

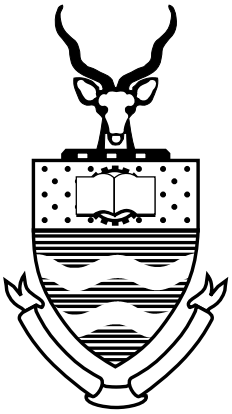
https://dspace.org/

Efficient curriculum generation for reinforcement learning

Dunn, Leroy

2021
https://hdl.handle.net/10539/32449

Downloaded from a DSpace repository. https://dspace.org/



UNIVERSITY OF THE WITWATERSRAND

EFFICIENT CURRICULUM GENERATION FOR REINFORCEMENT LEARNING

Research Dissertation

Leroy DUNN
744446

Supervised by Prof Benjamin Rosman and Dr Pravesh Ranchod

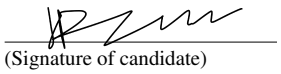
A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science.

June 21, 2021

Declaration

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the degree of Master of Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other university.



(Signature of candidate)

21 day of June 20 21 at Midrand

Abstract

Complex tasks involving large state spaces are difficult for Reinforcement Learning agents to solve since the agents learn by trial and error, requiring a large amount of interaction with their environment. Curriculum Learning in Reinforcement Learning aims to accelerate training or improve performance of a Reinforcement Learning agent on a complex target task by training the agent on a sequence of intermediate tasks (a curriculum) beforehand. Current automated approaches to generate a curriculum typically treat the cost of curriculum generation as a sunk cost since it is challenging to assess how much time, effort, and prior knowledge is used to design a curriculum, and focus instead on evaluating the curricula. This is a limitation when aiming to accelerate agent training since the time to generate a curriculum may outweigh the time to directly learn the target task.

We study the cost of curriculum generation with the objective to efficiently design high-quality curricula. We propose a simple measure of the cost of curriculum generation (the total time of intermediate training) and introduce a novel curriculum generation algorithm to minimize this cost. Our algorithm quantifies and learns task utility online to rank candidate curricula and excels against comparable curriculum generation methods when utilizing the proposed measure. We evaluate our algorithm in two domains and show that it generates high-quality curricula under time restrictions as compared to baseline methods and is faster than baseline methods to generate an optimal curriculum of a curriculum search space.

Acknowledgements

I would like to thank my supervisors, Prof Benjamin Rosman and Dr Pravesh Ranchod, as well as everyone at the Robotics, Autonomous Intelligence and Learning (RAIL) lab at the University of Witwatersrand for their gracious support and sincere guidance. This work is the culmination of knowledge and effort beyond that of the authors, and words cannot describe my level of gratitude towards everyone who has assisted in it.

Contents

Declaration i

Abstract ii

Acknowledgements iii

1 Introduction **1**

2 Reinforcement Learning **5**

2.1 Reinforcement Learning Objective 5

2.2 Markov Decision Process 6

2.3 Agent Policy 7

2.4 Value Function 7

2.5 Q Function 8

2.6 Learning Methods 9

2.6.1 Model-based and Model-free Methods 9

2.6.2 Q-Learning 9

2.6.3 SARSA 11

2.7 Exploration Strategies 12

2.7.1 Episilon Greedy Exploration 12

2.7.2 Other Exploration Strategies 12

2.8 Function Approximation 13

2.8.1 Tile Coding 13

2.9 Conclusion 14

3 Transfer Learning **15**

3.1 Knowledge Transfer Methods 15

3.2 Evaluation Metrics for Transfer Learning 16

3.3 Conclusion 18

4 Curriculum Learning **19**

4.1 Sample Sequencing 20

4.2 Co-learning 21

4.3 Reward Function, Initial State, and Terminal State Changes 22

4.4 MDP-based Sequencing 23

4.5 Combinatorial Optimization and Search 23

4.6 Graph-based Sequencing 24

4.7 Limitations 25

4.8 Conclusion 25

5 Problem Definition and Methodology **27**

5.1 Budgeted Curriculum Generation 27

5.2 Heuristic Curriculum Search 29

5.3 Conclusion 32

6 Experimental Results **33**

6.1 Block Dude 33

6.1.1 Task Description 34

6.1.2 Agent Description 34

6.2 Mini Grid 35

6.2.1 Task Description 35

6.2.2 Agent Description 37

6.3 Source Task Utility 37

6.4 Heuristic and Graph Traversal Algorithm Exploration 38

6.4.1 Graph Traversal Algorithms 38
6.4.2 Heuristic Functions 39
6.4.3 Results 40
6.5 Efficient Curriculum Generation 42
6.5.1 Baselines 42
6.5.2 Block Dude 43
6.5.3 Mini Grid 44
6.5.4 Analysis 45
6.6 Curriculum Search Space Statistics 47
6.7 Conclusion 48

7 Conclusion and Future Work **49**

Bibliography **51**

Chapter 1

Introduction

Reinforcement Learning (RL) agents have become increasingly popular for solving large and complex tasks [Narvekar *et al.* 2020]. They learn by trial and error, requiring a large amount of interaction with their environment. This requirement can be prohibitively expensive in realistic scenarios with large, continuous state spaces. Transfer Learning [Taylor and Stone 2009; Lazaric 2012] addresses this problem by transferring agent knowledge after learning a task, to an agent learning another task, for the purpose of accelerated or improved learning. Curriculum Learning [Bengio *et al.* 2009] shows how this idea of knowledge transfer can be improved by transferring agent knowledge across multiple tasks of increasing complexity (a curriculum), enabling an agent to incrementally solve modern, complex tasks. An example use of Curriculum Learning, depicted in Figure 1.1, is to teach an agent to play the game of Chess. The agent could be guided to learn independent skills by, for instance, learning subtasks of the game of Chess involving fewer Chess pieces and/or a smaller game board. It would then be challenged to expand its knowledge by solving increasingly difficult subtasks involving more Chess pieces and a larger game board. The agent would continue to train on intermediate tasks (a curriculum) until it has sufficient knowledge to effectively learn to play the complete game of Chess. The challenge of Curriculum Learning therefore, is to determine how to create and sequence a set of intermediate tasks into a curriculum for an agent to accelerate or improve its performance on some complex task. A recent study [Narvekar *et al.* 2020] suggests that many landmark results in Reinforcement Learning, from TD-Gammon [Tesauro 1995] to AlphaGo [Silver *et al.* 2016] have implicitly used curricula to guide training by, for example, reordering samples in a task using methods such as experience replay [Schaul *et al.* 2016], or by cooperation or competition between two agents which brings about a sequence of increasingly challenging tasks. Curriculum Learning is therefore likely to be useful for future AI challenges.

There are two main approaches which address the challenges of Curriculum Learning: task creation for a curriculum [Schmidhuber 2013; Narvekar *et al.* 2016; Silva and Costa 2018], and task sequencing of a predefined set of tasks [Florensa *et al.* 2018a; Narvekar *et al.* 2017; Svetlik *et al.* 2017]. The first approach, task creation, is concerned with how to create useful intermediate tasks for an agent, based on properties of the agent or task, or the agent's learning progress. The second approach, task sequencing, requires a set of (possibly handcrafted) tasks be selected and ordered most beneficially for an agent to solve more challenging tasks. We focus on the area of task sequencing to improve the efficiency of the sequencing process and enable researchers to rapidly solve complex problems in

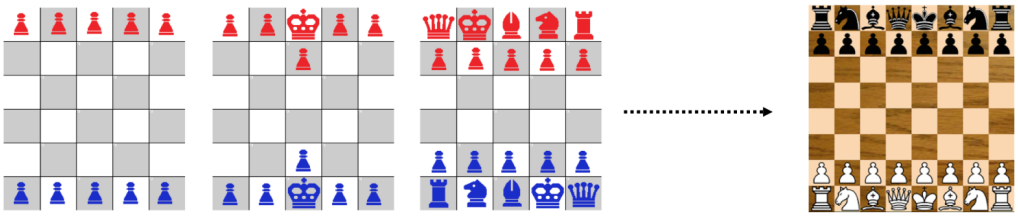


Figure 1.1: Subtasks of the game of Chess involving fewer Chess pieces and a smaller game board, and used to form a curriculum for learning the full game of Chess [Narvekar *et al.* 2020]

Reinforcement Learning.

Task sequencing methods typically treat the cost of curriculum generation as a sunk cost since it is challenging to assess how much time, effort, and prior knowledge is used to design a curriculum. This assessment is especially difficult for methods involving expert knowledge or intervention from humans. Instead, these methods focus on evaluating curricula by measuring the resultant performance change of an agent on some target task after the agent is trained on the curriculum. Treating the process of curriculum generation as a sunk cost is a limitation of many Curriculum Learning approaches, particularly for the use case of accelerating agent training, since the time to generate a curriculum may outweigh the time to directly learn the target task.

We study the cost of curriculum generation with the objective to efficiently design high-quality curricula. We define the quality of a curriculum as the resultant agent performance change on some task after the agent has trained on the curriculum. In studying the cost of curriculum generation we identify two use cases for efficient curriculum design. The first use case is when one may wish to generate a curriculum within a time limit and trade off optimal agent performance for the time required to generate a curriculum. The second use case is the application of Curriculum Learning to help an agent solve a class of target tasks as opposed to a single target task, thereby amortizing the cost of curriculum generation over multiple target tasks and advocating for the advancement of general intelligence [Insa-Cabrera *et al.* 2011].

We provide two contributions that address the challenge of assessing the cost or curriculum generation and provide a solution to the aforementioned use cases. Our first contribution is a simple measure of the cost of curriculum generation - the total time of all intermediate training. We ignore factors relating to human expertise for the sake of simplicity, which enables us to propose and directly compute the measure. This measure is ideal where little to no human intervention is required, such as the context of autonomous task sequencing. The efficiency of curriculum generation is then defined according to our proposed metric: efficient curriculum generation minimizes the intermediate training time required to create a curriculum. Our second contribution is a curriculum generation algorithm designed to minimize this proposed cost of curriculum generation. Our algorithm operates by learning online the utility of individual tasks and ranking candidate curricula by the aggregated utility of the tasks they comprise. It can be used with a time restriction to generate high-quality curricula and we show that time restrictions marginally degrade the quality of our algorithm's generated curricula. Moreover, it can be used to help an agent solve a class of target tasks as opposed to a single target task. We evaluate our algorithm in the Block Dude domain with a single target task and in the Mini Grid domain with a class of target tasks and show that it generates high-quality curricula under time restriction and is

faster than baseline methods to generate an optimal curriculum of a curriculum search space.

The remainder of this thesis is structured as follows. In Chapters 2 and 3 we detail background concepts related to our work, Reinforcement Learning and Transfer Learning respectively. In Chapter 4 we discuss related Curriculum Learning literature. In Chapter 5 we formalize our objective to limit the cost of curriculum generation and generate high-quality curricula under time restriction. We begin by formalizing the definition of a curriculum and the cost of curriculum generation in Section 5.1. We then formulate the problem of curriculum generation as a search problem and introduce a curriculum generation algorithm to search and evaluate a space of candidate curricula in Section 5.2. In Chapter 6 we perform experimental evaluation. We begin in Sections 6.1 and 6.2 by detailing the domains used in our experimentation, Block Dude and Mini Grid respectively. We then quantify the utility of individual source tasks for an agent and show how certain source tasks may be biased to create good curricula in Section 6.3. Next, we evaluate both common and handcrafted heuristics and graph traversal algorithms for our curriculum generation algorithm in Section 6.4. We then evaluate our algorithm using the best heuristic and graph traversal combination, and demonstrate that it outperforms benchmarks methods for our proposed cost of curriculum generation metric in Section 6.5. Lastly, in Chapter 7 we summarize the contribution of this research and provide ideas for future work.

Chapter 2

Reinforcement Learning

Reinforcement Learning (RL) has become increasingly popular to solve complex AI problems. For instance, it achieved superhuman performance in the game Go [Silver *et al.* 2016], estimated to have 10^{174} game states, more than the total number of atoms in the universe, as well as other landmarks in machine learning [Tesauro 1995; Vinyals *et al.* 2019]. However, modern AI problems have become progressively more difficult to solve with traditional RL methods, and the methods therefore require improvement to continue to solve these problems. Before addressing potential improvements to RL methods, in this chapter we provide a brief introduction to Reinforcement Learning, an area of machine learning wherein agents learn to perform tasks through interaction with an environment.

In Section 2.1 we present the objective of Reinforcement Learning and its relation to other machine learning techniques. In Section 2.2 we present the Markov Decision Process, a process used to model a Reinforcement Learning environment. In Section 2.3 we present the agent policy, an action-selection strategy in the environment. In Sections 2.4 and 2.5 we present the value function and Q-function respectively, used to approximate the value of environment states. In Section 2.6 we present two types learning methods, model-based and model-free methods, and detail the learning methods used in our work. In Section 2.7 we present common exploration methods used by the learning methods. Lastly, in Section 2.8, we present function approximation, a technique to work with large or continuous state spaces.

2.1 Reinforcement Learning Objective

Reinforcement Learning is a field of Machine Learning wherein an agent learns to solve a task by learning to behave optimally in an environment. The task objective is typically encoded using a numerical reward or feedback signal that the agent receives after executing an action in the environment. Learning optimal behaviour, and hence solving the task optimally, is therefore a sequential decision-making problem where an agent is required to take actions that maximize the sum of the numerical rewards it receives.

In Reinforcement Learning, training data is sequentially generated through the agent's repeated interaction with the environment. This is unlike other machine learning fields such as Supervised Learning

or Unsupervised Learning, where training data for the task objective is provided as input to the learning process. Furthermore, Reinforcement Learning is suited to solve sequential decision problems wherein a decision-making entity (the agent) attempts to optimize a sequence of decisions.

2.2 Markov Decision Process

A Reinforcement Learning environment is typically modelled as a Markov Decision Process (MDP) [Puterman 1990] $m = (S, A, p, r, \gamma)$ consisting of the following components:

- State Space S

  The state space S is the set of states the agent can occupy in the environment at any time t.

- Action Space $A(s)$

  The action space $A(s)$ is the set of actions available to the agent in each state $s \in S$. In simplistic environments it is common that all actions are available to the agent in every state. Reinforcement Learning agents use actions to transition between states in the environment.

- State Transition Probability Function $p(s, a, s')$

  The state transition probability function, or transition function, returns the probability of an agent transitioning to a state s' after taking an action a in state s. In deterministic environments, state transitions are deterministic and the output of this function is either 0 or 1. In non-deterministic or stochastic environments, the transition function is defined as a probability distribution over the set of states reachable from state s after taking action a.

- Reward function $r(s, a, s')$

  The reward function returns a numerical reward for taking action a in state s and transitioning to state s'. It typically encodes the task objective and is configured accordingly. The agent's objective is to maximise the sum of the numerical rewards it receives.

- Discount factor γ

  The discount factor $\gamma \in [0, 1)$ is applied to reward signals over time and is used in the determination of the value of future rewards.

If the aforementioned components are constrained to obey the Markov Property [Puterman 1990], that is, if the next state of the environment can be determined solely by the current state (Equation 2.1) then the environment model is referred to as a Markov Decision Process.

$$P(s_{t+1}|s_t) = P(s_{t+1}|s_t, s_{t-1}, s_{t-2}, \ldots, s_{t_0}) \tag{2.1}$$

2.3 Agent Policy

A Reinforcement Learning agent's policy, $\pi : S \mapsto A$ [Sutton and Barto 1999], is an action selection strategy in the environment. The agent interacts with its environment at each time step t by observing the current state s_t, choosing an action a_t according to its policy, and receiving a numerical reward r_t before transitioning to a new state s_{t+1}, depicted in Figure 2.1. Consequently, each environment interaction produces a training sample $\langle s, a, r, s' \rangle$. The goal of the agent is to learn an optimal policy π^*, which maximizes its episodic reward $G = \sum_{t=1}^{\infty} \gamma^{t-1} r_t$. A learning episode starts at some initial state s_0 and terminates when the agent reaches some terminal state s_f or after some maximum number of time steps t_f. The action space available to the agent is either discrete, where the agent selects an action from a finite set, or continuous, where the agent selects an action from an infinite set, typically by selecting a real-valued number. A policy $\pi(a|s)$ is therefore given by a probability distribution over all actions available to the agent in state s. A trajectory $\tau = s_0, a_0, s_1, a_1, \ldots, s_f, a_f$ is the result of transitioning between states using actions selected by a policy, beginning at the starting state and ending at a terminal state or after some maximum number of time steps.

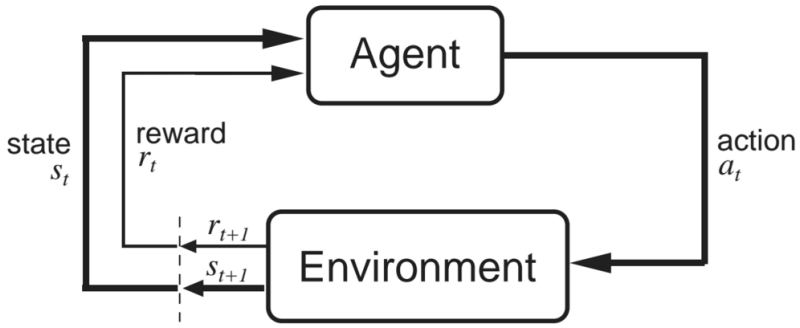


Figure 2.1: Agent-Environment interaction in Reinforcement Learning [Sutton and Barto 1999]

2.4 Value Function

To compare policies and learn an optimal policy we use the notion of a value function [Sutton and Barto 1999]. A value function, $V_\pi(s_t)$, assigns a value of a state s_t under policy π equal to the discounted sum of the future rewards observed when starting in the state and selecting actions from the policy thereafter:

$$V_\pi(s_t) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \tag{2.2}$$

where $T = \infty$ for continuing tasks (tasks that require the agent to act indefinitely to solve the task), or $T = t_f \in \mathbb{N}$ for episodic tasks (tasks that that require the agent to solve the task before a finite number of time steps).

Incorporating the state transition probabilities into the value function and writing it recursively yields

the Bellman equation [Sutton and Barto 1999]:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi \left(\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid s_t = s \right) \\ &= \mathbb{E}_\pi \left(r_{t+1} + \gamma \sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid s_t = s \right) \\ &= \sum_{a \in A(s)} \pi(s,a) \sum_{s'} p(s,a,s') \left[r(s,a,s') + \gamma \mathbb{E}_\pi \left(\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid s_t = s' \right) \right] \\ &= \sum_{a \in A(s)} \pi(s,a) \sum_{s'} p(s,a,s') \left[r(s,a,s') + \gamma V_\pi(s') \right] \end{aligned} \tag{2.3}$$

A policy π^* is optimal if it yields the best value for all states:

$$\pi^* = \arg\max_\pi V_\pi(s), \forall s \in S \tag{2.4}$$

The optimal value function $V^*(s) = V_{\pi^*}(s)$ is the value function of the optimal policy and is alternatively defined as:

$$V^*(s) = \max_{a \in A(s)} \sum_{s'} p(s,a,s') \left[r(s,a,s') + \gamma V^*(s') \right] \tag{2.5}$$

2.5 Q Function

When some components of the environment's MDP are not available (a complete model of the environment is not available) it is often easier to learn the optimal policy using the action-value function $Q_\pi(s,a)$ [Sutton and Barto 1999] which represents the expected outcome of taking action a in state s and thereafter following policy π:

$$Q(s,a) = \sum_{s'} p(s,a,s') \left[r(s,a,s') + \gamma V_\pi(s') \right] \tag{2.6}$$

The optimal action-value function Q^* can be written recursively:

$$Q^*(s,a) = \sum_{s'} p(s,a,s') \left[r(s,a,s') + \gamma \max_{a' \in A} Q^*(s',a') \right] \tag{2.7}$$

The optimal policy can then obtained by greedily selecting actions from the optimal action-value function which result in the highest q value in each state:

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s, a) \tag{2.8}$$

2.6 Learning Methods

To learn an agent policy we make use of Reinforcement Learning (RL) algorithms. We begin by presenting two classes of Reinforcement Learning algorithms in Section 2.6.1. We then detail common algorithms which are used in our work, Q-Learning and SARSA, in Sections 2.6.2 and 2.6.3 respectively.

2.6.1 Model-based and Model-free Methods

A model-based RL algorithm is an algorithm that uses the environment dynamics (a transition function and reward function) to learn a policy. The agent either has access to the environment dynamics, but more commonly, the agent learns the environment dynamics through its interaction with the environment, and uses this approximate model to learn a policy. However, when an approximate model is learned, the optimal policy may never be found because of the potential inaccuracy of the approximate model. Some examples of model-based algorithms are the dynamic programming algorithms, Value Iteration [Sutton and Barto 1999] and Policy Iteration [Sutton and Barto 1999].

A model-free RL algorithm is an algorithm that estimates the optimal policy without using or approximating the environment dynamics. In practice, a model-free algorithm either estimates a value function or a policy through interaction with the environment.

Model-free Reinforcement Learning algorithms can further be classified as either off-policy or on-policy. Off-policy algorithms update the agent's policy solely according to actions selected by an exploration strategy (Section 2.7), and not actions sampled from the agent's policy. Conversely, on-policy algorithms utilize actions sampled from the current agent policy to update the agent's policy. We present both off-policy and on-policy model-free algorithms used in our work, Q-Learning and SARSA respectively, in the subsequent sections.

2.6.2 Q-Learning

One widely-used, model-free, off-policy Reinforcement Learning algorithm is Q-learning [Sutton and Barto 1999]. We detail it below.

The Q-learning algorithm, Algorithm 1, takes as input an initial action-value function $Q(s, a)$ with default values initialized for every state-action pair, the learning environment e, and a learning rate α used to weigh updates to the action-value function. The algorithm initializes a variable, *episodeNum* (line 1), used to keep track of the number of episodes that have passed. It then begins an episode loop (line 2), iterating over episodes, and initializes the current episode by resetting the environment (line 3), incrementing the current episode number (line 4), setting the initial state (line 5), and initializing a state counter, *stateNum* (line 6). The state loop (line 7) then begins, iterating over states of an

Input: Environment e, Initialized action-value function $Q(s, a)$, Learning Rate α

```
1:  episodeNum ← −1
2:  while true do
3:    e.resetEpisode
4:    episodeNum ← episodeNum + 1
5:    s ← e.initialState
6:    stateNum ← 0
7:    while true do
8:      a ← SelectAction(Q, s)
9:      (r, s′) ← e.act(s, a)
10:     Q(s, a) ← Q(s, a) + α[r + γ max_a′ Q(s′, a′) − Q(s, a)]
11:     s ← s′
12:     stateNum ← stateNum + 1
13:     if IsTerminalState(e, s, stateNum) then
14:       break
15:     end if
16:   end while
17:   if IsTerminalEpisode(Q, episodeNum) then
18:     break
19:   end if
20: end while
21: return Q
```

Algorithm 1: `Q-Learning`

episode. The first step of the state loop is for the learning agent to select an action to take in the environment (`SelectAction` on line 8). We discuss various exploration strategies that can be applied across different learning agents in Section 2.7. The agent then acts in the environment and observes a reward and the next state (line 9) before updating the action-value function (line 10). The equation used to update the action-value function is the key element of the Q-learning algorithm. Finally, the current state is updated (line 11), the state counter is incremented (line 12), and the state loop ends if the current state is a terminal state (`isTerminalState` line 13). In practice, a state is defined as a terminal state by the environment definition or if a certain number of states has exceeded. After the state loop has concluded, the episode loop ends if the current episode is deemed terminal (`isTerminalEpisode` line 17). In practice, an episode is considered terminal if a certain number of episodes has exceeded, or if the action-value function has converged, that is, if the values of the action-value function have not changed by some delta value over some number of episodes.

For small, discrete environments the action-value function is typically stored in a Q-table: a table data structure where each cell stores the Q-value for the corresponding state row and action column (Table 2.1). For large or continuous environments, it may be infeasible to use a Q-table to store the action-value function because of computing limitations involving storage space or processing speed. Instead, function approximation, presented in Section 2.8, is used to encode states, effectively transforming the state space to one that is smaller in size.

		Action		
		a_1	a_2	a_3
State	s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$	$Q(s_1, a_3)$
	s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$	$Q(s_2, a_3)$
	s_3	$Q(s_3, a_1)$	$Q(s_3, a_2)$	$Q(s_3, a_3)$

Table 2.1: Example Q-table

2.6.3 SARSA

Another widely used, model-free, on-policy Reinforcement Learning algorithm is SARSA [Sutton and Barto 1999]. Its name is derived as an acronym for the parameters it requires to update the action-value function: current state (s), current action (a), current reward (r), next state (s'), and next action (a'). It is near identical to the Q-learning algorithm except for it being an on-policy algorithm - it updates the action-value function by relying on actions sampled from the current learned policy.

Input: Environment e, Initialized action-value function $Q(s, a)$, Learning Rate α

```
1:  episodeNum ← −1
2:  while true do
3:      e.resetEpisode
4:      episodeNum ← episodeNum + 1
5:      s ← e.initialState
6:      stateNum ← 0
7:      while true do
8:          a ← SelectAction(Q, s)
9:          (r, s′) ← e.act(s, a)
10:         a′ ← SelectAction(Q, s′)
11:         Q(s, a) ← Q(s, a) + α[r + γQ(s′, a′) − Q(s, a)]
12:         s ← s′
13:         stateNum ← stateNum + 1
14:         if IsTerminalState(e, s, stateNum) then
15:             break
16:         end if
17:     end while
18:     if IsTerminalEpisode(Q, episodeNum) then
19:         break
20:     end if
21: end while
22: return Q
```

Algorithm 2: SARSA

The SARSA algorithm is identical to that of the Q-learning algorithm except for how it updates the action-value function in an on policy manner, depicted in Algorithm 2 on lines 10-11.

2.7 Exploration Strategies

Exploration strategies are an essential component of RL algorithms. RL algorithms typically initially do not know how the environment will react according to their actions (they do not know which state they will transition to after executing an action and the reward signal they will receive as a result of the transition). To achieve their objective of obtaining an optimal cumulative reward, RL agents therefore face a dilemma about whether to take advantage of the best known action in each state (exploitation) or to collect information about unknown, potentially better, actions (exploration). The dilemma arises since best known actions could be suboptimal and executing them would result in a suboptimal cumulative reward, but best known actions could in fact be optimal, and executing exploratory actions would then result in a suboptimal cumulative reward. Exploration strategies attempt to address this dilemma by gathering sufficient information about actions before exploiting that information. In practice, agents typically begin with more exploratory actions since little to nothing is known about the environment. They then increasingly perform more exploitation actions as they learn about the environment, resulting in the learned policy getting closer to optimality. We discuss an exploration strategy used in our work, Epsilon Greedy Exploration, in Section 2.7.1 and give an overview of other exploration strategies in Section 2.7.2.

2.7.1 Episilon Greedy Exploration

Using Epsilon Greedy Exploration (ϵ-greedy Exploration) [Sutton and Barto 1999] an agent selects actions for exploration and exploitation according to a parameter $\epsilon \in [0, 1]$ such that it chooses a random action with probability ϵ (exploration) and chooses the action which is most likely to transition the agent to the state with the highest state value (exploitation) with probability $1 - \epsilon$. The ϵ parameter is typically gradually decreased throughout the training process after each training episode. The result of which is high exploration during initial training episodes and high exploitation during final training episodes. Gradually decreasing the ϵ parameter has two effects: it helps the agent learn more effectively by balancing exploration and exploitation, and it ensures the agent's policy converges, ideally to the optimal policy.

2.7.2 Other Exploration Strategies

More advanced exploration strategies include count-based exploration where state visitation or action counts are stored during learning and are used to bias agents towards infrequently visited states and infrequently performed actions [Thrun and Möller 1991; Thrun *et al.* 1991; Rosman and Ramamoorthy 2012; Tang *et al.* 2017]. Another approach is intrinsic motivation where agents explore based off a function of state novelty or a curiosity model [Baldassarre and Mirolli 2013; Pathak *et al.* 2017; Burda *et al.* 2018].

2.8 Function Approximation

For domains with large state spaces, which is often the case in real-world problems, finding a solution to the Bellman Equation (Equation 2.3) and learning an optimal agent policy is intractable. Furthermore, it may be infeasible to store the state space of a large domain in memory. To overcome this issue, the state space can be approximated using a technique called function approximation [Russell and Norvig 2002].

Function approximation approximates the state space with a set of features ϕ whose dimension is smaller than the size of the state space. The most common method of function approximation is to use a weighted linear combination of features to represent the value function:

$$V(s) \approx \sum_{i=1}^{N} \theta_i \phi_i(s) = \theta^T \phi(s) \tag{2.9}$$

where $\phi(s)$ produces a feature vector from state s and θ is a learned parameter vector.

Another benefit of function approximation is its ability to generalise knowledge to states not yet encountered. This is possible since state similarity is implicitly computed according to features, and knowledge from seen states can be applied to similar unseen states. For this reason function approximation can be used even in smaller, simpler domains to improve the generalization ability of the agent and hence the knowledge transfer ability of an agent.

In the next section we discuss the function approximation technique, Tile Coding, used in our work and that of other Curriculum Learning literature [Narvekar *et al.* 2017; Foglino *et al.* 2019b]. We make use of Tile Coding because of its successful application in other curriculum Literature and to focus our study on the cost of curriculum generation.

2.8.1 Tile Coding

Bucketization [Sutton and Barto 1999] is a common technique to approximate large or continuous state spaces. It works by dividing the state space into multiple, typically equally sized, regions (buckets), and transforming a state into a binary feature vector ($\phi(s)$) whose elements are zero except the element corresponding to the bucket where the state value lies. Tile coding [Sutton and Barto 1999] improves this technique by dividing the state space multiple times where each division is uniquely offset from the others, depicted in Figure 2.2. Laying multiple divisions over the state space improves the generalization ability of an agent since the similarity of states can be identified according to the number of shared buckets that the states occupy, and knowledge from seen states can be applied to similar unseen states. Tile coding results in several divisions (tilings) over the state space that transform a state into a binary feature vector ($\phi(s)$) whose elements are zero except for the bucket (tile) in each division where the state value falls into. The resultant binary feature vector has a size equal to the total number of tiles in all tilings, and has n number of non-zero elements, where n is the number of tilings over the state space.

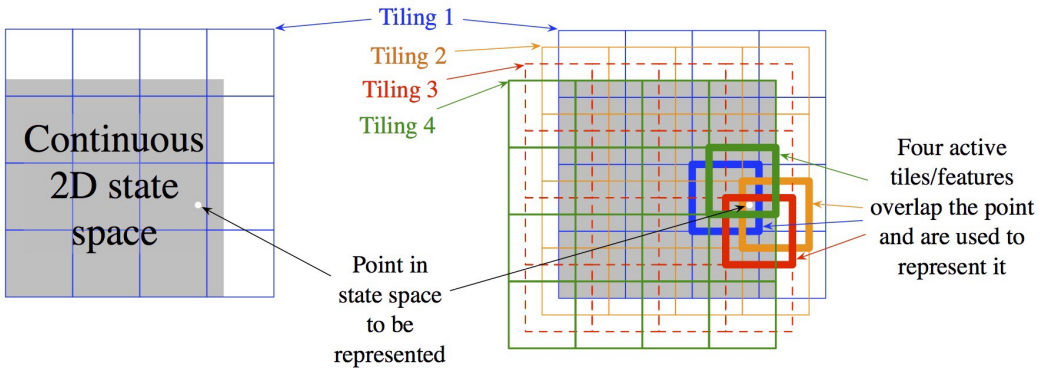


Figure 2.2: An example of tile coding over a 2D continuous state space [Sutton and Barto 1999]

2.9 Conclusion

In this chapter we discussed Reinforcement Learning (RL), a machine learning method designed to teach agents to perform a task through interaction with an environment. We began by examining various RL definitions and demonstrated how RL algorithms arise from these definitions. We then detailed common exploration strategies used by RL algorithms, and a method, function approximation, used to approximate an infeasibly large or continuous state space.

Chapter 3

Transfer Learning

In the standard Reinforcement Learning setting, learning a particularly challenging task can be slow or impossible. Transfer Learning (TL) [Taylor and Stone 2009; Lazaric 2012] addresses this problem by transferring agent knowledge across learned tasks for the purpose of accelerated or improved learning. In transfer learning, an agent learns to solve a **target task** by transferring knowledge across one or more previously learned **source tasks**.

In this chapter we provide a brief introduction to Transfer Learning. We present knowledge transfer methods in Section 3.1 and metrics used to measure the effectiveness of Transfer Learning in Section 3.2. We utilize both the knowledge transfer methods and metrics to compose and evaluate the methods in Chapter 6.

3.1 Knowledge Transfer Methods

Transfer Learning methods have been used to transfer knowledge between task MDPs that have different transition functions [Selfridge *et al.* 1985; Perkins *et al.* 1999], state spaces [Andre and Russell 2002; Ferguson and Mahadevan 2006], start states [Asada *et al.* 1994; Fernández and Veloso 2006], terminal states [Foster and Dayan 2002; Wilson *et al.* 2007], reward functions [Mehta *et al.* 2008; Asadi and Huber 2007], and/or action sets [Soni and Singh 2006; Talvitie and Singh 2007].

The knowledge transferred can fall into the categories of low-level or high-level knowledge. Low-level knowledge is used to directly initialize an agent on some target task whereas high-level knowledge only partially initializes the agent and is used to help guide exploration in the target task. Specifically, knowledge transferred can take the form of low-level knowledge, such as training samples, agent policies, task models, or value functions. Alternatively, it can take the form of high-level knowledge such as options. We discuss each knowledge transfer type in further detail below:

1. Samples:

   Training samples on a target task are reordered to improve learning [Lazaric *et al.* 2008; Lazaric and Restelli 2011; Schaul *et al.* 2016], for instance, by prioritizing and replaying training sam-

ples most expected to most improve learning progress [Schaul *et al.* 2016].

2. Policies

   A learned agent policy is transferred by initializing the policy parameters of a new agent to that of the learned policy [Fernández *et al.* 2010].

3. Models

   Model-based RL algorithms use the parameters of learned models to initialize the models of other model-based agents [Fachantidis *et al.* 2013; Marom and Rosman 2018].

4. Value Functions

   Value function transfer initializes the action-value or state-value function of an agent learning some target task using the parameters of an action-value or state-value function learned in a source task [Taylor *et al.* 2007].

5. High Level Actions:

   A high level action is a generalization of an action, used to express the idea that certain actions are composed of other sub-actions. One popular example is that of an option [Sutton *et al.* 1999], which is formally composed of a policy, a termination condition, and an initiation set. For instance the option "take a freekick" could involve a sequence of other actions, like "face the ball", "run towards the ball", "kick the ball" etc. Agents could therefore learn options independently and transfer option knowledge in the form of an option policy, a termination condition, and an initiation set [Soni and Singh 2006; James *et al.* 2020].

3.2 Evaluation Metrics for Transfer Learning

Several metrics have been proposed to evaluate Transfer Learning algorithms [Taylor and Stone 2009]. These metrics compare the learning trajectory of the agent on the target task with and without the transfer of prior knowledge. We describe these metrics below with reference to Figure 3.1.

The regret metric, P_r, is a comprehensive indicator of learning speed and optimally balances agent exploration and exploitation, requiring agents to minimize suboptimal actions. It is defined with respect to a specified performance threshold, g, as follows:

$$P_r = -\left(Ng - \sum_{t=1}^{N} G_t\right) \tag{3.1}$$

where N is the number of episodes executed in the target task, and $Ng - \sum_{t=1}^{N} G_t$ is the difference between the agent obtaining an episodic reward of g in every episode and its actual performance. This difference is the regret with respect to a policy achieving an episodic reward g, and is intended to be minimized, therefore it is multiplied by -1 (lower regret results in a higher value of the regret metric). Setting $g = 0$ can be a convenient starting point, making the regret metric equivalent to the total reward accumulated during training on the target task.

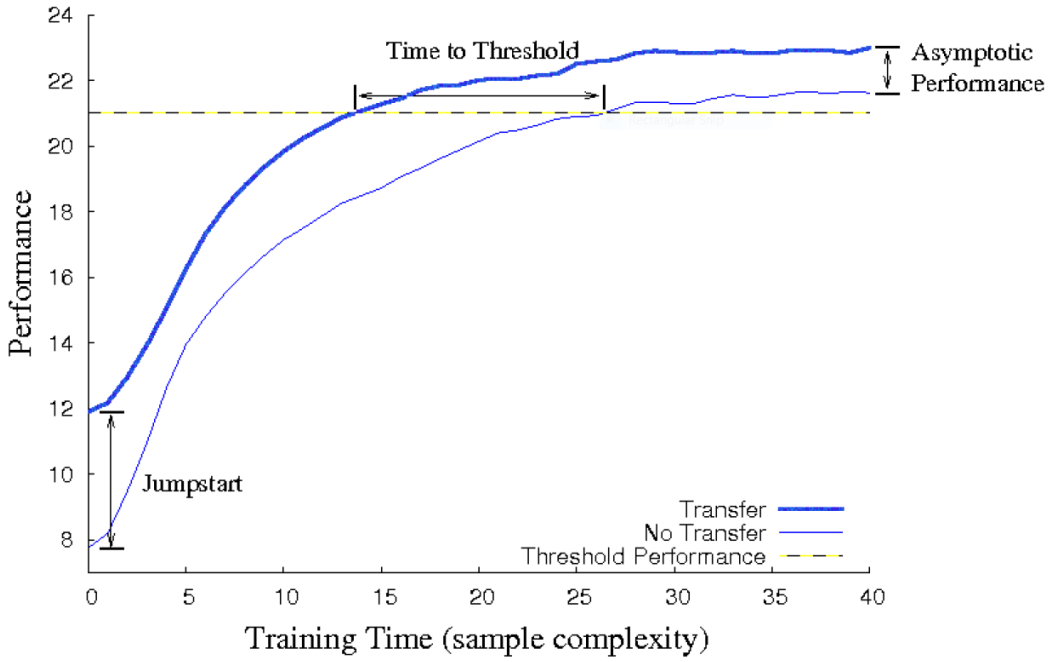


Figure 3.1: Evaluation metrics for Transfer Learning [Taylor and Stone 2009]. This graph highlights the benefits of the jumpstart, asymptotic performance, time to threshold, and regret (the area under the learning curve) metrics.

The jumpstart metric, P_j, is a modification of the regret metric that focuses only a few initial episodes. It evaluates the average reward of the agent for the first N episodes on the target task and is useful for tasks that require high initial performance.

$$P_j = \frac{1}{N} \sum_{t=1}^{N} G_t \tag{3.2}$$

The regret and jumpstart metrics measure the quality of agent exploration in the target task. The following two metrics measure the quality of the learned policy rather than exploration.

The asymptotic performance metric, P_a, measures the final agent performance for N evaluation episodes. Evaluation episodes are episodes where the agent selects the best actions from its learned policy and makes no changes to its policy. The asymptotic performance metric is defined as follows:

$$P_a = \frac{1}{N} \sum_{t=1}^{N} G_t^e \tag{3.3}$$

where G_e is the episodic reward received from an evaluation episode.

Lastly, the time-to-threshold metric, evaluates the number of actions executed during training on both the source and target tasks in order to achieve a given threshold performance g:

$$P_t = -\left(\sum_{m_s} a(m_s) + \sum_{m_t} a(m_t)\right) \tag{3.4}$$

where $a(m)$ returns the number of actions executed during training on task m, and m_s and m_t are source and target tasks respectively. The total number of actions is intended to minimized and is therefore multiplied by -1 (fewer actions executed results in a higher value of the time-to-threshold metric). Time-to-threshold is the only metric in which each task explicitly contributes to the metric result. In the other metrics, the intermediate tasks affect agent performance exclusively through transfer learning and its effect on the agent's behavior in the final task.

We utilize the regret (Equation 3.1) and jumpstart (Equation 3.2) metrics to evaluate methods in Chapter 6.

3.3 Conclusion

In this chapter we discussed Transfer Learning, a technique used to improve or accelerate an RL agent's performance on some complex target tasks, by transferring knowledge obtained from learning simpler source tasks. We first presented the types of knowledge transferred in current Transfer Learning literature, in Section 3.1. Then, we presented evaluation metrics for Transfer Learning to measure the effectiveness of knowledge transfer, in Section 3.2.

Chapter 4

Curriculum Learning

Extending upon Transfer Learning, Curriculum Learning demonstrates how the idea of knowledge transfer can be improved by transferring agent knowledge across multiple tasks of increasing complexity (a curriculum) [Bengio *et al.* 2009], and takes inspiration from the predominant use of curricula in human and animal education. It has become increasingly popular with successful applications in strategy games [Shao *et al.* 2018], shooter games [Wu and Tian 2017; Wu *et al.* 2018], and robotics [Riedmiller *et al.* 2018]. A recent survey [Narvekar *et al.* 2020] has suggested that the notion of a curriculum is not clearly and consistently defined in literature and that many landmark results in Reinforcement Learning, from TD-Gammon [Tesauro 1995] to AlphaGo [Silver *et al.* 2016] have implicitly used curricula to guide training, and that Curriculum Learning will therefore likely be used to solve future landmark problems in Reinforcement Learning. One example of implicit curricula use is the reordering of samples in the target task to define a curriculum [Schaul *et al.* 2016; Ren *et al.* 2018; Kim and Choi 2018; Andrychowicz *et al.* 2017; Fang *et al.* 2019]. Another example is the multi-agent or self-play approach to curriculum generation, where the cooperation or competition between two agents brings about a sequence of increasingly challenging tasks [Sukhbaatar *et al.* 2017; Pinto *et al.* 2017; Baker *et al.* 2019; Bansal *et al.* 2018; Vinyals *et al.* 2019]. The most common definition of a curriculum, which we utilize in our work, is an ordering of tasks without repetitions. In this chapter we discuss various implicit and explicit curriculum generation methods and their relation to our work.

We begin by presenting implicit curricula usage. These methods implicitly induce curricula via sample sequencing, Section 4.1, or multi-agent interaction or self-play, Section 4.2. We then present methods which explicitly create curricula. One approach to explicitly create curricula is to exclusively vary the reward function, initial states, and terminal states of the target task, presented in Section 4.3. This approach restricts MDPs of intermediate tasks since the intermediate tasks are derived from the target task MDP. The following approaches do not make any restriction on the MDPs of intermediate tasks. Another approach to explicitly create curricula involves a meta-MDP which is used to learn curricula, presented in Section 4.4. Another approach formulates curriculum generation as a combinatorial optimization problem where the objective is to find a permutation of tasks that is the best curriculum, presented in Section 4.5. The final approach formulates curriculum generation as a graph connectivity problem where the objective is to connect nodes with edges into a directed acyclic task graph to construct a curriculum, presented in Section 4.6.

4.1 Sample Sequencing

We begin by presenting implicit curriculum generation by means of sample reordering. This approach has proved successful in the context of Supervised Learning, where instead of shuffling training samples and presenting them to a training model randomly, training samples are initially ordered in increasing complexity [Bengio *et al.* 2009]. Bengio *et al.* [2009] demonstrated how this prior ordering of samples can improve the learning speed and generalization ability of the trained models. A similar approach has been taken in the context of Reinforcement Learning, where training samples from a target task are reordered to accelerate training or improve the generalization ability of a Reinforcement Learning agent. These methods have wide applicability since they can be applied without having to create intermediate tasks or rely on the interaction between multiple agents.

In the context of Reinforcement Learning, training samples are typically stored in a replay buffer, used to retrain an agent on previously seen samples to improve learning. The original Deep Q Network (DQN) [Mnih *et al.* 2015] formulation samples from the replay buffer uniformly whereas later modifications prioritize samples according to a measure of usefulness. We discuss some of these modifications below in the context of Deep Learning, that is, Reinforcement Learning involving DQNs.

Schaul *et al.* [2016] were the first to perform sample sequencing. They proposed a method, Prioritized Experience Replay (PER), that prioritizes and replays samples expected to most improve learning progress, where learning progress is quantified using temporal difference (TD) error [Sutton and Barto 1999]. The intuition of this method is to speed up learning by prioritizing and replaying samples which lead to large updates of the learned policy. As training progresses, the distribution of prioritized samples changes, resulting in an implicit curriculum over the samples.

Ren *et al.* [2018] propose an alternative metric for sample priority. They proposed a method that prioritizes and replays samples of appropriate difficulty while penalizing samples that are replayed frequently, using a complexity index function. The complexity index function is the composition of a "self-paced prioritized" function, which selects samples of an appropriate difficulty for the agent's current learning progress, and a "coverage penalty function", which penalizes samples that are replayed frequently.

Kim and Choi [2018] propose an alternative approach to PER by learning a metric for sample priority. Their method uses a secondary neural network to simultaneously learn sample priority as the agent trains. This allows for the method to predict training sample significance even if the sample has not yet been seen and could potentially be used to generate useful samples.

For practical, challenging environments with sparse rewards, some experience replay techniques may be rendered useless since easily accessible states may not initially offer any reward signal. Andrychowicz *et al.* [2017] addresses this issue by proposing a method, Hindsight Experience Replay (HER), that defines pseudo goal states during training to learn from early encountered states, as opposed to using a metric for sample priority. HER implicitly induces curricula during training from training episodes, where initial 'tasks' are initial episodes which end on easily accessible states and final 'tasks' are later episodes which end on harder to reach states.

4.2 Co-learning

Next, we present implicit curriculum generation by means of co-learning. Co-learning implicitly induces curricula through collaboration or competition between multiple different agents or multiple versions of the same agent. One common example of co-learning is self-play, used in many landmarks in Reinforcement Learning, from TD-Gammon [Tesauro 1995], to AlphaGo [Silver *et al.* 2016], and AlphaStar [Vinyals *et al.* 2019]. In self-play an agent repeatedly competes with an alternate, typically older, version of itself, to improve its knowledge, inducing an implicit curriculum of increasingly skilled agents. Below we present Curriculum Learning literature demonstrating how the objectives of multiple agents can facilitate co-learning.

Sukhbaatar *et al.* [2017] propose a method that enables an agent to learn about its environment without any reward signal and in an unsupervised manner. Their method, asymmetric self-play, consists of two seemingly adversarial agents, a teacher and a student, where the objective of the teacher is to identify and propose challenging sub-tasks for the student whilst trading off against the effort required to identify the tasks. The agents alternate learning and the sub-tasks identified by the teacher formulate a curriculum.

Pinto *et al.* [2017] propose a method, Robust Adversarial Reinforcement Learning (RARL), that helps robotic agents learn robust policies in environments involving varying physical parameters. Their method also trains two agents, a protagonist and an adversary, where the role of the adversary is to apply disturbance forces to the protagonist learning to complete a task. The objective of the adversary is to identify and apply the most challenging conditions for the protagonist agent, increasing the protagonist agent's robustness. The two agents alternate learning via a zero-sum game training procedure.

Baker *et al.* [2019] demonstrate how implicit curricula can be induced in the setting of multi-agent Reinforcement Learning (MARL) wherein multiple agents act in the same environment. The agents are tasked to play the game of Hide and Seek where one set of agents, the hiding team, has an objective to use obstacles and objects within the environment to hide, and another set of agents, the seeking team, has an objective to find all the agents of the hiding team. Baker *et al.* [2019] demonstrated that as one team became sufficiently skilled and predominantly won the game, the other team would counteract by improving its strategy and winning. This process was repeated, inducing a curriculum of increasingly competitive agents, and resulted in complex and unexpected behaviours.

Bansal *et al.* [2018] explored a similar idea, demonstrating how MARL can be used as an alternative to designing dense shaping rewards. Their method involves two agents, a learning agent and an adversarial agent, and two types of rewards, sparse multi-agent competitive rewards, and dense "exploration" rewards which are decreased over time. To increase agent robustness, the authors increase task stochasticity over time and sample previous versions of the adversarial agent to prevent the task from becoming too difficult.

Vinyals *et al.* [2019] introduce a method that helps an agent play the game of StarCraft, a complex, multi-agent, and partially observable game, which has not yet been solved by self-play methods alone. The authors propose a league of varying agents, where the objective of one group of agents is to win the game of StarCraft by competing against the other agents as normal, and the objective of another group of agents is to exploit weaknesses in other agents to help them improve. The authors

demonstrated how the main agent participating in the league was able to achieve grand master status in the game through a curriculum induced from playing against the agents in the league.

Thus far, we considered implicit curriculum generation by reordering samples from the target task (Section 4.1) and how multiple agents interacting may induce a curriculum (Section 4.2). In the subsequent sections we present approaches that explicitly create intermediate tasks for a curriculum.

4.3 Reward Function, Initial State, and Terminal State Changes

In this section we present an explicit curriculum generation approach that creates intermediate tasks by varying the reward function, initial states, and terminal states of the target task.

Asada *et al.* [1996], an early example of this approach, teach a robot to shoot a ball into a goal using reverse expansion. Their method involves creating a series of tasks for the robot to solve which have a starting state that becomes increasingly farther away from the goal, inducing a curriculum of tasks. The robot uses vision inputs and the distance to the terminal state is measured using the size of the goal in the camera image.

Florensa *et al.* [2017] proposed a more general method to perform reverse expansion. Their method, reverse curriculum generation, generates a distribution of starting states centered around goal states. Starting states are then selected for intermediate tasks by performing a random walk from existing starting states and if they result in agent performance between a predefined range. This range is tuned to ensure sufficient agent progress.

Florensa *et al.* [2018b] consider the opposite, forward expansion approach, to help guide exploration in an environment. Their method consists of a Generative Adversarial Network (GAN) [Goodfellow *et al.* 2014] that suggests goal regions for the agent, and a discriminator network that evaluates whether or not goal regions are sufficiently difficult for the agent.

Riedmiller *et al.* [2018] provide an alternative approach to those mentioned before by modifying the reward function as opposed initial or terminal states of the target task. They propose a method, SAC-X (Scheduled Auxiliary Control), that also helps guides exploration in the environment by creating and scheduling auxiliary tasks for the agent, where the auxiliary tasks are created by modifying the reward function of the target task.

Wu and Tian [2017] teach an agent to play the game of Doom using a curricula created from varying the transition and reward functions of the target task. Doom is a partially observable, 3D, first-person shooter game where the agent's objective is to progress through increasingly challenging levels by defeating enemies. Their adaptive Curriculum Learning method involves a distribution of curricula which is updated based on the agent's ability by shifting the distribution towards more difficult curricula after the agent performs sufficiently well on the current distribution. Intermediate tasks for the curricula are created by varying the domain parameters (e.g. initial agent health or agent movement speed) and with alternate reward functions. When the agent performs well on the current distribution, the probability distribution is shifted towards more difficult tasks

In this section we presented an approach to explicit curriculum generation in which intermediate tasks are derived from the target task MDP and are somewhat restricted by the target task MDP properties.

In the subsequent sections we present approaches that do not make any restriction on the MDPs of the intermediate tasks.

4.4 MDP-based Sequencing

In this section we present an explicit curriculum generation approach involving a meta-MDP used to create curricula. This approach typically involves two levels of MDPs: a standard MDP used to model the learning agent's environment, and a higher level meta-MDP used to create a curriculum by iteratively selecting tasks for the learning agent.

Narvekar *et al.* [2017] identify this meta-MDP as a curriculum MDP (CMDP), comprised of: a state space consisting of all possible agent policies resulting from learning a (predefined) intermediate task, an action space consisting of all the intermediate tasks, a transition function corresponding to the agent learning an intermediate task and updating its policy, and a reward function designed to minimize training time and ensure sufficient performance on the target task according to a performance threshold. Learning the CMDP is costly, so the authors propose a sample-based training methodology that collects a small number of training samples on the intermediate tasks and the target task. At each step of learning the CMDP, the intermediate task which results in maximum agent policy change is selected and appended to the curriculum under construction.

Matiisen *et al.* [2019] expand on this formulation by defining the meta-MDP as a partially observable MDP (POMDP) [Sutton and Barto 1999] where the meta-MDP cannot observe the agent policy. Instead, the meta-MDP observes and exploits agent performance on intermediate tasks. Additionally, the reward function is alternatively designed to maximize the sum of the performance of the learning agent in all intermediate tasks. To learn the CMDP, the authors propose a heuristic that selects intermediate tasks that cause the greatest absolute change in agent target task performance, that is, intermediate tasks causing the agent to make the most learning progress or causing the agent perform the worst on the target task. Tasks are initially sampled randomly before being sampled according to a weighted distribution of tasks where the weight of each task is the agent performance change.

Narvekar and Stone [2019] demonstrate how learning the meta-MDP directly is possible, eliminating the need for heuristics. Their formulation defines meta-MDP states as the parameters of the knowledge transfer representation (Section 3.1), for instance, if value function transfer is used, the state space consists of learned value function parameters. Their method then uses function approximation to learn a policy over the meta-MDP, termed the curriculum policy, which is still, however, very computationally expensive to learn.

4.5 Combinatorial Optimization and Search

Next we present an explicit curriculum generation approach that treats curriculum generation as an combinatorial optimization problem. The objective under this formulation is to find a superior permutation of a predefined and fixed set of tasks. Each task permutation (curriculum) is evaluated using one of the TL metrics (Section 3.2). We discuss methods which adopt this approach below.

Foglino *et al.* [2019b] design curricula using four modified metaheuristic algorithms. Metaheuristic algorithms are a popular class of methods suited for combinatorial optimization and can be applied to multiple problem domains (black-box). The first two metaheuristic algorithms evaluated, beam search [Ow and Morton 1988] and tabu search [Glover and Laguna 1998], are trajectory-based. Trajectory-based methods predict solution regions and explore the regions for improved solutions. The other two metaheuristic algorithms evaluated, genetic algorithms [Goldberg 1989] and ant colony optimization [Dorigo *et al.* 1996], are population-based. Population-based methods start with a set of candidate solutions and iteratively improve candidates in the set. We make note that the modified ant colony optimization method acts on a task graph (related methods are discussed in Section 4.6) but we include it in this section since the intention of the authors is to treat curriculum generation as an combinatorial optimization problem with the objective to find a superior, fixed length, permutation of a predefined set of tasks.

Foglino *et al.* [2019a] design curricula using a heuristic search algorithm that prioritizes intermediate tasks according to a transferability matrix. The transferability matrix computes the transferability between task pairs by using an intermediate task as a source task for another task and measuring the resultant agent performance change.

Jain and Tulabandhula [2017] propose four search methods to design curricula. The methods exploit online agent learning trajectories to select new tasks. The first method learns each intermediate task for a fixed number of initial steps and appends to a curriculum under construction, the task on which the agent performs the best. The second method calculates a transferability matrix between all task pairs and creates a curriculum by greedily chaining tasks backwards from the target tasks using the matrix. The third method extracts feature vectors for each task and uses the vectors as training data for a regression model to predict the transferability of tasks. Expanding on the third method, the fourth method extracts pair-wise feature vectors for each task pair and uses the vectors as training data for a regression model to predict transferability between tasks.

4.6 Graph-based Sequencing

The final explicit curriculum generation approach treats curriculum generation as a graph connectivity problem. In this context, a task graph consisting of nodes corresponding to intermediate tasks, is given and the objective is to connect the nodes of the graph into a directed acyclic task graph. The resultant graph is a curriculum where directed edges represent task order. Existing literature utilizes heuristics and/or domain knowledge to define the edges in the task graph.

Svetlik *et al.* [2017] require a set of intermediate tasks be predefined and each task be described by domain-specific feature descriptor that encodes properties of the domain. For instance, in the Chess domain, the task features could be the Chess pieces available for use and the number of each piece available. Their method creates the task graph using a heuristic, Transfer Potential, which promotes edges between tasks which share the most number states and have a similar state space size.

Silva and Costa [2018] adopt a similar approach, requiring domain-specific task feature descriptors, but model tasks as object-oriented MDPs (OOMDP) [Diuk *et al.* 2008] and include domain objects in the feature descriptors. They extend the concept of Transfer Potential by additionally grouping tasks according to their objects to define the edges in the task graph.

4.7 Limitations

We note that current CL literature that explicitly creates intermediate tasks for a curriculum, treat the cost of curriculum generation as a sunk cost. This is because it is challenging to assess how much time, effort, and prior knowledge is used to design a curriculum and is especially difficult for methods involving expert knowledge or intervention from humans. Instead, these methods focus on evaluating curricula by measuring the resultant performance change of an agent on some target task after the agent is trained on the curriculum. Treating the process of curriculum generation as a sunk cost is a limitation of many Curriculum Learning approaches, particularly for the use case of accelerating agent training, since the time to generate a curriculum may outweigh the time to directly learn the target task.

In our work we instead highlight the cost of curriculum generation by proposing a simple measure of the cost (the total time of all intermediate training) and imposing strict limits of this cost as a means to compare the generated curricula of curriculum generation algorithms (Section 5.1). Furthermore, we note that current CL literature focuses on creating curricula for a single target task. In our work we demonstrate how a curriculum could be used to train an agent to solve multiple target tasks (Section 5.1), amortizing the cost of curriculum generation.

4.8 Conclusion

In this chapter we presented various literature on implicit and explicit Curriculum Learning. Implicit Curriculum Learning methods implicitly produce a curriculum by either reordering samples from a target task or through the interaction between multiple agents. Alternatively, explicit Curriculum Learning methods explicitly create intermediate tasks for a curriculum. Our proposed method (Chapter 5) explicitly creates intermediate tasks by approaching task sequencing as a combinatorial optimization problem. Unlike other Curriculum Learning literature, we highlight and restrict a proposed cost of curriculum generation, and use Curriculum Learning to train an agent on multiple target tasks (Chapter 5).

Chapter 5

Problem Definition and Methodology

The objective of our work is to highlight and limit the cost of curriculum generation whilst generating high-quality curricula. In this chapter we formalize our objective to limit the cost of curriculum generation and generate high-quality curricula under this restriction in Section 5.1. We then formulate the problem of curriculum generation as a search problem and introduce a curriculum generation algorithm to search and evaluate a space of candidate curricula in Section 5.2.

5.1 Budgeted Curriculum Generation

We use the definition of a curriculum in Definition 1 below as defined by Foglino *et al.* [2019b]. Let T_s be a finite set of source tasks (defined in Chapter 3) for a curriculum where each task is modelled as an MDP. A curriculum, C, is defined as a sequence of tasks in T_s without repetitions:

Definition 1 (Curriculum) *Given a set of source tasks, T_s, a curriculum over T_s of length l is a sequence of tasks $C = (m_1, m_2, \ldots, m_l)$ where each $m_i \in T_s$ and $\forall\, m_i, m_j \in C, i \neq j \implies m_i \neq m_j$.*

We define a task class as a set of tasks modelled as MDPs, where each task in the set has a distinct state space and transition function, and an identical action space, reward function, and discount factor:

Definition 2 (Task Class) *A task class is a set of tasks, T, where each $m_i = (S_i, A_i, p_i, r_i, \gamma_i) \in T$ and $\forall\, m_i, m_j \in T, i \neq j \implies S_i \neq S_j, A_i = A_j, p_i \neq p_j, r_i = r_j, \gamma_i = \gamma_j$.*

To train an agent to solve a class of target tasks, T_f, we follow a process similar to how models are trained in Supervised Learning in Machine Learning. The process consists of two phases: a training phase where candidate curricula are selected for evaluation and evaluated with a subset of tasks from the target class, and a testing phase where generated curricula are evaluated with an alternate subset of tasks from the target class. The process therefore requires a class of target tasks be subdivided into

two sets, a target training set T_f^a, used in the training phase, and a target test set, T_f^b, used in the testing phase. This training process helps prevent overfitting agent knowledge to tasks that are presented to the agent, that is, it helps ensure the agent learns a general ability to solve all tasks in the target task class as opposed to only those tasks that are presented to it.

To train an agent on a curriculum, the agent is trained sequentially, to policy convergence, and with knowledge transfer (defined in Section 3.1) on each of the curriculum's source tasks (using a learning method (Section 2.6). To evaluate a curriculum, the performance of an agent after it is trained on the curriculum, is evaluated on a set of target tasks. In our setting, the set of target tasks used for curriculum evaluation is either the target training set in the training phase where candidate curriculum evaluation is part of the curriculum generation process, or the set of target tasks is the target test training set where the target test set is used to provide an unbiased evaluation of generated curricula and the agent's general task solving ability.

Let $a(m)$ be a function that returns the number of time steps elapsed during training on a task m. We define the cost of curriculum generation, κ, as the total time of all intermediate training, that is, the total training steps during the training phase on source tasks T_s and the target training set T_f^a.

Definition 3 (Cost of Curriculum Generation) *Given a set of source tasks T_s, a set of target training tasks T_f^a, and a function $a(m)$ that returns the time steps elapsed during training on a task m, the cost of curriculum generation, with respect to the source tasks T_s and target training tasks T_f^a is :*

$$\kappa = \sum_{m_s \in T_s} a(m_s) + \sum_{m_t \in T_f^a} a(m_t) \tag{5.1}$$

Let δ be a training budget, a maximum amount of permitted training time on the source tasks in T_s and the target training set T_f^a. Time can be measured in terms of CPU time, wall clock time, episodes, or steps (the number of actions taken). In our work we measure time in steps. A curriculum generator f generates a curriculum using the source tasks in T_s by designing candidate curricula and evaluating them with the target training set T_f^a and a training budget δ, defined as:

Definition 4 (Curriculum Generator) *Given a set of source tasks T_s, a set of target training tasks T_f^a, and a training budget δ, a curriculum generator over T_s with respect to the target training tasks T_f^a is a function $f(T_s, T_f^a, \delta) \rightarrow C$ such that $\kappa < \delta$.*

Using a training budget, δ, the cost of curriculum generation, κ, is therefore bounded, $\kappa \in [0, \delta)$, and is amortized proportionally to the number of tasks in the target test set. That is, the more tasks the agent is able to solve in the target test set, the less training time is required to solve each task.

Let F be a set of curriculum generators under evaluation. In a context where the curriculum generators are evaluated with the same candidate and target tasks, we omit candidate and target task arguments from our notation and define a curriculum generator as $f(\delta) \rightarrow C$, where the source and target tasks are implied. Given a performance metric $P(C) \rightarrow \mathbb{R}$, which evaluates a curriculum's utility for a learning agent on the target test set, and a training budget, δ, we consider the problem of finding an optimal curriculum generator f^*, such that:

$$P(f^*(\delta)) \geq P(f(\delta)), \forall f \in F$$

5.2 Heuristic Curriculum Search

It has been shown that a curriculum that improves an agent's training speed or performance may degrade another agent's training speed or performance on the same target task, and that the utility of curriculum is therefore subject to the agent learning it [Narvekar *et al.* 2017]. Expanding further, we hypothesize that the utility of a curriculum is a function of the utilities of each task that it comprises. This hypothesis forms the intuition behind our approach - to design curricula with tasks that are most beneficial for an agent. In this section we describe a heuristic curriculum search algorithm that uses training samples to quantify task utility and efficiently search and evaluate a space of candidate curricula for some optimal curriculum. The algorithm operates during the training phase using a set of source tasks, T_s, to compose candidate curricula, the target training set, T_f^a, to evaluate candidate curricula, and a training budget, δ, to budget intermediate training time.

We approach task sequencing is as a combinatorial optimization problem. That is, given a fixed set of tasks, to find the permutation of tasks that is the best curriculum. To do so, we formulate the problem of curriculum generation as a search problem by using the predefined set of source tasks to define all possible candidate curricula and structuring the candidates in a search tree, as demonstrated by Figure 5.1.

Within the search tree, each node represents a candidate curriculum and each edge the addition of a source task to a curriculum. The root node of the tree is the empty curriculum, representing the agent learning the target task directly. Henceforth we refer to tree nodes and candidate curricula synonymously. A key advantage of this formulation is that it allows our algorithm to avoid redundant intermediate training. This is possible since our algorithm reuses learned agent policies from shorter curricula to jump-start training on longer curricula.

The search tree is used along with a tree traversal algorithm to generate a curriculum. We apply the tree traversal algorithm, Depth First Search (DFS), to traverse the tree and select candidate curricula for evaluation. DFS starts at the root node and explores as far as possible along each branch before backtracking and prioritizes nodes according to a heuristic value applied to them. We specifically choose DFS as the graph traversal algorithm, and the heuristic function we later describe, since they outperformed various graph traversal algorithms and potential heuristic functions, detailed in Section 6.4. Nonetheless, the concept of our algorithm is general enough to substitute any graph traversal algorithm or heuristic function into it. The candidate curricula selected for evaluation by DFS are evaluated by training the agent on the source tasks of the curriculum before measuring agent performance on the target training set. This process of evaluation exhausts the training budget and stops when the training budget is depleted or all candidate curricula have been evaluated. Thereafter, the candidate resulting in the best agent performance on the target training set is returned.

A useful property of structuring candidate curricula in this way is that at each node of the search tree, the agent is initialized with the policy from learning the parent curriculum and only learns the task appended to the parent curriculum. In this way we avoid redundant training of source tasks and reduce the overall cost of curriculum generation. A limitation of this approach is its memory implications. If an agent policy is defined as the weights of a large neural network or the curriculum search tree is sufficiently large, storing learned policies in memory may be infeasible. One approach to address this limitation is to store agent policies in computer storage and load them into memory as and when they are needed.

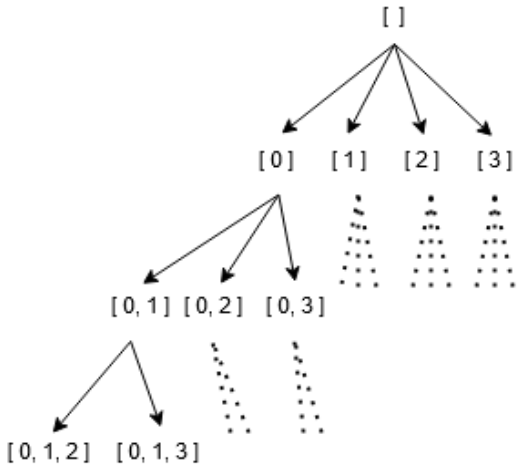


Figure 5.1: Example curriculum search tree with four source tasks (0, 1, 2, and 3), where each task appears at most once in a curriculum.

Our algorithm, depicted in Algorithm 3, summarized in Figure 5.2, and which we make reference to below, takes in as input the root node of the search tree, C, the target training set, T_f^a, and a training budget, δ. It then initializes a stack X to track the current traversal node and a stack Y to track evaluated nodes on lines 1-2. The root node is marked as visited and added to X as the initial traversal node on lines 3-4 and using the function `Push` (line 4). The DFS traversal loop is then started on line 5 and terminates when all nodes are evaluated or the training budget is exhausted. The unvisited children of the current node are retrieved using the function `UnvisitedChildren` (line 6) and a heuristic value is applied to each of them using the function `HeuristicValue` (line 9). The `HeuristicValue` function estimates candidate curricula utility using experience from previously evaluated candidates which share the same source tasks. It does so by first approximating the utility of each source task (`SourceTaskUtility`) as the average score of all evaluated curricula containing the source task (we soon describe how curricula are scored using the `EvaluateCurriculum` function). The heuristic value for each candidate is then calculated as the average approximated utility of the source tasks it comprises ($\texttt{HeuristicValue}(C) = \frac{1}{|C|}\sum_{m \in C} \texttt{SourceTaskUtility}(m)$). The candidate curriculum with highest heuristic value is extracted using the function `MaxHeuristicValue` (line 11) and evaluated and scored with the function `EvaluateCurriculum` (line 12). The `EvaluateCurriculum` function evaluates a candidate curriculum by sequentially training the agent to convergence on each of the curriculum's source tasks before observing agent performance on each target training task with a small, fixed number of episodes. The curriculum is assigned a score equal to the regret (Equation 3.1) of the agent over the target training set and the training budget is reduced by the number training episodes used on the curriculum and target training tasks, as shown on line 12. We make use of this evaluation approach, including the training strategy and performance metric, as previously utilized in other Curriculum Learning literature [Narvekar *et al.* 2017; Foglino *et al.* 2019b] and simply modify it to measure the performance of the agent over multiple target tasks as opposed to a single task. We also note that limiting the number of training episodes over the target training set helps to preserve the training budget. The algorithm then marks the evaluated curriculum as visited (line 13), sets it as the current traversal node by adding it to X using the function `Push` (line 14), and adds it to the list of evaluated curricula, Y, using the function `Push` (line 16) if the training budget is not yet exhausted, as shown on lines

15-17. If there are no child nodes left to evaluate at the current traversal node, the algorithm backtracks by popping the stack X using the function `Pop` (line 19) and setting the current traversal node to the previously evaluated node. This DFS process repeats until all candidates are evaluated or the training budget is exhausted, as previously mentioned, and shown on line 5. Thereafter, the evaluated curriculum with highest score is calculated by the function `MaxScore` (line 22) and returned by the algorithm.

1: $X \leftarrow \emptyset$
2: $Y \leftarrow \emptyset$
3: C.visited $\leftarrow$ true
4: `Push`(X, C)
5: **while** $C \neq$ null and $\delta \geq 0$ **do**
6: $U \leftarrow$ `UnvisitedChildren`(C)
7: **if** `Size`$(U) > 0$ **then**
8: **for** $D \in U$ **do**
9: D.heuristicValue $\leftarrow$ `HeuristicValue`(D)
10: **end for**
11: $C \leftarrow$ `MaxHeuristicValue`(U)
12: $(C.\text{score}, \delta) \leftarrow$ `EvaluateCurriculum`(C, T_f^a)
13: C.visited $\leftarrow$ true
14: `Push`(X, C)
15: **if** $\delta > 0$ **then**
16: `Push`(Y, C)
17: **end if**
18: **else**
19: $C \leftarrow$ `Pop`(X)
20: **end if**
21: **end while**
22: **return** `MaxScore`(Y)

Algorithm 3: `HeuristicCurriculumSearch`(C, T_f^a, δ)

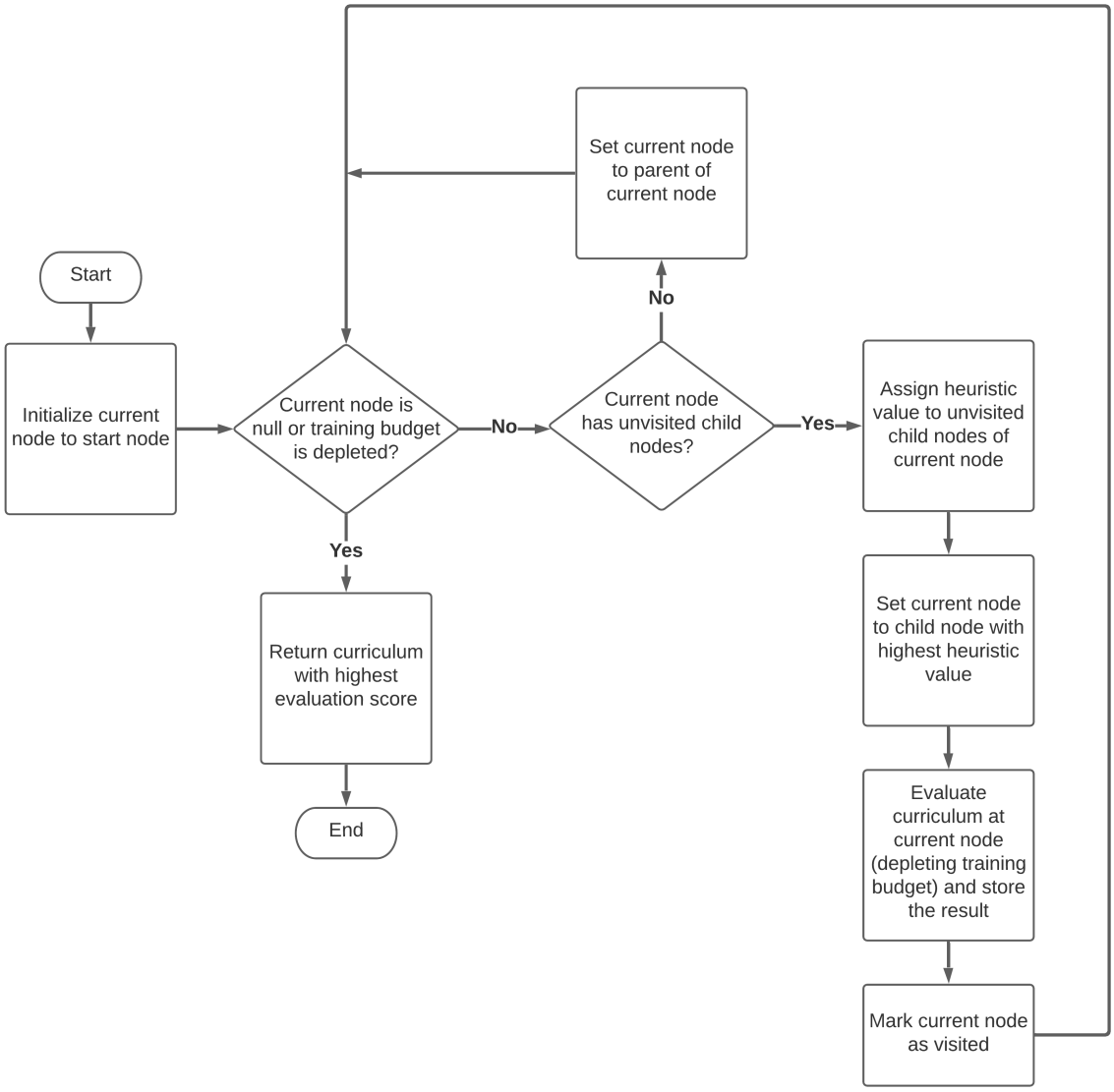


Figure 5.2: Flowchart of Heuristic Curriculum Search Algorithm

5.3 Conclusion

In this chapter we began by formalizing the cost of curriculum generation, a training budget, and our objective to generate high-quality curricula using a training budget, in Section 5.1. We then introduced a heuristic curriculum generation algorithm which utilizes the training budget to search and evaluate a space of candidate curricula for some optimal curriculum, in Section 5.2. The algorithm prioritizes candidate curricula for evaluation using a heuristic that calculates source task utility online and ranks candidate curricula according to the task utility of the source tasks that a candidate curriculum comprises.

Chapter 6

Experimental Results

In the previous chapter we formalized the cost of curriculum generation, a training budget, and our objective to generate high-quality curricula using a training budget. We then introduced a heuristic curriculum generation algorithm designed to generate high-quality curricula utilizing this training budget. In this chapter we present preliminary experimentation which provided insights for the formulation of our algorithm, and we present experimentation comparing our algorithm to baseline methods.

We evaluate our curriculum generation algorithm in two domains designed for and previously used for Curriculum Learning, Block Dude and Mini Grid, each with a unique learning agent suited for the domain. We begin by describing the Block Dude and Mini Grid domains and the learning agent used in each, in Sections 6.1 and 6.2 respectively. We then quantify individual source task utility and demonstrate how certain source tasks may be biased to create good curricula, which forms the intuition behind our algorithm's heuristic function, in Section 6.3. Next, we perform an evaluation of various common and handcrafted heuristic functions and graph traversal algorithms for our algorithm in Section 6.4. We then evaluate our algorithm using the best heuristic and graph traversal algorithm pair found by evaluating its generated curricula when restricted by a training budget and demonstrate that its generated curricula outperform that of baseline methods in Section 6.5. Lastly, we provide supplementary results, studying the memory and storage implications of our algorithm in Section 6.6.

6.1 Block Dude

Block Dude [Brown University 2014] is a puzzle game in which the agent has to arrange boxes to climb over walls and reach the exit. The state available to the agent is a 2D grid where each cell contains either the agent, a wall or floor, a box, the exit, or an empty space. The actions available to the agent are moving left, moving right, climbing onto a box or wall, picking up a box, and putting a box down. The agent receives a reward of 50 for reaching the exit state and a reward of -1 for every other action taken. It is available in the Burlap software library [Brown University 2014] and has previously been used in other Curriculum Learning literature [Svetlik *et al.* 2017; Foglino *et al.* 2019b], including literature which our baselines are inspired by.

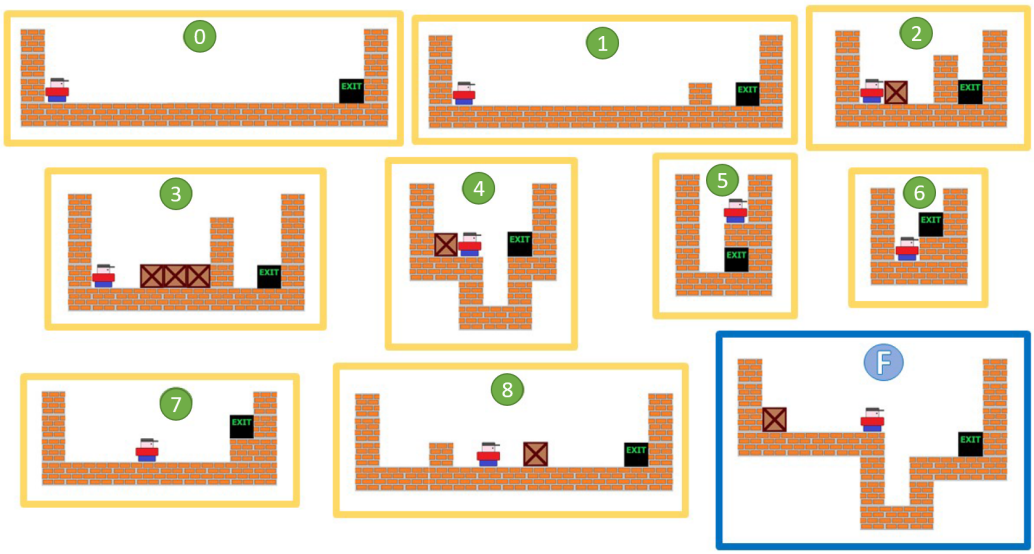


Figure 6.1: Source tasks (outlined in yellow) and the target task (outlined in blue) in the Block Dude domain [Foglino *et al.* 2019b]. The man is the agent, light brown squares are walls and floors, dark brown squares are boxes which the agent can move and climb onto, and the black square is the exit

6.1.1 Task Description

We define the source and target tasks (Figure 6.1) and limit our algorithm to only consider candidate curricula having a maximum length of three source tasks, identical to the work of Foglino *et al.* [2019b]. In both our work and that of the original authors, this relatively small domain is chosen to perform a thorough evaluation of the candidate curricula and our algorithm. This domain is configured to have a single target task used for both training and testing purposes and we later explore a case with multiple target tasks in our next domain.

During the training phase, to train the agent on curricula, we sequentially train the agent to convergence on each of the curriculum's source tasks. Our algorithm then evaluates and approximates candidate curricula utility by assigning each candidate a score equal to the agent performance on the target task after the agent has been trained on the curriculum and with a small number of episodes, 400. During the testing phase the generated curricula are evaluated by training the agent on the curriculum and then observing agent performance on the target task for 1400 episodes (enough episodes for the untrained agent to converge training). Agent performance is measured using the regret metric (Equation 3.1). We average the training curves of our agent on each task over 50 runs.

6.1.2 Agent Description

To facilitate transfer, we defined a feature space consisting of localized features with respect to our learning agent. It is defined as: the direction of the agent (binary), holding box indicator (binary), next to and facing wall indicator (binary), next to and facing box indicator (binary), can climb box indicator (binary), can climb wall indicator (binary), the agent's distance to the exit, and the average

distance of all boxes to the exit. Additionally, we scaled the distances in the feature space to be in the range of 0 and 1 so that the input to our value function approximator is invariant to the scale of the domain. This feature space is then used as input to a tile coding value function approximator with 6 tilings over all features and tile widths equal to 0.1.

We used a variant of the general SARSA algorithm (Algorithm 2), Gradient Descent SARSA(λ) to enable our agent to work with function approximation, and as used in other Curriculum Learning literature [Narvekar *et al.* 2017; Foglino *et al.* 2019b]. We used hyperparameters $\lambda = 0.5$, learning rate $\alpha = 0.02$ and discount factor $\gamma = 1$. The agent explores with a ϵ greedy policy, with fixed $\epsilon = 0.1$. The hyperparameters were optimized through manual tuning starting from implementations in other Curriculum Learning literature [Narvekar *et al.* 2017; Foglino *et al.* 2019b] and selected as the most conducive for Curriculum Learning in the domain after several trials and based on the agent's overall performance on the target task.

We apply value function transfer [Taylor and Stone 2009] to transfer learned knowledge across successive source tasks in a curriculum and finally to the target task.

6.2 Mini Grid

Mini Grid [Chevalier-Boisvert *et al.* 2018] is a 2D grid world in which an agent is tasked to complete an objective by exploring rooms and sequentially interacting with objects. It includes several configuration options which require the agent to achieve a particular objective by interacting with the objects in the environment in a particular sequence. We utilize one of its inbuilt objective configurations, `KeyCorridor`, which requires the agent to search rooms and find a key that allows it to get to an exit in a locked room. This configuration includes predefined tasks of increasing complexity for the purpose of Curriculum Learning and is particularly challenging because of the agent's ability to only partially observe the state space, which we describe in more detail in Section 6.2.2.

There are five actions available to the agent in each state: move forward, turn left, turn right, pick up key, and open door. The pick up key action only succeeds if a key is in front of the agent. The open door action only succeeds if a locked door is in front of the agent and the agent has picked up a key which can unlock the door (a key of the same colour as the door). The agent receives a reward of 50 for reaching the exit state and a reward of -1 for every other action taken. A learning episode terminates when the agent reaches the exit state or 400 time steps have elapsed.

6.2.1 Task Description

We utilize the default source and target tasks recommended for Curriculum Learning by the Mini Grid authors [Chevalier-Boisvert *et al.* 2018] and include an additional two source tasks (3 and 4), depicted in Figure 6.2. The additional source tasks help our agent bridge an identified knowledge gap on the target task. We limit our study to candidate curricula having a maximum length of three source tasks because of the computation time required to evaluate all candidate curricula.

In each task, Mini Grid randomly initializes the agent, key, and exit locations. The key is always initialized in a room on the left side of the grid. The agent is always initialized in the center corridor.

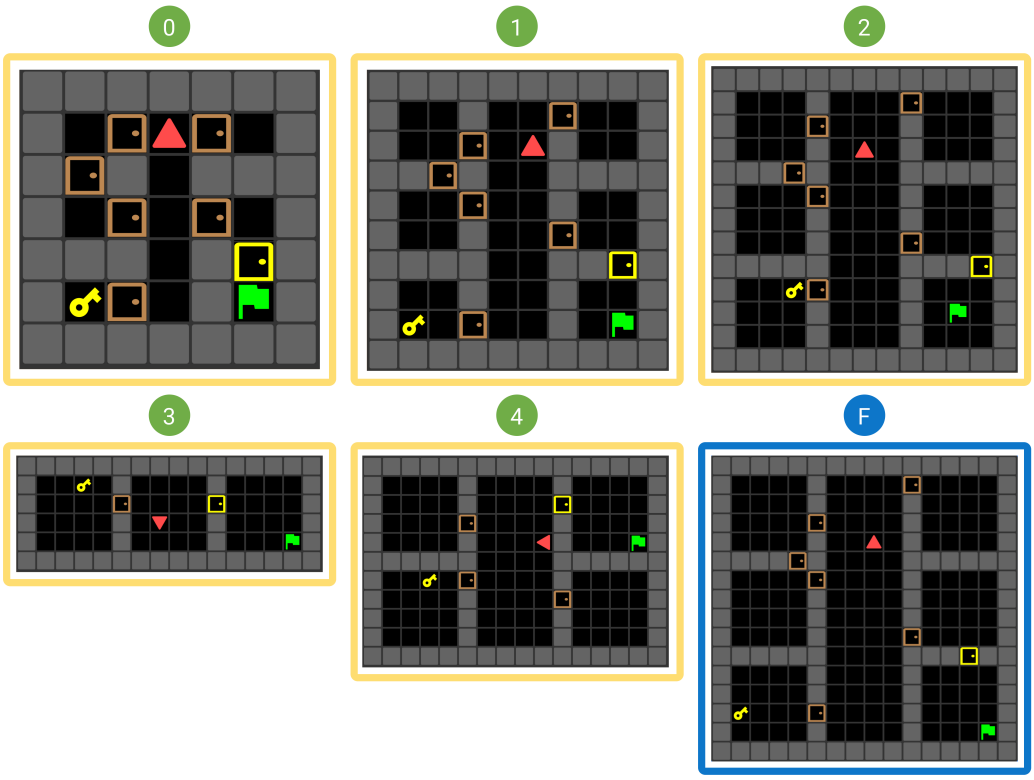


Figure 6.2: Source tasks (outlined in yellow) and the target task (outlined in blue) in the Mini Grid domain.

And the exit is always initialized in a room on the right side of the grid and behind a locked door. We instantiate and fix three configurations (varying key, door, agent, and exit positions) of each source task, five configurations of the target task as target training tasks, and an additional 15 configurations of the target task as target test tasks. We consider each target task configuration as a separate task and belonging to a single task class. Furthermore, we aim for the agent to learn a general ability to solve all tasks belonging to this class, and defining multiple configurations of each source task helps to achieve this aim.

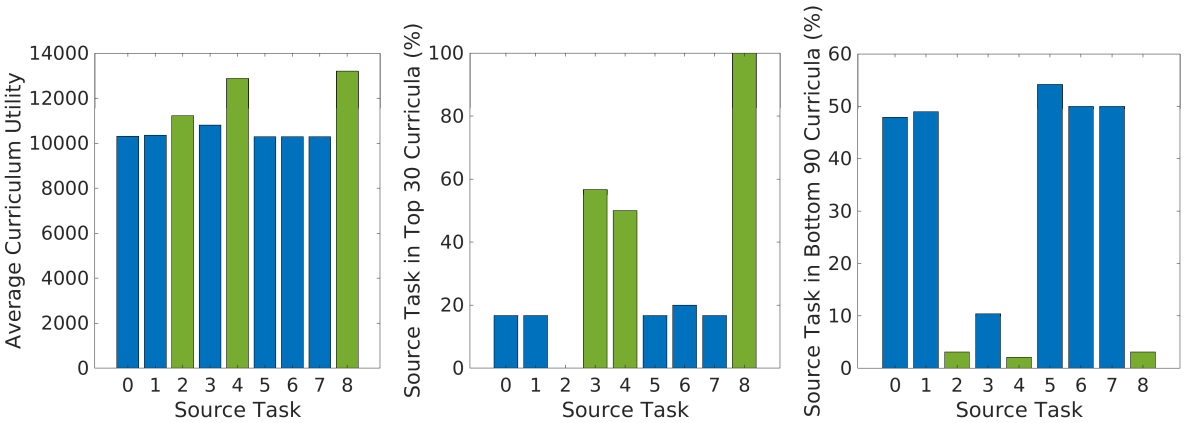
During the training phase, to train the agent on curricula, we sequentially train the agent to convergence on each of the curriculum's source task configurations. Our algorithm then evaluates and approximates candidate curricula utility by assigning each candidate a score equal to the agent performance over the target training set after the agent has been trained on the curriculum and for with a small number of episodes, 5000, on each target training task. During the testing phase the generated curricula are evaluated by training the agent on the curriculum and then observing agent performance over the target test set for 15 000 episodes on each task (enough episodes for the untrained agent to converge training). Agent performance is measured using the regret metric (Equation 3.1). We average the training curves of our agent on each task over each of the task's configurations and over 50 runs.

6.2.2 Agent Description

The agent's environment in Mini Grid is partially observable, restricting the agent to observing only its local surroundings. Each state observation is a $5\times 5\times 3$ array representing the 5×5 view in front of the agent with three channels (object IDs, color IDs, and a binary matrix capturing whether or not a door is open). We feed each state observation to a tabular Q-Learning agent with learning rate $\alpha = 0.1$, discount factor $\gamma = 0.99$, and initialize its q values to 0. The agent explores with a ϵ greedy policy, with fixed $\epsilon = 0.1$. The hyperparameters were optimized through manual tuning starting from implementations in other Curriculum Learning literature [Narvekar *et al.* 2017; Foglino *et al.* 2019b] and selected as the most conducive for Curriculum Learning in the domain after several trials and based on the agent's overall performance on the target tasks. We apply value function transfer [Taylor and Stone 2009] to transfer learned knowledge across successive source tasks in a curriculum and finally to the target task.

6.3 Source Task Utility

We begin our experimentation by demonstrating how individual source tasks may be biased to create good curricula. This finding inspired our algorithm's heuristic function that learns online the utility of individual source tasks and leverages that information to efficiently construct good curricula. We first quantify the utility of individual source tasks. To do so we measured the utility of all candidate curricula in the Block Dude domain. We measure curricula utility as the regret (Equation 3.1) of the agent on the target task after the agent has trained on the curriculum and for 1400 episodes on the target task (more than enough time for the untrained agent to converge training on the target task). We then quantify the utility of each source task as the average performance of all candidate curricula containing the task. The results are presented in Figure 6.3.



(a) Average curriculum utility of curricula containing a specified source task.

(b) Source tasks contained in top 30 performing curricula.

(c) Source tasks contained in bottom 90 performing curricula.

Figure 6.3: Source task utility in Block Dude. Green bars are the top 3 performing source tasks in each category of evaluation.

We observe from the results that candidate curricula containing source tasks 4 and 8 in the Block

Dude domain, on average, perform better than the other candidate curricula, as shown in Figure 6.3a. We also observe that the top 30 performing curricula comprise mostly of these tasks, as shown in Figure 6.3b, and that the bottom 90 candidate curricula mostly do not comprise of these tasks, as shown in Figure 6.3c.

We believe the reasons the source tasks 4 and 8 were most beneficial for the agent to solve the target task are since they:

1. modify and simplify the target task as opposed to creating distinctly different tasks.
2. remove specific and challenging aspects of the target task (as is the case with source task 8).
3. focus on specific and challenging aspects of the target task (as is the case with source task 4).

We therefore hypothesize that: 1) source tasks most beneficial for Curriculum Learning are those that decompose and simplify the target task, and 2) that good curricula are those that independently expose the agent to certain aspects of the target task.

We conclude from these findings that certain source tasks may be biased towards constructing good curricula for a learning agent and that they may be identified online during training. We also hypothesize that good source tasks and curricula can be created by independently exposing the agent to certain aspects of the target task.

6.4 Heuristic and Graph Traversal Algorithm Exploration

We formulated the problem of curriculum generation as a search problem in which we employ a graph traversal algorithm and heuristic function to guide exploration and select candidate curricula for evaluation. In this section we evaluate various common and handcrafted graph traversal algorithms and heuristics functions for our algorithm in both Block Dude and Mini Grid. The best performing graph traversal algorithm and heuristic function pair was chosen for the definition of our algorithm, as detailed in Section 5.2.

We evaluated 5 graph traversal algorithms and 5 heuristic functions, resulting in 25 permutations (5×5) of our algorithm for evaluation. We name each permutation of our algorithm as the concatenation of its heuristic function and graph traversal algorithm, for example `RegretBFS`. We present each graph traversal algorithm and heuristic function evaluated in Sections 6.4.1 and 6.4.2 respectively. We then present and analyze the results of the top 3 permutations of our algorithm in Section 6.4.3.

6.4.1 Graph Traversal Algorithms

The graph traversal algorithms evaluated are as follows:

1. `AStar` (A*)

   Keeps a record of child nodes of all expanded nodes starting from the root node and expands a child node with the maximum heuristic value at each step.

2. `BeamSearch`

   Keeps a record of the current tree level (of expansion) and child nodes of all expanded nodes at the tree level, starting at the root node. At each step it increments the current tree level and expands a maximum of w (the beam width) child nodes with the highest heuristic value at the current tree level. If there are no child nodes left to expand at the current tree level, it backtracks by decrementing the current tree level. This algorithm has been used in previous Curriculum Learning literature [Foglino *et al.* 2019b] and we set the beam width equal to the number of source tasks in each domain, identical to the original authors.

3. `BreadthFirstSearch (BFS)`

   Keeps a record of the current tree level and child nodes of all expanded nodes at the tree level, starting at the root node. At each step it increments the current tree level and expands all child nodes at the current tree level.

4. `DepthFirstSearch (DFS)`

   Keeps a record of the current tree level and child nodes of all expanded nodes at the tree level, starting at the root node. At each step it increments the current tree level and expands a single child nodes with the highest heuristic value at the current tree level. If there are no child nodes left to expand at the current tree level, it backtracks by decrementing the current tree level.

5. `LazyDepthFirstSearch (Lazy DFS)`

   Keeps a record of the current tree level and child nodes of all expanded nodes at the tree level, starting at the root node. At each step it increments the current tree level and expands all child nodes of the parent node with the highest heuristic value at the current tree level. If there are no child nodes left to expand at the current tree level, it backtracks by decrementing the current tree level.

6.4.2 Heuristic Functions

The heuristic functions we evaluated are as follows:

1. `PolicyChange`

   Prioritizes tasks which change the agent's policy the most. The heuristic is calculated as the percentage of the states for which the agent's learning policy action has changed as compared to the agent's learning policy before learning the task. This heuristic is inspired by previous Curriculum Learning literature [Narvekar *et al.* 2017].

2. `Regret`

   Prioritizes tasks which result in the best performance on the target task. Agent performance is measured using the regret metric (Equation 3.1).

3. `RegretChange`

   Prioritizes tasks which result in the greatest increase in performance on the target task. The heuristic is calculated as a ratio of the agent performance on the target task after having learned the task divided by the agent performance on the target task before having learned the task. Agent performance is measured using the regret metric (Equation 3.1).

4. `SourceTaskUtility`

   Prioritizes tasks which have the greatest estimated utility. The process of calculating task utility has been described in Section 5.2.

5. `CurriculumUtility`

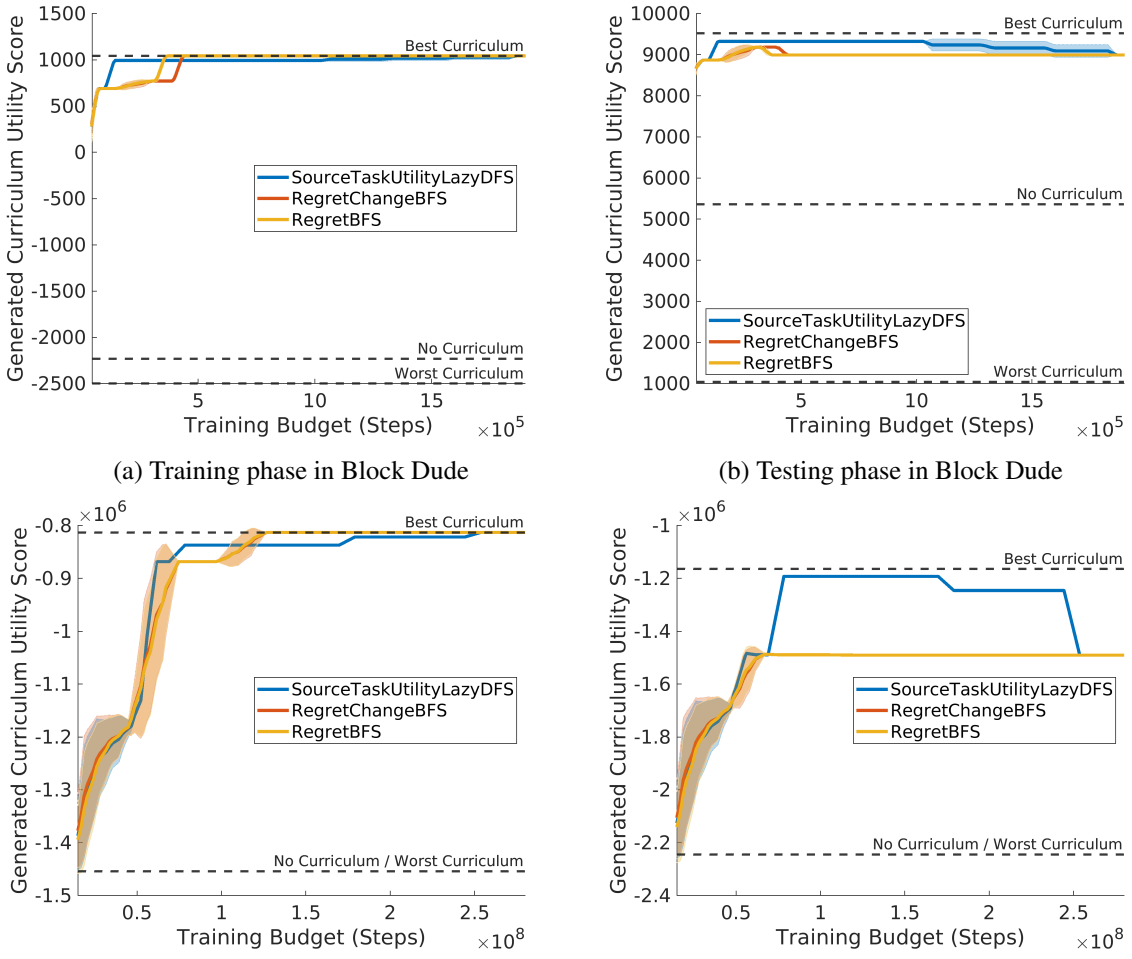
   Prioritizes tasks which result in the greatest estimated curriculum utility of the curriculum under construction. The estimated utility of a curriculum is calculated as the average estimated utility of the source tasks it comprises, and has been described in Section 5.2.

6.4.3 Results

Figure 6.4 presents the results of the top three permutations in Block Dude and Mini Grid. The first plot, Figure 6.4a, demonstrates (estimated) generated curriculum utility with varying training budgets in Block Dude, and during the training phase, where the generated curriculum utility is calculated as agent performance on the target training tasks for a small number of episodes. The second plot, Figure 6.4b, demonstrates (actual) generated curriculum utility with varying training budgets in Block Dude, but during the testing phase, where the generated curriculum utility is calculated as agent performance on the target testing tasks for sufficient episodes to reach policy convergence. The key difference between the training and testing phases is that during the training phase the algorithms use the training budget and observations from a set of target tasks dedicated for training (target training set) to generate curricula, and during the testing phase the generated curricula are evaluated against a set of target tasks dedicated for testing (target testing set) with no feedback to the algorithms. The tasks and training parameters are detailed for Block Dude and Mini Grid in Sections 6.1 and 6.2 respectively. Additionally, we plot as reference points the best curriculum (highest curriculum utility), worst curriculum (lowest curriculum utility), and empty curriculum (no preliminary agent training), during the training and testing phases. We plot similar results for the Mini Grid domain in Figures 6.4c and 6.4d.

We observe from the results that during the training phase in each domain, as the training budget increases, each permutation generates curricula of steadily increasing utility until they all converge to the curriculum with optimal utility. However, during the testing phase in each domain, we observe that as the training budget increases, each permutation converges to a sub-optimal curriculum. This is because the curriculum with optimal (estimated) utility during the training phase is in fact not optimal when evaluated during the testing phase. This lack of generalization could be because of two cases: 1) the target training set is not representative enough of the target class and the agent does not learn sufficient knowledge to apply to the target task class but instead overfits knowledge to the target training set, or 2) curriculum generation methods which estimate curriculum utility risk estimating it incorrectly.

It appears both cases caused the observed results. Firstly since in the Block Dude domain we experiment on a single target task and the first case would therefore not apply. We estimate curriculum utility using agent performance on the target training set and for a small number of episodes. We can therefore conclude that initial agent performance is not indicative of overall agent performance. And secondly since in the Mini Grid domain we define a target training set of 5 tasks randomly sampled from the target class of 20 tasks and we suspect this is too few tasks in the target training set



(a) Training phase in Block Dude

(b) Testing phase in Block Dude

(c) Training phase in Mini Grid

(d) Testing phase in Mini Grid

Figure 6.4: Top 3 heuristic and graph traversal combinations for our algorithm in Mini Grid and Block Dude. (Estimated) generated curriculum utility calculated during the training phase is compared to (actual) curriculum utility during the testing phase. Each curve was averaged over 50 runs and is plotted with standard deviation error bars.

since Supervised Learning methods (which inspired our training methodology) typically require large amounts of training data to be effective.

In the first case, there is opportunity to study the effect of target class size and how to subdivide it effectively for training and testing purposes in Curriculum Learning. In the second case, there is opportunity to study how best to estimate curriculum utility. A trivial or brute force approach could be to observe agent performance on the entire target class and until policy convergence. This approach is infeasible for large or complex tasks, for exceedingly many tasks, or for the objective to accelerate agent training where it may be preferable to train the agent on the target class directly. Furthermore, this approach would not be an estimate but rather actual agent performance on the target class. Both cases therefore offer opportunity for future work.

The top performing permutation, `SourceTaskUtilityLazyDFS`, generates superior curricula

during the testing phase in both domains, as seen in Figures 6.4b and 6.4d. This appears to be the case since:

1. the `SourceTaskUtility` heuristic leverages all training history to prioritize candidate curricula as opposed to only information about parent curricula in the candidate curriculum tree graph (Section 5.2). The `SourceTaskUtility` heuristic is therefore able to make more informed decisions.
2. the `LazyDFS` traversal strategy is best suited to effectively acquire the information required for the `SourceTaskUtility` heuristic. It acts in a greedy manner, iteratively appending to the curriculum under construction source tasks with maximum utility.

In conclusion, we identify from our findings a limitation where curriculum generation algorithms may inaccurately estimate candidate curricula utility and produce suboptimal curricula. Future work could be dedicated to address this issue and explore methods to better estimate candidate curriculum utility. Additionally, we conclude that the `SourceTaskUtility` heuristic function and `LazyDFS` traversal strategy is the best performing pair for our algorithm since it leverages all training history to make more informed decisions about candidate curricula as compared to other heuristics, and that it effectively selects candidate curricula for evaluation in order to acquire the information required to best estimate curricula utility.

6.5 Efficient Curriculum Generation

We aim to create an efficient curriculum generation algorithm, that is, a method which maximally utilizes its training budget to create good curricula. In this section we evaluate our algorithm using the optimal heuristic and graph traversal algorithm pair identified in the previous section, by evaluating its generated curricula when restricted by a training budget. We demonstrate that its generated curricula outperform that of baseline methods.

We begin by presenting baseline methods, methods adapted from existing Curriculum Learning literature for comparison with our work, in Section 6.5.1. We then present and analyze results of our algorithm in Block Dude and Mini Grid in Sections 6.5.2 and 6.5.3 respectively.

6.5.1 Baselines

At the time of experimentation we identified adaptive task sequencing methods as being most comparable with our work in the context of Curriculum Learning since they dynamically adapt to the learning progress of the agent (the same as our work) as opposed to statically generating curricula using properties of the domain and the agent. The work by Narvekar *et al.* [2017] is an example of an adaptive task sequencing method. The authors propose a greedy algorithm that iteratively selects and appends tasks to an empty curriculum. It is guided by a policy-change heuristic, prioritizing tasks which modify the agent's policy the most. The algorithm terminates when the agent outperforms some predefined performance threshold on the target task after training on the curriculum. The other example is the work by Foglino *et al.* [2019b]. The top performing algorithm in their work, Beam

Search, also works by greedily appending tasks to an empty curriculum. It builds several candidate curricula at a time, up to a maximum of w candidates (i.e. the beam width), before returning the best one and appends tasks which most improve the regret metric (Equation 3.1) of the agent on the target task. The algorithm terminates when a predefined maximum curriculum length is reached.

In both examples, the algorithms are unbounded by intermediate training time and are defined in such a way that they are unable to evaluate all possible candidate curricula. For direct comparison with our work, we modify the termination conditions of the algorithms to stop once a training budget has been exhausted and enable the algorithms to backtrack, allowing them to explore all possible candidate curricula. These modified algorithms are used as baselines: `PolicyChangeLazyDFS` based off the work by Narvekar *et al.* [2017], and `RegretBeamSearch` based off the work by Foglino *et al.* [2019b], where the naming convention of each baseline is the heuristic measure and approximate search strategy of each method. In our experiments we set the beam width equal to the number of source tasks in the domain, identical to the original authors.

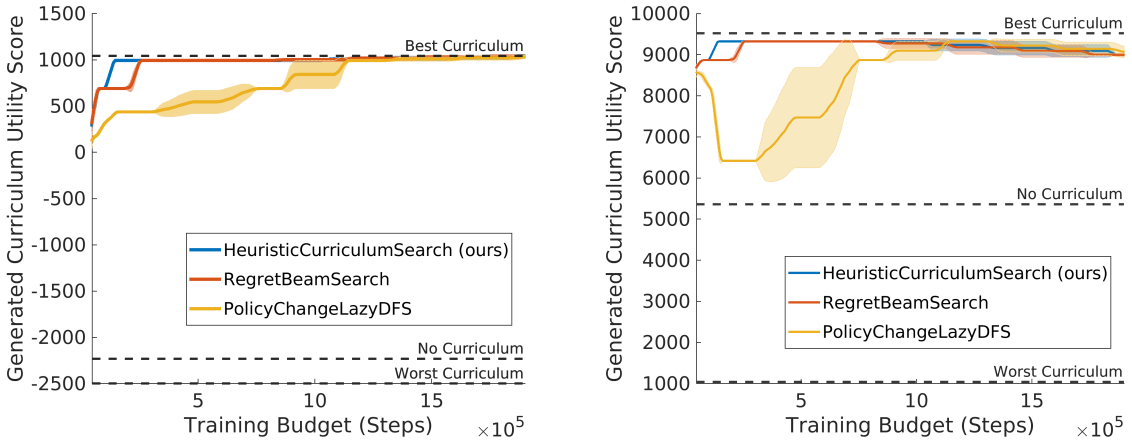
6.5.2 Block Dude

We analyzed the efficiency of our algorithm in Block Dude by studying how the training budget affects the utility of its generated curriculum, as shown in Figure 6.5. We score generated curricula by the agent performance (using the regret metric (Equation 3.1)) on the target task after having trained on the curriculum for 400 episodes during the training phase and 1400 episodes during the testing phase (detailed in Section 6.1.1). Additionally, we plot as reference points the best curriculum (highest curriculum utility), worst curriculum (lowest curriculum utility), and empty curriculum (no preliminary agent training), during the training and testing phases. We plot the performance of each algorithm over 50 runs and with standard deviation error bars.

We observe from the results that our algorithm requires the smallest training budget to generate the curriculum with (estimated) maximum utility, as shown in Figure 6.5b. Additionally, we observe that our algorithm always performs at least as well as the next best baseline, and outperforms it within a training budget range of around 100 000 and 200 000 steps, as shown in Figure 6.5b.

For reference, we compare the training curves of the learning agent on the target task after being trained with various generated and baseline curricula, shown in Figure 6.6. The curricula are generated with a training budget of 200 000 steps. This point is roughly the minimum training budget required by our algorithm to sequence the optimal (estimated) curriculum. We additionally plot as baselines the training curve of the agent when it is not trained on any curriculum (untrained agent) and the training curve of the agent when it is trained with the worst curriculum. Each curve was averaged over 50 runs and error bars are omitted for improved readability. We observe from the results a marginal improvement of the agent training curve when trained with our algorithm's curriculum as compared to the next best baseline, `RegretBeamSearch`, and significant improvement in the agent training curve when compared to the other baselines.

We perform further analysis of each algorithm in both domains in Section 6.5.4.



(a) Training phase in Block Dude

(b) Testing phase in Block Dude

Figure 6.5: Performance of curriculum generation algorithms when restricted by a training budget in the Block Dude domain. (Estimated) generated curriculum utility calculated during the training phase is compared to (actual) curriculum utility during the testing phase. Each curve was averaged over 50 runs and is plotted with standard deviation error bars.

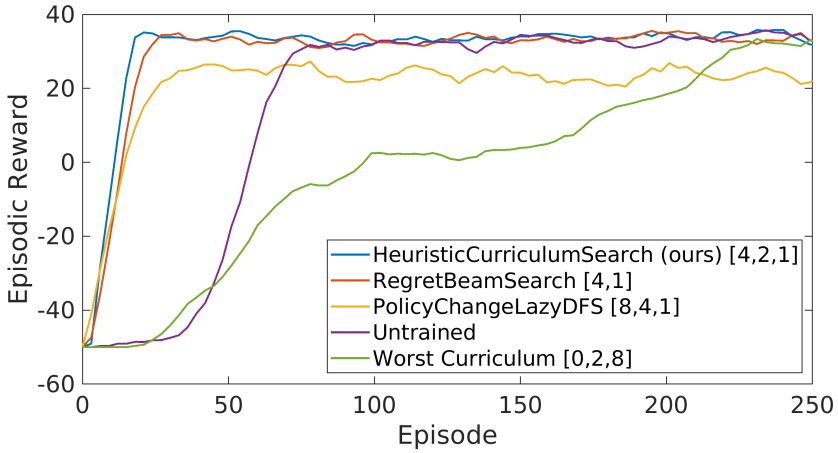
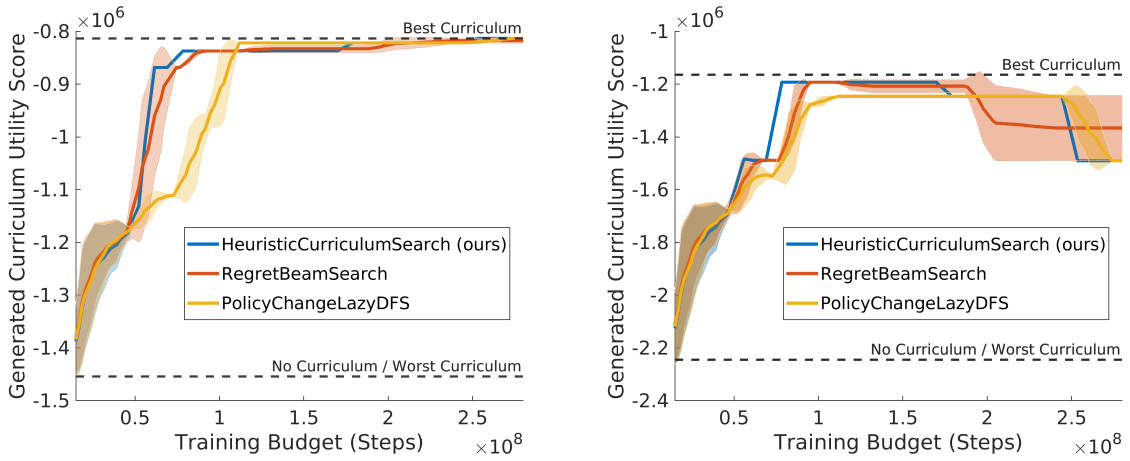


Figure 6.6: Agent performance on the Block Dude target task after training with various generated and baseline curricula. Each curve was averaged over 50 runs and error bars are omitted for improved readability. Generated curricula are appended to the algorithm name with reference to the domain's source tasks.

6.5.3 Mini Grid

We then analyzed the efficiency of our algorithm in Mini Grid by also studying how the training budget affects the utility of its generated curriculum, as shown in Figure 6.7. We score generated curricula by the agent performance (using the regret metric (Equation 3.1)) on the target task after having trained on the curriculum for 5 000 episodes during the training phase and 15 000 episodes during the testing phase (detailed in Section 6.2.1). Additionally, we plot as reference points the best curriculum

(highest curriculum utility), worst curriculum (lowest curriculum utility), and empty curriculum (no preliminary agent training), during the training and testing phases. We ran our experiment for 50 runs and plotted the average performance of each algorithm with standard deviation error bars.



(a) Training phase in Mini Grid

(b) Testing phase in Mini Grid

Figure 6.7: Performance of curriculum generation algorithms when restricted by a training budget in Mini Grid. Generated curricula are scored using agent performance over the target test set after the agent is trained on the generated curriculum. Each curve is averaged over 50 runs and plotted with standard deviation error bars.

We observe from the results that our algorithm requires the smallest training budget to generate the optimal (estimated) curriculum with a difference of roughly 10 million training steps required by the next best baseline to do the same, as shown in Figure 6.7b. We also observe that our algorithm is the most consistent with the smallest error bars in both training and testing phases.

For reference on the target task, we compare the training curves of the learning agent after being trained with various generated and baseline curricula, shown in Figure 6.8. The curricula are generated with a training budget of 80 million steps which is roughly the minimum training budget required by our algorithm to sequence the optimal curriculum. We additionally plot as baselines the training curve of the agent when it has not learned any curriculum (untrained agent), and the training curve of the agent when it is trained with the worst curriculum. We observe from the results a marginal improvement of the agent training curve when trained with our algorithm's curriculum as compared to the other baseline methods, where each algorithm is given an identical training budget. Each curve was averaged over all target test tasks and over 50 runs, and error bars are omitted for improved readability.

6.5.4 Analysis

We again observe that during the testing phase in each domain, as the training budget increases, each permutation converges to a sub-optimal curriculum, depicted in Figures 6.5b and 6.7b. This is because the curriculum with optimal (estimated) utility during the training phase is in fact not optimal when evaluated during the testing phase. We explained this phenomenon and its potential causes in

Section 6.4.3.

We note that our algorithm generates superior or at least equivalent curricula during the testing phase in both domains, as seen in Figures 6.5b and 6.7b. For the same reasons as identified in Section 6.4.3, it appears this is the case because:

1. the `SourceTaskUtility` heuristic leverages all training history to prioritize candidate curricula as opposed to only information about parent curricula in the candidate curriculum tree graph (Section 5.2). The `SourceTaskUtility` heuristic is therefore able to make more informed decisions.

2. the `LazyDFS` traversal strategy is best suited to effectively acquire the information required for the `SourceTaskUtility` heuristic. It acts in a greedy manner, iteratively appending to the curriculum under construction source tasks with maximum utility.

We also observe that our algorithm is more consistent, with smaller error bars, but we cannot explain this phenomenon at this stage. Lastly, we observe that the `PolicyChangeLazyDFS` baseline is biased to initially explore and evaluate sub-optimal candidates, as shown in Figures 6.5 and 6.7. It appears this is a result of its heuristic function, `PolicyChange`, which prioritizes tasks that change the agent's policy the most but policy change is not indicative of useful knowledge, for instance, the agent could learn skills which are not useful at all for the target task.

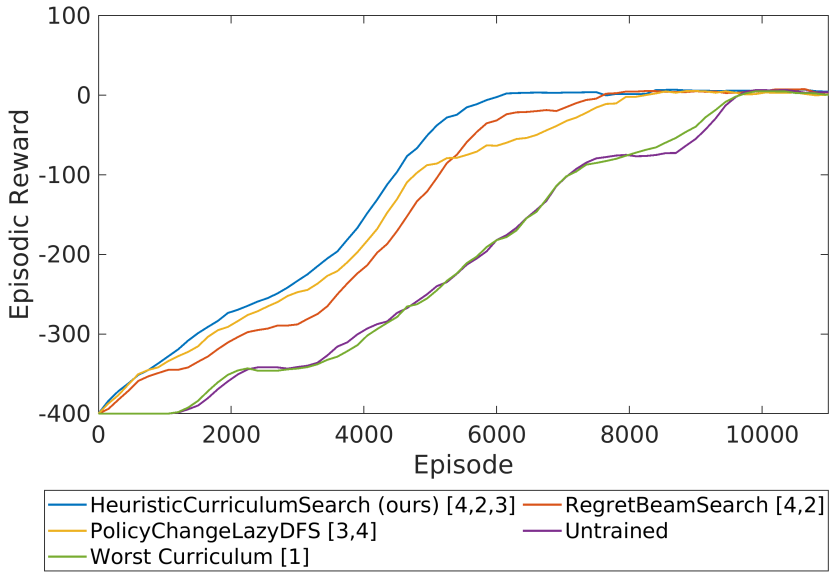


Figure 6.8: Performance on the Mini Grid target test tasks by our agent after training using various generated and baseline curricula. Each curve was averaged over all target test tasks and over 50 runs. Error bars are omitted for improved readability.

6.6 Curriculum Search Space Statistics

Our algorithm acts on a curriculum search tree (defined in Section 5.2). A useful property of structuring candidate curricula in this way is that at each node of the search tree, the agent is initialized with the policy from learning the parent curriculum and only learns the task appended to the parent curriculum. In this way we avoid redundant training of source tasks and reduce the overall cost of curriculum generation. A limitation of this approach is its memory or storage implications. If an agent policy is defined as the weights of a large neural network or the curriculum search space (all curricula of the curricula search tree) is large, storing learned policies in memory or storage may be infeasible.

In attempt to understand our algorithm's memory or storage implications, in this section we provide supplementary results, the statistics of our algorithm in the curriculum search space. Specifically, we measure the number of candidate curricula evaluations required to generate the estimated best curriculum (the curriculum with highest utility during the training phase), and the percentage of all candidate curricula evaluated to generate the estimated best curriculum. Since our algorithm stores agent policies after each candidate curriculum evaluation, the number of candidate curricula evaluations represents the number of agent policies stored.

We evaluate our algorithm with the same baselines as defined in Section 6.5.1 and average results over 50 runs. The results are depicted in Table 6.1.

	Block Dude		Mini Grid	
Curriculum evaluations to generate estimated best curriculum	Count	Percentage	Count	Percentage
SourceTaskUtilityLazyDFS (ours)	387.64	66.15%	39	45.35%
RegretBeamSearch	360.91	61.59%	39.19	45.57%
PolicyChangeDFS	445.33	75.99%	40.58	47.19%
Maximum	**586**	**100%**	**86**	**100%**

Table 6.1: Evaluation statistics of our algorithm in the Block Dude and Mini Grid curriculum search spaces. Results are averaged over 50 runs.

From the results we observe that our algorithm evaluates about 66% of curricula in Block Dude and about 45% of curricula in Mini Grid before generating the estimated best curriculum. This observation raises a concern about the scalability of our algorithm, that is, how feasible it is for our algorithm to operate in large curriculum search spaces because of the memory or storage implications. Additionally, this observation raises an opportunity to study the effect of our algorithm's performance in varying curriculum search space sizes, either by varying the max curriculum length or varying the number of source tasks available to include in candidate curricula.

We also observe from the results that the baseline, `RegretBeamSearch`, requires the fewest number of evaluations in Block Dude and our algorithm requires the fewest number of evaluations in Mini Grid to generate the estimated best curriculum. Our objective is to minimize the cost of intermediate training (measured in steps) to generate high quality curricula. These findings are therefore not indicative of good performance in our context since the selection of candidate curricula impacts the cost of intermediate training, where certain curricula could incur a greater cost. The results in Sections 6.4.3

and 6.5 are indicative of good algorithm performance in our context and these results serve simply to understand memory and storage implications.

We conclude from our results that our algorithm evaluates a significant percentage of the curriculum search space in each domain to generate the estimated best curriculum. This finding raises an opportunity to study the effect of our algorithm on varying curriculum search space sizes.

6.7 Conclusion

In this chapter we presented preliminary experimentation which provided insights for the formulation of our algorithm. We then analyzed our algorithm by measuring its efficiency when restricted by a training budget and showed that it outperformed baseline methods. Finally, we studied the memory and storage implications of our algorithm.

We summarize our findings as follows:

- we conclude that certain source tasks may be biased towards constructing good curricula for a learning agent and that they may be identified online during training (Section 6.3).
- we hypothesize that good source tasks and curricula can be created by exposing the agent to certain aspects of the target task (Section 6.3).
- there is opportunity to study the effect of target class size and how to subdivide it effectively for training and testing purposes in Curriculum Learning (Section 6.4).
- there is opportunity to study how to improve curriculum utility estimates (Section 6.4).
- we conclude that heuristics which leverage training history make more informed decisions about candidate curricula and outperform those that do not (Sections 6.4 and 6.5).
- there is opportunity to study the effect of curriculum search space size on our algorithm (Section 6.6).

Chapter 7

Conclusion and Future Work

In this work we studied the cost of curriculum generation to minimize the time required to design high-quality curricula. We began by proposing a measure of the cost of curriculum generation as the intermediate training time required to evaluate candidate curricula, restricted this cost by enforcing a training budget, and stated our objective to design a curriculum generation algorithm which constructs high-quality curricula under the restriction of a training budget, as described in Section 5.1. We then introduced a curriculum generation algorithm that efficiently utilizes the training budget to generate high-quality curricula, as described in Section 5.2. In our experimentation, we began by demonstrating how individual source tasks may be biased to construct good curricula, a finding which inspired a heuristic function that learns source task utility online, as described in Section 6.3. We then evaluated various common and handcrafted heuristics and graph traversal algorithms and selected a combination with superior performance for our algorithm, as detailed in Section 6.4. Lastly, evaluated our algorithm in Block Dude with a single target task and in Mini Grid with a class of target tasks, teaching our agent to solve a class of target tasks and thereby amortizing the cost of curriculum generation, as detailed in section 5.2. We observed how our algorithm is able to generate at least the same or higher quality curricula for various training budgets as compared to baseline methods.

We formulated the problem of curriculum generation as a search problem where our algorithm searches and evaluates a space of candidate curricula. The heuristic used to guide exploration in the space and the search algorithm used were evaluated and selected as the best of various common and handcrafted options from preliminary work. An opportunity for future work is to instead learn a heuristic function using a machine learning algorithm, where domain properties, agent properties, and feedback from intermediate training could be used to train the algorithm.

We defined a curriculum as a sequence of tasks without repetitions, for simplicity, and trained the agent on a curriculum by sequentially training the agent to convergence on each source task in the curriculum. Another opportunity for future work is to explore an alternative definition of a curriculum allowing for repeated tasks, and an alternative training strategy for curricula, not necessarily training the agent to convergence on each source task, which could improve curriculum generation efficiency.

During training we observed that our algorithm (and likely other Curriculum Learning literature which our work is based upon [Narvekar *et al.* 2017; Foglino *et al.* 2019b]) inaccurately estimates curriculum utility, albeit not wildly inaccurately. This is a potential area of improvement since even with a

sufficiently large training budget, the algorithm will always generate a suboptimal curriculum. We therefore identify an opportunity to study how to improve curriculum utility estimates.

We evaluated our algorithm in Mini Grid with a class of target tasks, teaching an agent to solve a class of 20 target tasks. We identify an opportunity to study the effect of target class size and training and testing split, specifically in the context of Curriculum Learning for Reinforcement Learning agents.

Lastly, our algorithm avoids redundant training of source tasks by reusing learned policies. This is made possible because candidate curricula are structured in a search tree. A limitation of this approach is its memory or storage implications. We identify an opportunity to study the effect of curriculum search space size on our algorithm, studying both our algorithm's performance and memory or storage consumption.

Bibliography

[Andre and Russell 2002] David Andre and Stuart J Russell. State abstraction for programmable reinforcement learning agents. In *Aaai/iaai*, pages 119–125, 2002.

[Andrychowicz *et al.* 2017] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in neural information processing systems*, pages 5048–5058, 2017.

[Asada *et al.* 1994] Minoru Asada, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda. Vision-based behavior acquisition for a shooting robot by using a reinforcement learning. In *Proc. of IAPR/IEEE Workshop on Visual Behaviors*, pages 112–118. Citeseer, 1994.

[Asada *et al.* 1996] Minoru Asada, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda. Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Machine learning*, 23(2):279–303, 1996.

[Asadi and Huber 2007] Mehran Asadi and Manfred Huber. Effective control knowledge transfer through learning skill and representation hierarchies. In *IJCAI*, volume 7, pages 2054–2059, 2007.

[Baker *et al.* 2019] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula. *arXiv preprint arXiv:1909.07528*, 2019.

[Baldassarre and Mirolli 2013] Gianluca Baldassarre and Marco Mirolli. *Intrinsically motivated learning in natural and artificial systems*. Springer, 2013.

[Bansal *et al.* 2018] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. In *International Conference on Learning Representations*, 2018.

[Bengio *et al.* 2009] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[Brown University 2014] Brown University. *Brown-UMBC Reinforcement Learning and Planning (BURLAP) java code library*, 2014.

[Burda *et al.* 2018] Yuri Burda, Harri Edwards, Deepak Pathak, Amos Storkey, Trevor Darrell, and Alexei A Efros. Large-scale study of curiosity-driven learning. *arXiv preprint arXiv:1808.04355*, 2018.

[Chevalier-Boisvert *et al.* 2018] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. *Minimalistic Gridworld Environment for OpenAI Gym*. `https://github.com/maximecb/gym-minigrid`, 2018.

[Diuk *et al.* 2008] Carlos Diuk, Andre Cohen, and Michael L Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pages 240–247, 2008.

[Dorigo *et al.* 1996] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

[Fachantidis *et al.* 2013] Anestis Fachantidis, Ioannis Partalas, Grigorios Tsoumakas, and Ioannis Vlahavas. Transferring task models in reinforcement learning agents. *Neurocomputing*, 107:23–32, 2013.

[Fang *et al.* 2019] Meng Fang, Tianyi Zhou, Yali Du, Lei Han, and Zhengyou Zhang. Curriculum-guided hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 12623–12634, 2019.

[Ferguson and Mahadevan 2006] Kimberly Ferguson and Sridhar Mahadevan. Proto-transfer learning in markov decision processes using spectral methods. *Computer Science Department Faculty Publication Series*, page 151, 2006.

[Fernández and Veloso 2006] Fernando Fernández and Manuela Veloso. Probabilistic policy reuse in a reinforcement learning agent. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 720–727, 2006.

[Fernández *et al.* 2010] Fernando Fernández, Javier García, and Manuela Veloso. Probabilistic policy reuse for inter-task transfer learning. *Robotics and Autonomous Systems*, 58(7):866–871, 2010.

[Florensa *et al.* 2017] Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. Reverse curriculum generation for reinforcement learning. In *Conference on robot learning*, pages 482–495. PMLR, 2017.

[Florensa *et al.* 2018a] Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In *International Conference on Machine Learning (ICML)*, 2018.

[Florensa *et al.* 2018b] Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In *International conference on machine learning*, pages 1515–1528. PMLR, 2018.

[Foglino *et al.* 2019a] F Foglino, C Coletto Christakou, R Luna Gutierrez, and M Leonetti. Curriculum learning for cumulative return maximization. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 2308–2314. IJCAI, 2019.

[Foglino *et al.* 2019b] Francesco Foglino, Christiano Coletto Christakou, and Matteo Leonetti. An optimization framework for task sequencing in curriculum learning. In *2019 Joint IEEE 9th International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob)*, pages 207–214. IEEE, 2019.

[Foster and Dayan 2002] David Foster and Peter Dayan. Structure in the space of value functions. *Machine Learning*, 49(2):325–346, 2002.

[Glover and Laguna 1998] Fred Glover and Manuel Laguna. Tabu search. In *Handbook of combinatorial optimization*, pages 2093–2229. Springer, 1998.

[Goldberg 1989] David E Goldberg. Genetic algorithms in search. *Optimization, and MachineLearning*, 1989.

[Goodfellow *et al.* 2014] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C Courville, and Yoshua Bengio. Generative adversarial nets. In *NIPS*, 2014.

[Insa-Cabrera *et al.* 2011] Javier Insa-Cabrera, David L Dowe, and José Hernández-Orallo. Evaluating a reinforcement learning algorithm with a general intelligence test. In *Conference of the Spanish Association for Artificial Intelligence*, pages 1–11. Springer, 2011.

[Jain and Tulabandhula 2017] Vikas Jain and Theja Tulabandhula. Faster reinforcement learning using active simulators. *arXiv preprint arXiv:1703.07853*, 2017.

[James *et al.* 2020] Steven James, Benjamin Rosman, and George Konidaris. Learning portable representations for high-level planning. In *International Conference on Machine Learning*, pages 4682–4691. PMLR, 2020.

[Kim and Choi 2018] Tae-Hoon Kim and Jonghyun Choi. Screenernet: Learning self-paced curriculum for deep neural networks. *arXiv preprint arXiv:1801.00904*, 2018.

[Lazaric and Restelli 2011] Alessandro Lazaric and Marcello Restelli. Transfer from multiple mdps. *Advances in Neural Information Processing Systems*, 24:1746–1754, 2011.

[Lazaric *et al.* 2008] Alessandro Lazaric, Marcello Restelli, and Andrea Bonarini. Transfer of samples in batch reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pages 544–551, 2008.

[Lazaric 2012] Alessandro Lazaric. Transfer in reinforcement learning: a framework and a survey. In *Reinforcement Learning*, pages 143–173. Springer, 2012.

[Marom and Rosman 2018] Ofir Marom and Benjamin Rosman. Zero-shot transfer with deictic object-oriented representation in reinforcement learning. In *NeurIPS*, pages 2297–2305, 2018.

[Matiisen *et al.* 2019] Tambet Matiisen, Avital Oliver, Taco Cohen, and John Schulman. Teacher–student curriculum learning. *IEEE transactions on neural networks and learning systems*, 31(9):3732–3740, 2019.

[Mehta *et al.* 2008] Neville Mehta, Sriraam Natarajan, Prasad Tadepalli, and Alan Fern. Transfer in variable-reward hierarchical reinforcement learning. *Machine Learning*, 73(3):289, 2008.

[Mnih *et al.* 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[Narvekar and Stone 2019] Sanmit Narvekar and Peter Stone. Learning curriculum policies for reinforcement learning. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 25–33, 2019.

[Narvekar *et al.* 2016] Sanmit Narvekar, Jivko Sinapov, Matteo Leonetti, and Peter Stone. Source task creation for curriculum learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 566–574, 2016.

[Narvekar *et al.* 2017] Sanmit Narvekar, Jivko Sinapov, and Peter Stone. Autonomous task sequencing for customized curriculum design in reinforcement learning. In *The 2017 International Joint Conference on Artificial Intelligenge (IJCAI)*, 2017.

[Narvekar *et al.* 2020] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *arXiv preprint arXiv:2003.04960*, 2020.

[Ow and Morton 1988] Peng Si Ow and Thomas E Morton. Filtered beam search in scheduling. *The International Journal Of Production Research*, 26(1):35–62, 1988.

[Pathak *et al.* 2017] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning*, pages 2778–2787. PMLR, 2017.

[Perkins *et al.* 1999] Theodore J Perkins, Doina Precup, et al. *Using options for knowledge transfer in reinforcement learning*. Technical report, Technical Report UM-CS-1999-034, The University of Massachusetts at Amherst, 1999.

[Pinto *et al.* 2017] Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. Robust adversarial reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 2817—2826, 2017.

[Puterman 1990] Martin L Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

[Ren *et al.* 2018] Zhipeng Ren, Daoyi Dong, Huaxiong Li, and Chunlin Chen. Self-paced prioritized curriculum learning with coverage penalty in deep reinforcement learning. *IEEE transactions on neural networks and learning systems*, 29(6):2216–2226, 2018.

[Riedmiller *et al.* 2018] Martin A Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Van de Wiele, Vlad Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing solving sparse reward tasks from scratch. In *ICML*, 2018.

[Rosman and Ramamoorthy 2012] Benjamin Rosman and Subramanian Ramamoorthy. What good are actions? accelerating learning using learned action priors. In *2012 IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL)*, pages 1–6. IEEE, 2012.

[Russell and Norvig 2002] Stuart Russell and Norvig. *Artificial intelligence: a modern approach*. Prentice Hall Upper Saddle River, NJ, USA, 2002.

[Schaul *et al.* 2016] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *International Conference on Learning Representations (ICLR)*, 2016.

[Schmidhuber 2013] Jürgen Schmidhuber. Powerplay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in psychology*, 4:313, 2013.

[Selfridge *et al.* 1985] Oliver G Selfridge, Richard S Sutton, and Andrew G Barto. Training and tracking in robotics. In *Ijcai*, pages 670–672, 1985.

[Shao *et al.* 2018] Kun Shao, Yuanheng Zhu, and Dongbin Zhao. Starcraft micromanagement with reinforcement learning and curriculum transfer learning. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 3(1):73–84, 2018.

[Silva and Costa 2018] Felipe Leno Da Silva and Anna Helena Reali Costa. Object-oriented curriculum generation for reinforcement learning. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1026–1034, 2018.

[Silver *et al.* 2016] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[Soni and Singh 2006] Vishal Soni and Satinder Singh. Using homomorphisms to transfer options across continuous reinforcement learning domains. In *AAAI*, volume 6, pages 494–499, 2006.

[Sukhbaatar *et al.* 2017] Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. In *International Conference on Learning Representations (ICLR)*, 2017.

[Sutton and Barto 1999] Richard S Sutton and Andrew G Barto. Reinforcement learning. *Journal of Cognitive Neuroscience*, 11(1):126–134, 1999.

[Sutton *et al.* 1999] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[Svetlik *et al.* 2017] Maxwell Svetlik, Matteo Leonetti, Jivko Sinapov, Rishi Shah, Nick Walker, and Peter Stone. Automatic curriculum graph generation for reinforcement learning agents. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[Talvitie and Singh 2007] Erik Talvitie and Satinder P Singh. An experts algorithm for transfer learning. In *IJCAI*, pages 1065–1070, 2007.

[Tang *et al.* 2017] Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. # exploration: A study of count-based exploration for deep reinforcement learning. In *31st Conference on Neural Information Processing Systems (NIPS)*, volume 30, pages 1–18, 2017.

[Taylor and Stone 2009] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009.

[Taylor *et al.* 2007] Matthew E Taylor, Peter Stone, and Yaxin Liu. Transfer learning via inter-task mappings for temporal difference learning. *Journal of Machine Learning Research*, 8(9), 2007.

[Tesauro 1995] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[Thrun and Möller 1991] Sebastian B Thrun and Knut Möller. *On planning and exploration in non-discrete environments*. Citeseer, 1991.

[Thrun *et al.* 1991] Sebastian B Thrun, Knut Möller, and A Linden. Active exploration in dynamic environments. In *NIPS*, pages 531–538, 1991.

[Vinyals *et al.* 2019] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[Wilson *et al.* 2007] Aaron Wilson, Alan Fern, Soumya Ray, and Prasad Tadepalli. Multi-task reinforcement learning: a hierarchical bayesian approach. In *Proceedings of the 24th international conference on Machine learning*, pages 1015–1022, 2007.

[Wu and Tian 2017] Yuxin Wu and Yuandong Tian. Training agent for first-person shooter game with actor-critic curriculum learning. In *International Conference on Learning Representations (ICLR)*, 2017.

[Wu *et al.* 2018] Yuechen Wu, Wei Zhang, and Ke Song. Master-slave curriculum design for reinforcement learning. In *IJCAI*, pages 1523–1529, 2018.