Autonomous Learning of Domain Models using Two-Dimensional Probability Distributions

Witold Słowiński
Computing Science, University of Aberdeen, Scotland
ro3ws8@abdn.ac.uk

Frank Guerin
Computing Science, University of Aberdeen, Scotland
f.guerin@abdn.ac.uk

Abstract—An autonomous agent placed without any prior knowledge in an environment without goals or a reward function will need to develop a model of that environment using an unguided approach by discovering patterns occurring in its observations. We expand on a prior algorithm which allows an agent to achieve that by learning clusters in probability distributions of one-dimensional sensory variables and propose a novel quadtree-based algorithm for two dimensions. We then evaluate it in a dynamic continuous domain involving a ball being thrown onto uneven terrain, simulated using a physics engine. Finally, we put forward criteria which can be used to evaluate a domain model without requiring goals and apply them to our work. We show that adding two-dimensional rules to the algorithm improves the model and that such models can be transferred to similar but previously-unseen environments.

I. INTRODUCTION

A developing agent without any prior knowledge in an environment without guidance in the form of goals or a reward function will need to develop a model of that environment using an unguided approach by discovering patterns occurring in its observations.

In this paper we extend an approach to learning models from clusters in probability distributions [1] by adding the possibility to learn clusters of two-dimensional variables for use in rule predicates. We accomplish this using a quadtree-based probability distribution data structure and clustering algorithm, described in Section II. With the extension of the one-dimensional technique to two dimensions we aim to make the model more expressive so that a two-dimensional region may be learnt in a single step and described with a single rule rather than requiring the learning of several individual one-dimensional rules.

Because of the lack of guidance in this kind of model learning, we propose certain criteria that can be useful when evaluating a model on a set of data, without requiring the agent to accomplish tasks or use planning. We present these in Section IV-A.

The basic premise behind learning predicates from rules is that in non-trivial domains involving continuous variables, continuous time and a dynamic world, the probability distributions of these variables will not be uniform. In physics problems a system will tend to go to equilibrium states which will skew the probability density towards them. This fact is exploited by our algorithm and we show that indeed only using regions derived from such distributions one can arrive at a potentially transferable model with predictive power. We report on our results in Section IV and IV-D in particular.

II. RELATED WORK

Autonomous model learning has been approached from various directions in related work largely depending on the world and model representations. In our work we focus on continuous domains with states described using a set of variables. In the past we have approached the problem of learning probability-distribution learning of one-dimensional rules [1].

The closest work to ours is that of Mugan and Kuipers [2], [3], who describe a comprehensive framework for learning one-dimensional rules over a qualitative space using landmarks, constructing an action hierarchy and using the actions for planning. Their approach applies information theory criteria to find the right landmark values which is an alternative or complementary approach to rule learning described in this paper.

Provost et al. [4] describe a technique for finding clusters in the state space using Self-Organising Maps [5] applied to solving robot navigation problems. Instead of developing an explicit model, they learn higher-level state features which are used to build high-level actions for use in reinforcement learning.

Jong and Stone [6] describe a method for learning a model of a continuous domain using averagers. Rather than using a direct representation of regions, they are encoded using a probability mass function which is utilised by the averager. Additionally, states are treated as atomic multidimensional vectors and there are no explicit rules referencing the individual variables in a state.

The technique proposed by Goodman et al. [7] also builds a probabilistic model of a continuous world using a discretisation of the state space into square regions. Their world representation similarly contains variables, however the number of variables may vary and the learning algorithm has to select which ones to include in the model.

In addition to the body of work relating to continuous domains, there are a number of techniques aimed at learning relational models of discrete domains, which are aimed at learning probabilistic rules or schemas [8], [9], [10].

Notably, Pasula et al. [11] describe a technique for learning a probabilistic model, consisting of a set of rules, of a discrete domain with states described using relational predicates. Their algorithm involves a set of operators that manipulate rules in order to improve the model. The rules are evaluated similarly as in our work.
III. MODEL LEARNING

The objective of an unguided model learning algorithm is to enable the learning of a world model based on past observations. It takes as input a sequence of observations of the world state, and produces a set of rules which can describe patterns generally holding true in the observations.

The algorithm we propose takes a statistical approach. First it estimates the continuous probability distributions, or probability density functions of variables and computes clusters in those distributions in order to obtain regions in variable value spaces. Then rules are created which try to predict when variables will enter those regions. These rules start out with small or empty preconditions which are then iteratively expanded with regions on other variables in order to increase the probability of success.

A. State and Model Representation

We base our state and model representation on our past work. Observations are represented using a sequence of states, where in each state there are variables taking on values which may vary from state to state. The world model is a set of rules which have a Bayesian representation similar to that proposed by [2], [3].

In addition to real scalar variables which can be represented in the works mentioned above, the representation proposed here also allows for two-dimensional real vectors and we also make a stronger distinction between instantaneous and temporal rules.

1) State Representation: The agent’s observations consist of an enumerable set of states, identified by natural numbers: \( S \subseteq \mathbb{N} \). In each state there are variables whose values can be observed. We denote the set of all variable names by \( V \). Variable values in particular states are inspected using the mapping value: \( V \times S \rightarrow \mathbb{R}^d \), where \( d \in \{1, 2\} \) is the dimensionality of the variable whose value is being inspected. We denote the dimensionality of variable \( v \) by \( \dim(v) \). As a shorthand, we denote the value of the variable \( v \) in state \( s \) by \( v_s = value(v, s) \).

There is a mandatory \( time \) variable which must be present in all domains; it represents the time, in seconds, since observations began. The time must be increasing with states, so for \( s, s’ \in S \), if \( s < s’ \) then \( time_s < time_{s’} \).

2) Model Representation: We define a world model to be a set of rules which can be used in aggregate to provide predictive power and an understanding of the domain.

Rules are built up of predicates on world states. A predicate \( p \in \mathcal{P} \) is a function \( p: S \rightarrow \{true, false\} \). For convenience, we will use \( p(s) \) as a shorthand for \( p(s) = true \).

A basic predicate examines the value of a variable and checks if it is contained in a certain subset, or region, of the variable’s value space. Given a variable \( v \) and a region \( c \subseteq \mathbb{R}^{\dim(c)} \), the basic predicate is of the form:

\[
in_{v,c}(s) = \begin{cases} 
true & \text{if } v_s \in c \\
false & \text{otherwise}
\end{cases}
\]

The set of all rules is denoted \( \mathcal{R} \). Each rule \( r \in \mathcal{R} \) is a tuple \( r = (\text{pre}(r), \text{post}(r), \text{deadline}(r)) \), where \( \text{pre}(r) \) is the precondition, \( \text{post}(r) \) is the postcondition and \( \text{deadline}(r) \in \mathbb{R} \) is the rule’s deadline in seconds. While some aspects of our work apply equally well to general predicates, we constrain the precondition to a logical conjunction of basic predicates and the postcondition to a basic predicate.

The deadline is a time interval such that if the precondition is true now, the postcondition will be true before the deadline passes, according to the rule. More formally, given any predicate \( p \) and deadline \( d \), if we define a \( \text{within}_{d,p}: S \rightarrow \{true, false\} \) as:

\[
\text{within}_{d,p}(s) = \begin{cases} 
true & \exists s’ \in S \text{ s.t. } 0 \leq time_{s’} - time_s \leq d \wedge p(s’), \\
false & \text{otherwise}
\end{cases}
\]

then a rule \( r \) formally evaluates as true in state \( s \) if and only if \( (\text{pre}(r))(s) \wedge \text{within}_{\text{deadline}(r), \text{post}(r)}(s) \).

We define the probability of predicate \( p \) being true as:

\[
P(p) = \frac{|\{i \in S: p(i) = \text{true}\}|}{|S|},
\]

and the conditional probability of \( q \), given that \( p \) is true, as:

\[
P(q \mid p) = \frac{|\{i \in S: p(i) = q(i) = \text{true}\}|}{|\{i \in S: p(i) = \text{true}\}|}.
\]

Hence, we define the probability of success of a rule \( r \) as:

\[
P(r) = P(\text{within}_{\text{deadline}(r), \text{post}(r)} \mid \text{pre}(r)).
\]

In this paper we limit possible values for \( \text{deadline}(r) \) to \([0s, 0.11s]\)—0.11s is chosen in order to suit the domains we evaluate rule learning in. We add an additional restriction in that if \( \text{deadline}(r) \neq 0 \), then \( \text{pre}(r) \) must include \( \neg \text{post}(r) \) as one of its predicates. Note that while \( \neg \text{post}(r) \) is not a basic predicate, it can be transformed into one by changing the region referred to in \( \text{post}(r) \) to its complement, and so we treat it conceptually like one.

These restriction result in rules being split into two separate categories: immediate and temporal rules. Immediate rules describe relations between variables which hold true at the same time, while temporal rules require the value of a variable to change so that at first \( \text{post}(r) = \text{false} \) but later it becomes true.

B. Learning Clusters in Quadtree Probability Distributions

We use a quadtree-based data structure which we call the QTPD to store variable samples and estimate the variable’s probability density function, which is used as input to the cluster learning algorithm which finds regions of uniform probability density for use in model rules.

The cluster learning algorithm works as follows: First, two-dimensional variable samples in the form of \((x, y)\) points are inserted into a QTPD. Second, the QTPD is compressed, which results in samples being reduced into statistical data estimating the variable’s probability density. Finally, this compressed QTPD is given as input to a cluster learning algorithm which produces regions in the variable’s value space. These steps are described in detail in this section.
1) Adding Samples to a Quadtree Probability Distribution:
A quadtree probability distribution is an extension of the one-dimensional compressible distribution data structure used in our past work [1]. Quadtrees are a standard data structure and we will only describe the basics of how they work in our method.

A quadtree distribution is a tree of nodes where each node $n$ is a tuple $\langle \text{bounds}(n), \text{quarters}(n) \rangle$, where $\text{bounds}(n)$ describes the square area encompassed by the node and contains the $(x, y)$ coordinates of the centre of the square and its side. This area is split into four square quarters—quarters $(n)$. Each quarter $q \in \text{quarters}(n)$ can be of three types: node—indicating that the quarter contains another node; data—in which case the quarter can store a list of $(x, y)$ points, where each point is contained within the quarter’s bounds; and count—where the quarter only stores the number of points contained therein, but not their coordinates.

A quadtree starts with a single node with four data quarters and no points added. When a point is added, it is directed to the leaf quarter whose bounds contain that point’s coordinates. If that quarter is a data quarter, it is added to the list of points. If it is a count quarter, its counter is incremented.

Data quarters are further subdivided into count quarters when the number of points stored in them exceeds a certain limit, but there is also a maximum depth at which quarters with too many points become count quarters.

In addition, if a point falls outside the root node’s bounds, then the tree is expanded and the root node becomes a quarter of a new, larger node.

Using a quadtree in this fashion allows the efficient storage of a variable’s sample values, and due to automatic partitioning into quarters it can be used to estimate a boundary around the samples. From this it is straightforward to obtain the sample density by dividing the number of samples in a quarter by the area of a quarter’s bounds, and the probability density if further divided by the total number of samples.

2) Estimating the Probability Density Function: Once all the sample values of a variable have been added QTPD, it is relatively easy to obtain an estimated probability density function for that variable.

This is achieved by compressing the QTPD. It works by modifying the tree in a bottom-up order, replacing all data quarters with count quarters and merging four quarters if they happen to have a similar sample density. See Algorithm 1 for a more detailed description.

3) Computing Clusters from the Distribution: Now that the QTPD contains an estimated probability density function, it can be used to extract regions with a similar density. The cluster learning algorithm which does this works in the following way: First, it creates a region matching the bounds of each leaf square in the QTPD. Then it adds all neighbouring pairs of regions to a priority queue, ordered by a similarity score metric. Finally, it iteratively joins the two most similar regions into a single region, while updating the neighbour pair data. It stops joining regions when a termination condition is reached.

### Algorithm 1 COMPRESS-QTPD(n) pseudo-code

```plaintext
1: Parameters: n - root node
2: for all q ∈ quarters(n) do
3:   if type(q) = node then
4:     COMPRESS-QTPD(node(q))
5:   else if type(q) = data then
6:     CONVERT-TO-COUNT(q)
7: end if
8: end for
9: if all of quarters(n) are count then
10:   d ← MIN-DENSITY(quarters(n))
11:   d ← d/MAX-DENSITY(quarters(n))
12:   d ← 1 - d
13:   if d ≤ 0.15 then
14:     p ← PARENT-QUARTER(n)
15:     CONVERT-TO-COUNT(p, quarters(n))
16:   end if
17: end if
```

a) Parameters: The parameters accepted by the clustering algorithm are: a QTPD distribution and a termination condition. A termination condition $t$ is a tuple $(\text{max-score}(t), \text{min-entropy}(t))$, containing two parameters such that if any one of them is exceeded, the algorithm finishes joining regions and returns.

b) Initial Setup: The algorithm maintains the following state: $Q$—neighbour pair priority queue; $E$—the set of current entries, containing the clusters. An entry $e$ is a tuple $(\text{squares}(e), \text{perimeter}(e))$, where $\text{squares}(e)$ is a set of squares corresponding to data quarters of the QTPD and $\text{perimeter}(e)$ is a set which contains all squares that neighbour entries other than entries belonging to $e$. Each square in $\text{squares}(e)$ is marked as belonging to $e$.

The perimeter is used to quickly find neighbouring entries: for each square in the perimeter, all neighbouring squares are obtained from the QTPD and their owning entries are added to the set of entries. Neighbouring squares are those which share a part of an edge or a corner of their respective square boundaries.

The total number of samples in $e$ is equal to the total number of samples in $\text{squares}(e)$, and similarly for the total area. These two values can be used to calculate the sample density and probability density of an entry.

Given entries $e, e'$, the neighbour priority queue is ordered according to the following metric:

\[
\text{score}(e, e') = \sqrt{\frac{1}{2}(P(e) + P(e'))} \times \left( 1 - \min\left(\frac{\text{density}(e), \text{density}(e')}{\max(\text{density}(e), \text{density}(e'))}\right) \right).
\]

The second factor is the similarity ratio which is equal to 0 for entries with identical density and approaches 1 for dissimilar regions, while the first factor is a bias against regions whose probability is very small, which causes them to be merged with other regions first.
As the first step of the initial setup, the algorithm populates $E$ such that for each square $q$ corresponding to a data quarter in the QTPD, the entry $\{q, \{q\} \}$ is added to $E$.

Finally, each pair of neighbouring entries $(e, e')$ is added to $Q$ with a priority of $score(e, e')$.

c) Iterative Merging of Entries: This is the main loop of the algorithm. At the beginning of each iteration it checks the termination condition. If the least priority in $Q$ is greater than the max-score of the termination condition or if the entropy of the regions in $E$ is less than min-entropy, the algorithm returns $E$.

d) Merging Two Entries: If the termination condition is not triggered, the next step is to merge the two neighbouring entries $e, e'$ with the least priority in $Q$. This is done as follows:

1. Any one of the entries is chosen as the destination entry. Let this be $e'$. (2) All neighbour pairs containing either $e$ or $e'$ are removed from $Q$. (3) All squares from perimeter($e$) are added to perimeter($e'$) and marked as owned by $e'$. (4) $e$ is removed from $E$. (5) Squares with no neighbouring entries other than $e'$ are removed from perimeter($e'$). (6) For each neighbour $e''$ of $e'$, the pair $(e'', e)$ is added to $Q$.

After the merge is performed, the algorithm goes back to the beginning of its main loop.

C. Learning Rules

Learning rules starts with an empty set of rules $R$. For each variable $v$, the rule learning algorithm creates either a QTPD or a compressible probability distribution $\{1\}$—depending on $dim(v)$—and adds all observed values of $v$ to its distribution. Now that this information is available, it proceeds to creating initial rules.

1) Creating Initial Rules: Regions are computed from each variable’s distribution. For each variable $v$ and each of its clusters $e$, the two rules: $\{\emptyset, in_{v,e}, 0\}$ and $\{\neg in_{v,e}, in_{v,e}, \emptyset\}$ are created and added to $R$, where $\emptyset$ is the deadline parameter (0.11s).

2) Specialising Rules: Specialising existing rules works similarly to the rule specialising algorithm described in $[1]$, where it is described in detail. Here we give an overview. For the purposes of specialising rules, the set of initial rules becomes the first generation. From each generation, a subsequent generation is produced. New rules arise from rules from the previous generation with one precondition predicate added, so that all rules in the $n$th generation have $n - 1$ or $n$ predicates in the precondition. Once a generation with no rules occurs, or a rule limit is reached, the process is stopped. Rules from every generation are added to the set of all rules.

In a given iteration of the algorithm, a specialisation procedure is invoked for every rule in the current generation. This procedure first computes, for each appropriate variable $v$, an approximate probability density function from the distribution $D_v(x) = P(v_i = x \mid post(r))$, which is stored in a compressible distribution data structure or QTPD. The variable $v$ can be any variable, except that for instantaneous rules, $v$ cannot be the same as the variable in $post(r)$ and it cannot appear twice in $pre(r)$. Similarly for temporal rules, with the additional allowance that $v$ may be the same as the variable in $post(r)$, and that variable is also allowed to appear twice in $pre(r)$.

As the next step it creates regions from this distribution and, for each region $c$, proceeds to create and evaluate a new rule, derived from the current rule but with an added precondition predicate of $v_i \in c$. At this point, the new rule $r'$ is evaluated. If there are at least $\omega$ occurrences and $P(r') - P(r) > \theta$, the evaluation continues, otherwise $r'$ is discarded.

IV. Evaluation

We evaluate the algorithm described in Section $[III]$ in a dynamic continuous domain involving a ball being thrown at hilly ground, simulated using the Box2D physics engine. In a scenario such as ours, where an agent learns about the world without guidance, the resultant model cannot be evaluated in the context of a particular task. To address this issue, we also put forward certain criteria which may serve as quantitative indicators of the quality of a model.

A. Criteria

Given a model consisting of set of rules $R$, there are many possible qualities to look for and in the absence of an absolute reference point, it is difficult to judge a model on its own, but it may be practical to compare it to another model $R'$ in order to assess the relative quality or the same model on a different set of data in order to assess transferability.

1) Model Size: An obvious factor to consider is the size of the model in terms of the number of rules, which is likely to affect any other metric to which the model is subjected. Therefore, in order for any of the following criteria to be used for comparison, the models $R$ and $R'$ need to have the same upper limit on the number of rules.

2) Mean Rule Probability of Success: It can easily be shown that if we were to use a probabilistic model for planning, the length of a feasible plan is limited by the probability of success of the rules the plan depends on. We compute this by averaging the probability of success of all rules in the model.

3) Mean Prediction Rate: Whenever a variable changes value, we collect all rules which have that variable in the postcondition, then remove all those whose preconditions would not have been true for that change in value. Among all remaining rules, the most desirable characteristics are high probability of success and small area—or interval for one dimension—of the region, since narrower postconditions make for more precise rules. We pick the rule $r$ with the highest $P(r')/area(region(post(r)))$. To compute this parameter we take the mean $Pr(r)$ for each variable change where we can pick such a rule $r$. The MPR is intended as a rough guide for how suitable for planning the model is.

4) Mean Mean Rule Contention: Whenever the preconditions of two different rules are true in a given state at the same time and these rules also make predictions about the same variable, a choice needs to be made as to which rule to follow. When this happens with $n$ rules, we say that the rules contend with each other, with the contention being $n - 1$ for each rule. We compute this parameter by taking the mean per-rule contention and then taking the mean among those values.
to how they are cached in our software.

rules use a one-dimensional-only model learnt using the same
dimensional and two-dimensional regions, while the last three
The first three rows describe a model with both mixed one-

D. Results

Table I: Evaluation results

<table>
<thead>
<tr>
<th>Test data</th>
<th>1D only</th>
<th>MS</th>
<th>MRPS</th>
<th>MPR</th>
<th>MMRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning set</td>
<td>no</td>
<td>851</td>
<td>0.4170</td>
<td>0.0402</td>
<td>28.8357</td>
</tr>
<tr>
<td>Eval. set 1</td>
<td>no</td>
<td>851</td>
<td>0.0119</td>
<td>25.6495</td>
<td></td>
</tr>
<tr>
<td>Eval. set 2</td>
<td>no</td>
<td>851</td>
<td>0.0114</td>
<td>26.2537</td>
<td></td>
</tr>
<tr>
<td>Learning set</td>
<td>yes</td>
<td>450</td>
<td>0.4193</td>
<td>0.0111</td>
<td>18.3077</td>
</tr>
<tr>
<td>Eval. set 1</td>
<td>yes</td>
<td>450</td>
<td>0.0111</td>
<td>18.4119</td>
<td></td>
</tr>
<tr>
<td>Eval. set 2</td>
<td>yes</td>
<td>450</td>
<td>0.0072</td>
<td>16.9411</td>
<td></td>
</tr>
</tbody>
</table>

B. Interpretation of the Criteria

Given two models of the same size, the above criteria
may tell us which model is more appropriate for planning
or recognition. Alternatively, if we evaluate the same model
in two different episodes or state sets belonging to the same
domain, they may tell us how well the model transfers to other
scenarios.

C. Experimental Setup

The ball simulation consists of a ball, uneven ground and
impassable screen edges, which the ball can bounce off. The
ball diameter is 2m, the screen is considered 60m tall and
proportionally wide. Every simulation run consists of 100
consecutive episodes where the ball is reset to the centre of
the screen, 48m above the bottom and dropped in a random
direction towards the ground at 10m/s. The force of gravity is
set up to accelerate the ball at $9.81 \text{ms}^{-1}$ downwards. The
simulation runs at 60 updates per second, and after each
update outputs variable values. One-dimensional variables are,
time, angle, angular velocity, collision distance, collision angle
and altitude; while two-dimensional variables are position and
velocity. Most are self-explanatory, except collision distance,
which is the estimated distance to the future collision point and
collision angle, which is the angle representing the slope of the
terrain at that point. Two-dimensional variables were accessible
to the learning algorithm both under a single vector variable
and as two one-dimensional variables, labelled e.g. position[0]
and position[1].

From this simulation we made three different state se-
quence logs: a baseline log used to learn the model from;
an evaluation log with the same terrain shape but different
randomly generated ball trajectories; and an evaluation log
with a different randomly generated terrain.

We ran the algorithm on the first log and then computed
evaluation parameters on all three logs separately. In addition,
we ran the algorithm with only one-dimensional variables
enabled. We left all learning parameters as specified in Section III. We set the rule limit at 1000. For the termination
criterion in generation $n$, we used a max-score of $0.042 \times 0.4^n$
and a min-entropy of $3.182 \times 1.67^n$ for two-dimensional
regions ($n$ starting at 0).

D. Results

Table I provides a summary of the evaluation results.
The first three rows describe a model with both mixed one-
dimensional and two-dimensional regions, while the last three
rules use a one-dimensional-only model learnt using the same
data and parameters. Some MRPS values are unavailable due
to how they are cached in our software.

What is apparent from the numbers is that the models
finished learning before reaching the maximum size.

The results also suggest that mixed models show an im-
provement over one-dimensional models in their predictive
power. While the one-dimensional model is smaller, the two-
dimensional model has the same one-dimensional rules and
therefore, on average, two-dimensional rules performed better
than one-dimensional rules.

Unsurprisingly, both models show a decline in prediction
rate when the ground data is changed in the environment. However, they still show a relatively high prediction rate
compared to the learning set.

The data also suggests that rule contention is positively
correlated with prediction rate, which suggests insufficient
measures to reduce contention while preserving prediction rate.

As an example, Figure 1 shows the progression from
sample data to regions for the position variable in the two-
dimensional learning set. Even though the data is relatively
sparse, the algorithm generalises sufficiently to produce solid
regions. Figure 2 shows some example rules and regions.

V. CONCLUSIONS AND FUTURE WORK

We have presented an extension to unguided rule learn-
ing involving a novel approach to learning two-dimensional
regions for rules in a world model from variable probability
distributions.

Our approach could be improved in several ways. Firstly,
our rule-learning is performed off-line from observation logs
which is not ideal for a fully autonomous agent. Because
probabilities are evaluated on the whole log, performance
scales with log size, and there are currently no mechanisms
to provide an upper bound for on-line updating.

A general drawback of discretising continuous spaces
based on probability distributions is the difficulty of captur-
ing linear relationships between variables or when they are
uniformly distributed. It seems inefficient to discretise such
variables into small regions; a predicate expressing a symbolic
relationship would be more suitable for such a case.

Our technique is still limited to one or two dimensions.
Real-world problems often have many dimensions. An ad-
antage of a quadtree is that it can be generalised easily to
higher dimensions and in principle the techniques presented
here could be generalised this way.

It would be useful to synthesise new variables by combin-
ing existing variables in certain ways: subtracting two vectors,
taking the length of a vector or automatically calculating a
derivative of a variable might improve autonomy.

Finally, the algorithm could be optimised to maximise the
rule set evaluation parameters in order to have more practical
use, which would require closer filtering of the rules and
perhaps learning chains of rules useful for planning.
Figure 1: (a) Screenshot of the ball simulation. (b) QTPD filled with ball position samples before being compressed. (c) Regions obtained from the QTPD shown in (b) using our quadtree cluster learning algorithm.

Figure 2: Random rule examples. <region> labels refer to two-dimensional regions. The value before the rule is the probability of success. Region images are at varying scales.

REFERENCES


