Using the Online Cross-Entropy Method to Learn Relational Policies for Playing Different Games

Samuel Sarjant, Bernhard Pfahringer, Kurt Driessens and Tony Smith

***Abstract*—By defining a video-game environment as a collection of objects, relations, actions and rewards, the relational reinforcement learning algorithm presented in this paper generates and optimises a set of concise, human-readable relational rules for achieving maximal reward. Rule learning is achieved using a combination of incremental specialisation of rules and a modified online cross-entropy method, which dynamically adjusts the rate of learning as the agent progresses. The algorithm is tested on the Ms. Pac-Man and Mario environments, with results indicating the agent learns an effective policy for acting within each environment.**

I. Introduction

There are countless AI algorithms for playing specific games (such as Tesauro's backgammon agent [1] and Samuel's checkers playing agent [2]), but general AI algorithms for playing a range of different games are rare. By using a representation of objects, relations, actions and rewards for defining games, *relational reinforcement learning* (RRL) algorithms can be used to learn behaviour for operating in games. RRL defines environments as groups of objects with relations and an agent takes actions that directly interact with those objects in pursuit of maximising a numerical reward signal. RRL allows dynamic environments and scalable parameterisable behaviour that regular reinforcement learning cannot handle without extensive abstraction. General Game Playing [3], the design of AI which can play many formally defined games, is strongly related to RRL, but it provides a full formal model of the game, which RRL may not do.

This paper describes CERRLA (Cross-Entropy Relational Reinforcement Learning Agent), a policy-search relational reinforcement learning agent capable of learning useful, human-readable behaviour in a range of relational environments without the requirement of a formal environment model. Because a relational environment can be of any size, CERRLA uses a policy-search algorithm (like GREY [4] or TREENPPG [5]) to search the space of policies (decision lists of rules) directly, as opposed to value-based algorithms (such as TRENDI [6] or MARLIE [7]) which learn a value function to approximate expected state reward. Abstraction can be used to reduce the state space, but that often requires prior knowledge — something that is not always known.

CERRLA's method of learning was initially based on a paper by Szita and Lörincz [8], but has since been expanded to relational domains and general environments. The cross-entropy method has previously been successfully applied to both Tetris [9], [10] and Ms. Pac-Man [8], though in each case the learning algorithm was optimised to match the environment.

CERRLA uses the cross-entropy method to search a number of parallel distributions of relational rules to probabilistically generate policies for acting effectively in environments. The agent creates rules by first determining the *relative least general generalisation* (RLGG) rules for each action and then searching incrementally specialised versions. These rules are then combined into a policy using the cross-entropy method for selection and ordering. The cross-entropy method is also used for guided specialisation of rules, to focus on interesting rules. The resulting converged distributions output low complexity, easy-to-read relational policies which obtain large rewards when evaluated in the environment.

II. Background

In order to properly explain the learning method CERRLA employs, some background concepts must first be explained.

A. Relational Reinforcement Learning

Relational reinforcement learning (RRL) is an extension of reinforcement learning (RL) in which an agent seeks to maximise a numerical reward by interacting with its environment through a number of actions [11]. RRL differs from RL by defining the environment as first-order logical relations between objects and actions taken upon those objects. RL problems can be formalised using the *Markov Decision Process* (MDP) framework, where the agent receives the *state* and the *actions* it can take from that state. The agent selects an action which changes the state based on the *transition function* and generates a *reward value* which is passed to the agent with the next state observation [12]. This continues until a terminal state within the environment is reached. Action selection is controlled using a *policy* $\pi : s \rightarrow a$, which when given a state s, returns an action a to take.

Relational environments may still be formalised using the MDP framework, but the state and action formalisms are now defined via relational predicates concerning objects present in the state (e.g. $observation(cat, hat)$, $action(cat)$). Each predicate's definition includes a declarative bias to constrain the object *types* it can take as arguments. The agent is provided with the environment predicate definitions, rather than being required to learn them.

Relational rules are defined as $condition_1, \ldots, condition_i \rightarrow action$, where $condition$ is a relation acting

Samuel Sarjant, Bernhard Pfahringer and Tony Smith are with the Faculty of Computing and Mathematical Sciences at The University of Waikato, New Zealand. Kurt Driessens is with the Department of Knowledge Engineering at Maastricht University, The Netherlands. Samuel Sarjant is the corresponding author. Email: sam.sarjant@gmail.com.

978-1-4577-0011-8/11/$26.00 ©2011 IEEE

Fig. 1. A screenshot of a portion of the Ms. Pac-Man environment.

TABLE I
THE PREDICATES DEFINITIONS FOR MS. PAC-MAN.

Observations	
$distance(Thing, \#D)$	$Thing$ is $\#D$ units from Ms. Pac-Man.
$edible(Ghost)$	$Ghost$ is edible.
$blinking(Ghost)$	$Ghost$ is blinking.
$junctionSafety(Junction, \#J)$	$Junction$ has safety value $\#J$.
Actions (where $\#D$ ($distance$) and $\#J$ ($junctionSafety$) are meta-information for resolving actions)	
$moveTo(Thing, \#D)$	Move towards $Thing$.
$moveFrom(Thing, \#D)$	Do not move to $Thing$.
$toJunction(Junction, \#J)$	Move to $Junction$.
Type Hierarchy	
$Thing \leftarrow Ghost$; $Fruit$; Dot; $PowerDot$; $GhostCentre$.	All objects are $Things$.
$Junction$	An intersection of paths.

upon one or more objects and $action$ is an action acting upon one or more objects defined by the conditions. An example rule is $cond_a(X, Y), cond_b(Y, sam) \rightarrow action_a(X, sam)$. In this rule there are two variable arguments X and Y, and a definite object sam. This rule also has implicit assertions: objects of a different name are unequal, and each object has a type assertion. For example, if $cond_a$ takes a $Thing$ and a $Gizmo$ as its first and second arguments, then $thing(X)$ and $gizmo(Y)$ are hidden type conditions. An object can have more than one type assertion, but only if the types are able to co-exist (e.g. $thing(X), gizmo(X)$ can only be true if $thing(X) \rightarrow gizmo(X)$ or vice-versa).

The main benefits of RRL over RL are: (1) states are flexible collections of objects and relations, (2) available actions can be automatically inferred using rules, (3) first-order variable generalisation allows general behaviour over similar objects rather than single objects, and (4) background knowledge may be defined to easily create new relations or rule out illegal states in an environment. However, the first-order structure also introduces new problems. Because states are dynamic, there is an enormous number of possible states, which becomes impossible to model with a brute force state-action table (hence the need for parameterisable variables), and first-order reasoning is generally slower than feature-based methods.

B. Environments

The agent has a number of episodic testing environments, where the agent is allowed a finite number of steps to reach the goal. In order for CERRLA to interact with an environment, a wrapper class must be provided which can extract and transform relational objects, relations and actions.

In each environment the agent selects actions using a relational rule policy created at the beginning of the episode (detailed in Section III). At the beginning of an experiment, the environment predicates are initialised and made available to the agent. State observations are asserted using the objects and relations currently present in the state and relational actions are evaluated by transforming them into the appropriate low-level action in the environment.

1) Ms. Pac-Man: Ms. Pac-Man is the (unauthorised) sequel to the famous Pac-Man arcade video game.[1] The goal of the game is for Ms. Pac-Man (the agent) to eat all dots in the level while avoiding the ghosts. Ms. Pac-Man has four simple directional actions, though these are abstracted into higher level actions for the purpose of learning strategic behaviour. There are four hostile ghosts with individual behaviour, each released periodically from their cage. Unlike Pac-Man, the ghosts in Ms. Pac-Man have a 25% chance of choosing a non-default behaviour direction at a junction (but cannot turn directly back) so a level cannot be completed by taking a predefined sequence of actions. The ghosts can be eaten for a short period of time after eating a powerdot, giving a large score bonus for each ghost eaten during this time. When a ghost is eaten, it returns to the cage and is released again after a short time. Also, once per level, a fruit appears for a limited time which can be eaten for a large score bonus.

Ms. Pac-Man works well as a reinforcement learning problem because it has a fully observable state, an obvious reward signal (the score), and non-deterministic ghost behaviour, making planning algorithms ineffective.

The relational state description for Ms. Pac-Man is given in Table I. The actions are defined as high-level move towards/from actions and a 'to junction' action with two arguments: the object to act upon, and the distance/junction safety of the object. Ms. Pac-Man's low-level (directional) movement is determined by evaluating the policy rule-by-rule. When a rule is evaluated, the object that is closest or has highest junction safety determines the direction (towards or from). If the rule fails to select a single direction (multiple objects of equal distance), the next rule is used to determine direction. If no single direction is reached, either take the same direction as last step, or a perpendicular direction.

The $distance(Thing, \#)$ predicate is defined as the length of Ms. Pac-Man's shortest path to the $Thing$. The $junctionSafety(Junction, \#)$ predicate is defined as the shortest distance between the $Junction$ and nearest $Ghost$ minus the distance between the $Junction$ and Ms. Pac-Man. E.g. a $JunctionSafety(Junction, 4)$ implies Ms. Pac-Man can reach the junction four steps prior to the nearest ghost.

[1]Pac-Man and Ms. Pac-Man are trademark Namco Bandai Games.

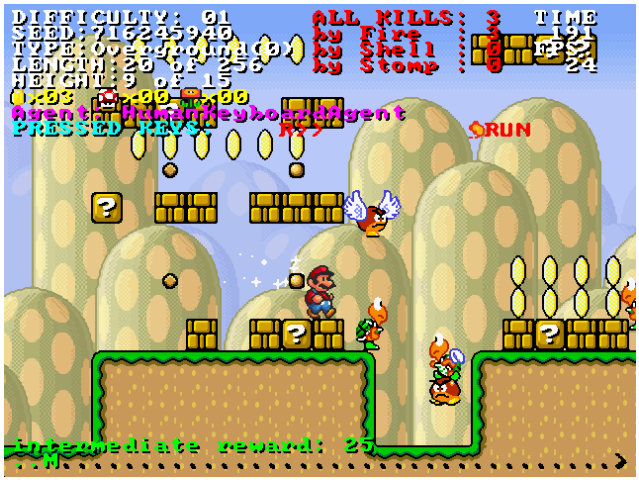

Fig. 2. A screenshot of the Mario environment.

In order for the agent to act effectively, each action will require that the $Thing$ being acted upon will be bound to a specific type (e.g. $dot(X), distance(X, D) \rightarrow moveTo(X, D)$). This is handled in Section III-A.

2) Mario: The Mario environment uses Infinite Mario, an open-source clone of the Super Mario Bros.[2] video-game. The agent is in control of Mario, who must traverse a fixed-length two-dimensional level of hazards in an attempt to reach the princess within a finite time period. Infinite Mario is able to randomly generate a completable level of varying levels of difficulty, so the agent must be able to handle many different situations. Each level consists of terrain of varying heights (including deadly pits), a number of different types of enemies, searchable bricks, and collectable coins and powerups. Mario has three modes (in decreasing order): fire, large and small. Whenever Mario is hit by an enemy, his mode decreases. Mario can increase his mode by searching a Box (by jumping into it from underneath), and collecting a $Mushroom$ or $FireFlower$ that may come out.

Mario can dispatch enemies by jumping on them (except for $PiranhaPlant$ and $Spiky$) or, if in fire mode, shooting a fireball (except for $Spiky$). If an enemy has wings (is $flying$), jumping on them only removes the wings. When a $Koopa$ is jumped on, it leaves behind a $Shell$ which can be picked up and fired to destroy $Enemies$ and $Bricks$.

The agent's reward is based on a combination of factors (enemies killed, items collected, time remaining, distance travelled, Mario's mode, etc.). The reward calculation formula is given in Section IV-2.

Mario's relational state description is given in Table II. $canJumpOnto$ and $canJumpOver$ are defined as objects that are feasibly in Mario's jump range from Mario's last grounded position (but not directly above). Mario's $move$ action may entail jumping over solid obstacles in the way of getting to the object. Mario's relational jumping actions involve $move$ing close enough to be able to jump onto/over the object and jumping long enough to complete the action (holding jump until Mario is both above and halfway closer to the object from his starting position).

[2]Super Mario Bros. was developed by Nintendo for the Nintendo Entertainment System.

TABLE II
THE PREDICATES DEFINITIONS FOR MARIO.

Observations	
$distance(Thing, \#D)$	$Thing$ is Euclidean $\#D$ from Mario. The sign of $\#D$ indicates direction.
$heightDiff(Thing, \#H)$	$Thing$ is $\#H$ units above/below Mario. The sign of $\#H$ indicates direction.
$canJumpOnto(Thing)$	Mario can feasibly jump onto $Thing$.
$canJumpOver(Thing)$	Mario can feasibly jump over $Thing$.
$flying(Enemy)$	$Enemy$ has wings.
$squashable(Enemy)$	$Enemy$ can be jumped on.
$blastable(Enemy)$	$Enemy$ can be shot with fireball.
$width(Pit, \#W)$	The size ($\#W$) of a Pit.
$carrying(Shell)$	If Mario is carrying $Shell$.
$passive(Shell)$	If $Shell$ is not moving.
Actions (where $\#D$ ($distance$) and $\#W$ ($width$) are meta-information for resolving actions)	
$move(Thing, \#D)$	Move towards $Thing$.
$search(Brick, \#D)$	Search $Brick$ (hit from beneath).
$jumpOnto(Thing, \#D)$	Jump onto $Thing$.
$jumpOver(Thing, \#D, \#W)$	Jump over $Thing$ of width $\#W$. $\#W = 1$ if undefined.
$pickup(Shell, \#D)$	Picks up a $Shell$.
$shootFireball(Enemy, \#D, MarioPower)$	Shoot $Enemy$ with a fireball (only when $MarioPower = fire$).
$shootShell(Enemy, \#D, Shell)$	Shoot $Enemy$ with a held $Shell$.
Type Hierarchy	
$Thing \leftarrow Brick$; $Enemy$; $Item$; $Goal$; Pit; $Shell$.	All objects are $Things$.
$Enemy \leftarrow Goomba$; $Koopa$; $PiranhaPlant$; $Spiky$; $BulletBill$.	Various $Enemy$ types.
$Koopa \leftarrow GreenKoopa$; $RedKoopa$.	Two types of $Koopa$.
$Item \leftarrow Mushroom$; $Coin$; $FireFlower$.	Mario can collect these.
$MarioPower$	Mario's modes: $fire$, $large$ or $small$.

The first rule to be triggered in the policy determines Mario's action. Mario acts on the closest object the rule defines. If multiple objects are closest, Mario uses the next rule to act. No decision will result in no action.

Mario also has a large type hierarchy, which can be used to guide the actions towards general (e.g. $Enemy$) or specific (e.g. $RedKoopa$) types of objects. For distance calculations, the $Goal$ is defined to be just out of view to the right.

C. Cross-Entropy Method

The cross-entropy (CE) method [13] is an optimisation algorithm which maintains a distribution of possible solutions to a problem. The CE method is used in parallel to maintain multiple distributions of relational rules for use in generating relational policies. A more in-depth review of the CE method is given by De Boer et al. [14].

The CE method consists of two main phases:

1) Generate N samples from the distribution.
2) Update the distribution based on the best N_E samples to increase the probability of sampling that data again, where N_E is a small proportion of the N samples.

Initially, the set of data $X = \{x_1, \ldots, x_n\}$ (rules) is stored within a distribution with probabilities $\mathbf{p}_0(X) :=$

TABLE III
PSEUDO-CODE SUMMARY OF CERRLA. SEE TEXT FOR FULL EXPLANATION.

1	initEnvironment($game$)	Initialise predicate definitions
2	$D_S := \{\}$	The slot distribution
3	$samples := \{\}$	The sample collection
4	$n := 0$	# policies evaluated
5	repeat	
6	π_n := generatePolicy(D_S)	Generate policy from slots
7	repeat	
8	s := observeState()	Relational state observations
9	π_n := maybeResample()	Resample if not progressing
10	a := selectAction(π_n, s)	Policy determines action
11	takeAction(a)	Evaluate relational action
12	until episodeComplete	Until terminal state
13	r_n := Environment.reward()	Note reward
14	$samples$.add(π_n, r_n)	Add the sample to collection
15	N_E := determineMinEliteSize(D_S)	Determine minimum # elites
16	if $n \geq 2 \cdot N_E$	Update after $2 \cdot N_E$ samples
17	$samples$.remove($\pi_i : r_i < r_{N_E}$)	Remove non-elite samples
18	$\alpha' := \alpha / max(N - n, N_E)$	Initially low α'
19	update(D_S, $samples$, α')	Step-wise update
20	maybeSpecialiseNewRules(D_S)	Specialise interesting rules
21	until converged(D_S)	Determined by sum updates

$\{p_1, \ldots, p_n\}$. The distribution is then sampled N times ($\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}$), selecting data based on its (initially equal) probability where $\mathbf{x}^{(i)} = x_j$ with probability p_j. The samples are then tested with function $f(\mathbf{x})$ and sorted in descending order. N_E 'elite' subsamples $E = \{\mathbf{e}^{(1)}, \ldots, \mathbf{e}^{(N_E)}\}$ are then extracted from the samples (where $\mathbf{e}^{(i)} = \mathbf{x}^{(j)}$). Elite samples are samples with $f(\mathbf{x}^{(j)}) \geq f(\mathbf{x}^{\rho \cdot N})$ where typically $\rho := 0.05$.

The observed distribution $\mathbf{p}'(X) := \{p'_1, \ldots, p'_n\}$ is then calculated using the frequency of data seen within the elite samples, given by:

$$p'_j := \Big(\sum_{\mathbf{e}^{(i)} \in E} \left\{ \begin{array}{ll} 1 & \text{if } \mathbf{e}^{(i)} = x_j \\ 0 & \text{otherwise} \end{array} \right\} \Big) \Big/ N_E$$

which means that p'_j is equal to the count of all elite samples that represent data x_j divided by the total number of elite samples N_E. For example, if half of the samples in E are the same value x_j, $p'_j = 0.5$.

To allow for variance across the iterations of the algorithm, the distribution is updated in a step-wise fashion, using α (typically $\alpha := 0.6$) to modify the distribution probabilities:

$$p_j := \alpha \cdot p'_j + (1 - \alpha) \cdot p_j \quad (1)$$

The idea is that every update will cause the distribution to produce useful data more often, eventually resulting in a converged distribution. The algorithm stops after a finite number of iterations, or when the distribution has converged.

The CE method can also optimise multiple distributions in parallel. Given a vector of size m, with each slot of the vector containing a distribution, we can test and update multiple distributions at once. The resulting converged distributions represent an optimal *combination* of data for the problem.

III. METHODOLOGY

CERRLA uses the cross-entropy method to optimise a set of automatically discovered, iteratively specialised relational rules for solving the environment problem as a form of direct policy search. The algorithm *explores* the set of relational rules for solving the problem and *exploits* high-achieving rules by increasing their probability of being selected. Initially, all rules are equally likely to be sampled, but over time better rules are sampled more frequently while maintaining a gradually smaller chance of exploring the other rules. If an agent does not have any rules for selecting an action, it creates a new one (Section III-A).

1) Policy Generation: In CERRLA, a sample is a policy π, which is generated from a distribution D_S of rule distributions (a collection of slots S, where each slot has probability p_S and size $|S|$). Each slot contains a distribution of relational rules with the same action predicate $\mathbf{a}$: $S^{\mathbf{a}} = \{r_1^{\mathbf{a}}, \ldots, r_n^{\mathbf{a}}\}$ where each rule r_j has the same action predicate $\mathbf{a}$ and probability p_j. Each slot also has two parameters: $\mu(S) \in [0, \infty]$ (initially $\mu(S) = 0.5$), the average number of times a slot is used per policy and $\sigma(S) \in [0, 0.5]$ (initially $\sigma(S) = 0.5$), the standard deviation of usage.

A policy is generated (step 6 in Table III) by sampling every slot from D_S, where slots with high p_S are likely to come first. When a slot is sampled, a Gaussian normally distributed value g_S with parameters $\mu(S)$ and $\sigma(S)$ represents the probability of the slot being used in the policy. $g_S \geq 1$ means the slot will be used at least once, but $g_S < 1$ mean the slot may not be used at all. For example, if $g_S = 1.3$, then S will be present in the policy at least once, or twice with 30% probability. Once a slot is sampled, a rule r_i is sampled from the slot using its probability p_i.

2) Policy Resampling: Step 7–12 in Table III outline the evaluation loop of the policy against the environment. Note that each policy is evaluated through 3 episodes and the reward is averaged to reduce variance.

At step 9, the policy may optionally be resampled if the agent is not progressing. Every episode, the agent records which states it has encountered. If the agent encounters a previously visited state, and is not receiving more than average reward (repeating the same state may be lucrative), a resampling variable $\chi \in [0, 1]$ is incremented by $(0.1 \cdot avSteps)^{-1}$, where $avSteps$ is the average observed number of steps per episode and 0.1 represents the maximum proportion of the episode the agent can visit the same states. χ is decremented by the same amount if a new state is encountered. χ represents the probability that the agent will resample the policy in that step (which then resets $\chi := 0$). The value of 0.1 was selected arbitrarily and further experimentation is required to find an ideal value.

The policy that completes the most steps within the episode is the one to which the episode's reward is associated. Policies could be associated with the reward obtained while they were active, but in environments where the reward is only received at the end of the episode, only the final policy would receive reward.

3) Online Cross-Entropy Method: Step 15 introduces a dynamic elite sample size method. Because the number of rules is constantly changing, the number of elites should

change to reflect this:

$$N_E = max\Big\lceil \frac{\sum_{S \in D_S} \big(\mu(S) \cdot |S|\big)}{\sum_{S \in D_S} \mu(S)}, \sum_{S \in D_S} \mu(S) \Big\rceil$$

and $N = N_E/\rho$. N_E is either equal to the average number of rules for relatively high $\mu(S)$ slots, or is equal to the sum of $\mu(S)$ across all slots (whichever is larger).

Step 16–20 represents a deviation from the standard CE method. Because the agent is frequently creating new rules and thus needs quick feedback on which rules are interesting, an online CE method is utilised. Szita and Lörincz [15] detail such a method which processes samples incrementally using a sliding window of N samples. This has been modified to process samples almost immediately.

After $2 \cdot N_E$ samples have been evaluated, the agent begins updating the distribution using a modified step-wise parameter α' (see step 18–19 in Table III). The agent then continues to sample policies and revising the elite samples, performing α' updates every iteration. Note that any samples in E which have been present for more than N steps are removed. After N updates, α' matches the α in Szita and Lörincz's online CE method.

4) Updating Probabilities: At step 19 the elite E samples are used to update the slot distribution D_S and rule distributions within each S in parallel as defined in Section II-C. When updating the slot distribution D_S, the observed probability p'_{S_j} for each S_j in D_S is calculated by first determining the average position of S_j within E:

$$q_{S_j} = 1 - \frac{1}{|E_{S_j}|} \sum_{\pi \in E(S_j)} \frac{index(S_j, \pi)}{|\pi|}$$

where $|\pi|$ is the size of π, $E(S_j)$ are the policies in E that utilise slot S_j, and $index(S_j, \pi) \in [0, |\pi|)$ returns the index of S_j in the policy, where 0 is first. If $E(S_j)$ is empty, $q_{S_j} = 0.5$ (the average position), because if S_j is not present in E, we cannot determine its position probability (though $\mu(S_j)$ will decrease). To form a valid probability distribution, q_S needs to be normalised:

$$p'_{S_j} = \frac{q_{S_j}}{\sum_{S \in D_S} q_S}$$

Because every slot is sampled when generating a new policy, p_S represents the probablity of the slot being sampled first.

The extra slot parameters are also step-wise updated like Equation (1): $\mu'(S)$ is the average number of times S appears per policy in E and $\sigma'(S)$ is the observed standard deviation of slot use in E.

In the update process, only utilised rules and slots in the policy are updated, therefore unused rules and slots are implicitly negatively updated, resulting in minimally sized policies containing only useful rules.

After the update process (step 20), the agent may create new rules, if a rule is ready to specialise (see Section III-A2).

The Kullback-Leibler divergence is used to determine when the optimisation has converged: when the divergence between all rule distribution updates and slot value updates is less than $\beta \cdot \alpha$ where $\beta = 0.01$ in experiments.

TABLE IV
PSEUDO-CODE SUMMARY OF THE RLGG PROCESS.

Input: r^a_{RLGG} (if one already exists)	% Existing RLGG rule
Input: s	% The current state
Input: $A(s)$	% The available actions in s
for each $a^{\mathbf{a}(\Phi)}$ in $A(s)$	% For every action
$C_\Phi :=$ facts in s regarding objects Φ	% Form the conditions
$r^{\mathbf{a}(\Phi)} := C_\Phi \rightarrow a^{\mathbf{a}(\Phi)}$	% Form the rule
if objects in $r^{\mathbf{a}}_{\text{RLGG}} \neq r^{\mathbf{a}(\Phi)}$	% Differing action arguments
swap objects for variables	
$r^{\mathbf{a}}_{\text{RLGG}} := r^{\mathbf{a}}_{\text{RLGG}} \cap r^{\mathbf{a}(\Phi)}$	% Create the intersection
end loop	

A. Creating Relational Rules

CERRLA's rule exploration is accomplished by first determining the relative least general generalization (RLGG) rule r^a_{RLGG} for every action a available for the agent to take, splitting the RLGG slot into sub-slots, and then gradually specialising new rules, using the rule probabilities as a guide for selecting which rules to specialise.

1) Creating the RLGG: The RLGG rules define the least general conditions for covering every possible action for every state. Whenever the agent encounters a state where the RLGG rules do not cover every possible action, the RLGG is generalised to cover the un-covered actions.

Given the set of available actions $A(s)$ for the state s where $a^{\mathbf{a}(\Phi)}$ is one of the actions with action predicate $\mathbf{a}$ and arguments Φ, a rule $r^{\mathbf{a}(\Phi)}$ can be created using $a^{\mathbf{a}(\Phi)}$ as the rule's action and all facts in the state regarding Φ as the rule conditions C_Φ.

An example Ms. Pac-Man case:

- $A(s) = \{moveTo(dot_1, 5), moveTo(ghost_1, 12), \ldots\}$;
- $a^{\mathbf{a}(\Phi)} = moveTo(dot_1, 5)$;
- $\mathbf{a} = moveTo$;
- $\Phi = dot_1, 5$;
- $r^{\mathbf{a}(\Phi)} = distance(dot_1, 5), dot(dot_1), \ldots \rightarrow moveTo(dot_1, 5)$.

The RLGG of every rule is computed by replacing all non-numerical instances of objects in Φ in each rule for parameterisable variables if the objects do not match. All other objects in the rules that do not match are replaced for an anonymous variable '?.' The rules are joined by intersection to become the RLGG rule. Any conditions containing only anonymous variables are removed from the resulting RLGG rule. Table IV summarises the algorithm in pseudo-code. See [16] for a more detailed definition of the RLGG process.

Numerical values are a special case when computing the RLGG. Numerical values are stored in a numerical range variable, which defines a minimum and maximum value.

For example, using the Ms. Pac-Man environment, assume the agent has at least three $moveTo$ actions that it can take at a given example state: $moveTo(g_1, 5)$, $moveTo(g_2, 10)$, and $moveTo(d_3, 8)$, where g_1 and g_2 are $ghosts$ and d_3 is a dot. Using only the observations regarding those objects

produces the following rules:

$$edible(g_1), distance(g_1, 5), ghost(g_1), thing(g_1) \rightarrow moveTo(g_1, 5). \quad (Rule_1)$$

$$distance(g_2, 10), ghost(g_2), thing(g_2) \rightarrow moveTo(g_2, 10). \quad (Rule_2)$$

$$distance(d_3, 8), dot(d_3), thing(d_3) \rightarrow moveTo(d_3, 8). \quad (Rule_3)$$

The RLGG for $Rule_1$ and $Rule_2$ is:

$$distance(\mathbf{X}, (5.0 \leq \mathbf{D} \leq 10.0)), ghost(\mathbf{X}), thing(\mathbf{X}) \rightarrow moveTo(\mathbf{X}, \mathbf{D}) \quad (RLGG_{1,2})$$

Combining this with $Rule_3$ gives us the minimally general rule for covering the three actions:

$$distance(\mathbf{X}, (5.0 \leq \mathbf{D} \leq 10.0)), thing(\mathbf{X}) \rightarrow moveTo(\mathbf{X}, \mathbf{D}) \quad (RLGG_{1,2,3})$$

2) Creating Specialised Rules: The RLGG operation only creates one minimally general rule for every action, but for more specialised behaviour, the agent needs more specialised rules. When a rule r is specialised, *all* possible single-step specialisations $\{r'_1, \ldots, r'_i\}$ are created using two specialisation methods: *guided specialisation* and *range splitting*. Each r' is added to the rule distribution with an initially average probability of being selected ($p_{r'} = |S|^{-1}$).

Guided specialisation creates new rules by incrementally adding particular conditions (or negated non-type conditions) to the rule. However, only conditions that have previously been present when the rule's action has been available will be added (as noted by the agent). For example, in the Mario environment, the $flying(X)$ condition will never be added to a $search(X, D)$ rule because whenever $search(X, D)$ is true, $flying(X)$ is never true. Examples of guided specialisation of $distance(X, D), thing(X) \rightarrow move(X, D)$ are:

$$distance(X, D), thing(X), flying(X) \rightarrow move(X, D);$$
$$distance(X, D), thing(X), enemy(X) \rightarrow move(X, D);$$
$$distance(X, D), thing(X), \neg passive(X) \rightarrow move(X, D).$$

A problem with guided specialisation is that it may introduce illegal ($enemy(X), coin(X) \rightarrow action(X)$; X cannot be both enemy and coin) or redundant rules ($canJumpOver(X), canJumpOn(X) \rightarrow action(X)$; If Mario can jump over X he can obviously jump on to X too). To stop these rules from being created, the agent learns a basic partial model of associations between state observations to infer a set of rules that define which state observations are *always* true, *never* true, and *occasionally* true when a particular observation is true (e.g. whenever $canJumpOver(X)$ is true, $canJumpOn(X)$ is also always true). Equivalence rules can also be created by combining inference rules to simplify the state space (e.g. $B \Leftrightarrow \neg A$). The agent can also utilise any known environment background knowledge.

Range splitting creates specialised rules by splitting an existing range of size r into an arbitrary number of smaller subranges. The splitting mechanic creates three main subranges: $(min) \cdots (min + \frac{r}{2})$, $(min + \frac{r}{2}) \cdots (max)$, and $(min + \frac{r}{4}) \cdots (min + \frac{3r}{4})$. Also, if the range includes 0.0, subranges are created for each side: $min \cdots 0.0$ and $0.0 \cdots max$. For example, the specialisation of the range $(-10.0 \leq D \leq 15.0)$ produces $(-10.0 \leq D \leq 2.5)$, $(2.5 \leq D \leq 15.0)$, $(-3.75 \leq D \leq 8.75)$, $(-10.0 \leq D \leq 0.0)$, and $(0.0 \leq D \leq 15.0)$.

3) Slot Splitting: Depending on the conditions defining it, an agent's action can have very different consequences, based on the objects it is acting upon. For example, in Ms. Pac-Man, the $moveTo$ action can be beneficial when moving to $dots$ or $edible$ ghosts, but unhelpful when moving to a hostile $ghost$. Though specialisations of the $moveTo$ RLGG rule can be created to act appropriately, all $moveTo$ rules are contained within the same slot, generally resulting in only a single $moveTo$ rule dominating the slot.

Slot splitting is achieved by using the immediate guided specialisations of each RLGG rule as seeds for each split slot (such that every rule in the split slots will contain a common, non-RLGG rule condition). This results in each slot defining a common sub-behaviour of the action (e.g. moving towards $dots$). By shifting part of the rule learning process into the slot distribution, the agent can focus on learning specialised behaviour for a particular form of action.

B. Controlling Rule Specialisations

The agent now has the tools to create rules, but the question is now *when* should the rules be created? The RLGG rules are created/modified when they do not cover all actions, but creating all specialisations at once swamps the agent and results in a brute force search over every possible solution.

Whenever an RLGG rule is created/modified, or the set of possible guided specialisations changes, the slot splitting procedure is called to generate/modify slots for each single-step specialisation of the RLGG rule. Each slot is then filled with a further set of specialisations (both guided and range splitting) using the slot's seed (or RLGG) rule as the rule to specialise. Pre-existing rules within modified slots that are no longer valid (according to the agent's beliefs) are removed. Similar to *beam search* [17], useful rules are then specialised further, until specialisations fail to improve the performance. Specialisation candidates are selected by sampling a rule r from each slot after every slot update. Specialisation of a rule is restricted by four conditions:

- It has not been used for specialisation before,
- It has been sampled at least N_E times,
- It is more likely to be sampled than its 'parent' (the rule that created this rule), that is $p_r > p_{parent(r)}$,
- The rule's slot is not full ($|S|$ is less than the maximum number of possible rule specialisations for action $\mathbf{a}$: the number of possible guided specialisations + the number of range splits).

Although specialisation is restricted, every specialisation increases the number of rules, thereby increasing N_E (and N) and slowing the rate of learning. If a rule is rarely

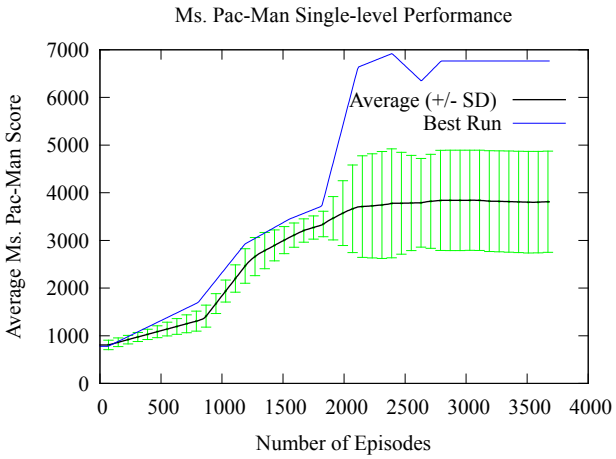

Fig. 3. The 10-fold averaged results for single-level Ms. Pac-Man. Error bars indicate standard deviation between the 10 runs.

positively updated, it can be culled from the distribution, and restricted from being created again. After every update, if a rule r_i within a slot has a selection probability of $p_i \leq (1-\alpha)^{\Theta} \cdot |S|^{-1}$ it is culled, where α is the CE step-wise update value, $|S|$ is the size of slot S, and $\Theta \cdot N$ is the minimum number of updates required for a useless rule to be culled (i.e. the rule is never present in E). $\Theta = 2$ in all experiments.

Because newly created rules are added to a slot with an average probability of being selected, they have a fair chance of being sampled and tested. If a newly specialised rule proves to be useful, its probability of being selected will increase and it may be selected for further specialisation. This method of restricted specialisation confines exploration of the possible rule space to only a useful subset of rules.

IV. Results

CERRLA has been tested on two environments: Ms. Pac-Man and Mario. The graphs shown in Figures 3 and 5 represent the average performance over 10 learning runs initialised with different random seeds.

1) Ms. Pac-Man: The reward structure in Ms. Pac-Man allocates 10 points per dot eaten (there are a total of 242 dots in the first level); 50 points per powerdot (there are a total of 4 powerdots); 750 points for the fruit; 200, 400, 800, 1600 points for each consecutive ghost eaten when ghosts are edible. In a perfect game, the maximum score for the first level is 15,370 points, though this is very difficult to achieve.

Figure 3 shows results for a sub-set of the Ms Pac-Man environment: maximising score in a single level. The episode ends when either the agent loses all lives, or completes the level. The agent takes at most 3700 episodes to converge to a result of approximately 3800 points. The plateau in reward may be a result of a too low β convergence measure.

As a reward comparison, the agent presented in [8] achieves 8186 points on average using CE optimisation of hand-coded rules and 6382 points using CE optimisation of random rules. However, note that CERRLA was designed to learn in *any* relational environment (resulting in a loss in performance), whereas the agent in [8] was designed specifically for playing Ms. Pac-Man. Furthermore, the Ms. Pac-Man environments used are likely to be different in execution.

```
edible(X), distance(X, (1.0 ≤ D ≤ 26.0))
   → moveTo(X, D)
distance(X, (1.0 ≤ D ≤ 26.0))
   → moveTo(X, D)
junctionSafety(X, (0.0 ≤ J ≤ 28.0))
   → toJunction(X, J)
```

Fig. 4. An example policy the agent generates after convergence in the Ms. Pac-Man environment.

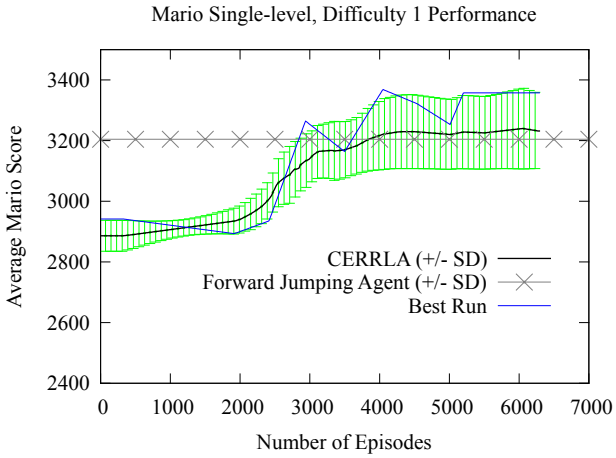

Fig. 5. The CERRLA 10-fold averaged results compared against the average reward for 'Forward Jumping Agent' for Mario. Error bars indicate standard deviation between the 10 runs.

Figure 4 shows an example policy generated by CERRLA for the Ms. Pac-Man environment. The first rule chases and consumes edible ghosts 1–26 units away. The second rule keeps Ms. Pac-Man moving towards things (though this could be non-edible ghosts), and the third rule keeps Ms. Pac-Man moving to the safest junctions. At the beginning of a level, the agent moves towards the safest junction, which generally leads to a powerdot, causing the agent to pursue edible ghosts.

2) Mario: The Mario experiment setup requires Mario to complete a single level of difficulty 1 (no *Spikies* or *BulletBills*, some enemies *Flying*). Because each level is randomly generated, the perfect score is unknown. An episode's total reward is equal to:

$$1024 \times isGoal + 32 \times marioPower + distancePixels + 64 \times fireFlowers + 58 \times mushrooms + 16 \times coins + 24 \times hiddenBlocks + 42 \times kills + 12 \times jumpKills + 4 \times fireballKills + 17 \times shellKills + 8 \times timeLeft$$

Figure 5 shows results for the Mario environment. As a comparison, a simple, but surprisingly effective, 'Forward Jumping Agent', which runs right and jumps whenever an obstacle or enemy is in the way, is also shown. The agent basically learns the 'forward jumping' behaviour, with extra strategy thrown in for collecting items. By training the

```
canJumpOver(X), distance(X, (-160 ≤ D ≤ 160)),
   squashable(X), width(X, (1 ≤ W ≤ 16)),
   ¬flying(X) → jumpOver(X, D, W)
squashable(X), distance(X, (-238 ≤ D ≤ 383)),
   ¬canJumpOver(X), marioPower(fire)
   → shootFireball(X, D, fire)
canJumpOver(X), distance(X, (-160 ≤ D ≤ 160)),
   width(X, (1 ≤ W ≤ 16)), goomba(X)
   → jumpOver(X, D, W)
¬canJumpOn(X), heightDiff(X, (35 ≤ H ≤ 59)),
   brick(X) → search(X, H)
canJumpOver(X), distance(X, (-160 ≤ D ≤ 160)),
   flying(X), width(X, (1 ≤ W ≤ 16)),
   greenKoopa(X) → jumpOver(X, D, W)
goal(X), distance(X, (-160 ≤ D ≤ 160))
   → jumpOnto(X, D)
```

Fig. 6. An example policy the agent generates after convergence in the Mario environment.

agent on low difficulty levels, then gradually increasing the difficulty, the agent could potentially learn better behaviour.

Sometimes, it is better to evade an enemy, rather than attempt to kill it (e.g. run underneath a flying enemy). Currently, there is no explicit action for this behaviour and future experiments will include this action.

Figure 6 shows an example policy generated by CERRLA for the Mario environment. The policy largely deals with jumping over enemies, though there are rules for shooting enemies and searching bricks.

V. FUTURE WORK

Obvious future work includes investigating different learning parameters to balance speed and performance. A slower learning rate (larger N_E) may result in better performance. Another alternative is to investigate bootstrapping rules, by restarting the learning process with previously learned rules as initial fixed rules.

Real time strategy (RTS) games, such as StarCraft[3], are ideal testing environments for relational learners, due to the dynamic, relational nature of the state and multiple levels of learning. An agent operating within an RTS game needs to be able to function both in micro-actions (combat, resource collecting, etc.) and macro-actions (seeking the enemy, upgrades, etc.). Some of this can be automated to simplify the problem, but generally the agent will be required to learn on multiple levels.

A possible solution for this problem is the use of relational *options* [18]. Options define higher level behaviour for achieving a sub-goal within an environment. Options are defined like any other policy, but can be utilised as actions by the agent. By splitting problems into sub-problems, the agent can quickly learn behaviour for the sub-problems and combine the behaviour to solve the problem.

VI. CONCLUSIONS

This paper outlined CERRLA, a policy learner for relational environments using a modified online cross-entropy method and incremental rule refinements. The algorithm formulates its own rules for acting using state observations and guided rule specialisations. Because the algorithm produces relational policies, the agent's behaviour can scale to larger instances of the problem and is able to deal with objects with common properties to those the agent trained on. The cross-entropy method has been shown to be effective for both forming relational policies, but also for guiding the exploration of rule specialisation during learning. The resulting policies created by CERRLA are comprehensible and scalable. The agent may not perform at the same level as a domain specific AI, but it compensates by having the potential to learn across a range of environments without the need for pre-defined rules.

[3]StarCraft was developed by Blizzard Entertainment.

REFERENCES

[1] G. Tesauro, "TD-Gammon, a self-teaching backgammon program, achieves master-level play," *Neural computation*, vol. 6, no. 2, pp. 215–219, 1994.

[2] A. Samuel, "Some studies in machine learning using the game of checkers. II–Recent progress," *Annual Review in Automatic Programming*, vol. 6, pp. 1–36, 1969.

[3] M. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the AAAI competition," *AI Magazine*, vol. 26, no. 2, p. 62, 2005.

[4] M. van Otterlo and T. De Vuyst, "Evolving and Transferring Probabilistic Policies for Relational Reinforcement Learning," in *BNAIC 2009: Benelux Conference on Artificial Intelligence*, October 2009.

[5] K. Kersting and K. Driessens, "Non-parametric policy gradients: A unified treatment of propositional and relational domains," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 456–463.

[6] K. Driessens and S. Džeroski, "Combining model-based and instance-based learning for first order regression," in *Proceedings of the 22nd international conference on Machine learning*. ACM, 2005, pp. 193–200.

[7] T. Croonenborghs, J. Ramon, H. Blockeel, and M. Bruynooghe, "Online learning and exploiting relational models in reinforcement learning," in *Proc. of the Int. Conf. on Artificial Intelligence (IJCAI)*, 2007, pp. 726–731.

[8] I. Szita and A. Lörincz, "Learning to play using low-complexity rule-based policies: Illustrations through Ms. Pac-Man," *Journal of Artificial Intelligence Research*, vol. 30, no. 1, pp. 659–684, 2007.

[9] ——, "Learning Tetris using the noisy cross-entropy method," *Neural Computation*, vol. 18, no. 12, pp. 2936–2941, 2006.

[10] S. Kistemaker, F. Oliehoek, and S. Whiteso, "Cross-entropy method for reinforcement learning," 2008.

[11] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press, March 1998.

[12] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1994.

[13] R. Rubinstein, "Optimization of computer simulation models with rare events," *European Journal of Operational Research*, vol. 99, no. 1, pp. 89–112, 1997.

[14] P. De Boer, D. Kroese, S. Mannor, and R. Rubinstein, "A tutorial on the cross-entropy method," *Annals of Operations Research*, vol. 134, no. 1, pp. 19–67, 2004.

[15] I. Szita and A. Lörincz, "Online variants of the cross-entropy method," *CoRR*, vol. abs/0801.1988, 2008.

[16] G. Plotkin, "A note on inductive generalization," *Machine intelligence*, vol. 5, no. 153-163, p. 178, 1970.

[17] S. Shapiro, D. Eckroth, and G. Vallasi, Eds., *Encyclopedia of Artificial Intelligence*. Wiley, 1987.

[18] T. Croonenborghs, K. Driessens, and M. Bruynooghe, "Learning relational options for inductive transfer in relational reinforcement learning," *Lecture Notes in Computer Science*, vol. 4894, p. 88, 2008.