

Titre: Reward Shaping in Reinforcement Learning with Constraint
Title: Programming

Auteur: Chao Yin
Author:

Date: 2024

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Yin, C. (2024). Reward Shaping in Reinforcement Learning with Constraint
Citation: Programming [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie.
<https://publications.polymtl.ca/61334/>

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/61334/>
PolyPublie URL:

**Directeurs de
recherche:** Gilles Pesant, & Quentin Cappart
Advisors:

Programme: Génie Informatique
Program:

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Reward shaping in reinforcement learning with constraint programming

CHAO YIN

Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

Génie informatique

Décembre 2024

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

Reward shaping in reinforcement learning with constraint programming

présenté par **Chao YIN**

en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

a été dûment accepté par le jury d'examen constitué de :

Daniel ALOISE, président

Gilles PESANT, membre et directeur de recherche

Quentin CAPPART, membre et codirecteur de recherche

Antoine LEGRAIN, membre

DEDICATION

*To my family,
who made this journey possible.*

ACKNOWLEDGEMENTS

I would like to thank my advisor Gilles Pesant for supervising me in this research project. He has dedicated tremendous time and effort in helping my research project come to fruition. During moments when I felt stuck in my project, he provided valuable ideas and insight to point me in the right direction. There were other times when I had doubts about my work's worth of being submitted to be published and it is thanks to his insistence and belief in my work that I managed to publish my conference paper at the CPAIOR 2024 conference.

I would also like to thank my co-advisor Quentin Cappart for overseeing my project and providing valuable insights when I have questions regarding Machine Learning. It is also thanks to his help I was able to publish my first conference paper. Despite having a lot of students, he has found time to meet with me regularly, which I truly appreciate.

I'd like to thank my colleagues at the QUOSSECA group for their support, without them this journey would've been a lot more difficult. I could always count on them to provide valuable insight and helpful feedback.

Finally, I thank my family whom I love for always believing in me.

RÉSUMÉ

Le façonnage de la récompense est une technique utilisée en apprentissage par renforcement pour modifier le comportement de l'agent en donnant des récompenses à des comportements désirables. Cette technique est aussi utilisée pour accélérer l'entraînement de l'agent en guidant l'agent dans la bonne direction pendant l'entraînement. Par contre, il est typique que la récompense soit façonnée manuellement à l'aide des experts humains qui peuvent identifier des comportements désirables. Nous proposons un façonnage de la récompense basé sur le potentiel qui sert à automatiser le façonnage de récompense à partir d'un modèle du problème en programmation par contraintes. En utilisant notre méthode, nous pouvons spécifier des comportements désirables à partir de l'interface de programmation par contraintes et l'utiliser pour peaufiner un réseau de neurones génératif préentraîné. Notre méthode peut aussi servir à accélérer l'entraînement pour un problème d'apprentissage par renforcement. Alors que notre méthode nécessite un modèle de programmation par contraintes, l'identification des comportements désirables est faite automatiquement grâce au mécanisme de raisonnement d'un solveur de programmation par contraintes. Une autre nouveauté de notre approche est le fait que le solveur que nous utilisons utilise une approche probabiliste à ses raisonnements en utilisant la propagation des convictions; nous démontrons que ceci améliore notre approche. Dans la première partie de notre travail, nous considérons le problème du peaufinage d'un modèle génératif d'IA pour la génération de musique. Dans ce problème, le modèle peaufiné doit respecter les règles imposées par un utilisateur et retenir en même temps un niveau de similarité comparé au modèle avant le peaufinage. Nous démontrons que notre approche de façonnage de récompense à base de potentiel améliore la performance finale et accélère la vitesse d'entraînement en faisant une comparaison avec un travail existant qui utilise aussi le façonnage de récompense avec la programmation par contraintes pour le même problème. Dans la deuxième partie de notre travail, nous convertissons un modèle SAS+ pour un modèle de planification d'IA en un modèle de programmation par contraintes que nous utilisons pour créer des récompenses façonnées. Nous utilisons une approche d'apprentissage par renforcement existante sur le problème de Blocks World avec une récompense clairsemée pour démontrer que notre approche accélère l'entraînement et améliore le modèle entraîné final.

ABSTRACT

Reward shaping has been applied in reinforcement learning to shape the behavior of an agent by rewarding desirable behaviors. It can also be used to speed up the training of the agent by pointing the agent in the right direction during training. However, reward shaping is usually handcrafted with the help of a human expert who is able to identify the desired behaviors. It also needs to be done with care as poorly designed reward shaping can train the agent to have unforeseen behavior. We propose a potential-based reward shaping method to automate the creation of shaped rewards by modeling a given problem using constraint programming. Using our method, we are able to specify desirable behaviors through the interface of constraint programming and use it to finetune a pretrained neural network. We are also able to generate shaped rewards for a given reinforcement learning problem to improve training. While our method requires the creation of a constraint programming model, the identification of desirable behaviors is done in an automated fashion using the reasoning mechanism of a constraint programming solver. Another novelty of our approach is that the solver we use incorporates probabilistic approximations in its reasoning with belief propagation, which we show is beneficial to our reward shaping method. In the first part of our work, we consider the problem of finetuning a generative AI model for music generation. A successful finetuning of this model entails that this model respects a set of rules imposed by a user and that it retains a level of similarity to itself prior to finetuning. Using an existing work for this problem which also uses reward shaping with the same constraint programming solver as ours, we demonstrate that our potential-based reward shaping method accelerates training and improves the final performance of the finetuned model. In the second part of our work, we consider the problem of solving AI-planning problems. We give an algorithm which converts a SAS+ model for an AI-planning problem to a constraint programming one that we then use to generate shaped rewards. Using an existing work which trains an agent to solve the problem of Blocks World and a sparse reward, we show that our shaped reward improves training as well as the final performance of the trained model.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
RÉSUMÉ	v
ABSTRACT	vi
TABLE OF CONTENTS	vii
LIST OF TABLES	x
LIST OF FIGURES	xi
LIST OF SYMBOLS AND ACRONYMS	xiii
CHAPTER 1 INTRODUCTION	1
1.1 Improving sample efficiency in RL with environment model information . . .	2
1.2 Representing RL problems using CP models	2
1.3 Research objectives	3
1.4 Outline of thesis	4
CHAPTER 2 TECHNICAL BACKGROUND	5
2.1 Machine learning	5
2.1.1 Supervised learning	5
2.1.2 Neural networks	6
2.1.3 Reinforcement learning	10
2.1.4 Markov decision process	11
2.1.5 Value based methods	12
2.1.6 Policy gradient methods	13
2.2 Constraint programming	17
2.2.1 Basic definitions	17
2.2.2 Local consistency algorithm	19
2.2.3 Backtrack solution searching	20
2.2.4 Approximating the solution density	20
2.3 AI planning problems	25

CHAPTER 3	LITTERATURE REVIEW	29
3.0.1	Provably efficient RL in the tabular setting	29
3.0.2	Sample efficiency challenges in DRL and recent trends	30
3.1	Variance reduction for policy gradient methods	32
3.2	The problem of reward shaping	33
3.3	Constraint programming for reward shaping	34
CHAPTER 4	ARTICLE 1: AN IMPROVED NEURO-SYMBOLIC ARCHITECTURE TO FINE-TUNE GENERATIVE AI SYSTEMS	36
4.1	Abstract	36
4.2	Introduction	36
4.3	Background	37
4.3.1	Reinforcement Learning	37
4.3.2	Belief Propagation-Based Constraint Programming	38
4.4	Related Work	38
4.5	Proposed Improved Framework	39
4.5.1	Simplified Architecture	40
4.5.2	Evaluating Constraint Violations	41
4.5.3	Reward Signal	41
4.6	Empirical Evaluation	42
4.6.1	CP Model	42
4.6.2	DQN training and evaluation	43
4.6.3	Comparing reward signals	43
4.6.4	Experimental results	44
4.7	Conclusion	44
CHAPTER 5	REWARD SHAPING FOR AI-PLANNING IN RL	46
5.1	RL setting for AI-planning	46
5.2	Creating a CP model for an AI-planning problem	49
5.2.1	Atomic factor as a cost-regular constraint	50
5.2.2	Potential function for reward shaping	52
5.3	Reward shaping algorithm using CP	52
5.3.1	Interpretation of the potential function	54
5.3.2	Design choices for the CP model affecting the shaped reward.	56
5.3.3	Curriculum learning	57
5.4	Experimentation	57
5.4.1	CP models for Blocks World	59

5.4.2 Results	60
CHAPTER 6 CONCLUSION	65
6.1 Summary of works	65
6.2 Limitations	66
6.3 Future research	66
REFERENCES	67

LIST OF TABLES

Table 4.1	Hyper-parameters for our models	44
Table 5.1	Solutions for the CP model of Blocks World	56
Table 5.2	Hyper-parameters for training in the Blocks World problem	58
Table 5.3	Optimality gaps for BW problems of varying numbers of blocks.	62
Table 5.4	Success rate for BW problems of varying numbers of blocks.	62

LIST OF FIGURES

Figure 1.1	At time t , the state of the environment is s_t . After observing s_t , the agent decides to take action a_t and receives reward r_{t+1} and observes state s_{t+1}	2
Figure 2.1	Feedforward neural network with two hidden layers.	8
Figure 2.2	RNN in a many-to-many setting.	9
Figure 2.3	Factor graph for the CP model	22
Figure 2.4	BP algorithm implementation in MiniCPBP [1]	24
Figure 2.5	True marginals for the example CSP [1].	24
Figure 2.6	Approximated marginals using MiniCPBP [1].	25
Figure 2.7	Blocks World problem example	25
Figure 2.8	SAS+ model for a Blocks World problem of two blocks	27
Figure 2.9	Induced transition system of a Blocks World problem	27
Figure 2.10	Induced FTS $\langle \Theta^A, \Theta^B \rangle$ for Blocks World example of two blocks . . .	28
Figure 4.1	CP-based RL-Tuner framework and our proposed modification. . . .	40
Figure 4.2	Evaluation of agents (using four different reward signals) during training after each epoch. Agents generate 10 melodies of length 32 and 64 which are then evaluated using our two metrics: LSTM similarity and constraint satisfaction. The curves and shaded areas respectively represent the mean and standard deviation.	45
Figure 5.1	BW planning problem in the RL setting with potential-based reward shaping.	48
Figure 5.2	Automata for the constraints cost-regular(Θ^B) and cost-regular(Θ^A) .	56
Figure 5.3	Reward multiplier hyperparameters for our experiments.	59
Figure 5.4	SAS+ model Π_1 for Blocks World	59
Figure 5.5	SAS+ model Π_2 for Blocks World, $Equal(\cdot, \cdot)$ is 1 if both arguments are equal, 0 otherwise.	60
Figure 5.6	SAS+ model Π_3 for Blocks World	61
Figure 5.7	Curriculum learning based training of Blocks World from instances of 2 blocks to 12 blocks. Green: Π_2 (BP), blue: Π_1 (BP), red: Π_3 (BP), black-dashed: sparse baseline.	62

Figure 5.8	Curriculum training of Blocks World from instances of 2 blocks to 10 blocks. Blue-dashed: Π_1 (no BP), green-dashed: Π_3 (no BP), yellow: Π_1 (BP), red: Π_3 (BP), purple-dashed: Π_2 (no BP) black-dashed: sparse baseline	64
------------	---	----

LIST OF SYMBOLS AND ACRONYMS

NN	Neural Network
RL	Reinforcement Learning
CP	Constraint Programming
AI	Artificial Intelligence
SL	Supervised Learning
MSE	Mean Square Error
FF	Feed Forward Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short Term Memory
MDP	Markov Decision Process
DQN	Deep Q-network
BP	Belief Propagation
CSP	Constraint Satisfaction Problem
GAC	Generalized Arc Consistency
NLM	Neural Logic Machine
PBRs	Potential-Based Reward Shaping
BW	Blocks World

CHAPTER 1 INTRODUCTION

The abundance of data and advancements made in both hardware and software technologies to process them have yielded impressive achievements in solving a wide variety of difficult problems through the use of neural networks (NNs). This is due to the ability of deep NN to learn low dimensional representations for high dimensional data. Reinforcement learning (RL) is cast as the problem of an agent learning the best actions to take in a given context by interacting with an environment and receiving rewards for its actions. Reinforcement learning has also seen renewed interest in its application to solve problems with large numbers of states by leveraging the ability of NNs to process high dimensional data. For instance, the ability to process images from a screen can be used to play ATARI games at a level above that of a human expert on a select number of games [2]. Some notable success stories for NNs include their use in AlphaGoZero to train an AI system to defeat the world’s best player at the game of Go [3], a game with a number of legal board positions greater than the number of atoms in the observable universe [4]. AlphaStar [5] was able to defeat the world’s best players in a real time strategy game called Starcraft II.

These achievements, however, required using a huge amount of data. AlphaGo used 4.9 million games of self play, AlphaStar used 200 years of StarCraft II gameplay. In the latter’s work, training was also augmented with guidance from experts of the game to improve sample efficiency. This is done by identifying states of the game that are "valuable" and rewarding the agent for entering them. Because large amount of training data is required for the successful application of the aforementioned methods, their use is seriously limited in domains where the collection of data is more costly. For example in robotics [6] where the collection of data can be time consuming and autonomous driving [7] which has the additional problem of being risk averse. In an effort to stimulate research on reducing this dependency on abundant data, the MineRL Competition on Sample Efficient Reinforcement Learning Using Human Priors [8] was created where an agent in the game of Minecraft must obtain a diamond, a task that is highly hierarchical in nature and can span a very long duration, which makes learning very difficult. The competitors are given a limited dataset of human demonstrations and a constraint on the computational budget to train their agent.

In the same spirit, our work aims to make contributions in RL by improving sample efficiency. Constraint programming (CP) is a tool that can perform automated reasoning on assignment problems where a set of constraints must be satisfied. We propose a method that uses CP to automate the extraction of useful information of an environment in order to efficiently train

an RL agent.

1.1 Improving sample efficiency in RL with environment model information

In reinforcement learning, an agent has to learn an optimal policy to take actions for any given context the agent finds itself in so as to maximize the collection of rewards given by a reward function (see figure 1.1). In prevalent RL techniques such as Q-learning [9] and REINFORCE [10] that are *model-free*, it is assumed that the agent does not have access to the environment model, as opposed to model-based methods, such as policy iteration, which either assumes such access or attempts at learning the transition function of the model. Model-free methods offer more flexibility and use less space due to the lack of the need to specify or learn the environment model. However, there is empirical evidence that suggests such methods can be less sample efficient — we discuss this in our literature review. The agent must interact with the environment and receive rewards from a reward function which it treats as a black box without having privileged access to the dynamic of the environment or the reward function itself. In our work, we develop a CP-based framework where the RL environment is modeled by a CP model which allows us to extract useful information about the structure of the reward function and use it, for instance, to modify the reward received by the agent to ease learning.

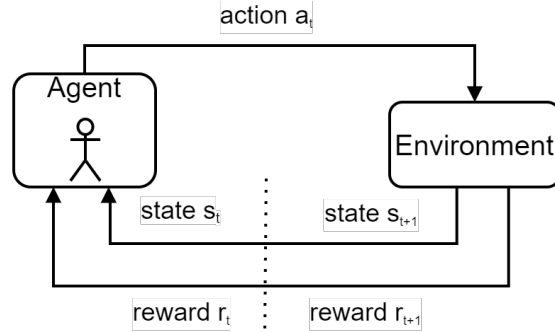


Figure 1.1 At time t , the state of the environment is s_t . After observing s_t , the agent decides to take action a_t and receives reward r_{t+1} and observes state s_{t+1} .

1.2 Representing RL problems using CP models

There are several challenges to create a CP model for a RL environment model. CP deals primarily with combinatorial problems, in which variables have a finite number of values they can take. This is incompatible with some environments where the set of actions that an agent can take and the set of states it finds itself in can be infinite or even uncountable. Although

discretization is possible in some cases, we will limit the scope of our work to only consider RL problems that CP is known to be able to solve. The first type of problem we consider is the shaping of the behaviour of a generative model for sequences of discrete values. The discrete nature of this task allows for constraints we would like to impose on the generated sequence to be directly modeled using CP. This allows a user to shape the behaviour of the model by specifying some rules. The second type of problem we consider is AI planning, which is the problem of taking a sequence of actions to go from an initial state to a goal state while minimizing the cost. Both RL and CP are well positioned to solve AI-planning problems. While popular RL algorithms that are *model-free* solve sequential decision problems through learning without learning the model itself, a model must be explicitly defined in CP in order to solve it with a CP solver. We propose in our method to extract useful information about the environment by modeling the problem using CP. We hypothesize that it will alleviate some the inefficiencies that have been empirically observed in the literature in model free methods where the environment model is assumed to be a black box. A challenge comes from choosing the appropriate CP model, or even obtaining a single one for a given environment model. There is potentially a dilemma in choosing between a higher performing CP model which can extract more meaningful information from the environment or a computationally cheaper model that extracts less information. We process our CP model in the MiniCPBP solver [1] equipped with the ability to estimate a probability distribution for each of the variables by using an inference technique called belief propagation (BP). Traditionally, a CP solver filters out values from the domain of each variable that are not possible (probability of zero) in order to be a solution. The remaining values are thus believed to have a probability of being greater than 0 but BP gives us an estimated value of the probability. However, the use of BP can be costly, we will study whether the estimated distribution improves learning performance over an equiprobable distribution over the values.

1.3 Research objectives

In this work, we aim to develop a method using CP which improves sample efficiency in RL as well as be used to shape the behavior of the agent using a set of user-specified rules expressed as constraints. We will provide a theoretical framework under which our method can be interpreted as an addition to techniques used in well-established methods. We study through experiments the effects of 1. the choices of the CP models and 2. use of BP on the effectiveness of our reward shaping method.

We aim to answer the following research questions:

- (Q1) Which RL problems are suitable for the application of our method?
- (Q2) Can the use of CP improve sample efficiency in RL?
- (Q3) How does the choice of the CP model influence training performance?
- (Q4) Does the use of BP improve training performance?

1.4 Outline of thesis

We organize the thesis in five chapters:

- Chapter 2 - The necessary background to understand our proposed method.
- Chapter 3 - Literature review on existing work on techniques improving sample efficiency.
- Chapter 4 - Improvements on the original CP based RL-Tuner.
- Chapter 5 - Application of our method on RL for AI planning tasks.
- Chapter 6 - Limitations and future work.

CHAPTER 2 TECHNICAL BACKGROUND

2.1 Machine learning

Machine learning is a subset of artificial intelligence that learns to make predictions from data by using algorithms and statistical models. A formal definition from the ML textbook [11] defines ML as a computer program learning from experience in the sense of the following quote taken from it. "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E " (Thomas, 1999). There are three major paradigms of machine learning which are supervised learning, unsupervised learning and reinforcement learning. Preference for any paradigm depends on the nature of the task at hand but they are not mutually exclusive and can compliment each other in the appropriate setting. Although our contributions are in reinforcement learning, deep reinforcement learning makes extensive use of supervised learning techniques to process large space and action spaces.

2.1.1 Supervised learning

In supervised learning (SL), a model is trained using labeled data, meaning the dataset used to train the model is : $\mathcal{D} = \{(x_1, y_1) .. (x_n, y_n)\}$. For each example in $\mathcal{D}$, there is a vector x_k of features and its corresponding label y_k . The supervised learning problem is to learn a function that maps features to their labels that approximates the true function from which the dataset was sampled. This usually begins by considering a set of hypothetical functions $\mathcal{H}$, which we also refer to as the model, from which we search for the function that best fits $\mathcal{D}$. Usually, a model contains functions h_W parameterized by weights W and biases b that must be learned. For instance, our model could be linear functions in the form of $h_W(x) = W^T x + b = \hat{y}$. Even though this model would be limited in its expressivity and would not perfectly fit data generated by, say, the quadratic function, the simpler model is preferred over more complex ones if they "fit" the data just as well. We consider two kinds of tasks in SL. In a regression task, the output belongs to a continuous set of real numbers. For example, predicting the height of a person given some input which represents some physical attributes. In a classification task, the label y belongs to a finite set of classes. This could be, for example, predicting the sentiment of a text as being either positive or negative. One approach to the classification problem is to learn a *discriminant function* which maps the input directly to its label. A different approach to the classification problem is to learn a

generative model. This involves learning the probability distribution (or density) over all input and label pairs (x, y) rather than simply predicting the label given an input. With such a model, it is possible to generate new data by sampling synthetic input and label pairs from the learned function.

Loss function

In order to find the function $h_W \in \mathcal{H}$ that fits the dataset $\mathcal{D}$, we need a loss function to define what "fitting" a function on a dataset means. A loss function's purpose is to quantify the disparity between the predicted label and the true label. A widely used loss for the regression task is $\sum_{(x,y) \in \mathcal{D}} |h_W(x) - y|^2 / |\mathcal{D}|$ which is known as the mean square error (MSE) loss. If our model was a linear function and we used the *MSE* loss to fit a dataset, then we would have the very well studied problem of linear least squares which has an analytical solution for the optimal weights given by $(X^T X)^{-1} X^T y$ where X is a $N \times K$ matrix containing in its rows the features of size K for each of its N samples and y is a vector of size N of labels. However, analytical solutions might not always exist for more complex models so we need a more generic method of optimizing the weights. Because the h_W is differentiable with respect to the weights and the loss is also differentiable with respect to the output of the model, it follows from the chain rule that the loss is differentiable with respect to the weights. This allows us to use gradient methods to search for the optimal set of weights by computing the gradient of the loss and using it to iteratively update the weights. For the classification task, a widely used loss function is the cross-entropy loss for models whose output is a probability distribution over classes. The cross-entropy loss is given as $\sum_{(x,y) \in \mathcal{D}} CE(\hat{y}, y) / |\mathcal{D}|$ where $CE(\hat{y}, y) = -\log(p(\hat{y} = y))$, where $p(\hat{y} = y)$ is the predicted probability of the correct label y . In the next section, we introduce an important model called feedforward neural networks. We then use them to illustrate the optimization process by using the backpropagation algorithm, which is an efficient method to compute the gradient of the loss in order to find the optimal weights.

2.1.2 Neural networks

An important tool in machine learning are neural networks. In this section, we describe neural networks in its simplest form, the feedforward neural network (FF) which, despite its simplicity, is theoretically proven to have the ability to approximate any function given enough depth or width. The FF has multiple layers, each having a set of neurons which are the weighted sum of the neurons on the previous layer followed by a nonlinear activation function. Consider a FF of L layers and width M and some nonlinear function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$.

At each hidden layer of depth ℓ , there are $M^\ell \leq M$ neurons. For simplicity sake, we write $\sigma(\bar{x})$ as an element wise operation using $\sigma(\cdot)$ if $\bar{x}$ is a vector. We define the j^{th} pre-activation unit of the ℓ^{th} layer z_j^ℓ and the activation unit of the same layer a_j^ℓ . We write z^ℓ and a^ℓ to denote the vector of pre-activation and activation units of size M . The values of z^ℓ can be obtained from the activation units of the previous layer with the following formula.

$$z^\ell = w^\ell \cdot a^{\ell-1} + b^\ell \quad (2.1)$$

Where w^ℓ is a weight matrix of size $M^\ell \times M^{\ell-1}$ and term a^ℓ is the activation unit is defined as $a^\ell = \sigma(z^\ell)$. The last term b^ℓ is a vector of bias values. Note that the number of units M^L in the final layer dictates the shape of the output of the FF. If $N = 1$ for example, the output is a single scalar value. The figure 2.1 depicts a FF of 2 hidden layers. Each edge represents the weight used to compute the pre-activation unit.

Backpropagation for FF

We now describe a gradient based method for optimizing the parameters in the FF. Namely, the weights $w_{i,j}^\ell$ which are the (i, j) entries of the weight matrix in layer ℓ and b_i^ℓ which is the i^{th} bias term in layer ℓ . To do this, we must compute the partial derivative $\partial C / \partial w_{i,j}^\ell$ and $\partial C / \partial b_i^\ell$ where C is the loss which we seek to minimize. We define the following term which we call the error of layer ℓ .

$$\delta_j^\ell = \partial C / \partial z_j^\ell \quad (2.2)$$

We first make the observation that the error can be calculated as follows by using the chain rule.

$$\delta_j^\ell = \frac{\partial C}{\partial a_j^\ell} \sigma'(z_j^\ell) \quad (2.3)$$

Secondly, we observe that the error δ^ℓ can be calculated from the error of the next layer $\delta^{\ell+1}$.

$$\delta^\ell = ((w^{\ell+1})^T \cdot \delta^{\ell+1}) \odot \sigma'(z^\ell) \quad (2.4)$$

Where $\odot$ is the element-wise product between two vectors (or matrices) of the same size. With these equations, we are able to calculate the error of each layer, starting from the final layer. We now give the equations for calculating the partial derivatives we need for gradient descent. The partial derivative with respect to the bias term is the same as its corresponding error of the same layer.

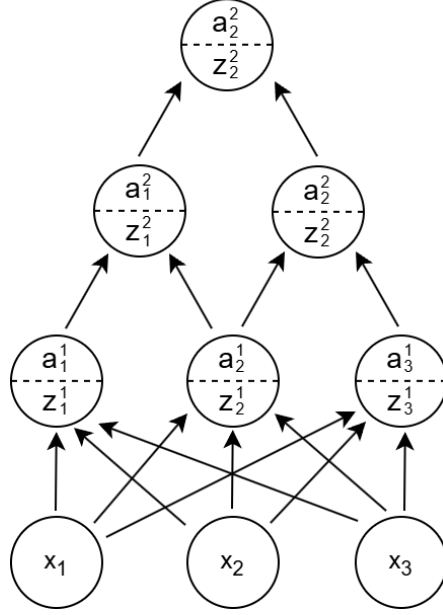


Figure 2.1 Feedforward neural network with two hidden layers.

$$\frac{\partial C}{\partial b_i^\ell} = \delta_i^\ell \quad (2.5)$$

The partial derivative with respect to the weight is given as follows:

$$\frac{\partial C}{\partial w_{i,j}^\ell} = a_j^{\ell-1} \delta_i^\ell. \quad (2.6)$$

Thus, the partial derivatives with respect to the weights as well as the bias terms can be efficiently calculated. First, the activated units a^ℓ are calculated by feeding each layer to the next starting with the input layer using Equation 2.1. Next, the error δ^ℓ of each layer is calculated starting from the final layer 2.4. The partial derivatives can then be obtained using Equation 2.5 and 2.6.

$$b^\ell = b^\ell + \alpha \frac{\partial C}{\partial b_i^\ell} \quad (2.7)$$

$$w_{i,j}^\ell = w_{i,j}^\ell + \alpha \frac{\partial C}{\partial w_{i,j}^\ell} \quad (2.8)$$

We then update our parameters using the equations 2.7 and 2.8 by adding the gradient values multiplied by α which is also known as the step size.

Recurrent neural networks

Recurrent neural networks (RNNs) are a family of neural networks which are used to process sequential data. What differentiates it from FF is its ability to take as input sequences of arbitrary length. We can define an RNN as follows. Suppose we have a sequential input $\bar{x} = x_1, \dots, x_N$ of real vector inputs. At time step t of the sequence, we give the following recursive definition for the hidden state of size H of the RNN.

$$z_{t+1} = W_h x_t + U_h h_t + b_h, \quad \forall t \in \{1..N\}$$

$$h_{t+1} = \sigma(z_{t+1})$$

Where pre-activation state z_{t+1} has a size of H . The weight matrices W_h and U_h are of the sizes $H \times M$ and $H \times H$ and the bias vector b_h is of the size H , they contain the parameters that need to be learned. The hidden state is then obtained using some nonlinear function σ on the pre-activation state z of the same time step. There is also an initial hidden state h_0 which can be initial. Each hidden state at any time step can be used for prediction. This can be done by using a FF which takes the hidden state as the input. In a many-to-one setting, a single prediction is made using the final state h_N . This could be, for instance, predicting the sentiment as being positive or negative after reading a sequence of words. In a many-to-many setting, the hidden state at the time step t can be used to predict the next x_{t+1} . This can be used to train a model to generate music by taking as input a sequence of previous musical notes to generate the next. The RNN can be seen as a FF of arbitrary depth as the sequence of linear and nonlinear operations is repeated along the length of the sequential input, with the difference being that the same parameters W_h, U_h, b_h are used at any time step (or depth).

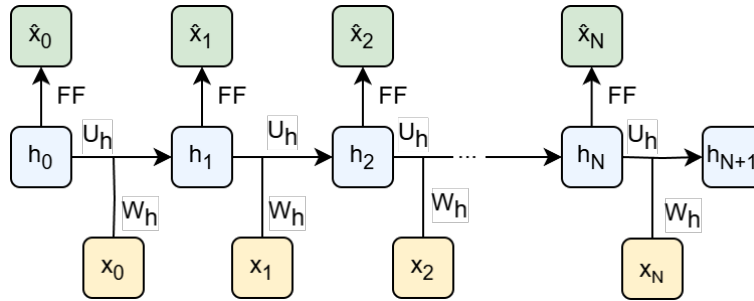


Figure 2.2 RNN in a many-to-many setting.

RNN as a generative model

Figure 2.2 depicts an RNN receiving as input the sequence $\bar{x} = x_0..x_N$ and outputting $\hat{x}_0..\hat{x}_{N+1}$ by feeding each hidden state to a FF. Suppose we wish to train an RNN to predict the next input of the sequence and that each $x_t \in \bar{x}$ belongs to a class $\mathcal{C}$. We would use a FF which outputs a probability distribution over $\mathcal{C}$. The output $\hat{y}_t$ is interpreted as the probability of each class appearing as the next input conditioned on the previous inputs $x_0..x_{t-1}$. Using the input of the next step x_t as target of each output $\hat{x}_t$, we obtain the following cross entropy loss for the sequence $\bar{x}$ which can be used to train the RNN using gradient descent.

$$\sum_{t=0}^N CE(\hat{x}_t, x_t)$$

This RNN can then be used to generate sequences similar to the ones seen during training. This is done by sequentially sampling x_t from the distribution of the output at each time step and using it to generate the output of the next time step $\hat{y}_{t+1}$.

2.1.3 Reinforcement learning

The use of SL techniques requires labeled data, which is not always available. Reinforcement learning is an ML paradigm which deals with the problem of an agent learning the actions to take in some environment by interacting with it in order to maximize some numerical value, or the reward received. Unique challenges arise from this type of problem since the data used to train the agent needs to be collected by the agent whereas in SL, the dataset is assumed to be given. The quality of the data also evolves as the behavior of the agent is being updated, which adds another layer of complexity compared to SL.

Exploration-exploitation dilemma

A challenge in RL known as the exploration-exploitation dilemma involves making the decision to either exploit the current knowledge or to take exploratory actions which might improve the result. We illustrate this with a simple bandit problem. The bandit problem involves taking actions, which is to choose one out of slot machines $\{m_1, m_2\}$ to receive a reward with the respective probabilities of $\{p_1, p_2\}$ to receive a reward of 1 and 0 otherwise, we would like to maximize the expected reward, how do we make the optimal choices without knowing p_1 and p_2 ? Data must be obtained by sampling from the machines, we can then use it to estimate the expected reward for each of the actions. However, discovering the optimal action requires us to sample from suboptimal actions. We must first explore our choices by

gathering samples from both actions before being able to exploit our findings by choosing the one with the highest estimate. A sampling strategy known as ϵ -greedy involves taking the action with the highest average reward with a probability of $(1-\epsilon)$ or else take a random action.

2.1.4 Markov decision process

A (finite) markov decision process (MDP) is a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P})$ where $\mathcal{S}$ is a (finite) set of states, $\mathcal{A}$ is a (finite) set of actions and $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the transition function of the MDP such that $\sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s') = 1$ for all states s and actions a . $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}$ is a function describing the probability of the reward $r \in \mathbb{R}$ received after taking action a at state s , we impose that $\int_{\mathbb{R}} \mathcal{R}(s, a, r) dr = 1$ for any s and a . We interchangeably write $\mathcal{P}(s'|s, a)$ and $\mathcal{R}(r|s, a)$. Suppose an agent sequentially chooses actions in an MDP, we have $t \in \mathbb{N}_0$ which denotes the time step. At state S_t , the agent takes an action A_t and receives a reward with a distribution given by $\mathcal{R}(r_t|S_t, A_t)$, the agent moves to the next state S_{t+1} with a distribution given by $\mathcal{P}(S_{t+1}|S_t, A_t)$. We say the MDP is episodic if there is some terminal state s that cannot be left once entered and $\mathcal{R}(s, a, 0) = 1$ for all actions a . For episodic MDP, we use R_t to denote the discounted sum of rewards obtained after timestep t_0 : $R_t = \sum_{t=t_0}^T \gamma^{t-t_0} r_t$ with T being the length of the episode and $\gamma \in (0, 1]$ being the discount factor. We say the return is undiscounted if $\gamma = 1$. If the MDP is not episodic, we take the limit of $T \rightarrow \infty$ and a discount factor < 1 so the limit exists. In our work, we always assume the MDP is episodic.

Value function

We define a policy $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ as a function that gives the probability of taking an action a at a state s . We impose that $\sum_{a \in \mathcal{A}} \pi(s, a) = 1$ and also write $\pi(a|s)$ to make it clear this is a distribution is over $\mathcal{A}$ for a given $s \in \mathcal{S}$.

Given policy π , we define the value function $V_\pi : \mathcal{S} \rightarrow \mathbb{R}$ that gives the expected return of a state when following the policy π from that state.

$$V_\pi(s) = \mathbb{E}_\pi[R_t | S_t = s]$$

We also define the Q value function which gives the expected return when taking an action and following π subsequently.

$$Q_\pi(s, a) = \mathbb{E}_\pi[R_t | S_t = s, A_t = a]$$

We define the optimal policy π^* as a policy with the following property

$$\pi^*(s) = \max_{\pi} V_{\pi}(s), \forall s \in \mathcal{S}$$

Even though there can be different optimal policies, their value functions and Q value functions are all the same.

2.1.5 Value based methods

By having the function Q^* , which is the Q value function of the optimal policy, we can obtain an optimal policy by assigning non zero probabilities to actions with the highest Q -values [10]. We present an algorithm called Q -learning which uses this principle to learn the optimal policy. In Algorithm 1, a Q -table is initialized. The agent learns from interacting

Algorithm 1 Tabular Q -learning (ϵ – greedy)

Require: $N_{episodes} \in \mathbb{N}, \epsilon > 0$

Initialize Q as $|\mathcal{S}| \times |\mathcal{A}|$ array with 0's everywhere.

for $i = 1, \dots, N_{episodes}$ **do**

Initialize $s \in \mathcal{S}$ as the starting state

while episode is not finished **do**

$r \leftarrow \text{random}([0, 1])$

▷ Randomly sample a double from $[0, 1]$.

if $r > \epsilon$ **then**

$a \leftarrow \text{arg_maximum}(Q[s, ..])$

▷ Choose randomly to break ties.

else

$a \leftarrow \text{arg_random}(Q[s, ..])$

$s', r \leftarrow \text{env.step}(a)$

▷ Take action a , receive reward r and next state s' .

$Q[s, a] \leftarrow r + \gamma \cdot \max(Q[s', ..])$

$s \leftarrow s'$

with an environment over $N_{episodes}$ episodes. The agent uses the ϵ -greedy strategy, and samples $r \in [0, 1]$ to take a random action if $r < \epsilon$ or to take the action with the highest value from the Q -table with the current state. After taking the action, the agent receives a reward r and next state s' which it then uses to update the Q -table with the following formula.

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a') \quad (2.9)$$

Where the leftmost term is the current estimate of the state action pair (s, a) . The rightmost term is the target used to update our estimate by taking it as the sum between the immediate reward and the estimate of the next state s' . Q -learning has the property that the learned Q values are guaranteed to converge towards Q^* if all state action pairs are continuously visited,

this condition is met since we use the ϵ -greedy exploration strategy. We also note that this does not require the transition or reward functions of the MDP, which makes this method a *model free* method. This makes Q-learning a very flexible algorithm since no assumption is made about the environment model, except for the fact that it is an MDP. This method is also *off-policy* because the trajectories are obtained using the ϵ -greedy policy but the actual learned policy sets ϵ to 0, it is also an *online* learning algorithm, since the policy is updated after each action whereas an *offline* learning algorithm requires the episode to finish before updates. This means that Q-learning can be used in a setting where an episode never terminates.

Deep Q-learning

Although Q-learning has convergence guarantees, when the number of states and actions are very large, it becomes intractable to use. This is a serious limitation to the application of this algorithm to many interesting problems, such as playing chess which has around 10^{120} board states or a video game whose state space can be uncountable. Luckily for us, we can use techniques in SL to learn an approximated mapping between states and values, which is known as the regression task. Instead of keeping a table of Q -values, a NN processes the features of a state and outputs the approximated Q -value. The Deep Q-network (DQN) by Mnih et al. 2015 [2] is used to train an agent to learn to play ATARI games which attained a level of play superior to a human expert on 57 games. They also use a replay buffer which keeps in memory the tuple (s, a, s', r) where s , in order to continuously learn from previous interactions. The above algorithm is similar to the tabular Q-learning algorithm we presented. The difference being that the Q -table is replaced with the output of a policy network. Initially, the replay buffer M is initialized along with the policy and target networks. At the state s , the agent takes the action a and receives the reward r and the next state s' . This transition (s, a, s', r) is then stored inside the replay buffer. The networks are then updated by sampling a batch of transitions from the replay buffer. Using Formula 2.9, the policy network is used on the leftmost term and the target network is used on the right most term. The MSE loss between the estimated Q value and the target Q value to update the parameters in the policy network. The target network is then updated with the $\tau \in [0, 1]$ which is a very small value, it is used to help stabilize training.

2.1.6 Policy gradient methods

So far we have introduced the concept of a value function and methods to learn the optimal value function in order to learn the optimal policy by greedily choosing the action with the

Algorithm 2 Deep Q-learning (ϵ - greedy)

Require: $N_{episodes} \in \mathbb{N}, \epsilon > 0$

Initialize M as an empty set (replay buffer)

Initialize $Q_{\pi_{\theta}^{target}}$ the NN with parameters θ

Initialize $Q_{\pi_{\theta'}^{policy}}$ with parameters θ' as a copy of θ

for $i = 1, \dots, N_{episodes}$ **do**

Initialize $s \in \mathcal{S}$ as the starting state

while episode is not finished **do**

$r \leftarrow \text{random}([0, 1])$

▷ Randomly sample a double from $[0, 1]$.

if $r > \epsilon$ **then**

$a \leftarrow \arg_max(Q_{\pi_{\theta}^{policy}}(s))$

▷ Choose randomly to break ties.

else

$a \leftarrow \arg_random(Q_{\pi_{\theta'}^{policy}})$

$s', r \leftarrow \text{env.step}(a)$

▷ Take action a , receive reward r and next state s' .

$M \leftarrow (s, a, s', r)$

$s \leftarrow s'$

update_network()

procedure UPDATE_NETWORK()

Sample batch of tuples (S, A, S', R) from M

$Q_{pred} \leftarrow Q_{\pi_{\theta}^{policy}}(S)$

$Q_{target} \leftarrow r + \gamma \cdot Q_{\pi_{\theta}^{target}}(S')$

$(\nabla_{\theta}(Q_{target} - Q_{pred})^2).mean().backpropagate()$

$\theta_{target} \leftarrow \tau \cdot \theta_{pred} + (1 - \tau) \cdot \theta_{target}$

highest Q value. Another important class of methods focus solely on learning the optimal policy directly without learning a value function. Suppose we have a policy π_θ parameterized by θ , we define the objective function with respect to θ as the expected return starting from the initial state.

$$J(\theta) = \mathbb{E}_{\pi_\theta}[R_0]$$

The policy gradient theorem states that the gradient of J with respect to θ is given by the following expression [10].

$$\nabla_\theta J(\theta) \propto \sum_s \mu_{\pi_\theta}(s) \sum_a q_{\pi_\theta}(s, a) \nabla \pi_\theta(a|s) \quad (2.10)$$

Where $\mu_{\pi_\theta}(s)$ gives the proportion that a state is visited when following the policy π_θ , also known as the stationary distribution. We can introduce the random variable S_t by noticing that the previous expression has a sum over all states weighted by their chance of being visited, we replace it by taking the expectation over the random variable S_t .

$$\sum_s \mu_{\pi_\theta}(s) \sum_a q_{\pi_\theta}(s, a) \nabla \pi_\theta(a|s) = \mathbb{E}_{\pi_\theta}[\sum_a q_{\pi_\theta}(S_t, a) \nabla \pi_\theta(a|S_t)]$$

We introduce A_t in a similar fashion. Since the sum over actions is not weighted, we multiply it by $1 = \pi_\theta(a|S_t)/\pi_\theta(a|S_t)$.

$$= \mathbb{E}_{\pi_\theta} \left[\sum_a \pi_\theta(a|S_t) q_{\pi_\theta}(S_t, a) \frac{\nabla \pi_\theta(a|S_t)}{\pi_\theta(a|S_t)} \right]$$

We interpret $\pi_\theta(a|S_t)$ as the weight for each action and replace the sum over a with the expected value.

$$\begin{aligned} &= \mathbb{E}_{\pi_\theta} \left[q_{\pi_\theta}(S_t, A_t) \cdot \frac{\nabla \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \right] \\ &= \mathbb{E}_{\pi_\theta} \left[G_t \cdot \frac{\nabla \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \right] \\ &= \mathbb{E}_{\pi_\theta} [G_t \cdot \nabla \log(\pi_\theta(A_t|S_t))] \end{aligned} \quad (2.11)$$

REINFORCE algorithm

We now describe REINFORCE, a policy gradient algorithm which uses the aforementioned estimator for the gradient $\nabla_\theta J(\theta)$ to learn an optimal policy.

In Algorithm 3, we estimate the gradient 2.11 randomly sampling trajectories using the policy π_θ which are the sequences of states $s_0..s_T$, actions $a_1..a_T$ and rewards $r_1..r_T$. After

Algorithm 3 REINFORCE

Require: $N_{episodes} \in \mathbb{N}, \epsilon > 0$

Initialize policy network π_θ

for $i = 1, \dots, N_{episodes}$ **do**

Initialize env

$loss \leftarrow run_episode(\pi_\theta, env)$

$\nabla_\theta(loss).backpropagate()$

procedure $run_episode(\pi_\theta, env)$

$\ell \leftarrow 0$

$S, A, R \leftarrow \text{Array}, \text{Array}, \text{Array}$

$s \leftarrow env.s$

$S[0] \leftarrow s$

while episode is not finished **do**

$a \leftarrow \pi_\theta(s)$

$s', r \leftarrow env.step(a)$

$S[\ell - 1], A[\ell], R[\ell] \leftarrow s', a, r$

$\ell \leftarrow \ell + 1$

$s' \leftarrow s$
 $loss \leftarrow 0$

for $t = 0.. \ell - 1$ **do**

$G \leftarrow \sum_{j=t.. \ell} \gamma^{j-t} \cdot R[j]$

$s, a \leftarrow S[t], A[t]$

$loss \leftarrow loss + G \cdot \log(\pi_\theta(s, a))$

return $loss/\ell$

▷ Collect the trajectory of an episode
▷ Randomly sample action from distribution

▷ Computing the loss and updating the network

each episode, we calculate the average gradient of $G_t \cdot \log(\pi_\theta(s_t, a_t))$ and use it to update the policy θ . This algorithm requires the episode to be completed in order to update its policy, which makes it an offline policy. Like the Q-learning algorithm we presented, it does not require the environment model, which makes it a model free algorithm. In this method, we are estimating the expected value in 2.11 by repeatedly sampling it from the distribution and taking the average. This estimation method is known as a Monte Carlo estimation method. This method is known to have a problem of high variance, which can slow down significantly the learning algorithm. In the next section, we discuss some ways to reduce variance for policy gradient methods.

2.2 Constraint programming

Constraint programming is a subfield of AI that aims to find solutions to combinatorial problems that must satisfy constraints, one advantage of CP being the versatility in the kinds of problems that can be expressed as a CP model due to the diversity of constraints it offers. This usually starts with the design of a model for the problem by declaring the variables, their domain which contains the values that the variables are allowed to take and constraints that an assignment of values to variables must satisfy. The model is then given to a CP solver that searches for solutions using a backtrack strategy and by using efficient *filtering* algorithms that identify values from each domain that necessarily violate constraints and remove them from the domain. The removal of a value by a filtering algorithm might allow the other filtering algorithms to make further deductions. This process is repeated until we arrive at some fixpoint where no more values can be filtered. This is the core mechanism used in CP to cut down the search space of solutions. We will now introduce some basic definitions and concepts and motivate them with a simple example. Then, we will see a widely-used filtering algorithm.

2.2.1 Basic definitions

Definition 2.2.1. Variable space is a tuple of variables $\mathcal{X} = \langle x_1, \dots, x_n \rangle$ of cardinality n with domains $dom(x_i) \forall i = 1 \dots n$. By convention, we treat $\mathcal{X}$ as a set and we write $x \in \mathcal{X}$ for a variable belonging to $\mathcal{X}$. We call an **assignment** α of $\mathcal{X}$ a partial function with the domain $\mathcal{X}$ such that $\alpha(x_i) \in dom(x_i)$, if $x_i \in \mathcal{X}$ is defined. We also write $\alpha[x_i]$ to denote the assignment of α for the variable x_i . We call $vars(\alpha)$ the set of variables α assigns a value to. We say α_1 is consistent with α_2 iff $\alpha_1[x] = \alpha_2[x] \forall x \in vars(\alpha_1) \cap vars(\alpha_2)$ where $vars(\alpha)$ gives the set of variables such that $\alpha[x]$ is defined $\forall x \in vars(\alpha)$. We call $\mathcal{A}(\mathcal{X})$ the set of all assignments of $\mathcal{X}$.

Definition 2.2.2. We call the tuple $\mathcal{M} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ a **constraint satisfaction problem** (CSP) where $\mathcal{X}$ is a variable space, $\mathcal{D}$ is a finite set of domains $dom(x)$ for each variable $x \in \mathcal{X}$ and $\mathcal{C}$ is a set of constraints. Let $\mathcal{X} = \langle x_1, \dots, x_n \rangle$ be the variable space, a constraint $c \in \mathcal{C}$ on a set of variables $X \subseteq \mathcal{X}$ is a subset of *total* assignments $\alpha \in \mathcal{A}(X)$, we mean total in the sense that every variable is assigned a value. We call $|X|$ the arity of c . We say an assignment α satisfies c if $\exists \alpha_c \in c$ such that α is consistent with α_c and $\alpha[x] \in dom(x) \forall x \in vars(\alpha)$. The set of solutions $Sol(\mathcal{M})$ is a set of total assignments that satisfy all constraints. We also interchangeably used the term *CP model* with CSP.

Example 2.2.1. Let $\mathcal{M}$ be a CSP with three variables $\mathcal{X} = \langle x_1, x_2, x_3 \rangle$ with domains $dom(x_1) = \{1, 2\}$, $dom(x_2) = \{1, 2\}$, $dom(x_3) = \{1, 2, 3\}$. The constraints are $all\text{-}different(x_1, x_2)$, $all\text{-}different(x_1, x_3)$, $all\text{-}different(x_2, x_3)$ which contain assignments such that all variables in the arguments must take on different values. For instance, $c_1 = all\text{-}different(x_1, x_3) = \{(1, 2), (1, 3), (2, 1), (2, 3)\}$. The set of solutions is $Sol(\mathcal{M}) = \{(1, 2, 3), (2, 1, 3)\}$.

In Example 2.2.1, it is trivial to realize at first glance that variable x_3 must take on value 3 in any solution, therefore there exists some values in $dom(x_3)$ that are *inconsistent* with respect to the constraints. When all values in all domains are consistent when considering any individual constraint (think of a value as a partial assignment), we say the model is *locally consistent*. Formally, a CSP is locally consistent if for any constraint c and for any variable x , $\forall v \in dom(x)$ there exists an $\alpha \in c$ such that $\alpha[x] = v$. However, just because the CSP is locally consistent does not mean all values lead to a solution that satisfies all constraints: we call the CSP *globally consistent* in the latter case. Notice in example 2.2.1 that each constraint is locally consistent. It is easy to see however that any solution α must assign $\alpha[x_3] = 3$. For this CSP to be globally consistent, only the value of 3 should be in domain of x_3 . The CSP in Example 2.2.1 is not globally consistent, since the values 1 and 2 in $dom(x_3)$ do not occur in any solution to $\mathcal{M}$. We now present an important constraint used in chapter 5.

Definition 2.2.3. A **cost-regular** constraint on the variables $X = \langle x_1 \dots x_n, c \rangle$ requires a $S \times L$ transition matrix called T , a $S \times L$ cost matrix called C , an initial state $s_I \in \{1..S\}$ and a set of accepting states $S_{accept} \subseteq \{1..S\}$. Suppose we define s_0 as s_I , we have the recursive definition for the next state $s_{t+1} = T[s_t, \alpha[x_t]]$ for $t \in \{1..n+1\}$. We have that $cost\text{-}regular(x_1, \dots, x_n, c, T, C, S, s_I, S_{accept}) = \{\alpha | \alpha \in \mathcal{A}(X), s_{n+1} \in S_{accept}, \alpha[c] = \sum_{t=1}^n C[s_{t-1}, \alpha[x_t]]\}$.

The cost-regular constraint defined in 2.2.3 constrains a sequence of variables to be accepted by a deterministic finite automaton. This automaton consists of a set of states, an initial

state and a set of accepting states. The sequence of *action* variables must lead a sequence of transitions from the initial state to any accepting state. There is also a *cost* variable which represents the cost of the sequence of actions.

2.2.2 Local consistency algorithm

In a CSP, removing inconsistent values from the domain is done automatically through the *filtering* algorithms for each constraint that removes inconsistent values for their respective constraints. We now present the a constraint propagation algorithm. This algorithm begins

Algorithm 4 constraint propagation algorithm

Require: CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$

$worklist \leftarrow \mathcal{C}$

$reduced \leftarrow True$

while $|worklist| > 0$ **do**

 select $c \in worklist$

$worklist \leftarrow worklist \setminus \{c\}$

$reduced \leftarrow REDUCE(c)$

if $reduced$ **then**

for $c' \in \{\hat{c} \in \mathcal{C} \mid vars(\hat{c}) \cap vars(c) \neq \emptyset\}$ **do**

$worklist \leftarrow worklist \cup \{c'\}$

procedure REDUCE(c)

$X \leftarrow vars(c)$

$reduced \leftarrow False$

for $x \in X$ **do**

for $v_x \in dom(x)$ **do**

 Build $s = \{\alpha \mid \alpha \in c \text{ and } \alpha[x] = v_x\}$

if $|s| = 0$ **then**

$reduce \leftarrow True$

$dom(x) \leftarrow dom(x) \setminus v_x$

if $|dom(x)| = 0$ **then**

 throw *inconsistency* error

 return $reduced$

by initializing *worklist*, which contains all constraints of the CSP — they are considered to be *active*. It then loops until *worklist* is empty by selecting active constraints and using the *REDUCE* function to filter out inconsistent values in each domain for each of the variables in the scope of the constraint. If any value was filtered, every other constraint with a scope with a non empty intersection with the scope of that constraint must be re-added to *worklist*, if they aren't already in it. This process is repeated until *worklist* is empty or an *inconsistency* error has been detected, which means there are no solutions in the CSP. Because each call of

the *REDUCE* function only considers the local consistency of the given constraint c_i , it can not take into consideration the consistency with respect to all other constraints. However, using *REDUCE* on another constraint c_j sharing a variable and successfully reducing a value might allow c_i to remove more values on a subsequent call of *REDUCE*, this is why we put back constraints in line 8. At the end of the algorithm, we arrive at a fix point where calling *REDUCE* can no longer filter out any values from any domains and the resulting CSP is therefore arc consistent.

2.2.3 Backtrack solution searching

Because local consistency does not allow us to achieve global consistency, there can be assignments of values to variables that do not lead to a solution. To search for a solution, a CP solver can use a depth-first search with backtracking whenever an inconsistency has been found. After making a branching decision of assigning a value to a variable, GAC is used to filter out values before making the next branching decision which has the effect of pruning entire subtrees of the search space where no solutions exist. It also allows us to realize quickly if our current branch will not lead us to any solution when GAC returns an inconsistency in which case we backtrack to a previous state. At each branching decision, the choices for the variable as well as the value are of important consideration as they can have an impact on the chance of successfully finding a solution. A simple heuristic for the choice of variable is the size of the domain. We can also choose the variable with the smallest domain with the belief that the chosen variable is more likely to fail and reduce the depth of the search branch, this is also known as a fail first strategy.

2.2.4 Approximating the solution density

In this section, we present a value branching heuristic based on the solution density of the CP model. We give the definition below.

Definition 2.2.4. Consider the CP model $\mathcal{M} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ such that $\mathcal{X} = \langle x_1, \dots, x_n \rangle$. Let $P : \text{Sol}(\mathcal{M}) \subseteq \mathcal{A}(\mathcal{X}) \rightarrow [0, 1]$ be a uniform probability distribution over all solutions. We define the marginal distribution of $\mathcal{M}$ for variable $x_i \in \mathcal{X}$ as $\theta_{x_i} : \text{dom}(x_i) \rightarrow [0, 1]$. It is given as the following expression.

$$\theta_{x_i}(v) = \sum_{\alpha \in \text{Sol}(\mathcal{M}) \mid \alpha[x_i]=v} P(\alpha)$$

Since P is uniform, we can obtain the marginal probability with the following expression.

Let $\mathbb{I}(\alpha)$ be the indicator function of whether $\alpha \in \text{Sol}(\mathcal{M})$.

$$\theta_{x_i}(v) = \frac{|\{\alpha \mid \alpha \in \text{Sol}(\mathcal{M}) \wedge \alpha[x_i] = v\}|}{|\text{Sol}(\mathcal{M})|} = \sum_{x_1 \in \text{dom}(x_1)} \dots \sum_{x_i=v} \dots \sum_{x_n \in \text{dom}(x_n)} \mathbb{I}(\alpha(x_1 \dots x_n)) / |\text{Sol}(\mathcal{M})|$$

It can be verified that θ_{x_i} is a probability distribution since $\sum_{v \in \text{dom}(x_i)} \theta_{x_i}(v) = 1$. This heuristic is a probability distribution of all solutions over all value assignments for a given variable. To use this heuristic, we branch on the value with the highest probability distribution, the intuition being that it has a higher likelihood of finding a solution. To obtain the marginal probability, we must calculate the nested sum. The naive way of calculating this sum would be intractable as the number of operations grows exponentially with respect to the number of variables. This sum can be computed more efficiently by exploiting the factorization of the function as a product of functions on subsets of the variables. For the indicator function of the CP model we have the following factorization,

$$\mathbb{I}(\alpha) = \prod_{c \in \mathcal{C}} \mathbb{I}_c(\alpha[\text{vars}(c)]) \quad (2.12)$$

where $\mathbb{I}_c : \text{vars}(c) \rightarrow \{0, 1\}$ is an indicator function of whether the assignment α of variables in $\text{vars}(c)$ satisfies constraint c that is, if $\alpha[\text{vars}(c)] \in c$. Belief propagation (BP) [12] is an efficient algorithm which can be used to approximate marginal distributions in the form of nested sums like the one we presented. In the case where the factorization has no "loops", the approximation is exact, otherwise it is not. BP has been used to compute approximate $\hat{\theta}_{x_i}(v)$ in the context of CP in the work of Pesant et al. (2020) [1].

Belief propagation for CP

We describe the BP algorithm [12] to calculate the nested sum in Equation 2.12. We first introduce the concept of a factor graph which is a graphical representation for a function written as a product of factors.

Definition 2.2.5. Consider the function $f(X) = \prod_{i=1}^m f_i(X_i)$ where $X_i \subseteq X = \{x_1, \dots, x_n\}$, its **factor graph** is an undirected graph with a **variable node** for each variable x_j as well as a **factor node** for each of the m factors. We have that each factor node $f_i(X_i)$ has an edge with a variable node x_j iff $x_j \in X_i$.

Using the factorization in Equation 2.12, the variable nodes are the decision variables and the factor nodes are the constraints imposed on those variables. Consider the function defined as above and its factor graph, we define two types of *messages* sent from variables to constraints

and from constraints to variables.

$$\mu_{x_j \rightarrow c_i}(v) = \prod_{c \in ne(x_j) \setminus \{c_i\}} \mu_{c \rightarrow x_j}(v) \quad (2.13)$$

The message $\mu_{x_j \rightarrow c_i}(v)$ sent by x_j to c_i is the product of messages from constraints in the neighborhood of x_i excluding c_i , which is defined as follows:

$$\mu_{c_i \rightarrow x_j}(v) = \sum_{X_i: X_i[x_j]=v} f_i(X_i) \prod_{x \in ne(c_i) \setminus \{x_j\}} \mu_{x \rightarrow c_i}(v) \quad (2.14)$$

We provide some intuitions on how to interpret these two definitions using the following example. We then provide the more general BP algorithm.

Example 2.2.2. Consider variables $\mathcal{X} = \langle x_1, x_2, x_3, x_4, x_5 \rangle$ with domains $dom(x_i) = \{1..D\}$, $\forall i = 1..5$, and the following constraints: $c_2(x_2, x_3, x_4)$, $c_1(x_1, x_2)$, $c_3(x_3, x_5)$. Suppose we want to calculate the value of v in the marginal distribution θ_{x_2} . The summation would be the following,

$$1/M \sum_{x_1} \sum_{x_3, x_4} \sum_{x_5} \mathbb{I}_{c_1}(x_1, v) \mathbb{I}_{c_2}(v, x_3, x_4) \mathbb{I}_{c_3}(x_3, x_5)$$

where M is the normalization factor so that the resulting function is a probability distribution. This sum can be immediately rearranged so it can be more efficiently calculated:

$$1/M \sum_{x_1} \mathbb{I}_{c_1}(x_1, v) \sum_{x_3, x_4} \mathbb{I}_{c_2}(v, x_3, x_4) \sum_{x_5} \mathbb{I}_{c_3}(x_3, x_5).$$

We notice that since the factors to the left and right of x_2 do not share variables as x_2 is fixed to v , we can rewrite this sum as the product of two smaller sums which can be obtained independently from one another:

$$1/M \left(\sum_{x_1} \mathbb{I}_{c_1}(x_1, v) \right) \left(\sum_{x_3, x_4} \mathbb{I}_{c_2}(v, x_3, x_4) \sum_{x_5} \mathbb{I}_{c_3}(x_3, x_5) \right). \quad (2.15)$$

We can now evaluate the original nested sum as a product of two partial sums which can be individually evaluated. Consider the following expression which is the product of the

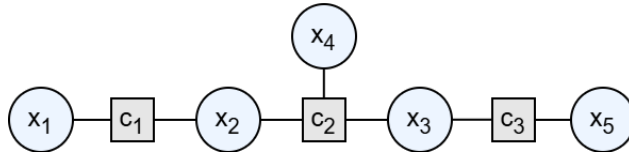


Figure 2.3 Factor graph for the CP model

messages variable x_2 receives from its neighboring constraints:

$$1/M \cdot \mu_{c_1 \rightarrow x_2}(v) \cdot \mu_{c_2 \rightarrow x_2}(v).$$

We expand the constraint messages using Definition 2.14:

$$1/M \sum_{x_1} \mathbb{I}_{c_1}(x_1, v) \mu_{x_1 \rightarrow c_1}(v) \cdot \sum_{x_3, x_4} \mathbb{I}_{c_2}(v, x_3, x_4) \mu_{x_3 \rightarrow c_2}(v).$$

Expanding this expression further and replacing the variable messages with 1 for leaf variables, we obtain the original expression 2.15.

In the previous example we saw that the marginal probability of a variable can be obtained as the product of the messages it receives from neighboring constraints. We can thus efficiently obtain the marginal probability of all variables by obtaining the $\mu_{c \rightarrow x}$ messages for all constraint-variable pairs. However, in order for a constraint to send a message to a variable, it must first obtain the incoming messages from every *other* variable, the same applies for variable to constraint messages. One method to compute messages would be to start from leaf variables and propagate the messages through the factor graph. In practice, BP can calculate messages in an arbitrary order in an iterative fashion. At each iteration, constraints send a message $\mu_{c \rightarrow x}^{t+1}$ using the incoming messages $\mu_{x \rightarrow c}^t$ of the current iteration. At $t = 0$, the incoming messages are assumed to be 1. The number of iterations required for the messages to be exact (so that the marginal distributions are exact) depends on the distance between the variable nodes so that information from one node can travel to the other.

So far we described an exact computation for the marginal distribution of a factor graph without cycles. If a cycle were to exist, the separation of the sum in Equation 2.15 into a product of two smaller sums would not be possible. It turns out that the marginal distributions computed with BP for a factor graph with loops is still approximately good enough, despite the errors induced by violating the assumption in Equation 2.15.

MiniCPBP implementation of BP for CP

In this section, we describe the implementation of BP in the CP solver we use called MiniCPBP. Figure 2.4 describes the BP implementation in MiniCPBP. Each constraint is visited and receives messages from its variables. Using these messages, it then computes and sends messages to those variables. This process is repeated for a number of iterations before stopping. Recall that the constraint to variable messages $\mu_{c \rightarrow x}$ are expressed as a sum in Equation 2.14, which can be interpreted as counting solutions for that constraint weighted

Algorithm 5: Solver.beliefPropa()

```

foreach variable  $x$  do
|   reset  $x$ .marginal;
foreach constraint  $c$  do
|   reset  $c$ .localBelief;
repeat
|   foreach constraint  $c$  do
|   |    $c$ .receiveMessages();
|   foreach variable  $x$  do
|   |   reset  $x$ .marginal;
|   foreach constraint  $c$  do
|   |    $c$ .sendMessages();
|   foreach variable  $x$  do
|   |   normalize  $x$ .marginal;
until stopping condition;

```

Figure 2.4 BP algorithm implementation in MiniCPBP [1]

by incoming messages. If a constraint were to have a large arity, the sum would be difficult to compute naively. For some constraints, it is possible to efficiently perform a weighted counting of solutions. In MiniCPBP, specialized algorithms for each constraint are used to perform this weighted counting. It can be done exactly for the cost-regular constraint but would be very costly for the all-different constraint so this weighted sum is instead approximated. Consider the following example taken from Pesant et al. (2019) [1].

Example 2.2.3. Variables are a, b, c, d each with domains $\{1..4\}$. The constraints are all-different(a, b, c), $a + b + c + d = 7$, $c \leq d$.

We have the true marginal distribution for this CSP in figure 2.5.

Using MiniCPBP's implementation of BP, we obtain the approximated marginals after 10

	1	2	3	4
θ_a	0	0.5	0.5	0
θ_b	0	0.5	0.5	0
θ_c	1	0	0	0
θ_d	1	0	0	0

Figure 2.5 True marginals for the example CSP [1].

	1	2	3	4
θ_a	0.01	0.52	0.46	0.01
θ_b	0.01	0.52	0.46	0.01
θ_c	0.98	0.02	0.00	0.00
θ_d	0.90	0.10	0.00	0.00

Figure 2.6 Approximated marginals using MiniCPBP [1].

iterations of BP in figure 2.6.

2.3 AI planning problems

We now consider AI planning problems which involves choosing a sequence of actions to arrive at a goal state starting from an initial state. Unlike CP problems, this class of problems has the additional difficulty of not knowing how many actions are required to solve the problem. One classic planning problem called Blocks World (BW) is a block stacking problem where an initial configuration of the placement of the blocks is given along with a goal configuration. The problem is to determine a sequence of actions which involve placing one block on top of another or placing it on the ground in order to reach the goal configuration.

Example 2.3.1. In Figure 2.7, we have a simple BW problem with block A initially placed on top of block B , the goal is to have block B be placed on top of A by taking $move(x, y)$ actions which places the block x on top of y which could either be the ground or another block. It is easy to see that the problem can be solved by $move(B, ground)$, $move(A, B)$.

AI-planning problems are typically defined with formalisms so that they may be processed by a planning solver. In our work, we use the formalism called the SAS+ [13]. Though many formalisms exist for planning problems such as PDDL [14] which is more widely used

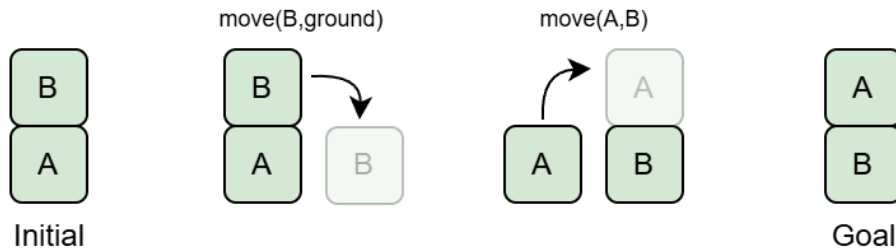


Figure 2.7 Blocks World problem example

in AI planning competitions, we use SAS+ since it allows us to view planning problems as automata, which we can readily model using CP.

Definition 2.3.1. We call a **SAS+ model** of an AI planning problem the tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_I, s_G \rangle$ such that

- $\mathcal{V}$ is the variable space.
- $\mathcal{O}$ is a finite set of operators such that for each operator $o \in \mathcal{O}$ we have $pre(o)$, $eff(o)$, and $cost(o)$ which are respectively the preconditions and effects, which are partial assignments, and the real cost of o .
- s_I is the initial state and s_G is the goal state.

For the BW problem presented in figure 2.7, we present its SAS+ model in figure 2.8. Note however that the model for this problem is not unique and the choice of the model has consequences on the resulting CP model

We now introduce the concept of transition systems which serves as a bridge between AI-planning and CP. It allows us to view a SAS+ model as an automaton that can be readily expressed as a constraint in CP.

Definition 2.3.2. We call **transition system** a tuple $\Theta = (S, L, c, T, S_I, S_G)$ such that S is a finite set of states, L is a finite set of labels, c is a cost function $L \rightarrow \mathbb{R}$. $S_I, S_G \subseteq S$ are the set of accepting and rejection states, respectively $T \subseteq S \times L \times S$ is a set of transitions, we also write $s \xrightarrow{\ell} s' \in T$ to denote the transition from the state s to s' after using the label ℓ (think of labels as actions).

For a planning problem Π , we have the **induced transition system** of Π which we call $\Theta(\Pi)$ by taking the assignments of $\mathcal{V}$ as states. The operators and their costs are taken directly as the labels and costs of labels. A transition $s \xrightarrow{\ell} t$ is in T if s is consistent with $pre(\ell)$ and t is consistent $eff(\ell)$ and $s[v] = t[v]$ for all $v \notin eff(\ell)$, $S_I = \{s_I\}$ and $S_G = \{s_G\}$. Figure 2.9 depicts the induced transition system of the previous SAS+ model. At each state s , if an operator ℓ is not represented by a named edge, it either is not legal ($s \xrightarrow{\ell} s' \notin T$) or is represented by the dotted edge. The blue box represents s_I and the green box represents s_G . Unless specified otherwise, each transition has a cost of 1.

The size of the automaton represented by the induced transition system might be too large, which makes computations difficult when expressed in CP. We can instead express this automaton as the direct product of *atomic factors* which are the decomposition of the induced transition system into smaller transition systems, one for each variable.

$\mathcal{V} =$	$\langle x_A, x_B \rangle :$	$dom(x_A) = \{air, B\}$
		$dom(x_B) = \{air, A\}$
$\mathcal{O} = \{$	$move(A, ground):$	$pre(move(A, ground)) = \{x_A \rightarrow air\}$
		$eff(move(A, ground)) = \{\}$
	$move(A, B):$	$pre(move(A, B)) = \{x_A \rightarrow air, x_B \rightarrow air\}$
		$eff(move(A, B)) = \{x_B \rightarrow A\}$
	$move(B, ground):$	$pre(move(B, ground)) = \{x_B \rightarrow air\}$
		$eff(move(B, ground)) = \{\}$
	$move(B, A):$	$pre(move(B, A)) = \{x_A \rightarrow air, x_B \rightarrow air\}$
		$eff(move(B, A)) = \{x_A \rightarrow B\}$
$s_I =$	$\{x_A \rightarrow B, x_B \rightarrow air\}$	
$s_G =$	$\{x_B \rightarrow A, x_A \rightarrow air\}$	

Figure 2.8 SAS+ model for a Blocks World problem of two blocks

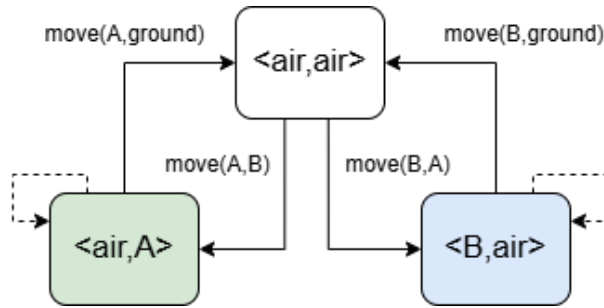


Figure 2.9 Induced transition system of a Blocks World problem

Definition 2.3.3. For a planning problem $\Pi = \langle \mathcal{V}, \mathcal{O}, s_I, s_G \rangle$, we call **atomic factor** for the variable $v \in \mathcal{V}$ the transition system $\Theta^v = \langle \text{dom}(v), \mathcal{O}, c, T^v, S_I^v, S_G^v \rangle$. $T = \{d \xrightarrow{\ell} d' \mid (v \notin \text{vars}(\text{pre}(\ell)) \vee \text{pre}(\ell)[v] = d) \wedge ((v \notin \text{vars}(\text{eff}(\ell)) \wedge d = d') \vee (\text{eff}(\ell) = d))\}$

Definition 2.3.4. A **factorized transition system** (FTS) is the tuple $(\Theta^1.. \Theta^n)$ of transition systems, each defined by $\Theta^i = \langle S^i, L, c, T^i, S_I^i, S_G^i \rangle$ sharing the same labels and transition costs. The induced FTS of a SAS+ model is the tuple $(\Theta^{v_1}.. \Theta^{v_n})$ containing the atomic factors of each variable.

We consider the induced FTS (see figure 2.10) of the previous SAS+ model. We get two transition systems of two states each. It is easy to see that the synchronized behavior of the two atomic factors is identical to the induced transition system. Note also that each atomic factor are smaller in the sense that they have less states.

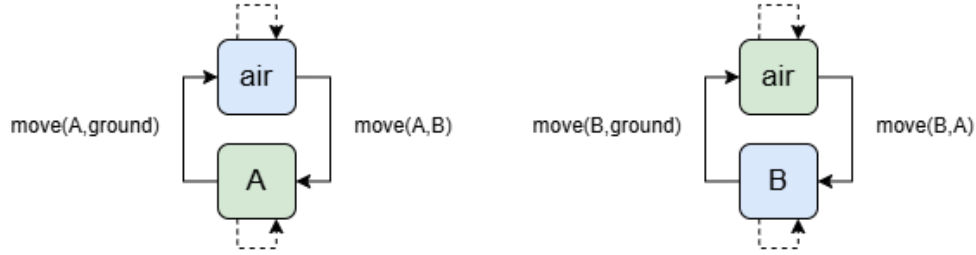


Figure 2.10 Induced FTS $\langle \Theta^A, \Theta^B \rangle$ for Blocks World example of two blocks

CHAPTER 3 LITTERATURE REVIEW

Sample efficiency in RL refers to the ability of a learning algorithm to perform certain tasks without requiring large amounts of data. Although the question of how to fully characterize the limitations of sample efficiency remains an open question, methods have been established in the literature to evaluate and compare the sample efficiency of algorithms using a combination of theoretical and empirical results. We begin our literature review on the theoretical tools developed to measure sample efficiency in the tabular RL setting where the number of states is relatively small. We follow this with innovations to address the sample efficiency challenges in the DRL setting posed by the large number of states and the line of research building on the influential DQN [2] algorithm. We then limit our discussion to the method of *reward shaping*, a method which improves efficiency by modifying the reward function.

3.0.1 Provably efficient RL in the tabular setting

Regret bound analysis is one of the most prevalent tools of analyzing the sample efficiency of algorithms in RL. The regret $\mathcal{R}$ of an RL algorithm is the difference between the value function of the initial state using the policy learned by the algorithm and the optimal value of the initial state. In the episodic setting of K episodes, this can be defined as the following:

$$\mathcal{R} = \sum_{k=1}^K (V^{\pi^*}(s_1^k) - V^{\pi^k}(s_1^k))$$

where π^k is the policy used at episode k and V^π is the value function following policy π . The starting states s_1^k are picked by an adversary which seeks to maximize the regret for that episode $V^{\pi^*}(s_1^k) - V^{\pi^k}(s_1^k)$. Intuitively, a sample-efficient RL algorithm which is able to quickly learn good quality policies would reduce regret more quickly under the same number of episodes. Upper bound for the expected value of $\mathcal{R}$ for an algorithm can therefore serve as a guarantee on the algorithm's performance and lower bound on the expected value of $\mathcal{R}$ can indicate the tightness of the upper bound. Regret bound analysis is extensively used in the literature for bandit problems, which are essentially MDPs with a single state. For many variations of the bandit problem, optimal regret bounds are established [15] and most of them can be obtained by using the upper confidence bound (UCB) exploration strategy. This method, which follows the optimism-in-the-face-of-uncertainty principle, involves estimating the upper bound on the return of each action and choosing the action with the highest upper bound. Much of the formalism and analysis techniques to obtain regret bounds used in

the bandit literature were adapted in works for model-based tabular RL [16–18] to use the upper confidence bound method as an exploration strategy to achieve asymptotic optimal regret bound. For model-free methods, such an analysis is more sparse in the literature and usually comes with more restrictive assumptions such as MDPs having finite horizons [19]. A theorem by Dominguez et al. [20] derived a lower bound $\Omega(\mathbb{E}[\mathcal{R}])$ for episodic MDPs using *any* algorithm. The optimality of a particular algorithm can be proven by matching the upper bound of the algorithm with the lower bound of episodic MDPs. Azar et al. [17] provide a model-based algorithm that matches this bound *asymptotically*, meaning the algorithm has optimal regret after obtaining a large number of samples. This incurs, however, a huge burn-in cost (when $T > S^3 A^4 H^6$) where T is the number of samples, S is the number of states, A is the number of actions and H is the length of an episode. Jin et al. [19] provide a model-free algorithm which is near-optimal by a factor of $\sqrt{H}$, but without the burn-in cost.

3.0.2 Sample efficiency challenges in DRL and recent trends

In the DRL setting, sample efficiency of an algorithm is more often evaluated empirically with experiments on how quickly the agent learns and theoretical analysis [21] are limited to linear function approximators which lack expressivity. Empirical evaluation of sample efficiency can be done by recording the performance of the agent per number of interactions with the environment, and the final performance of the agent after training on a budget of a given number of interactions. Yu et al. (2018) [22] identify several aspects of DRL where sample efficiency challenges arise and recent advances to address them. We give a brief summary on aspects we selected to get a general grasp on the body of research.

1. **Exploration:** Successful application of RL requires clever exploration strategies to deviate from the current policy to find better ones. Common exploration strategies like ϵ -greedy and Gibbs sampling encourage exploration by adding noise to the action space. This can result in inefficiencies due to the fact that in the space of parameters, the latent policy could potentially be far away from the current parameters. Plappert et al. (2017) [23] showed adding noise in the parameter space can be more efficient and Fortunato et al. (2017) [24] gives an intermediate approach. These methods, however, rely on "blind" searches and can repeat the same paths. Curiosity driven exploration encourages visiting states that are less frequently visited [25] by recording the number of visitations. To make this viable in the DRL setting with large numbers of states, Pathak et al. (2017) [26] proposed the Intrinsic Curiosity Module which predicts whether a state has been visited.

2. **Environment modeling:** Model-based methods have seen success where the environment model can be efficiently constructed, which is challenging in the DRL setting. One trending idea is to extract useful information from the learned model and make use of it while not being fully dependent on it. Tamar et al. (2016) [27] proposed the Value Iteration Network, a neural network which uses a model-based algorithm to learn to plan on a learned model, this plan is then used as augmented features for the policy network. The Imagination-Augmented agents proposed by Weber et al. Pong et al. (2018) [28] proposes to learn a Q-function to predict a distance towards a goal, this Q-function can then be used to do reward shaping. The environment can also be manually created to generate samples cheaply in lieu of more costly real life samples (e.g. simulators for aircraft or robot design).
3. **Abstraction:** Yu et al. (2018) [22] argue that abstraction is the core aspect of the sample efficiency problem in RL, where the RL acts in an abstraction of the state-space with lower dimensions. Hierarchical learning is a branch of RL which deals with the problem of learning abstraction of actions. The agent can learn with *options*, which are a sequence of actions defined by an entrance condition and exit condition [29]. In Hierarchical of Abstract Machines, the agent learns sub policies predefined as automata [30]. Recent work have focused on learning the options rather than predefining them [31]. Neural symbolic architectures are another approach to abstraction. These architectures can be seen as computers with differentiable parts. Graves et al. (2016) [32] proposed the Memory Network which is a neural architecture with the ability to read and write in memory cells. The neural logic machine (NLM) proposed by Dong et al. (2018) [33] learns logic reasoning. It takes a set of objects and performs a sequence of pre-defined *meta-rules* in order to generate at each step predicates of increasing complexity and the final layer of predicates is then used to make predictions using a FF. The internal structure of the NLM is defined by its breadth, meaning the maximum arity of predicates, and depth which is the length of the sequence of *meta-rules* applied. The advantage of this type of model is its generalization capabilities thanks to its ability to learn logical rules from smaller instances.

Improvements on the DQN

Since the publication of the original DQN paper [2], several innovations have been made to improve its efficiency. The prioritized replay buffer [34] is an improvement on the buffer used in the original [2] DQN which prioritizes transitions that have the highest discrepancy with the current value network. Double Q-learning which is an algorithm that addresses the max-

imization bias [10] of Q-learning has been adapted for the DQN [35]. Distributional RL [36] aims at modeling the distribution of the reward received by the agent rather than the expectation of it. Along with other innovations, the Rainbow [37] algorithm is an implementation of the DQN with these innovations which could attain human level performance on average over 57 different Atari games after training for 17 million frames compared to the original DQN which required 200 million frames. Another strategy for increasing sample efficiency is to increase the *replay ratio* [38] which is defined as the number of updates to the network per interaction with the environment, which can be naturally done for any off-policy method such as DQN since the same data can be reused to update the network.

3.1 Variance reduction for policy gradient methods

Large variance in the estimation of $\nabla_{\theta} J(\theta)$ in the REINFORCE algorithm can drastically slow down learning. The benefit of reducing this variance was noticed in the original REINFORCE paper [39]. Williams et al. (1992) proposed to subtract a *baseline* b from the return G_t in Equation 2.8. For the gradient estimation to remain unbiased, they choose b to be a constant. Since then, an *adaptive* baseline that depends on the current state has been introduced to reduce variance further. This method can be viewed as a *control variate* method. It is used to reduce the variance in the Monte Carlo estimation for some random variable X by subtracting from it a random variable Y , the control variate, such that $\mathbb{E}[Y] = 0$ so that $\mathbb{E}[X - Y] = X$. It turns out that X and Y must be highly correlated for $X - Y$ to have a lower variance. The problem then is to find a suitable control variate Y for the policy gradient. Consider the following identity.

$$\sum_a b(s) \nabla \pi_{\theta}(a|s) = b(s) \nabla \sum_a \pi_{\theta}(a|s) = 0 \quad (3.1)$$

Using the same technique from 2.8, we replace the sum over a with the following expected value.

$$\mathbb{E}_{\pi_{\theta}}[b(S_t) \cdot \nabla \log(\pi_{\theta}(A_t|S_t))] = 0 \quad (3.2)$$

We can subtract it from the expected value of the policy gradient and obtain the following. Since the expected value of the previous expression is 0, the expected value resulting from the subtraction is unchanged.

$$\begin{aligned} & \mathbb{E}_{\pi_{\theta}}[G_t \cdot \nabla \log(\pi_{\theta}(A_t|S_t))] - \mathbb{E}_{\pi_{\theta}}[b(S_t) \cdot \nabla \log(\pi_{\theta}(A_t|S_t))] \\ &= \mathbb{E}_{\pi_{\theta}}[(G_t - b(S_t)) \cdot \nabla \log(\pi_{\theta}(A_t|S_t))] \end{aligned} \quad (3.3)$$

With this new estimator for the policy gradient, we only need to modify Algorithm 3 by subtracting from each return G_t by the baseline $b(S_t)$. Although the expected value in Equation 3.3 is unchanged, its variance is modified. It has been proposed [10] to use the value function as a baseline. Since the value function is $\mathbb{E}[G_t]$ in Equation 2.11, it should be highly correlated. It can be shown however that the value function is not the optimal baseline [40]. More recently, an action dependent baseline has been proposed by Liu et al. (2017) [41] which is able to shave off even more variance.

3.2 The problem of reward shaping

Sparse rewards are rewards that are rarely positive, for example, a function which is 1 when the task is completed with success and 0 otherwise. Sparse rewards are easy to specify but make training inefficient. *Reward shaping* is the modification of the reward function in order to shape the behaviour of the learning agent [42]. This is a way to inject heuristic knowledge about the environment model which potentially makes the RL task easier. Rewards can either be handcrafted by human experts [43–45] or created in an automated fashion through learning [46, 47]. Naively modifying the reward function, however, can have unintended consequences [45, 48] as the optimal policies using the modified reward function can be undesirable when evaluated using the original. For example, in the work of Popov et al., 2017 [45] where an agent is taught to stack a red block on top of a blue block, one unexpected behaviour encountered by the authors when designing the reward function is that the agent learned to flip the red block instead of stacking it on top of the blue block. Since the agent is rewarded based on the height of the bottom face of the red block, this undesired behaviour is rewarded. Another example is training an agent to play the game *CoastRunners* at OpenAI [48] where the intended goal is for the player to finish the boat race as quickly as possible and *boost* items are laid out in the map to provide speed ups. They found that since the boost items can respawn and the agents are rewarded for acquiring them (through reward shaping), their agent learned to get a high score by repeatedly taking boost items as they respawn instead of finishing the race. Fortunately there exists a reward shaping method called potential-based rewards shaping (PBRs) [49] which preserves optimal policies of the original MDP. For a given MDP, we call $\phi : \mathcal{S} \rightarrow \mathbb{R}$ a potential function which assigns to each state a real value. Consider the following modification on the reward function,

$$\mathcal{R}'(s, a, s', r) = \mathcal{R}(s, a, s', r + \phi(s') - \phi(s)),$$

where $\mathcal{R}$ and $\mathcal{R}'$ are the original and resulting reward functions, respectively. It has been shown that the resulting MDP retains its optimal policies [49], assuming the MDP is not

episodic. Eck et al. (2016) [50] showed, however, that PBRS adds bias in the finite horizon MDP setting. Beyond guaranteeing policy invariance, theoretical analysis has also shown that PBRS makes the resulting MDP easier to learn. To understand what makes reward shaping successful, Laud et al. (2003) [51] proposed the concept of *reward horizon* which can be characterized as the minimum number of steps taken by an agent before receiving meaningful feedback. One easily sees that sparse rewards would have high reward horizons. They showed, using Q-learning, that MDPs with lower reward horizons are easier to learn. It must be said that practitioners [43, 44, 46, 52] who incorporate PBRS in their work, while finding solace in the policy invariance guarantee [49], rely on experimental evidence rather than providing guarantees for sample efficiency.

3.3 Constraint programming for reward shaping

RL-Tuner by Jaques et al. (2017) [53] is an algorithm used to finetune a sequential generative NN in an RL framework by specifying a set of rules and adjusting the NN to generate sequences that minimize the violation of those rules. In their application to the domain of music, the NN is first pre-trained with SL using a music dataset, and then used as a policy network in a DQN in the RL framework where states are the partial generations of the sequences and actions are the next musical note to append to the sequence. The reward received by the agent is a weighted sum of the penalties from breaking rules as well as the log-probability of the output of the original NN. The goal is to encourage the generation of sequences respecting the specified rules while remaining faithful to the behavior of the original NN. Lafleur et al. (2022) [54] address an inefficiency of the RL-Tuner which is that the penalties are given in an *ad-hoc* fashion. However, rule violations can be detected earlier by using reasoning on the interplay between the user specified rules. A delayed feedback hinders learning as shown in the analysis by Laud et al. (2003) [51]. Lafleur et al. uses a CP solver to perform such reasoning to detect rule violations. To shape the reward, the authors use the probability of no violation being introduced through the choice of the next note and the minimum additional violation, the reward received by the agent being a hand-crafted weighted sum of these two terms. To the best of our knowledge, no other work besides Lafleur et al. (2022) [54] uses CP to do reward shaping. Another particularity of their work is the use of BP to estimate marginal distributions. They have shown that the reward generated using BP, although more computationally expensive, improves training. In the next chapter, we present our improvement on their work by using PBRS to finetune a generative model. We show that by using our reward shaping method, we improve sample efficiency as well as the final performance of the trained models using the same metrics as

Lafleur et al. even without the use of BP.

CHAPTER 4 ARTICLE 1: AN IMPROVED NEURO-SYMBOLIC ARCHITECTURE TO FINE-TUNE GENERATIVE AI SYSTEMS

In this chapter, we present our article titled *An Improved Neuro-Symbolic Architecture to Fine-Tune Generative AI Systems* [55], authored by Chao Yin, Quentin Cappart and Gilles Pesant. Published in the 21st International Conference on Integration of Constraint Programming, Artificial Intelligence and Operations Research (CPAIOR 2024), on 25 May 2024.

4.1 Abstract

Deep generative models excel at replicating the mechanisms that generate a specific set of sequential data. However, learning the underlying constraints preventing the generation of forbidden sequences poses a challenge. Recently, RL-Tuner, a reinforcement learning framework designed for the *ad hoc* fine-tuning of a neural model to adhere to given constraints, was enhanced to learn from the output of two constraint programming models. The first model computes a score representing the number of constraint violations from the currently generated token while the second model provides the marginal probability of that token being generated if no additional violation is allowed. In this paper, we significantly enhance the latter framework in three ways. First, we propose a simplified architecture that requires only a single constraint programming model. Second, we evaluate constraint violations in a more accurate and consistent manner. Third, we propose a reward signal based on belief propagation on this new model that further improves performance. Our experiments, conducted on the same learning task of music generation, demonstrate that our approach surpasses the previous framework both in terms of convergence speed during training and in post-training accuracy. Additionally, our approach exhibits superior generalization to longer sequences than those used during training.

4.2 Introduction

Sequential data exists everywhere we look, from text corpus to musical compositions and even stock market time series. They are important as much as they are abundant, as showcased by impressive advances in natural language processing through the development of Large Language Models (LLM). Their impact will only continue to grow as new advances are made in the field of machine learning. As powerful as generative models can be, it is increasingly desirable to control what is generated amid concerns about factual accuracy and potential

bias [56], but also when the output is only valid if it exhibits a given (combinatorial) structure. Learning such structure from data can be difficult, may require a prohibitive amount of training data, or suffer from slow convergence. Consider the following approach: given a generative neural network pre-trained for a certain task, experts create a CP model to enforce domain-specific rules. For music it could be classic rules of counterpoint whereas for robotics it could be safety rules when operating around humans or to avoid breaking the robot. This CP model can then be used to fine-tune the neural network using reinforcement learning.

RL-Tuner [53], a reinforcement learning framework to fine-tune a neural model in an *ad hoc* fashion in order to satisfy given structural constraints, was recently enhanced [54] by replacing such *ad hoc* constraint penalties with a pair of constraint programming (CP) models, one which computes a score representing the number of constraint violations from the currently generated token and the other, the marginal probability of that token being generated if no additional violation is allowed, through the use of a CP solver computing such probability distributions over finite domains with belief propagation [1]. Our contribution builds on the idea of leveraging marginal probabilities to craft a reward signal to further increase the performance of the RL agent while simplifying the previous architecture. First, we redesign the CP part to require a single (soft) model. Second, we evaluate constraint violations in a more accurate and consistent manner. Third, we propose a reward signal based on the mathematical expectation of the number of constraint violations represented as a discrete random variable.

We give next some fundamental background in Section 4.3 and then review some related work in Section 4.4. The neuro-symbolic architecture we contribute is detailed in Section 4.5. It is followed by an empirical evaluation in Section 4.6. Finally, Section 4.7 makes some concluding remarks.

4.3 Background

4.3.1 Reinforcement Learning

Reinforcement learning is a machine learning paradigm in which an agent tries to maximize rewards collected by interacting with an environment and by learning from experience gained from previous interactions to make better decisions. Each time the agent carries out an action, it moves from a state to another one. The sequence of states encountered by the agent is called an *episode*. One approach to find the optimal policy is to learn the value of an action A_t when taken at a particular state S_t through the reward R_t it receives: such approaches are known as value-based methods. Q-Learning [9] is a specific value-based method that

approximates the value of an action at a given state. To use it on problems with large numbers of states and actions, it is paired with a deep neural network, thus giving it the name Deep Q-Learning (DQN) [2], to learn the representation of state-action pairs. Following previous work [54], we will use a DQN to approximate the state-action values and to sample actions for the agent to take.

4.3.2 Belief Propagation-Based Constraint Programming

Standard constraint propagation, exchanging information between constraints and variables about unsupported variable-value pairs, has been generalized to exchanging information about each variable-value pair’s likelihood of being part of a solution (CP-BP) [1], an unsupported pair corresponding to the special case of zero likelihood. Such message passing typically will not reach a fixpoint but will iterate (until some stopping criterion) between constraint-to-variable and variable-to-constraint message phases, following the classic belief propagation message update equations [57]. These enriched messages coming from constraints are computed using weighted counting algorithms specialized for each type of constraint. As a result, this CP-BP framework computes for a given constraint satisfaction problem (CSP) a marginal probability distribution $\hat{\theta}_v$ over the domain of each variable v which approximates the true marginal probability $\theta_v(d)$ of variable v being assigned value d in a solution drawn uniformly at random. Such information has been used to design novel branching heuristics to solve CSPs [58, 59] and to build neuro-symbolic AI systems featuring CP [54].

4.4 Related Work

In recent years there have been many attempts at addressing the difficulty of learning structured output in neural networks. We restrict our attention to approaches able to handle fairly general structure and group them into layer-based and loss-based methods.

Layer-based methods add a differentiable layer in a neural network architecture, through which back-propagation can take place. SATNet [60] designs a differentiable maximum satisfiability (MAXSAT) solver. Neural Logic Machines [33] adopt a tensor representation of logic predicates to build a neural architecture. Semantic Probabilistic Layers [61] translate hard symbolic constraints into a differentiable logic circuit.

Loss-based methods are a looser combination proposing regularization terms in the loss function. A Semantic Loss function [62] was defined for an arbitrary sentence in propositional logic. Neuro-Symbolic Entropy Regularization [63] relies on logic circuits. CL-STE [64] combines Straight-Through Estimators with an encoding of logical constraints in conjunc-

tive normal form as a loss function. DRNets [65] and CLR-DRNets [66] use the concept of entropy to define loss functions for discrete constraints. An adaptation of the negative pseudo-loglikelihood loss function is used to learn discrete graphical models [67].

Outside such classification, NeurASP [68] and DeepProbLog [69] integrate neural networks into their (probabilistic) logic programming language.

RL-Tuner [53] is a reinforcement learning architecture built to fine-tune a generative neural model using domain knowledge in the form of constraints. Even though it was used on a task in the music domain, in principle it is applicable to an arbitrary generative task in any domain. A trained generative neural network is cloned (blue boxes in Fig. 4.1) so that its copy can be fine-tuned through reinforcement learning (green boxes in Fig. 4.1) using combined rewards from the original neural network ($\log p(A_t|S_t)$) to reflect the similarity to the trained network and from the domain knowledge constraints to reflect the satisfaction of the constraints. The contribution from the latter consisted of some manually-derived weighted sum of penalties.

CP-based RL-Tuner [54] replaced the *ad hoc* constraints and penalties with a pair of CP models (pink box in Fig. 4.1): the *Violations* model to evaluate the degree of constraint violation from the currently generated token, and the *Marginals* model to evaluate the probability of that token being generated if no additional violation is allowed. On a melody generation task, it was shown that a reward signal combining the number of additional violations and the marginal probability of no additional violation after each action taken increased constraint satisfiability with respect to the original RL-Tuner while maintaining similarity to the output of the pretrained neural network (NN).

4.5 Proposed Improved Framework

However, the CP-based RL-Tuner framework [54] brought some technical challenges and inefficiencies at the level of the CP models, which we explain below. Because it is in the nature of an RL agent to sometimes take illegal actions in the course of an episode, both CP models must tolerate past constraint violations. To isolate the impact of the next action, regardless of previous violations, a constraint’s scope is restricted to future actions, with some adjustments. For example, the upper bound of an ATMOST constraint may be reduced depending on whether the value whose number of occurrences we restrict appeared in previous actions. Or for a REGULAR constraint we play out on the corresponding automaton the sequence of actions so far in order to determine the current “initial” state. And therefore after each new action two new CP models must be created: the *Marginals* model, featuring *hard* constraints

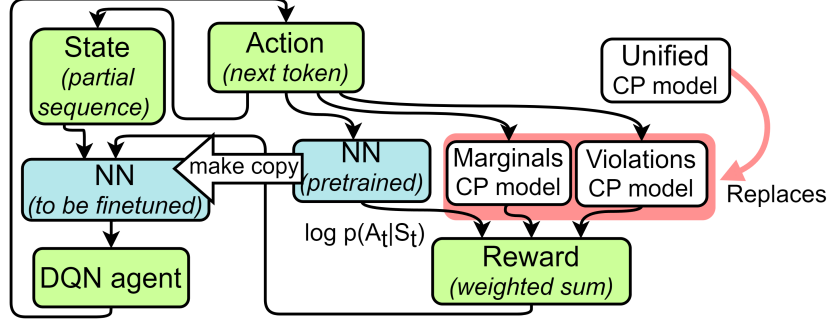


Figure 4.1 CP-based RL-Tuner framework and our proposed modification.

so that the computed marginals correspond to having no future constraint violation, and the *Violations* model, featuring *soft* constraints (through reification or a soft/optimization version) in order to allow and then count constraint violations. This repeated creation of slightly different models is technically cumbersome and comes at a computational cost. The architecture we propose greatly improves on this.

4.5.1 Simplified Architecture

Figure 4.1 gives a schematic representation of our generic architecture, highlighting what we change with respect to the CP-based RL-Tuner architecture. The RL/DQN part driving the learning remains the same and is generic. The neural network (and its clone) serves the generative task in the same way but its nature (e.g. LSTM [70], Transformer [71]) can change depending on the domain. What does change is the CP part, where we replace the two CP models by a single one as explained below.

The design choice made by Lafleur et al. (2022) [54] of using both a hard and a soft CP model was motivated by the desire to reflect in the reward both the number of constraint violations brought about by the current action (requiring the soft model) and the likelihood of that action leading to an episode with no further constraint violation (requiring a hard model so that the corresponding marginal probability $\hat{\theta}_{A_t}(a)$ only reflects violation-free episodes). In other words, a combination of the present and of the future (or promise). It was also remarked by Lafleur et al. (2022) that the two complement each other well during training. In order to simplify this architecture, we give up on the “future” component but, as we will see in Section 4.5.3, replace it by something in the same spirit.

We define a single soft CP model and, even more importantly, use the same one throughout an episode as opposed to a sequence of slightly-different ones. It essentially corresponds to the initial *Violations* model in CP-based RL-Tuner where, instead of restricting the scope

and adjusting the arguments of constraints (and create a new model) as an episode proceeds, we simply perform variable assignments on the same model.

The CP model is defined over a sequence of finite-domain variables $\langle A_t \rangle_{t=1}^n$ representing the actions taken by the agent during the episode (or equivalently the tokens generated). It also includes a variable V corresponding to the number of constraint violations. The relevant constraints are formulated in their reified or soft form and to each we associate a violation variable representing its individual number of violations. A SUM constraint relates them to variable V .

4.5.2 Evaluating Constraint Violations

To count the number of constraint violations from the *Violations* model, Lafleur et al. (2022) [54] use V_{min} , the minimum value in the domain of the variable that sums violations from every constraint. Despite adjusting constraints to ignore violations due to past actions, some inconsistencies about counting violations can still creep in. Consider the following example. We wish to generate a sequence of two tokens in which values a and b should each appear, and count one violation for each missing value. We can use a SOFT-GCC constraint [72] on these values to model this. Suppose the first token generated is c : the constraint reports a minimum of one violation because there is only one other token to generate and two missing required values. So $V_{min} = 1$. A new SOFT-GCC on the remaining token is then posted without adjusting the requirement for a and b since none have been met. Suppose the second token is again c : this time $V_{min} = 2$. So this second action will receive a negative reward twice as large as the first even though they are equally responsible. A fairly simple way to fix this, which we investigate, is to use instead $\Delta V_{min} = V_{min}^t - V_{min}^{t-1}$, where V^t represents the violations variable at time step t (i.e. after the t^{th} token has been generated). This would correct the previous inequity and is in general more consistent.

4.5.3 Reward Signal

The probability-based reward $\hat{\theta}_{A_t}(a)$ of [54] only takes into account sequences for which no additional violations are allowed. On the one hand, good quality solutions of very low numbers of additional violations are ignored. On the other hand, if constraints are softened then that reward takes into account all solutions good and bad. We address this by considering the marginals for the violations variable instead of that for the current action. We propose a reward signal based on the change in the expected value of V , which we call ΔE . After an action A_t the expected value of the cost variable V is given by $\mathbb{E}[V^t] = \sum_{i=V^t, \min()}^{V^t, \max()} i \cdot \hat{\theta}_V^t(i)$. The difference between it and the expected value at the previous time step gives us our pro-

posed reward signal: $\Delta E = \mathbb{E}[V^t] - \mathbb{E}[V^{t-1}]$. Note that this is another advantage of having a probability distribution over domains with the CP-BP framework: one can compute expected values.

For example consider generating a binary sequence of length 5 with the rule that each repeated note creates one violation, enforced by a `COSTREGULAR` constraint on variables $A_1, \dots, A_5$ with the number of violations represented by variable V . Suppose the first three actions have been taken: $A_1 = 0, A_2 = 1, A_3 = 1$. The remaining sequences are 01100, 01101, 01110, and 01111 which have 2, 1, 2, and 3 violations respectively, so $\mathbb{E}[V^{t=3}] = 2$. If the agent chooses action $A_4 = 1$, $\mathbb{E}[V^{t=4}] = 2.5$ and so $\Delta E = 2.5 - 2 = 0.5$ whereas action $A_4 = 0$ yields $\Delta E = 1.5 - 2 = -0.5$.

4.6 Empirical Evaluation

We conducted an empirical evaluation¹ of our improved framework, especially its new reward signal but also the way constraint violations are counted, by reusing the learning task on which RL-Tuner and CP-based RL-Tuner were previously evaluated. Given a data set of melodic lines, the goal is to learn to generate melodies that resemble the style of the input while respecting basic rules from music theory. To obtain a more direct comparison with Lafleur et al. (2022) [54], we opt to use the same musical corpus which is the soprano voices of Bach’s chorales obtained using the Music21 [73] library and transpose the compositions to C major. The dataset is split into 329 compositions for training and 75 for validation. Following the original paper [53], we use a neural architecture based on long short-term memory (LSTM) cells. This network is pretrained using the music corpus to predict the next musical note, we were able to obtain a final validation accuracy of 54% (similar to [54] due to the small size of the corpus). We then use one copy of this network for our DQN to sample actions (next musical note to generate) and another copy to evaluate the similarity of the musical note generated with respect to the original music corpus.

4.6.1 CP Model

A melody is represented as a temporal sequence of note variables whose domain is the set of pitches. Auxiliary variables are also defined, such as those representing intervals (difference between consecutive pitches), to ease the expression of the relevant constraints. Many of the music rules are naturally modeled using `REGULAR` constraints: for our soft model we use `COSTREGULAR` [74] with a unit cost associated to each transition corresponding to a

¹https://github.com/ChYinn/RL_Tuner_CPBP

possible violation. For rules about undesirable pitches or intervals, an AMONG constraint associates a variable to the number of occurrences, which we use as the number of violations. For simpler constraints such as equality, we reify them and use the Boolean variable as the violation variable. We refer the reader to [54] for details about the variety of music rules being enforced. We implement this model using the MiniCPBP solver [75] providing marginal probabilities to compute the reward signals discussed in the previous section. We perform 10 iterations of belief propagation. During training, the solver receives the action taken by the agent (pitch value chosen) at step t and assigns it to variable A_t , constraint propagation and belief propagation are then performed before computing and returning the rewards.

4.6.2 DQN training and evaluation

We use the hyper-parameters shown in Table 4.1 to pre-train our LSTM using musical data and to fine-tune it within the RL-Tuner framework. Training involves generating musical notes one by one until we obtain a melody of 32 notes. After 400 episodes of training, we evaluate our model by randomly sampling a melody of length 32 and 64 to evaluate it by taking the average $\log p(A_t|S_t)$ reward obtained from the pre-trained LSTM and by calculating the total number of violations using the CP model.

4.6.3 Comparing reward signals

There are two metrics when evaluating our models. We would like our agents to learn to respect the constraints imposed on their actions while also keeping a level of similarity to the neural network that was pretrained on the musical corpus. We therefore consider a reward signal to be better when the trained model is able to generate sequences that are equivalent in one metric but superior in another, or superior in both.

The reward signal received by the agent is of the form $\log p(A_t|S_t) + k \times R$ where k is a balance parameter and R is the CP-based reward. Lafleur et al. (2022) [54] found that the combination $k = 2$ and $R = 40 \times \hat{\theta}_{A_t}(a) + V_{min}$ worked best. They also experimented with $k = 2$ and $R = V_{min}$. For our $R = \Delta E$, we chose $k = 4$ after experimenting with a few values. Notice that another advantage of our single-CP-model architecture is that one does not also need to tune the balance between the reward component from each of the two CP models, making tuning much simpler. We also compare $2 \times \Delta V_{min}$ against $2 \times V_{min}$ to illustrate how the *Violations* model can be improved by simply changing how violations are counted (Section 4.5.2).

Table 4.1 Hyper-parameters for our models

Parameter	Value	Parameter	Value
number of layers (LSTM)	2	learning rate	0.001
hidden size (LSTM)	100	memory size	32000
discount rate	0.5	episodes per epoch	400
batch size	128	number of epochs	8
target network	0.01	random action	0.1
update rate		probability	

4.6.4 Experimental results

Looking at the top two graphs at Figure 4.2, we notice that $4 \times \Delta E$ has the lowest number of violations (and fastest convergence) when generating sequences of the training length (32) while also obtaining higher $\log p(A_t|S_t)$ based rewards when compared to $2 \times (40 \times \hat{\theta}_{A_t}(a) + V)$. It is only slightly surpassed by $2 \times V_{min}$ and $2 \times \Delta V_{min}$ in terms of $\log p(A_t|S_t)$ but at the expense of much higher numbers of violations. This means the agent trained using our new reward is able to generate sequences that are competitively similar with respect to the pretrained LSTM while also obtaining the lowest number of violations. Looking at the bottom graphs, we observe that the gain in performance increases for both metrics when generating sequences twice as long as those seen during training: this exemplifies better generalization.

Comparing $2 \times \Delta V_{min}$ and $2 \times V_{min}$, we observe comparable results on both metrics for sequences of length 32 but a marked advantage of the former for length 64. This illustrates the improvement made by simply correcting how violations are counted.

4.7 Conclusion

We improved a reinforcement learning framework designed to fine-tune a neural model to adhere to given structural constraints. We greatly simplified its architecture with respect to the CP part, reducing it to a single CP model throughout an episode and making its implementation considerably more straightforward. We introduced a reward signal that improves the resulting model’s ability to generate sequences respecting constraints while slightly diminishing its similarity to the original model. The results also show that agents trained using our reward signal exhibit better generalization.

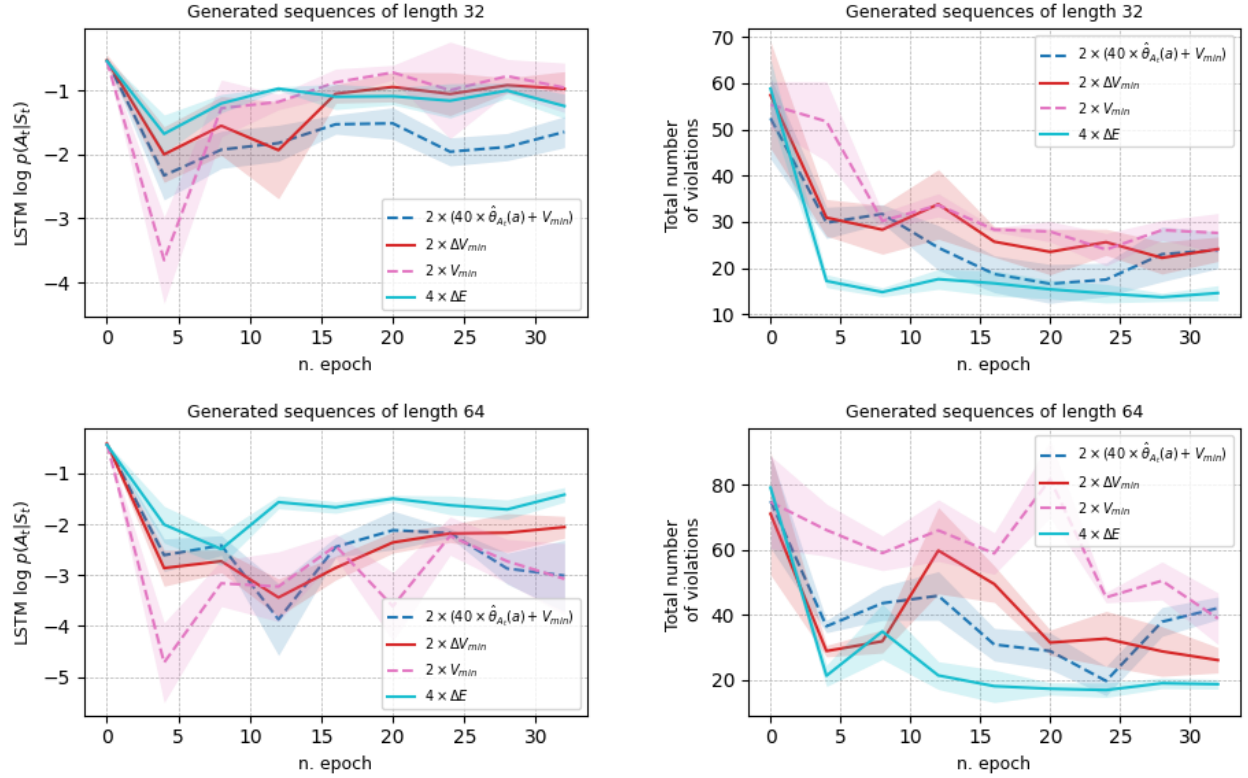


Figure 4.2 Evaluation of agents (using four different reward signals) during training after each epoch. Agents generate 10 melodies of length 32 and 64 which are then evaluated using our two metrics: LSTM similarity and constraint satisfaction. The curves and shaded areas respectively represent the mean and standard deviation.

CHAPTER 5 REWARD SHAPING FOR AI-PLANNING IN RL

We saw in the previous chapter that CP can be used to specify a reward signal in an RL setting to finetune a generative NN to generate sequences that respect user-specified constraints. In this chapter, we explore the application of our reward shaping method to the problem of AI-planning. There are two key challenges posed by the problem of AI-planning which makes it fundamentally different from our previous application.

Firstly, in the previous task of finetuning a generative model, the generated sequences are not required to respect hard constraints, which means every possible sequence is a solution and only the cost of the solution (number of constraint violations) is of interest. In the problem of RL for AI-planning, solutions are sequences of decisions leading the transition from an initial state to a goal state with the costs of these solutions being, for example, their length. In this sense, the condition that the final state must be the goal state is a hard constraint. There are also hard constraints on the allowable operators since their preconditions must be met. Secondly, the number of actions required to solve a planning problem is undetermined whereas in the previous problem of generating a sequence, the length is predetermined. In this chapter, we propose a reward shaping method similar to the one from the previous chapter to improve learning in RL for solving AI-planning problems. We investigate the effect of important design choices when creating such a CP model for an AI-planning problem on our reward shaping method’s effectiveness. Additionally, we investigate the effect of using BP, which is a reasoning technique unique to the CP solver we use. We hypothesize that the use of BP improves the rewards generated albeit at the additional cost of its use. We first propose a method of creating a CP model from a given a SAS+ model. We then apply this method on a well-known problem called Blocks World by creating multiple CP models for this problem. From these models, we propose a reward function similar to the one we used in our previous application. We then experiment in an RL setting on the problem of BW by training using reward functions from the CP models and a sparse reward function as a baseline. In our experiments, we compare the sample efficiency of each method by comparing the performance of learned policies at each epoch of training. We also compare the final performance of each model after training.

5.1 RL setting for AI-planning

We now consider the RL problem of solving the AI-planning problem. We assume a finite horizon, meaning the RL episode terminates after predetermined finite number of actions H

have taken. This is a practical assumption that we make for computational efficiency and because it is the setting used in the work that we borrow [33] to apply our method in our experiments. We start by formally defining the RL setting that we consider by describing the states, actions and transitions. Recall that we can express an AI planning problem using a SAS+ model which has an induced FTS $\langle \Theta^{v_1}.. \Theta^{v_m} \rangle$ with atomic factors Θ^{v_i} . Each action taken by the agent in the RL setting corresponds to an operator $o \in \mathcal{O}$ of induced the FTS which operates on the each of the atomic factors of the FTS to transition to their respective next states. This is done by transitioning the initial state $s_I^{v_i}$ of each atomic factor into the next state s' if the transition $s_I^{v_i} \xrightarrow{o} s' \in T^{v_i}$. Otherwise, the operator o has no effect and the FTS remains unchanged. Therefore, in addition to the FTS the environment must also keep track of the number of steps taken and terminate once it has exceeded the limit. So far, we can define the state of the environment as the FTS with the remaining number of steps. The transitions of the environment are defined by the effect of the operator chosen by the agent on the FTS which we described as well as whether the step limit has been exceeded. For the reward function, one can specify a function that returns a small negative reward for each action taken, say -0.01 , and a positive reward of 1 for an action which solves the problem. Such a reward encourages the agent to solve problems using a minimum of steps. Although this kind of sparse reward is easy to specify, it is not sample efficient as discussed in Chapter 3. Consider now a potential based reward shaping method in the following example.

Example 5.1.1. Consider the BW problem illustrated in Figure 2.7 with its induced FTS depicted in Figure 2.10, the RL setting is depicted in Figure 5.1. We have that the atomic factor Θ^A has an initial state B and goal state air and Θ^B has an initial state air and goal state A and 8 steps to solve the problem. Initially, we have a potential $\phi(s_0)$ which is an estimation of how many steps need to be taken for solving this problem, which is 3.33. After the agent takes the action $A_1 = move(B, ground)$, the initial state of Θ^A has changed from B to air , which corresponds to its goal state. The new potential is 2.33 and the reward r_1 obtained by the agent is $-(2.33 - 3.33) = 1$. The agent takes the action $A_2 = move(B, ground)$ which changes the initial state of Θ^B from air to A , which corresponds to its goal state. The problem is now solved since all atomic factors are such that their initial states corresponds to their goal states. The agent receives the reward $r_2 = -(2.33 - 2) = 0.33$.

In the previous example, the reward received by the agent is a PBRS reward where the potential ϕ is equal to an estimated number of steps used to solve the problem (or the estimated cost where the cost of each action is 1 in this case). We propose to use CP to estimate this value. For the remainder of this chapter, we describe an automated reward

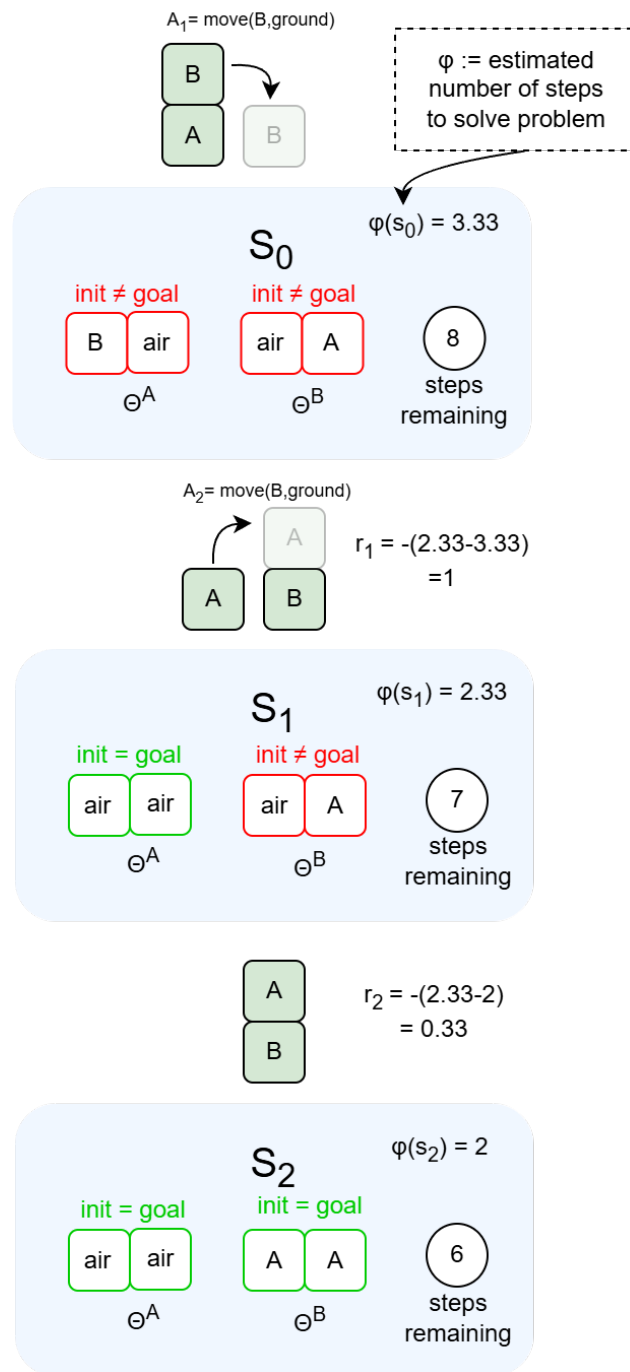


Figure 5.1 BW planning problem in the RL setting with potential-based reward shaping.

shaping method that is more sample efficient than the sparse reward. Our reward shaping method is a potential-based reward shaping method. The potential function defined on each state in the RL setting is calculated with the help of CP. This method, however, requires that the AI-planning problem already has a SAS+ model with an induced FTS that can be readily transformed into a CP model using a method which we describe in the next section. Note that the induced FTS can be trivially obtained once a SAS+ model has been specified by following the rules described in Chapter 2. In this sense, our method is not fully automated as a SAS+ model still needs to be specified and we explain how the choice of the SAS+ model for a given planning problem affects the resulting potential function of our method. However, it is typical that AI planning problems are manually formalized in a SAS+ model or PDDL which can be automatically converted to a SAS+ model.

5.2 Creating a CP model for an AI-planning problem

In this section, we propose a method of creating a CP model for an AI-planning problem and we discuss the important design choices for the CP model with the goal of reward shaping in mind. Suppose we have the following SAS+ model for a planning problem $\Pi = \langle \mathcal{V}, \mathcal{O}, s_I, s_G \rangle$ where $\mathcal{V}$ is a variable space $\langle v_1, \dots, v_n \rangle$ of n variables with each $v_i \in \text{dom}(v_i)$. There is the set of operators $\mathcal{O}$ and s_I and s_G are the initial state and goal state, respectively. To easily express Π as a CP model, we consider its induced transition system which is essentially an automaton with costs associated to each transition and an initial state and goal state. The problem of finding a path between an initial state to a goal state inside an automaton can be stated as a cost-regular constraint. We could therefore immediately stop here and create one cost-regular constraint by copying the parameters of this automaton (states, transitions, operators, costs) and using a sequence of action variables $\langle x_1, \dots, x_H \rangle$ of length H such that $x_i \in \mathcal{O}$ — we detail this in the next section. This CP model would contain in its solution space all sequences of actions of length at most H which lead from the initial state to the goal state. However, interesting planning problems usually have a very large number of states, which results in a cost-regular constraint that makes computations for the CP solver intractable. Recall that the planning problem has an induced FTS $\langle \Theta^{v_1}, \dots, \Theta^{v_n} \rangle$ built from the atomic factors Θ^{v_i} . Instead of using a single cost-regular constraint to express one large automaton, we can instead use multiple cost-regular constraints to model each of the much smaller atomic factors. In the next section, we explain how each atomic factor of an FTS can be transformed into a cost-regular constraint.

5.2.1 Atomic factor as a cost-regular constraint

We now describe our proposed method to express an atomic factor as a constraint in a CP model. Suppose there are n action variables in our CP model, the values in the domain of each action variable are the operators. From a given atomic factor $\Theta^v = \langle S^v, L, T^v, C^v, s_I[v], s_G[v] \rangle$, we create a cost-regular constraint which we denote by $\text{cost-regular}(\Theta^v)$. We have the action variables $x_1, \dots, x_n$.

- The set of states is taken as $S = S_v \cup \{reject, accept\}$. The initial state is s_I and the sole accepting state is $accept$.
- The domain for each variable is taken as $\text{dom}(x_i) = L_v \cup \{finish\}$, for $i = 1..n$. The *finish* operator is added as a placeholder action so that if the problem is solved using fewer than n actions, the rest of the actions can be *finish*.
- T is the transition matrix $|S| \times |L|$. For any $s \in S^v$ and $\ell \in L^v$ we have that $T[s, \ell] = t$ if there exists $t \in S^v$ such that $s \xrightarrow{\ell} t \in T^v$, otherwise $T[s, \ell] = reject$. $T[accept, \ell] = accept$ if $\ell = finish$, $T[accept, \ell] = reject$ otherwise. $T[reject, \ell] = reject$ for all $\ell \in L$. If $s = s_G$, $T[s, finish] = accept$, otherwise $T[s, finish] = reject$.

The transition matrix is such that it agrees with every transition in T^v . Every state-label pair not found in L^v is *illegal* which results in a transition to the *reject* state, which cannot be escaped from. This has the effect of only keeping sequences of operators $\ell \in L^v$ such that the preconditions $pre(\ell)[v]$ are always respected. The only transitions to the *accept* state is to either use *finish* at the state s_G or to use *finish* at the *accept* state. Solutions of this CP model lead from s_I to s_G , we can obtain them by removing the *finish* placeholder variables. The same solution can be obtained for placeholders placed differently, so we impose that *finish* can only be succeeded by *finish* to avoid having redundant solutions in the solution space. For example, $o_1, o_2, finish, o_3$ is equivalent to $o_1, o_2, o_3, finish$ for some $o_1, o_2, o_3 \in \mathcal{I}$.

Algorithm 5 takes as input the induced FTS as well as the number of operator variables H . It begins by initializing the action variables and a cost variable. For simplicity's sake, we assume that the cost of each operator, as prescribed by the FTS, is 1. For each of the atomic factors in the FTS, a cost-regular constraint is created. First, the cost and transition matrices are initialized. They are then filled according to the rules we described previously. The algorithm finishes by returning the set of cost-regular constraints.

Algorithm 5 Creating cost-regular constraints from induced FTS.

```

procedure get_cost_regular( $\langle \Theta^{v_1}.. \Theta^{v_m} \rangle, m$ )    ▷ FTS consisting of atomic factors  $\Theta^{v_i}$ .
   $L \leftarrow (\Theta^{v_0}).L$                                 ▷ Labels of the atomic factor.
  Initialize vars  $\langle x_1..x_m \rangle$  s.t.  $dom(x_i) = \{1..L+1\}$ ,  $i \in \{1..m\}$     ▷ Operator variables.
  Initialize var  $c$  s.t.  $dom(c) = \{0..m\}$                                 ▷ Cost variable.
   $constraints \leftarrow [empty]$ 
  for  $i = 1..m$  do                                ▷ Creating a cost-regular for each atomic factor  $\Theta^{v_i}$ .
     $\langle S^{v_i}, L, T^{v_i}, C^{v_i}, s_I^{v_i}, s_G^{v_i} \rangle \leftarrow \Theta^{v_i}$ 
     $Ts \leftarrow matrix(|dom(v_i)| + 2, |L| + 1)$                                 ▷ Cost-regular transition matrix.
     $Cs \leftarrow matrix(|dom(v_i)| + 2, |L| + 1)$                                 ▷ Cost-regular cost matrix.
    for  $s = 1..|dom(v_i)| + 2$  do
      for  $o = 1..|L| + 1$  do
        if  $s = reject$  then                                ▷ The 'reject' state cannot be left.
           $Ts[s, o] \leftarrow reject$ 
        else if  $s = accept$  then
          if  $o = finish$  then                                ▷ Only 'finish' allowed in the 'accept' state.
             $Ts[s, o] \leftarrow accept$ 
          else
             $Ts[s, o] \leftarrow reject$ 
        else if  $o = finish$  then                                ▷ Can use 'finish' in the  $s_G$  or 'accept' states.
          if  $s = s_G$  then
             $Ts[s, o] \leftarrow accept$ 
          else
             $Ts[s, o] \leftarrow reject$ 
        else if  $(\exists s' \text{ such that } (s \xrightarrow{o} s') \in T^{v_i})$  then
           $Ts[s, o] \leftarrow s'$ 
           $Cs[s, o] \leftarrow 1$ 
        else                                ▷ Not a legal operator, reject.
           $Ts[s, o] \leftarrow reject$ 
     $constraints \leftarrow cost\_regular(\langle x_1..x_H \rangle, Ts, c, Cs, s_I^{v_i}, \{s_G^{v_i}, accept\})$ 
  return  $constraints, c$ 

```

Algorithm 6 Estimating the cost of solving the planning problem.

```

procedure  $\phi^{remaining}(\langle \Theta^{v_1} .. \Theta^{v_m} \rangle, m)$             $\triangleright$  FTS and number of operator variables  $H$ .
   $\mathcal{M} \leftarrow makeSolver()$                                 $\triangleright$  Initialize CP solver.
   $constraints, c \leftarrow get\_cost\_regular(\langle \Theta^{v_1} .. \Theta^{v_m} \rangle, m)$ 
   $\mathcal{M}.add\_constraints(constraints)$ 
   $\mathcal{M}.propagate\_constraints()$                                 $\triangleright$  Constraint propagation.
   $\mathcal{M}.propagate\_beliefs()$                                     $\triangleright$  BP for marginal distributions.
   $p \leftarrow 0$ 
  for  $v \in dom(c)$  do
     $p \leftarrow p + v \cdot \hat{\theta}_c[v]$ 
  return  $p$ 

```

5.2.2 Potential function for reward shaping

We now define the potential function $\phi(s)$ which maps each state to a real value called the potential of the state s where the state s is the current FTS of the environment. Algorithm 6 begins with the creation of a CP model $\mathcal{M}$ for the FTS given as input. It uses algorithm 5 described previously to obtain the cost-regular constraints and the cost variable c associated with them. After constraint propagation and belief propagation, the cost variable c now has a marginal distribution $\hat{\theta}_c$ over its domain. From this distribution, we compute the expected value p . This value is interpreted as the average cost of all solutions in $Sol(\mathcal{M})$. We call this function $\phi^{remaining}$ to highlight the fact that it returns an estimation of the cost of the remaining actions to solve the problem. The potential function we use in our reward shaping method, however, is $\phi = \phi^{previous} + \phi^{remaining}$ where $\phi^{previous}$ is the cost of the previous actions. We explain in section 5.3.1 why it makes sense to use ϕ as our potential function. Figure 5.1 provides some intuition on ϕ , notice how initially when no actions have taken $\phi = \phi^{remaining}$ and when the episode terminates upon the problem being solved, $\phi = \phi^{previous}$.

5.3 Reward shaping algorithm using CP

Algorithm 7 describes the calculation of the PBRs reward after taking a step in the environment. The environment when initialized calculates the potential p_{prev} . Each time the agent takes an action, the environment updates its state by updating the FTS. This is done by transitioning the initial state s_I of each atomic factor in the FTS according to their respective transitions. The reward given is the difference between the new potential p calculated using the updated FTS and the previous one ($p - self.p_{prev}$).

Algorithm 7 Environment functions.

```

procedure Initialize( $\langle \Theta^{v_1}, \dots, \Theta^{v_n} \rangle, m, H$ )
  self. $\Theta \leftarrow \langle \Theta^{v_1}, \dots, \Theta^{v_n} \rangle$ 
  self. $m \leftarrow m$ 
  self. $\phi^{previous} \leftarrow 0$ 
  self. $p_{prev} \leftarrow \phi()$ 
  self.steps  $\leftarrow 0$ 
  self. $H \leftarrow H$ 

procedure Update_state( $a$ ) ▷ Updates the FTS  $\Theta$  according to  $a$ .
  goals  $\leftarrow 0$ 
  for  $i = 1..n$  do
     $s_{next} \leftarrow s$ 
    if  $\exists s' \in (\Theta^{v_i}).S$  such that  $\langle s, a, s' \rangle \in (\Theta^{v_i}).T$  then
       $(\Theta^{v_i}).s_I \leftarrow s_{next}$  ▷ Update  $s_I$  as the next state.
      if  $(\Theta^{v_i}).s_I = (\Theta^{v_i}).s_G$  then ▷ Goal state reached for  $\Theta^{v_i}$ .
        goals  $+= 1$ 
    if goals  $= n$  then ▷ The problem has been solved.
      self.done, self.success  $\leftarrow True, True$ 
    else if self.steps  $= H$  then ▷ Maximum actions reached.
      self.done, self.success  $\leftarrow True, False$ 
    else
      self.done, self.success  $\leftarrow False, False$ 

procedure  $\phi(\cdot)$  ▷ Potential of the current state.
  return  $-(self.\phi^{previous} + \phi^{remaining}(self.\Theta, self.m))$ 

procedure Step( $a$ ) ▷ Action  $a$  taken by agent
  self. $\phi^{previous} \leftarrow self.\phi^{previous} + self.\Theta.c(a)$  ▷ Add cost of  $a$  to  $\phi^{previous}$ 
  Update_state( $a, self.\Theta$ )
   $p \leftarrow \phi()$ 
   $r \leftarrow p - self.p_{prev}$ 
  this. $p_{prev} \leftarrow p$ 
  return  $r, self.done, self.success$ 

```

5.3.1 Interpretation of the potential function

In the previous section, we considered an episodic RL setting to solve a planning problem, the agent being allowed H number of steps to reach the goal state s_G from some initial state s_I before the episode finishes. The environment's states and actions as well as transition dynamics are defined by the FTS of the planning problem $\langle \Theta^{v_1}.. \Theta^{v_n} \rangle$ such that the operators $\mathcal{O}$ and planning cost c are shared for all atomic factors. Let us first consider a simple reward function for which it is easy to see that an optimal policy would be to solve the planning problem with the minimum planning cost, we then show that our shaped reward is equivalent to this reward function from which we subtract a state-dependent *baseline* function $b(s)$. The consequence of this would be that our reward shaping method, when used with the REINFORCE learning algorithm, can be interpreted as an unbiased variance reduction method for the simple reward function. One challenge in reward shaping is the training of an agent to learn unintended behaviors. This equivalence offers us a guarantee that the policies learned by the agent are ranked according to their cost. Consider a simple reward function which returns the negative of the cost for the action a taken by the agent $-c[a]$. If the episode terminates without reaching s_G , a reward of $-H$ is given if the episode did not succeed after H actions have been taken. In this setting, the agent learns a policy to solve the planning problems within H steps while minimizing the cost of the actions taken. For an episode of length T the return R_k^{simple} for the action a_k taken at time k would be given as $R_k^{simple} = \sum_{t=k}^T -c(a_t)$. Let us now look at the return of the agent when using our PBRS reward:

$$R_k^\phi = \sum_{t=k+1}^T (\phi(s_t) - \phi(s_{t-1})) = \phi(s_T) - \phi(s_k) \quad (5.1)$$

Recall that the potential $\phi(s_k)$ function in Algorithm 7 is given as follows:

$$\phi(s_k) = -(\phi^{previous}(s_k) + \phi^{remaining}(s_k)) \quad (5.2)$$

$$= -\left(\sum_{t=1}^k c(a_t) + \mathbb{E} \left[\sum_{t=k+1}^T c(a_t) \middle| a_{k+1}..a_T \sim \dot{\pi} \right] \right) \quad (5.3)$$

The term on the left is the total cost for the actions taken prior to time k . Recall that our CP model is used to approximate the average cost of all sequence of actions leading the current state s_k to the goal state s_G , which is $\phi^{remaining}(s_k)$. Consider a policy $\dot{\pi}$ such that the sequences of actions $\bar{s} \in Sol(\mathcal{M})$ are equally likely to be taken when following $\dot{\pi}$ and that there is a probability of 0 of taking a sequence of action that is not in $Sol(\mathcal{M})$. The term on the right is an expected value for the cost of the remaining actions in order to reach the goal state s_G , assuming that we follow the policy $\dot{\pi}$. We can interpret it as the value function

$V_{\hat{\pi}}(s)$. Now replacing the appropriate term in equation 5.1 using equation 5.3, we get the following:

$$R_k^{\phi} = -\left(\sum_{t=k}^T c(a_t) + V_{\hat{\pi}}(s_k)\right) = R_k^{simple} - V_{\hat{\pi}}(s_k)$$

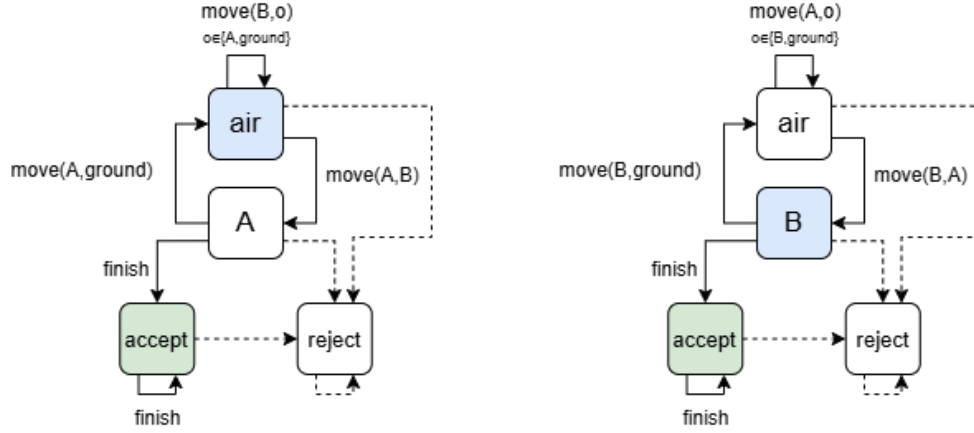
This shows that the return R_k^{ϕ} using our PBRS reward is equivalent to the return R_k of the simple reward function minus a state-dependent baseline value $V_{\hat{\pi}}(s_k)$. This can be interpreted as a variance reduction method for policy gradient methods that adds no bias since our potential function $V_{\hat{\pi}}(s)$ is state dependent. Thus, we have the guarantee that the optimal policies when using R_k and R_k^{ϕ} are the same. We discussed in Chapter 3 that it is desirable for the baseline to be adaptive such as using the value function $V_{\pi}(s)$ which evolves as agent's policy π is updated. A limitation of our approach is that the policy $\hat{\pi}$ is fixed which means that the baseline $V_{\hat{\pi}}(s)$ does not change as training progresses. It needs to be said that the expected value in 5.3 is calculated by the CP solver could be an approximation instead of the true expected value.

Example 5.3.1. We now illustrate our method of calculating the potential function using a simple example of a Blocks World problem. Consider the Blocks World problem described in Figure 2.7 consisting of two blocks and the induced FTS of a SAS+ model of this problem described in Figure 2.10. From it, we create the cost-regular constraints whose automata are depicted in Figure 5.2 where the blue boxes represent initial states and the green boxes represent accepting states. If a transition is not represented by a named arc, it is represented by the dashed arc.

Suppose now that our CP model $\mathcal{M}$ consists of the constraints $\text{cost-regular}(\Theta^A)$ and $\text{cost-regular}(\Theta^B)$ and we have chosen the number of action variables to be $m = 4$. There are three solutions for this CP model as shown in Table 5.1. Assuming global consistency on the cost variable, the values in its domain would be $\{2, 4\}$. By using MiniCPBP, we can approximate the true marginal distribution of both of these values, which is $1/3$ for the value 2 and $2/3$ for 4. We then use the distribution to estimate the average cost of all solutions $(2 + 4 + 4)/3 = (1/3) \cdot 2 + (2/3) \cdot 4 = 3.\bar{3}$. This is the potential calculated by the ϕ function in Algorithm 7. Without approximating the distribution (i.e. without BP), the values remaining in the domain after filtering can be interpreted as having a non-zero probability. For lack of more precise information, we can assume their probabilities to be equal and calculate a potential by taking the average of the remaining values $(2 + 4)/2 = 3$.

x_1	x_2	x_3	x_4	$cost$
$m(A, ground)$	$m(B, A)$	$finish$	$finish$	2
$m(A, ground)$	$m(A, B)$	$m(A, ground)$	$m(B, A)$	4
$m(A, ground)$	$m(B, A)$	$m(B, ground)$	$m(B, A)$	4

Table 5.1 Solutions for the CP model of Blocks World

Figure 5.2 Automata for the constraints $cost-regular(\Theta^B)$ and $cost-regular(\Theta^A)$

5.3.2 Design choices for the CP model affecting the shaped reward.

Since our goal is to calculate a potential to do PBRs with the help of a CP model, there are two main considerations when designing the CP model. The first being the ability to filter values from the cost variable and the second being the quality of the solutions in the solution space. Without the use of BP, we can calculate a potential function using a given CP model by taking the average of the values in the cost variable as seen in the previous example. We would expect that the estimated value using a model with a stronger filtering ability to be closer to the true average value, since it could filter out more impossible values. In this sense, the filtering ability of our CP model is tied to the accuracy of our chosen heuristic. To achieve higher filtering ability, recall that for an operator to be used for a transition, the preconditions on all of the atomic factors must be met. From the perspective of a single $cost-regular(\Theta^{v_i})$ constraint, if it can be determined that the local state does not meet the precondition $pre(o)[v_i]$ on an operator $o \in dom(x_t)$, it must be filtered out. One mechanism for this reasoning to take into account the preconditions on all variables is done naturally through constraint propagation, which is not perfect since it does not lead to global consistency. To increase filtering ability, we can create incorporate within the local state information on the states of other variables, which would of course increase the size

of the state space. We give a concrete example in 5.4.1 where we provide for the problem of Blocks World 3 CP models with an increasing ability to filter out values from the cost variable. The choice for the number of action variables m determines the length of the solutions of the CP model. There needs to be enough action variables to at least ensure the solution space is not empty, which requires a lower bound on the problem. If the cost of operators are all non-negative, then the greater m is, the higher the cost of the solutions present in $Sol(\mathcal{M})$. In this sense, m affects the quality of the solutions in the solution space and thus our potential function. In our work, we treat m as a hyperparameter that needs to be pre determined, it can be taken as a known lower bound for the problem.

5.3.3 Curriculum learning

We now describe the training procedure, called curriculum learning [76], used in Dong et al. [33] which we borrow in our experiments. Algorithm 8 describes curriculum learning for the problem of Blocks World. The agent is trained on BW instances of $B = B^{init}$ to B^{grad} . The agent is periodically evaluated by randomly generating BW problems of B blocks which is B^{init} initially. If the success rate of these problems reaches above a certain threshold, B is increased by 1. The evaluation repeats until either the agent fails to reach the threshold, where the training then stops, or the agent successfully solved all instances when evaluated on problems of B^{grad} blocks, in which case we say the agent has graduated. If the agent has not graduated after evaluation, the training continues.

5.4 Experimentation

In this section, we provide an empirical evaluation of our reward shaping method on the planning problem of Blocks World. The goals of our experiments are to determine the effects of the different CP models used as well as the use of BP on sample efficiency as well as the final performance of the trained models. To do this, we borrow the Neural Logic Machines [33] proposed by Dong et al. (2019) as a baseline. The NLM is a neural architecture which learns to solve the Blocks World problem in an RL setting. They use a sparse reward function which returns 1 if the agent successfully solves the problem and a negative reward of -0.01 for each action taken. We replace the sparse reward function used in their RL framework with our shaped reward function to experimentally demonstrate the improvement in training performance. Our shaped reward function requires a CP model for the problem of Blocks World which we detail in 5.4.1.

The same hyper-parameters used in the experiments done by [33] are re-used for our ex-

Algorithm 8 Curriculum learning for Blocks World

```

Initialize  $NLM, B$  as  $B^{init}$ 
 $graduated \leftarrow False$ 
for  $i = 1, \dots, N_{epochs}$  do
  Initialize problem  $env^B$  ▷ Random BW problem of  $B$  blocks.
   $loss \leftarrow \text{run\_episode}(NLM, env^B)$  ▷ From REINFORCE Algorithm 3.
   $loss.backpropagate()$ 
  while  $i \% 10 = 0$  and  $eval\_model(NLM)$  and (not  $graduated$ ) do
     $B \leftarrow B + 1$ 
    if  $B = B^{grad}$  then
       $graduated \leftarrow True$ 
return  $graduated$ 

```

Table 5.2 Hyper-parameters for training in the Blocks World problem

Parameter	Value	Parameter	Value
depth (NLM)	7	B^{init}	2
breadth (NLM)	2	B^{grad}	12
discount rate	0.99	H	$4 \cdot B$
learning rate	0.005	learning rate decay	0.9
entropy beta	0.2	entropy beta decay	0.8

periments (see Figure 5.2), except for the discount rate. We modified the discount rate to train our models when using our shaped rewards which was originally 0.99 to 1.0 as we have not considered the use of a discount factor in our analysis in 5.1. We also introduced a new hyperparameter in our experiments, which is a constant we use to multiply the return received by the agent, we call this the *reward multiplier*. To choose this hyperparameter, we simply tested some constants to see if the curriculum level was able to increase from 2 to 3 and chose the first one that succeeded. We noticed that when using our shaped reward without BP, the magnitude was very low even when the episode ends in success. We therefore use a reward multiplier of 2 to amplify the reward. We find that when the episode ends in failure, the return received by the agent using our shaped reward is negative and has a very high magnitude which destabilizes the training. We therefore multiply the return by 1/20 when the episode ends in failure. The effect of using a different multiplier for the cases of when the episode ends in a success or failure is that the potential function π is no longer solely dependent on the state as the probability of success is dependent on the action chosen. This could potentially mean that some bias has been introduced by losing the theoretical guarantee of PBRS [49]. Such is the practical choice we make to manage the high magnitude of the reward, we observe experimentally whether this introduces bias.

	Sparse reward (NLM baseline)	Shaped reward (no BP)	Shaped reward (BP)
Success	1	2	1
Failure	1	2/20	1/20

Figure 5.3 Reward multiplier hyperparameters for our experiments.

5.4.1 CP models for Blocks World

For the problem of Blocks World with n blocks, we provide three SAS+ models $\Pi_1.. \Pi_3$ which are then used to create the CP models by taking their induced FTS and using the algorithm we described in the previous section. We will see that they result in three different CP models $\Theta_1, \Theta_2, \Theta_3$. We verify experimentally that Θ_3 is able to filter out more values from the cost variable than Θ_2 and the latter filters more value than Θ_1 . We will see that the increase in filtering ability of the resulting CP models come at the cost of identifying insights about the problem which makes their creation more dependent on human expertise. We therefore investigate the relevance of this increase in filtering power in our experiments.

In the SAS+ model Π_1 at Figure 5.4, each variable X_i in $\mathcal{V}$ represents the entity that is on top of block x_i , which can be another block x_j that is not x_i or x_0 (*air* which we represent with x_0). In this SAS+ model, each variable contains information on the above entity only. The actions of moving x_i to another block x_j called $move(x_i, x_j)$ require that the block on top of x_i and x_j is x_0 (the air). Similarly for $move(x_i, ground)$ without any preconditions on *ground*.

In model Π_2 described in Figure 5.5, each variable represents the entity above as well as an indicator of whether the current block is on top of the correct entity in the goal state. We therefore have twice the number of states as Figure 5.4. The move operators are defined as $move^{(y_i, y_j)}(x_i, x_j)$ for $y_i, y_j \in \{0, 1\}$ where y_i indicates whether the block below x_i is correct.

$\mathcal{V}$	$\langle X_1..X_n \rangle :$	$dom(X_i) = \{x_k k \in \{0..n\}, k \neq i\}$
$\mathcal{O}$	$move(x_i, ground):$ $move(x_i, x_j), i, j \in \{1..n\}, i \neq j:$	$pre(move(x_i, ground)) = \{X_i \rightarrow x_0\}$ $eff(move(x_i, ground)) = \{\}$ $pre(move(x_i, x_j)) = \{X_i \rightarrow x_0, X_j \rightarrow x_0\}$ $eff(move(x_i, x_j)) = \{X_j \rightarrow x_i\}$
$s_I =$	$\{\langle X_1..X_n \rangle \rightarrow \langle x_{1,I}..x_{n,I} \rangle\}$	
$s_G =$	$\{\langle X_1..X_n \rangle \rightarrow \langle x_{1,G}..x_{n,G} \rangle\}$	

Figure 5.4 SAS+ model Π_1 for Blocks World

$\mathcal{V} =$	$\langle X_1..X_n \rangle :$	$dom(X_i) = \{\langle x_k, y \rangle \mid k \in \{0..n\}, k \neq i, y \in \{0, 1\}\}$
$\mathcal{O} =$	$move^{\langle y_i, y_j \rangle}(x_i, x_j), \quad i, j \in \{1..n\}, i \neq j, y_i, y_j \in \{0, 1\}:$ $move^{\langle y_i, 1 \rangle}(x_i, ground), \quad i \in \{1..n\}, y_i \in \{0, 1\}:$	$pre(move^{\langle y_i, y_j \rangle}(x_i, x_j)) = \{X_i \rightarrow \langle x_0, y_i \rangle, X_j \rightarrow \langle x_0, y_j \rangle\}$ $eff(move^{\langle y_i, y_j \rangle}(x_i, x_j)) = \{X_i \rightarrow \langle x_0, Equal(x_j, x_{i,G}) \rangle, X_j \rightarrow \langle x_i, y_j \rangle\}$ $pre(move^{\langle y_i, 1 \rangle}(x_i, ground)) = \{X_i \rightarrow \langle x_0, y_i \rangle\}$ $eff(move^{\langle y_i, 1 \rangle}(ground)) = \{X_i \rightarrow \langle x_0, Equal(ground, x_{i,G}) \rangle\}$
$s_I =$	$\{\langle X_1..X_n \rangle \rightarrow \langle \langle x_{1,I}, y_{1,I} \rangle .. \langle x_{n,I}, y_{n,I} \rangle \rangle\}$	
$s_G =$	$\{\langle X_1..X_n \rangle \rightarrow \langle \langle x_{1,G}, 1 \rangle .. \langle x_{n,G}, 1 \rangle \rangle\}$	

Figure 5.5 SAS+ model Π_2 for Blocks World, $Equal(\cdot, \cdot)$ is 1 if both arguments are equal, 0 otherwise.

Notice that in this SAS+ model, we have multiple copies of the same operator but with different pre-requisites. We amend our Algorithm 6 by representing such equivalent actions as a single action in the transition matrix.

Finally, we create Π_3 from Π_2 by adding a variable T in the variable space which represents the number of stacks of blocks. In model Π_3 , the domain size for each X_i is twice that of Π_2 since it must additionally indicate whether the entity below is the ground. This information is required to determine the effect of an operator on the T variable which keeps track of the total number of stacks. When a block resting on the ground is moved on top of another block, the number of stacks decreases by 1. When a block resting on another block is moved to the ground, the number of stacks increases by 1.

5.4.2 Results

We used the methodology outlined in the previous section to train the agent using different types of rewards and to compare them. We use the states visited in the agent's trajectory during curriculum learning using randomly generated BW instances from 2 blocks to 5 blocks. Using each of these states, we compared the domain values of the cost variable c_1, c_2, c_3 in each of the three CP models we created from $\Pi_1.. \Pi_3$. We verified that $dom(c_3) \subseteq dom(c_2) \subseteq dom(c_1)$. We did this verification on 2698, 3637, 1309, 3074 such states for BW of 2,3,4,5 blocks, respectively. We train our models using shaped rewards created from our three

$\mathcal{V} =$	$\langle X_1..X_n, T \rangle :$	$dom(X_i) = \{\langle x_k, y, z \rangle \mid k \in \{0..n\}, k \neq i, y \in \{0, 1\}, z \in \{0, 1\}\}, dom(T) = \{1..n\}$
$\mathcal{O} = \{$	$move^{\langle y_i, z_i, y_j, z_j, t \rangle}(x_i, x_j),$ $i, j, t \in \{1..n\}, i \neq j,$ $y_i, z_i, y_j, z_j \in \{0, 1\}:$ $move^{\langle y_i, z_i, 1, 1, t \rangle}(x_i, ground),$ $i, t \in \{1..n\}, y_i, z_i \in \{0, 1\}:$	$pre(move^{\langle y_i, z_i, y_j, z_j, t \rangle}(x_i, x_j)) = \{X_i \rightarrow \langle x_0, y_i, 1 \rangle, X_j \rightarrow \langle x_0, y_j, z_j \rangle, T \rightarrow t\}$ $eff(move^{\langle y_i, z_i, y_j, z_j \rangle}(x_i, x_j)) = \{X_i \rightarrow \langle x_0, Equal(x_j, x_{i,G}), 0 \rangle, X_j \rightarrow \langle x_i, y_j, z_j \rangle, T \rightarrow t - z_i\}$ $pre(move^{\langle y_i, z_i, 1, 1, t \rangle}(x_i, ground)) = \{X_i \rightarrow \langle x_0, y_i, z_i \rangle, T \rightarrow t\}$ $eff(move^{\langle y_i, z_i, 1, 1, t \rangle}(ground)) = \{X_i \rightarrow \langle x_0, Equal(ground, x_{i,G}), z_i \rangle, T \rightarrow t + (1 - z_i)\}$
$s_I =$	$\{\langle X_1..X_n, T \rangle \rightarrow \langle \langle x_{1,I}, y_{1,I} \rangle .. \langle x_{n,I}, y_{n,I} \rangle \rangle, t_I\}$	
$s_G =$	$\{\langle X_1..X_n, T \rangle \rightarrow \langle \langle x_{1,G}, 1 \rangle .. \langle x_{n,G}, 1 \rangle \rangle, t_G\}$	

Figure 5.6 SAS+ model Π_3 for Blocks World

proposed CP models $\Pi_1.. \Pi_3$ with and without the use of BP. Using the CP rewards without BP, we were unable to train the agents beyond curriculum level 10 without the performance of the agent collapsing to 0% success rate which is not the case when using BP. This tells us immediately that shaped rewards using BP improves training by allowing the curriculum level to reach higher. To compare models trained with rewards without BP, we conduct two separate experiments, one in which we train from level 2 to 12 using only shaped reward with BP, the other from 2 to 10 where we compare the shaped rewards with and without BP. For each of the types of rewards, we used 10 different seeds and report the median value and shade between its 25th and 75th quantiles. In Figure 5.7 and Figure 5.8, we plot two separate graphs sharing the same x-axis, which is the total number of epochs used for training ranging from 0 to 500. The y-axis of the top graph is the median curriculum level across all 10 seeds. The y-axis of the bottom graph is the cumulative ratio of seeds where the agent has graduated. For example, at epoch 300, the agent trained using the sparse reward has graduated in 30% of the seeds whereas it is 100% for the agent trained with Θ_2 (BP).

Looking at the top graph in Figure 5.7, we see that the agents trained using the shaped rewards from all three models Π_1 , Π_2 and Π_3 are able to increase the curriculum faster than the agent trained using the sparse reward. The bottom graph also shows that the agents using shaped rewards graduate earlier than the agent using the sparse reward. They also

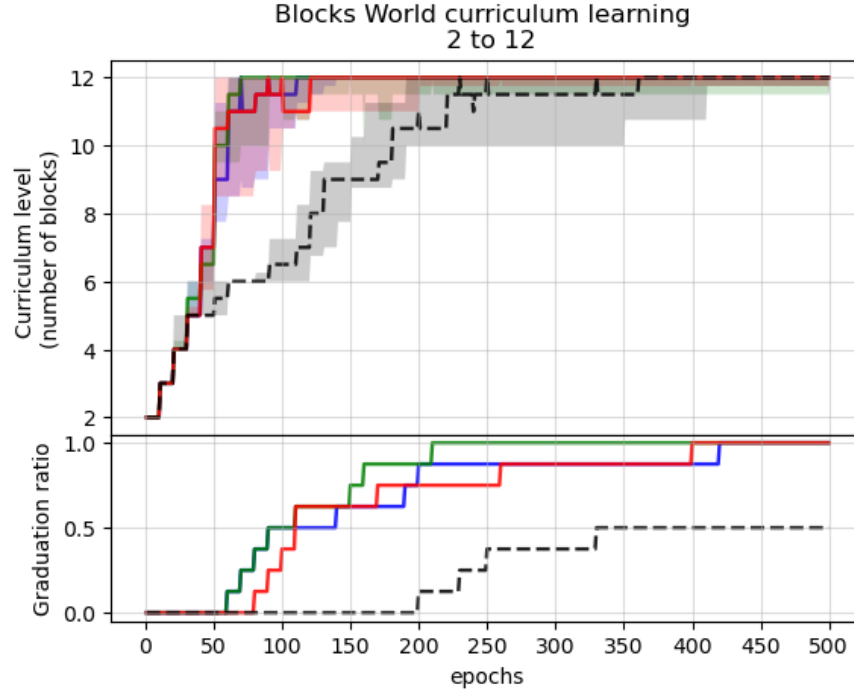


Figure 5.7 Curriculum learning based training of Blocks World from instances of 2 blocks to 12 blocks. Green: Π_2 (BP), blue: Π_1 (BP), red: Π_3 (BP), black-dashed: sparse baseline.

Model \ N. Blocks	N. Blocks				
	10	20	30	40	50
sparse baseline	0.19 ± 0.15	0.21 ± 0.11	0.20 ± 0.09	0.19 ± 0.07	0.18 ± 0.06
Π_1 (with BP)	0.21 ± 0.15	0.19 ± 0.10	0.17 ± 0.07	0.15 ± 0.06	0.14 ± 0.05
Π_2 (with BP)	0.19 ± 0.14	0.19 ± 0.10	0.18 ± 0.08	0.16 ± 0.06	0.16 ± 0.06
Π_3 (with BP)	0.21 ± 0.14	0.19 ± 0.10	0.17 ± 0.08	0.15 ± 0.06	0.14 ± 0.05

Table 5.3 Optimality gaps for BW problems of varying numbers of blocks.

Model \ N. Blocks	N. Blocks				
	10	20	30	40	50
sparse baseline	0.87	0.8	0.75	0.72	0.69
Π_1 (with BP)	1	1	1	1	0.99
Π_2 (with BP)	1	0.98	0.89	0.83	0.80
Π_3 (with BP)	1	1	0.98	0.96	0.94

Table 5.4 Success rate for BW problems of varying numbers of blocks.

graduate more consistently as they have graduated in all 10 seeds whereas the agent with the sparse rewards graduated in 6 out of the 10 seeds. This is consistent with the original experiments by Dong et al. (2018) [33]. We also note that surprisingly, the CP model used did not affect much the training performance. We present below the formula used to calculate the *optimality gap* which is a metric we use to evaluate the final models.

$$\text{Optimality gap} = \frac{\text{cost}_{\text{obtained}} - \text{cost}_{\text{optimal}}}{\text{cost}_{\text{optimal}}}$$

Where $\text{cost}_{\text{obtained}}$ is the cost of the solution for the BW problem instance provided by the agent and $\text{cost}_{\text{optimal}}$ is the optimal cost. In the case of BW it would be the minimum number of steps to solve the problem. In Table 5.3, we selected for each model seeds for which the model has graduated and the success rate is 100% on all test problems. For the sparse reward baseline there are 4 such seeds. For Π_1 with BP there are 4, For Π_2 with BP there are 3 and for Π_3 with BP there are 6. In the aforementioned table, we show the mean and standard deviation of the optimality gaps when using the chosen seeds for each model to solve BW problems of 10,20..50 blocks. For lower numbers of blocks, the optimality gaps seem identical across all models. We observe that the mean optimality gaps of $\Pi_1.. \Pi_3$ (with BP) become slightly lower than the mean sparse baseline optimality gap as the number of blocks increases. In Table 5.4, we evaluate the success ratio in solving BW instances across all seeds. Here we can see that the success ratios of the models $\Pi_1.. \Pi_3$ (with BP) are substantially higher than the baseline across all numbers of blocks. This is because we include seeds that have not graduated after 500 epochs of training.

We now look at the top graph of Figure 5.8 where we can see that once again the agents using shaped rewards outperform the agent using the sparse reward as the curriculum level advances more quickly. We also notice that there seems to be a slightly slower advancement in curriculum level for Π_1 without BP compared to Π_1 with BP, Π_2, Π_3 without BP. Similarly, the bottom graph shows that Π_1 without BP also has a lower ratio of graduated seeds compared to Π_2 without BP and the latter having a lower ratio than Π_3 without BP after 500 epochs of training. This result shows that while the training performance is similar for both Π_1 and Π_3 when BP is used, Π_3 outperforms Π_2 and the latter outperforms Π_1 when BP is not used. This would suggest that the use of BP is able to bridge the gap between both models when used. To conclude our analysis, our results show that our potential based reward shaping method, when used with BP, is more sample efficient than the sparse reward and is able to train an agent with a final performance that is competitive with the sparse reward. We also showed that without BP, the effectiveness of our shaped rewards is heavily affected by the choice of the CP model as seen in Figure 5.8.

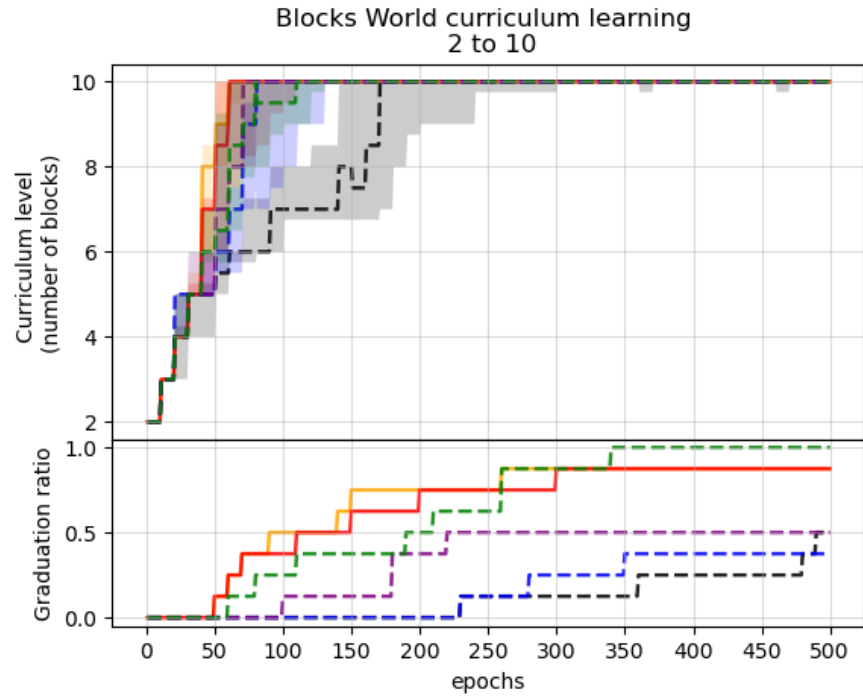


Figure 5.8 Curriculum training of Blocks World from instances of 2 blocks to 10 blocks. Blue-dashed: Π_1 (no BP), green-dashed: Π_3 (no BP), yellow: Π_1 (BP), red: Π_3 (BP), purple-dashed: Π_2 (no BP) black-dashed: sparse baseline

CHAPTER 6 CONCLUSION

6.1 Summary of works

Our work proposed an automated reward shaping method to improve sample efficiency in RL. After providing a CP model for a given problem, a shaped reward can be created using information extracted from the model with the help of a CP solver. The answer to Q1 is that problems which can be expressed using CP models are suitable for our method to be applied, such as musical constraints used to finetune a generative model or AI planning problems with discrete variable domains and set of operators. We found through our experiments that in the latter two problems, the use of BP improved sample efficiency as well as a pronounced improvement on the final trained model in the case of finetuning, this answers Q4. To answer our research question Q2, we have shown that CP can improve sample efficiency in RL by a successful application of reward shaping. To answer Q3, we investigated the effects of using CP models with increasing ability to filter values from the cost variable’s domain for the problem of Blocks World and found that while an increase in filtering benefits our reward shaping method’s effectiveness, this gap in benefit disappears when we used BP. In the first part of our work, we applied our reward shaping method to the problem of finetuning generative AI for music to respect user-specified rules. We compared our method against previous work for the same task and showed that our method improves training performance when considering the metrics used by the original authors. More specifically, our trained models generate music that features fewer rule violations and is more faithful to the original model prior to finetuning. We also found that our shaped reward using BP had the highest performance. In addition, these improvements remain when generating sequences longer than those seen during training. In the second part of our work, we applied our method to AI-planning problems. We proposed a framework that produces shaped rewards which involves taking a SAS+ model and converting it into a CP model. We applied this method to an existing work on the task of block stacking called Blocks World where the authors used sparse rewards to train their model in a curriculum learning setting. We showed that our shaped reward speeds up the training by advancing more rapidly in the curriculum and does this more consistently over 10 randomized seeds. Results also showed that without using BP, a more costly CP model with a higher ability to filter values from the cost variable’s domain had better training performance than the cheaper CP model. However with the use of BP this disparity seems to disappear and can be used to train the agent to reach a higher curriculum level than without the use of BP.

6.2 Limitations

The first limitation to our work is the use of CP which requires that the problem is of a discrete nature. Because CP solvers cannot typically handle continuous variables, applying our method requires discretization of continuous variables. It is not clear what effect this would have on the quality of the shaped rewards since all of our experiments are done on discrete problems. For example, while it is easy to imagine our method being applied to a game like Sokoban where the position of objects can be represented by integers and the number of actions is finite, it would be difficult to apply it to a real time game like Minecraft where the environment is almost entirely continuous. It would also be difficult to finetune AI generative models which generate images or sound waves. The second limitation to our work is the lack of differentiable parts in the generated reward. This means that the reward obtained by the agent for the same state action pairs is the same throughout training. If we interpret our reward shaping method as a baseline for the REINFORCE algorithm as we have discussed in Chapter 5, it would be beneficial for the baseline to be updated during training.

6.3 Future research

In future work, our method could be applied to problems that have continuous variables. The first approach to dealing with such problems is to discretize the continuous variables if possible. Another would be to partially model the RL environment in such a way to ignore the continuous components. For example, one can imagine a continuous-time version of Blocks World where a robot arm is trained to stack blocks. Our proposed reward shaping method would still work albeit the CP model used would not be a full representation of the environment. Work can also be done to make the BP component in the CP solver we use differentiable. Currently BP counts each solution without weighting it. We could imagine encoding the CP factor graph as a graphical neural network such that the cost node is used as a differentiable value function in order to replace the BP algorithm used in the MiniCPBP solver we use.

REFERENCES

- [1] G. Pesant, “From Support Propagation to Belief Propagation in Constraint Programming,” *Journal of Artificial Intelligence Research*, vol. 66, Sep. 2019. [Online]. Available: <https://jair.org/index.php/jair/article/view/11487>
- [2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: <https://www.nature.com/articles/nature14236>
- [3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016. [Online]. Available: <https://www.nature.com/articles/nature16961>
- [4] J. Tromp and G. Farnebäck, “Combinatorics of Go,” in *Computers and Games*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, H. J. Van Den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4630, pp. 84–99, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-540-75538-8_8
- [5] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, “Grandmaster level in StarCraft II using multi-agent reinforcement learning,” *Nature*, vol. 575, no. 7782, pp. 350–354, Nov. 2019. [Online]. Available: <https://www.nature.com/articles/s41586-019-1724-z>

- [6] N. Sünderhauf, O. Brock, W. Scheirer, R. Hadsell, D. Fox, J. Leitner, B. Upcroft, P. Abbeel, W. Burgard, M. Milford, and P. Corke, “The limits and potentials of deep learning for robotics,” *The International Journal of Robotics Research*, vol. 37, no. 4-5, pp. 405–420, 2018.
- [7] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, “End to End Learning for Self-Driving Cars,” Apr. 2016, arXiv:1604.07316. [Online]. Available: <http://arxiv.org/abs/1604.07316>
- [8] W. H. Guss, C. Codel, K. Hofmann, B. Houghton, N. Kuno, S. Milani, S. Mohanty, D. P. Liebana, R. Salakhutdinov, N. Topin, M. Veloso, and P. Wang, “The MineRL 2019 Competition on Sample Efficient Reinforcement Learning using Human Priors,” *CoRR*, 2019.
- [9] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992. [Online]. Available: <https://doi.org/10.1007/BF00992698>
- [10] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, Oct. 2018.
- [11] T. M. Mitchell, *Machine Learning*, 1st ed. USA: McGraw-Hill, Inc., 1997.
- [12] J. Pearl, “Reverend Bayes on Inference Engines: A Distributed Hierarchical Approach,” in *Proceedings of the National Conference on Artificial Intelligence, Pittsburgh, PA, USA, August 18-20, 1982*, D. L. Waltz, Ed. AAAI Press, 1982, pp. 133–136. [Online]. Available: <http://www.aaai.org/Library/AAAI/1982/aaai82-032.php>
- [13] C. Bäckström, “Expressive equivalence of planning formalisms,” *Artificial Intelligence*, vol. 76, no. 1-2, pp. 17–34, Jul. 1995. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/000437029400081B>
- [14] G. Malik, “PDDL | The Planning Domain Definition Language,” Oct. 1998. [Online]. Available: <https://www.cs.cmu.edu/~mmv/planning/readings/98aips-PDDL.pdf>
- [15] S. Bubeck and N. Cesa-Bianchi, “Regret analysis of stochastic and nonstochastic multi-armed bandit problems,” *Foundations and Trends in Machine Learning*, vol. 5, no. 1, pp. 1–122, 2012.
- [16] S. Agrawal and R. Jia, “Optimistic posterior sampling for reinforcement learning: worst-case regret bounds,” in *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc., 2017.

- [17] M. G. Azar, I. Osband, and R. Munos, “Minimax regret bounds for reinforcement learning,” in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 06–11 Aug 2017, pp. 263–272.
- [18] P. Auer, T. Jaksch, and R. Ortner, “Near-optimal Regret Bounds for Reinforcement Learning,” in *Advances in Neural Information Processing Systems*, vol. 21. Curran Associates, Inc., 2008.
- [19] C. Jin, Z. Allen-Zhu, S. Bubeck, and M. I. Jordan, “Is Q-Learning Provably Efficient?” in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018.
- [20] O. D. Domingues, P. Ménard, E. Kaufmann, and M. Valko, “Episodic Reinforcement Learning in Finite MDPs: Minimax Lower Bounds Revisited,” in *Proceedings of the 32nd International Conference on Algorithmic Learning Theory*. PMLR, Mar. 2021, pp. 578–598, iSSN: 2640-3498. [Online]. Available: <https://proceedings.mlr.press/v132/domingues21a.html>
- [21] Y. Abbasi-Yadkori, P. Bartlett, K. Bhatia, N. Lazic, C. Szepesvari, and G. Weisz, “POLITEX: Regret Bounds for Policy Iteration using Expert Prediction,” in *Proceedings of the 36th International Conference on Machine Learning*. PMLR, May 2019, pp. 3692–3702, iSSN: 2640-3498. [Online]. Available: <https://proceedings.mlr.press/v97/lazic19a.html>
- [22] Y. Yu, “Towards Sample Efficient Reinforcement Learning,” in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*. Stockholm, Sweden: International Joint Conferences on Artificial Intelligence Organization, Jul. 2018, pp. 5739–5743. [Online]. Available: <https://www.ijcai.org/proceedings/2018/820>
- [23] M. Plappert, R. Houthoofd, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz, “Parameter space noise for exploration,” in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: <https://openreview.net/forum?id=ByBA12eAZ>
- [24] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, “Noisy networks for exploration,” *CoRR*, vol. abs/1706.10295, 2017. [Online]. Available: <http://arxiv.org/abs/1706.10295>

- [25] S. Singh, A. G. Barto, and N. Chentanez, “Intrinsically Motivated Reinforcement Learning;,” Defense Technical Information Center, Fort Belvoir, VA, Tech. Rep., Jan. 2005. [Online]. Available: <http://www.dtic.mil/docs/citations/ADA440280>
- [26] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, “Curiosity-driven exploration by self-supervised prediction,” in *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 2017, pp. 2778–2787. [Online]. Available: <http://proceedings.mlr.press/v70/pathak17a.html>
- [27] A. Tamar, Y. WU, G. Thomas, S. Levine, and P. Abbeel, “Value iteration networks,” in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29. Curran Associates, Inc., 2016. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2016/file/c21002f464c5fc5bee3b98ced83963b8-Paper.pdf
- [28] V. Pong, S. Gu, M. Dalal, and S. Levine, “Temporal difference models: Model-free deep RL for model-based control,” in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: <https://openreview.net/forum?id=Skw0n-W0Z>
- [29] R. S. Sutton, D. Precup, and S. Singh, “Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning,” *Artificial Intelligence*, vol. 112, no. 1-2, pp. 181–211, Aug. 1999. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S0004370299000521>
- [30] R. Parr and S. Russell, “Reinforcement learning with hierarchies of machines,” in *Advances in Neural Information Processing Systems*, M. Jordan, M. Kearns, and S. Solla, Eds., vol. 10. MIT Press, 1997. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1997/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf
- [31] P.-L. Bacon, J. Harb, and D. Precup, “The option-critic architecture,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, Feb. 2017. [Online]. Available: <https://ojs.aaai.org/index.php/AAAI/article/view/10916>
- [32] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, A. P. Badia, K. M. Hermann, Y. Zwols, G. Ostrovski, A. Cain, H. King, C. Summerfield, P. Blunsom,

- K. Kavukcuoglu, and D. Hassabis, “Hybrid computing using a neural network with dynamic external memory,” *Nature*, vol. 538, no. 7626, pp. 471–476.
- [33] H. Dong, J. Mao, T. Lin, C. Wang, L. Li, and D. Zhou, “Neural logic machines,” in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: <https://openreview.net/forum?id=B1xY-hRctX>
- [34] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized experience replay,” in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: <http://arxiv.org/abs/1511.05952>
- [35] H. Van Hasselt, A. Guez, and D. Silver, “Deep Reinforcement Learning with Double Q-Learning,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, Mar. 2016. [Online]. Available: <https://ojs.aaai.org/index.php/AAAI/article/view/10295>
- [36] M. G. Bellemare, W. Dabney, and R. Munos, “A distributional perspective on reinforcement learning,” in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML’17. JMLR.org, 2017, p. 449–458.
- [37] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, “Rainbow: Combining improvements in deep reinforcement learning,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, Apr. 2018.
- [38] W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, and W. Dabney, “Revisiting fundamentals of experience replay,” in *Proceedings of the 37th International Conference on Machine Learning*, ser. ICML’20. JMLR.org, 2020.
- [39] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Machine Learning*, vol. 8, no. 3, pp. 229–256, May 1992. [Online]. Available: <https://doi.org/10.1007/BF00992696>
- [40] J. Peters and S. Schaal, “Reinforcement learning of motor skills with policy gradients,” *Neural Networks*, vol. 21, no. 4, pp. 682–697, May 2008. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S0893608008000701>

- [41] H. Liu, Y. Feng, Y. Mao, D. Zhou, J. Peng, and Q. Liu, “Action-depedent Control Variates for Policy Optimization via Stein’s Identity,” *International Conference on Learning Representations (ICLR)*, 2018.
- [42] J. Randløv and P. Alstrøm, “Learning to drive a bicycle using reinforcement learning and shaping,” in *Proceedings of the Fifteenth International Conference on Machine Learning*, ser. ICML ’98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, p. 463–471.
- [43] D. Ye, G. Chen, W. Zhang, S. Chen, B. Yuan, B. Liu, J. Chen, Z. Liu, F. Qiu, H. Yu, Y. Yin, B. Shi, L. Wang, T. Shi, Q. Fu, W. Yang, L. Huang, and W. Liu, “Towards playing full moba games with deep reinforcement learning,” in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS ’20. Red Hook, NY, USA: Curran Associates Inc., 2020.
- [44] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, “Dota 2 with large scale deep reinforcement learning,” *CoRR*, vol. abs/1912.06680, 2019. [Online]. Available: <http://arxiv.org/abs/1912.06680>
- [45] I. Popov, N. Heess, T. P. Lillicrap, R. Hafner, G. Barth-Maron, M. Vecerík, T. Lampe, Y. Tassa, T. Erez, and M. A. Riedmiller, “Data-efficient deep reinforcement learning for dexterous manipulation,” *CoRR*, vol. abs/1704.03073, 2017. [Online]. Available: <http://arxiv.org/abs/1704.03073>
- [46] M. Grzes and D. Kudenko, “Plan-based reward shaping for reinforcement learning,” in *2008 4th International IEEE Conference Intelligent Systems*, vol. 2, Sep. 2008, pp. 10–22–10–29, iSSN: 1941-1294. [Online]. Available: <https://ieeexplore.ieee.org/document/4670492>
- [47] R. Toro Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith, “Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning,” *Journal of Artificial Intelligence Research*, vol. 73, pp. 173–208, Jan. 2022. [Online]. Available: <https://jair.org/index.php/jair/article/view/12440>
- [48] J. Clark and D. Amodei, “Faulty reward functions in the wild.” [Online]. Available: <https://openai.com/index/faulty-reward-functions/>

- [49] A. Y. Ng, D. Harada, and S. J. Russell, “Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping,” in *Proceedings of the Sixteenth International Conference on Machine Learning*, ser. ICML ’99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Jun. 1999, pp. 278–287.
- [50] A. Eck, L.-K. Soh, S. Devlin, and D. Kudenko, “Potential-based reward shaping for finite horizon online POMDP planning,” *Autonomous Agents and Multi-Agent Systems*, vol. 30, no. 3, pp. 403–445, May 2016.
- [51] A. Laud and G. DeJong, “The influence of reward on the speed of reinforcement learning: an analysis of shaping,” in *Proceedings of the Twentieth International Conference on Machine Learning*, ser. ICML’03. Washington, DC, USA: AAAI Press, Aug. 2003, pp. 440–447.
- [52] O. Marom and B. Rosman, “Belief Reward Shaping in Reinforcement Learning,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, Apr. 2018, number: 1. [Online]. Available: <https://ojs.aaai.org/index.php/AAAI/article/view/11741>
- [53] N. Jaques, S. Gu, R. E. Turner, and D. Eck, “Tuning recurrent neural networks with reinforcement learning,” in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net, 2017. [Online]. Available: <https://openreview.net/forum?id=Syvv2e-Kx>
- [54] D. Lafleur, S. Chandar, and G. Pesant, “Combining Reinforcement Learning and Constraint Programming for Sequence-Generation Tasks with Hard Constraints,” in *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), C. Solnon, Ed., vol. 235. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 30:1–30:16.
- [55] C. Yin, Q. Cappart, and G. Pesant, “An Improved Neuro-Symbolic Architecture to Fine-Tune Generative AI Systems,” in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, B. Dilkina, Ed. Cham: Springer Nature Switzerland, 2024, vol. 14743, pp. 279–288, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-031-60599-4_19
- [56] D. Leslie and F. Rossi, “ACM TechBrief: Generative Artificial Intelligence,” Association for Computing Machinery, New York, NY, USA, Tech. Rep., 2023.

- [57] J. Pearl, “Reverend bayes on inference engines: A distributed hierarchical approach,” in *Proceedings of the National Conference on Artificial Intelligence, Pittsburgh, PA, USA, August 18-20, 1982*, D. L. Waltz, Ed. AAAI Press, 1982, pp. 133–136. [Online]. Available: <http://www.aaai.org/Library/AAAI/1982/aaai82-032.php>
- [58] B. Babaki, B. Omrani, and G. Pesant, “Combinatorial search in cp-based iterated belief propagation,” in *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, ser. Lecture Notes in Computer Science, H. Simonis, Ed., vol. 12333. Springer, 2020, pp. 21–36. [Online]. Available: https://doi.org/10.1007/978-3-030-58475-7_2
- [59] A. Burlats and G. Pesant, “Exploiting entropy in constraint programming,” in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 20th International Conference, CPAIOR 2023, Nice, France, May 29 - June 1, 2023, Proceedings*, ser. Lecture Notes in Computer Science, A. A. Ciré, Ed., vol. 13884. Springer, 2023, pp. 320–335.
- [60] P. Wang, P. L. Donti, B. Wilder, and J. Z. Kolter, “Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver,” in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 2019, pp. 6545–6554. [Online]. Available: <http://proceedings.mlr.press/v97/wang19e.html>
- [61] K. Ahmed, S. Teso, K. Chang, G. V. den Broeck, and A. Vergari, “Semantic probabilistic layers for neuro-symbolic learning,” in *NeurIPS*, 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/c182ec594f38926b7fcb827635b9a8f4-Abstract-Conference.html
- [62] J. Xu, Z. Zhang, T. Friedman, Y. Liang, and G. V. den Broeck, “A semantic loss function for deep learning with symbolic knowledge,” in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, ser. Proceedings of Machine Learning Research, J. G. Dy and A. Krause, Eds., vol. 80. PMLR, 2018, pp. 5498–5507. [Online]. Available: <http://proceedings.mlr.press/v80/xu18h.html>
- [63] K. Ahmed, E. Wang, K. Chang, and G. V. den Broeck, “Neuro-symbolic entropy regularization,” in *Uncertainty in Artificial Intelligence, Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence, UAI 2022, 1-5 August*

- 2022, *Eindhoven, The Netherlands*, ser. Proceedings of Machine Learning Research, J. Cussens and K. Zhang, Eds., vol. 180. PMLR, 2022, pp. 43–53. [Online]. Available: <https://proceedings.mlr.press/v180/ahmed22a.html>
- [64] Z. Yang, J. Lee, and C. Park, “Injecting logical constraints into neural networks via straight-through estimators,” in *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvári, G. Niu, and S. Sabato, Eds., vol. 162. PMLR, 2022, pp. 25 096–25 122. [Online]. Available: <https://proceedings.mlr.press/v162/yang22h.html>
- [65] D. Chen, Y. Bai, W. Zhao, S. Ament, J. Gregoire, and C. Gomes, “Deep reasoning networks for unsupervised pattern de-mixing with constraint reasoning,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 1500–1509.
- [66] Y. Bai, D. Chen, and C. P. Gomes, “Clr-drnets: Curriculum learning with restarts to solve visual combinatorial games,” in *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.
- [67] M. Defresne, S. Barbe, and T. Schiex, “Scalable coupling of deep learning with logical reasoning,” in *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. ijcai.org, 2023, pp. 3615–3623.
- [68] Z. Yang, A. Ishay, and J. Lee, “Neurasp: Embracing neural networks into answer set programming,” in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, C. Bessiere, Ed. ijcai.org, 2020, pp. 1755–1762. [Online]. Available: <https://doi.org/10.24963/ijcai.2020/243>
- [69] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, and L. D. Raedt, “Neural probabilistic logic programming in deepproblog,” *Artif. Intell.*, vol. 298, p. 103504, 2021. [Online]. Available: <https://doi.org/10.1016/j.artint.2021.103504>
- [70] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [71] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.

- [72] W. J. van Hoeve, G. Pesant, and L. Rousseau, “On global warming: Flow-based soft global constraints,” *J. Heuristics*, vol. 12, no. 4-5, pp. 347–373, 2006. [Online]. Available: <https://doi.org/10.1007/s10732-006-6550-4>
- [73] M. S. Cuthbert and C. Ariza, “Music21: A toolkit for computer-aided musicology and symbolic music data.” in *ISMIR*, J. S. Downie and R. C. Veltkamp, Eds. International Society for Music Information Retrieval, 2010, pp. 637–642. [Online]. Available: <http://dblp.uni-trier.de/db/conf/ismir/ismir2010.html#CuthbertA10>
- [74] S. Demassey, G. Pesant, and L.-M. Rousseau, “A Cost-Regular Based Hybrid Column Generation Approach,” *Constraints*, vol. 11, no. 4, pp. 315–333, Nov. 2006.
- [75] G. Pesant, “The MiniCPBP Solver,” <https://github.com/PesantGilles/MiniCPBP>.
- [76] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, “Curriculum learning,” in *Proceedings of the 26th Annual International Conference on Machine Learning*. Montreal Quebec Canada: ACM, Jun. 2009, pp. 41–48. [Online]. Available: <https://dl.acm.org/doi/10.1145/1553374.1553380>