A Comparison of Case-Based Reasoning and Probabilistic Graphical Models in the Context of Learning from Observation

by

Amrik Sacha Elapata Gunaratne

A Thesis submitted to

the Faculty of Graduate and Postdoctoral Affairs

in partial fulfillment of

the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, ON

Canada

May 7, 2018

Abstract

Learning from observation is a technique whereby learning occurs through observation or experience. In this work, we compare two existing techniques of learning from observation: Probabilistic Graphical Models (PGM) and Case-Based Reasoning (CBR) with the goal of identifying a preferred approach for future improvement. We show that the Naive Bayes Classifier is better than a previously used PGM model in learning behavior in a vacuum cleaner domain and introduce a state-based retrieval technique in CBR and show that there is no once-size-fits-all approach to learn state-based behavior. We also compare the two learning techniques in fully and partially observable continuous domains, namely Cartpole V0, obstacle avoidance, and 2D RoboCup. We show that the CBR approach works best in Cartpole V0, the PGM approach works best in obstacle avoidance, and the PGM approach works best in 2D RoboCup. Ultimately, we show that the preferred technique is generally behavior and domain specific.

Dedication

To mom, dad, nats, aiti and rala. Couldn't have done it without you.

Abbreviations

Table 1: Abbreviations

BN	Bayesian Network
CBR	Case-Based Reasoning
CPD	Conditional Probability Distribution
CPT	Conditional Probability Table
DBN	Dynamic Bayesian Network
DGBN	Dynamic Gaussian Bayesian Network
EM	Expectation Maximization
GNBC	Gaussian Naive Bayes Classifier
LfO	Learning from Observation
PGM	Probabilistic Graphical Model
PPT	Past Problem Threshold
RPT	Recent Problem Threshold
SEQ	Fixed Sequence Agent
ST	Solution Threshold
SWF	Smart WallFollower Agent
TB	Temporal Backtracking
WF	WallFollower Agent
ZZ	Zig Zag Agent

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 Why Learning from Observation?

The goal of a Learning from Observation (LfO) agent is to reproduce expert behavior by capturing traces of the behavior and using it as training data [1]. This data generally is in the form of an input (made up of a set of features) and an action. During testing and deployment, the agent is presented with a new, potentially unseen input and will use the insight gained from the training data to choose an action. LfO agents can be trained and used in multiple domains, and as such, they have been successfully used to learn 2D simulated soccer [2], play real-time strategy games [3], and learn simulated vacuum cleaner behavior [4].

In LfO, the learning agent is not provided with the expert's goal and has to infer it from the training data. There is also no aspect of reinforcement learning.

This is because LfO is used in cases where designing a reward function is too complex and requires considerable domain expertize, or where the user has no programming ability and has to simply demonstrate the required behavior to the agent.

1.1.2 Different Types of Expert Behavior

There are two main types of behavior learned using LfO: *reactive* and *state-based*. Reactive behavior is reliant only on the current environment state. A reactive agent will always perform the same action for a given environment state. State-based behavior is reliant on the current and past environment state and actions. Therefore, it may perform different actions for a given environment state, based on its past behavior. To keep track of the past relevant environment states, and actions, and inform the current decision-making process, the state-based agent has some internal state. While learning reactive behavior only involves identifying the function mapping environment states to action, learning state-based behavior also involves learning the internal state of the expert, which can be more complicated.

1.1.3 Different Approaches to LfO

There are multiple techniques used to learn behavior in LfO. Floyd uses Case-Based Reasoning (CBR) to learn the expert behavior of a soccer player in 2D simulated soccer [2]. Ontañón et al. uses Probabilistic Graphical Models (PGM) and Neural Networks (NN) to learn the behavior of a vacuum cleaner expert [4].

CBR represents the problem by converting the expert traces into input-action pairs and then stores them for the agent to access when making new decisions. When the CBR agent needs to make a new decision, it retrieves the cases most similar to the current query. PGMs identify probability distributions that the inputs follow and attempt to learn the parameters of the distributions. NNs approximate the function that was used to choose the actions given the input. All of the techniques will have some inductive *bias* to help make a decision in previously unseen situations, which will affect how the actions are chosen. The term bias is used to refer to "any basis for choosing one generalization over another, other than strict consistency with the observed training instances" as defined in [5].

Given these different approaches and this idea of bias, some of the questions that can be asked are: Is there a way to choose which algorithm will work best for a given expert behavior? Is there a one-size-fits-all algorithm that can learn all types of behavior? Can we determine whether the PGM or CBR technique is the most promising in order to investigate it further?

1.1.4 State-Based CBR

The CBR approach to learning *state-based* behavior is the Temporal Backtracking algorithm [6]. This algorithm infers the internal state of the expert by looking at a history of environment state-action pairs. TB applies a specific type of inductive bias when choosing actions, specifically a recency bias. There could be other biases in the behavior that might not be captured using this algorithm. Therefore, there should be other general algorithms that can learn different types of biases.

1.1.5 Issues with Current Frameworks for LfO

A LfO framework should be one that allows the user to compare and contrast multiple techniques to learn different expert behavior across domains using a variety of performance measures. It should ideally have multiple algorithms for learning state-based and reactive behavior.

As far as we are aware, such a framework does not exist. For example, the framework used in [4]:

- is directly tied to performing LfO in the discrete 2D Vacuum cleaner domain,
- the tools used for performing learning and inference were limited to the discrete domain, and
- the framework was only able to output accuracy as a performance measure when performing testing, which can be a limitation with imbalanced datasets.

A different framework called Java Learning from Observation Framework (jLOAF) [7], which used CBR, was initially tightly coupled with the 2D simulated soccer domain RoboCup. It has been modularized to be able to handle multiple domains; however, it lacked functionality for performing cross-validation testing, producing performance measures and writing the results to databases. It also only had a bare-bones version of a k-nearest neighbor algorithm (kNN) as its retrieval algorithm for CBR. This limited functionality was not ideal for testing state-of-the-art LfO techniques.

In order to determine if one type of learning algorithm is better at imitating a specific expert behavior in a specific domain, the algorithms should ideally be applied within the same framework, on the same data, and compared using the same performance metrics. This unification of domains, performance measures, and multiple LfO techniques under one umbrella will create a baseline for comparison of the state of the art in LfO. It will provide a starting point for answering the questions asked previously, as well as identifying flaws of and improvements to the techniques in question.

1.2 Contributions

First, the changes made to unify the framework, and the methodology for testing *state-based* learning, and the contributions to jLOAF are presented in Chapter 4. Using this unified framework, we were able to identify areas for immediate improvement and investigation across the CBR and PGM techniques, with the objective of identifying a preferred technique for future development. Figure 1.1 depicts the areas in which contributions are made in this thesis. As can be seen, there are two main areas of focus: bias and comparative testing in the continuous domain. Further description of each contribution is provided in the sections below.

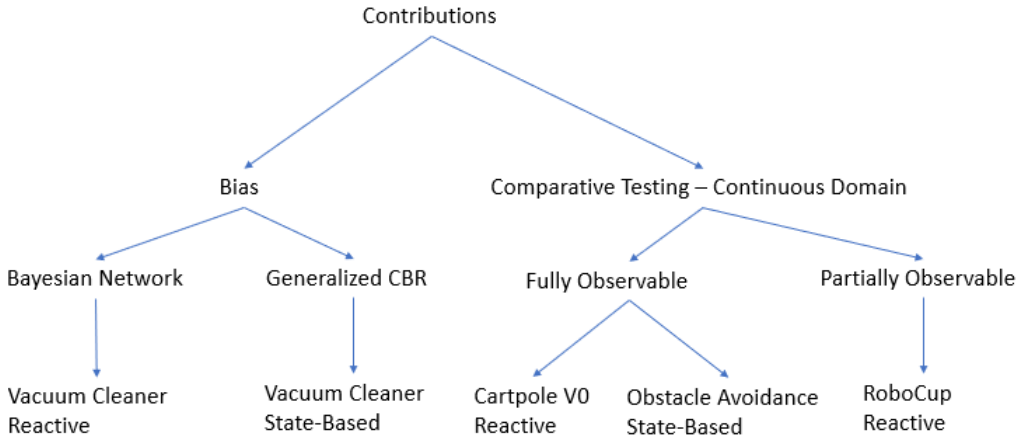


Figure 1.1: Road map of contributions

The bottom row describes the domain and the type of behavior being tested.

1.2.1 Performance Testing of State-Based Behavior (Chapter 4)

In the PGM framework, the authors used DBNs to learn *state-based* behavior. Recall that in *state-based* behavior, the past actions and environment states can affect the current behavior. During testing of DBN-based learning, if the learning agent predicts an action that is different from the expert's, and stores it in its memory, it will lead the agent on a different trajectory from the expert trajectory. Since the expert and agent are no longer on the same trajectory, the expert's action and the agent's action can no longer be compared. To ensure that the agent and expert are always on the same trajectory, the agent's action is intercepted and replaced by the expert action. We found that the DBN agent was saving the action that it performed, and this was the cause for the poor reproduction of the DBN

learner's results in the vacuum cleaner domain in [8]. Once the fix specified above was implemented, the results that were produced matched those of the original paper and can be seen in [9].

Unlike the DBN approach, the TB algorithm doesn't need to keep a memory of the past. It is able to infer the internal state of the expert from the expert traces it is provided with. The ability to account for the differences in testing TB and DBN approaches, and other functionality was implemented in the jLOAF framework as described in the next subsection. Note that while improving jLOAF is mostly a contribution of a technical nature instead of a contribution to knowledge, the discussion is included as it informs the rest of the work that we performed.

Additions to jLOAF

As mentioned above, there were two frameworks that were performing LfO using different techniques - jLOAF and the PGM Framework. The structure of jLOAF was already modular and, therefore, it made sense to move the required functionality from the PGM framework to jLOAF. This included the addition of Bayesian networks (BN), Dynamic Bayesian networks (DBN) and neural networks. These learning methods interfaced with the BNET toolbox [10], and required Matlab to perform learning and inference. The cross-validation technique used in the PGM framework for performing testing were also moved over to the jLOAF framework. This allowed for testing the DBN and BN learning agents against the CBR approaches in different domains to determine if there was a preferred type of algorithm for a specific behavior and domain.

1.2.2 Bias (Chapter 5)

Adding Bias into the Bayesian Network (Chapter 5.1)

As mentioned previously, machine learning algorithms require bias for choosing actions in previously unseen situations, given a particular environment state. In the case of the BNs used by Ontañón in [4], there was very little bias due to the structure of the model. The model created a large look-up table during training and queried it when a new input was provided. We add bias into the BNs by changing its structure to the Naive Bayes classifier. We use the two techniques in the vacuum cleaner domain to learn *reactive* behavior, and show that given the use of complex maps, the Naive Bayes classifier outperforms Ontañón's model.

Generalized Case-Based Reasoning (Chapter 5.2)

An important change that was made to the jLOAF framework was the representation of a case. As described previously, a case represents an input-action pair and is generally thought to be independent of other cases. However, if the behavior is *state-based*, it means there could be temporal connections between the cases. In order to represent this, the case is redefined to include the input, the action and the history of inputs and actions. This representation is defined in [6] and will also be presented in chapters 3 and 5. This change in structure led to the generalized version of CBR, which was able to handle learning both *state-based* and *reactive* behavior. In the case of *state-based* behavior, it introduces the possibility for the user to specify the similarity metrics that compare histories of past cases and not

just individual cases.

Given the new functionality, we developed new similarity metrics that compared entire histories, and created *state-based* experts that behaved according to certain biases in order to test them. The expert behavior was learned by pairing the similarity metrics with a retrieval algorithm and the performance was compared to the TB algorithm. We were able to show that TB is not the only CBR method for performing *state-based* learning, and that it may not always be the best approach.

1.2.3 Comparison of CBR-Based and PGM-Based Techniques in the Continuous Domains (Chapter 6)

Comparison of CBR-Based and PGM-Based Techniques in Fully Observable Continuous Domains (Chapter 6.1 and Chapter 6.2)

Initially, the models that were used in the PGM framework were only able to handle the discrete domain. This was because they were tied to the vacuum cleaner domain. Since one of the focuses of this thesis is characterizing the performance of CBR and PGM techniques across multiple domains, there was a requirement to test them against each other in domains other than the discrete vacuum cleaner domain. This was also an avenue of future work expressed in [8]. Therefore, models were designed in order to handle mixed domains. A comparative test between the CBR and PGM techniques used to learn *reactive* behavior was performed in the *fully observable continuous domain* - Cartpole V0 [11]. The techniques used to learn *state-based* behavior were tested in a fully observable continuous domain -

obstacle avoidance (developed by [6]).

Comparison of CBR-Based and PGM-Based Techniques in a Partially Observable Continuous Domain (Chapter 6.3)

One of the areas that CBR is able to handle well is the partially observable domain. Therefore, it was used in the 2D simulated soccer domain, RoboCup [2]. A new BN model was created to test the PGM's ability to handle the partially observable domain, and compared to the CBR approach as well as the preexisting BN model from the fully observable domain. However, the model being so specific to the domain raises the question about the ability to generalize to other domains.

1.3 Publications

Two peer-reviewed publications resulted from the work that has been done for this thesis:

- Gunaratne, Amrik, Esfandiari, Babak, and Chan, Caleb. Towards a Framework for Testing Learning from Observation of State-Based Agents. *AAAI Spring Symposium Series, California, USA*, SS-17-06:499505, 2017

- Gunaratne, Amrik, Esfandiari, Babak, and Fawaz, Ali. A Case-Based Reasoning Approach to Learning State-Based Behavior. *The Thirty-First Proceedings of the Florida Artificial Intelligence Research Society (FLAIRS), Florida, USA*, Special Track **Case-Based Reasoning**, 2018

1.4 Structure

Chapter 2 provides the background required to understand the proposed contributions. Chapter 3 briefly describes the state of the art and some of the approaches taken to solve some of the problems described in the introduction. Chapter 4 describes the testing methodology used to test *state-based* behavior and provides preliminary results and insight into the contributions. Chapter 5 describes the improvements made using the notion of bias in the BNs and the improvements made in the *state-based* CBR learning techniques. Chapter 6 describes the comparative study of the CBR and PGM techniques in learning behavior in the fully and partially observable continuous domains. Chapter 7 presents conclusions, recommendations and future work.

Chapter 2

Background

2.1 Learning from Observation

2.1.1 LfO: A Definition

LfO was introduced briefly in Chapter 1 and now will be defined in a more complete manner: the expert that the agent is learning from is situated in an environment E, which is a finite set of discrete, instantaneous states (here we will use the definitions and notations of [12]):

$$E = \{e_0, e_1, e_2, .., e_n\}$$

and is able to perform a set of actions Ac, which transforms the state of environment:

$$Ac = \{a_1, a_2, a_3..., a_n\}$$

An agent's goal when learning from observation is to reproduce the expert's behavior, which is in the form of sequences of environment state-action pairs. A visual representation of this is shown in Figure 2.1. The learning agent doesn't have access to the decision-making process of the expert (i.e., its internal state), but has to infer it from the environment state-action pairs.

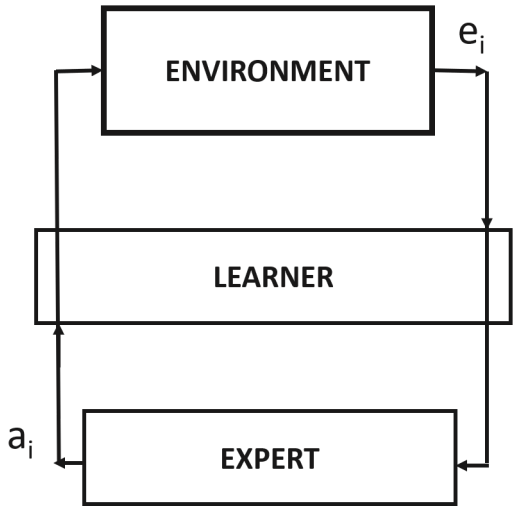


Figure 2.1: Learning from observation cycle

The next section will define the agent and its behavior.

2.2 Agents and Behavior

2.2.1 Defining Runs

A sequence of environment state-action pairs is built in the following manner: the environment is in some initial state and the expert chooses an action that affects the environment. As a result, the environment will be in a new state and the expert will choose an action based on this new state. This sequence of environment state-action pairs is called a *run* [12] and any given run *r* can be represented in the following manner (where the e_i represent environment states and the a_i represent the agent actions):

$$r : e_0 \xrightarrow{a_0} e_1 \xrightarrow{a_1} ... \xrightarrow{a_n} e_n$$

Let R be the set of all possible finite runs, R^E be the subset of R that contains runs that end in an environment state and R^{Ac} be the subset of *R* that contains runs that end in an action.

2.2.2 Defining Agents

An agent can be formalized as a function Ag:

$$Ag : R^E \rightarrow Ac$$

The agent requires the run to end in an environment state so that it can choose an action in response. This definition captures the fact that an agent's past actions

and environment states may affect its current action. Woolridge defines this as a *standard* agent [12].

A *reactive* agent is a type of standard agent that only depends on the current environment state to make a decision about its current action. It is defined below:

$$Ag_{Reactive} : E \rightarrow Ac$$

The standard agent model can be used to represent agents that are influenced by the past, through the runs. An agent that relies on its past events to make its current decisions can also be called a *state-based* agent. Woolridge provides a definition for the *state-based* agent as well.

$$action : I \times E \rightarrow Ac$$

$$next : I \times E \rightarrow I$$

In the above equations, I is an internal state that is updated each time the agent receives a new environmental state. This internal state captures the salient past events stored in the expert memory. According to Woolridge, the standard agent and the state-based agent are equivalent in expressive power, and one can always be converted into the other. An investigation into this claim is shown in Chapter 2.4.

The next section describes the different characteristics that the environments in which the agents are situated can have. It also describes the domains used for the comparative performance study of CBR vs PGM, as well as validation of the

improvements made due to the introduction of bias.

2.3 Environments

Environments can be categorized based on certain characteristics. The following characteristics are relevant to the investigation into LfO because they encapsulate a large subspace of environments, and we wanted to test the CBR and PGM approaches across multiple domains.

2.3.1 Discrete vs Continuous

If there is a distinct and countable number of environment states and actions, the domain is said to be discrete. An example of this is chess. The features of the chess game would be the positions of the pieces on the board, and one of the actions available to the player would be to move a piece to another square. Conversely, a driving simulation will have uncountably infinite environment states and actions, as the angles of the steering wheel or the velocity of the car are continuous. In general, if one of the features of the environment is based on a real value, it is considered a continuous environment. Therefore, most real-world settings are continuous.

2.3.2 Partially Observable vs Fully Observable

If the agent can sense the whole environment, it is fully observable. Once again, chess is an example of this. There are no hidden environment states or information

to which the agent does not have access. Conversely, the game of poker is partially observable. The agent has access only to the cards in its hand and the cards on the table. It does not have access to the cards in the deck or the cards in the opponent's hands.

2.3.3 Domains used for Testing and Validation

A summary of the domains used for testing purposes is shown in Table 2.1. Each domain was chosen because it represents a different combination of the characteristics of environments described above.

Domain Name	Discrete or Continuous	Observability
Vacuum Cleaner	Discrete	Fully Observable
CartPole V0 (Open AI)	Continuous	Fully Observable
Obstacle Avoidance	Continuous	Fully Observable
RoboCup	Mixed	Partially Observable

Table 2.1: Domains used for testing

The domains are described in detail below.

Fully Observable Discrete Domain: Vacuum Cleaner

The vacuum cleaner domain consists of a grid world that contains obstacles, dirt and walls (see Figure 2.2). For the purposes of our agents, the walls and obstacles are indistinguishable. The environment state is represented by 8 binary features, which can be separated into 4 pairs. The pairs represent the features perceived by the agent in the NORTH, EAST, SOUTH and WEST directions. The first value in the pair represents whether the agent sees *dirt* (represented by 1) or *obstacle/wall*

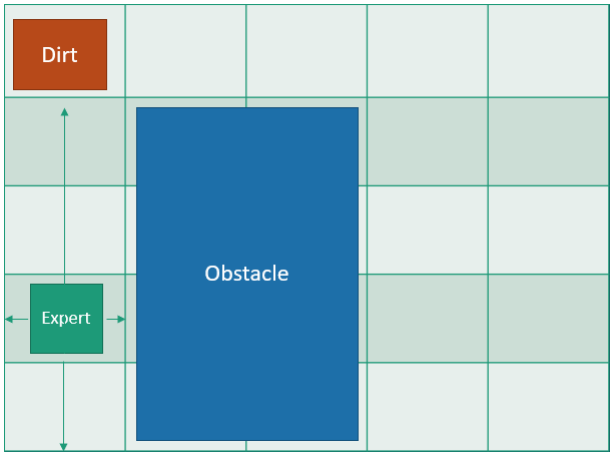


Figure 2.2: Agent in vacuum cleaner domain

The Environment State the agent perceives is 110001001 ([*dirt*, *far*],[*wall*,*near*],[*wall*,*far*],[*wall*,*near*])

(represented by 0). The second value represents whether the object seen by the agent is *near* (represented by 0) or *far* (represented by 1), with "near" meaning in a square adjacent to the agent. The agent is able to perform 5 actions: *Up*, *Down*, *Left*, *Right* and *Stand*. Agents that exhibit various reactive and state-based behaviors can be built and tested in this domain. Some of the behaviors used in the domain are described below (created by [4]):

Reactive Agents

- WallFollower (WF) - This agent follows the *wall* in a clockwise direction. If there is no *wall* being touched, it will move *Right*.
- SmartWallFollower (SWF) - This agent has the same behavior as the WF agent, but if it sees *dirt*, it will go straight for it.

State-Based Agents

- ZigZag (ZZ) - This agent moves in zig-zags. It moves all the way to the right, until it collides with an obstacle or wall, then moves down one square and moves to the left until it collides with an obstacle or wall. When it cannot go down any further, it repeats the behavior, but going up, and so on. To perform this behavior, an agent must remember whether it is going left or right, and also whether it must go down or up after colliding with an object.

- Fixed Sequence (SEQ) - This agent loops over the same fixed sequence of actions (15 actions long), and is independent of the environment state.

Fully Observable Continuous Domain: Cartpole V0 - Open AI Gym

Cartpole V0 [11] is used to test and validate the CBR and PGM methods for learning *reactive* behavior in a fully observable continuous domain. The goal of the game is to balance a pole on a cart by moving the cart left or right. The game play is shown in Figure 2.3. This game is geared towards testing and validating reinforcement learning algorithms, but once the expert is created, learning traces of expert behavior can be generated and used to train an LfO agent.

The expert has access to two actions, *Left* and *Right*, represented by 0 and 1. Each cycle, the environment state is represented by 4 observations. These are shown in Table 2.2.

A sample trace is shown in Table 2.3:

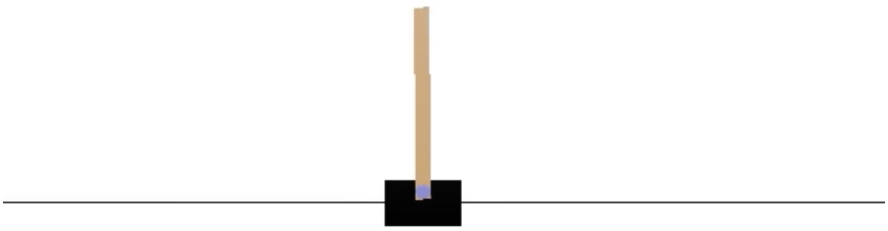


Figure 2.3: Cartpole V0 - game play

Number	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-41.8	41.8
3	Pole Velocity at Tip	-Inf	Inf

Table 2.2: Observations of environment state (Cartpole V0)

Fully Observable Continuous Domain: Obstacle Avoidance

This domain is used to test and validate the CBR and PGM methods for learning *state-based* behavior in a fully observable continuous domain. In this game, which was used in [6], an expert will traverse a maze and attempt to avoid the obstacles. Each environment state is represented by one continuous feature: *Sonar*. The expert has access to five actions: *Left*, *Right*, *Forward*, *Reverse (Turn 180°)* and *Backward*. A sample trace is shown in Table 2.4.

The expert that is used to generate the sample trace is defined in Algorithm 1.

Cart Position	Cart Velocity	Pole Angle	Pole Velocity at Tip	Action
-0.048983009	0.039441863	0.012501582	-0.026031438	0
-0.048194172	-0.15585711	0.011980953	0.27056951	0
-0.051311314	-0.35114797	0.017392344	0.56700708	0
-0.058334273	-0.54650952	0.028732485	0.86511818	1

Table 2.3: Sample trace (Cartpole V0)

Sonar	Action
3.86	Forward
3.63	Forward
2.26	Right
3.55	Forward
2.25	Left

Table 2.4: Sample trace (obstacle avoidance)

The expert will go *Forward* if the *Sonar* is greater than 3, it will turn *left* or *Right* if the *Sonar* is in between 2 and 3, depending on whether the expert turned *left* or *Right*, the last time *Sonar* was in between 2 and 3. It will *Reverse* if the *Sonar* is in between 2 and 1, and it will go *Backward* if the *Sonar* is less than 1. The expert is *state-based* because it has to remember whether it last turned *Left* or *Right* when $2 < Sonar < 3$.

Partially Observable Continuous Domain: 2D Simulated RoboCup

To test the ability of the CBR and PGM approaches to learn partially observable continuous *reactive* behavior, we chose 2D RoboCup simulated soccer. RoboCup's main research goals focus on "cooperative multi-robot, and multi-agent systems in dynamic adversarial environments" [13]. It began in 1997 and since then has hosted both robotic and simulated soccer tournaments with autonomous agents in

Algorithm 1 Left-Right Toggle Agent

```
1:  turnLeft ← False
2:  function GETACTION(input, turnLeft):Action
3:      Sonar ← input.getSonar()
4:      if Sonar<1 then
5:          return Action: Backward
6:      else if Sonar < 2 then
7:          return Action: Reverse
8:      else if Sonar < 3 then
9:          if turnLeft == True then
10:             turnLeft ← False
11:             return Action: Left
12:         else
13:             turnLeft ← True
14:             return Action: Right
15:     else
16:         return Action: Forward
```

numerous countries across the world.

Agents in the game have access to three actions: *Kick*, *Dash* and *Turn*. These actions take parameters such as *Angle* and *Power*. The agent's view of the environment is in the shape of a cone that spans +45 and -45 degrees from its forward-facing direction. It receives perceptions of the location of the ball, goals, players and flags in terms of relative angle and distance. The angle data ranges from -90 to 90 and the distance data ranges from 0 to 500. Each cycle, the agent sees the field and then performs an action. These make up the environment state-action pairs that represent the agent behavior. Figure 2.4 shows the field and the agent's view.

In Figure 2.4, the player is able to see objects c, d, e and f, as they are in its field of vision. It cannot see objects b and g. Object a is a special case because it

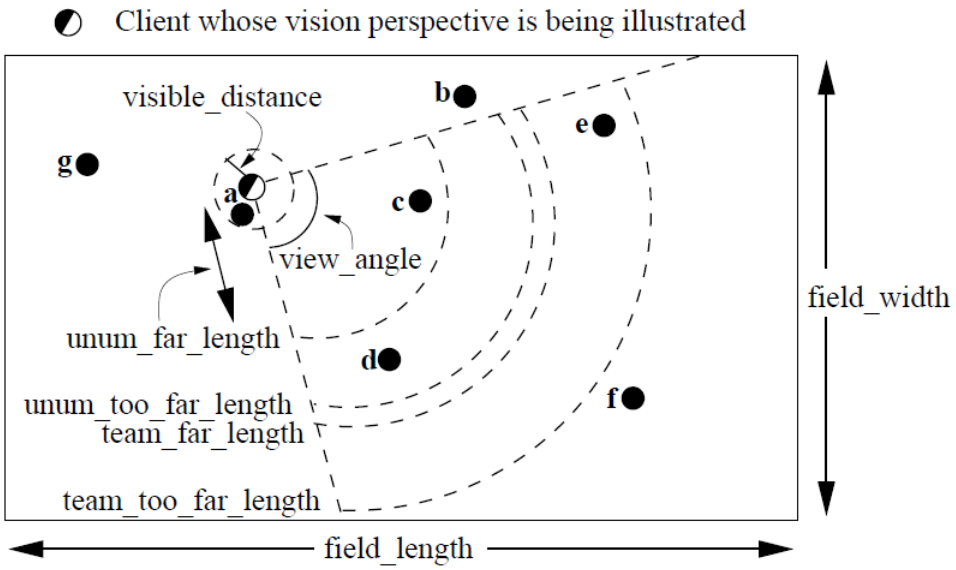


Figure 2.4: Representation of RoboCup field

RoboCup User Manual [14]

is behind the player, but is considered close enough to be visible.

The domain provides real valued features that are sometimes not observed, and therefore is suitable for testing the continuous and partially observable capabilities of the CBR and PGM approaches. The following agents were used for testing purposes: Krislet (Reactive), defined in Algorithm 2, and KrisletiflastDashed (State-Based), defined in Algorithm 3.

Krislet has the following behavior: It will *Turn* until it can see the ball, *Dash* until it is next to the ball, and then if it cannot see the goal it will *Turn* until it does, and then *Kick* the ball in the direction of the goal. If it is near the ball and can see the goal it will *Kick* the ball in the direction of the goal.

Algorithm 2 Krislet Agent

```
1:  function GETACTION(input):Action
2:      Ball ← getBallInfo(input)
3:      Goal ← getGoalInfo(input)
4:      if Ball==null then
5:          return Action: Turn(40)
6:      else if Ball.distance > 1.5 then
7:          if ¬(Ball.direction == 0) then
8:              return Action: Turn(Ball.direction)
9:          else
10:             return Action: Dash(10*Ball.distance)
11:     else
12:         if Goal==null then
13:             return Action: Turn(40)
14:         else
15:             return Action: Kick(100,Goal.direction)
```

Algorithm 3 KrisletiflastDashed Agent

```
1:  lastDashed ← False
2:  function GETACTION(input, lastDashed):(Action, lastDashed)
3:      Ball ← getBallInfo(input)
4:      Goal ← getGoalInfo(input)
5:      if Ball==null then
6:          lastDashed ← False
7:          return Action: Turn(40)
8:      else if Ball.distance > 1.5 then
9:          if ¬(Ball.direction == 0) then
10:             lastDashed ← False
11:             return Action: Turn(Ball.direction)
12:         else
13:             lastDashed ← True
14:             return Action: Dash(10*Ball.distance)
15:     else
16:         if Goal==null then
17:             lastDashed ← False
18:             return Action: Turn(40)
19:         else
20:             if lastDashed==False then
21:                 return Action: Kick(100,Goal.direction)
22:             else
23:                 lastDashed ← False
24:                 return Action: Turn(40)
```

The design choice behind KrisletiflastDashed is arbitrary. There is one state variable that captures whether the last action performed was Dash. As can be seen from algorithm 3, when the expert is near the ball and can see the goal, it requires memory of the past actions. It has the same behavior as Krislet, except for the additional condition of checking whether the last action was *Dash* or not. While this behavior is not an improvement on Krislet, the test is determining how well the learning methods can reproduce this behavior.

A discretized version of RoboCup was created for testing purposes as well. It had the following three binary features: SeeBall, SeeGoal and BallClose. Extraneous observations such as flags and players are not used in this context.

Table 2.5 shows a sample of a trace file for the discretized version.

SeeGoal	BallClose	SeeBall	Action
1	0	0	Turn
0	0	0	Turn
1	0	1	Dash
1	1	1	Kick

Table 2.5: Sample trace (discretized RoboCup)

2.4 Converting Runs to State Machines

As mentioned previously, Woolridge proposed that every *standard* agent can be converted into an equivalent *state-based* agent and vice versa. The requirement for this to be possible is that the set R should contain all the possible runs of the agent. If this condition is not met, there are many state machines that can be built that will give rise to the runs in R. An example of this is shown below.

Consider the following run r_1 from a state-based expert in the discrete vacuum cleaner domain:

$$r_1 : 01010101 \xrightarrow{Up} 00010101 \xrightarrow{Down} 01010101 \xrightarrow{Down} 01010001 \xrightarrow{Up} 01010101 \xrightarrow{Up}$$

We can see in r_1 that given the environmental state 01010101, two differing actions, namely Up and $Down$, are observed (01010101 corresponds to perceiving a **wall(0) far(1)** in the NORTH, EAST, SOUTH and WEST direction). Therefore, it can be said that the latest environment state alone is not sufficient to explain the behavior. This is assuming that

- the agent's behavior is not stochastic,
- that we are capturing all the features,
- and that there is no noise in the observed run.

Therefore, the expert's memory (i.e., its internal state), which cannot be directly observed, must be a factor in the decision.

For the purposes of the following algorithms, a state machine will be described as a set of connections $G = \{G_1, G_2, ...G_n\}$ with the state space $\mathcal{G} = \mathcal{V} \times \mathcal{V} \times E \times Ac$, where $\mathcal{V}$ is the set of nodes and where $G_i = (v_i, v_j, e_t, a_t)$ is an instance that represents a directed edge from $v_i \in \mathcal{V}$ to $v_j \in \mathcal{V}$ where the edge has an associated environment state $e_t \in E$ and action $a_t \in Ac$.

Using the above definitions, algorithm 4 can be used to convert a run into a state machine:

Algorithm 4 SimpleRunToStateMachineConverter(Run):G

```
1:  Create CATCH node
2:  Create a START node
3:  Current ← START
4:  G ← ∅
5:  for all p = (e,a) ∈ Run do
6:      Create a NEW node
7:      Previous ← Current
8:      Current ← NEW
9:      G ← G ∪ (Previous, Current, e, a)
10:     G ← G ∪ (Previous, CATCH, ¬e, -)
11: return G
```

The algorithm runs as follows: Create a start state, and every time an action is performed, create a new state and transition from the current state to the new state. The algorithm accounts for the unseen environment states, by transitioning to a CATCH state where the agent will not perform any action (denoted by "-" in Line 10) The state machine built from r_1 using algorithm 4 is shown in Figure 2.5.

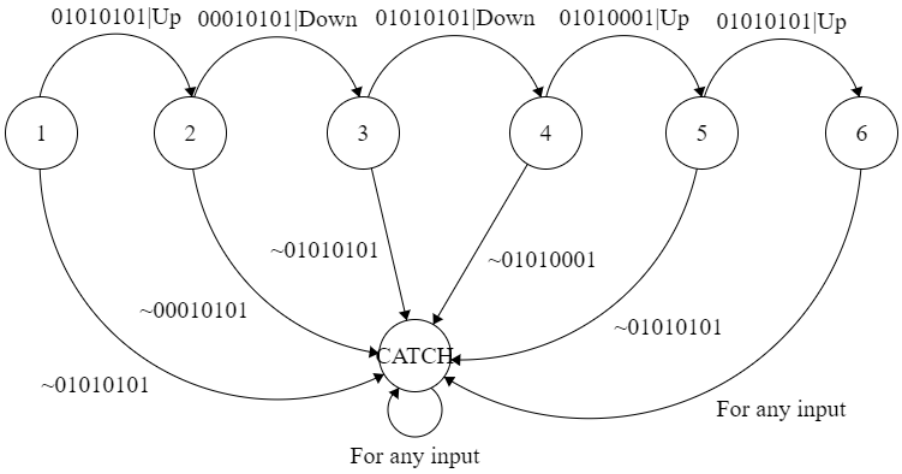


Figure 2.5: Simple state machine #1

Each numbered State has a transition to the next numbered state given that it follows the same input as the run. In order to handle events it may not have encountered a CATCH state has been included.

A more compact state machine is shown in Figure 2.6.The algorithm used to build this state machine utilizes the idea that the state transition occurs only when a different action has been performed for an environment state that has already been seen. When this occurs, a new state is created and the algorithm transitions to it. However, if there already exists a state in which that environment state-action pair is already performed, then the algorithm transitions to that state. Algorithm 5 describes this below:

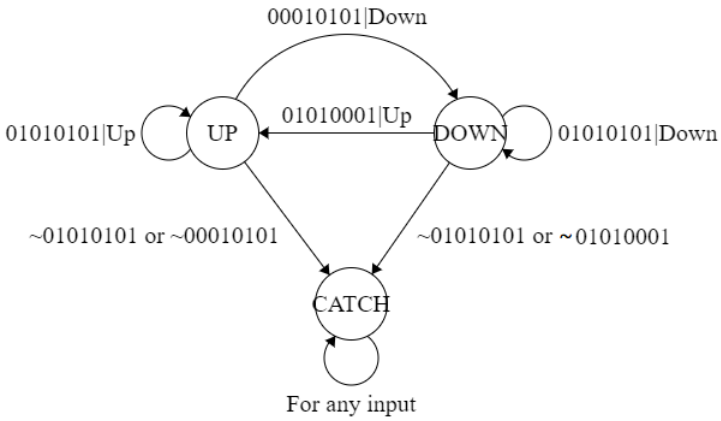


Figure 2.6: Compact state machine

State 'UP': the action performed is always Up unless a **wall(0)** is **near(0)** to the NORTH. State 'DOWN': the action performed is always $Down$, unless a **wall(0)** is **near(0)** to the SOUTH. There is a CATCH state to handle events, that may have not been seen yet.

Algorithm 5 RunToStateMachineConverter(Run):G

Create CATCH node
Create a START node
Current ← START
G ← ∅
Pool ← $emptylist$
for all p=(e,a) ∈ Run **do**
    if ∃ (e_p,a_p) ∈ Pool | e=e_p ∧ a ≠ a_p **then**
        Previous ← Current
        if ∃ ($v_{i,g}$, $v_{j,g}$, e_g, a_g) ∈ G | e=e_g ∧ a =a_g **then**
            Current ← $v_{i,g}$
        else
            Create a NEW node
            Current ← NEW
        PreviousP=(e_{pp}, a_{pp}) ← Pool[size(Pool)-1]
        G ← G ∪ (Previous, Current, e_{pp}, a_{pp})
        G ← G ∪ (Previous, CATCH, ¬e_{pp}, -)
        Pool ← Pool - PreviousP
        for all p = (e, a) ∈ Pool **do**
            G ← G ∪ (Previous, Previous, e, a)
            G ← G ∪ (Previous, CATCH, ¬ e, -)
        Pool ← $emptylist$
    Pool ← Pool + p
return G

Both state machines will generate the example run (r_1) (assuming the initial state is UP and 1). However, these state machines are not equivalent because they can generate runs that are not captured by each other. This is seen if we take a look at the next run, r_2:

$$r_2 : 01010101 \xrightarrow{Down} 01010001 \xrightarrow{Up} 01010101 \xrightarrow{Up} 00010101 \xrightarrow{Down} 01010101 \xrightarrow{Down}$$

The simple algorithm will generate a slightly different state machine as seen in Figure 2.7. However, the compact state machine will be the exact same, which means that the compact state machine can generate a run which is not captured by Figure 2.5.

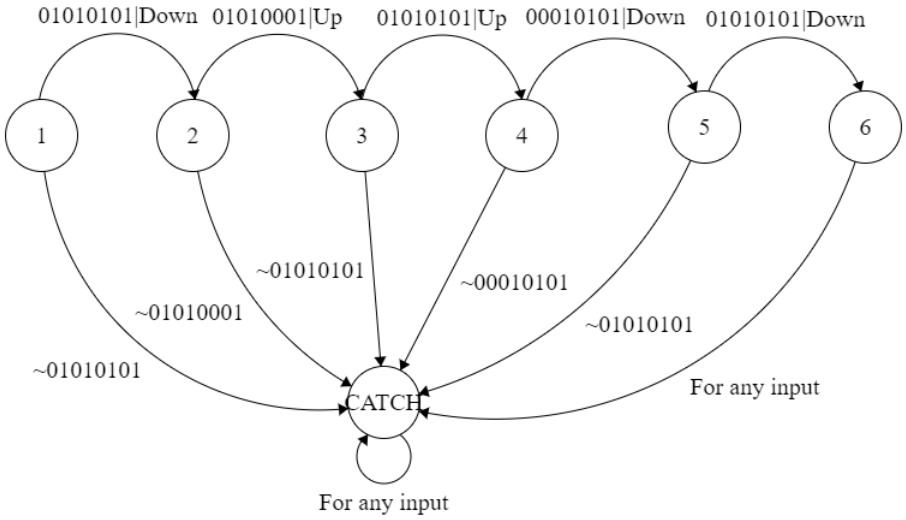


Figure 2.7: Simple state machine #2

Each numbered State has a transition to the next numbered state given that it follows the same input as the run. In order to handle events it may not have encountered a CATCH state has been included.

Looking at the state machines, it is difficult to say if any of the state machines are more "correct" in capturing the behavior because they are using a potentially incomplete set of data. If we had access to all the runs that could possibly be generated by the true state machine, then it may be possible to recreate the true state machine. The main issue with Algorithm 5 is that it has performed rote-learning on the data it has seen and has no way to make inferences on data that it hasn't seen (it doesn't create transitions for events that it hasn't seen). As described by

Mitchell [5], there is a need to introduce bias in order to generalize. In order for this algorithm to perform well, it would have to see a very large statistically significant portion of the dataset. This way, the resulting state machine has hope to have the correct answer for all the potential queries. In that case, there would be no need for machine learning. But in general, there is limited data and inductive leaps have to be made in order to predict the action for an environment state that hasn't been seen yet. This is the essence of machine learning.

2.5 Conclusion

In this Chapter we gave a description of *reactive* agent, *state-based* agents and the environments they are situated in. We described the domains used in this work for testing and validation. We also discussed the challenges of learning *state-based* behavior, which leads us to the next question: how does an agent learn the expert behavior it is observing? How does an agent capture the state machine that was used to generate the expert runs? The next Chapter presents a few state of the art approaches that have been used to start answering these questions.

Chapter 3

State of the Art

In this chapter we present two very different approaches that answer the questions asked in the previous chapter. How does an agent learn state-based behavior? One approach is probabilistic graphical models (PGM), and the other is case-based reasoning (CBR). These techniques initially were limited to learning *reactive* behavior. However, the authors wanted techniques that could also learn *state-based* behavior and therefore had to come up with extensions for the techniques that would allow this. Their contributions are the application of Dynamic Bayesian Networks [4] and the creation of the Temporal Backtracking algorithm [6]. We will present the progression of both techniques starting with their application to *reactive* behavior, then describe the limitations of the techniques, and then describe the solutions that the authors proposed.

We will first present the probabilistic graphical model approach to LfO.

3.1 Learning State-Based Behavior using PGMs

Graphical models are probabilistic models for which a graph provides the conditional dependency structure between random variables [15]. The random variables are depicted as nodes in the graphs and the relationships between the nodes are represented by edges.

3.1.1 Learning Behavior using PGMs

Ontañón et al. [4] presents a framework that uses PGMs to learn by observation. The paper characterizes different types of behavior such as strict imitation[1], reactive, state-based and stochastic. This thesis will focus on the first three types of behavior mentioned, two which have already been described in detail in chapter 2. The domain that was used to perform the testing and validation of the techniques was the discrete vacuum cleaner domain described in chapter 2.3.3. One downside of this framework is that it is domain specific and is tightly coupled to the testing domain.

The learning problem can be broken down into the following: X_t is a random vector that describes the environment state at time *t* and Y_t is a random variable that represents the action that the agent performs at time *t* (y_t and x_t will be used to denote specific values that Y_t and X_t can take). An agent's goal is to observe the (X_t,Y_t) pairs and imitate the behavior. The simplest PGM used to imitate behavior in this paper is the Bayesian network (BN).

[1] Strict imitation is independent of the environment state.

Bayesian Networks

A Bayesian network is compact representation of a probability distribution. The structure of the BN denotes an assumption that each node in the network is conditionally independent of its non-descendants given its parents[2]. Figure 3.1 shows that Y_t is only dependent on its parent node X_t. This approach means that each environment state-action pair (X_t,Y_t) can be treated independently. Since, there is no connection to past states, a BN with this type of structure can only learn *reactive* behavior. Also since there is is no dependence of time, X_t and Y_t can be collapsed down into X and Y.

Figure 3.1: Structure of Bayesian Network

X represent the environment state and Y represents the actions. There is no temporal connection between BNs at different times.

In the context of this thesis, each node will represent a discrete or continuous feature of the environment state X, or a discrete action Y. An example of this can be see in Figure 3.2 which depicts the structure used in the previously described vacuum cleaner domain. Once the conditional probabilities distributions (CPD) for $P(Y|X)$ and $P(X)$ are learned, the question that is asked is, what is the probability that the agent should take a specific action given some state

[2]A parent node is situated at the tail of a directed edge between two nodes

Environment State (X)	$P(Y = Left\|X)$	$P(Y = Right\|X)$	$P(Y = Up\|X)$	$P(Y = Down\|X)$
00010101	0.00	0.97	0.01	0.02
00110101	0.02	0.93	0.02	0.03
01000111	0.20	0.10	0.63	0.07

Table 3.1: Sample of the conditional probability table $P(Y|X)$ (vacuum cleaner domain)

of the environment? The agent performs a look-up for the specific environment state in the CPD $P(Y|X)$, and chooses the action with the highest probability ($argmax_y P(Y = y|X = x)$.

In this paper, the BN was used to learn the behavior of the WallFollower agent described in the chapter 2. The BN was able to learn this behavior and reproduce it with 92% accuracy.

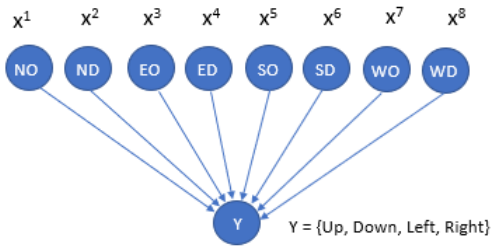


Figure 3.2: Bayesian Network: vacuum cleaner domain

$x^1, x^2, x^3, x^4, x^5, x^6, x^7, x^8$ represent the 8 features of the environment state X. (NO represents North Object, where Object is *dirt* or *wall*, and ND represents North Distance where distance is *near* or *far*. EO represents East Object, ED represents East Distance, SO represents South Object, SD represents South Distance, and WO represents West Object, WD represents West Distance) and Y represents the Actions.

A sample of the learned conditional probabilities is shown in Table 3.1. There are 256 (2^8) rows of probabilities because of the 8 binary features of the environment.

Given the environment state $x = \{00010101\}$ the action that the agent should

perform is $Right$ because it has probability of 0.97 versus the probability of 0.0 for $Left$, 0.01 for Up and 0.02 for $Down$. This is a simple example of how a trained BN is used to imitate *reactive* expert behavior in this paper.

If we want to learn state-based behavior, there has to be a temporal link that connects the BNs at each time slice. Therefore, the solution presented in the paper is a specific type of Dynamic Bayesian Network (DBN) which is described below.

Dynamic Bayesian Networks

A DBN is a BN that is replicated through time. At each time step, a BN is created and the probabilities are learned and then it is connected to the BN at the next time step and so on. This type of structure captures the temporal dependency between time steps, and therefore is able to learn state-based behavior.

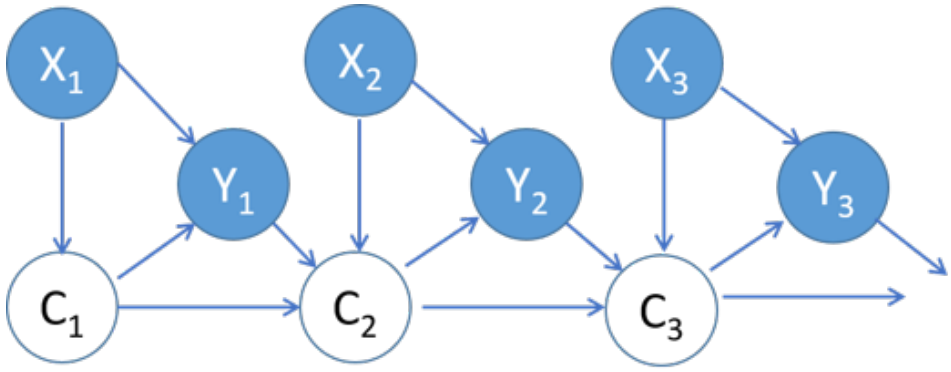


Figure 3.3: Structure of Dynamic Bayesian Network

X_t is the environment state and Y_t is the Action and C_t is the hidden internal state.

The most important characteristic of the DBN that allows it to learn state-based behavior is the hidden internal state node C_t, as shown in Figure 3.3. This node is

responsible for capturing the salient information from the past environment state-action pairs and propagating it forward as time goes on. This fits the definition of the state-based agent described in chapter 2.2.

One of the properties used here is the local Markov property [16], which states that, when conditioned on its neighbors, X_t is independent of all other variables in the graph. Given the structure shown in Figure 3.3, the only information required to perform inference at time step *t* is the information at *t* and *t-1*.

In all of the domains used in this thesis, the DBN model is temporally homogeneous and can be represented as a two-time-slice model instead of an DBN of length *T*. This is shown in Figure 3.4. Once the conditional probabilities which are $P(C_t|X_t)$, $P(C_t|Y_{t-1}, C_{t-1}, X_t)$ and $P(Y_t|C_t, X_t)$ are learned, inference can be performed.

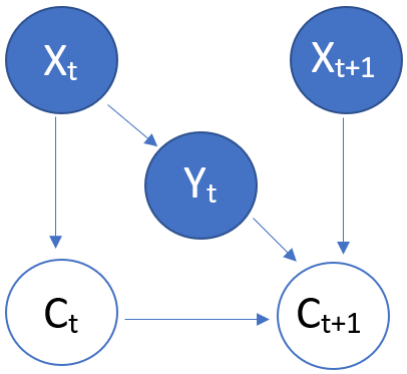


Figure 3.4: Structure of two-time-slice DBN

Dynamic Bayesian Network: X_t is the environment state and Y_t is the Action and C_t is the hidden internal state.

The DBN was used to learn the behavior of the ZigZag Agent, in the vacuum cleaner domain (recall the definition of the ZigZag agent from chapter 2). The

behavior is state-based because the agent has to keep a record of whether it is going left or right, and if it is going up or down. The DBN is able to learn and reproduce this behavior with 83% accuracy.

One of the pitfalls of this work is that accuracy is the only measure reported, but accuracy is not the best measure of performance when datasets are imbalanced. Ideally, F1 score would be used for a more complete picture of the performance. Another pitfall, as mentioned in chapter 1, is that the implementations are built specifically to cater to discrete input variables X_t, and have to be restructured in order to function in the continuous domain. Therefore, there is no indication of the performance of the BN and DBN in a more complex environment made up of continuous features. It is also tightly coupled to the vacuum cleaner domain, which doesn't allow for testing these techniques in other domains and comparing them to other techniques such as case-based reasoning.

In the next section, we present the other method used for performing the learning aspect of LfO: case-based reasoning.

3.2 Case-Based Reasoning

Case-based reasoning in general uses past experiences to understand and solve new problems [17]. It does this by comparing the similarities between the past problem-solution pairs and the current problem, and choosing the most similar pair. The CBR cycle is as follows: Retrieve, Reuse, Revise (Adapt) and Retain [18].

- Retrieve: select the most similar problem-solution pairs from memory.
- Reuse: select the best solution from the retrieved pairs.
- Revise (Adapt): iteratively adapt and evaluate the solution until some condition is met.
- Store: place the new problem-solution pair into memory for future use.

In the context of LfO, the CBR cycle consists only of **Retrieval** and **Reuse**. The **Revise** step requires a form of teacher feedback or reinforcement, which is out of scope for LfO. Since there is no feedback, there aren't any new problem-solutions to **Store** for future use.

3.2.1 Reactive Retrieval

When applying case-based reasoning to LfO, a case $c \in C$ is an instance from the set of environment state-action pair $C = E \times Ac$ (Recall that E represents the set of environment states, and that Ac represented the set of actions). When a case-based reasoning agent receives an environment state as a query, it searches through the casebase, and returns the action from the case that has an environment state that is most similar to the query. This idea of similarity can also be extended to the action as well, which leads to the general version of the similarity function which compares the similarity between two *elements*, where $Element = E \cup Ac$. The similarity function used to compare two elements can be defined in the following

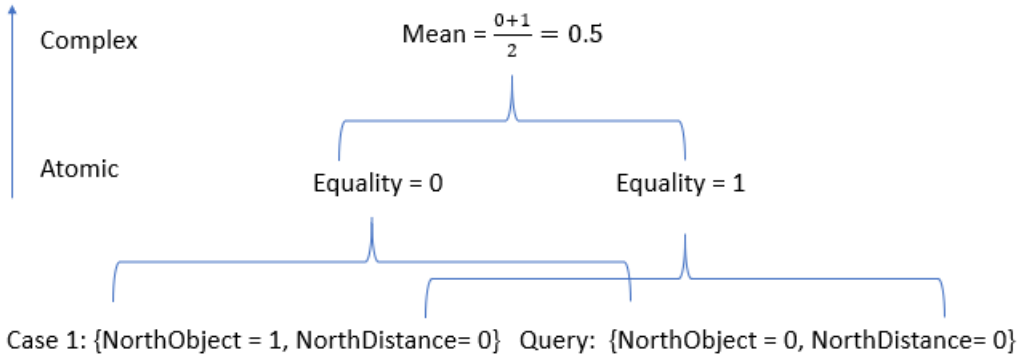


Figure 3.5: Example similarity calculation (Reactive)

Example of similarity calculation between a query and a case in a subset of the vacuum cleaner domain using *Atomic* and *Complex* similarity metrics.

manner:

$$Sim : Element \times Element \rightarrow [0, 1]$$

It may be the case that an environment state contains many features which are elements, where the environment state itself is an element. In this situation, a leaf element is called an *Atomic* element, and an element that contains other elements is called a *Complex* element. Therefore, there are similarity metrics that compare *Atomic* elements and *Complex* elements. A representation of this using the subset of the vacuum cleaner domain is shown in Figure 3.5. The *Atomic* similarity metric used is *equality* which returns 1 if the elements are the same and 0 if not, and the *Complex* similarity metric used is *Mean* which aggregates the similarities of all of its subelements.

The retrieval method considers each case independently from each other. Since there is no temporal link between the cases, this retrieval method is not conducive

to learning *state-based* behavior.

3.2.2 State-Based CBR - TB algorithm

In order to extend CBR to learn *state-based* behavior, Floyd [6] proposed we redefine the case C to contain all the past history until that point in time:

$$C = R^E \times Ac \tag{3.1}$$

Now, each case is temporally linked with its past (recall from chapter 2.2, R^E represents the subset of *R* that contains runs that end in an environment state).

If for the given environment state there are many differing actions suggested by the cases in the casebase, the Temporal Backtracking algorithm (TB) [6] considers that the reason for the discrepancy is situated somewhere earlier in the run, and so it dynamically backtracks through the runs until it has found consensus on the action it should perform. The inductive bias of TB is thus to give more weight to recent events and actions. In the paper, the TB algorithm was used to learn the behavior of an agent that toggles whether it turns left or right each time it hits an obstacle. The results showed that TB was able to outperform the reactive retrieval.

During run time, when the algorithm compares an element in the query with an element in the run, it decides to keep the run in its set of candidate runs only if the compared element's similarity exceeds some threshold. This threshold is preset by the user. There are three different thresholds - Recent Problem Threshold (RPT), Past Problem Threshold (PPT) and Solution Threshold (ST). ST is the

threshold used when comparing actions, RPT is the threshold used when comparing the environment state at the current time and PPT is the threshold used when comparing environment states in the past. There was some concern expressed in the original work that there is work to be done in determining the best possible way to set these parameters.

Let's look at an example which uses TB (ST=PPT=RPT=0.99) and reactive retrieval to predict the behavior of the Left-Right Toggle agent in the previously described obstacle avoidance domain where the environment state has one continuous feature - Sonar (Section 2.3.3):

$$time : t-6 \xrightarrow{t-5} t-4 \xrightarrow{t-3} t-2 \xrightarrow{t-1} t \xrightarrow{t+1}$$

$$Case1 : 3.94 \xrightarrow{Forward} 2.4 \xrightarrow{Right} 3.7 \xrightarrow{Forward} 2.78 \xrightarrow{Left}$$

$$Case2 : 3.56 \xrightarrow{Forward} 2.1 \xrightarrow{Left} 3.9 \xrightarrow{Forward} 2.65 \xrightarrow{Right}$$

$$Query : 3.65 \xrightarrow{Forward} 2.23 \xrightarrow{Right} 3.8 \xrightarrow{Forward} 2.70 \xrightarrow{?}$$

If we set *k*=3 for the kNN reactive retrieval algorithm, it will have three actions to choose from. Since kNN looks at the *k* most similar environment states to the current environment state of $\{2.7, ?\}$, it will choose $\{2.4, Right\}$, $\{2.78, Left\}$ and $\{2.65, Right\}$. The action chosen will be $Right$ because it is the majority action at $\frac{2}{3}$. TB on the other hand, realizes that there is no consensus between the actions and moves backwards in time, comparing the actions, then the environment-state until there is consensus. This occurs at *t-3*, where Case 1 performs $Right$

for the sonar value of 2.4, and Case 2 performs $Left$ for the sonar value of 2.1. The Query performs $Right$ as well. The TB algorithm takes this as a sign that the most similar Case is Case 1, and therefore predicts the action to be $Left$ (the most recent action from the most similar Case). Since, we already know that the behavior that generated these cases is a Left-Right Toggle agent (we have access to the true state machine that generated the run) we can confirm that this is the expected action.

The framework that was used to perform the experiments above is called Java Learning from Observation Framework (jLOAF) and was created to perform general purpose LfO using CBR [7]. The framework in its current form has functionality to perform reactive retrieval in any problem space. The framework is domain independent and can be plugged in as a brain into any agent that is situated in some environment. This framework provides the building block to create a *state-based* LfO framework which is domain independent, has multiple learning techniques and can learn generalized behavior, be it *state-based* or otherwise.

3.3 Extensions to State-Based Learning using CBR

There was preliminary work done in extending CBR to allow for *state-based* learning by [8]. Jaccard and Edit distances were used to calculate the similarity between runs. They were used in the vacuum cleaner domain and compared to the TB algorithm. In the case of Jaccard distance, the two runs being compared are converted into sets and then the compared based on the number of unique ele-

ments the two sets share. The implicit bias that exists in Jaccard is that the count of unique elements is not important. Edit distance is applied between runs, and determines how many changes have to be made to make the cases equal. The Edit distance has a bias that gives equal weight to each case in the time window that it is comparing. The Jaccard and Edit distance did not outperform the TB algorithm in learning behaviors tested in the vacuum cleaner domain. Another area that the author investigated was reproducing the results of Ontañón's work [4] using the PGM framework to calculate performance measures with mean and standard deviation. In this process, it was discovered that the DBN learner was performing poorly when learning *state-based* behavior and the results of the original paper were not reproducible. One of the possible issues that was considered was that some variation or bug was introduced into the code during testing. Understanding why this occurred is important in order to compare the DBN approach to the TB approach when learning *state-based* behavior. This will be discussed in Chapter 4.

3.4 Time Complexity

One aspect of the CBR and PGM techniques that we don't delve into in this thesis is the training and testing time. However, for the sake of completeness, we discuss it briefly here. For CBR there is no training time because once the cases are gathered, the agent is ready to be tested or deployed. However, during runtime, the agent may have to compare the query to every single case in the case base.

This can be an issue if the casebase is large, and the agent has to make decisions quickly. Conversely, the PGM approach can have a long training time. However, during runtime, the agent has to perform inference calculations using data from the learned conditional probability tables which is generally very quick. Therefore, the time component can be an important factor when choosing which technique to use.

3.5 Conclusion

As can be seen above, there are two approaches that are able to learn *state-based* behavior in specific domains. In the case of the PGM framework, the performance measures presented are not broad enough and the techniques are tied specifically to the discrete vacuum cleaner domain. In the case of jLOAF, there is no support for PGM techniques, and it has currently only been tested in a few domains. It also only has TB as a technique to learn *state-based* behavior. This leads us back to questions that were asked in the Introduction: is there a way to determine which technique is better? Will some techniques work better in certain domains and worse in others? Is there a way to choose the best technique based on the type of behavior? Can we apply them against each other across multiple domains to determine how they perform? The techniques are currently applied in two different frameworks, with different benchmarks and it is difficult to say if one is better than the other. If we can identify one as being objectively better, then we can focus on making improvements for that technique.

For this to occur, the techniques need to be unified under one framework, with the same benchmarks, which have access to the same domains. There is also a need to have a proper methodology for testing *state-based* behavior. Once this methodology is implemented, we can focus on identifying the areas of improvement by performing intermediary testing. The next chapter presents the proposed design methodology for testing *state-based* behavior, the results of using TB, BN and DBN to learn behavior in the discrete vacuum cleaner and discrete RoboCup domains, and the proposed areas for improvement in the CBR and PGM techniques.

Chapter 4

Testing State-Based Learning: A Design Methodology

In order to answer the questions asked in the previous chapter, there was a need to have a unified flexible framework where the learning techniques could be plugged in and out, tested in different domains, and could be compared using common performance metrics. The two frameworks that existed were the PGM framework which was tightly coupled to the discrete vacuum cleaner domain, and jLOAF which was set up only to perform reactive retrieval but was not tied to a specific domain. The unified framework was created by

- implementing the PGM learning methods from the PGM framework in jLOAF,
- and implementing the redefined case structure which allowed for the implementation of the TB algorithm, and potentially opened up the door for building more state-based CBR techniques.

Once the framework was unified, one of the next steps that was accomplished was to identify the reason for the poor performance of the DBN in learning state-based behavior in the vacuum cleaner domain in [8]. We realized that the DBN learner would store the memory of the action it last performed, and use it to predict the action at the next time step. The issue is, if the action stored was an incorrect action, the DBN learner would end up on a different trajectory than the expert and comparing any actions between the expert and DBN learner from that point onwards would be meaningless. Once this issue was solved, the results improved dramatically and matched those of the original paper [4]. The results can be seen in [9]. Section 4.1 presents a visual representation of why storing an incorrect action is an issue when testing state-based behavior, how it was solved, and the resulting contribution which was the development of a testing framework for state-based behavior.

Section 4.2 presents comparative results of applying TB, BN and DBN to learning *reactive* and *state-based* behavior in the vacuum cleaner domain, and the discrete RoboCup domain. We found that the TB algorithm was able to perform as well as the DBN and BN in learning behavior in both domains. This validated the need to benchmark the techniques against each other in a wider variety of domains. The results provided insight into the areas for improvement in both the CBR-based and PGM-based learning - specifically introducing bias into the Bayesian network and using the notion of bias to create new similarity metrics that can be used to perform state-based retrieval.

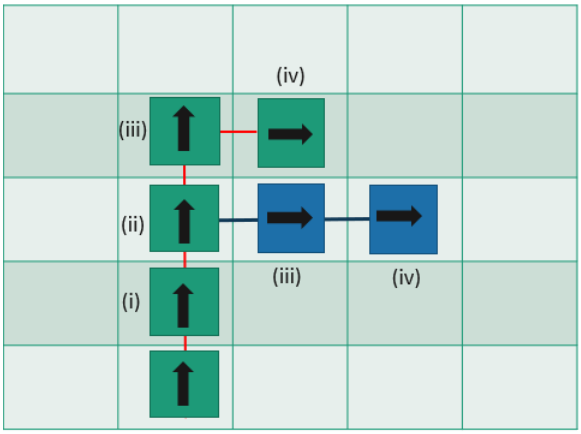
4.1 Towards a Framework for Testing State-Based Behavior

Recall the definition of the reactive agent:

$$Ag_{Reactive} : E \rightarrow Ac$$

Since it is a function mapping from the environment state to the action, there is only one action per environment state. So when testing the performance of the learning agent, one can simply provide an environment state-action pair without any of the past information as the query and compare the prediction to the query's action. This is the procedure used in regular classification.

However, there is more complexity when testing *state-based* behavior. Since the behavior relies on past actions and environment states, there may be multiple actions that can be performed given the current environment state, as we saw in the TB example in the previous chapter. One cannot ask the expert what the correct action is unless the internal state of the learning agent and the expert are the same, which means we must ensure that the expert and the learning agent are on the same trajectory. This means that they should experience the exact same environment states in sequence, and that they should perform the same past actions so that their current internal states are the exact same. An example of the learning agent diverging from the trajectory of the expert is shown in Figure 4.1. It shows why the divergence can cause unreliable test results.



Time	Expert	Learner	Result
i	UP	UP	Correct
ii	UP	UP	Correct
iii	UP	RIGHT	Wrong
iv	RIGHT	RIGHT	Correct

Figure 4.1: Testing state-based learning

Trajectory of a state-based expert (Green) and a state-based learner (Blue). The time of each event is depicted by (*i*). Record of actions performed by expert and learning agent

In Figure 4.1, the time (*i*) and (*ii*) show the learning agent predicting actions which are compared to the expert actions. Up till this point, the learning agent and the expert are on the same trajectory. Time (*iii*) shows the expert and agent after their third move. The agent has predicted the wrong action, and they are **not** on the same trajectory. In this situation, the internal state of the expert and the agent is different because the action stored by the agent is $Right$ and the action stored by the expert is Up. Time (*iv*) shows the expert and agent after their fourth move. The agent has predicted the same action as the expert, however, they are still **not** on the same trajectory. Therefore, comparing the actions of the agent and expert in this situation of divergence has no meaning (even though the predicted action and the expert action are the same). The expert and agent should always be on the same path if the predicted actions are to be compared with the expert actions.

To prevent this divergence from occurring, when the learning agent predicts

an action, it should be compared with the expert action and stored separately for calculating performance measures. The action that is stored in the learning agent's memory and passed to the environment should be that of the expert's. The learning agent should not interact with the environment except to receive the run R^E which contains all the environment state-action pairs until the time *t-1* and the environment state at time *t*, which is all the information necessary to choose its next action. This encapsulation will ensure that the learning agent does not diverge from the trajectory. The design for this testing methodology can be seen in Figure 4.2.

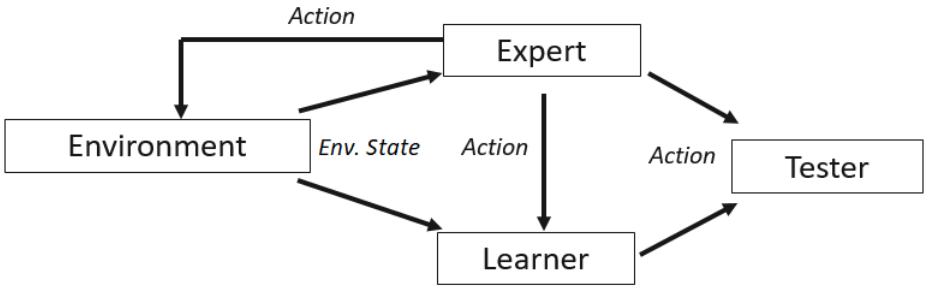


Figure 4.2: The framework for performance testing state-based learning in an environment

4.1.1 Performance Testing

In the context of this work, performance testing does not require connecting the agent to the environment. Instead, the expert behavior is generated beforehand in the form of a trace file which contains the environment state-action pairs. These traces are then separated into training and testing sets during "leave one out" test-

ing. During testing, at each time step, all the environment state-action pairs until the current time step are passed to the learning agent (runs from R^{Ac}, which is the subset of R with runs that end in an action), and the learning agent has to predict an action. The predicted action is stored for performance calculation, along with the most recent action from the $r \in R^{Ac}$ and the cycle continues. The predicted action does not in any way affect the environment. Therefore, the learning agent will always receive the environment state that the expert is in.

The two main state-based learning methods, DBN and TB, can be matched to the state-based agent and the standard agent respectively (described in chapter 2). The state-based agent has some internal state which records the salient past information, and the standard agent potentially looks at the whole history to infer the internal state. Similarly, the DBN learner has to remember the past action (which is the most recent action from $r \in R^{Ac}$), and keep track of its internal state, and the TB learner will potentially backtrack through the whole history to infer the internal state.

The framework handles this by passing $r \in R^{Ac}$ to both the DBN and TB learners. The TB learner will use the entire run, whereas the DBN learner will only use the most recent case, because it will remember the last expert action and have some internal state. Therefore, a slight modification is performed to the framework design and is shown in Figure 4.3.

Once this was implemented, the DBN and TB learners were tested, and the results are shown in the next section.

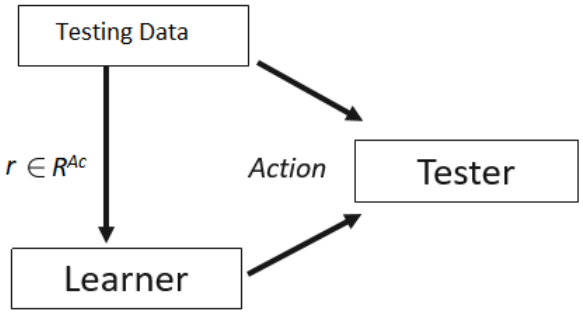


Figure 4.3: The framework for performance testing state-based learning without an environment

4.2 PGM vs CBR: Discrete Domain

Once the design was used to solve the DBN learner's issue, the results of learning in the vacuum cleaner domain were reproduced. A determination of how TB would fare against the PGM models was made. Table 4.1 show TB performing at least as well as the PGM methods across all types of behavior - *reactive* and *state-based*.

Agent	BN	DBN	TB
WF	0.92 ± 0.15	0.92 ± 0.16	1.00 ± 0.01
SWF	0.94 ± 0.15	0.94 ± 0.14	1.00 ± 0.01
ZZ	0.47 ± 0.02	0.91 ± 0.07	0.94 ± 0.04
SEQ	0.40 ± 0.12	0.51 ± 0.24	0.66 ± 0.32

Table 4.1: Reproduced accuracy results (vacuum cleaner domain)

The two techniques were also applied to the previously described discretized RoboCup domain (chapter 2). At the time, the PGM techniques were not able to handle continuous inputs, which is why the domain was discretized. The traces of

10 Krislet agents playing 5 vs 5, as well as 10 Krisletiflastdashed agents playing 5 vs 5 were acquired. 10-Fold cross validation was performed and the results are shown in Tables 4.2, 4.3 and 4.4.

Learner	Krislet	Krislet-KickIfLastDashed
BN	0.79 ± 0.02	0.77 ± 0.03
DBN	0.71 ± 0.01	0.68 ± 0.04
TB	0.79 ± 0.12	0.72 ± 0.04

Table 4.2: Accuracy of learned soccer playing agent behavior (RoboCup domain)

Action	Turn	Dash	Kick
BN	0.20 ± 0.01	0.88 ± 0.01	0.57 ± 0.06
DBN	0.39 ± 0.01	0.81 ± 0.01	0.60 ± 0.07
TB	0.56 ± 0.25	0.86 ± 0.08	0.86 ± 0.08

Table 4.3: F1 scores of learned Krislet behavior (RoboCup domain)

Action	Turn	Dash	Kick
BN	0.38 ± 0.08	0.86 ± 0.02	0.00 ± 0.00
DBN	0.47 ± 0.06	0.77 ± 0.04	0.56 ± 0.16
TB	0.53 ± 0.04	0.80 ± 0.04	0.74 ± 0.09

Table 4.4: F1 scores of learned Krislet-KickIfLastDash behavior (RoboCup domain)

The TB algorithm was able to perform similarly to the BN and DBN learners in learning the Krislet and KrisletIfLastDash behavior.

4.3 Conclusion

Looking at these preliminary results, there are three areas that are worth investigating further.

- Firstly, we noticed that when learning was performed using Ontañón's Bayesian network, the learning was generating the conditional probability tables (CPT) for $P(Y|X)$ directly. If X is a large vector, the generated CPT is going to be very large[1], and it is likely that the dataset does not contain every possible combination from the set of environment states. If inference is to be performed on an unseen environment state, the probability of each action will be evenly distributed, which doesn't provide any information. Even if a particular environment state is seen once, it is not statistically stable and has a high variance. Traditionally in machine learning, in order to predict actions given an environment state that hasn't been seen, an inductive leap (bias) has to be made. This isn't the case in the method that is used in Ontañón's paper. In the next chapter, we propose using the Naive Bayes Classifier in order to introduce bias into the model.

- Secondly, the results shown above indicated that the TB algorithm is comparative to the DBN and BN at learning different types of behavior across multiple domains. As mentioned in the previous points, a proposed improvement to the BN learning method is to introduce bias. TB already has a particular type of bias - a recency bias which gives more weight to recent elements in the run. Are there cases where this bias will cause TB to perform poorly? Counter-examples that show TB is not always the best algorithm to use are introduced and new similarity metrics paired with kNN to perform state-based retrieval and capture other inductive biases are proposed in the

[1] 8 binary features is 2^8 = 256 Rows

next chapter.

- Finally, there is a need to characterize how the performance of the PGM techniques compare with the CBR techniques in the fully and partially observable continuous domains. This was a question posed in C. Chan's [8] thesis and we endeavor to answer it in chapter 6, through the Cartpole V0, obstacle avoidance and RoboCup domains.

Chapter 5

Contributions to LfO I: Bias

In [5], the author describes an *unbiased generalization* as "If generalization is the problem of guessing the class of instances to which the positive training instances belong, then an unbiased generalizer is one that makes no a priori assumptions about which classes of instances are most likely, but bases all its choices on the observed data." The author then goes on to describe the pitfalls of using an unbiased generalizer with the following statement "An unbiased learning systems ability to classify new instances is no better than if it simply stored all the training instances and performed a lookup when asked to classify a subsequent instance." The conclusion the author comes to is that in order to perform an *inductive leap* and classify new instances, the generalization system must have some bias.

The PGM-based approach and CBR-based approach are two ways in which the generalization is being performed. As mentioned in the previous chapter, both of these techniques have areas in which bias can be introduced. Keeping this in

mind, this chapter is divided in the following manner:

- Section 5.1 proposes a change to the Bayesian Network model presented in [4], which was used to learn reactive WallFollower behavior in the discrete vacuum cleaner domain. The changes consist of of changing the direction of the directed edges to resemble the structure of a Bayesian Classifier, and thus introducing the Naive Bayes assumption. Preliminary results of learning the behavior on the original maps used by Ontañón don't show a significant difference between the two models, however, this is due to the maps having square dimensions. We designed specific maps in which the expert had to travel in an asymmetric manner, and showed that the Naive Bayes classifier outperforms Ontañón's proposed model.

- In most of the testing, the TB algorithm has performed similarly to DBN in learning state-based behavior. It seems to be able to handle different types of domains with a wide variety of behavior. As mentioned in the previous chapter, the TB algorithm has a particular type of bias. Section 5.2 presents examples that show that there are cases where TB does not perform well, and that there may be state-based retrieval methods that can outperform it. The section presents new similarity metrics that capture different biases and are pitted against TB to learn handcrafted behavior that causes TB to perform poorly.

5.1 Introducing Bias into Ontañón's Bayesian Network Model

5.1.1 Problem Statement

This section will address the issues with Ontañón's model, which was used to learn the WallFollower (WF) behavior in the vacuum cleaner domain [4]. We will discuss the model's limitations and then show how the application of the Naive Bayes Classifier can improve the performance, illustrated using an example. Finally, we test the two models against each other in learning WF behavior in specifically designed maps and show that the Naive Bayes approach is better at learning the behavior.

5.1.2 Ontañón's Model

The BN model that Ontañón proposed is shown in Figure 5.1.

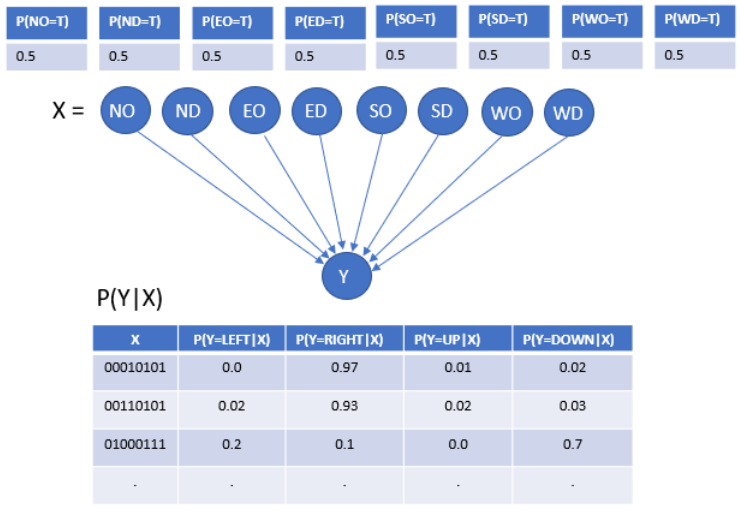


Figure 5.1: Ontañón's proposed model (vacuum cleaner domain)

NO, ND, EO, ED, SO, SD, WO, WD represent the 8 features of the environment state X. (NO represents North Object, where Object is *dirt* or *wall*, and ND represents North Distance where distance is *near* or *far*. EO represents East Object, ED represents East Distance, SO represents South Object, SD represents South Distance, and WO represents West Object, WD represents West Distance) and Y represents the Actions.

The model depicts the conditional dependencies between the environment state nodes and the action node, as well as the conditional probability tables. The problem with this model is that the table of $P(Y|X)$ has 256 (2^8) rows, most of which are not observed. When a query is made to this model, it looks up the probabilities of each action in the row corresponding to the environment state of the query, and chooses the action with the highest probability. If the environment state hasn't been seen before, each action has an equal probability of being chosen, and if it has been seen, there is not enough data to make it stable and reliable as there is too much possible variance. Therefore, in this model there is no inductive leap being made when dealing with queries with unseen or rarely seen environment

states. This model can be described as an *unbiased generalizer*. The only way for this model to perform well is for it to have seen a very large subset of the dataset that it is being trained and tested on, which generally isn't the case in machine learning problems.

In the next section, we present the Naive Bayes Classifier as an alternative model that solves the issues described with the model Ontañón proposed.

5.1.3 Alternative Model: Naive Bayes Classifier

Bayesian classification, as described in [19], is when a Bayesian network is applied to a classification task. This is used extensively in filtering spam. The structure of the network used is that the class node is the parent of the input nodes. In the context of LfO, the Action node would be the parent of the environment state nodes. This model can be seen in Figure 5.2.

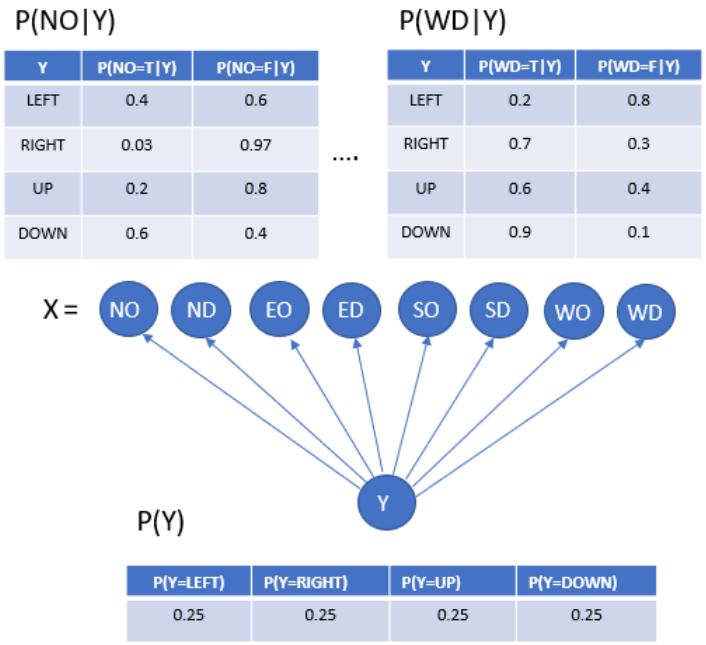


Figure 5.2: Naive Bayes Classifier (vacuum cleaner domain)

NO, ND, EO, ED, SO, SD, WO, WD represent the 8 features of the environment state X. (NO represents North Object, where Object is *dirt* or *wall*, and ND represents North Distance where distance is *near* or *far*. EO represents East Object, ED represents East Distance, SO represents South Object, SD represents South Distance, and WO represents West Object, WD represents West Distance) and Y represents the Actions.

Note that the directed edges in this Bayesian classifier are the opposite of the ones in Ontañón's proposed model (Figure 5.1). This model results in learning smaller CPT's than Ontañón's model. and during inference given some x, the network allows us to compute the $P(Y|X)$ using the Bayes Theorem (as opposed to looking up a CPT):

$$P(Y = y|X = x) = \frac{P(X = x|Y = y)P(Y = y)}{P(X = x)}$$

Since in the Naive Bayes assumption each feature $x^i \in X$ is conditionally independent of each other given the action node Y.

We have

$$P(X = x|Y = y) = \prod_i P(X^i = x^i|Y = y)$$

where i represents the index of the individual features in the environment state vector X.

The introduction of bias occurs in the Naive Bayes classifier due to the strong independence assumptions between the features of the environment and the change in the direction of the directed edges, which creates a completely different model to that proposed by Ontanon. The bias that is introduced allows the model to perform an inductive leap when choosing the action for an unseen environment state.

An example using the discrete RoboCup domain is shown below to illustrate one of the pitfalls of Ontañón's approach - the inability to generalize due to the limited bias.

5.1.4 An Example

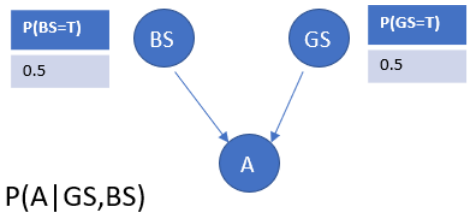
An example of performing inference is shown below for the original model. Consider the problem space here to be a subset of the RoboCup domain, where the Actions available to the agents are *Turn* and *Dash*, and the environment state has 2 feature - *BALLSEEN* (BS) and *GOALSEEN* (GS). Table 5.1 contains the training data that will be used to populate the CPTs of the models.

BALLSEEN	*GOALSEEN*	*ACTION*
T	F	Turn
T	T	Dash
F	T	Turn
T	F	Dash

Table 5.1: Training data (subset of a RoboCup trace)

Features - *BALLSEEN* and *GOALSEEN*, Actions - *Turn* and *Dash*

Applying Ontañón's proposed model to the RoboCup domain yields the model and CPTs shown in Figure 5.3:



P(A|GS,BS)

BS	GS	P(A=Turn|BS, GS)	P(A = Dash |BS, GS)
F	F		
F	T	1	0
T	F	0.5	0.5
T	T	0	1

Figure 5.3: Ontañón's proposed model (RoboCup domain)

BS and GS represent the 2 features of the environment state X. Node A represents the Actions. The first row of the CPT of (P|A, BS, GS) is empty because that event was not seen in the dataset.

The probabilities for P(*ACTION*|*BALLSEEN*=F,*GOALSEEN*=F) are empty because that event wasn't observed in the training dataset. In this case when inference is being performed, there is an equal probability of choosing each action - P(*ACTION*=*Turn*|*BALLSEEN*=F,*GOALSEEN*=F) = 0.5.

The model and CPTs for the Naive Bayes Classifier can be seen in Figure 5.4:

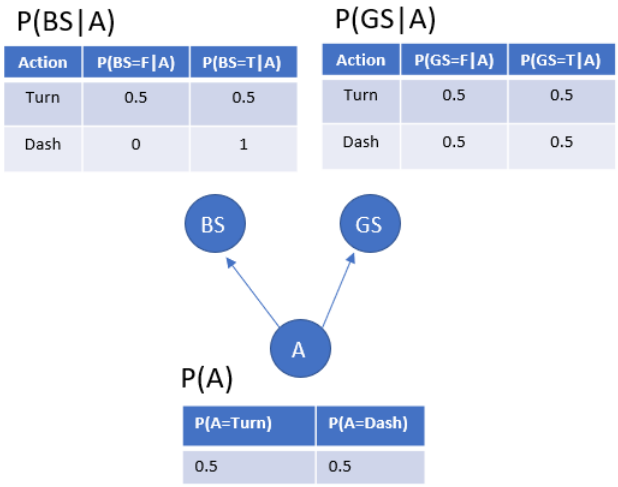


Figure 5.4: Naive Bayes Classifier (RoboCup domain)

BS and GS represent the 2 features of the environment state X. Node A represents the Actions.

If the previous query of P(*ACTION*|*BALLSEEN*=F,*GOALSEEN*=F) is made to the Naive Bayes Classifier, the following steps will be taken:

Apply Bayes Law:

$$P(A = Turn|BS = F, GS = F) = \frac{P(BS = F, GS = F|A = Turn)P(A = Turn)}{P(BS = F, GS = F)}$$

Apply Naive Bayes assumption:

$$... = \frac{P(BS = F|A = Turn)P(GS = F|A = Turn)P(A = Turn)}{P(BS = F, GS = F)}$$

Substitute probabilities from tables:

$$P(A = Turn|BS = F, GS = F) = \frac{0.5 \times 0.5 \times 0.5}{P(BS = F, GS = F)} = \frac{0.125}{P(BS = F, GS = F)}$$

Apply Bayes Law:

$$P(A = Dash|BS = F, GS = F) = \frac{P(BS = F, GS = F|A = Dash)P(A = Dash)}{P(BS = F, GS = F)}$$

Apply Naive Bayes assumption:

$$... = \frac{P(BS = F|A = Dash)P(GS = F|A = Dash)P(A = Dash)}{P(BS = F, GS = F)}$$

Substitute probabilities from tables:

$$P(A = Dash|BS = F, GS = F) = \frac{0 \times 0.5 \times 0.5}{P(BS = F, GS = F)} = \frac{0}{P(BS = F, GS = F)}$$

Since the denominators are the same in both cases, they do not need to be calculated. The Action chosen is *Turn* because $P(A = Turn|BS = F, GS = F) > P(A = Dash|BS = F, GS = F)$. This behavior corresponds to the behavior of the Krislet agent, where it will turn if it cannot see the ball or see the goal.

This shows Ontañón's proposed models inability to generalize on new unseen environment states.

5.1.5 Hypothesis

The hypothesis is that the Naive Bayes Classifier will outperform Ontañón's proposed model in learning the WallFollower agent behavior in the discrete vacuum cleaner domain. This is due to the Naive Bayes Classifier's ability to perform an inductive leap when dealing with new unseen environment states. Preliminary testing was performed using the original 7 maps that Ontañón had used and the results are shown in Table 5.2:

Model	Accuracy	Average F1 Score
Ontañón's Model	0.92 ± 0.14	0.88 ± 0.25
Naive Bayes Classifier	0.98 ± 0.03	0.97 ± 0.03

Table 5.2: Preliminary results in testing the use of Naive Bayes Assumption during inference

The results are for the WallFollower agent. The maps used are the same maps used in the original paper.

Upon further inspection the maps used were symmetrical in dimension, and the behavior of the WallFollower Agent in all the maps was to travel in a square. This would explain the similar results from both models. In order to determine if true WallFollower behavior can be captured by both models, the maps were designed so that the movement of the agent was asymmetric. The specially designed maps and the results which show that the Naive Bayes Classifier outperforms Ontañón's proposed model can be seen in the next section.

5.1.6 Experiment Design and Results

As was mentioned previously, in order to eliminate the possibility that the Ontañón's proposed agent was learning to travel in a square and not actually learning Wall-Follower behavior, 3 new maps with varying dimensions were used. The dimensions of the maps are the following: 8x8, 10x14, 32x32. The maps had an assortment of obstacles and dirt scattered around to add complexity, but the main goal was to identify if Ontañón's model could learn WallFollower behavior if the expert didn't travel in squares or rectangles. The 8x8 map has an area where the expert is in between a *wall* to the *NORTH* and a *wall* to the *SOUTH*. This causes the expert to move *Left* and *Right* in a cycle, until the end of the run. The 32x32 map causes the expert to travel in an asymmetrical manner. The 10x14 map is a rectangle and the agent will follow a rectangular trajectory.

The 3 maps that were used are shown in Figures 5.5, 5.6, and 5.7.

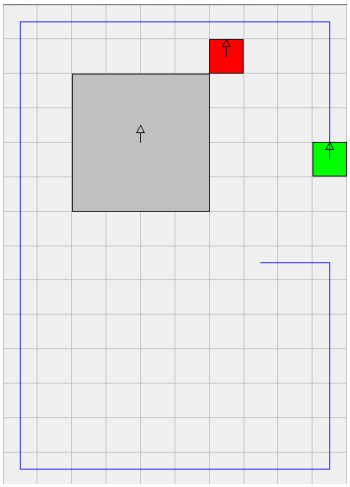


Figure 5.5: Map 1: 10x14 (WF Agent)

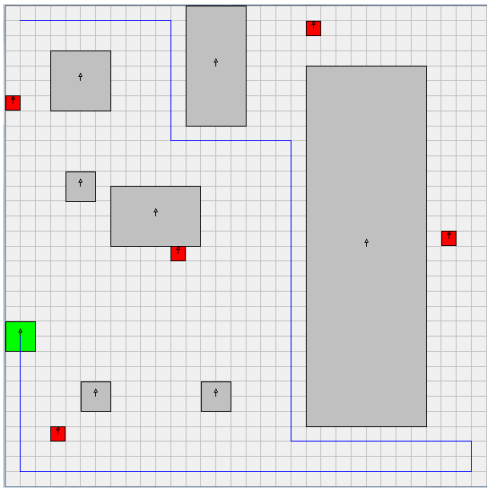


Figure 5.6: Map 2: 32x32 (WF Agent)

Results

The performance results on the 3 maps for the WF using Ontañón's model and the Naive Bayes Classifier are shown in Table 5.3.

	Map 1		Map 2		Map 3	
Model	Acc	F1	Acc	F1	Acc	F1
Original	1.00	1.00	0.75	0.76	0.5	0.33
NaiveBayes	1.00	1.00	0.85	0.84	0.97	0.81

Table 5.3: Results for learning the WallFollower Agent behavior using Ontañón's model and the Naive Bayes Classifier

The results are for Maps 1, 2, and 3.

5.1.7 Discussion

The Naive Bayes model is able to outperform Ontañón's model in the 8x8 and 32x32 maps because the expert behavior in those maps is unlike the behavior in

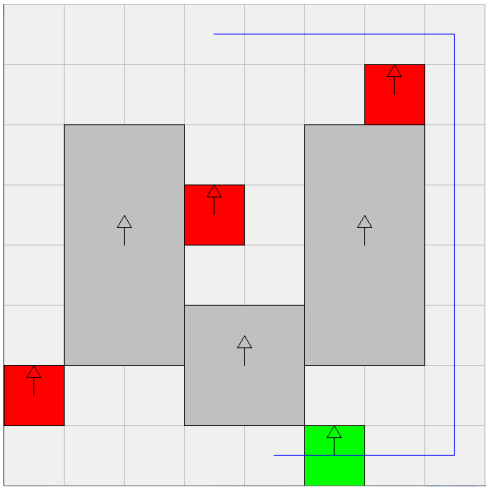


Figure 5.7: Map 3: 8x8 (WF Agent)

the 10x14 map. Ontañón's model is able to learn the correct behavior for the 10x14 map because the expert simply travels in a rectangle. However, when the maps are modified so the behavior is no longer symmetric, Ontañón's model is unable to predict the correct behavior.

5.1.8 Summary

The results show that given a situation where the testing set has new unseen environment state, the Naive Bayes approach is able to outperform Ontañón's proposed method because it can make an inductive leap. The Naive Bayes Classifier does have a high bias and low variance, and it would be interesting to determine if there are domains where this is a disadvantage in comparison to Ontañón's model. Further discussion of this can be seen in chapter 7.3.

In this section we've seen an application of bias into the PGM space, which

led to an improvement in performance. In the next section, we use the notion of bias to extend the state-based learning techniques in the CBR space. We propose new *Run* similarity metrics, which can be paired with a retrieval algorithm (kNN) to perform *state-based* retrieval, as an alternative to the TB algorithm.

5.2 Introducing Bias into the State-Based CBR Domain

5.2.1 Problem Statement

Temporal backtracking has been used as the extension of CBR, to learn state-based behavior. It performs comparably to DBN in most of the state-based learning in the vacuum cleaner and RoboCup domain as seen in section 4. There is of course, always the question to be asked: are there situations where TB won't perform well? We will be able to show this in the obstacle avoidance domain in chapter 6.2, where the TB algorithm had difficulty learning the expected behavior of the Left-Right Toggle agent. Then are there other state-based CBR learning methods that can be used instead of TB?

Recall the definition of a case, which Floyd [6] proposed for learning state-based behavior:

$$C = R^E \times Ac$$

Where R^E is the subset of R that contains all the runs that end in an environment state. During kNN's retrieval phase, the most similar cases to the query are found,

by applying some similarity metric between the elements in the cases (recall the *Atomic* and *Complex* similarity metrics from chapter 3). However, now that the cases contain runs, there have to be some similarity metrics that can be used to compare runs (i.e., *Run* similarity metrics). This new case structure allows for performing *state-based* retrieval and opens up the door plugging in many different types of similarity metrics, and comparing the performance to TB.

A visualization of state-based retrieval using all three types of similarity metrics is shown in Figure 5.8. The *n-ordered* metric is used in the example (presented in section 5.2.2), as the *Run* similarity metric which aggregates the similarities between each case at different times.

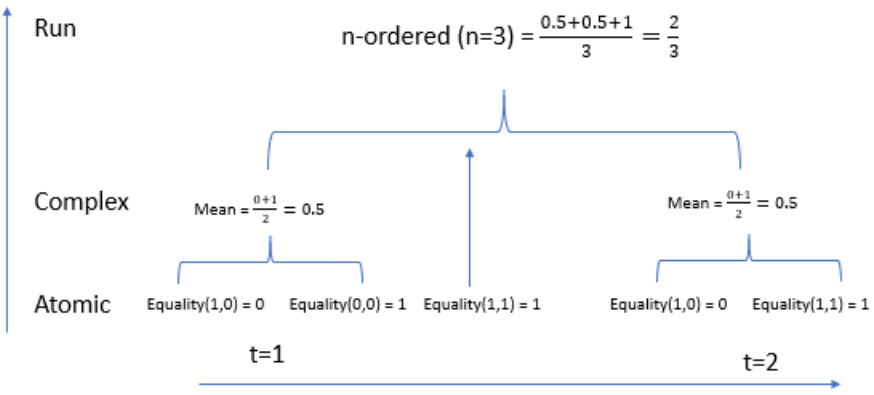


Figure 5.8: Example similarity calculation (State-Based)

A similarity calculation between a query and a case in a subset of the vacuum cleaner domain using Atomic, Complex and Run similarity metrics.

In our framework[1], the user presets the *Atomic* and *Complex* similarity metric for each element and the *Run* similarity metric for the runs.

[1] available at https://github.com/NMAI-lab/jLOAF

In this section we

- propose three new *Run* similarity metrics which are paired with the kNN algorithm to be used in place of TB to perform state-based retrieval,
- provide handcrafted examples to show how these new metrics used with kNN are an improvement over TB in learning certain types of behavior and
- show that there is no one-size fits all algorithm for learning state-based behavior.

5.2.2 N-ordered Similarity

The *n-ordered* similarity metric only considers the most recent n elements of the runs as important, i.e., it is biased towards the more recent elements. It performs a one-to-one comparison of each element in two runs using *Sim*, from most recent element to the n'th element. The formula is shown in equation 5.1. The notation $|r|$ is used here to show the number of elements in run r. r[t] refers to the t'th element of the run in the case, where the elements alternate between environment states and actions. q refers to a query which is always a run which is looking for an action to perform.

$$n-orderedSim(q,r,n) = \frac{\sum_{t=0}^{min(|q|,|r|,k)} Sim(q[t],r[t])}{min(|q|,n)} \tag{5.1}$$

Whereas the TB algorithm dynamically backtracks until it reaches some consensus or it reaches the end of the run, the *n-ordered* metric measures the sim-

ilarity of a preset number of most recent elements. In TB, if there is mismatch of environment states or actions during the backtrack, certain runs will get eliminated. However, it may be the case that there are more similarities closer to the end of the run, which will overpower the initial decision. So, while TB would not consider these runs due to early elimination, the state-based retrieval with *n-ordered* as a *Run* similarity metric will not make any decisions until all cases in the casebase have been considered.

This metric would work well on learning behavior that followed a fixed sequence, if the value of *n* is large enough to encompass more than half the sequence. Let's take a look at the following runs from a fixed sequence agent in the vacuum cleaner domain. The agent loops over the same fixed sequence of Up, $Right$, $Down$, $Down$, and $Left$ independent of the environment state.

$$time : t-6 \xrightarrow{t-5} t-4 \xrightarrow{t-3} t-2 \xrightarrow{t-1} t \xrightarrow{t+1}$$

$$Run\,1 : 01010101 \xrightarrow{Right} 01010101 \xrightarrow{Down} 01010111 \xrightarrow{Down} 01110101 \xrightarrow{Left}$$

$$Run\,2 : 01010101 \xrightarrow{Up} 01010101 \xrightarrow{Right} 01010101 \xrightarrow{Down} 01110101 \xrightarrow{Down}$$

$$Query : 01010101 \xrightarrow{Up} 01010101 \xrightarrow{Right} 01010111 \xrightarrow{Down} 01110101 \xrightarrow{?}$$

If we apply the TB algorithm using an equality metric *Sim*, and setting PPT, RPT and ST to be 0.99 (See Chapter 3.2.2 for explanation of these settings), there is no consensus of the action at time *t*, so at time *t-1* the actions in both cases is the same as the query, so it moves to *t-2*, where the environment state of the query

is the same as the Run 1 and different from Run 2. Therefore, Run 2 is eliminated and the action chosen is the most recent action from Run 1: $Left$. However, the expected action is $Down$. If the *n-ordered* metric (n=7) is applied during the retrieval phase of the kNN (k=1) algorithm we will see the results in Table 5.4:

	t-6	t-5	t-4	t-3	t-2	t-1	t	Run Similarity
q	01010101	Up	01010101	Right	01010111	Down	01110101	
r1	01010101	Right	01010101	Down	01010111	Down	01110101	
sim(r1[t],q[t])	1	0	1	0	1	1	1	0.71
r2	01010101	Up	01010101	Right	01010101	Down	01110101	
sim(r2[t],q[t])	1	1	1	1	0	1	1	0.86

Table 5.4: Application of n-ordered similarity metric to compare a query to two runs, r1 and r2.

As can be seen in Table 5.4, the kNN algorithm will choose Run 2 because it is most similar case to the Query based on the *n-ordered* metric. It avoids the pitfall of prematurely rejecting candidate runs, like the TB algorithm does.

5.2.3 N-unordered Similarity

The *n-unordered* similarity metric is inspired by the Jaccard distance. It focuses on comparing the distributions of environment state and actions in the runs. Practically, the metric compares the count of elements in two runs in the most recent n elements.

N-unordered will be able to learn behavior that is dependent on the count of past environment states and actions. For example: a vacuum cleaner agent, that has to turn left or right when it hits an obstacle depending on the number of times it saw dirt to its left.

In the following formula for the *n-unordered* similarity (equation 5.5), the function *countRatio* is used to calculate the ratio of an element contained in two runs. The *count* function is used in *countRatio* to get the count of an element in a run (equation 5.3).

The Kronecker delta function is used in the *count* function and is defined below:

$$\delta(i,j) = \begin{cases} 0 & \text{if i} \neq \text{j} \\ 1 & \text{if i = j} \end{cases} \tag{5.2}$$

$$count(element, r) = \sum_{t=0}^{|r|} \delta(element, r[t]) \tag{5.3}$$

The formula for count-ratio is shown in equation 5.4:

$$countRatio(e, r1, r2) = \frac{min(count(e, r1), count(e, r2))}{max(count(e, r1), count(e, r2))} \tag{5.4}$$

The formula for the *n-unordered* similarity metric is shown in equation 5.5. The runs are converted into sets in order to get the unique elements.

qSet $\leftarrow$ n most recent elements of q as a Set

rSet $\leftarrow$ n most recent elements of r as a Set

qk $\leftarrow$ n most recent elements of q as a list

rk $\leftarrow$ n most recent elements of r as a list

$$n{-}unorderedSim(q, r, n) = \frac{\sum_{e \in qSet} countRatio(e, qk, rk)}{|qSet|} \tag{5.5}$$

In the following example, we show why the TB algorithm can have difficulty

learning count-based behavior. Let's take a look at the following runs from a "count agent" in the vacuum cleaner domain. The agent will count the number of dirt it sees to the WEST, and when it hits an obstacle/wall to the NORTH, it will turn $Left$ if the dirt count is odd, and $Right$ if the dirt counter is even.

$$time : t-4 \xrightarrow{t-3} t-2 \xrightarrow{t-1} t \xrightarrow{t+1}$$

$$Run\,1 : 01010111 \xrightarrow{Up} 01010111 \xrightarrow{Up} 00010101 \xrightarrow{Right}$$

$$Run\,2 : 01010111 \xrightarrow{Up} 01110101 \xrightarrow{Up} 00010101 \xrightarrow{Left}$$

$$Query : 01110101 \xrightarrow{Up} 01010111 \xrightarrow{Up} 00010101 \xrightarrow{?}$$

If we apply the TB algorithm (using an equality metric *Sim*, and setting PPT, RPT and ST to be 0.99) there is no consensus of the action at time *t*, so at time *t-1* the actions in both cases are the same as the query, so it moves to *t-2* where the environment state of the query is the same as the Run 1 and different from Run 2. Therefore, Run 2 is eliminated and the action chosen is the most recent action from Run 1: $Right$. However, the expected action is $Left$, because the Query's dirt counter is 1 (Odd). If we apply a *n-unordered* metric (n=5) to cases during the retrieval procedure of kNN (k=1), the results of the count are shown below:

$$Count\ for\ Run\,1 : (Up : 2, 01010111 : 2, 00010101 : 1, 01110101 : 0)$$

$$Count\ for\ Run\,2 : (Up : 2, 01010111 : 1, 00010101 : 1, 01110101 : 1)$$

$$Count\ for\ Query : (Up : 2, 01010111 : 1, 00010101 : 1, 01110101 : 1)$$

Using these we can calculate n-unorderedSim(Query, Run 1, 5) to be 0.5, and n-unorderedSim(Query, Run 2, 5) to be 1.0 (Intuitively, it can be seen that Query and Run 2 have the same number of matched elements). Therefore, the action chosen is the expected action: $Left$.

5.2.4 Weighted Similarity

The *weighted* similarity metric is very similar to the one presented in [8], where the author gives decaying weights to the more recent elements in the case.

This metric compares each element in two cases from the most recent element to the last element and weights the more recent elements more heavily. This algorithm is biased towards the more recent elements but still considers all the elements in the case. The formula is shown in equation 5.6. ω is a weighting factor that determines how heavily the most recent elements in the case are weighted. The weighting is assigned in an exponentially decreasing manner with the lowest weighting being 1.

$$weightedSim(q, r, \omega) = \frac{\sum_{t=0}^{min(|q|,|r|)} sim(q[t], r[t]) \times (1 + e^{-t} \times \omega)}{|q|} \quad (5.6)$$

The *n-ordered*, *n-unordered* and *weighted* similarity metrics are a subset of the possible metrics that could be used to measure the similarity between cases. They were chosen because they filled gaps which the TB algorithm had when it came

to learning different types of behavior. In the next section, the proposed metrics used with kNN are tested against the TB algorithm in learning different types of state-based behavior. A particularly focus was placed on learning behavior that was known to cause the TB algorithm to perform poorly, and would theoretically allow the presented metrics to perform well.

5.2.5 Experiment Design and Results

Two state-based agents from Ontañón's paper: the ZigZag agent and the Fixed Sequence agent, were used to test the state-based retrieval method with the new similarity metrics against the TB algorithm. The ZigZag agent was used because TB algorithm has been able to capture the behavior well [9]. The Fixed Sequence agent was used because the TB algorithm has difficulty imitating the behavior well. An agent called the DirCount agent was designed specifically so that the TB algorithm theoretically would have trouble reproducing the behavior. The agent is described in the next section.

DirtCount Agent

The agent moves down and counts the number of times it sees dirt to the left and the number of times it is next to a wall on its right or left. When the agent collides with a wall or obstacle below it, if the dirt count is greater than the wall count it moves left, otherwise it moves right. Then the agent repeats this behavior but in the "up" direction. The behavior is described formally in Algorithm 6.

Algorithm 6 DirtCount Agent

```
1: Down ← True
2: DirtCount ← 0
3: WallCount ← 0
4: function GETACTION(input):Action
5:     if DirtLeft(input) then
6:         DirtCount ++
7:     if WallCloseLeftOrRight(input) then
8:         WallCount ++
9:     if (WallCloseDown(input) & Down == True) || (WallCloseUp(input) &
   Down == False) then
10:        Down ← ¬ Down
11:        if DirtCount>WallCount then
12:            return Action: Left
13:        else
14:            return Action: Right
15:    else
16:        if Down then
17:            return Action: Down
18:        else
19:            return Action: Up
```

Experiment Setup

The experiments for the DirtCount agent were performed using 3-fold cross-validation, with 3 traces containing a 100 cases each. Each trace contains the expert behavior on a different map. The maps were specifically designed to limit the TB algorithm's ability to predict the behavior of the DirtCount agent, and are shown in Figures 5.9, 5.10 and 5.11. This was done by varying the number of dirt (in red) and nearby obstacles (in gray) to cause the agent (in green) to differ on whether it moves left or right when it comes into contact with a wall or obstacle

above or below it[2]. For example, if the agent is going *Down* on Map 1, it will move *Left* when it comes into contact with the obstacle, while in Map 2 and 3 it will move *Right*. This is because in Map 1 the DirtCount agent saw dirt to its left, but didn't in Map 2 and 3.

The maps used to generate the behavior of the Fixed Sequence agent and ZigZag agent were those used in the original paper [4]. For testing, 7-Fold cross validation was performed with each trace having 1000 cases. The results from the experiments are shown in the next subsection.

The value of *n* in the *n-ordered* and *n-unordered* algorithms were chosen based on the agent that was being tested. For the Fixed Sequence agent the *n* was set to half the length of the sequence, which was 11. This is because at any point, the learner should have approximately half the sequence in the portion of the run it is comparing with other runs. It theoretically should be easy to identify at what point in the run it is and then choose the next action. This works for both the *n-ordered* and *n-unordered* algorithms. For the ZigZag agent, the algorithms had a *n* of 11 because the information that determines the agents movement at a wall is approximately the length of distance from the opposite wall. This distance was generally small, since most of the maps are 8x8. Therefore, there is no requirement to go further back. For the DirtCount agent, we chose a *n* of 11 as well; this was about twice the length that the agent would travel before it made the decision to turn *Left* or *Right*.

The TB algorithm was run with the following thresholds: (PPT=RPT=ST=0.99)

[2]https://github.com/sachag678/VacuumCleaner

and (PPT=0,RPT=ST=0.99). With the PPT=0.99, the TB algorithm will consider all past environment states, whereas with PPT=0, it will not consider the past environment states. In the case of the Fixed Sequence agent, the behavior is only dependent on the past actions. Therefore, we expect the TB algorithm with the PPT=0, to perform well in learning that behavior. In behaviors where the past environment states are important like the ZigZag agent, we expect the algorithm with PPT=0.99 to perform well.

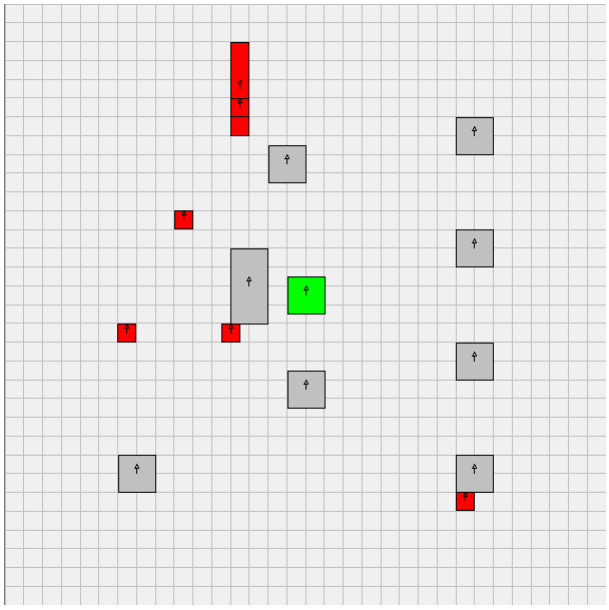


Figure 5.9: Map 1: 32x32 (DirtCount Agent)

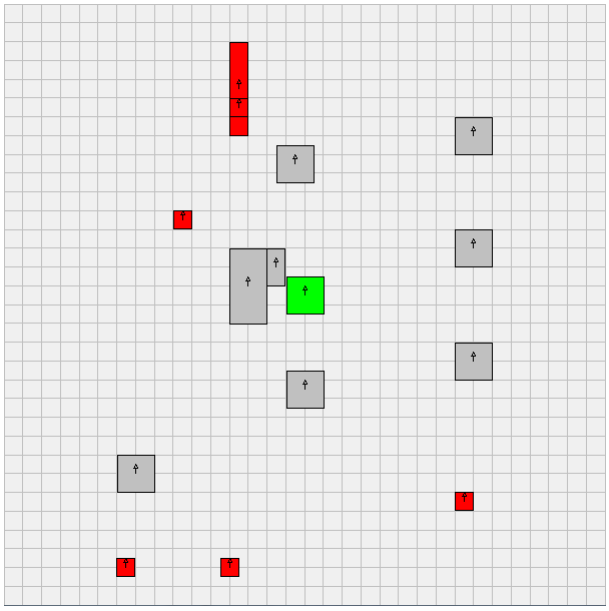


Figure 5.10: Map 2: 32x32 (DirtCount Agent)

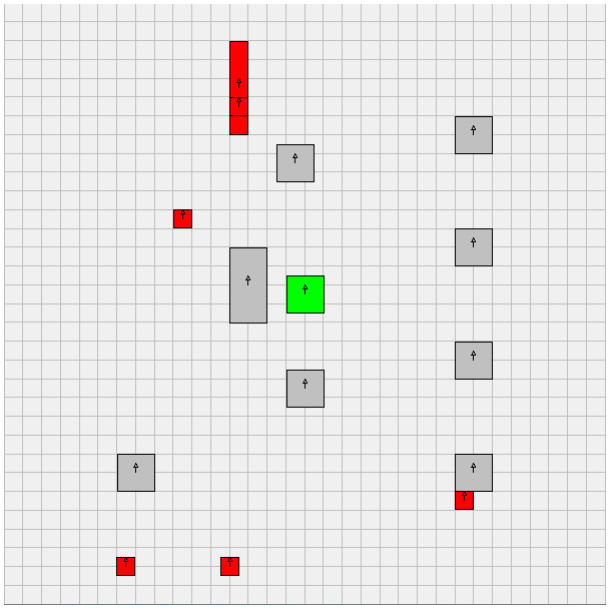


Figure 5.11: Map 3: 32x32 (DirtCount Agent)

Results

The results of the experiments are summarized in Tables 5.5, 5.6 and 5.7. The tables contain the Accuracy (Acc) and Average F1 Score (F1) score of predicting the expert behavior.

	Map 1		Map 2		Map 3	
Similarity	Acc	F1	Acc	F1	Acc	F1
TB	0.88	0.60	0.92	0.75	0.79	0.87
TB(PPT=0)	0.88	0.60	0.92	0.75	0.74	0.61
n-ordered	1.00	1.00	0.94	0.8	0.88	0.48
n-unordered	0.93	0.71	0.94	0.8	0.93	0.66
weighted	0.99	0.95	0.94	0.73	0.88	0.49

Table 5.5: Results for DirtCount Agent: Maps 1, 2, and 3

Similarity	Accuracy	Average F1 Score
TB	0.66±0.32	0.62±0.35
TB (PPT=0)	1.00±0.00	1.00±0.00
n-ordered	0.98±0.01	0.98±0.01
n-unordered	0.99±0.01	0.99±0.01
weighted	0.98±0.01	0.98±0.01

Table 5.6: Results for Fixed Sequence Agent: Original Maps

Similarity	Accuracy	Average F1 Score
TB	0.95±0.00	0.78±0.06
TB(PPT=0)	0.83±0.16	0.72±0.10
n-ordered	0.85±0.06	0.73±0.13
n-unordered	0.79±0.06	0.67±0.07
weighted	0.86±0.04	0.70±0.13

Table 5.7: Results for ZigZag Agent: Original Maps

5.2.6 Discussion

DirtCount Agent

The results for the DirtCount agent are shown in Table 5.5. The reason the TB algorithm doesn't perform poorly is because for the most part, the actions of the DirtCount agent are *Up* and *Down*. These actions are determined by whether boolean Down is true or false. The TB algorithm will have no issue learning these actions. However, the TB algorithm should have trouble predicting whether to turn *Left* or *Right* when the agent hits a wall above or below it. The maps were designed to increase the amount of times this occurred, however, it was limited by the fact that in between two obstacles above or below, there had to be dirt or a wall to increase the counters of the agent. The *n-ordered* and *n-unordered* metrics outperform the TB algorithm, with the exception of average F1 Score in Map 3.

Fixed Sequence Agent

The TB algorithm with PPT=0.99 has a relatively low accuracy in comparison to the other algorithms for Map 3 which is shown in Table 5.6. Since the TB algorithm dynamically backtracks one time step at a time, and gives equal weight to environment states and actions, it is unable to imitate a sequence of actions that is not dependent on the environment. However, if the PPT=0, the **TB algorithm** has excellent results. This is because it will not consider any environment states other than for the current environment state, and only consider the past actions. The *n-ordered* and *n-unordered* algorithms are able to perform better for different

reasons. The *n-ordered* metric is able to capture the behavior because of the cyclical nature of the sequence. If the *n-ordered* metric is able to have a n value that is half the size of the sequence then it will be able to capture the sequence behavior. The *n-unordered* metrics function based on the count of elements in the run. In this case, the unique number of actions that occur before another action allow the two metrics to capture the behavior as well.

ZigZag Agent

The results for the ZigZag agent are shown in Table 5.7. This agent was used to showcase the limitations of the count-based similarity metric. The behavior of the ZigZag agent is dependent on the action it performed the last time it collided with an obstacle. This action occurs approximately 10 time steps into the past. The cases further in the past don't have a bearing on the current decisions. The **n-unordered** metric under-performs relative to the other algorithms. The counts of the actions or the environment states are not important factors in this behavior, and this could be the reason for the poor performance.

5.2.7 Summary

We have shown that we can come up with arbitrary counter examples which show the limit of TB's ability to predict the correct behavior, specifically with the Dirt-Count agent. We have also shown that there are experts, such as the ZigZag agent, that theoretically limit some of the proposed similarity metrics' ability to predict the correct behavior. The results reaffirm Floyd's claim that the thresholds in TB

have to be tuned for the specific behavior, and that tuning them incorrectly can lead to poor results. The same can be said of the value of n in the proposed similarity metrics. Based on the results, it is difficult to conclusively say that a specific metric or algorithm will always be able to learn a specific type of behavior. This reiterates our claim that there isn't a universal one-size-fits-all bias for learning state-based behavior at this point.

5.3 Conclusion: Bias

There have been contributions in two areas in this section.

- A Naive Bayes classifier approach was tested against Ontañón's discriminative approach, which didn't show much improvement in the vacuum cleaner domain in the original maps. However, when maps which had varied size, and caused the WallFollower agent to travel in an asymmetric manner were used, the Naive Bayes approach outperformed the Ontañón's approach.

- The new *Run* similarity metrics which compared cases that now contain runs were presented with examples in the vacuum cleaner domain, and we showed that there is no one-size fits all metric for state-based learning in the CBR domain.

The next chapter will focus on the other branch of contributions that was described in the introduction: the comparative study of the PGM and CBR approaches in the continuous domain.

Chapter 6

Contributions to LfO II: Comparative Study of CBR vs PGM in the Continuous Domain

This section presents a comparison of the reactive retrieval and TB algorithms (CBR) with the BN and DBN models (PGM) in learning reactive and state-based behavior in the continuous domain, namely in Cartpole V0, obstacle avoidance and RoboCup. Since the PGM and CBR approaches were initially compared in the discrete vacuum cleaner domain, this extension into the continuous domain is an important next step in the comparative study between the two approaches. Can these approaches that perform well in the discrete domain, generalize and learn behavior in an environment with continuous features?

This Chapter is divided into the following sections:

- Section 6.1 presents a comparison of the reactive retrieval with the Gaussian Naive Bayes Classifier (GNBC). As mentioned in the previous chapter, this was a question posed in C. Chan's [8] future work section. The testing and validation was performed in the continuous Cartpole V0 domain. The results showed that in the Cartpole V0 domain, the CBR approach outperforms the GNBC approach.

- Section 6.2 presents a comparison of the TB algorithm with the Dynamic Gaussian Bayes Network (DGBN) model. The testing and validation was performed in the obstacle avoidance domain, In the obstacle avoidance domain both the DGBN and TB had similar results, where they had difficult predicting *Left* or *Right* correctly. The DGBN learner did not perform as expected because it did not learn the correct CPDs. Therefore, a DGBN leaner with handcrafted CPDs was used in the testing, and this showed that that DGBN learner can imitate the behavior extremely well if it has learned the correct CPDs.

- Section 6.3 presents a PGM model called the Indicator model for learning behavior in the partially observable continuous domain. The partially observable domain presents an interesting challenge because when data is not observed, it can still provide information that the agent can use. Simple examples from the RoboCup domain are provided to validate the use of the Indicator model, and the experimental results show that the Indicator model outperforms the Gaussian Naive Bayes Classifier model. The test results

show that the regular and Indicator models outperform the CBR approach in performing the *Kick* and *Turn* actions. The *Dash* action has similar F1 scores for both CBR and BN approaches, which is most likely due to *Dash* being the majority action. The models were also used to play the game, and qualitative results showed that the CBR model performed was able to locate and dash to the ball, even if there was unobserved information. Conversely, the GNBC model had difficulty performing the correct actions when there was unobserved information. The CBR model was unable to *Kick* the ball in the correct direction, whereas the Indicator was able to *Kick* the ball into the goal.

6.1 Fully Observable Continuous Domain (Reactive) - Cartpole V0

6.1.1 Problem Statement

The problem that this section will address is: how do the reactive retrieval CBR and BN approaches compare in learning *reactive* behavior in a fully observable continuous domain such as Cartpole V0 [11]: First, the BN model used to learn the behavior will be presented, along with the parameters used in the CBR approach. Second, a hypothesis will be made based on the preliminary analysis of the dataset, and finally the results and discussion will be presented.

6.1.2 Proposed Model: Gaussian Naive Bayes Classifier

The Bayesian network model used in the Cartpole V0 domain is shown in Figure 6.1. It is a Gaussian Naive Bayes classifier (GNBC), which is the continuous version of the model that was used in Chapter 5.1.

One thing to note is that if we used Ontañón's model (Figure 5.1) with X_t as continuous vector, we would have had a model which had a continuous parent for a discrete child. The discrete child node would have to be converted into a Softmax node which would output the probability of each action given the continuous inputs. This model doesn't have to learn a very large CPT because it will learn the parameters of the Softmax function. The main issue is that the BNET toolbox doesn't support this model when the domain is partially observable, or when there are hidden variables to be learned (as is the case in the DBN models) because the techniques to perform learning and inference in these cases are still being developed [20]. We wanted to stay consistent with the model structure throughout the continuous domain, and therefore using the GNBC model was the best option.

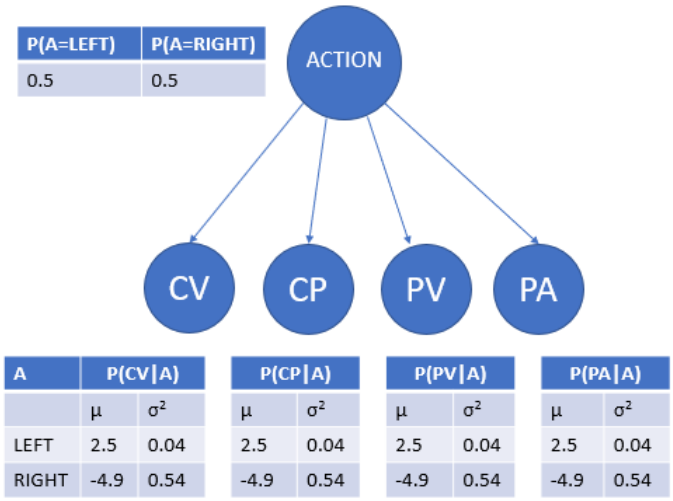


Figure 6.1: Gaussian Naive Bayes Classifier

Action is the **action** node, the children nodes are the feature of the environment state with CP: Cart Position, CV: Cart Velocity, PA: Pole Angle, PV: Pole Velocity. The CPT's and CPD's are shown for simulated data.

For reactive retrieval in CBR, a weighted K-Nearest Neighbor approach with a k of 5 was used.

6.1.3 Hypothesis

Assuming the training dataset follows a Gaussian distribution, the mean and variance of the each features from one of the training datasets are shown in Table 6.1.

As can be seen in Figure 6.2 there is significant overlap between the distributions of each feature for the *Left* and *Right* actions.

Assuming that the GNBC is able to learn the correct parameters for each dis-

Action	Cart Position	Cart Velocity	Pole Angle	Pole Velocity
Left	$\mathcal{N}(0.15, 0.004)$	$\mathcal{N}(0.14, 0.03)$	$\mathcal{N}(-0.0017, 0.0004)$	$\mathcal{N}(-0.14, 0.04)$
Right	$\mathcal{N}(0.14, 0.006)$	$\mathcal{N}(-0.05, 0.03)$	$\mathcal{N}(0.0023, 0.0007)$	$\mathcal{N}(0.15, 0.04)$

Table 6.1: Parameters of Features (Cartpole V0)

$\mathcal{N}(\mu, \sigma^2)$ represents a Gaussian Distribution with mean μ and variance σ^2

tribution, it will have difficulty deciding which Action the feature belongs to if the value of the feature lies in between the two distributions.

The CBR approach shouldn't have this issue because it makes no assumption about the distribution of the data. Since the dataset is very balanced, there will be no issue with a class imbalance during the majority check of the kNN algorithm. The expectation is that the CBR approach will outperform the BN approach in learning this behavior.

The following section presents the results of the experiments.

6.1.4 Results

The results of 10 - Fold cross validation using generated expert traces with 1000 cases in each trace were used and the results are as shown in Tables 6.2 and 6.3:

Model	Accuracy	Average F1 Score
Bayesian	0.80$\pm$0.01	0.79$\pm$0.01
CBR	0.92$\pm$0.01	0.93$\pm$0.01

Table 6.2: Accuracy and Average F1 Score results: Comparison of CBR and Gaussian Naive Bayes Classifier (Cartpole V0)

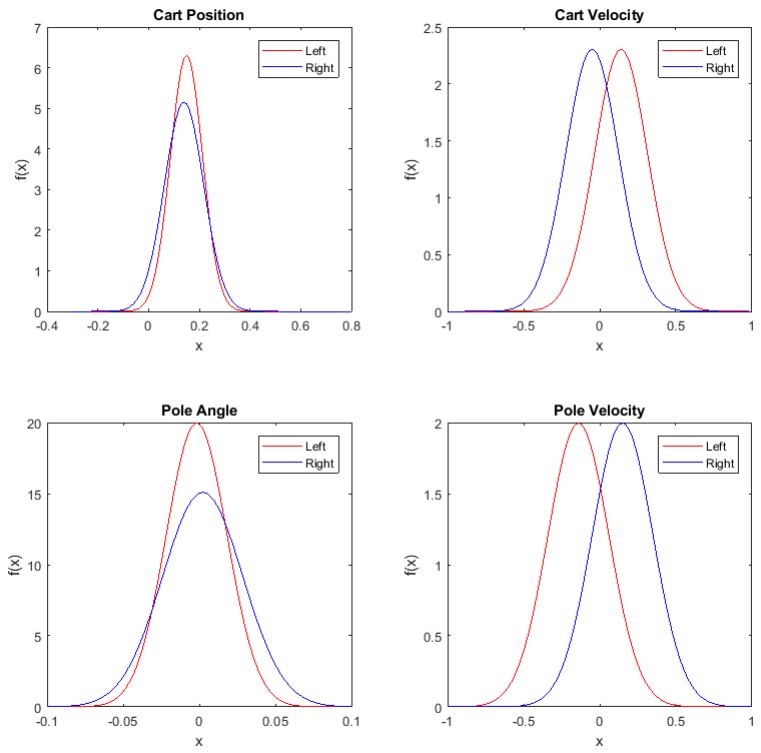


Figure 6.2: Probability Distributions of Cartpole Features

6.1.5 Discussion

The accuracy and average F1 score results in Tables 6.2 and 6.3 show that the CBR agent is able to learn the behavior much better than the GNBC agent. As stated in the hypothesis, the GNBC is able to choose the action correctly when the features values are distinctly on the outside edges of the distributions, but as the features in the query approach the mid point between distributions, then the GNBC model has trouble distinguishing which class the features belong to. The

Model	F1 Left	F1 Right
Bayesian	0.79±0.02	0.80±0.00
CBR	0.93±0.01	0.93±0.01

Table 6.3: F1 results per Action: Comparison of CBR and Gaussian Naive Bayes Classifier (Cartpole V0)

CBR has no such issue because it relies on similarity based on Euclidean distance, and chooses a majority class. In order to confirm the reason for the behavior, let us take a look at the distributions that the GNBC model learned from the same dataset used in the preliminary investigation:

$P(A = LEFT)$=0.5

A	P(Position|A)	P(Velocity|A)	P(P-Angle|A)	P(P-Velocity|A)
LE	$\mathcal{N}(0.15, 0.01)$	$\mathcal{N}(0.14, 0.04)$	$\mathcal{N}(-0.0017, 0.01)$	$\mathcal{N}(-0.14, 0.05)$
RI	$\mathcal{N}(0.14, 0.02)$	$\mathcal{N}(-0.05, 0.04)$	$\mathcal{N}(0.0023, 0.01)$	$\mathcal{N}(0.15, 0.05)$

Table 6.4: Parameters of Features: Probabilities of environment state given ACTION (Cartpole V0)

$\mathcal{N}(\mu, \sigma^2)$ represents a Gaussian Distribution with mean μ and variance σ^2.

The learned means match the means of the distributions presented in the preliminary investigation (Table 6.1). However, the learned variances do not match. Even if the GNBC was able to learn the correct variances it would still be capped in its performance due to the significant overlap in the distributions of the features. Therefore, this is the best the GNBC can perform for this dataset.

Now that we have examined the PGM vs CBR techniques to learn *reactive* behavior, we can see how they extend into learning *state-based* behavior. The next section will transform the Gaussian Naive Bayesian Classifier presented above into a Dynamic Gaussian Bayesian Network in order to learn *state-based* behavior.

6.2 Fully Observable Continuous Domain (State-Based) - Obstacle Avoidance

6.2.1 Problem Statement

The problem that this section endeavors to answer is how do the TB approach and DBN approach compare when learning *state-based* behavior in the fully observable continuous obstacle avoidance domain. The *state-based* behavior that is being learned is the Left-Right Toggle agent described in Chapter 2. The agent toggles between turning *Left* and *Right* when the *Sonar* value lies in between 2 and 3. The agent is *state-based* because it has to remember whether it last turned *Left* or *Right*. The structure of this section is the following: First, the DBN model used to learn the behavior will be presented. Second, a hypothesis will be made based on the preliminary analysis of the dataset. Finally, the results, a discussion and a DBN model with handcrafted CPDs are presented.

6.2.2 Proposed Model: Dynamic Gaussian Bayesian Network (DGBN)

Figure 6.3 is the DGBN model for learning *state-based* behavior in the obstacle avoidance domain. The CPTs and CPDs are shown for simulated training data. Similar to the continuous BN model presented in the previous section, the continuous DBN model uses the Gaussian Naive Bayes structure to inform the direction of the arrows between the *ACTION* and *SONAR*. The internal state was chosen

to be binary due to our prior knowledge of the domain, i.e., the Left-Right Toggle agent has a binary internal state that records whether it last turned left or not. Inference is performed by the BNET toolbox and can be complex, therefore, a hand-crafted example will not be provided.

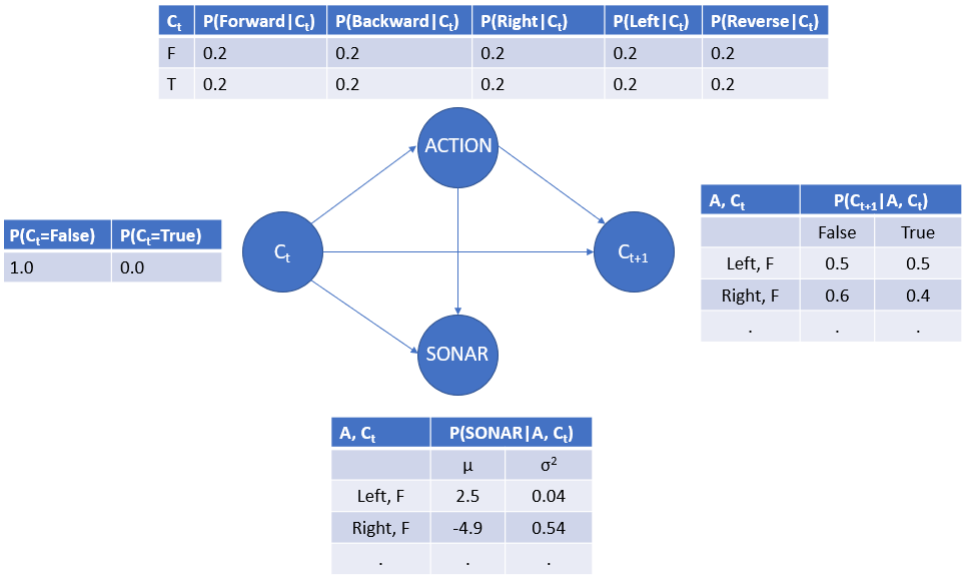


Figure 6.3: Dynamic Gaussian Bayesian Network model with CPTs and CPDs

ACTION is the action node, *SONAR* is the feature of the environment state, C_t is the internal state at time *t* and C_{t+1} is the internal state at time *t+1*. The · symbol is used to show that the table extends further down.

The TB algorithm will use a PPT = 0, RPT = 0.99, and ST = 0.99 and will use absolute distance to measure the similarity between features. The behavior of the Left-Right Toggle agent is such that it doesn't remember the past environment states, only the last time it turned *Left* or *Right*. So during the backtrack of the TB algorithm, there is no requirement to compare the past environment states, therefore, the value of PPT was set 0. The algorithm does need to compare the

past actions in order to find the last time the agent turned *Left* or *Right*, so ST is set to 0.99. RPT is set to 0.99 to ensure that only the most similar environment states to the current environment states are retrieved for the initial set of candidate runs.

6.2.3 Hypothesis

Assuming the dataset follows a normal distribution, the mean and variance of the features from one of the training datasets are shown in Table 6.5.

Action	SONAR
Forward	$\mathcal{N}(8.01, 16.9)$
Backward	$\mathcal{N}(0.52, 0.06)$
Left	$\mathcal{N}(2.56, 0.07)$
Right	$\mathcal{N}(2.59, 0.06)$
Reverse	$\mathcal{N}(1.54, 0.12)$

Table 6.5: Parameters of Features (obstacle avoidance domain): P(SONAR|Action)

$\mathcal{N}(\mu, \sigma^2)$ represents a Gaussian Distribution with mean μ and variance σ^2.

Figure 6.4 shows that the *SONAR* feature has good separation per action. Therefore, if the DBN learner is able to learn the parameters of the data and the internal state, it should perform very well. The TB agent will have difficulty performing the *Left* and *Right* actions because it gives equal weight to all the actions in the history. So when it is backtracking, it may prematurely eliminate candidate runs that don't match the action sequence of the query, and not identify the most important aspect - whether the expert turned *Left* or *Right*, the last time that 3 $>$*SONAR* $>$2.

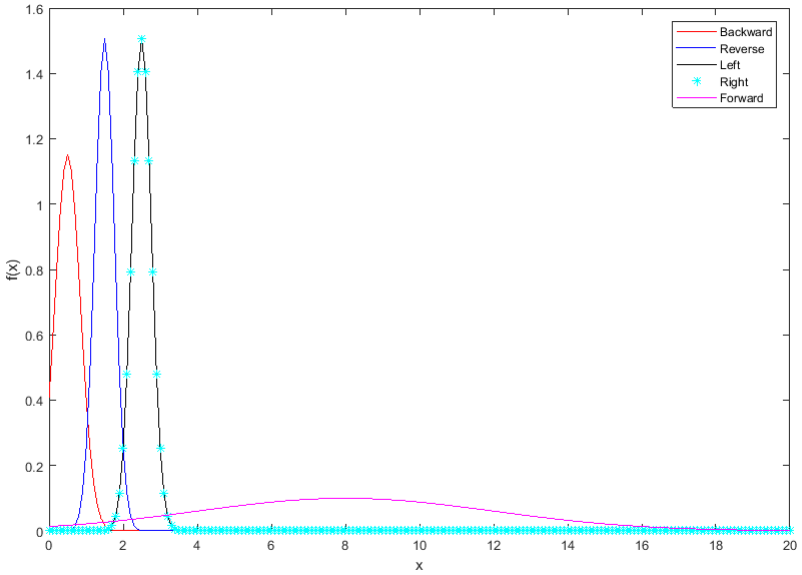


Figure 6.4: Probability Distribution of Sonar per Action

The next section presents the results.

6.2.4 Results

There were 10 traces with 2500 cases each which were used for 10-fold cross validation. Each traces represented an expert traversing a different map. Therefore, during cross-validation, the learning agent will be navigating a map that it hasn't seen before. The maps used were designed by [2].

Model	Accuracy	Average F1 Score
DBN	0.94±0.01	0.76±0.02
TB	0.96±0.01	0.82±0.01

Table 6.6: Accuracy and F1 results: Comparison of TB and DGBN Models (obstacle avoidance domain)

Model	F1 Left	F1 Right
DBN	0.40±0.04	0.45±0.06
TB	0.54±0.03	0.56±0.02

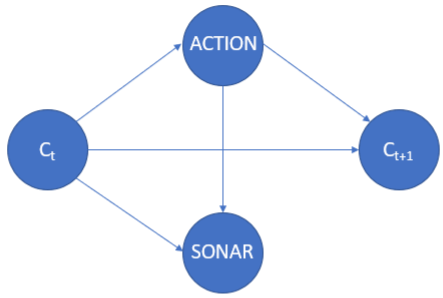
Table 6.7: F1 results per Action: Comparison of TB and DGBN Models (obstacle avoidance domain)

6.2.5 Discussion

As seen in Table 6.7, both the TB agent and DGBN agent have trouble performing the *Left* and *Right* actions, as seen by the F1 scores. The TB agent is performing as expected, however the DGBN agent should have performed better. Let us take a look at the learned CPDs of the DGBN model shown in Figure 6.5.

C_t	P(Forward | C_t)	P(Backward | C_t)	P(Right | C_t)	P(Left | C_t)	P(Reverse | C_t)
F	0.69	0.05	0.08	0.08	0.1
T	0.0	0.99	0.0	0.0	0.01

P(C_t=False)	P(C_t=True)
1.0	0.0



A, C_t	P(C_{t+1} | A, C_t)	
	False	True
Forward, F	1.0	0.0
Forward, T	0.16	0.84
Backward, F	1.0	0.0
Backward, T	1.0	0.0
Right, F	0.80	0.20
Right, T	0.73	0.27
Left, F	0.84	0.16
Left, T	0.92	0.08
Reverse, F	0.51	0.49
Reverse, T	0.99	0.01

A, C_t	P(SONAR | A, C_t)	
	μ	σ^2
Forward, F	5.48	2.81
Forward, T	12.6	12.3
Backward, F	0.49	0.13
Backward, T	0.21	0.09
Right, F	2.53	0.08
Right, T	2.56	0.07
Left, F	2.53	0.09
Left, T	2.44	0.08
Reverse, F	1.59	0.09
Reverse, T	1.43	0.08

Figure 6.5: Learned CPDs of DGBN model (obstacle avoidance domain)

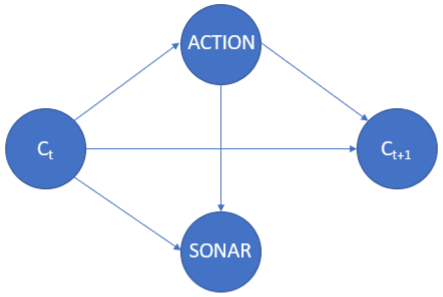
As can be seen in the CPDs, the variance of the distribution for *Forward* causes it to overlap with the distributions of *Left* and *Right*. If the *SONAR* value is 2.5, and the internal state is False, the action chosen is *Right*. This is in line with the behavior of the expert. However, if the *SONAR* value decreases toward 2.0, the overlap of the distribution of *Forward* will cause the *Forward* action to be chosen, which is not in line with expert behavior. The other issue is that because all the probabilities of P(C_{t+1}=False|C_t=False,A) are greater than P(C_{t+1}=True|C_t=False,A), if the initial state C_t is False, then C_{t+1} will always be False, and since C_0 is False,

the internal state C_t will always be false. This is not in line with the expert behavior. Ideally, the internal state C_t will toggle only when the actions *Left* and *Right* are performed as per the expert behavior.

A DGBN agent with handcrafted CPDs are shown in Figure 6.6.

C_t	P(Forward|C_t)	P(Backward|C_t)	P(Right|C_t)	P(Left|C_t)	P(Reverse|C_t)
F	0.24	0.24	0.24	0.04	0.24
T	0.24	0.24	0.04	0.24	0.24

P(C_t=False)	P(C_t=True)
0.9	0.1



A, C_t	P(C_{t+1}|A, C_t)	
	False	True
Forward, F	0.9	0.1
Forward, T	0.1	0.9
Backward, F	0.9	0.1
Backward, T	0.1	0.9
Right, F	0.9	0.1
Right, T	0.1	0.9
Left, F	0.1	0.9
Left, T	0.9	0.1
Reverse, F	0.9	0.1
Reverse, T	0.1	0.9

A, C_t	P(SONAR|A, C_t)	
	μ	σ^2
Forward, F	8	4
Forward, T	8	4
Backward, F	0.5	0.016
Backward, T	0.5	0.016
Right, F	2.5	0.016
Right, T	2.5	0.016
Left, F	2.5	0.016
Left, T	2.4	0.016
Reverse, F	1.5	0.016
Reverse, T	1.45	0.016

Figure 6.6: Handcrafted CPDs of DGBN model (obstacle avoidance domain)

The CPDs in Figure 6.6 show that *Left* or *Right* should occur based on the internal state C_t, that the distributions have very little overlap and that the new internal state C_{t+1} changes only when the *Left* or *Right* action are taken.

The results from using the DGBN agent with the above handcrafted CPDs are

Performance Measure	Result
Accuracy	0.99 $\pm$0.00
Average F1 Score	0.97 $\pm$0.00
F1 Left	0.95 $\pm$0.01
F1 Right	0.96 $\pm$0.02

Table 6.8: Performance Measures using the handcrafted DGBN agent (obstacle avoidance domain)

shown in Table 6.8. The results show that the DBN agent can indeed perform very well, given that it can learn the correct parameters for the CPDs. One of the possible reasons for the agent not learning the correct CPDs could be that the EM algorithm in the BNET toolbox required more training data than what was supplied, or it could be stuck in some local optima or saddle point during run time. It could also be that it has difficulty learning the internal states when the features have a large range (The feature values range from 0 to 20).

Now that we have looked at the fully observable domain, we focus on the partially observable domain. The next section modifies the previously used Gaussian Naive Bayes Classifier (GNBC) to account for the partially observable nature of the domain. It presents performance results of the proposed model, the CBR approach and GNBC imitating Krislet behavior in the RoboCup domain.

6.3 Partially Observable Continuous Domain (Reactive) - 2D Simulated RoboCup

6.3.1 Problem Statement

As mentioned in Chapter 2, RoboCup is a partially observable domain, in which the agent can only view certain portions of the environment. At each time-step of the game, the agent will receive an observation of the game, which will contain the objects that are in its field of view. For example, this observation may contain the ball, the goal and other players. But in some cases, it may only see the goal, or just the other players. An important fact to note is that even when there is no explicit information about an object in a specific observation, there is still knowledge of its location. For example, if the agent doesn't see a ball in front of it, then it knows the ball is somewhere behind it. The question is then: how does one represent this in the context of Bayesian networks?

First, we will show why the model presented in the previous section (Cartpole V0) may not be the best fit for this domain using an example. Then we will present an alternative model that attempts to solve the issues of the first model, and stitch in the partially observed nature of the environment. Then we will present a hypothesis on the performance of the two models compared with that of the reactive retrieval algorithm. The results of the experiments, as well as the discussion will be presented as well.

6.3.2 Regular Model: Gaussian Naive Bayes Classifier

Consider a subset of the RoboCup domain, where the Actions available to the agents are *Turn* and *Dash*, and the environment state only has 1 feature - *BALLDIRECTION*.

BALLDIRECTION	*ACTION*
2.4	Turn
0	Dash
N/A	Turn
0	Dash

Table 6.9: Training data (subset of a RoboCup trace)

Features - *BALLDIRECTION*, Actions - *Turn* and *Dash* (N/A cell represents unobserved information)

The model with the learned CPDs for the provided training data in Table 6.9 is shown in Figure 6.7.

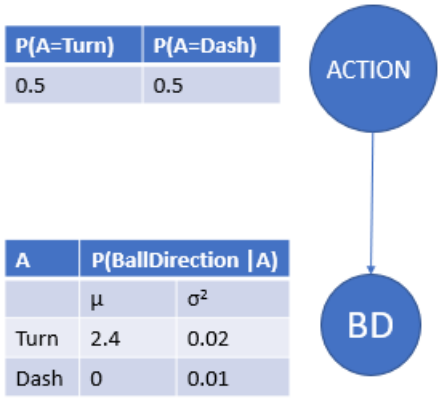


Figure 6.7: Learned CPDs of GNBC model (RoboCup domain)

Since there are empty cells in the training set, the Expectation Maximization (EM) algorithm must be used for learning. A key feature of EM is its ability

to handle missing data. During the Expectation-Step, the missing values will be filled in based on the current expectation of the missing value, given the observed values [21]. But in the context of partial observability, it doesn't make sense to fill in the value, because it is not missing: it is not observed. The imputation of the missing values could cause the learned distributions to be different from the distribution based on the observed information. The Indicator model presented in the next section solves this issue by creating two different Gaussian distributions for observed and unobserved data.

If there is no evidence provided because *BALLDIRECTION* was not observed, the following $P(A)$s are obtained ($f(x|y)$ is used to represent a mixed conditional probability density function, $f(x, y)$ is used to represent a mixed joint density function and $P(x)$ is used to represent a probability mass function):

Since BD is continuous and unobserved we have to integrate over all possible values of BD:

$$P(ACTION = Turn) = \int_{BD} f(A = Turn, BD) dBD$$

The probability distribution of BD given A=Turn is a probability density function taken from the table in Figure 6.9:

$$P(ACTION = Turn) = P(A = Turn) \int_{-\infty}^{+\infty} f(BD|A = Turn) dBD$$

By the definition of probability density function the integral evaluates to 1:

$$P(ACTION = Turn) = 0.5 \times 1 = 0.5$$

Repeat the same steps above for Action = Dash:

$$P(ACTION = Dash) = \int_{BD} f(A = Dash, BD) dBD$$

$$P(ACTION = Dash) = P(A = Dash) \int_{-\infty}^{+\infty} f(BD|A = Dash) dBD$$

$$P(ACTION = Dash) = 0.5 \times 1 = 0.5$$

If *BALLDIRECTION* is not observed, this model does not account for it because there are no probabilities or random variables which represent *BALLDIRECTION* being observed or not observed, which is another issue that the Indicator model solves.

6.3.3 Proposed Model: Gaussian Naive Bayes Classifier with Indicator Variables (Indicator Model)

The features in the Indicator model can be broken down into two parts - a continuous variable that represents the feature when it is observed, and a discrete variable that indicates whether it is observed or not (Indicator). Using this idea, the proposed model contains a binary node *BALLSEEN* which represents whether the continuous node *BALLDIRECTION* is observed (The training dataset in Table

6.10 has an additional column for *BALLSEEN*).

BALLSEEN	*BALLDIRECTION*	*ACTION*
1	2.4	Turn
1	0	Dash
0	N/A	Turn
1	0	Dash

Table 6.10: Training data (subset of a RoboCup trace)

Features - *BALLSEEN* and *BALLDIRECTION*, Actions - *Turn* and *Dash* (N/A represents an unobserved data point)

In this model, there is a direct relationship between *BALLSEEN* and *BALLDIRECTION*. If *BALLDIRECTION* is not observed, *BALLSEEN* will be false. This is represented by an arrow connecting the two nodes as seen in the model shown in Figure 6.8.

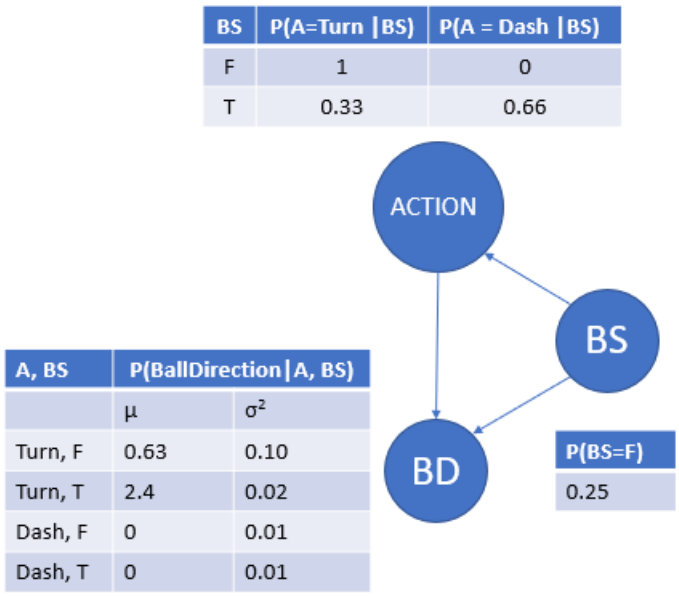


Figure 6.8: Learned CPDs of Indicator model (RoboCup Domain)

The binary node is a parent to the Action node

In this model, *BALLSEEN* is a parent of the *ACTION* node.

Using the model, if *BALLDIRECTION* was not observed, the following inference will be made:

$$P(A = Turn|BS = F) = \frac{f(BS = F, A = Turn)}{P(BS = F)}$$

$$P(A = Turn|BS = F) = \frac{\int_{BD} f(BS = F, A = Turn, BD)dBD}{P(BS = F)}$$

$$... = \frac{P(A = Turn|BS = F)P(BS = F)\int_{-\infty}^{+\infty} f(BD|A = Turn, BS = F)dBD}{P(BS = F)}$$

$$... = \frac{1 \times 0.25 \times 1}{P(BS = F)} = \frac{0.25}{P(BS = F)}$$

$$P(A = Dash|BS = F) = \frac{\int_{BD} f(BD, BS = F, A = Dash)}{P(BS = F)}$$

$$... = \frac{(A = Dash|BS = F)P(BS = F)\int_{-\infty}^{+\infty} f(BD|A = Dash, BS = F)dBD}{P(BS = F)}$$

$$... = \frac{0 \times 0.25 \times 1}{P(BS = F)} = \frac{0}{P(BS = F)}$$

In this case, *ACTION* is conditioned by *BALLSEEN* and therefore, this model doesn't leave out as much information as the regular model. In the regular model, given that *BALLSEEN* is False the probabilities of choosing one of the actions is 0.5, whereas in the proposed model the most likely action for this query is *Turn*, which aligns with the behavior of the Krislet agent turning when it cannot see the ball. In the proposed model, $P(Action|BS)$ is a CPT with rows conditioned on the value of *BALLSEEN*. This is more of a rote-learning method like Ontañón's,

as described in Chapter 5.1, but because *BALLSEEN* is binary, there are a limited number of rows to populate, and all the potential combinations will be observed by the learning agent.

The GNBC model potentially had an issue with the EM algorithm imputing values when data was not observed, however, in the Indicator model, if the data is not observed it will impute the value of the missing information into a different distribution than the distribution that describes the observed information. Thereby, it will not be skewing the distribution of the observed information in any way.

6.3.4 Hypothesis

The CBR reactive retrieval approach is able to handle the partial observable nature of the environment because when it compares cases, it does so in a set-like manner - comparing each feature in the case to the same feature in the query. If the case doesn't have a feature, there won't be a comparison, but the fact that it was unobserved will be taken into account through the application of a penalty when the mean similarity between the case and query is determined. An example of this can be seen in Figure 6.9, where even though *GoalDistance* is not compared to anything, when *Mean* aggregates the similarities, it divides the similarity by the size of the query, which is the penalty for the mismatch in size.

As such, the CBR method should perform well with respect to predicting the *Turn* and *Dash* actions, but due to the large class imbalance, as seen in Figure 6.10, it could have issues predicting the *Kick* action (K-nearest neighbor has issues predicting a minority class if there is a large class imbalance).

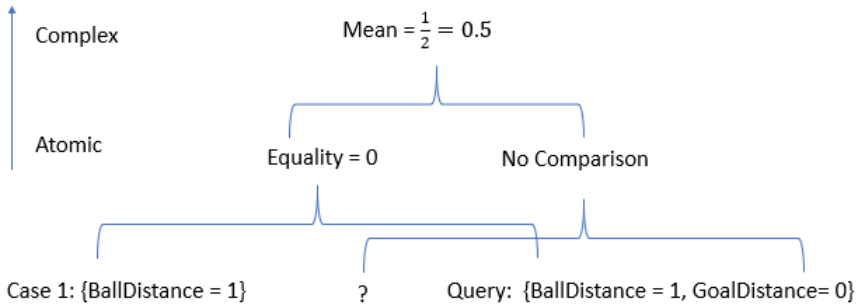


Figure 6.9: Comparing cases of different sizes in a partially observable domain

The Indicator model should be able to predict *Turn* and *Dash* well, whereas the *Kick* action may be affected by the class imbalance. However, since the environment state in which the *Kick* action occurs is different from when the *Dash* occurs, there may be less of an impact of class imbalance. The class imbalance will be seen in the CPT of the Action node, which can be outweighed by the probability from the environment feature nodes, unlike in the CBR case where the class imbalance directly affects the choice of Action. The GNBC should have worse results for predicting *Turn* and *Kick* because it only relies on the priors when data is not observed, and the priors will cause the agent to *Dash*.

The GNBC and Indicator models were used to learn Krislet behavior in the RoboCup domain. They were compared to the CBR reactive retrieval approach and the results of these experiments can be seen in the next section. The environment features include the *Distance* and *Direction* for the *Ball*, *Goal Left*, and *Goal Right* object. In the case of the Indicator model there are 4 binary indicator features which are *BALLSEEN*, *BALLCLOSE* (BC) *GOALSEENLEFT*, and *GOALSEENRIGHT*. (The *BALLCLOSE* variable is only a parent to the Action

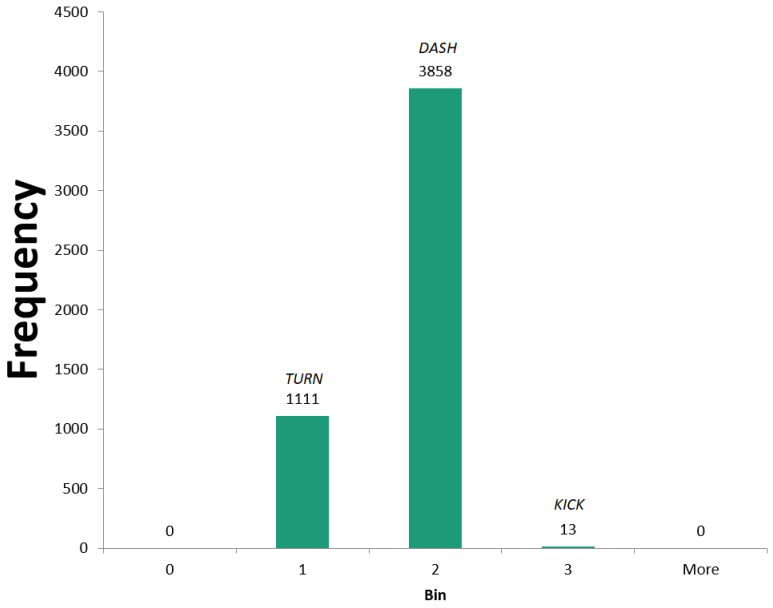


Figure 6.10: Distribution of Actions in the RoboCup domain.

node, since connecting it to the *BALLDISTANCE* node would not provide any additional information)

6.3.5 Results

The results of 5 - fold cross-validation using expert traces ($\tilde{4}000$ cases per trace) of the Krislet agent as training data are shown in Tables 6.11 and 6.12.

Model	Accuracy	Average F1 Score
Indicator	0.99±0.00	0.99±0.00
GNBC	0.97±0.00	0.83±0.04
CBR	0.93±0.01	0.76±0.06

Table 6.11: Comparison of Indicator model, GNBC and reactive retrieval models: Accuracy and Average F1 Score

Model	F1 Kick	F1 Turn	F1 Dash
Indicator	1.00±0.00	0.99±0.00	0.99±0.00
GNBC	0.56±0.12	0.94±0.01	0.99±0.00
CBR	0.51±0.17	0.82±0.02	0.95±0.01

Table 6.12: Comparison of Indicator model, GNBC and reactive retrieval models: Individual F1 Scores

6.3.6 Discussion

The Indicator model has higher accuracy and average F1 scores than the GNBC and CBR approaches. The F1 scores of the individual actions provide more details: As expected, the Indicator model outperforms the GNBC in terms of the F1 scores for *Turn* and *Kick* actions. Let's take a look at the CPT of the Action node (Table 6.13) of the Indicator model to determine why this is the case.

BS,BC,GSL,GSR	P(Turn|BS,BC,GSL,GSR)	P(Dash|BS,BC,GSL,GSR)	P(Kick|BS,BC,GSL,GSR)
F,F,F,F	1.00	0.00	0.00
F,F,F,T	1.00	0.00	0.00
F,F,T,F	1.00	0.00	0.00
F,F,T,T	0.34	0.33	0.33
F,T,F,F	0.34	0.33	0.33
F,T,F,T	0.34	0.33	0.33
F,T,T,F	0.34	0.33	0.33
F,T,T,T	0.34	0.33	0.33
T,F,F,F	0.18	0.82	0.00
T,F,F,T	0.19	0.81	0.00
T,F,T,F	0.28	0.72	0.00
T,F,T,T	0.34	0.33	0.33
T,T,F,F	1.00	0.00	0.00
T,T,F,T	0.00	0.00	1.00
T,T,T,F	1.00	0.00	0.00
T,T,T,T	0.34	0.33	0.33

Table 6.13: Probabilities of *ACTION* given *BALLSEEN*, *BALLCLOSE*, *GOALSEENLEFT*, *GOALSEENRIGHT*

It can be seen that any time *BALLSEEN*=False (the first 8 rows of the table),

the action is always *Turn*. This is exactly the Krislet behavior. If *BALLSEEN*=True, and *BALLCLOSE*=False (the next 4 rows of the table), there is a higher probability of *Dash*, a lower probability of *Turn* and very low probability of *Kick*. The final four rows show the distinction between *Kick* and *Turn* based on whether the *GOALSEENRIGHT* is true or false. These probabilities align with Krislet behavior.

The reason for the higher *Turn* F1 score in the Indicator model as compared to the GNBC is due to the addition of the indicator variable, which ensures the agent will always *Turn* when the ball is not seen. The reason for the higher F1 score for *Kick* in the Indicator model is due to the addition of the indicator variable as well. This can be explained by taking a look at the distributions of *Distance* and *Direction* of *Goal Left* between the Indicator model and the GNBC (Table 6.14). In the GNBC, if the ball is not seen but the left goal is located slightly to the left of and close to the agent, then the agent is going to *Kick*. Whereas, the Indicator model will always *Turn* if the ball is not seen, regardless of any other information.

Model	**P(Goal Left Direction|Action=Kick)**	**P(Goal Left Distance|Action=Kick)**
Indicator	$\mathcal{N}(0, 0.01)$	$\mathcal{N}(0, 0.01)$
GNBC	$\mathcal{N}(-1.19, 100)$	$\mathcal{N}(0.61, 100)$

Table 6.14: The probabilities of the *Goal Left* given *Kick* for the Indicator and GNBC models, given that *BALLSEEN*=False

The CBR agent has similar F1 scores for *Dash*, but has lower F1 scores for *Turn* and *Kick*. As hypothesized, the CBR agent's results are most likely caused due to the class imbalance. The Indicator model is able to outperform the CBR approach because there is separation in the distributions of the features per action

as shown in Figure 6.11.

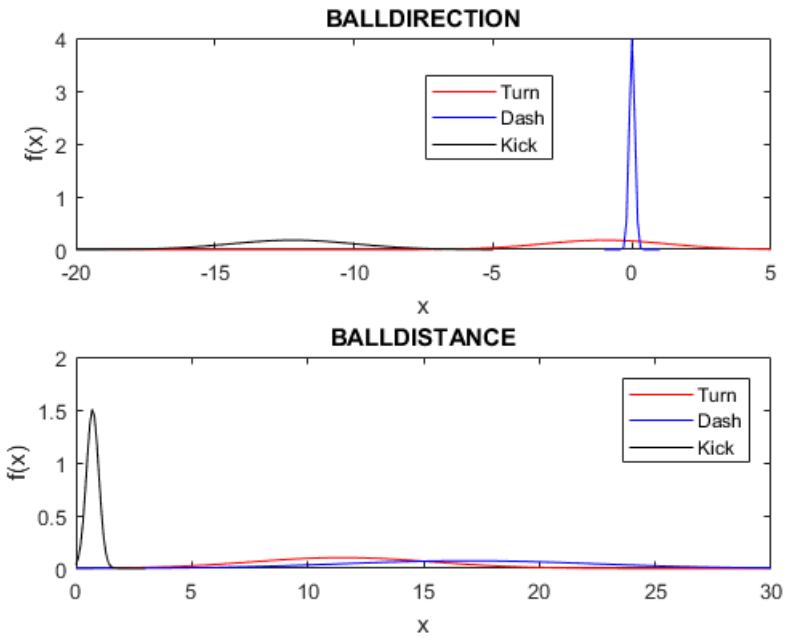


Figure 6.11: Distribution of the learned parameters of the Indicator model
Distribution for *BALLDIRECTION* and *BALLDISTANCE*, when *BALLSEEN* is **TRUE**

The distribution for *BALLDIRECTION* when the Action is *Dash* and *BALLSEEN* is True is centered around 0 with a very small variance, where as for *Turn* and *BALLSEEN* is True the distribution is centered around 0, but has a large variance. The *Kick* action only occurs when the ball is slightly behind the player, since the agent generally has to turn around to find the goal before it kicks (recall that the player can see the ball behind it, if the ball is within 5 units of distance of the player). Similarly, for *BALLDISTANCE* the *Kick* action occurs only when the player is very close to the ball. These all align with the behavior of Krislet.

It is important that the *Kick* action be performed at the correct moment because without it, goals cannot be scored. However, without being able to get to the ball, being able to *Kick* is not very important. Therefore, all three actions are important. We deployed each agent in the environment to determine if the performance testing results are reflected in the behavior during game play. The qualitative results show that the CBR agent is able to turn and locate the ball and dash towards it. It is able to kick, but not in the direction of the goal. It kicks as soon as it comes close to the ball, even if it cannot see the goal, which leads to the ball being kicked out of bounds quite often. The GNBC agent has similar behavior but has a tendency to dash off the map if it cannot see the ball, which is expected behavior. The Indicator model has very similar behavior to the Krislet expert. It is able to turn and locate the ball, and kick in the direction of the goal. Interestingly, even though the CBR agent has slightly lower results than the GNBC, it is still able to perform better during game play. This shows that while the testing results provide certain information about the performance between the models, the qualitative results can be as important.

One thing to note is that the Indicator model is built specifically for the RoboCup domain, and its ability to generalize to other domains is a question that is yet to be answered.

6.4 Conclusion: Comparative Testing in the Continuous Domain

In this section the BNs and DBNs were modified to function in the fully and partially observable continuous domains and were compared to the reactive retrieval and TB approach in Cartpole V0, obstacle avoidance and RoboCup domains. The conclusions are as follows:

- In the Cartpole V0 domain, the reactive retrieval approach was able to overcome the overlapping nature of the dataset and outperform the GNBC learner.
- In the obstacle avoidance domain, both the TB and the DGBN learner had trouble performing the action that was performed based on the internal state. We were able to show that a DGBN learner with handcrafted CPDs would perform very well.
- In the RoboCup domain, the proposed Indicator model was able to outperform the standard Gaussian Naive Bayes Classifier model as well as the CBR approach in learning and imitating Krislet behavior. During game play, the Indicator model performance was similar to the expert behavior, the CBR model was able to turn and dash towards the ball, but was unable to kick the ball in the direction of the goal and the GNBC model, as predicted, dashed off the map when it couldn't see the ball.

The next section will summarize the contributions of this thesis, provide rec-

ommendations based on the contributions and present some avenues for future work.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The main objective of this thesis was to compare the two main approaches to LfO: CBR and PGM, and decide which approach was the most promising to build on moving forward. In order to do this, a unified framework which contained both approaches was built. This way the two approaches had access to the same domains, and could be compared based on the same performance measures. The framework was designed in such a way that it was able to handle testing both state-based and reactive behavior. The state-based testing methodology required designing the framework in such a way that the cases contained the history of the environment state-action pairs. This allowed for using the two state of the art state-based learning methods: DBN and TB, and allowed for the creation of a new state-based retrieval method.

Once this was implemented, preliminary testing was performed in the domains that the PGM and CBR approaches were initially tested in - the discrete RoboCup and the discrete vacuum cleaner domain. These preliminary results helped identify some areas in which there was opportunity to immediately improve the PGM and CBR approaches, in order to gain an understanding of which approach is the most promising. The following are the areas in which improvements were made, and where a comparison of the two approaches was performed in the continuous domain:

- **Comparison of Naive Bayes Classifier to Ontañón's Proposed Model** - The proposed model presented in Ontañón's work was compared to a Naive Bayes classifier in the Vacuum cleaner domain. The original maps that were used to test the learned behavior were symmetrical and the behavior of the WallFollower agent was to travel in a square. Therefore, during the preliminary testing, the two approaches performed similarly. However, once maps that required the expert to travel in an asymmetric manner were used, the results showed that the Naive Bayes Classifier was able to handle unseen environment states better than the proposed model.

- **Comparison of New Similarity Metrics with Temporal Backtracking** - The TB algorithm was compared to the state-based retrieval method with new similarity metrics - *n-ordered*, *n-unordered* and *weighted* in the discrete vacuum cleaner domain. We showed that we could come up with expert behavior where the TB algorithm would perform poorly compared

to the state-based retrieval method with the proposed metrics. We created and tested these experts in custom maps, as well as in the original maps presented in Ontañón's work. We were able to show that, currently, there is no one-size fits all algorithm for learning state-based behavior using CBR.

- **Comparison of Bayesian Networks and Case-Based Reasoning in the Fully Observable Continuous Domain with Reactive Behavior** - The continuous BN was compared to reactive retrieval in the Cartpole V0 domain. The reactive retrieval method was able to perform very well compared to the BN method. The reason for the under performance of the BN was the overlapping nature of the distributions of the features. If the features in the query lie in between the two peaks of the two distributions for the two actions, the BN model has difficulty in determining which distribution the feature belongs to. Therefore, it can be said that in this kind of domain, the BN model will have difficulty learning the behavior.

- **Comparison of Dynamic Bayesian Networks and Temporal Backtracking in the Fully Observable Continuous Domain with State-Based Behavior** - The continuous DBN model was compared to TB in the obstacle avoidance domain. Both the algorithms were unable to learn the proper behavior of the left-right toggle expert. Specifically, they were unable to learn when to turn left vs when to turn right. The reason for the DBN learner under-performing was because of the incorrectly learned CPDs. The TB algorithm has trouble learning because it gives equal weight to each past

action whereas it should focus only on the *Left* and *Right* actions. A DBN learner with handcrafted CPDs was designed and used in the testing. We showed that it could reproduce the behavior with 99% accuracy and 97% average F1 score. This showed that if the correct parameters are learned, the DBN learner could excel in this domain.

- **Comparison of Bayesian Networks and Case-Based Reasoning in the Partially Observable Continuous Domain with Reactive Behavior** - The Indicator model has a binary node depicting whether the object was seen or not as the parent to the action node. In the model, the binary node is connected to the action node and is a parent of the continuous node representing the direction or distance to the object. The CBR with reactive retrieval performed the worst in comparison to the Indicator and GNBC models, by having high F1 scores for *Turn* and *Dash* but a lower results for *Kick*. The Indicator model had higher F1 scores for all the Actions, and was able to outperform the GNBC model. During game play, the Indicator model was able to perform the expert behavior as expected, the CBR model was able to locate the ball, dash towards it, but had difficulty kicking at the correct moment. The GNBC model as expected often traveled off the map when it couldn't see the ball. This shows that while performance measures are important, having qualitative results can provide valuable information.

7.2 Recommendation

After the unification work, the improvements that were made, and the experiments that were run we can make the following recommendations of which approach seems the most promising for investigating further. They are broken down by the environmental characteristics, and types of behavior (Note that these recommendations need to be further tested in the environments and benchmarks with similar properties).

- Discrete domain:
  - Reactive (PGM) - The recommendation for the Bayesian network model to be used in this domain is the Naive Bayes Classifier, over Ontañón's proposed model. This is due to the Naive Bayes Classifiers ability to perform inference when dealing with potentially unseen data. This advantage is prevalent in a domain where the CPT of the Action node that is created is very large. In domains where the CPT is small, the advantage of the Naive Bayes Classifier may be reduced.
  - State-Based (CBR) - The recommendation for when to use TB vs the state-based retrieval is dependent on the expert behavior that is being learned. As was determined during testing, there is no one-size fits all algorithm that can learn any behavior.
- Continuous domain:
  - Fully observable (Reactive) - In domains where there is overlap of

the feature distributions like Cartpole V0, CBR is the recommended approach.

- Fully observable (State-Based) - For types of state-based behavior where only specific actions in the past are important and domains where there is little overlap in the feature distributions per action, a DBN-based learner is the best approach (given it can learn the correct CPDs).
- Partially Observable (Reactive) - The addition of the indicator variables to account for the partially observable nature, and thereby having separate distributions for the observed and unobserved data, is the reason for the improved performance of the Indicator model. Based on the qualitative game play results, and the testing results, the Indicator model approach is the recommended approach.

This work has investigated the introduction of bias into two different streams of learning techniques - Bayesian Networks in the PGM domain, and state-based retrieval in CBR. It has compared CBR and PGM learning techniques across multiple domains with different environment features, and characterized the behavior. However, there are some limitations and issues that have been identified which should be investigated further and are described in the next section.

7.3 Future Work

We propose the following items as areas of investigation:

- **Comparison of the Naive Bayes Classifier and Ontañón's classifier in Noisy, Class Imbalanced Domains** - There is merit in investigating discrete domains that have more noise in the data, as well as have a class imbalance to categorize the behavior of the generative Naive Bayes model and Ontañón's proposed methods in the context of LfO. The discretized RoboCup domain would be well suited to this, since there is noise in the dataset due to the random nature of the environment, and there is a class imbalance because the *Kick* action only occurs a small percentage of the time.

- **Investigating the Use of the New Similarity Metrics in the Continuous Domain** - The *Run* similarity metrics were used in the discrete domain because it was easy to identify why they would not perform well. It was also useful in being able to come up with behavior that would cause certain metrics to fail. The next steps would be to investigate how well they perform learning in a more complex domain.

- **Investigate why the Expectation Maximization Algorithm is Learning Incorrect Conditional Probability Tables for the Internal States in the Dynamic Gaussian Bayesian Network** - There seems to be an issue with learning the correct CPDs for the continuous DBNs using the EM algorithm, in the BNET toolbox, as seen in the obstacle avoidance domains. Working with smaller problems and a smaller dataset and potentially calculating the results manually would be useful in identifying the reason for the

EM algorithm not being able to learn the correct CPDs. The reason could be that the algorithm is stuck at a local minima or saddle point. Introducing an annealing or a momentum into the algorithm, as well as letting perform more iterations may help with finding the correct global minima. Another reason could be that the algorithm has difficulty dealing with features that have a large range, as the values of the SONAR feature ranges from 0 to 20. Mapping the features between 0 and 1 could solve this, however as seen in the Cartpole domain, the toolbox has difficulty with learning very small numbers.

- **Investigate how reducing Class Imbalance in the RoboCup Domain Affects the Learning Model's Performance** - Currently, the large class imbalance seems to be affecting the performance of the CBR and GNBC model's ability to *Kick* the ball at the correct time. There is merit investigating how filtering the dataset either through a sampling algorithm or a clustering algorithm would improve the performance during testing and during game play. Would this improve the agent's ability to *Kick*?

- **Investigate POMDP as a Method for Learning State-Based Behavior in the Partially Observable Domain** - The structures used in the partially observable domain were designed specifically for the RoboCup domain. There is still a requirement to investigate how well they generalize to other domains. One of the techniques used to deal with partially observable domains are Partially Observable Markov Decision Processes (POMDPS). The use

of POMDPs in the context of LfO would be an area to investigate.

- **Long Short Term Memory Neural Networks for Learning State-Based Behavior** - One of the other areas that is currently gaining traction in the machine learning world is neural networks. Recurrent Neural Networks (RNN) specifically with the Long-Short Term Memory (LSTM) nodes are being used in Reinforcement learning for state-based behavior. There is no doubt that the use of LSTMs for LfO in learning state-based behavior is a low hanging fruit that should be investigated. It should be benchmarked against the techniques presented in this paper, to determine if given the size of the dataset, it can learn the behavior well.

References

[1] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A Survey of Robot Learning from Demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.

[2] Michael W. Floyd, Babak Esfandiari, and Kevin Lam. A Case-Based Reasoning Approach to Imitating RoboCup Players. *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference, Florida, USA*, pages 251–256, 2008.

[3] Michael W. Floyd and Santiago Ontañón. A Comparison of Case Acquisition Strategies for Learning from Observations of State-Based Experts. *Proceedings of the Twenty-Sixth International Florida Artificial Intelligence Research Society Conference, Florida, USA*, pages 387–392, 2013.

[4] Santiago Ontañón, José L. Montaña, and Avelino J. Gonzalez. A Dynamic-Bayesian Network Framework for Modeling and Evaluating Learning from Observation. *Expert Systems with Applications*, 41(11):5212–5226, 2014.

[5] Tom M. Mitchell. *The Need for Biases in Learning Generalizations*. Department of Computer Science, Rutgers University, CBM-TR-117, 1980.

[6] Michael W. Floyd and Babak Esfandiari. Learning State-Based Behaviour using Temporally Related Cases. *Proceedings of the Sixteenth UK Workshop on Case-Based Reasoning, Cambrdige, United Kingdom*, 829:9–11, 2011.

[7] Michael W. Floyd and Babak Esfandiari. A Case-Based Reasoning Framework for Developing Agents Using Learning by Observation. *Twenty-Third IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 531–538, 2011.

[8] Caleb Chan. *Contributions to Techniques for Learning Non-reactive Behaviour from Observation*. Master's thesis, Carleton University, 2015.

[9] Amrik Gunaratne, Babak Esfandiari, and Caleb Chan. Towards a Framework for Testing Learning from Observation of State-Based Agents. *AAAI Spring Symposium Series, California, USA*, SS-17-06:499–505, 2017.

[10] Kevin P. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, University of California, Berkeley, 2002.

[11] OpenAI Gym. Cartpole V0 - Classic Control Problems. `https://gym.openai.com/envs/CartPole-v0/`, 2017 (accessed December 17, 2017).

[12] Michael Woolridge. *An Introduction to Multiagent Systems*. John Wiley and Sons, LTD, West Sussex, England, 2002.

[13] Daniel Polani, Minoru Asada, Kitano Hiroaki, and Manuela M. Veloso. Welcome to RoboCup Soccer. http://www.robocup.org/domains/1, 2016 (accessed August 7, 2016).

[14] Mao Chen, Klaus Dorer, Ehsan Foroughi, and Fredrik Heintz. Users Manual - RoboCup Soccer Server for Soccer Server Version 7.07 and later. https://sourceforge.net/project/sserver/files/rcssmanual/manual-7.08.1/manual.pdf/download, 2003 (accessed August 7 2016).

[15] S. L. Lauritzen. *Graphical Models*. Oxford University Press, Oxford, UK, 1996.

[16] D. Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.

[17] Janet L. Kolodner. An Introduction to Case-Based Reasoning. *Artificial Intelligence Review*, 6(1):3–34, 1992.

[18] Agnar Aamodt and Enric Plaza. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, 7(1):39–59, 1994.

[19] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A Bayesian Approach to Filtering Junk E-Mail. *Learning for Text Categorization: Papers from the AAAI Workshop*, WS-98-05(Cohen):55–62, 1998.

[20] Kevin Murphy. A Variational Approximation for Bayesian Networks with Discrete and Continuous Latent Variables. *Proceedings of the Fifteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–99)*, pages 457–466, 1999.

[21] Zoubin Ghahramani and Michael I. Jordan. Supervised learning from Incomplete Data via an EM Approach. *Advances in Neural Information Processing Systems*, 6:120–127, 1994.