TU WIEN Informatics

Balancing Learning and Planning

A Case Study on Combining Reinforcement Learning With Answer Set Programming in the Blocks World

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Elias Datler
Registration Number

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Thomas Eiter
Assistance: Dipl.-Ing. Johannes Oetsch

Vienna, 14th August, 2020

Elias Datler	Thomas Eiter

Abstract

The aim of this work is to explore the effectiveness of combining reinforcement learning with answer set programming, where the latter is used to both represent the underlying Markov decision processes of the learning algorithm as well as to generate plans. From these plans we extract actions of high value, thereby trying to accelerate the reinforcement learning process, and we measure the effects of different planning schedules when solving problems in the well-known blocks world domain. Furthermore, we give a detailed description of the proposed framework and test it on blocks worlds of varying sizes to quantify the framework's performance.

Contents

Abstract **iii**

1 Introduction **1**

2 Preliminaries **3**

2.1 Answer Set Programming 3

2.2 Reinforcement Learning 5

2.3 The Blocks World 11

3 The Framework **17**

3.1 The Basic Framework 17

3.2 The Planning Component 23

4 Experiments **27**

4.1 The Trial-and-Error Based Agent 27

4.2 The Planning Agent 29

5 Findings and Discussion **33**

5.1 General Observations 33

5.2 Specific Observations 34

6 Related Work **39**

7 Conclusion **41**

Bibliography **43**

CHAPTER 1

Introduction

Reinforcement learning [SB18, Sze10] is an active field of research within machine learning that relies on a scalar reward signal to learn what action is best to be performed by an agent in a given state. There are reinforcement learning algorithms using a so-called *model* of the world for predicting the effects of actions, and other, *model-free* algorithms, learning only from direct interactions with the environment.

Most reinforcement learning methods use trial-and-error based strategies to explore state spaces. However, if states of high value are sparse and hard to discover, learning is usually slow. In this work we examine whether answer set programming [BKI14, BET11, EIK09, Lif08, Lif19], a highly declarative form of programming aiming to solve computationally hard search problems, can aid a trial-and-error based reinforcement learning method in finding policies in large state spaces.

In our framework, the use of answer set programming is twofold. First, we use it to simulate the agent's environment, that is, our system outsources the computation of state transitions, rewards, and possible actions the agent might perform to an answer set solver. Second, answer set programming is used to search for actions that lead to states of high value at decision time, thereby combining reinforcement learning with planning. The best action found by the answer set solver is then "recommended" to the agent, which either chooses to perform that action, or performs another action it found to be more effective. Furthermore, we consider different invocation strategies of this planning component as well as different settings for the planning horizon, which acts as an upper bound on the search depth in terms of the number of consecutive actions.

A system similar to ours was developed by Nickles [Nic12a, Nic12b], where a reinforcement learning software system using answer set programming to represent the environment was introduced. However, Nickles does not use answer set programming for planning as we do; for more information, see Chapter 6.

Although in this thesis the answer set program has access to a perfect model of the environment, our approach also works if that model is incomplete or (partially) faulty. This is possible because the agent still keeps exploring the state-action space, learns only from real experience, and can correct errors resulting from the model.

The main contribution of this work is thus a framework that enables the combination of reinforcement learning with answer set programming. For our reinforcement learning algorithm an extended version of a first-visit Monte Carlo Exploring Starts method presented in the book by Sutton and Barto [SB18] is used.

To evaluate our framework, we use the widely recognized NP-complete blocks world domain to generate problems of varying complexity. Based on these experiments we then try to empirically answer the following questions:

1. How does our framework perform at solving different blocks world sizes?
2. How does our framework scale when presented with bigger problem instances?
3. Can the planning component aid the agent in solving bigger blocks worlds?
4. How do different planning strategies influence the learning behavior?
5. How does the planning horizon influence the convergence speed of the learning?

To answer these questions, a purely trial-and-error based agent is compared to an agent using the planning system, and the effects of different planning strategies on the learning progress are examined.

One of the main insights gained in this work is that the use of occasional planning can greatly improve the convergence speed to optimal policies. Moreover, we found that using the planning component in states which were never visited before is effective, even if the plans generated are rather shortsighted.

After discussing preliminary topics in Chapter 2, we introduce our framework in Chapter 3. The experiments are described in Chapter 4 and discussed in Chapter 5. Lastly, related work is presented in Chapter 6, and Chapter 7 concludes our work.

CHAPTER 2

Preliminaries

Before describing the main framework developed in this thesis in more detail, we first discuss preliminary topics, which are of relevance for understanding the forthcoming chapters.

2.1 Answer Set Programming

Answer set programming is a highly declarative form of programming oriented towards solving computationally difficult, NP-hard search problems [Lif08]. Rooted in artificial intelligence and computational logic research, answer set programming is a rather new programming methodology used across a vast array of disciplines in science and technology, including planning and causal reasoning, constraint satisfaction problems as well as bioinformatics, to name a few examples.

While traditional, *imperative* programming languages steer the programmer towards implementing an algorithm that describes *how* to solve a problem, the *declarative* approach follows a different idea: the programmer should only describe *what* properties the solution should possess, and not come up with an algorithm on how to get to that solution. Furthermore, answer set programming is also different from *functional* programming, a large group within declarative programming, as it is mostly attributed to the paradigm of *logic* programming [Lif19].

2.1.1 Syntax of Answer Set Programs

The key components of answer set programs are *atoms, literals* and *rules*, where atoms are either true or false, and literals are atoms (e.g. a) and their negations ($not\ a$). Answer set programs consist of a finite set of rules, where rules are expressions of the form

$$a \leftarrow b_1, \ldots, b_m,\ not\ c_1, \ldots,\ not\ c_n,$$

where a, all b_i's and all c_i's are atoms.

For example, an answer set program might look like the following:

$$start\_car \leftarrow fueled,\ not\ broken. \tag{2.1}$$

This answer set program makes use of the literal *broken* in its negated form. *Default negation*, as expressed by the operator *not*, semantically means a statement only evaluates to true if we cannot derive the atom connected to it, or as in this case, the atom *broken*.

While the concept of default negation is used to represent the absence of information, *strong negation*, denoted by "¬", expresses the presence of "negative information". If we were to replace the default negation with strong negation in Listing 2.1, $\neg broken$ would have to be derivable in order to justify the rule. Note that strong negation is syntactic sugar, since every program P can be replaced by a semantically equivalent program P' not containing strong negations, but containing new atoms representing the negated forms of their counterparts [BET11].

Putting it all together, the *head* of the rule in Listing 2.1 ($start\_car$) may only be derived if the so-called *body*, sitting on the right side of the arrow, also holds. The arrow, connecting the head of a rule to its body, is sometimes referred to as the *neck*, and can be read like the mathematical implication, where the left side may only be derived if the right side holds. Concretely, this means that only if $fueled$ can be consistently assumed, and we have no evidence that *broken* holds, we may also derive $start\_car$.

If at a later time, the program in Listing 2.1 would be extended by the addition of a rule enabling the derivation of *broken*, we would not be able to derive $start\_car$ anymore. The retraction of already derived conclusions is very common in answer set programming and gives rise to the *nonmonotonic* behavior of answer set programs.

2.1.2 Semantics of Answer Set Programs

Formally, an interpretation is a consistent set of literals. An interpretation M is an *answer set* of a program P if and only if it is a *subset-minimal model* of P^M, where P^M is the *Gelfond-Lifschitz reduct* of P [GL88].

To get a better understanding of the notion of answer sets, consider the following program:

$$\begin{aligned} &swim \leftarrow sweating,\ pool.\\ &pool \leftarrow swim.\\ &sweating \leftarrow hot.\\ &hot. \end{aligned} \tag{2.2}$$

Here, $hot.$ (which is shorthand for $hot \leftarrow .$) represents a rule with no preconditions[1], meaning it can always be derived and thus must be unconditionally true. After deriving

[1]We call rules without a body *facts*.

hot, we can also assume *sweating* from the third rule, since all of its preconditions are fulfilled. Looking at the first two rules, one can see that *swim* may only be established if we can assume *pool* (and *sweating*, which we already successfully derived), but, conversely, we can only assume *swim* if we can derive *pool*. This circular dependency cannot be broken as long as we do not add further rules allowing for the derivation of *pool* or *swim* (or both) [BET11].

Since an answer set per definition encompasses all literals that can be consistently derived without creating inconsistencies, the answer set of Listing 2.2 is $\{hot,\ sweating\}$. While each answer set of a program is also a model, not every model is an answer set, since models do not necessarily satisfy the requirement of being *founded*, meaning that all atoms are derivable from the program like in the example above.

2.1.3 Answer Set Solvers and Grounders

Answer set solvers are sophisticated software systems used for calculating valid answer sets of a given program [Lif19]. In practice, programs usually contain schematic variables. Many answer set solvers, like the one used by the framework developed in this thesis, require programs to be *grounded* before determining the answer sets of the program.

Given a program P, the ground version of P, denoted by $grnd(P)$, is calculated using the atoms from the *Herbrand base*, where the Herbrand base of P consists of all ground (variable free) atoms constructible from the predicate symbols and constants appearing in P [BKI14].

To illustrate the grounding process, consider the following program:

$$\begin{aligned} &edge(a, b). \\ &path(X, Y) \leftarrow edge(X, Y). \end{aligned} \tag{2.3}$$

This program contains two variables, namely X and Y, which, with the use of a grounder, are replaced by the instances a and b, yielding $path(a, b) \leftarrow edge(a, b)$, $path(a, a) \leftarrow edge(a, a)$, $path(b, a) \leftarrow edge(b, a)$ and $path(b, b) \leftarrow edge(b, b)$ on the second line.

After grounding is completed, answer set solvers, which share many similarities with satisfiability (SAT) solvers [Lif19, BET11], are used to perform efficient search for all answer sets of the input program. In the case of the grounded version of Listing 2.3, only $path(a, b) \leftarrow edge(a, b)$ can be satisfied, thus the answer set is equal to $\{path(a, b),\ edge(a, b)\}$.

2.2 Reinforcement Learning

Reinforcement learning is a subfield of machine learning that uses a numerical reward signal to learn what action is best to be performed in a given situation [SB18]. The following sections give an overview on what reinforcement learning is, and which specific methods are used in the framework developed in this thesis.

2.2.1 Characteristics of Reinforcement Learning

Reinforcement learning is based on a repeated loop of an agent's interaction with the environment it is situated in. At each step t, the learner and decision maker, which we also call the *agent*, executes an action $A_t \in \mathcal{A}(S_t)$, and receives a new state S_{t+1} together with a reward signal R_{t+1} from the environment, shown in Figure 2.1. This loop is usually repeated numerous times (until some terminal or absorbing state is reached), and all the steps together form what is termed an *episode*. These so-called *experiences* of the agent's trial-and-error based interactions with the environment provide the stream of data used to power the learning process.

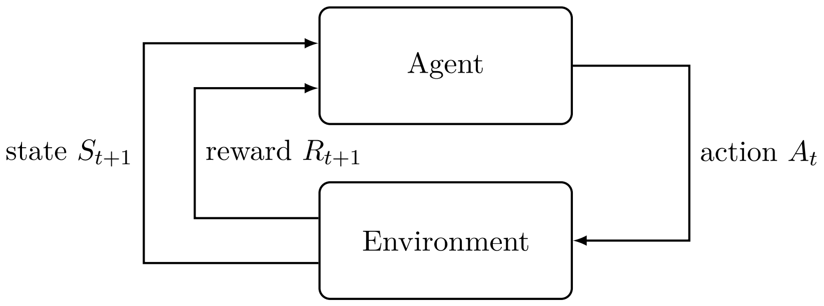


Figure 2.1: The agent-environment interaction loop in reinforcement learning.

In reinforcement learning, the rewards emitted by the environment play a fundamentally important role, as they represent the only metric used to improve the agent's behavior. A reward R_t is a scalar feedback signal (like -1 or $+10$) and indicates how well the agent is doing at step t. Using this reward signal, the agent tries to choose actions maximizing its *cumulative reward*, that is, the total future reward during an episode. Trying to maximize not the immediate but future reward, the agent might sacrifice a high immediate reward for even higher rewards in the long run. It is worth noting that the agent might not get immediate feedback on its actions, but rather receives delayed feedback after a number of steps.

In order to categorize different reinforcement learning approaches, we shall look at three important components an agent might use. First, a reinforcement learning agent may use a so-called *policy* to represent its knowledge about what actions to take in certain states. Those policies can be deterministic functions mapping states to actions, but they can also be stochastic. Second, an agent can use a *value function* to predict future rewards of a state and/or action, and is used to evaluate the goodness or badness of a state. Third, the agent may leverage a *model* representing the agent's view of the environment and predict how it is going to change after choosing a certain action. Since the environment is initially unknown in a reinforcement learning problem, a model may help the agent to understand its surroundings better.

A problem often faced in reinforcement learning is the trade-off between *exploration* and *exploitation* [SB18]. To learn more about the environment and find possible ways of achieving a higher reward, the agent needs to take actions it did not yet perform in order to discover uncharted parts of the environment. However, the agent has to eventually stop exploring new moves and try to exploit all the information it has learned about the environment up until now in order to maximize its total reward.

To give an example, in online advertising an agent may either decide to show the most successful advert to a user, exploiting the knowledge it has gained so far, or it can choose to perform an explorative move and display a new advert, which might be even more successful.

This dilemma of balancing exploration and exploitation is special to reinforcement learning, setting it apart from the other two major machine learning paradigms, namely supervised and unsupervised learning.

2.2.2 Markov Decision Processes

Markov decision processes (MDPs) [SW11, SB18, Sze10] are a way of describing and formalizing sequential decision making and represent the underlying framework used by reinforcement learning. Standard MDPs use actions, states and rewards in a fully observable[2], stochastic environment to formally describe the process of reinforcement learning, allowing us to find good or even optimal policies for selecting actions.

In an MDP, we define an agent's goal by assigning a high *value* to states we consider good, and the agent then learns to choose actions to maximize its cumulative reward in the long run, called the *return*. The return is denoted by G_t, which is defined by

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \;=\; \sum_{k=0}^{T} \gamma^k R_{t+k+1},$$

where T is some terminal state in an episodic task. The discount factor $\gamma \in [0, 1]$ expresses that the value of receiving return R after $k + 1$ time steps is $\gamma^k R$. This has the effect that the agent values immediate reward above delayed reward, and also eliminates problems with infinite rewards in continuing tasks where $T = \infty$, if we set $\gamma < 1$. Since in this thesis, however, the task the agent tries to learn naturally breaks down into finite episodes, we are not forced to use discounting, which is why an *undiscounted* formulation of MPDs (where $\gamma = 1$) is used.

Putting it all together, an MDP is formally described by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ and $\mathcal{A}$ are finite sets of states and actions, respectively, $\mathcal{P}$ describes the state transition probability matrix, $\mathcal{R}$ is a reward function and γ is a discount factor between 0 and 1 (both inclusive) [SB18].

Additionally, the MDP requires that all states in $\mathcal{S}$ have the *Markov* property. A state S_t is *Markov* if and only if

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, \ldots, S_t].$$

[2]Although MDPs can also be used in partially observable environments, then called *POMDPs*, we use the standard interpretation, because the problem used in this thesis is fully observable.

Intuitively, this means that the state S_t captures all relevant information for determining the future, or informally, that "the future is independent of the past given the present".

From the MPDs, we try to estimate the *value function*, or often more useful, the *action-value function*, a real-valued function that yields the long-term value of a state-action pair. Formally, an action-value function is given by

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a],$$

where $\mathbb{E}_\pi$ is the expected return starting in state S_t, taking action A_t and thereafter following policy π. It is easy to see that the action-value function decomposes into two parts: the immediate reward of the current state (left side) plus the discounted value of the subsequent state S_{t+1} and action A_{t+1} (right side).

Action-value functions allow us to describe the optimization problem used to determine optimal policies and behavior for the agent. For small instances, the action-value function can be represented *tabularly* as an array or table. However, in many applications, the state-action space is too large to be stored in any table, thus only approximations of action-value functions (with less parameters than state-action pairs) can be calculated, which in turn can only approximate optimal behavior.

In order to achieve the best performance in an MDP, we are interested in the *optimal* action-value function q_*, which is the maximum action-value function over all policies, or

$$q_*(s,a) \doteq \max_\pi \; q_\pi(s,a).$$

In the end, we want our agent to learn an optimal policy which fully defines its behavior, so we need to define a partial ordering over policies. A policy π' is better than or equal to a policy π if the value of *each* state-action pair (s,a) under policy π' is greater than or equal to the value of the corresponding state-action pair (s,a) under policy π.

We find this optimal policy by maximizing over $q_*(s,a)$, and for any MDP, we can find such an optimal policy, which is also deterministic as well as stationary (independent of t). To calculate q_*, one can use an iterative form of the *Bellman optimality equation.*

2.2.3 Solving Markov Decision Processes

Having defined Markov decision processes (MDPs) in the last section, we now consider possible methods on how to practically solve MDPs computationally. When we say *solve* an MDP, we mean to find a way how to best perform in that MDP, such that we "get the most reward out of it".

In general, we distinguish between approaches to solve reinforcement learning problems based on whether they use a model, then called *model-based*, or not. Model-based approaches rely on knowing the precise transition probabilities and rewards of the environment, and since MDPs satisfy the principle of optimality and sub-solutions can be reused and stored, dynamic programming algorithms can be applied. For dividing an MDP into sub-problems, dynamic programming algorithms rely on the Bellman equation, allowing for an efficient recursive decomposition of the full MDP. With dynamic programming, we can compute the maximum value function q_*, and derive an optimal policy from it. Although dynamic programming requires a perfect representation of the environment in form of an MDP and comes at great computational cost, dynamic programming algorithms are well understood and provide a good foundation for other algorithms which do not rely on the use of a model.

The biggest drawback of dynamic programming approaches is the required perfect knowledge of the environment. In most real world applications of reinforcement learning, the exact characteristics of the environment are unknown, and it is often impossible to determine them. This motivates the development of *model-free* algorithms to solve MDPs, where the MDP of the environment does not need to be known beforehand. Model-free algorithms learn directly from experience with the environment, sampling many different (random) episodes. Interestingly, even though the MDP is unknown and only gets sampled multiple times, we can still calculate optimal policies and value functions with model-free methods, as the number of episodes the agent "sees" approaches infinity.

Monte Carlo Methods

A simple way of implementing model-free algorithms are Monte Carlo methods. Monte Carlo methods learn from *complete* episodes (they do not bootstrap), and use the idea that the value of each state or state-action pair is equal to the experienced average return starting from that state. Note that Monte Carlo methods only work in episodic MDPs, where all episodes terminate.

In order to improve a policy, we make use of both *prediction* and *control*, where the former refers to evaluating the value of a given policy, and the latter to finding the best policy, usually done by performing incremental improvements. When evaluating a policy π, we want to calculate the effectiveness of π, which can be achieved by determining its value function v_π, to see how much return we "get out of the MDP" when following π.

For a concrete algorithm for evaluating a policy, we shall look at the first-visit Monte Carlo policy evaluation method [SB18]. In this algorithm, we maintain a counter for each state s (or state-action pair (s, a)), which we increase each time a state is first visited in an episode.[3] Additionally, we keep track of all returns achieved from that state s (or state-action pair (s, a)) onwards by incrementally adding the current return each time that state is visited. From that, a simple division of the total number of achieved returns from state s and the counter representing the number of first visits of s yields the estimated value $V(s)$ of s. To improve performance, we calculate the estimated value

[3] *Every-visit* Monte Carlo methods work just like first-visit methods, with the only difference of counting *every* experience of a state in an episode, and not just the first.

$V(s)$ incrementally with

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t)),$$

where $N(S_t)$ is the number of first visits to S_t, and the return G_t represents the *unbiased* estimate of $v_\pi(S_t)$. Like this, we continuously adjust the arithmetic mean into the direction of the error $(G_t - V(S_t))$ of our expectation. By the law of large numbers, as the number of visits $N(S_t)$ approaches infinity, the expected return of state S_t converges to its true value $v_\pi(S_t)$.

Improving a policy on the other hand requires many iterations of evaluation and incremental improvements. Policy iteration starts from an arbitrary initial policy π and then improves this policy (in all states visited during that episode) by *greedily* choosing the action that yields the greatest future return. Such an improvement always guarantees that the new greedy policy π' is always better than or equal to the policy before the improvement ($v_{\pi'} \geq v_\pi$) [SB18].

An algorithm performing policy iteration would have to maintain both an approximate action-value[4] function as well as an approximate policy. As already stated, we create our greedy policy by simply choosing an action with the maximum action-value (with ties broken arbitrarily). The new, optimized policy π' after one improvement is given by

$$\pi'(s) \doteq arg \max_a q(s, a).$$

After this improvement, we evaluate the new policy π', update the value function accordingly, and repeat the process. In case the new policy π' is equal to π, they are both optimal policies and improvement stops.

In order for policy iteration to converge to the optimal policy π_*, we also need to maintain exploration, and theoretically have to experience each state-action pair an infinite number of times. This important explorative factor is accomplished by a version of Monte Carlo search called *Exploring Starts*, where each individual episode starts in a random state $s \in \mathcal{S}$, performs one random action $a \in \mathcal{A}(s)$ and thereafter follows policy π.

It was proven that the Monte Carlo Exploring Starts algorithm converges to the optimal policy π_* for deterministic and even for the more general Optimal Policy Feed-Forward environments [WR20].

Other Methods

In addition to Monte Carlo methods discussed in the previous section, there exist other model-free algorithms to solve MDPs.

[4]Note that if we were to work on *state-value* functions, a model of the environment would be needed to determine the value of an action.

A widely known alternative to Monte Carlo methods is called *temporal difference* learning [SB18]. Temporal difference learning, as opposed to Monte Carlo methods, works on *unfinished* episodes and exploits the Markov property for approximating the most likely underlying Markov model of the environment. Furthermore, temporal difference learning works even if the end of an episode is never reached, since the learning process is occurring step-by-step, or *online*. In most cases, temporal difference learning has lower variance but is more biased than Monte Carlo methods, since a biased *estimate* of the value function is used, which allows for *bootstrapping*.

In Markovian environments, temporal difference learning often converges faster, while Monte Carlo methods are less susceptible to the initial values and have better convergence properties.

For the framework developed in this thesis, an on-policy, first-visit Monte Carlo Exploring Starts algorithm is used, discussed in greater detail in Section 3.1.1.

2.3 The Blocks World

Arguably one of the most famous and widely used domains for planning problems in artificial intelligence is the blocks world. The blocks world is a so-called toy problem, because it does not directly yield any real life applications. However, given the characteristics of the problem, which will be discussed later in this section, the blocks world is a suitable environment for benchmarking planning algorithms. This is the reason we chose it as a realm for testing our approach of combining reinforcement learning with answer set planning.

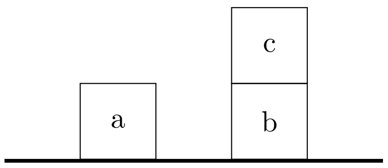
2.3.1 The Environment

The fundamental components of the blocks world environment are a fixed number of unique blocks and a table. The blocks can be stacked on top of one another, forming towers, while only one block can be directly on top of another. An exception to this rule is the table, on which infinitely many blocks can be placed, ensuring that it never runs out of free space [Lif19, ST01].

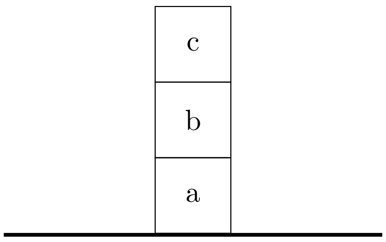
Formally, each individual tower in a blocks world containing a total of n blocks consists of $1 \leq b \leq n$ blocks, while the sum of the blocks of all towers is equal to n. Note that the order in which the towers are placed on the table does not matter.

The initial state in a blocks world is usually presented by a random set of towers placed on the table. An agent's goal, given its ability to move one block at a time is to find a finite set of actions eventually leading to a specified goal state, that is, a specific arrangement of blocks on the table. Figure 2.2 shows one initial start state (2.2a) for $n = 3$ blocks with a desired goal state (2.2b).

Several other, slightly modified versions of the blocks world domain have been proposed. In one version, for example, the agent has a certain probability of failing to pick up a



(a) An initial start state.



(b) The desired goal state.

Figure 2.2: The start and goal states of a blocks world.

block, which causes the block to fall on the table. This introduction of probability distributions over truth values of fluents turns the blocks world into a stochastic, nondeterministic environment [Nic12a]. Other configurations allow picking up multiple blocks simultaneously [Lif19], multiple occurrences of the same block [Che91], or feature different shapes and sizes of blocks [Fah74], to mention a few examples.

Within the scope of this thesis, however, we only consider the classic interpretation of the blocks world, since its structure is already sufficiently complex for our purposes.

2.3.2 The Blocks World as a Dynamic System

The blocks world is usually modelled as a *dynamic system* [Lif19, BKI14], that is a system, whose states can be changed by performing actions. Using answer set programming, we will make use of the blocks world as a dynamic system for both looking at prediction problems as well as planning problems, where the answer sets can be interpreted as the solutions to the problem encoded in a given answer set program.

In a prediction problem, the goal is to determine how the environment will change after executing certain actions, while a planning problem is about finding a finite sequence of actions that will transform the current state into (one of) the desired goal state(s). The agent is situated in an environment which is finite, fully observable and deterministic, and for simplicity, we view the movement of a block from one location to another as an atomic action, disregarding possible divisions into sub-actions like the mere act of picking up a block.

Since dynamic systems are subject to change over time, we need a way to represent time in our answer set program to account for the changing truth values of certain predicates. This motivates the introduction of so-called *fluents*, which, in addition to normal predicates, also contain a nonnegative integer T denoting a discrete time step starting at 0 until some defined time horizon h.

At each time step, the agent has the ability to pick up one block a and stack it onto another block b (or the table), provided that there is currently no other block c on top of a or b. This restriction serves as a precondition for the action $move(a, b)$, which moves block a on top of block b. The *qualification problem* deals with those preconditions of actions, which are conditions that must hold right before executing an action. If one or more preconditions are unsatisfied, this specific action may not be performed at that time. The postcondition of actions like $move(a, b)$ is that block a is now on top of block b, which is often expressed as a fluent of the form $holds((on(a, b), t)$, where t represents a specific time at which the statement $on(a, b)$ is true.

Another problem that is often present when dealing with dynamic systems in answer set programming is the so-called *frame problem*. The frame problem describes the difficulty of transferring valid predicates of the knowledge base onto the next state. For example, in case of the block world, each time after the agent executes an action, many facts about the environment which were true before the execution are still true after the transition into the new state.

While it may seem obvious that each block does not change its location if it was not moved, this needs to be specified explicitly in an answer set program to prevent unwanted changes of facts, and this is usually done by specifying *frame axioms*. In case of the blocks world, we can express this frame axiom by adding a rule of the form

$$holds(on(X, Y), T + 1) \leftarrow holds(on(X, Y), T),\ not\ \neg holds(on(X, Y), T + 1)$$

to the answer set program.

2.3.3 Number of States and NP-Completeness

To develop a more profound understanding of the growing complexity of the blocks world domain with an increasing number of blocks, we shall look at the computation of its possible states as well as the complexity theoretic aspects of the problem.

Number of States

Since the number of states in the blocks world is known to grow exponentially [ST01], the blocks world is a particularly suitable environment for testing a multitude of planning and search algorithms against problems of varying difficulty.

If, for example, there are 2 blocks, there are 3 distinct states in which the blocks world can be. Either both blocks are placed on the table, or they are stacked on top of one another, where the ordering matters, resulting in 3 legal states. Increasing the number of blocks to 3, there are 13 legal states, while 4 blocks already yield 73 possible combinations.

Looking for a way to efficiently count those states for a larger number of blocks, there are both iteratively and recursively defined formulas [ST01] for calculating the number of possible states in the blocks world as a function of the number of blocks. Looking at a slightly more general problem, the terms of *grounded* and *ungrounded* towers are introduced, where a tower is considered grounded if and only if it ends on the table, otherwise it is considered an ungrounded tower. The formulas use n to refer to the number of ungrounded towers, while k represents the number of grounded towers, and the function $g(n, k)$ counts all possible states of the blocks world having n grounded and k ungrounded towers.

The recursive definition of the formula starts by defining a base case, which in our case is $g(0, k) = 1$. Intuitively, this means if all towers already end with the table, there is precisely one possible way to arrange them, since we declared the ordering of the towers on the table as irrelevant. In the second part of the definition, we consider $g(n + 1, k)$. Having one more ungrounded tower which we want to place somewhere, it can either go on top of the table, or on one of the remaining $(n + k)$ towers. Placing it on the table produces one additional grounded tower, leaving $g(n, k + 1)$ possible states. On the other hand, if we place the new tower on another already existing tower, there are $(n + k)$ towers to choose from, each leaving $g(n, k)$ possible extensions.

Putting those two parts together, the definition of the complete formula is

$$g(0, k) = 1,$$

$$g(n + 1, k) = g(n, k + 1) + (n + k)g(n, k).$$

For calculating just the number of states in a blocks world containing n blocks, a special case of the formula is used where $k = 0$, or

$$\mathit{states}(n) = g(n, 0).$$

It is important to note that those numbers rapidly grow for bigger n. For example, $\mathit{states}(40) = 2681748411130962366250327593765860804139695383855921$. Table 2.1 shows the number of states up until $n = 10$.

n	number of states
1	1
2	3
3	13
4	73
5	501
6	4,051
7	37,633
8	394,353
9	4,596,553
10	58,941,091

Table 2.1: Growing number of states for n blocks.

The iterative definition of the formula [ST01], shown in 2.4, is not as intuitive as the recursive one, but allows for faster computation of the possible states.

$$g(n,k) = \sum_{i=0}^{n} \binom{n}{i} \frac{(n+k-1)!}{(i+k-1)!}. \tag{2.4}$$

It is worth noting that 2.4 seems to break down for $k = 0$, as it yields $(-1)!$, which is commonly known to be undefined. However, the formula works for $k > 0$. It turns out that only a small adaptation needs to be performed to fix this formula for $k = 0$ and $n > 0$, depicted in 2.5.

$$\mathit{states}(n) = \sum_{i=1}^{n} \binom{n}{i} \frac{(n-1)!}{(i-1)!}. \tag{2.5}$$

In addition to looking at different states in which the blocks world can be, we also have to consider how many possible *actions* an agent can take at any given time in its environment.

The number of possible actions, if all blocks are on the table, are $n(n-1)$, since each of the n blocks can be moved on top of one of the remaining $(n-1)$ ones. If, instead, we look at n *towers*, there are n^2 possibilities, since each topmost block, in addition to the other towers, can also be moved onto the table. Finally, if the current state contains n towers, where m of them contain more than one block, the agent has $n(n-1)+m$ different actions to choose from [Lif19].

NP-Completeness

In order to discuss complexity theoretic aspects of the blocks world, we first need to introduce the idea of *deadlocks*, since they have the biggest influence on the hardness of a blocks world planning problem.[5] A deadlock in a blocks world instance with goal state G is defined [GN92] as follows:

> The set of blocks $b_1, b_2, \ldots, b_p$ is *deadlocked* in the state S if there is a set of blocks $d_1, d_2, \ldots, d_p$ such that the following three conditions hold:
>
> 1. In S, b_i is above d_i for $i = 1, 2, \ldots, p$
> 2. In G, b_i is above d_{i+1} for $i = 1, 2, \ldots, p-1$, and b_p is above d_1
> 3. In S, none of $b_1, b_2, \ldots, b_p$ are in their final positions (if $p > 1$, then the other two conditions entail this condition)

In addition to deadlocks, we also consider so-called deleted-condition interactions, which are interactions where the agent has to sacrifice an already established subgoal for another,

[5]Note that in the following we only consider blocks world problems of size $n > 1$, as problems of size $n = 1$ can only be in one state.

not yet reached subgoal. Lastly, we define an optimal plan as a plan which reaches the specified goal state within the minimum number of steps, or in other words, such that there exists no shorter plan.

As it turns out, the *only* aspect that makes planning in blocks world *hard* are deadlocks, since the deleted-condition interactions can be easily solved by a planner. For example, given a deadlock free problem, it is proven that a hill-climbing strategy suffices in order to find an optimal plan in time $O(n^3)$ [GN92].

On the contrary, if presented with blocks world instances which do contain deadlocks, optimal planning is NP-complete. In order to prove the general NP-completeness of the blocks world problem, we shall look at an optimization problem of the following form:

> Given a blocks world problem B, is there a plan containing no more than m moves leading to the goal state?

The NP-completeness is proven by showing how every 3-Satisfiability (3SAT) instance can be transformed into a blocks world problem (in which multiple occurrences of the same block are allowed) in a polynomial number of steps [Che91]. A similar proof converts the blocks world problem into a yes/no decision problem and reduces the NP-complete feedback arc set (FAS) problem to it [GN92].

CHAPTER 3

The Framework

In this chapter, we introduce a framework for efficiently combining reinforcement learning with answer set programming for many problems which can be represented using deterministic Markov decision processes (MDPs).

3.1 The Basic Framework

The framework developed in this thesis aims to efficiently "solve" MDPs using a Monte Carlo reinforcement learning algorithm, where the entire environment is represented by an answer set program.

3.1.1 The Reinforcement Learning Algorithm

As for the reinforcement learning algorithm, an on-policy first-visit Monte Carlo Exploring Starts algorithm similar to what is presented in the book by Sutton and Barto [SB18] is used because of its good convergence properties and ease of implementation.

Concretely, the algorithm developed works on state-action pairs and starts with an empty policy π; an optimal policy π_* is calculated iteratively. We decided to start the learning process with an empty policy and not a random policy, because initializing a random policy beforehand quickly becomes infeasible, as the number of state-action pairs can be too large for many problems.

Since the policy is empty at the beginning, each time a state s is first visited (over all episodes), we determine all actions applicable in that state, denoted by $a_i \in \mathcal{A}(s)$ using an answer set program discussed later in this section. These state-action pairs (s, a_i) are then initialized with a value of zero, and each time a state is first visited in an individual episode, the experienced value of taking action a_i in state s is calculated, where the average of those experiences is stored in the action-value function $\mathcal{Q}$. Each episode starts in a random state. The agent chooses an action applicable in this start state uniformly at random, regardless of what the policy recommends. Then, the agent greedily selects actions, meaning the agent is going to select the action with the highest expected return for a given state. Note that since this algorithm is sensitive to the initial value of state-action pairs, a higher initial value increases the likelihood of the agent picking an unexplored action, resulting in greater exploration.

As the number of episodes approaches infinity, each state-action pair is experienced infinitely many times, and the action-value function $\mathcal{Q}$ converges to the true, optimal action-value function q_*. Since we greedily choose the action with the highest expected return from that function, our policy π also converges to an optimal policy π_*.

The pseudocode of the implementation of the reinforcement learning algorithm can be seen in Algorithm 3.1, where the generation of episodes is shown separately in Algorithm 3.2. In both algorithms, $\mathcal{S}$ stands for the exhaustive list of all possible states of the environment, and T stands for the final time step of an episode.

Algorithm 3.1: First-visit Monte Carlo Exploring Starts for estimating $\pi \approx \pi_*$.

```
Input: Discounting factor $\gamma$, number of episodes to generate
Output: Estimation of $\pi \approx \pi_*$
$\pi \leftarrow$ empty dictionary
$\mathcal{Q}(s, a) \leftarrow$ empty dictionary
$Visits(s, a) \leftarrow$ empty dictionary
$\mathcal{S} \leftarrow$ exhaustive list of all states, determined in ASP
for number of episodes do
    $\mathcal{E} \leftarrow$ Episode($\pi$, random $S_0 \in \mathcal{S}$)
    $G \leftarrow 0$
    for each step of $\mathcal{E}$, $t = T, T-1, \ldots, 0$ do
        $G \leftarrow \gamma G + R_t$
        if the pair $S_t, A_t$ does not appear in a preceding step in $\mathcal{E}$ then
            if $S_t$ has no entry in $\mathcal{Q}$ then
                for each action $A$ applicable in $S_t$, determined in ASP do
                    $Visits(S_t, A) \leftarrow 0$
                    $\mathcal{Q}(S_t, A) \leftarrow 0$
                end
            end
            $Visits(S_t, A_t) \leftarrow Visits(S_t, A_t) + 1$
            $\mathcal{Q}(S_t, A_t) \leftarrow \mathcal{Q}(S_t, A_t) + \frac{G - \mathcal{Q}(S_t, A_t)}{Visits(S_t, A_t)}$
            $\pi(S_t) \leftarrow argmax_a \mathcal{Q}(S_t, a)$
        end
    end
end
```

Algorithm 3.2: Episode generation with outsourced ASP calls.

```
Input: A policy $\pi$ and a random start state $S_0$
Output: Episode $\mathcal{E}$
function Episode($\pi$, $S_0$) do
    $\mathcal{E} \leftarrow$ empty list
    $\mathcal{A}(S_0) \leftarrow$ calculate in ASP with input $S_0$, $horizon = 0$
    $count \leftarrow 0$
    while $count \leq$ maximum number of steps do
        if $S_t$ has entry in $\pi$ and count is not 0 then
            $A_t \leftarrow \pi(S_t)$
        else
            $A_t \leftarrow$ random $A_t \in \mathcal{A}(S_t)$
        end
        if $A_t$ is empty then
            break
        end
        $S_{t+1}$, $R_{t+1}$, $\mathcal{A}(S_{t+1}) \leftarrow$ calculate in ASP with input $A_t$, $S_t$ and
         $horizon = 1$
        Append $(S_t, R_{t+1}, A_t)$ to $\mathcal{E}$
        $S_t \leftarrow S_{t+1}$
        $\mathcal{A}(S_t) \leftarrow \mathcal{A}(S_{t+1})$
        $count \leftarrow count + 1$
    end
    return $\mathcal{E}$
end
```

3.1.2 Representing the Environment With Answer Set Programs

Having discussed the reinforcement learning algorithm to power the learning process, we shall now look at how the environment is represented as a dynamic system using an answer set program as well as how applicable actions, state transitions, rewards and initial start states are determined.

Since the blocks world problem, as introduced in Section 2.3, represents a commonly used toy problem for benchmarking learning algorithms, we decided to implement the blocks world problem as a dynamic system and use it as the testbed for our framework. Concretely, we implemented an answer set program describing the underlying MDP used to model the environment of the blocks world.

On a high level, the agent chooses one of the applicable actions given by the answer set program, and then receives a new state as well as a numerical reward from the environment in return. This process happens multiple times in an episode, and many episodes are generated to provide the stream of data used by the reinforcement learning algorithm.

Computation of Random Start States

In order to determine the start state of an episode, an answer set program first calculates all possible states of the environment. Since our blocks world environment is fully observable, known, and finite, we can naively enumerate all possible states, as Listing 3.1 illustrates.

```
% input atoms %%%%%%%%%%
block(a; b; c; d).

% define state and locations
state(on(X,Y)) :- on(X,Y).
location(table).
location(X) :- block(X).

{ on(X,L) : location(L), L != X } = 1 :- block(X).
:- block(X), { on(Y,X) : block(Y) } > 1.

% every block has to be supported, directly or indirectly, by
   the table
supported(X) :- on(X,table).
supported(X) :- on(X,Y), supported(Y).
:- block(X), not supported(X).
```

Listing 3.1: The answer set encoding of admissible initial states.

As input, we only need to specify a number of different blocks using the `block/1` predicate, so for example, we can declare a blocks world consisting of four blocks by writing `block(a; b; c; d).`[1] As a result, each individual answer set describes one possible state of the blocks world, described by `state/1` atoms. From those, we choose one of the states at random and declare it as the starting point S_0 of our episode.

Note that the syntax of the logic programs presented here corresponds to the syntax used by CLINGO[2], an answer set programming system for grounding and solving logic programs. CLINGO is part of the Potsdam Answer Set Solving Collection (Potassco)[3] developed at the University of Potsdam in Germany.

Since the generation of random start states using enumeration can quickly become infeasible for larger problems (see Sections 2.3.3 and 5.2) as the number of blocks reaches 10 or more, our system switches to an algorithm to generate only *one* pseudo-random start state. However, it was shown [ST01] that this algorithm has some bias towards a

[1]The table is always instantiated automatically.
[2]`https://potassco.org/clingo/`
[3]`https://potassco.org`

certain category of states, and does not generate start states as representative as our enumeration algorithm does.

Another possibility is to not generate *all* answer sets representing the states, but rather to create only one random start state using *parity* or *XOR* constraints directly on the answer set solver level [EJKS19].

Simulating Actions, State Transitions, and Rewards

As already stated, the framework also uses an answer set program to describe the underlying Markov decision processes of the environment. On episode generation, as shown in Algorithm 3.1 at Line 6, we start our new episode in state $S_0 \in \mathcal{S}$ by randomly choosing one of the answer sets produced by Listing 3.1.

After that, the answer set program shown in Listing 3.2 is called multiple times in order to compute state transitions and determine applicable actions as well as the rewards emitted by the environment. This answer set program takes several predicates as input.

First, we have to define the current state of the environment by declaring a number of `current/1` atoms. These atoms are interpreted by the program as the current state of the world, so for the blocks world example, we could provide the program with the atoms `current(on(a, table))` and `current(on(b, table))` to declare a state where `a` and `b` are currently on the table.

Second, the program takes at most one `action/1` atom. Given this atom, the program applies the action specified to the current state, and in turn generates a new state S_{t+1}, represented by a number of `state/1` atoms. For example, given the current state from the previous paragraph, say we provide the program with an action like `action(move(b, a))`. Then, the program would return state predicates representing the new state after executing the provided action, namely `state(on(a, table))` and `state(on(b, a))`.

Third, the program needs a number of `subgoal/2` atoms describing the goal state. For instance, we can add the atoms `subgoal(a, table)` and `subgoal(b, a)`, if we want to define our desired goal state as having block `a` on the table, and block `b` on top of `a`.

Fourth, we use an integer constant called `horizon` to specify the state of the current episode.[4] Concretely, if we call the program with `horizon` equal to zero, we indicate that we just started an episode and want to determine all possible actions (`executable/1`) given our start state S_0. On the other hand, we can set `horizon` to one, and provide a state S_t as well as an action A_t. Then, the resulting new state S_{t+1}, as well as all new applicable actions and the received reward, indicated by the atom `nextReward/1`, are computed. Calls like these are used to compute state transitions as soon as the agent has exited the initial state of an episode.

[4]Note that this constant is also used as the planning horizon, as described in the next section.

The program in Listing 3.2 describes a blocks world with four blocks and a goal state requiring all blocks to be ordered alphabetically.

```
% input atoms %%%%%%%%%
% current/1 atoms describing the current state
% action/1 the action that has been chosen, optional
% #const horizon defining the episode state/planning horizon
% subgoal/2 describing the goal state
subgoal(a, table). subgoal(b, a). subgoal(c, b). subgoal(d, c).

reward(100,T) :- goal(T), not goal(T-1). % goal reward
reward(-1,T) :- move(_,_,T-1). % penalty for each move

block(X) :- on(X,_,0).
location(table).
location(X) :- block(X).
time(0..horizon).
occupied(X,T) :- block(X), on(_,X,T).
free(X,T) :- location(X), time(T), not occupied(X,T).
on(X,Y,T+1) :- on(X,Y,T), not -on(X,Y,T+1), time(T).

executable(move(X,Y)) :- executable(X,Y,1), horizon > 0.
executable(move(X,Y)) :- executable(X,Y,0), horizon = 0.
on(X,Y,0) :- current(on(X,Y)).
move(X,Y,0) :- action(move(X,Y)).
state(on(X,Y)) :- on(X,Y,1).
nextReward(N) :- totalReward(N,1).
bestAction(move(X,Y)) :- move(X,Y,0).
goal(T) :- time(T), { not on(X,Y,T) : subgoal(X,Y) } = 0.

executable(X,Y,T) :- block(X), free(X,T), free(Y,T), not on(X,Y
   ,T), X != Y, not goal(T).
{ move(X,Y,T) : executable(X,Y,T) } = 1 :- time(T), T < horizon
   , not goal(T).
-on(X,Y,T+1) :- move(X,_,T), on(X,Y,T), time(T).
on(X,Y,T+1) :- move(X,Y,T), time(T).

totalReward(S,T) :- time(T), S = #sum { R : reward(R,T) }.
```

Listing 3.2: The answer set encoding of the blocks world MDP.

Once the program does not return any applicable actions anymore, that is when the state and action given to the program already yield a goal state, episodes get terminated. Additionally, episodes are terminated if they do not reach a goal state within a maximum number of steps.

Each time an action is performed, this answer set program gives a negative reward of -1, and if all subgoals are satisfied, a positive reward of $+100$ is established. Note that Listing 3.2 shows the basic structure of the answer set encoding only, with the remaining input predicates being added dynamically at runtime, if needed.

3.2 The Planning Component

Up until now, we only examined the basic framework, thereby omitting the planning component which we aspire to combine with the Monte Carlo reinforcement learning algorithm. More importantly, the algorithms introduced in the previous section are all *model-free* reinforcement learning methods. Model-free methods are lacking the ability to leverage a *model* of the environment to predict possible future situations without having to actually experience them.

We want to give our algorithm the ability to (occasionally) use a model for planning. While there are reinforcement learning systems that use the typical trial-and-error approach but also *learn* a model of the environment and then use that learned model for planning [SB18], the algorithms presented here need to know the model beforehand.

In order to give the agent the capability to plan, an extended version of the answer set program introduced in the previous section is used. Since the rewards are already part of the answer set program in Listing 3.2, a simple extension, shown in Listing 3.3, suffices to generate optimal plans.

```
% maximize the reward function
maxReward(S) :- S = #sum { R,T : reward(R,T) }.
#maximize { S : maxReward(S) }.
```

Listing 3.3: The extension for the answer set encoding to enable planning.

In this extension, we determine the best plan by maximizing over the sum of the experienced rewards.[5] After that, we choose the action yielding the highest reward within the specified planning horizon denoted by the atom `bestAction`. In other words, we always get an optimal action for the agent to take if a goal state is reachable within the number of steps assigned to `#const horizon`.

To incorporate the adapted answer set encoding into the learning process, we need to change the way episodes are generated. During episode generation, after specifying how and when to plan, separate calls to the answer set program are made to make use of the planning component. The pseudocode for an updated version of the episode generation is shown in Algorithm 3.3.

[5]Note that the planning component does not take a discounting factor into account (i.e., $\gamma = 1$).

The algorithm has three different parameters to specify which planning strategy the agent should use.

First, one can specify a number $i \in [0, 1]$ which is attributed to the `planningFactor` parameter. This value directly represents the probability of the agent choosing to use the planning component to recommend an action. If the value of the recommended action has a higher value than the action the current policy would suggest, the action provided by the planning component is selected. Otherwise, the agent acts according to the policy. For example, if we were to set the `planningFactor` to 0.1, each time an action has to be chosen, the agent is going to follow the current policy π with a probability of 0.9, and with a probability 0.1 chooses the better option between the action suggested by the planning component and the current policy π. If the policy is empty for the current state, and the agent does not plan, a random action is chosen.

The second option to invoke the planning component is to set the Boolean parameter `planOnEmptyPolicy`. This parameter, if enabled, makes the agent choose to plan instead of executing a random action, as it would normally be the case if the policy was empty for the current state. Naturally, the frequency of how often the agent has to plan with this strategy deceases over time, provided the environment is finite, as the agent explores the world and more and more states are "seen".

In order to control the quality of the actions the program suggests, the `planningHorizon` parameter is of particular importance to the system, as it defines how many consecutive steps a plan should consider. Given that plan, the program then gives the best action the agent can perform next, represented by `bestAction`. The larger the planning horizon, the higher the quality of the plans generated, as the plans become more farsighted.

For example, if we want to plan in a way such that the program always returns optimal actions in a blocks world consisting of n blocks, it suffices to set `planningHorizon` in our algorithm to $2(n-1)$, as an optimal plan never takes more steps than that.

It is important to note that the `planningHorizon` parameter directly corresponds to the constant `horizon` in the answer set program presented in Listing 3.2, so `horizon` is now not only used to specify the episode state, but also the planning horizon.

To sum up, if we use the extended versions of the algorithm and answer set program presented here, we are able to control the agent's planning behavior as well as the length of plans, and therefore the quality of the suggested actions. However, using the updated system, we are still able to use a version of the agent which never uses planning by setting `planningFactor` to zero and `planOnEmptyPolicy` to be false. Like this, we can balance the strictly trial-and-error based Monte Carlo learning with deliberate invocations of a planning component.

Concrete Implementation

An implementation of the described framework was created using PYTHON[6] and Potassco's CLINGO in a CONDA[7] environment.

The implementation uses the powerful CLINGO API[8] provided by Potassco in `ClingoBridge.py`, thereby forming the link between answer set programs and PYTHON classes. The class `BlocksWorld.py` then parses and handles the inputs and outputs of CLINGO, while `MonteCarlo.py` provides the implementation of the Monte Carlo reinforcement learning algorithm as well as an algorithm for episode generation.

To access the full source code, the reader is referred to the GITHUB repository available at `https://github.com/fxgst/RLASP`.

[6] https://www.python.org

[7] https://docs.conda.io/en/latest/

[8] https://potassco.org/clingo/python-api/5.4

Algorithm 3.3: Updated episode generation with planning.

```
Input: A policy $\pi$, random start state $S_0$, planningFactor $\in [0,1]$,
       planOnEmptyPolicy $\in \{true, false\}$, planningHorizon $\in \mathbb{N}$
Output: Episode $\mathcal{E}$
1  function EpisodeWithPlanning($\pi$, $S_0$, $pf$, poep, ph) do
2      $\mathcal{E} \leftarrow$ empty list
3      $\mathcal{A}(S_0) \leftarrow$ calculate in ASP with input $S_0$, $horizon = 0$
4      $count \leftarrow 0$
5      while $count \leq$ maximum number of steps do
6          if $planningFactor \leq$ random $i \in [0,1]$ then
7              if $S_t$ has entry in $\pi$ then
8                  if count is 0 then
9                      $A_t \leftarrow$ random $A_t \in \mathcal{A}(S_t)$
10                 else
11                     $A_t \leftarrow \pi(S_t)$
12                 end
13             else if planOnEmptyPolicy then
14                 $A_t \leftarrow$ get bestAction from ASP with input $S_t$,
                    $horizon = planningHorizon$
15             else
16                 $A_t \leftarrow$ random $A_t \in \mathcal{A}(S_t)$
17             end
18         else
19             $a_p \leftarrow$ get bestAction from ASP with input $S_t$,
                $horizon = planningHorizon$
20             $A_t \leftarrow argmax_a\{Q(S_t, a_p),\ Q(S_t, \pi(S_t))\}$
21         end
22         if $A_t$ is empty then
23             break
24         end
25         $S_{t+1}$, $R_{t+1}$, $\mathcal{A}(S_{t+1}) \leftarrow$ calculate in ASP with input $A_t$, $S_t$ and
            $horizon = 1$
26         Append $(S_t, R_{t+1}, A_t)$ to $\mathcal{E}$
27         $S_t \leftarrow S_{t+1}$
28         $\mathcal{A}(S_t) \leftarrow \mathcal{A}(S_{t+1})$
29         $count \leftarrow count + 1$
30     end
31     return $\mathcal{E}$
32 end
```

CHAPTER 4

Experiments

To test and benchmark the framework described in the previous chapter, several experiments were conducted. A detailed discussion of the findings presented in this section will be given in Chapter 5.

In order to evaluate our experiments and the quality of the policy π learned by the agent, we shall consider the *return ratio* of consecutive episodes during the learning process. We define the return ratio as the ratio of the maximum return (r_{max}) the agent can achieve during an episode following an optimal policy, and the *actual* return (r_{actual}) received during that episode. The maximum return an episode can yield is calculated using our answer set program, where we simply calculate the return of an optimal plan.

Since we want to avoid negative return ratios, the absolute value of the minimum reward (r_{min}) of an episode is added to both the numerator and the denominator. The minimum return is indirectly given by the bound we set that terminates long episodes that are stuck in an infinite loop.

Putting it all together, the return ratio ρ is defined as

$$\rho = \frac{r_{actual} + |r_{min}|}{r_{max} + |r_{min}|}.$$

What we ultimately want to see is the return ratio converging to 1.0, since that would imply that the agent is always achieving the maximum return and therefore must behave according to an optimal policy.

4.1 The Trial-and-Error Based Agent

For the first experiment, let us consider a purely trial-and-error based agent not using the planning component at any point during an episode.

We tested multiple blocks world instances with 4, 5, 6 and 7 blocks, and considered 5,000 episodes each. Since we want to test the agent without the use of any planning, we set `planOnEmptyPolicy` to be false and the `planningFactor` to zero. The maximum episode length is fixed at three times the number of blocks in each respective blocks world to also allow for suboptimal solutions, but terminates episodes not making progress.

We name the blocks in our experiment from `a` through `g` and define the goal state as all blocks being ordered alphabetically in ascending order from the table. The reinforcement learning environment is configured in a way such that the agent receives a negative reward of -1 for each step performed, and a positive reward of $+100$ if every block is in its goal position.

The results are presented in Figure 4.1, where we averaged over 20 runs with the same settings, and plotted 150 points per color.[1]

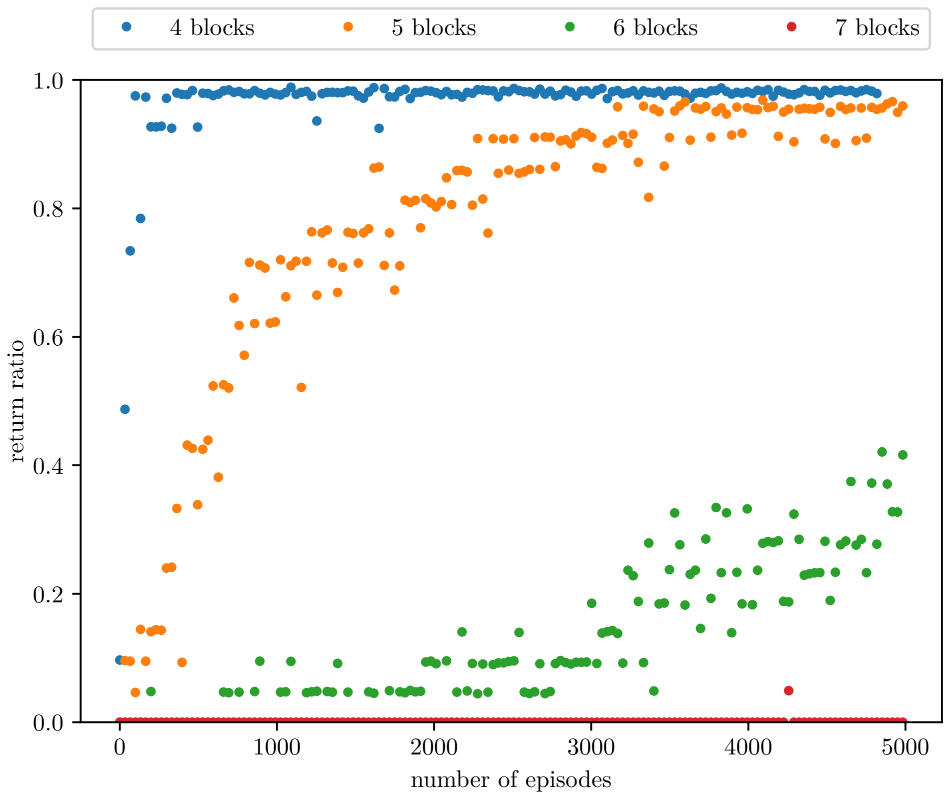


Figure 4.1: The agent's learning progress without planning in differently sized blocks worlds.

[1]Note that since individual points may be overlapping, less points of one color may actually be visible in the plots.

4.2 The Planning Agent

In this section, we shall look at the agent's performance when using different planning strategies.

The first experiment conducted with the planning component uses a blocks world consisting of 7 blocks. We now only learn based on 2,000 episodes, with `planOnEmptyPolicy` set to be true, and `planningFactor` set to zero. For the quality of the plans, we look at four different settings, where we set `planningHorizon` to 3, 4, 5 or 6.

All the other parameters not mentioned are unchanged from our previous experiment, leaving the `planningHorizon` (abbreviated pH) as the only variable parameter in this experiment. This time, we calculate the mean return ratio over 40 runs, and plotted 250 points per color; the results can be seen in Figure 4.2.

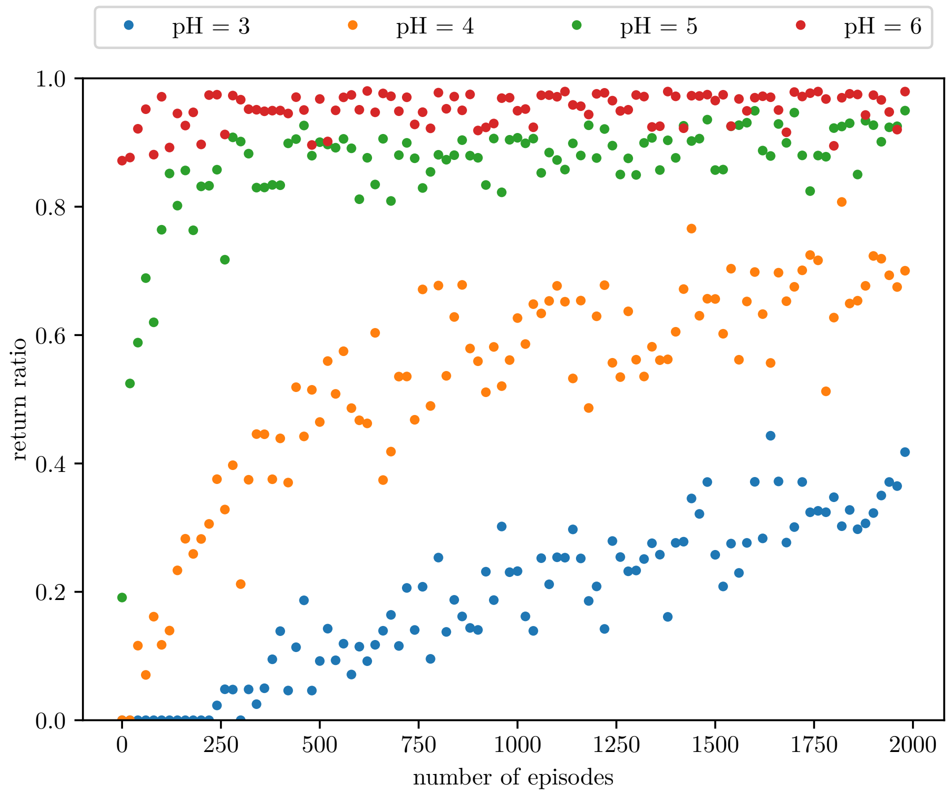


Figure 4.2: The agent's learning progress in a blocks world with 7 blocks using different planning horizons.

For our next experiment, we shall look at the influence of modifying the `planningFactor`.

Let us consider a blocks world of 7 blocks again, where we set `planOnEmptyPolicy` to be false, and declare the `planningHorizon` to always be 5. Stretching over 2,000 episodes, we conduct 4 trials, each using different settings for the `planningFactor` (abbreviated pF): 0.4, 0.5, 0.6 and 0.7. All other parameters remain constant and unchanged from our previous example. The results are shown in Figure 4.3.

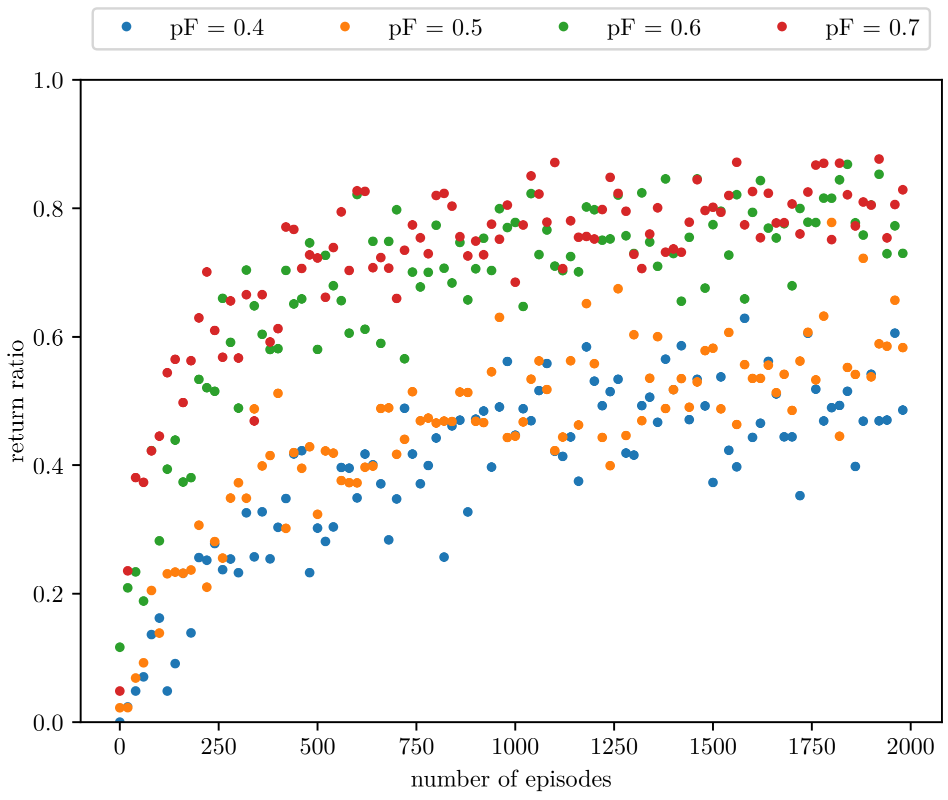


Figure 4.3: The agent's learning progress in a blocks world with 7 blocks using different planning factors.

For our last experiment, we want to analyze the learning behavior in even bigger blocks worlds. Looking at Figure 4.2, we can see that the return ratio, using a planning horizon of 6, quickly converges after about 500 episodes.

Based on this result, let us examine the return ratio's convergence for other blocks worlds, where the planning horizon is also close to the number of blocks in that respective blocks world. Concretely, let each blocks world of size n use a planning horizon of $n - 1$, which is exactly half as many steps as needed in the longest possible optimal plan. Let us set `planOnEmptyPolicy` to be true, and `planningFactor` equal to zero. We average over 20 runs; all other parameters are left unchanged. The results are illustrated in Figure 4.4.

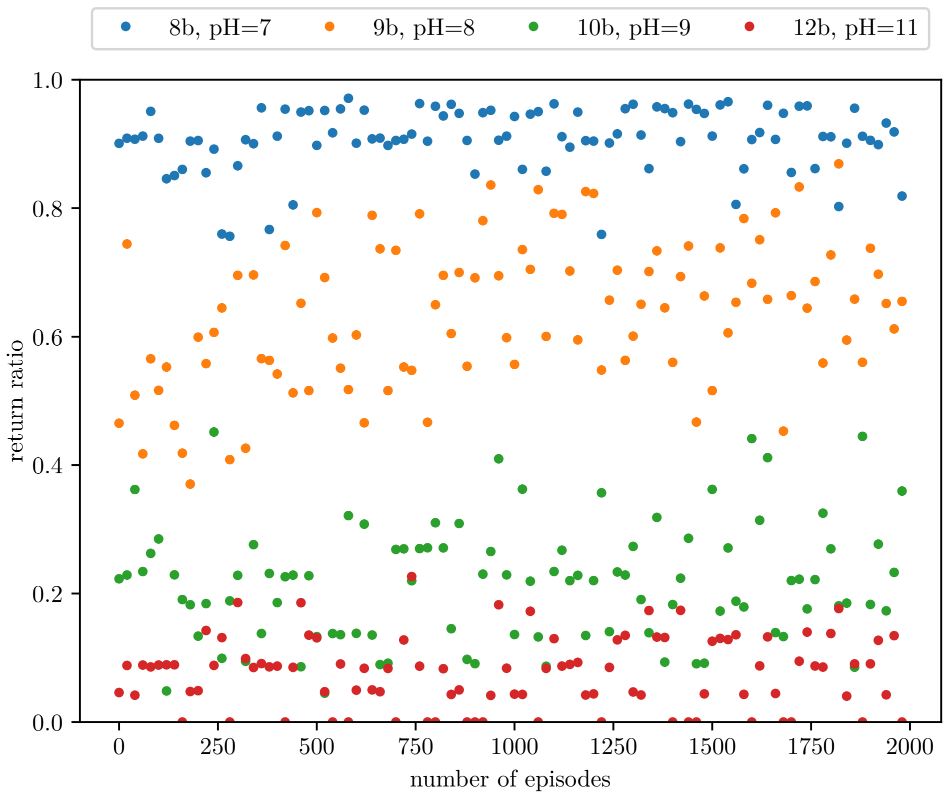


Figure 4.4: The agent's learning progress in blocks worlds with n blocks using a planning horizon of $n - 1$.

CHAPTER 5

Findings and Discussion

Armed with the results from the previous section, we now discuss our findings and provide answers to the questions we presented at the beginning of this thesis.

5.1 General Observations

Before proceeding with the questions about the performance of the framework, let us first address the phenomenon of all plots (especially Figure 4.1) from the previous chapter showing horizontal, terraced clusters of points.

These artefacts are due to the fact that we average the return ratios over multiple runs to create more stable results and limit the effect of “lucky guesses” the agent might take when performing actions at random.

To understand the reason why the clusters appear in the plots, let us look at the two major kinds of outcomes of an episode. In the first scenario, the agent reaches the goal state and receives a total reward of $+100$ minus the number of steps it took (which are limited by three times the number of blocks). In total, this still results in a high positive reward in the blocks world sizes we considered. The other outcome that can occur is that the agent does not reach a goal state, resulting in a negative reward as high as how many steps it took during that episode, which is exactly the upper bound of the maximal length of episodes, that is, three times the number of blocks.

Since those two scenarios result in either a high return ratio or a return ratio close to zero, there are two horizontal lines in our plot: one at the top, representing the episodes that reached the goal state, and another one at the bottom, where the goal state was not reached.

If we now average over r runs, there are usually $r + 1$ horizontal clusters. This is because we have episodes where, for example, during the first run, the goal was reached, but not in the second one. This yields an average return ratio of around 0.5, and we have three horizontal lines in the plot, where the top line contains the episodes where both runs reached the goal state, the middle one where one run reached the goal state, and the line close to zero where no run reached the goal state.

Shifting the attention to the scattering *within* the horizontal clusters, two factors play a role.

First, since we are using an Exploring Starts algorithm, we always select a random action at the beginning of each episode. This is why the plots never fully converge to a return ratio of 1.0, because the agent keeps exploring, even though an optimal policy is already found. It is important to note that Exploring Starts can have effects of varying magnitude on the return ratio, even if the policy may already be optimal, since for different lengths of paths to the goal state, a suboptimal first action can have a smaller or bigger impact on the total return.

Second, the scattering is dependent on the size of our reward for reaching the goal state. The higher the reward for reaching a goal state, the less scattered the lines are, since suboptimal actions have less effect on the total return. For example, consider an episode with three steps to the goal state and a reward of $+100$ for that goal state. Then, we have a total return of $+97$, since each step gets penalized with a negative reward of -1.

In contrast, if we set the reward for the goal state to be $+10$, the total reward will be $+7$. If the agent would perform an additional step because it chose a suboptimal action at the beginning of the episode (at random), the total reward would decrease to $+96$ or $+6$, respectively, which has a bigger *relative* effect on the total return where the smaller reward was attributed to the goal state.

This is the reason why we have comparatively little scattering in the horizontal, stair-like clusters; occurring suboptimal actions get compensated by the big reward of $+100$ at the end of a "successful" episode. Independent of the rewards given during an episode, if we were to average over more runs, the curves in the figures in the previous chapter would become smoother as we increase the number of runs.

5.2 Specific Observations

Let us now try to answer the five questions that we asked at the beginning of the thesis regarding the performance and scalability of our framework.

1. How does our framework perform at solving different blocks world sizes?

To answer this question, we use the number of episodes as an objective measure of performance, since metrics like time consumption are largely dependent on the computing resources available to the program.

While in theory, our Monte Carlo algorithm should always converge to an optimal policy (see Sections 2.2.3 and 3.1.1), the results achieved from our testing do not exhibit this behavior, because we only consider a finite number of episodes. Looking at the chart in Figure 4.1, one can easily see that the purely trial-and-error based Monte Carlo algorithm we developed reaches its limits when presented with a blocks world featuring 7 blocks.

While the return ratio quickly converges to our desired return ratio of 1.0 for 4 blocks, increasing the number of blocks to 6 already performs substantially worse. It is save to say that for any blocks world with $n > 6$ blocks the algorithm cannot achieve any noticeable learning progress within 5,000 episodes. On the other hand, if we consider the framework with the planning component enabled, like in the chart in Figure 4.4, it can be seen that the framework can perform well in bigger blocks world sizes as well.

Our second question was:

2. How does our framework scale when presented with bigger problem instances?

The main bottleneck in terms of scalability is the required enumeration of all states of the environment. As the number of blocks is increased, the memory used to store all states using our implementation with Python and pickle[1] quickly becomes unmanageable, as Table 5.1 illustrates.

n	memory (MB)
5	0.06
6	0.54
7	5.71
8	67.18
9	867.20

Table 5.1: Memory consumption of blocks worlds with n blocks.

Trying to generate a blocks world consisting of 10 blocks using our enumeration algorithm did not succeed on a number of consumer notebooks we tested, as they do not provide enough memory.

To overcome this barrier, we use an algorithm to generate pseudo-random start states for blocks worlds with 10 or more blocks (see Section 3.1.2). Since this algorithm generates certain categories of states more often than others [ST01], our learning algorithm gets trained on a somewhat biased data set. However, convergence to an optimal policy is still guaranteed if enough episodes are considered.

Let us move on to the third question:

3. Can the planning component aid the agent in solving bigger blocks worlds?

[1] pickle allows for the (de-)serialization of Python objects. For more information, see `https://docs.python.org/3/library/pickle.html`.

We first take a look at the first chart, Figure 4.1, where we can see that the agent stops to learn (within 5,000 episodes) as we increase the number of blocks to 7, as already stated. The chart in Figure 4.4 demonstrates that the agent incorporating planning into the system gains a clear edge over our “normal” algorithm, as it is able to receive a high return, even for bigger blocks worlds, all within fewer episodes. We can thus answer Question 3 positively.

The fourth question was:

4. How do different planning strategies influence the learning behavior?

To answer this, let us consider the second as well as the third chart in Figures 4.2 and 4.3, respectively. In both cases, we take advantage of the planning component, and both charts work on a blocks world consisting of 7 blocks. The experiment using a planning horizon as small as 5 already shows that the return achieved quickly increases and is close to optimal after just 1500 episodes.

Shifting our attention to the usage of the planning *factor*, which is described in more detail in Section 3.2, it is important to emphasize that for creating the plot in Figure 4.3 a *constant* planning horizon of 5 was used.

Even if the planning component is used with a probability of 40% before performing any action, the results are rather underwhelming, since the convergence speed is not improved by much. This is most likely because the planning horizon is comparatively small. However, increasing the probability to 70% yields a return ratio of about 0.8 after approximately 1500 episodes, resulting in a substantially better performance than the trial-and-error based agent.

Overall, we found the strategy of only planning when a state is first experienced (over all episodes) to cause less invocations of the planning component than using a planning factor of 70%, but the results are similar. We thus conclude that the planning factor is not as efficient as the “planning on empty policy” strategy.

Our fifth and last question was:

5. How does the planning horizon influence the convergence speed of the learning?

In order to answer this question, let us consider Figure 4.2 again. It shows that as the planning horizon is increased, the return ratio converges quicker, which is exactly what we expected, since a larger planning horizon is more farsighted and can therefore recommend better actions to the system.

In general, the learning appears to be quite sensitive to the value of the planning horizon, as the increase from 3 to 6 produces very different convergence behavior. Furthermore, it is worth noting that even a comparatively small planning horizon of 3 or 4 already steepens the curve noticeably. This is interesting for two reasons.

First, given that some (optimal) plans in a blocks world of 7 blocks are still 12 moves long, the relatively small planning horizon of 3 still achieves an observable effect after few episodes. Second, given that a planning horizon of 3 can only recommend a sensible action if a goal state is no farther than 3 moves away, it is still able to accelerate the learning process for instances where the goal is more than 3 steps away.

To summarize, one can safely say that the plain reinforcement learning algorithm we developed becomes inadequate as the number of blocks is increased to more than 6 blocks. However, as our experiments showed, the learning progress can be significantly improved by taking advantage of the planning component. The enhanced algorithm occasionally using answer set planning can solve bigger blocks worlds, even if the plans only look a few steps ahead.

CHAPTER 6

Related Work

This section presents and touches on work similar to what is done in this thesis. Specifically, we mention literature also combining reinforcement learning with answer set programming.

Nickles. Similar work was conducted by Nickles [Nic12a, Nic12b], where a complete software solution for integrating reinforcement learning with answer set programming was developed. The software also uses answer set programming for representing the environment as well as constraining actions and state transitions. The experiments Nickles presented test several relational reinforcement learning algorithms like *Q-Learning* and *SARSA*, and a number of different answer set grounders and solvers are supported. All variations show a significant improvement in the average number of steps taken to reach a goal state, where the performance gain is in part due to the constrained state-action space.

Our framework differs from the work by Nickles, as we employ Monte Carlo methods for reinforcement learning. Additionally, we use answer set programming not only for representing the environment and actions, but also for planning at decision time. Also, it is worth noting that Nickles' software is no longer maintained and did not run on the machines we tested.

Džeroski et al. Another similar concept was introduced by Džeroski, De Raedt and Driessens [DDRD01]. Like in this thesis, the blocks world is used to benchmark their combination of reinforcement learning with relational learning or inductive logic programming. Unlike our approach, they employ relational representations abstracting from specific goals to reuse learned behavior in more complex, new situations.

Specifically, they use structured representations enabling the description of infinite worlds and variables, allowing the agent to gather a more abstract knowledge of the concept of goals and blocks world sizes. On the contrary, our framework learns a policy tied to a specific goal state and a specific blocks world size, thus the knowledge cannot be reused in new situations or blocks worlds of different size.

Illanes et al. Work by Illanes, Yan, Icarte and McIlraith [IYIM20] describes a framework using high-level symbolic action models to define final-state goal tasks for automatically producing their corresponding reward functions. Additionally, automated planning is used to synthesize plans to guide trial-and-error based reinforcement learning to learn policies more efficiently.

Similar to our findings, their experimental results show that their approach can significantly outperform Q-Learning and hierarchical reinforcement learning methods if the number of training steps is limited. For instance, they consider the office world domain, where their approach needs only 70,000 training steps to converge to a policy that results in traces that were typically 10 steps away from being optimal. In contrast, hierarchical reinforcement learning needs at least 1,800,000 steps before finding a policy of comparable quality.

Interestingly, their enhanced approach reaches policies that result in slightly worse solutions than the hierarchical reinforcement learning method. Our enhanced approach differs in that regard, since it *always* finds better solutions than the purely trial-and-error based counterpart.

Dos Santos et al. Lastly, the work presented by dos Santos, Santos, Ferreira et al. [dSSF$^+$19] solves various relaxed versions of spatial puzzles, where an algorithm combining answer set programming with Markov decision processes learns heuristics to accelerate the learning process. Answer set programming is used to represent the planning domain as a Markov decision process, and a Q-Learning algorithm is used to find optimal policies. While our work only considered deterministic problems, their system works on both deterministic and nondeterministic instances.

In addition, they compared four different learning algorithms and found that the heuristic accelerated versions outperform their counterparts in all cases. They concluded that these heuristics provide important information to guide and accelerate the learning process.

CHAPTER 7

Conclusion

In this thesis, we introduced a framework for combining reinforcement learning with answer set programming, where the latter is used to both simulate the environment, states and actions as well as to guide action selection by generating plans.

While seeing that our implemented Monte Carlo reinforcement learning algorithm quickly reaches its limits in bigger blocks worlds, we found that the occasional usage of planning can greatly improve the convergence speed of policies. In particular, we found planning in states which were never experienced before to be highly effective.

In addition to that, we discovered that even small planning horizons in the range of the number of blocks in the environment can reach near optimal policies for bigger blocks world instances after just a few hundred episodes. Moreover, we found that describing the environment with answer set programming can be done in a concise manner allowing for great extensibility, and using the API provided by Potassco, the integration with PYTHON's object-oriented classes can be done efficiently. Another option is to use HEXLITE[1], a system developed at the Institute of Logic and Computation at the TU Wien that integrates answer set programming with external computations.

Future work will implement a random start state generator using parity constraints to further decouple our framework from the blocks world domain. Furthermore, the planning factor will add a flag to mark states in which the planning component was already used to avoid multiple invocations of the planning component, and we will allow for the gradual reduction of the planning factor during a series of episodes. Lastly, CLINGO will continue to run throughout the learning process to allow for new facts to be added dynamically, without having to restart CLINGO.

[1]`https://github.com/hexhex/hexlite`

Bibliography

[BET11] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[BKI14] Christoph Beierle and Gabriele Kern-Isberner. *Methoden wissensbasierter Systeme: Grundlagen, Algorithmen, Anwendungen.* Springer Fachmedien Wiesbaden, Wiesbaden, Germany, 2014.

[Che91] Stephen V. Chenoweth. On the NP-hardness of blocks world. In *AAAI-91 Proceedings*, pages 623–628. The AAAI Press, 1991.

[DDRD01] Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine Learning*, 43:7–52, 2001.

[dSSF$^+$19] Thiago Freitas dos Santos, Paulo E. Santos, Leonardo Anjoletto Ferreira, Reinaldo A. C. Bianchi, and Pedro Cabalar. Heuristics, answer set programming and Markov decision process for solving a set of spatial puzzles. *ArXiv*, abs/1903.03411, 2019.

[EIK09] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In *Reasoning Web. Semantic Technologies for Information Systems. Lecture Notes in Computer Science*, volume 5689, pages 40–110. Springer Berlin Heidelberg, 2009.

[EJKS19] Flavio Everardo, Tomi Janhunen, Roland Kaminski, and Torsten Schaub. The return of xorro. In *Logic Programming and Nonmonotonic Reasoning*, pages 284–297. Springer International Publishing, 2019.

[Fah74] Scott Elliott Fahlman. A planning system for robot construction tasks. *Artificial Intelligence*, 5(1):1–49, 1974.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of International Logic Programming Conference and Symposium*, volume 88, pages 1070–1080. The MIT Press, 1988.

[GN92] Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2):223–254, 1992.

[IYIM20] León Illanes, Xi Yan, Rodrigo Toro Icarte, and Sheila A. McIlraith. Symbolic plans as high-level instructions for reinforcement learning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 540–550, 2020.

[Lif08] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence*, volume 3, pages 1594–1597. The AAAI Press, 2008.

[Lif19] Vladimir Lifschitz. *Answer Set Programming.* Springer Nature Switzerland AG, Cham, Switzerland, 2019.

[Nic12a] Matthias Nickles. Integrating relational reinforcement learning with reasoning about actions and change. In *Inductive Logic Programming*, pages 255–269. Springer Berlin Heidelberg, 2012.

[Nic12b] Matthias Nickles. A system for the use of answer set programming in reinforcement learning. In *Logics in Artificial Intelligence*, pages 488–491. Springer Berlin Heidelberg, 2012.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, Cambridge, Massachusetts, 2nd edition, 2018.

[ST01] John Slaney and Sylvie Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1):119–153, 2001.

[SW11] Claude Sammut and Geoffrey I. Webb. *Encyclopedia of Machine Learning.* Springer Science & Business Media, New York, New York, 2011.

[Sze10] Csaba Szepesvári. Algorithms for reinforcement learning. In *Synthesis Lectures on Artificial Intelligence and Machine Learning.* Morgan & Claypool Publishers, 2010.

[WR20] Che Wang and Keith W. Ross. On the convergence of the Monte Carlo Exploring Starts algorithm for reinforcement learning. *ArXiv*, abs/2002.03585, 2020.