &= \max_a ov^P(r_a)(s) \text{ respectively} \\ V_P^\pi(s, 0) &= \max_a ud^P(r_a)(s). \end{aligned}$$

Because $ud^P(r_a)(s) \leq r_a(s) \leq ov^P(r_a)(s)$ our lemma holds for $d = 0$. For the induction we now assume that $d = n + 1$

$$V^*(s, n+1) = \max_a r_a(s) + \sum_a p_a(s, s') \cdot V^*(s, n)$$

We already know that the Lemma holds for each $r_a(s)$ so we just have to look at $\sum_a p_a(s, s') \cdot V^*(s', n)$. The transitions probabilities stay the same and point to the reduced form of their respective $V^*(s', n)$. In the reduced form it can happen that some transitions are counted twice but $\sum_a p_a(s, s') = 1$ always holds. From our basis we know that for each reduced $V^*(s', n)$ our Lemma holds. So it holds for $V^*(s, n + 1)$ □

Given this lemma one can even go further and think about the combination of theses two boundaries. In the following we'll describe $V_P^\pi(s, d)$ as $V_{P_{max}}^\pi(s, d)$ if its corresponding probabilistic planning task was reduced with the overestimate approach and $V_P^\pi(s, d)$ as $V_{P_{min}}^\pi(s, d)$ if its corresponding probabilistic planning task was reduced with the underestimate approach. When our approach is bounded we can safely assume

$$\begin{aligned} V_{P_{max}}^\pi(s, d) &= V^*(s, d) + x_1 \mid \ x_1 \in \mathbb{R} \text{ and} \\ V_{P_{min}}^\pi(s, d) &= V^*(s, d) - x_2 \mid \ x_2 \in \mathbb{R}. \end{aligned}$$

we introduce a mixing factor $x_3 \in [0, 1]$

$$x_3 \cdot V_{P_{max}}^\pi(s, d) + (1 - x_3) \cdot V_{P_{min}}^\pi(s, d) = x_3 \cdot (V^*(s, d) + x_1) + (1 - x) \cdot (V^*(s, d) - x_2)$$

To obtain the real reward we want it to equal $V^*(s, d)$ and we want to solve the equation for the mixing factor x_3

$$\begin{aligned} V^*(s, d) &= x_3 \cdot (V^*(s, d) + x_1) + (1 - x_3) \cdot (V^*(s, d) + x_2) \\ x_3 &= \frac{x_2}{x_1 + x_2} \end{aligned}$$

So our factor is dependent on the magnitude of the error of the underestimation respectively overestimation technique. This factor can be sampled from a training set. With a training set which is big enough the factor should be representative. This method is only useful if for a given x_3

$$\mid x_3 \cdot (x_1 + x_2) - x_2 \mid < \min(x_1, x_2)$$

holds.
having explored some properties of the heuristics in connection with the optimal policy calculation we now want to define a method which assigns every combination of state and steps to go an heuristic value.

4.2 Single Patterns

This section describes a simple method with which one can calculate the heuristic values for a probabilistic planning task.

Definition 12 (Value iteration)**.** Given a probabilistic planning task $\langle \mathcal{V}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{H}, s_0 \rangle$, $p(s, s')$ as induced earlier and a set $\widehat{S} \subseteq S$ which holds all reachable states, we can define the following algorithm

1. $\forall s \in \widehat{S}, h(s, 0) = 0$
2. $\forall s \in \widehat{S}, h(s, d+1) = \max_{a \in \mathcal{A}} r_a(s) + \sum_{s'} p_a(s, s') \cdot h(s, d)$

Repeat two until $d = \mathcal{H}$

Given these estimates we now look for the best action in each state s with d steps to go which produces the highest h(s,d). This simple technique can also be used to evaluate abstract probabilistic planning tasks. One has to replace the reward and transition formulas with their reduced equivalent. The reduced transition formulas then again induce their respective $p_a(s, s')$. If we want to improve the accuracy of the given method for abstract probabilistic planning tasks we can make use of one of the properties described in the section Heuristic. The property which seems the most useful is the one which describes the possibility of combining two estimates with each other. That way we can, given a sampled x and a $p_a(s, s')$ induced by t_a^P, reformulate the value iteration formula as

$$\begin{aligned}
&\forall s \in \widehat{S}, h(s, 0) = V^{\pi}_{P_{max}}(s, 0) = V^{\pi}_{P_{min}}(s, 0) = 0 \\
&\forall s \in \widehat{S}, V^{\pi}_{P_{max}}(s, d+1) = \max_{a \in \mathcal{A}} ov(r_a)(s) + \sum_{s'} p_a(s, s') \cdot V^{\pi}_{P_{max}}(s, d) \\
&\forall s \in \widehat{S}, V^{\pi}_{P_{min}}(s, d+1) = \max_{a \in \mathcal{A}} ud(r_a)(s) + \sum_{s'} p_a(s, s') \cdot V^{\pi}_{P_{min}}(s, d) \\
&\forall s \in \widehat{S}, h(s, d+1) = x \cdot V^{\pi}_{P_{max}}(s, d+1) + (1 - x) \cdot V^{\pi}_{P_{min}}(s, d+1)
\end{aligned}$$

This approach just approximates $h(s, d)$ to be as close as possible to $V^*(s, d)$. Given this we could already start testing our implementation. But one problem we still have left unaddressed. What can one do to improve his coverage of variables when the task contains 100 variables but it is only feasible to include 12 variables in a pattern. An approach for this is to use multiple patterns. This is to say we sample the probabilistic planning task for more than one pattern. Each of theses patterns creates another abstract probabilistic planning task. The informations obtained from the single abstract tasks are then recombined to better be able to solve the original task. The evident question here is how to create and make use of these patterns.

4.3 Multiple Patterns

In this chapter we'd like to introduce two possible options to create multiple patterns. The canonical heuristic is reliant on independence while the other technique is always applicable.

4.3.1 Canonical Heuristic

As proposed in [3] canonical heuristics can be used as a system to evaluate multiple patterns.

Definition 13 (Canonical Heuristic)**.** A canonical heuristic is an heuristic as given in Definition 11 with the function $h^C(s,d) = \max_{J \in A} \sum_{P \in J} V^{\pi}_{P_{max}}(s,d)$ for

- $A = \{J_0, \ldots, J_n\}$ is a set of maximal cliques where
  - J is a set of patterns $\{P_0, \ldots, P_n\}$ where all variables $v_k \in P_i$ are independent of all variables $v_l \in P_j \mid j \neq i$. This Definition for Independence used here is the one from Definition 14.

The heuristic adds up the progress towards independently reachable goals. Variables are independent in the sense of the canonical heuristic if they're independent as defined in Definition 15.

Definition 14 (Independence in Canonical Heuristics)**.** Given a probabilistic planning task, $indep(v_i, v_j)$ holds if v_i is independent of v_j with $i \neq j$. This holds if $\nexists a, s$ where

$$(p(s, v_i, a) > 0 \mid v_i \notin s \text{ or } p(s, v_i, a) < 1 \mid v_i \in s)$$

and

$$(p(s, v_j, a) > 0 \mid v_j \notin s \text{ or } p(s, v_j, a) < 1 \mid v_j \in s)$$

holds.

So a variable is dependent on another variable if there exists a combination of a state and an action which induces a change to both variables. As for an example:

- $\mathcal{V} = \{v_0, v_1, v_2, v_3\}$
- $\mathcal{A} = \{move_1, move_2, take_1, take_2\}$
- $\mathcal{T} = \{t_{move_1}, t_{move_2}, t_{take_1}, t_{take_2}\}$ where
  - $t_{move_1}(s) = \{\{t^{v_0}_{move_1} = 1\}, \{t^{v_1}_{move_1} = 0\}, \{t^{v_2}_{move_1} = v_2\}, \{t^{v_3}_{move_1} = v_3\}\}$
  - $t_{move_2}(s) = \{\{t^{v_0}_{move_2} = 0\}, \{t^{v_1}_{move_2} = 1\}, \{t^{v_2}_{move_2} = v_2\}, \{t^{v_3}_{move_2} = v_3\}\}$
  - $t_{take_1}(s) = \{\{t^{v_0}_{take_1} = v_0\}, \{t^{v_1}_{take_1} = v_1\}, \{t^{v_2}_{take_1} = 1\}, \{t^{v_3}_{take_1} = v_3\}\}$
  - $t_{take_2}(s) = \{\{t^{v_0}_{take_2} = v_0\}, \{t^{v_1}_{take_2} = v_1\}, \{t^{v_2}_{take_2} = v_2\}, \{t^{v_3}_{take_2} = 1\}\}$

The only things that depend on each other are obviously the locations of the agent. Now we can determine two patterns $P_1 = \{v_2\}$ and $P_2 = \{v_3\}$. These two patterns can be added and contribute to a more precise result. Informally speaking does the progress in one goal doesn't advance you towards the other goal and so adding both advancements results in more precise guess for the total advancement.
The problematic part is now to prove independence between variables since this is almost never as obvious as in these transition formulas. Finding independent variables can be very complicated and in any case computationally expensive. With our methods we were not able to prove a sufficient number of independent variables for any instance of any domain given. So this approach proves to be a dead end, at least for our setting. The general idea can still be of use, though. We just have to not rely on independence. This exact train of thought is followed in the next subsection.

4.3.2 Multiple Patterns without Independence

Going back to the general idea of multiple patterns, we can say that it was born out of the need to cover more variables while still keeping the computational effort low. So the simplest approach for this would be the following: The biggest union with the least computational effort is created when we create a pattern for every variable. Given n variables the effort reduces from 2^n to $2n$. Sadly this problem can't be solved as easily. The minimum requirement for a pattern is that is at least contains one variable which appears in the reward formula. Otherwise heuristic values all evaluate to 0.
So we just make a pattern for every reward variable and the add the remaining variables to the aforementioned. This is not quite optimal either since one has to keep in mind that as long as the union of two patterns is feasible it is better to calculate a solution for the union than for the single patterns. This is true for obvious reasons.
The next thing one could argue is that we just have to distribute all variables equally to patterns which are small enough and contain some reward variables so that we don't surpass our computational limit and still have sufficiently sized patterns.
This is a good idea but we can't do this distribution arbitrarily. One variable is important for the evaluation for another variable if it appears in one of its conditions. So we're back at the point where we started from with the idea by Boutilier and Dearden.
Summing up: given a probabilistic planning task $< \mathcal{V}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{H}, s_0 >$, a set of patterns $PD = \{\mathcal{V}_0^P, \ldots, \mathcal{V}_m^P\}$, an initial pattern $\mathcal{V}_{\text{init}}^P$ which is of maximal size and the dependency graph G we want the set of patterns to full fill the following properties as good as possible:

- Maximize the coverage, $max_{PD} \cap_i \mathcal{V}_i^P$.
- The combination of any pattern should not be feasible, $\Theta(V_i^P \cap V_j^P) > \Theta(\mathcal{V}_{\text{init}}^P) \mid i \neq j$.
- Any variable v_i within a pattern should be reachable from a another variable v_j within the same pattern V_i^P through the dependency graph G without having to pass a variable $v_k \notin V_i^P$.

Keeping these properties in mind we propose the following algorithm to create multiple patterns without independence. For this algorithm we assume that we already have an initial maximal sized set $\mathcal{V}_{max}^P$, a dependency graph G and a set $\mathcal{V}^{\mathcal{R}}$ which contains all variables which appear in the reward formulas.

Algorithm 6 Creation of Multiple Patterns

```
function MULTIPLE PATTERNS(V^P_max, G, V^R)
    initialise pattern collection PD
    for v ∈ V^P_max do
        if v ∈ V^R then
            remove v from V^P_max
            create pattern with v and add it to PD
        end if
    end for
    while V^P_max is not empty do
        for v ∈ V^P_max do
            for pd ∈ PD do
                if G contains an edge from any variable v̂ ∈ pd to v then
                    remove v from V^P_max
                    create pattern with v and add it to PD
                    break loop
                end if
            end for
        end for
    end while
    fuse patterns if the resulting PD is still feasible
    while true do
        for pd ∈ PD do
            add variable to pd with Boutilier and Dearden method
            fuse patterns if the resulting PD is still feasible
            if Σ_{pd∈PD} size(pd) ≥ size_max then
                return PD
            end if
        end for
    end while
end function
```

The only question which remains is how to include these multiple patterns into our calculation for $h(s,d)$. In order to do this we can again go back to the Lemma 1 introduced in the section Heuristic. Say we had two estimates $V^{\pi}_{P^0_{max}}(s,d)$ and $V^{\pi}_{P^1_{max}}(s,d)$. Then we could safely assume that given $V^{\pi}_{P^0_{max}}(s,d) \geq V^{\pi}_{P^1_{max}}(s,d) \rightarrow V^{\pi}_{P^0_{max}}(s,d) \geq V^{\pi}_{P^1_{max}}(s,d) \geq V^{*}(s,d)$. So $V^{\pi}_{P^1_{max}}(s,d)$ is a better estimate in this case. Given we have already calculated the overestimating optimal policy for all $\mathcal{V}^P_i \in PD$ which we'll denote with $V^{\pi}_{P^i_{max}}(s,d)$ we can say that

$$V^{\pi}_{PD_{max}}(s,d) = \min_i V^{\pi}_{P^i_{max}}(s,d)$$

The same thing can be done with $V^{\pi}_{P^i_{min}}(s,d)$ with the obvious adjustments. We again can arrive at two estimates. These two estimates could again be combined to:

$$\forall s \in \widehat{S}, h(s,d+1) = x \cdot V^{\pi}_{PD_{max}}(s,d+1) + (1-x) \cdot V^{\pi}_{PD_{min}}(s,d+1)$$

To bring this chapter to a conclusion we'd like to summarize the things we've introduced here. At the beginning of the chapter we've asked ourselves what to do with the abstract probabilistic planning tasks we can create now.
We've first approached this question from a more general viewpoint by defining Heuristics

in general. These Heuristics do estimate the reward to be gained starting from a state with a limited number of steps to go. The structure of our reduction in combination with the algorithm which creates the optimal policy can be exploited to create a good estimates for the real reward to be obtained. Then we've described in general how this can be used for a single abstract probabilistic planning tasks. From there on out we've tried to also approach the idea of creating more than one pattern and therefore covering more variables. Our first approach failed because it relies on fact that some of the reward variables are independently influenceable. With the algorithms used by us to this point we can't satisfy this requirement for any domain. The next thing we've tried is to create more than pattern without this requirement. This led to 3 requirements which should be optimized in order to get a good collection of patterns. We tried to incorporate this in the algorithm following thereafter.
Having the possibility to now create one or multiple abstract probabilistic planning tasks from one probabilistic planning task and to calculate a solution with both approaches we're now ready to evaluate the results obtained by our implementation.

Chapter 5

Evaluation

In this chapter we first will introduce the parameter with which our implementation was tested. Thereafter we'll present some results obtained with our techniques. These results will be obtained by the use of our techniques as a standalone planner and a heuristic for an A^* algorithm.

5.1 Identification of Parameters

When testing an approach it is important to identify and evaluate a restricted set of parameters. This guarantees comparability towards former set-ups. It helps to identify weaknesses and strengths of an approach. In general we always used the pattern creation method from Boutilier an Dearden when it came to the creation of single patterns. The other parameters were interchangeable as follows.

1. Use of a single pattern or multiple patterns
   Valuerange :$\{true, false\}$
   It can be chosen whether to just evaluate one pattern and therefore get access to all the evaluation methods or whether to work with multiple pattern and in consequence get the advantage of being able to compare different patterns with each other. Here the canonical heuristic is presumably the best way to deal with multiple patterns. Since it is not applicable we use our algorithm. This should yield promising results for sufficiently sized problems. The use of multiple patterns is indicated with the identifier **mP**.

2. Number of variables to add
   Valuerange : $\mathbb{N}$
   Evidently the more variables you include in you pattern, the better your policy gets. But of course this number has to be limited because of the exponential growth of possible states. This parameter nV limits the number of states to 2^{nV}.

3. Abstract reward formula evaluation method
   Valuerange :$\{Dearden, Ignore, Overestimate, Underestimate\}$

   - Dearden $\longrightarrow$ Evaluationmethod as described in [1] with the change that we adapt the factor with a sampled value. The use of this method is indicated with the identifier **bD**
   - Ignore $\longrightarrow$ Reduced rewardformula where variables that are not contained in the pattern are ignored. The use of this method is indicated with the identifier **Ignore**
   - Overestimate $\longrightarrow$ Intervalarithmetic evaluation of the reward. The maximum value is taken. The use of this method is indicated with the identifier **ov**.

Choosing this is a important part since this influences the whole computation which comes thereafter.

5.2 Performance

Calculations were performed at sciCORE (http://scicore.unibas.ch/) scientific computing core facility at University of Basel. The calculations are compared to the result two results. We choose to compare it to the current version of the UCTStar algorithm which

- in the setting described with the identifier **IDS** uses an iterative deepening search algorithm as a heuristic and
- in the setting described with the identifier **Uniform** uses an uniform evaluation search algorithm a heuristic.

Rather than presenting average rewards, we show normalized scores which were averaged over 100 cycles of tests. The tests were not realized over all instances of the 12 domains used in the IPPC. We've excluded the results form the path-finding domains because it is well known that they are optimally solvable with invariance synthesis and haven't been solved well with other approaches similar to ours. This leaves us with 8 domains.
Beforehand the difference between this evaluation methods has to be underlined. This refers to the kind of emphasis one places on a method. While we always want a optimal result it is preferable for a heuristic to be calculated fast in order to maximize the profit gained from the abstraction. Here we search for a trade-off with emphasis on the speed. A heuristic is of no use when it takes 30 minutes to calculate its solution. This is opposed to the setting when one devises a standalone planner. Here the emphasis lays rather on the result (the score obtained) than on the speed. We have to make sure that any planner uses a similar amount of time to make them comparable. The maximal size of states to evaluates is set to 2^{12} which has proven to be a good limit to evaluate the performance. The time consuming part is the part where the value iteration on the abstract tasks is performed. From there on out the only thing this technique does is making look-ups and recalculating some results if need be.
The maximal results obtained in the instance are marked red.

5.2.1 As a Standalone Planner

The best performance could evidently be reached when including more variables into the calculation. To stand a fair chance against other planner we successively added variables to the pattern until we obtained a similar calculation time as the established ones. This enabled our planner to solve the small instances optimally and give a better estimate for the big instances. The thing we've noticed is that given a good estimate the approach inspired by Boutilier and Dearden produced one of the best results. Again if this estimate was bad the estimate was never worse than just the overestimation approach. These both were only tested with multiple patterns. The simple "Ignore method" had one of the most stable performances. This was probably due to the fact that the actual reward is already well approximated when just ignoring the influence of the variables which are not contained in the pattern. The acceptable results in sysadmin and game of life could be explained by the fact that the span evaluation seems to abstract the given domains quite good. Other domains where we also had to include the method which sampled the dependency graph only delivered meagre results. We at least can argue that, only taking into account the domains defined by us, our planner acts informed. This we derive from the fact that the standalone planners performance is better in total than that of the UCTStar with the uniform search as a heuristic.

	wildfire	academic	elevators	tamarisk	sysadmin	recon	game	skill	Total
UCTStar [IDS]	0.75	0.32	**1.0**	**1.0**	0.98	**1.0**	0.99	0.99	0.87
UCTStar [Uniform]	0.11	0.14	0.77	0.35	0.34	0.95	0.48	0.76	0.48
SAFT bD mP	0.74	0.4	0.48	0.47	0.86	0.0	0.64	0.42	0.5
SAFT ignore	0.91	0.48	0.29	0.64	0.81	0.0	0.63	0.44	0.52
SAFT ov mP	0.74	0.4	0.48	0.44	0.86	0.0	0.64	0.42	0.49

Figure 5.1: IPPC Score obtained with the given parameters. The total is based on the given 8 domains.

We now go on to evaluate our approach as a heuristic.

5.2.2 As a Heuristic

Here we did use our algorithm as a heuristic for the UCTStar implementation. The performance was not as good as in the standalone planner. This was amongst other things due to the fact that the lookups weren't fast enough. These lookups aren't as fast as they could be due to the fact that we use maps to store our heuristic values. The necessity to use maps comes from the combination of two techniques. The first one is the use of a perfect hash. This means since we can describe any state in a binary manner we can convert the binary number to a decimal. This again can be used as a hashkey. This for itself is a good technique, but in combination with the way we limit the size of our pattern it leads to rather poor results. Since we limit the size of our pattern by number of states it can create we can get patterns of a big size while still having a small state space. So for example if the size of the pattern was 32 and its state space was only of the size 80 we still would have to reserve 2^{32} times the size of the data type of the heuristic value. This obliges us to use maps from a certain point on. When already handicapped in that area we simply can't expect to stand a chance against the established planners.

	wildfire	academic	elevators	tamarisk	sysadmin	recon	game	skill	Total
UCTStar [IDS]	0.81	0.32	0.99	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	0.89
UCTStar [Uniform]	0.11	0.14	0.77	0.35	0.34	0.95	0.49	0.76	0.55
UCTStar [SAFT bD mP]	0.45	0.39	0.34	0.14	0.57	**0.96**	0.54	0.49	0.47
UCTStar [SAFT ignore]	0.53	0.24	0.4	0.35	0.49	0.25	0.56	0.51	0.41
UCTStar [SAFT ov mP]	0.19	0.19	0.36	0.45	0.57	0.22	0.53	0.63	0.39

Figure 5.2: IPPC Score obtained with the given parameters. The total is based on the given 8 domains.

After having evaluated the different settings for the planner we want to give an outlook on techniques which may improve the evaluation even further and could be explored in some future works

Chapter 6

Outlook

This chapter will contain a short list one might consider for further improvement of our works. The section thereafter will contain some concluding remarks.

6.1 Possible Improvements

- **Invariance analysis (Invariant Synthesis)**
  In some domains the combinatorial state space is truncated heavily through the transitions. Imagine we have a state space S . Now we restrict this state space by a function $f : S \mapsto \{true, false\}$ where f is defined implicitly through the transitions. This leads to $\widehat{S} = \{s_0, \ldots, s_n\} \mid \forall s, f(s) = true$. This leads $\mid \widehat{S} \mid \leq \mid S \mid$ therefore $\Theta(\widehat{S}) \leq \Theta(S)$. Going about this on the transition level is an advancement as opposed to the brute force method we've applied so far.
  One idea is to search with different real world restrictive functions and whether they are applicable to our problem.Take for example the crossing traffic problem. Here the agent can only be in one place at a time as we know this to be a fact in moving problems. So a problem of this kind with originally 10 variables describing the location of the agent (inducing 2^{10} combinations) facilitates to variable with 11 different assignments if no obstacles are included.
  Implementing this would lead to

  1. performance boost via truncated state space,
  2. optimal solutions for many path finding problems in a reasonable amount of time,
  3. widened range of independent problems and

- **Simplification of the Logical Expressions**
  The main problem with interval arithmetic is that it's not very precise and the imprecision also affects the proportion of the heuristics. This again leads to worse results. An approach for that problem would be to analyse the expressions (transition and reward formulas) and simplify them in a way that every variable appears the least possible in the formula. Maybe one could go about this with a BDD structure. This for itself is a NP-Hard problem which can be (or rather has to be) tackled with heuristics [6]. In general a simplification and the following improvement of the estimate can look like this.

$$\begin{aligned} f(x) =& x - x * 4y & \mapsto& x(1 - 4y) \\ & [-4, 1] & \mapsto& [-3, 0] \end{aligned}$$

- **Selective imprecision**
  When it comes to abstractions we're only interested in a rough picture. So when we calculate the optimal policy for a state we could explore the possibility of ignoring transitions which happen only with a certain probability. While doing the value iteration we would need to assure that the dynamic of the system would not be disturbed or only marginally altered. If we achieved this it would be possible to include more variables into the pattern without sacrificing too much accuracy. As this is probably instance dependent we could learn the minimal transition probability from a small training set and the include it in the actual big one.

These are only some ideas which may be applicable to area of problems treated in this thesis. Now to bring this thesis to an end some concluding remarks.

6.2 Concluding Remarks

We have started this thesis by introducing the idea by Boutilier and Dearden with an example. From this example we identified three explicit and one implicit step. These were in order of application: Set a limit for your pattern, create an initial pattern from the reward formula(s), with this initial pattern as a basis create your final pattern by looking at the dependency of the variables and then reduce the initial task with the pattern.
We then elaborated these steps with a bottom up approach. An important part of this elaboration was to make method proposed by Boutilier and Dearden for the creation of the initial set feasible. The most important part was however the investigation of the different methods with which one could reduce a task given a pattern. The findings from this section were later reused and led to some interesting results.
After having resolved all issues and therefore having obtained an reduced task, we then turned to the question what to do with such a task. With the definition of the optimal policy we derived a more general approach to assign an evaluation to a state depending on the steps we still could make at this point of the game. In combination with the proper reduction method it could be shown that this evaluation for an abstract state could be bounded upwards or downwards by the evaluation of the original state it was derived from.
We then introduced a technique to calculate an evaluation for an abstract task. We then made use of the general properties defined earlier to obtain a more accurate result by combining the upper and lower bound evaluation. Having up until this point only worked with single patterns we started thinking about how to cover more variables. This led us to the idea of multiple patterns. Here we first tried to implement the canonical heuristic [3] which failed because we couldn't find enough independent variables. In consequence we had to think of a method which still used multiple patterns but didn't rely on independence.
This was then achieved in the next step where following criteria were imposed on the pattern collection: Maximize the coverage, the combination of any pattern should not be feasible and any variable within a pattern should be reachable from a another variable within the same pattern through the dependency graph without having to pass a variable which is not in the pattern. These criteria were supposed to maximize the effect of the pattern collection without having to rely on independence. Having elaborated an algorithm which tried to satisfy these criteria and combined the patterns on the basis of the properties elaborated earlier in this chapter we then moved on to the evaluation. After having evaluated our approach we've noticed that the results produced by our algorithm were acceptable but that there was still room for further improvements. These were then addressed in the last part. Here we would expect a great improvement if we could implement the feature of the invariance analysis.
Finally we'd like to say that, in our opinion, the challenge for further works on this concept

will be to keep the duality of heuristic and standalone planner intact to provide a wide range of possibilities for the future of this concept.

Bibliography

[1] Craig Boutilier and R. Dearden.
Using abstractions for decision-theoretic planning with time constraints, 1994.

[2] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson.
Planning with deadlines in stochastic domains, 1993.

[3] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig.
Domain-independent construction of pattern database heuristics for cost-optimal planning, 2007.

[4] Ronald A. Howard.
Dynamic probabilistic systems, 1971.

[5] Thomas Keller and Malte Helmert.
Trial-based heuristic tree search for finite horizon mdps, 2013.

[6] Sanjay Kulhari Michael Rice.
A survey of static variable ordering heuristics for efficient bdd/mdd construction.

UNIVERSITÄT BASEL

PHILOSOPHISCH-NATURWISSENSCHAFTLICHE FAKULTÄT

Erklärung zur wissenschaftlichen Redlichkeit
(beinhaltet Erklärung zu Plagiat und Betrug)

Bachelorarbeit / ~~Masterarbeit~~ *(nicht Zutreffendes bitte streichen)*

Titel der Arbeit *(Druckschrift)*:

~~ABB~~ Abstractions in probabilistic planning

Name, Vorname *(Druckschrift)*: Maximilian Linus Grüner

Matrikelnummer: 13-061-007

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe.

Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Diese Erklärung wird ergänzt durch eine separat abgeschlossene Vereinbarung bezüglich der Veröffentlichung oder öffentlichen Zugänglichkeit dieser Arbeit.

☐ ja ☒ nein

Ort, Datum: Basel, ~~Januar~~ 20.1.2016

Unterschrift:

Dieses Blatt ist in die Bachelor-, resp. Masterarbeit einzufügen.

UNI BASEL