A Direct Policy-Search Algorithm for Relational Reinforcement Learning

Samuel Sarjant[1](✉), Bernhard Pfahringer[1], Kurt Driessens[2], and Tony Smith[1]

[1] The University of Waikato, Waikato, New Zealand
{sarjant,bernhard,tcs}@waikato.ac.nz
[2] Maastricht University, Maastricht, The Netherlands
kurt.driessens@maastrichtuniversity.nl

Abstract. In the field of relational reinforcement learning — a representational generalisation of reinforcement learning — the first-order representation of environments results in a potentially infinite number of possible states, requiring learning agents to use some form of abstraction to learn effectively. Instead of forming an abstraction over the state-action space, an alternative technique is to create behaviour directly through policy-search. The algorithm named Cerrla presented in this paper uses the cross-entropy method to learn behaviour directly in the form of decision-lists of relation rules for solving problems in a range of different environments, without the need for expert guidance in the learning process. The behaviour produced by the algorithm is easy to comprehend and is biased towards compactness. The results obtained show that Cerrla is competitive in both the standard testing environment and in Ms. Pac-Man and Carcassonne, two large and complex game environments.

1 Introduction

Reinforcement Learning (RL) is a subfield of machine learning in which an *agent* interacts with an *environment* using *actions* and receives numerical *reward* as feedback [1]. An agent selects actions using a *policy*: a decision-making structure that produces an action when given observations for the current *state*. As the field of RL matures, the need for more advanced testing environments increases as algorithms become progressively 'smarter.' In order to represent these complex environments, the field of Relational Reinforcement Learning (RRL) came about, where environments could be represented by variable numbers of objects and relations [2–4]. This representation allows environments with any number of objects and relations to be represented with the same common formalism.

Attempting to learn the value function directly can be impossible in environments consisting of an infinite number of states so a common technique in RRL is to learn an approximate value function for estimating the utility of an action in every state [5–7]. These methods attempt to approximate the value function for every state (and action) in the environment and use the values to *extract* a policy that selects actions with the largest predicted reward. However,

G. Zaverucha et al. (Eds.): ILP 2013, LNAI 8812, pp. 76–92, 2014.
DOI: 10.1007/978-3-662-44923-3_6

in order to learn a reasonably accurate approximate value function, the agent must first discover which states are rewarding. This can be achieved by methods such as random exploration, which is ineffective in complex environments, or using some form of initial guidance such as injecting an expert trace of behaviour [8], which requires some intervention from an external agent (such as a human or pre-existing model).

Instead of approximating values for every state, then extracting a greedy policy from these values, an alternative is to attempt to learn the policy directly. *Policy-search* methods have some advantages over value-function approximation methods: policies are typically smaller than value-functions, as they only need to represent which action to take in a state; and changes in the reward received will not change the policy if the best action for a state remains constant. A disadvantage is that policy-search methods typically require a large number of episodes for training, though this value is usually unaffected by the scale of the environment. Existing policy-search RRL algorithms such as Grey and Gapi have been shown to learn optimal policies in the Blocks World, but testing has been limited to smaller environments [9,10].

Like Grey and Gapi, the algorithm presented in this paper performs direct policy-search where an agent's policy is represented as a decision-list of condition-action rules. The Cross-Entropy Method (CEM), originally developed by [11], is an optimisation algorithm already shown to be effective for learning agent behaviour [12–14], as well as a number of other domains, such as clustering, control and navigation, and continuous optimisation to name a few [15]. We use the CEM's probabilistic optimisation approach to control the policy-creation aspect of the algorithm.

This paper describes Cross-Entropy Relational Reinforcement Learning Algorithm (Cerrla), an application of the CEM for learning behaviour in a range of different relational environments. The CEM is used to identify the best combination of relational condition-action rules acting as the agent's policy. Rules are created in a top-down manner by gradually specialising useful rules in search of better policies. The policies produced by Cerrla should be effective, concise, and easily understood by a human.

To test the general applicability of the algorithm to different environments, we evaluate Cerrla on three separate environments: the standard RRL Blocks World environment, where it achieves excellent results regardless of problem scale; and two game environments, Ms. Pac-Man and Carcassonne, which provide large state spaces and complex action-interactions. Included with the results are example policies produced by Cerrla for each environment.

2 Related Work

The Cerrla algorithm was originally inspired by the algorithm presented in [12]: a Ms. Pac-Man playing agent that uses the Cross-Entropy Method (CEM) to generate and test rule-based policies. The algorithm begins with a set of 42 hand-coded candidate rules that are used to create a rule-based, deterministic policy of maximum size 30. The algorithm learns better behaviour by randomly sampling rules for each of the 30 possible positions in the policy and then adjusting the rule sampling probabilities to produce better performing policies more frequently. This paper also looked at randomly created rules, which did not perform as well as the hand-coded rules, but still performed well. This algorithm formed the core design behind CERRLA, though CERRLA has since expanded upon this design in the following aspects: CERRLA starts without *any* rules or policy size restrictions and creates new rules as it learns; CERRLA learns relational rules/policies for a range of relational environments rather than the single Ms. Pac-Man environment; and CERRLA learns using an iterative CEM, rather than population-based, to quickly integrate newly created rules.

CERRLA uses a similar learning process as the two policy-search RRL algorithms GREY and GAPI: use an evolutionary algorithm to learn a rule-based policy. Both GREY and GAPI use a standard genetic algorithm implementation [16], treating entire policies as chromosomes to be mutated for the recombination operation. However, mutation operations also take place on the rules within each policy by randomly adding/removing literals or replacing variables with constants. Both algorithms were tested in the standard BLOCKS WORLD environment, where they each successfully created goal-achieving policies, but the policies sometimes included useless or detrimental rules. The cross-entropy method employed by CERRLA actively reduces the likelihood of including useless rules, and the rule creation process is bottom-up, resulting in fewer useless literals in rules. GAPI was also tested in a 'gold-finding' environment, which is a step towards more complicated environments. We take this further by testing CERRLA in real-world games that are challenging for humans as well as AI.

The FOXCS system creates rule-based policies by utilising the XCS system for the first-order setting [17,18]. Learning is achieved by maintaining an expected reward and accuracy value for that reward for every rule. These values are used to identify useful rules and guide rule mutation (using standard mutation operations). CERRLA also maintains a value for every rule, but the need to maintain an expected value for every rule limits the scalability of FOXCS. This can be seen in [19], where FOXCS performs worse as the size of the BLOCKS WORLD environment grows. Because CERRLA uses probabilities of *utility* for each rule, the learning rate remains roughly proportional to the number of rules, rather than size of the environment.

Two more RRL systems also deserve a mention, as they perform well on large environments. The LRW-API approach learns a policy by iteratively performing batches of *policy rollouts* as an *approximate policy iteration* algorithm [20]. At any given state, the algorithm updates the Q-value for every action by creating w policy trajectories of length h to identify the most advantageous action to perform (most difference between expected reward and actual reward). The algorithm is able to offset the cost of the rollouts by beginning learning in artificially smaller environments defined as the state reached after n random actions from the initial state. By beginning in small environments and increasing n, the algorithm can quickly scale to large and complex environments. The main

Relational State Observations:

block(a)	*thing(b)*	*clear(fl)*	*above(a, b)*	*height(b, 1)*
block(b)	*thing(c)*	*highest(a)*	*above(a, fl)*	*height(c, 1)*
block(c)	*thing(fl)*	*on(a, b)*	*above(b, fl)*	*height(fl, 0)*
floor(fl)	*clear(a)*	*on(b, fl)*	*above(c, fl)*	
thing(a)	*clear(c)*	*on(c, fl)*	*height(a, 2)*	

Valid Actions:

move(a, c) *move(a, fl)* *move(c, a)*

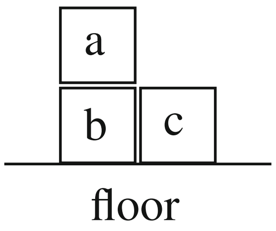


Fig. 1. A 3-block Blocks World state observation example. a is on b which is on the *floor*, and c is also on the *floor*.

disadvantage of this method is the 'controlled experiment' assumption that the world model can be accessed at any state, whereas RRL world models are typically 'black boxes' that only allow a single action per state.

The NPPG algorithm also uses bootstrapping to overcome the problem of large environments in the form of policy-gradient *boosting* [21]. The algorithm iteratively builds regression models to approximate the value function (using batches of episode traces as training data) by layering the model on top of existing models, where each regression model is created to cover the examples previous models do not adequately cover. Each model also receives a weighting to reflect the utility of its predictions. The algorithm performs very well on a 10-block Blocks World environment (thus far, the 10-block environment was typically too large for value-based algorithms to tackle directly), though it does use expert traces to seed the learning with positive examples. A downside to NPPG is the output behaviour is largely incomprehensible, as it is made up of many different weighted models.

3 Terminology

The relational representation used throughout this paper is as follows: a *constant* c is a lowercase symbol representing a uniquely named object of a given *type* (e.g. *thing*, *block*, *enemy*). A *variable* V is an uppercase symbol representing an abstract object. A *term* t may be either a constant or variable. A *predicate* p is a relation acting upon one or more objects with specifically *typed* arguments. Environments are defined by *state predicates* $P_s = \{p_{s,1}, \ldots, p_{s,n}\}$ (which include *type predicates* $P_t = \{p_{t,1}, \ldots, p_{t,n}\}$) and *action predicates* $P_a = \{p_{a,1}, \ldots, p_{a,n}\}$. An *atom* $p(t_1, \ldots, t_n)$ is a predicate with terms for arguments. A *ground atom* $p(c_1, \ldots, c_n)$ only uses constants for arguments. A *goal variable* G_i is a special indexed variable representing one of the constants in the goal and is substituted by the appropriate goal constant when the variable is evaluated. The *anonymous variable* '?' represents any object.

An environment's state observations consist of a complete description of the state $s = \{p_{s,1}(c_{1,1}, \ldots, c_{1,n}), \ldots, p_{s,m}(c_{m,1}, \ldots, c_{m,n})\}$ and the current available actions $A(s) = \{p_{a,1}(c_{1,1}, \ldots, c_{1,n}), \ldots, p_{a,m}(c_{m,1}, \ldots, c_{m,n})\}$. Any constants

$$\begin{aligned}&\mathit{clear(G_0), clear(G_1), block(G_0)} \rightarrow \mathit{move(G_0, G_1)}\\&\mathit{above(X, G_1), clear(X), floor(Y)} \rightarrow \mathit{move(X, Y)}\\&\mathit{above(X, G_0), clear(X), floor(Y)} \rightarrow \mathit{move(X, Y)}\end{aligned}$$

Fig. 2. An optimal Blocks World *onAB* policy generated by Cerrla. Note that G_0 and G_1 are parameterisable goal constants.

directly related to the environment's goal are also provided to the agent. Accompanying each state observation is a reward value. There is no guarantee that an environment will be defined by a Relational Markov Decision Process (RMDP) [3]; the learning agent must simply select an action without absolute knowledge of what state will follow.

3.1 Blocks World

The Blocks World environment is the most commonly used testing environment in the RRL and planning fields due to its simple, but fundamental dynamics. The Blocks World environment will be used for examples in the following sections. The environment consists of a number of blocks stack on top of each other, all stacked on the floor. An agent may move a block on to another block, or on to the floor. A Blocks World state is described by: $P_s = \{clear(Thing), on(Block, Thing), above(Block, Thing), highest(Block), height(Thing, \mathbb{N})\}$ and type predicates $P_t = \{thing, block, floor\}$. The only action predicate is $P_a = \{move(Block, Thing)\}$. Figure 1 shows an example state for a 3-block Blocks World with the listing of all state and action observations for the current state.

Commonly used goals include the *onAB* goal: place block G_0 onto block G_1 (G_0 and G_1 are randomly defined blocks at the start of every episode); and the *stack* goal: stack every block into a tower. Each episode runs for a maximum of $2n$ steps, where n is the number of blocks. The reward received is 1 if the goal is achieved in minimal steps, or some value linearly distributed between 1 and 0 inversely proportional to the number of steps over the minimum the agent took to complete the goal.

4 CERRLA Algorithm

The Cross-Entropy Relational Reinforcement Learning Algorithm (Cerrla) generates policies for a RRL agent by combining a number of randomly sampled condition-action rules into a single decision-list policy (Fig. 2).[1] When the policy is evaluated against the current state observations, it produces one or more actions depending on which rule conditions match the observations. Each rule

[1] Source code, experiment files and videos of Cerrla in action can be found at www.samsarjant.com/cerrla.

is sampled from a separate distribution of similar rules, where the probability of the rule is dynamically adjusted based on the rule's utility. This process is known as the Cross-Entropy Method (CEM) and it forms the backbone of the *probability optimisation* aspect of CERRLA, though some modifications to the core CEM are described in Sect. 4.3.

The other aspect of CERRLA is the *rule discovery* aspect (Sect. 4.2). Rules are created in CERRLA by learning general conditions for every possible action, then gradually applying specific specialisation operators to empirically useful rules to create more complex 'child' rules.

4.1 Cross-Entropy Method

The CEM is a population-based optimisation algorithm, similar to evolutionary algorithms, that uses guided random sampling from one or more distributions to produce effective combinatorial solutions to a given problem. Distributions may be discrete or continuous, but in the context of CERRLA, the distributions are sets of condition-action rules, each with a corresponding probability of being sampled. A sample is a combination of sampled rules represented in a decision-list format (the order of the rules is defined in Sect. 4.3). For a comprehensive exploration of the CEM, see [15].

The CEM is essentially composed of two repeating steps:

1. Generate and test N samples from a distribution of data.
2. Update the distribution such that the top subset of the sampled data is more likely to be generated again in the next iteration.

At every iteration, each distribution produces a randomly selected rule. These rules combine to form a deterministic policy which is then evaluated against the environment which returns the total reward received under the policy. The subset of policies that receive the greatest reward within a population of policies are used to update the rule sampling probabilities, such that the policies' rules are more likely to be sampled again. Intuitively, the algorithm works as follows: in the early stages, the algorithm does not perform any worse than random guessing, but as it gathers samples, it shapes the distribution such that guessing becomes more and more biased towards high-value samples.

Formally, using CERRLA as context, the CEM algorithm is as follows: the algorithm begins with K distributions of rules ($D_k \leftarrow \{r_{1,k}, \ldots, r_{n,k}\}$), where each rule $r_{i,k}$ has a corresponding sampling probability $p_{i,k} \in [0,1] : \sum_{j=1}^{n} p_{j,k} = 1$ (a distribution is typically uniform at the outset). N samples are generated ($\mathbf{X} \leftarrow \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$), where each sample contains a single randomly sampled rule from each distribution (arranged via some heuristic), and tested with evaluation function $f(\mathbf{x})$ (for CERRLA, this is the total reward received). The samples are then sorted into descending order according to $f(\mathbf{x})$ and all samples with $f(\mathbf{x}) \geq \gamma_{t+1}$ are extracted as 'elite samples' E_{t+1}, where γ_{t+1} is equal to the value of the N_E^{th} sorted sample. The minimum number of elite samples is defined as $N_E \leftarrow \rho \cdot N$ (typically $\rho \leftarrow 0.05$). Note that there may be more than N_E elite samples, as multiple samples could have a value equal to the threshold.

The observed distribution D'_k is then calculated for every distribution D_k as the frequency of rules within the elite samples:

$$p'_{j,k} \leftarrow \Big(\sum_{\mathbf{x}_i \in E_{t+1}} \begin{Bmatrix} 1 \text{ if } \mathbf{x}_{i,k} = r_{j,k} \\ 0 \text{ otherwise} \end{Bmatrix} \Big) \Big/ |E_{t+1}| \tag{1}$$

meaning $p'_{j,k}$ is equal to the proportion of elite samples containing rule $r_{j,k}$ from distribution D_k.

The distribution probabilities are then updated using a step-size parameter α (typically α is between 0.4 and 0.9 [15]) to smoothly modify the distribution probabilities:

$$p_{j,k,t+1} \leftarrow \alpha \cdot p'_{j,k} + (1 - \alpha) \cdot p_{j,k,t} \tag{2}$$

This sample-update loop repeats until some convergence measure is reached: (1) a predefined finite number of iterations have passed; (2) probabilities have converged to 0 or 1; or (3) the distributions sufficiently match the observed elites distributions for a given number of iterations.

4.2 Rule Discovery

Cerrla begins the rule discovery process by first calculating the Relative Least General Generalisation (RLGG) for each action in the environment [22]. Further rules are created in a top-down fashion by iteratively specialising empirically useful rules. Each rule is simplified with inferred simplification rules to remove redundant conditions and identify illegal condition combinations, removing illegal and semantically-identical redundant rules from the set of possible rules.

Learning the RLGG. The first rules created by Cerrla are the RLGG rules for every action in the environment. Because *all* state information is available, the RLGG operation only needs to perform *lgg* operations (background knowledge is considered to be part of the state). Each rule encodes the least general set of conditions that have been observed to be true whenever the rule's action is available for the current state. Contrary to the RLGG process in [22], this RLGG process uses a *lossy inverse substitution* to only record information relevant to the rule's action; other information is discarded. That is, the process only considers literals containing constants found in the action, or defined in the environment goal. This lossy inverse substitution focuses learning on the core literals involved in the rule's action, reducing the search space of rules; but has the drawback of losing potentially useful information.

Given a state s and a set of valid actions $A(s) = \{a_1, \ldots, a_n\} : a_i = p_{a,i}(c_{i,1}, c_{i,2}, \ldots)$, the RLGG conditions for each action are defined as:

$$r^a_{RLGG,t} = lgg\big(r^a_{RLGG,t-1}, \theta^{-1} r(s, a_i)\big)$$

where $r^a_{RLGG,t-1}$ is the existing RLGG rule for action predicate p_a and $r(s, a_i)$ is a rule composed of atomic action a_i and every state observation containing one or more of the constants in a_i. The RLGG of the two rules uses a lossy inverse substitution defined by the current arguments of the atomic action a_i, such that $\theta_{a_i}^{-1} = \{c_{i,1}/X, c_{i,2}/Y, \ldots\}$. Any non-numerical constants not included in $\theta_{a_i}^{-1}$ are replaced by the anonymous variable '?' which can be substituted for any constant when evaluated. Numerical constants are replaced by free variables representing the number. The resulting rule encodes a near-least general set of conditions (due to lossy inverse substitution) required for taking action p_a.

Example 1. Referring to Fig. 1, the RLGG calculation process for the three valid actions *move(a, c)*, *move(a, fl)*, *move(c, a)* is described in the following example, processing one rule at a time (beginning with $t = 1$):

$r_{move(a,\ c)}$ = *block(a)*, *block(c)*, *thing(a)*, *thing(c)*, *clear(a)*, *clear(c)*, *on(a, b)*, *on(c, fl)*, *above(a, b)*, *above(a, fl)*, *above(c, fl)* → *move(a, c)*
$\theta_{move(a,\ c)}^{-1} = \{a/X, c/Y\}$
$r_{RLGG,1}^{move}$ = *block(X)*, *block(Y)*, *thing(X)*, *thing(Y)*, *clear(X)*, *clear(Y)*, *on(X, ?)*, *on(Y, ?)*, *above(X, ?)*, *above(Y, ?)* → *move(X, Y)*

This is already very close to the actual RLGG; only the conditions *block(Y)*, *on(Y, ?)*, and *above(Y, ?)* are not always true, as evidenced in the following example:

$r_{move(a,\ fl)}$ = *block(a)*, *floor(fl)*, *thing(a)*, *thing(fl)*, *clear(a)*, *clear(fl)*, *on(a, b)*, *on(b, fl)*, *on(c, fl)*, *above(a, b)*, *above(a, fl)*, *above(b, fl)*, *above(c, fl)* → *move(a, fl)*
$\theta_{move(a,\ fl)}^{-1} = \{a/X, fl/Y\}$
$r_{RLGG,2}^{move}$ = *block(X)*, *thing(X)*, *thing(Y)*, *clear(X)*, *clear(Y)*, *on(X, ?)*, *above(X, ?)* → *move(X, Y)*

This rule is in fact the RLGG for the Blocks World *move* action, so there is no need to describe the process for the final action of the state (as the rule cannot generalise any further). Many of the conditions in this rule are redundant with respect to other facts though (e.g. *on(X, ?)* is always true if *above(X, ?)* is true) and can be removed using the simplification rules described in the rule simplification section. The simplified rule is:

$$r_{\mathrm{RLGG},\ 2}^{move} = \mathit{clear(X)}, \mathit{clear(Y)}, \mathit{block(X)} \rightarrow \mathit{move(X,\ Y)} \tag{3}$$

Rule Specialisation. Cerrla uses three specialisation operators: (1) *additive*, (2) *goal-replacement*, and (3) *range-splitting*. Each specialisation operator creates a new rule with more specialised conditions.

(1) *Additive specialisation* specialises a rule by adding a condition to it. Instead of adding an arbitrary condition to a rule r^a with any possible argument bindings, Cerrla restricts the set of specialisation conditions to those that include action-related conditions and have been *observed* to be true (but not in the RLGG conditions) when action a is available. Each specialisation

condition is recorded with inversely-substituted arguments (replace all action-related constants with variables *and* replace all goal-related constants with goal variables). Negated specialisation conditions are also used to specialise a rule.

(2) For every constant in the environment goal, *goal-replacement* replaces all occurrences of one of the action's arguments with an indexed 'goal variable' G_i representing one of the constants in the environment's current goal. After replacing the variable, if all conditions containing the goal variable in the rule have previously been observed to be possible (i.e. ensure the rule's conditions can feasibly be met), the specialised rule is valid, otherwise it is invalid and discarded.

(3) *Range splitting* creates specialised rules by splitting an existing range (or a variable representing a number) into up to five overlapping sub-ranges: the *lower half*, the *upper half*, a *central half*, and if applicable, a negative sub-range (*lower bound* to 0), and a positive sub-range (0 to *upper bound*). Except when 0 is a bound, the range bounds are expressed as variable fractions of the observed minimum and maximum bounds so the rule does not need to change when the bounds change. For instance, a range from $[-4, 8]$ would be split into the following subranges: $[-4, 2]$, $[2, 8]$, $[-1, 5]$, $[-4, 0]$, and $[0, 8]$ (all represented as variable fractions of the original range).

Example 2. The RLGG rule from the prior example (Eq. 3) can be specialised into the following example rules:

$$r_1^{move} = clear(X), clear(Y), block(X), floor(Y) \rightarrow move(X,\ Y)$$
$$r_2^{move} = clear(X), clear(Y), block(X), not(highest(X)) \rightarrow move(X,\ Y)$$
$$r_3^{move} = clear(G_0), clear(Y), block(G_0) \rightarrow move(G_0, Y)$$
$$r_4^{move} = clear(X), clear(Y), block(X), on(X, G_0) \rightarrow move(X,\ Y)$$

As there are no variables representing numerical arguments, the range splitting specialisation does not produce any rules. Note that some rules contain redundant conditions and can be simplified (see the following section).

Rule Simplification. To avoid creating illegal rules or rules containing redundant conditions, CERRLA also infers *simplification rules* for the environment.

Simplification rules are created using the RLGG method in Sect. 4.2 but instead of calculating the RLGG relative to each *action predicate* p_a, it is calculated relative to each *state predicate* p_s. The resulting set of RLGG conditions encode the relationship between each state predicate p_s and all other predicates, producing implication rules in the form $A \Rightarrow B$, such that condition-action rules containing both A and B remove condition B. Furthermore, rules containing A and $\neg B$ are marked as illegal rules and are deleted from CERRLA's distributions. If $B \Rightarrow A$ as well, the rule is instead recorded as an equivalence rule $A \Leftrightarrow B$, such that condition-action rules containing B replace it with A.

The RLGG can also be calculated for the set of atoms that are *never true* when p_s is present as well. The variable representation of the lossy inverse substitution $\theta^{-1}_{p_s(t_1,\ldots,t_n)}$ results in a finite set of possible atoms given the set of state predicates (as all constants are replaced by variable $X, Y, \ldots$ or '?'). By removing the inversely substituted true atoms from this set, the RLGG of *never true* atoms relative to p_s can also be calculated to produce simplification rules involving negated conditions.

Example 3. Some of the simplification rules created for Blocks World include:

$$\begin{array}{ll} on(X,\ Y) \Rightarrow above(X,\ Y), & floor(Y) \Leftrightarrow clear(Y), on(X,\ Y), \\ highest(X) \Rightarrow not(on(?,\ X)), & block(X) \Leftrightarrow on(X,\ ?) \\ block(X) \Rightarrow not(floor(X)) & \end{array}$$

Whenever the right-hand side of a simplification rule subsumes a rule's literal(s), they are removed (or replaced by the substituted left-hand side literals for equivalence rules). Note that '?' explicitly represents the anonymous variable when performing rule simplification (i.e. all '?' are treated as constants).

4.3 Policy-Search Process

The Cerrla algorithm (Algorithm 1) uses a modified version of the CEM to produce policies. Each distribution of candidate rules is sampled to produce a single rule, which is included into the policy in a specific order. The policy is then deterministically evaluated throughout the episode to produce the agent's actions. The policies that received the largest reward form an 'elite distribution' which the rule sampling probabilities are adjusted toward, such that rules in an elite policy are more likely to be sampled again.

Algorithm 1. Pseudocode summary of Cerrla.

```
Initialise the distribution set $\mathbb{D}$        ▷ Initially empty. Learns RLGG rules to start
repeat
    Generate a policy $\pi$ from $\mathbb{D}$    ▷ Sample $\leq 1$ rule from each $D$ in $\mathbb{D}$ and order into policy
    Evaluate $\pi$, receiving average reward $R$      ▷ Run three times and average
    Update elite samples $E$ with sample $\pi$ and value $R$      ▷ If $\pi$ is good, add to $E$
    Update $\mathbb{D}$ using $E$   ▷ Adjust probabilities for each $D$ in $\mathbb{D}$ to be closer to distribution in $E$
    Specialise rules (if $\mathbb{D}$ is ready)      ▷ If a rule is highly probable, branch it to a new $D$
until $\mathbb{D}$ has converged      ▷ Until no more branching is possible
```

Instead of a population-based approach, Cerrla uses an *online variation* of the CEM, similar to the CEM variant in [23], which updates the distributions after every sample. Instead of sampling batches of N samples, the algorithm maintains a sliding window of N samples, such that the elites E consist of the best samples from the last N samples (instead of the best samples in a batch). The online variation is able to adapt to a changing number of rules and distributions as Cerrla creates new specialisations.

Initialisation. Cerrla begins with no rules or distributions ($\mathbb{D} \leftarrow \{\}$) but quickly creates distributions by firstly observing the RLGG for every available action, then creating all immediate specialisations of the RLGG (as described in Sect. 4.2). Each of these rules acts as a *seed* for a new distribution, such that a distribution consists of a uniform distribution of the seed rule and all immediate specialisations of the seed rule. Each D also has two properties: the probability that a rule from D is present within a policy, $p(D) \in [0, 1]$ (initially $p(D) \leftarrow 0.5$); and the average relative positions of sampled rules within generated policies, $q(D) \in [0, 1]$, where 0 represents the first position and 1 represents the last (initially $q(D) \leftarrow 0.5$).

The number of rules within D is written as $|D|$ and $KL(D)$ represents the Kullback-Leibler (KL) size, or distance from the uniform distribution, of D such that:

$$KL(D) \leftarrow \max \left[|D| \cdot \left(1 - \sum_{r \in D} p_r \log_{|D|}(|D| \cdot p_r) \right), 1 \right] \tag{4}$$

A uniform distribution has $KL(D) = |D|$, but a distribution with a single high probability rule (e.g. $p_j \geq 0.95$) has $KL(D) = 1$.

Policy Generation. A policy π is generated by sampling a rule from every distribution D in $\mathbb{D}$. For each D, a rule will only be sampled from D with probability $p(D)$. The position of the sampled rule is determined by the relative ranking to all other rules in the policy. This relative value $relQ(D)$ is sampled from a normal distribution with parameters $q(D)$ for the mean, and $1-|1-2p(D)|$ as the standard deviation. Rules are ordered in the policy in ascending order according to their respective $relQ(D)$. When D is initialised with $p(D) = 0.5$, the relative position of each sampled rules varies wildly, but as the $p(D)$ converges towards 0 or 1, the relative position fluctuates less.

The policy π is evaluated at every decision step using the current state observations. Starting with the first rule in the policy, each rule is evaluated as a query on the state. If a substitution(s) for the rule's conditions is found, the rule's action is returned with the substituted values applied (there may be more than one substitution). If multiple actions are returned by the agent, an environment-specific selection mechanism (e.g. sort by distance) selects and resolves one of the actions and advances to the next state. The value of a policy ($f(\pi)$) is equal to the average total-episodic-reward received for a given number of episodes (we use three episodes in experiments).

Updating. After a policy π has been evaluated, it is added to the floating set of elite samples E if $f(\pi) \geq \gamma$ (the worst elite sample). Before this occurs, any elite samples that have existed for greater than N iterations are removed to ensure the elites represent recent samples. Rather than using a fixed elite sample size, CERRLA dynamically changes the number of elites N_E based on the state of the current distributions:

$$N_E \leftarrow \max \left\lceil \underbrace{\arg\max_{D \in \mathbb{D}} \big(KL(D) \cdot p(D)\big)}_{\text{largest distribution}}, \quad \underbrace{\sum_{D \in \mathbb{D}} p(D)}_{\text{sum distribution means}} \right\rceil \tag{5}$$

where $N \leftarrow N_E/\rho$, as with the regular CEM. This equation results in a large N_E when CERRLA contains new, undertested distributions, and a smaller N_E when rule probabilities and distribution means are close to 0 or 1.

After potentially adding the sample to the elite samples, the distributions are updated. The increased frequency of updates requires a reduced update step-size to 'smooth' the learning rate. Instead of α, the single-step update parameter $\alpha_1 \leftarrow \alpha/N$ is used, which is approximately equal to the standard α update rate.

A restriction applies to updates: a distribution D is only updated if it has produced $C \cdot |D|$ samples. This reduces update bias towards early samples, and provides enough samples for a distribution to be fairly represented in the elite samples. $C = 3$ is used in experiments as it provides 95 % coverage of all rules in a uniform distribution [24].

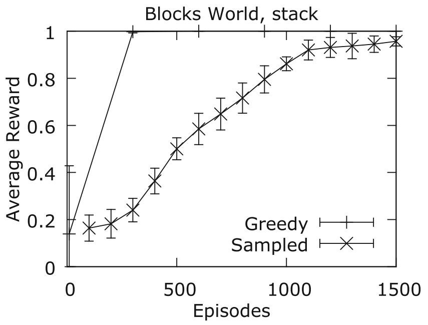
The update process consists of updating every candidate rule distribution in $\mathbb{D}$ (as per Sect. 4.1), as well as their properties $p(D)$ and $q(D)$. Only the rules that fired throughout the sample's testing episodes matter, therefore unused rules (and their distributions) are not included in the update and, implicitly, negatively updated. Each update operation uses Eq. 2 to adjust the values in a step-wise manner. The observed value for $p(D)$ is simply the proportion of policies containing D within the elites. The observed value for $q(D)$ is the average relative position of rules from D within the elite policies, where 0 represents the first position and 1 represents the last.

Rule Exploration. When a rule has a sufficiently high probability, it 'branches,' creating new rules with more specialised conditions in hopes of finding better rules. This process is triggered when a distribution's $KL(D) \leq \delta \cdot |D|$, where $\delta = \min \left[(d+1)^{-1}, p(D)\right]$, representing the splitting point with respect to the *depth* d of the distribution or number of branches from the RLGG distribution. The highest probability rule r from D is removed from D, and is used to seed a new distribution, populating the distribution with r and all immediate specialisations of r. The only restriction to this exploration process is if r originally seeded D, in which case no branching occurs.

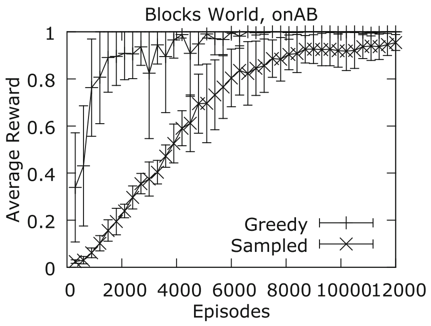
Convergence. CERRLA is considered converged when each distribution is considered converged. A distribution is converged when the sum divergence of the rule probabilities between updates is less than $\alpha_1 \cdot \beta$ (a convergence threshold). Alternatively, experiments can specify a fixed number of training episodes. Upon convergence, the current best elite sample is also output as the best solution.

(a) Performances comparison in Blocks World with 3–10 blocks for various RRL algorithms. Note that some figures are approximate readings from a graph.

Algorithm	Average Reward		# of Training Episodes ($\times 10^3$)	
	stack	*onAB*	*stack*	*onAB*
Cerrla	**1.00**	**0.99**	**1.60**	**10.30**
P-RRL [2]	1.00	0.65	0.05	0.05
RRL-TG [5]	0.88	0.92	0.50	12.50
RRL-TG (P) [5]	1.00	0.92	30.00	30.00
RRL-RIB [5]	0.98	0.90	0.50	2.50
RRL-KBR [5]	1.00	0.98	0.50	2.50
Trendi [6]	1.00	0.99	0.50	2.50
TreeNPPG [21]	—	0.99	—	2.00
Marlie [7]	1.00	0.98	2.00	2.00
Foxcs [18]	1.00	0.98	20.00	50.00



(b) *Stack* goal in 100-block Blocks World.



(c) *OnAB* goal in 100-block Blocks World.

Fig. 3. Cerrla's performance in the Blocks World environment.

5 Evaluation

Cerrla is evaluated in three separate environments: Blocks World, Ms. Pac-Man, and Carcassonne.[2] Each environment presents a different problem for the agent to solve, though all share a common RRL representational format. All reported results are averaged across 10 separate experiments. Each figure contains two performances: *sampled performance*, the reward received during training; and *greedy performance*, the average reward received for 100 testing policies (not included in training time/number of episodes) using the current best elite sample.

5.1 Blocks World

Figure 3a compares Cerrla's Blocks World performance against other RRL algorithms for both *stack* and *onAB* goals. A summary of current RRL algorithms can be found in [4]. Like most RRL algorithms it is able to learn optimal or near-optimal behaviour in both goals, though it requires more episodes

[2] An additional environment detailed in [25] was omitted for space reasons.

than the value-based algorithms to do so (training time for each goal was 6 and 34 s respectively). Figure 3b and c demonstrates a powerful property of CERRLA: even in a BLOCKS WORLD of 100 blocks, the learning rate remains roughly constant (training time is only $\sim$50 times longer compared to exponential state increase). In most other compared approaches, learning rate deteriorates as the number of blocks increase.

The effects of the simplification rules are tested by comparing two 12,000 fixed-episode experiments for the *onAB* goal: one using rule simplification, the other not using rule simplification (denoted in brackets). CERRLA initially creates 244 (1,150) rules spread over 17 (32) distributions. At 12,000 training episodes, the algorithm has 330 (1,405) rules spread over 27 (40) distributions, with an average greedy performance of 0.99 (0.86), and an average training time of 43 (165) seconds. It is clear that rule simplification is highly effective in all aspects of CERRLA's learning.

5.2 Ms. Pac-Man

MS. PAC-MAN is a famous arcade video-game in which the player (CERRLA) controls a character that eats dots for points inside a finite maze while avoiding four hostile ghosts. When one of four 'power dots' is eaten, the ghosts become non-hostile for a short time and can be eaten for an increasingly larger number of points. CERRLA's control of the character is defined by high-level actions that resolve into low-level directional movement. If multiple actions are predicted by the same rule, the action with the closest object to the agent is acted upon. If multiple objects are equidistant in different directions, the next rule in the policy breaks the tie (otherwise choose randomly).

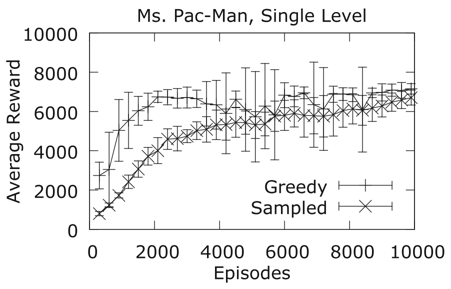
A MS. PAC-MAN state is described by similar predicates seen in [12]: $P_s = \{distance(Thing, \mathbb{N}), junctionSafety(Junction, \mathbb{N}), blinking(Ghost), edible(Ghost)\}$ and type predicates $P_t = \{thing, dot, ghost, powerDot, ghostCentre, junction\}$. The action predicates are $P_a = \{moveTo(Thing, \mathbb{N}), moveFrom(Thing, \mathbb{N}), toJunction(Junction, \mathbb{N})\}$ (the numeric value is meta-data for resolving the action).

Within a single level, CERRLA achieves an average greedy performance of 7196 points per episode. Compared to the conceptually equivalent CE-RANDOMRB agent from [12] (6382 points), CERRLA learns a slightly better policy. CERRLA performs slightly worse than the reported hand-coded and human average scores of 8186 and 8064 points respectively of [12]. An agent can achieve a theoretical maximum of 15,600 points in a single level, so CERRLA's performance could be improved. Figure 4a shows an example policy produced by CERRLA that focuses primarily on eating *edible ghosts*, *powerDots*, and *dots* in that order.

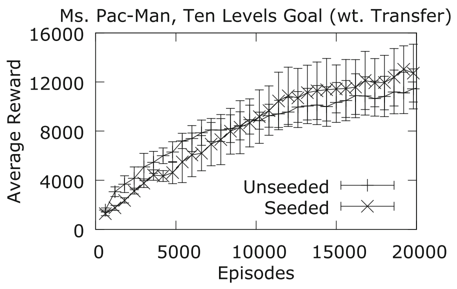
CERRLA's rule-based representation can also facilitate *transfer learning* (transfer learned behaviour for a source goal into behaviour for a target goal). During initialisation for the target goal, each rule in the greedy policy for the source goal seeds a new distribution, providing a headstart in the specialisation process. When the behaviour learned in the *Single-Level* goal is used to initialise the algorithm for a *Ten-Levels* goal, an improvement can be seen in the resulting behaviour (Fig. 4c).

edible(X), distance(X, Y) $\rightarrow$ *moveTo(X, Y)*
powerDot(X), distance(X, Y) $\rightarrow$ *moveTo(X, Y)*
thing(X), distance(X, Y), not(ghost(X)), not(ghostCentre(X)) $\rightarrow$ *moveTo(X, Y)*
dot(X), distance(X, (26 $\leq$ *Y* $\leq$ *52))* $\rightarrow$ *moveFrom(X, Y)*

(a) Example *Single Level* Ms. Pac-Man policy generated by Cerrla. Achieves an average reward of 7534.



(b) Single level of Ms. Pac-Man, limited to 10,000 episodes.



(c) Ten levels of Ms. Pac-Man, with *Single Level* seeded policy and unseeded learning, limited to 20,000 episodes.

Fig. 4. Cerrla's performance in the Ms. Pac-Man environment.

5.3 Carcassonne

Carcassonne is a turn-based, medieval-themed board game in which players attempt to control terrain via tokens called 'meeples' to score points. Each player has two actions per turn: place a randomly drawn tile adjacent to existing tiles such that all edges match up, then optionally place a meeple on any terrain on the placed tile. An episode ends when all tiles have been placed, at which point any unfinished terrain is scored.

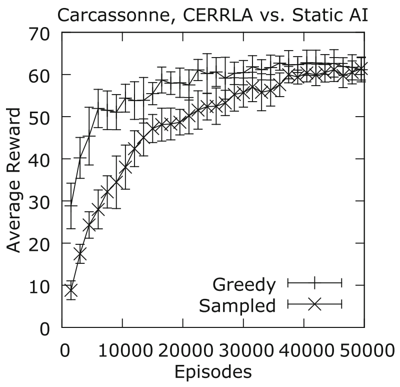
A Carcassonne state is described using a combination of the 22 state predicates and 10 type predicates (a full specification can be found in [25]), with the valid actions described as one of two actions: $P_a = \{placeTile(Player, Tile, Location, Orientation), placeMeeple(Player, Tile, Terrain)\}$.

In a Carcassonne game against a static AI using a min-max strategy for making decisions (the default AI for JCloisterZone[3]), Cerrla achieves an average score of 63 per game. In comparison, the min-max AI scores 92 and a related Monte-Carlo Tree Search approach achieves approximately 85 [26]. Carcassonne's complex dynamics prove to be challenging for Cerrla, but it is able to learn an 'easy opponent' strategy. Figure 5a shows a policy produced by Cerrla. The policy places tiles in close groups or near *cloisters*, or anywhere by default. Meeples are typically placed on high *worth* (>3) terrain, or any *open* terrain if Cerrla has ≥4 meeples left.

[3] http://jcloisterzone.com/en/

currentPlayer(X), controls(X, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, 4.5 $\leq$ V $\leq$ 8.0) $\rightarrow$ placeTile(X, Y, Z, W)
currentPlayer(X), meepleLoc(Y, Z), worth(Z, 3.0 $\leq$ V $\leq$ 6.0), not(nextTo(?, ?, Z)) $\rightarrow$ placeMeeple(X, Y, Z)
currentPlayer(X), controls(X, ?), meepleLoc(Y, Z), worth(Z, 3.0 $\leq$ V $\leq$ 6.0) $\rightarrow$ placeMeeple(X, Y, Z)
currentPlayer(X), meeplesLeft(X, 4.0 $\leq$ U $\leq$ 7.0), meepleLoc(Y, Z), worth(Z, 1.5 $\leq$ V $\leq$ 4.5), not(completed(Z)) $\rightarrow$ placeMeeple(X, Y, Z)
currentPlayer(X), controls(X, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, 3.625 $\leq$ V $\leq$ 5.375) $\rightarrow$ placeTile(X, Y, Z, W)
currentPlayer(X), meeplesLeft(X, 4.0 $\leq$ U $\leq$ 7.0), meepleLoc(Y, Z), tileEdge(Y, ?, Z), open(Z, V) $\rightarrow$ placeMeeple(X, Y, Z)
currentPlayer(X), validLoc(Y, Z, W), numSurroundingTiles(Z, 2.75 $\leq$ V $\leq$ 6.25), cloisterZone(Z, ?) $\rightarrow$ placeTile(X, Y, Z, W)
currentPlayer(X), controls(X, ?), validLoc(Y, Z, W), numSurroundingTiles(Z, 2.75 $\leq$ V $\leq$ 6.25) $\rightarrow$ placeTile(X, Y, Z, W)
currentPlayer(X), validLoc(Y, Z, W) $\rightarrow$ placeTile(X, Y, Z, W)

(a) Example CARCASSONNE policy generated by CERRLA. Achieves an average reward of 65.



(b) CERRLA vs. Min-max AI in CARCASSONNE, limited to 50,000 episodes.

Fig. 5. CERRLA's performance in the CARCASSONNE environment.

6 Conclusions

The application of the CEM to RRL — CERRLA — has been shown to be capable of creating and combining sets of relational condition-action rules into effective policies in a range of different environments. Although the number of training episodes exceed value-based methods, the learning rate remains constant with increased scale of the problem and the simplified rules minimise rule evaluation time. It should be noted that the representation of MS. PAC-MAN and CARCASSONNE may not be ideal, and may even limit CERRLA's behaviour, but it is clear that CERRLA can create effective behaviour with it. In general, CERRLA exhibits good scalability and, given only a problem's high-level specification and state observations, produces human-readable policies that are competitive with more specialised single-domain approaches.

References

1. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning). The MIT Press, Cambridge (1998)
2. Džeroski, S., De Raedt, L., Driessens, K.: Relational reinforcement learning. Mach. Learn. **43**, 7–52 (2001)
3. van Otterlo, M.: The Logic of Adaptive Behaviour: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains. IOS Press, Amsterdam (2009)
4. Wiering, M., van Otterlo, M. (eds.): Reinforcement Learning: State-Of-The-Art, vol. 12. Springer-Verlag New York Incorporated, New York (2012)
5. Driessens, K.: Relational reinforcement learning. Ph.D. thesis, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (2004)

6. Driessens, K., Džeroski, S.: Combining model-based and instance-based learning for first order regression. In: Proceedings of the 22nd International Conference on Machine Learning, pp. 193–200. ACM (2005)
7. Croonenborghs, T., Ramon, J., Blockeel, H., Bruynooghe, M.: Online learning and exploiting relational models in reinforcement learning. In: Proceeding of the International Conference on Artificial Intelligence (IJCAI), pp. 726–731 (2007)
8. Driessens, K., Džeroski, S.: Integrating guidance into relational reinforcement learning. Mach. Learn. **57**(3), 271–304 (2004)
9. Muller, T., van Otterlo, M.: Evolutionary reinforcement learning in relational domains. In: Proceedings of the 7th European Workshop on Reinforcement Learning, Citeseer (2005)
10. van Otterlo, M., De Vuyst, T.: Evolving and transferring probabilistic policies for relational reinforcement learning. In: BNAIC 2009: Benelux Conference on Artificial Intelligence, October 2009
11. Rubinstein, R.Y.: Optimization of computer simulation models with rare events. Eur. J. Oper. Res. **99**(1), 89–112 (1997)
12. Szita, I., Lörincz, A.: Learning to play using low-complexity rule-based policies: illustrations through Ms. Pac-Man. J. Artif. Int. Res. **30**(1), 659–684 (2007)
13. Kistemaker, S., Oliehoek, F., Whiteso, S.: Cross-entropy method for reinforcement learning. Bachelor thesis, University of Amsterdam, Amsterdam, The Netherlands, June 2008
14. Tak, M.: The cross-entropy method applied to SameGame. Bachelor thesis, Maastricht University, Maastricht, The Netherlands (2010)
15. De Boer, P., Kroese, D., Mannor, S., Rubinstein, R.: A tutorial on the cross-entropy method. Ann. Oper. Res. **134**(1), 19–67 (2004)
16. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
17. Wilson, S.W.: Classifier fitness based on accuracy. Evol. Comput. **3**(2), 149–175 (1995)
18. Mellor, D., Mellor, D.: A learning classifier system approach to relational reinforcement learning. In: Takadama, K., et al. (eds.) IWLCS 2006 and IWLCS 2007. LNCS (LNAI), vol. 4998, pp. 169–188. Springer, Heidelberg (2008)
19. Mellor, D.: A learning classifier system approach to relational reinforcement learning. Ph.D. thesis, School of Electrical Engineering and Computer Science, The University of Newcastle, Australia (2008)
20. Fern, A., Yoon, S., Givan, R.: Approximate policy iteration with a policy language bias: solving relational markov decision processes. J. Artif. Int. Res. **25**(1), 75–118 (2006)
21. Kersting, K., Driessens, K.: Non-parametric policy gradients: a unified treatment of propositional and relational domains. In: Proceedings of the 25th International Conference on Machine Learning, ICML '08, pp. 456–463. ACM, New York (2008)
22. Plotkin, G.D.: A note on inductive generalization. Mach. Intell. **5**, 153–163 (1970)
23. Szita, I., Lörincz, A.: Online variants of the cross-entropy method. Technical report, arXiv:0801.1988 (2008)
24. Aslam, J.A., Popa, R.A., Rivest, R.L.: On estimating the size and confidence of a statistical audit. In: Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology, EVT'07, pp. 8–8. USENIX Association, Berkeley (2007)
25. Sarjant, S.: Policy search based relational reinforcement learning using the cross-entropy method. Ph.D. thesis, The University of Waikato (2013)
26. Heyden, C.: Implementing a computer player for Carcassonne. Master's thesis, Maastricht University (2009)