Episodic Memory for Cognitive Robots in Dynamic, Unstructured Environments

Colm Flanagan

February 17, 2022

Supervisor: Prof. Claude Sammut
Co-supervisor: Dr. Bernhard Hengst

School of Computer Science Engineering
Faculty of Engineering
UNSW

UNSW
SYDNEY

Thesis Title

Episodic Memory for Cognitive Robots in Dynamic, Unstructured Environments

Thesis Abstract

Elements from cognitive psychology have been applied in a variety of ways to artificial intelligence. One of the lesser studied areas is in how episodic memory can assist learning in cognitive robots. In this dissertation, we investigate how episodic memories can assist a cognitive robot in learning which behaviours are suited to different co ntexts. We demonstrate the learning system in a domestic robot designed to assist human occupants of a house.

People are generally good at anticipating the intentions of others. When around people that we are familiar with, we can predict what they are likely to do next, based on what we have observed them doing before. Our ability to record and recall different types of events that we know are relevant to those types of events is one reason our cognition is so powerful. For a robot to assist rather than hinder a person, artificial agents too require this functionality.

This work makes three main contributions. Since episodic memory requires context, we first propose a novel approach to segmenting a metric map into a collection of ro oms and corridors. Our approach is based on identifying critical points on a Generalised Voronoi Diagram and creating regions around these critical points. Our results s how state of the art accuracy with 98\% precision and 96\% recall.

Our second contribution is our approach to event recall in episodic memory. We take a novel approach in which events in memory are typed and a unique recall policy is learned for each type of event. These policies are learned incrementally, using only information presented to the agent and without any need to take that agent off line. Ri pple Down Rules provide a suitable learning mechanism. Our results show that when trained appropriately we achieve a near perfect recall of episodes that match to an observation.

Finally we propose a novel approach to how recall policies are trained. Commonly an RDR policy is trained using a human guide where the instructor has the option to di scard information that is irrelevant to the situation. However, we show that by using Inductive Logic Programming it is possible to train a recall policy for a given type of e vent after only a few observations of that type of event.

UNSW is supportive of candidates publishing their research results during their candidature as detailed in the UNSW Thesis Examination Procedure.

Publications can be used in the candidate's thesis in lieu of a Chapter provided:

- The candidate contributed **greater than 50%** of the content in the publication and are the "primary author", i.e. they were responsible primarily for the planning, execution and preparation of the work for publication.
- The candidate has obtained approval to include the publication in their thesis in lieu of a Chapter from their Supervisor and Postgraduate Coordinator.
- The publication is not subject to any obligations or contractual agreements with a third party that would constrain its inclusion in the thesis.

☑ The candidate has declared that **some of the work described in their thesis has been published and has been documented in the relevant Chapters with acknowledgement**.

A short statement on where this work appears in the thesis and how this work is acknowledged within chapter/s:

> Chapter 3 on topological mapping was accepted for publication at the 2020 Australasian Conference on Robotics and Autonomation.
>
> Chapter 4 on event recall policies was accepted as a poster presentation at the 2020 Advances in Cognitive Systems conference. Both have been cited and acknowledged at the beginning of both chapters.

Candidate's Declaration

I declare that I have complied with the Thesis Examination Procedure.

Acknowledgements

I would like to thank my supervisor Prof. Claude Sammut whose expertise helped to formulate a topic of research and establish the relevant methodologies that have resulted in the work presented here today. I would also like to thank Dr. David Rajaratnam who has provided consistent support and feedback and who co-authored the work on topological mapping that is an integral part of this thesis. Without this feedback and the many a Friday evening drink it would have been a decidedly different experience!

Prof. Paul Compton has provided invaluable feedback on the research relating to Ripple Down Rules and Dr. Bernhard Hengst was very helpful in providing feedback on a number of the chapters in this dissertation on I would like to thank them for this.

Considerable work has been done by the UNSW@Home robotics team. Without this work it would not have been possible to test and evaluate a lot of the research that we have conducted over the course of this dissertation. In particular I would like to thank Germán Castro, Peter Kidd, Oliver Richards, Stathi Weir and Joshua Goncalves.

Finally I would like to thank my parents Peter and Molly Flanagan for continually supporting me over the last number of years and to my Grandfather Séamus Ó hÚallacháin who has provided considerable feedback that has helped develop my writing style throughout this PhD.

Algorithm Structure

For the purpose of this thesis we have our own format for algorithms. The main body of each algorithm contains numbered lines, with the initial call to the algorithm appearing directly before line 1. In the header, we have inputs and comments. The numbered lines next to each comment correspond to the line in the algorithm that the comment refers to. The inputs to the algorithm are defined at the very top. An example is shown below.

Calls to other methods will be highlighted in bold and class or structure member variables will be in italics succeeding a period.

Algorithm 1 algorithmExample: An example of our algorithm structure

Input: someInput
Comments:
1: *This is a comment referring to line 1 of this algorithm*
2: *Call some other algorithm named callSomeFunction*
3: *Here we access the class member variable, var*
algorithmExample(*someInput*)
1: line1 : See comment above for clarity
2: **callSomeFunction**()
3: instanceOfClass.*var*

Abstract

Elements from cognitive psychology have been applied in a variety of ways to artificial intelligence. One of the lesser studied areas is in how episodic memory can assist learning in cognitive robots. In this dissertation, we investigate how episodic memories can assist a cognitive robot in learning which behaviours are suited to different contexts. We demonstrate the learning system in a domestic robot designed to assist human occupants of a house.

People are generally good at anticipating the intentions of others. When around people that we are familiar with, we can predict what they are likely to do next, based on what we have observed them doing before. Our ability to record and recall different types of events that we know are relevant to those types of events is one reason our cognition is so powerful. For a robot to assist rather than hinder a person, artificial agents too require this functionality.

This work makes three main contributions. Since episodic memory requires context, we first propose a novel approach to segmenting a metric map into a collection of rooms and corridors. Our approach is based on identifying critical points on a Generalised Voronoi Diagram and creating regions around these critical points. Our results show state of the art accuracy with 98% precision and 96% recall.

Our second contribution is our approach to event recall in episodic memory. We take a novel approach in which events in memory are typed and a unique recall policy is learned for each type of event. These policies are learned incrementally, using only information presented to the agent and without any need to take that agent off line. Ripple Down Rules provide a suitable learning mechanism. Our results show that when trained appropriately we achieve a near perfect recall of episodes that match to an observation.

Finally we propose a novel approach to how recall policies are trained. Commonly an RDR policy is trained using a human guide where the instructor has the option to discard information that is irrelevant to the situation. However, we show that by using Inductive Logic Programming it is possible to train a recall policy for a given type of event after only a few observations of that type of event.

Contents

Chapter 1

Introduction

1.1 Overview of the Research

The term *episodic memory* was first introduced by Tulving in 1972 [1]. In this work he distinguishes between episodic and semantic memory which he states are both separate components of declarative memory. Episodic memory is a collection of events obtained through personal experience with both a spatial and temporal relevance, providing a context for events. Whereas, semantic memories are decontextualised, that is knowledge that is independent of time and location. The aim of this work is to design and evaluate an episodic memory architecture from the perspective of artificial intelligence.

Initially, each event will have a time and a location associated with it, as well as the action that created the event. However, it is not necessarily the case that that information is relevant to the event. To assist in differentiating between relevant and irrelevant information, each event is given a *type*, where one type of event may have information that is different to another. The concept of typed events is not mentioned in psychology literature on episodic memory.

Although episodic memory is a component of declarative memory that is agreed to exist, there is no good understanding of how episodic memories

are created, stored, represented or recalled. Therefore, at no point in this dissertation do we claim to replicate human-like episodic memory or claim that our approach more closely resembles human episodic memory than other approaches. Rather, this research aims to design a system that has the same functional advantages of human episodic memory, without trying to mimic a human-like memory mechanism.

This research makes three significant contributions. We have already noted that episodic memories are spatio-temporally relevant memories. We present a method for discovering qualitative descriptions of the location of objects in the environment. For example, if a person is asked where they went on holidays they will say the name of the country or city rather than geographical co-ordinates. That is locations are grounded in symbols that can be used to communicate information to other agents, and can be used by a task-level planner. As our research takes place in a domestic environment, we need a way to segment a metric map of the environment into rooms and corridors. In Chapter 3 we propose a novel approach to topological mapping that achieves state of the art accuracy when evaluated using a precision and recall metric as proposed by Bormann *et al* [2].

The second significant contribution of our research is in how events are recalled from memory. By distinguishing between different types of events we are able to customise unique recall policies for each type of event. When recalling events, many previous approaches use a nearest neighbour technique to match an observation to an event in memory. If the goals that the agent is expected to achieve are known in advance and a finite amount of information can be shown to the agent then this approach works very effectively. However, in a partially observable, unstructured environment such as a domestic environment, this may not be the case and unexpected things can happen. An agent may need to perform an unknown number of tasks and the types of information presented to the robot might also not be known in advance. Thus, we require a recall policy that is able to capture the different nature of different types of events. By separating events into different types and assigning each type its own recall policy we can effectively recall events from memory. Our results show that with this kind of

recall policy, after the system has been sufficiently trained, we can achieve almost perfect recall. By this we mean that a recall policy can correctly classify almost all instances of a sub-class of event and correctly discard almost all instances of other sub-classes of events.

Our final contribution is in how our recall policies are trained. By using a form of interactive machine learning, we can train our recall policies using only information that is relevant to that type of event. Each recall policy is represented as a Ripple Down Rule (RDR) [3, 4, 5, 6]. This means that our policies can be trained incrementally and without the need for the agent to be taken offline. Moreover, we learn only using information that is relevant and so we do not waste time collecting unnecessary data.

The evaluations show that with a small number of observations of any given type of event we can train recall policies that are capable of effectively perfect recall. Depending on the length of an event sequence the number of observations required to train a recall policy changes slightly. However, we have found that the average number observations of a type of event needed to train these policies is less than 5.

In the introduction to this thesis we provide the reader with a brief overview of the motivation behind our research and detail each sub-discipline of artificial intelligence that is relevant to our work. In some of these disciplines we make clear and distinct contributions while in others we are using previous work as we believe it to be complimentary to our work or work that we believe might be complementary to extensions to our work.

1.2 Motivation and Significance of the Research

Artificial intelligence is a broad field with many sub-disciplines. Our work focuses on cognitive robots and its applications to robots operating in partially observable, unstructured environments. Specifically, our research focuses on a domestic robot. Domestic environments are of interest because

of their variety. Each domestic environment is slightly different from every other environment, with different layouts, structures and occupants. Within each environment, any number of different types of events might occur and it is important that an agent can learn from and react to these events.

Before continuing we clarify what is meant by an unstructured environment. An unstructured environment is one that typically contains many obstacles, the positions of which are liable to change for any reason. Due to the dynamic nature of such environments, robots cannot rely entirely on having complete knowledge of an environment and must continually make adjustments to account for any changes. Consequently, many decisions are made with a believed rather than known state-of-the-world. For the most part, we evaluate this thesis in domestic environment settings. As domestic environments have most of these properties they are unstructured in nature.

Contrary to research that is focused on training a robot to perform a task, this research is directed towards understanding the context in which the task is performed, and how the context changes the requirements of the task.

We show that by using episodic memory, where events are typed, we can characterise the variety of context information and train the system to recall appropriate memories for a given context. This research does attempt to teach a robot how to perform a task, but rather on understanding and reasoning about different contexts that indicate to an agent when a task should be performed.

To explain why this is significant we present two examples of two different types of events, both of which may occur in typical home environment, but are not connected. An agent should be able to distinguish between the two different types of events and distinguish between the types of information relevant to each type of event. This should enable an agent to effectively recall related past experiences and any behaviour associated with that type of event.

While the environments that we refer to in this dissertation are unstructured, a robot is still equipped with a certain degree of *apriori* knowledge. For example, an agent is endowed with a world knowledge that informs the agent about the last known state of the world, the agent has a map of the environment and a topological map where the environment has been segmented into individual rooms and corridors. For example, in Section 1.2.1, we describe the case where a glass falls off a table. This is a type of event that is typically in a domestic environment. However, in order for the robot to know that the glass has changed state, it needed to know that it was previously on the table. This information is contained within the robot's world model and is an example of *apriori* information.

1.2.1 The Falling Glass Problem

In the following example a robot witnesses a glass falling off the edge of a table. That same robot then witnesses someone cleaning up the glass with a brush. The first observation is the falling glass, the second observation, which occurs at a time soon after the first observation is of someone cleaning up the glass. How these two events are established and connected is covered at a later stage in this dissertation. Here, we demonstrate the types of events that an agent may need to recognise and reason about.

For an agent to recall an event in memory, it must have an understanding of what information is relevant in the event just observed. This is the event type's recall policy which is a Ripple Down Rule [3]. In this example, the robot should learn that the only information relevant to the event is that the glass was observed falling. When recalling this type of event, the agent does not have to consider the time, location or anything else as none of it is relevant.

1.2.2 A Friend Coming to Visit

In the previous example, the agent should discard time and location as being relevant to that type of event, as it doesn't matter what time it is or where it is, a cleanup generally follows.

However, time and location might be relevant for other types of events. Consider when a friend comes to visit on a Monday morning. This might be a ritual, one that both friends adhere to every week. In recalling this event, the robot should not discard the time as it may be relevant. For example, when a friend comes to visit on a Monday, the agent should make the house occupant and their friend a cup of tea. However, if the same friend instead comes to visit on a Friday evening, the required behaviour might be to fetch two beers from the fridge. Thus, the agent should not discard the time of the event in this case.

These examples demonstrate that a single retrieval mechanism or *cue* to recall events from memory is not practical. Hence our motivation to learn recall policies specialised for the type of event. In doing so, if the event has been previously observed, we can recall that event with near perfect accuracy and thus recall any behaviours associated with that type of event. For example, on observing a friend visiting on Monday morning the robot should recall this and, without prompting make two cups of tea.

As we will demonstrate, previous research in this field, such as SOAR [7, 8, 9] or Homem *et al* [10], in case based reasoning, do not address episodic recall in this way, instead attempting to fit single retrieval cues to all cases or events. This approach is well suited to domains where agents have a finite number of goals and actions. Our approach is intended to work in unstructured environments where the types of events and behaviours are not known in advance and must be learned.

While the two examples presented above did not include location, it is often relevant to the robot's behaviour in the home. Therefore, our research also requires extensions to be made in topological mapping.

1.3 Overview of Method

In this section we give an overview of the thesis, the methods that we employ and explain how we evaluate each contribution. In Chapter 2 we review the work that is most relevant to our research. Chapter 3 covers our approach to topological mapping. This approach consists of a five-stage pipeline.

Episodic memories need context, in particular, time and location. We assume that the robot has mapped its environment, which is usually done by some form of simultaneous localisation and mapping (SLAM), producing an occupancy grid map. Phase-one of the pipeline involves removing errors from the occupancy grid. Maps generated using SLAM are likely to contain errors and so it is important that these errors are addressed before we attempt to segment the environment into rooms and corridors. We then create a Generalised Voronoi Diagram (GVD) by re-implementing an approach proposed by Lau *et al* [11]. From the GVD, we identify specific nodes that are considered to be critical points. Ideally, these critical points would be doorway points but as will become clear, in cluttered environments, often other points are misinterpreted as being doorway points. Regions are then created around these critical points as detailed in Section 3.2.5. In an ideal environment, these regions would represent entire rooms and corridors, however as we have already stated, clutter in the environment often leads to additional critical points being selected and thus, additional regions being created that are not rooms or corridors. To account for this we merge regions that are part of the same room or corridor.

We evaluate our approach using the precision and recall metric proposed by Bormann *et al* [2]. This is considered the standard metric to evaluate approaches to topological mapping. We achieve state of the art accuracy with 98% precision and 96% recall, which, to the best of our knowledge, are the most accurate results to date.

In Chapter 4, we explain how events are created and recalled from memory. Our hypothesis is that different types of events have different types of infor-

mation that are relevant to them and thus require different recall policies. Recall policies are learned incrementally and, without any need to take the agent offline. We use Ripple Down Rules (RDR) [3, 4, 5, 6] to learn recall policies as they can be learned incrementally through interaction with the human occupant, who acts as a trainer.

We construct individual recall policies for each type of event. This means that when an agent attempts to recall an event from memory, it does so considering only the information that has proven to be relevant to that type of event. A recall policy is attached to the generic representation of an event, where its purpose is to establish if an observation is an instance of that type of event.

An event is created when the agent observes a sequence of related actions, which we refer to as a *critical* action. A *critical* action is an action that has a significant effect on the environment. Depending on the domain, what is considered significant can change. It is also likely that there are several states that are considered to be significant. For example, having broken glass on the floor is a significant state because a person might hurt themselves. Therefore, an action whose effect is that broken glass is on the floor is considered a *critical* action and thus, a new event is created.

On creating a new event, we use the policies assigned to the event types already in memory to try to establish if what we have observed is a new type of event or if it is an instance of an event type already in memory and consequently recall that type of event from memory. If the observation is a new type of event then we create a new event type.

Figure 1.1 provides a very high-level overview of the individual components of this thesis that are most relevant. Components highlighted in blue are novel contributions of this thesis and respectively make up a chapter each.

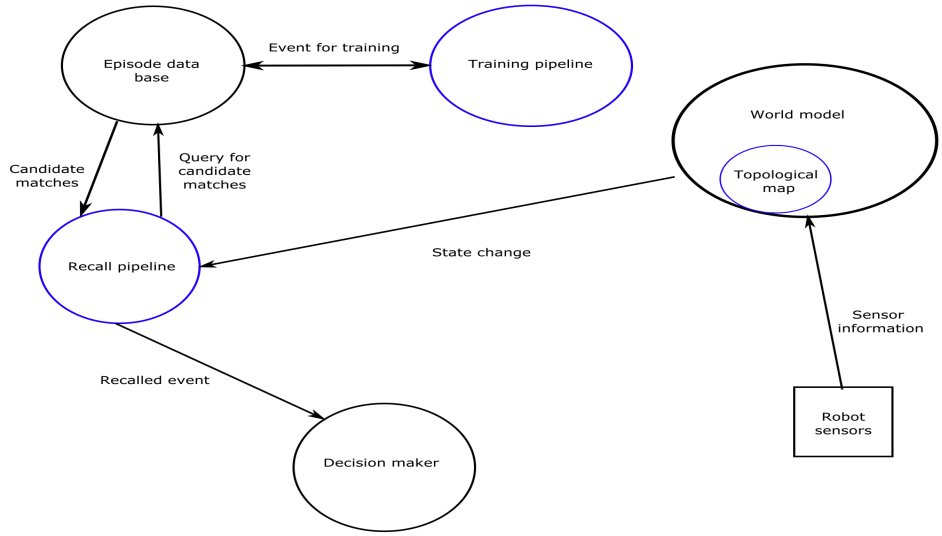


Figure 1.1: The components of the thesis. Components are represented by circles and connections between components are shown through arrows. A bi-directional arrow indicates a service call. For example, the when an event is written to the episodic data base it is then sent to the training module, the policy for that type of event is updated and sent back to the data base. Components in blue are novel contributions. If a component is a sub-module of another component then it is shown inside that component. For example, a topological map is a sub-module of a world model.

Our evaluation is in a simulated domain. There are a number of reasons for choosing to evaluate in simulation.. The main reason is that we can collect much larger data sets than is possible using the physical robot. In our simulation, we collect a database of different types of events. Events are represented by frames that contain information including, for example, time and location, which are also represented as frames. Each generic frame is given its own matching or recall policy that indicates when a new observation is an instance of that type of frame. Thus, it is not only events that are assigned recall policies but all types of frames.

We collect nearly 50 different types of events and, for each event type, we collect over 300 different examples. For each type of event we synthesise instances of that type and also synthesise instances of the other frame types that each event is constructed from. This will become clear in Section 6.2.

We evaluate event recall as follows. A synthetic data set is constructed from hand-crafted recall policies for each type of frame. The aim is for the learning system to reverse engineer the policies from examples randomly generated from the hand-crafted policies. For each type of event we have approximately 375 synthesised instances of that type. Each instance contains different information, as it is randomly generated, making the data set highly varied. As noted, the purpose of the event recall policy is to determine if an observed event is an instance of a stored event type. Therefore, when evaluating our system, we check that we correctly classify all valid instances of a type of event and correctly reject all instances of other types of events. We do this under varying levels of stress including varying levels of distraction, perception noise and misclassification noise. With correctly trained policies we can achieve almost perfect recall of events from memory. This evaluation and the results are explained in Section 6.2.

Our final contribution is how we train these recall policies. Typically, RDRs are trained manually using a human expert to guide the learning. However, by using Inductive Logic Programming (ILP) we can autonomously train event recall policies. This does come with the caveat that our recall policies cannot be trained as quickly. For example, when manually training recall policies using a human guide, it is possible to train a successful recall policy after only one observation of that type of event. However, this can only be done after many questions are asked by the agent. Thus, it is not a feasible approach to train recall policies and so we must sacrifice some training time in exchange for a more automated approach. We evaluate the method by checking how many observations of a given type of event are required to train a policy that we know to be correct. One should remember that we evaluate our approach to episodic recall by generating training examples from hand-crafted policies that we know to be correct. Our objective is to train a new policy that matches to the hand-crafted one. We further evaluate our training in the same way that we evaluate our recall. Namely we test how many training instances are required before the policy achieves acceptable recall, $< 5\%$, under different levels of stress.

Different types of events have different complexities and some events may

be part of an event sequence, that is when multiple events are connected to each other. For example, a friend arriving is one event and it is succeeded by the agent making two cups of tea. As one would expect, events that are part of a long event sequence take longer to train. Thus, in our evaluation we present the results for a number of event sequence lengths. To summarise however, we found that single events required only an average of 2.03 observations after the initial observation to correctly induce a recall policy.

Chapters 34 and 5 describe the above methods in detail. Our evaluation and results are presented in Chapter 6, with our concluding remarks in Chapter 7.

1.4 Conclusion

Throughout the introduction we have presented an overview of the research conducted in this dissertation. We present two examples of use cases where the research objective is clearly visible. The evaluation details significantly more use cases however the two examples presented were chosen due to the fact that they are decidedly different in nature showing how the theory is generalisable and can be applied to a diverse range of situations that a robot may encounter daily.

We present a high level account of some of the crucial components necessary to the research and justify the necessity of each. To the fields that we make an explicit contribution we detail these contributions in Chapters 3 to 5.

Episodic memory presents a range of challenges for artificial intelligence. We believe that the three most significant of these challenges relate to how an agent can recall an event, how an agent can create and represent an event and how an agent can forget events. The latter of these this dissertation does not focus on although we recognise the importance of it. Agents will likely observe hundreds if not thousands of different types of events throughout their life. An agents memory however has a limited

capacity for space. Thus, it is important to have some means by which less relevant types of events can be removed from memory and free up space for types of events that are deemed more pertinent.

We believe that this has the potential to be an interesting extension to this research however we do not focus on it here. Instead the research in this dissertation primarily focuses on topological mapping and in a system that can acquire domain specific knowledge for each type of event that the agent observes and, in doing so it can create unique recall policies for each type of event so that they can be correctly recalled from memory on observing an instance of that type of event. Our main contributions are as follows:

1. We first propose a novel approach to segmenting a metric map into a collection of rooms and corridors. Our approach is based on identifying critical points on a Generalised Voronoi Diagram and creating regions around these critical points. Other approaches have relied on identifying critical points to segment an environment, however, they often use arbitrary, hard coded metrics for determining what a critical point is making these approaches less robust to more complex environments. We evaluate our approach using the precision and recall metric proposed by Bormann *et al* [2]. Our results show state of the art accuracy with 98% precision and 96% recall.

2. Our second contribution is our approach to event recall in episodic memory. We take a novel approach in which events in memory are typed and a unique recall policy is learned for each type of event. These policies are learned incrementally, using only information presented to the agent and without any need to take that agent off line. Ripple Down Rules provide a suitable learning mechanism. Our results show that when trained appropriately we achieve a near perfect recall of episodes that match to an observation.

3. Finally we propose a novel approach to how recall policies are trained. Commonly an RDR policy is trained using a human guide where the instructor has the option to discard information that is irrelevant to the situation. However, we show that by using inductive reasoning it

is possible to train a recall policy for a given type of event after only a few observations of that type of event.

This dissertation is organised as follows. In Chapter 2, we cover the related research in this field. In particular we focus on areas of cognitive robotics that endow an agent with episodic memory and other fields of artificial intelligence that also make use of episodic memory. Chapter 3, we present our novel approach to topological mapping which allows an agent to localise itself and other objects at an abstract level. This is essential for episodic memory. Chapter 4 covers our approach to retrieving events from memory using a novel approach that utilises the power of Ripple Down Rules. Our final theory chapter, Chapter 5 details our approach to training recall policies. In Chapters 6 and 7 we explain our evaluation process, present our results and conclude this dissertation.

Chapter 2

Literature Review

In this chapter we review research related to acquiring and recalling episodes in memory as well as some other related topics.

2.1 Episodic Memory and Natural Language

In this section we review some of the work in Natural Language Processing that is most relevant to our research.

An approach to improving conversational dialogue is presented by Kasap *et al*[12]. They examine how episodic memory can be used to improve Human Robot Interaction (HRI). The research investigates how to model episodic memory that supports long-term interaction with people. The specifically embed emotional cues into episodes and use those cues amongst other information such as natural language to generate a belief state based on past experience. Goals are then represented with a Hierarchical Task Network and are realised through a natural language, Finite-State-Machine based dialogue system to produce the correct response to the given context.

Preliminary results of this research are largely qualitative and demonstrate only a simple example of an agent name Eva interacting with two people. They demonstrate their system by showing how Eva responds differently

to the two people who she interacted based on the emotional cues of the first interaction. For example, if one of the subjects was rude to Eva then she was less friendly the second time around.

Our proposed system presents a more generalisable episodic memory implementation. As already clarified, our work distinguishes between different types of events. One can think of a conversation between two people or an agent and a person as a specific type of event. Thus, our system is capable of capturing conversational events as well as other types also.

Xiong *et al* [13] and Kumar *et al* [14] use a dynamic memory network (DMN) to improve question answering of a natural language processor. The system consists of four modules. The first is the input module. An input is taken in and encoded via a recurrent neural network (RNN). This input is a sentence or sentences given in natural language. The second module is the question module, and functions in a very similar manner to the input module. The third is the episodic memory module (EMM) and consists of internal memory and an attention mechanism. The RNN takes the input or question and creates an episode. This episode then updates the EMM. On each input there may be a need for multiple episodes. This is because on the first pass, a pass being an episode created by the RNN, the system may discover that additional information is required. Kumar *et al* give an example to explain when this need for additional information may occur. A question is presented to the system, *where is the football?*. A previous input to the system was, *John put down the football*. Only by taking another pass can the system determine that it needs to find the location of John so that the question can be answered.

Each episode is concatenated into a vector which is then passed to the answer module (AM), which is the final module in the DMN. This module reasons about all of the given inputs and from this produces an answer. This is significant in that it enables the system to extrapolate from indirect information.

Li *et al* [15] and Sutskever *et al* [16] use reinforcement learning to model future reward in NLP. Another method however could be to use the Hid-

den Markov Model (HMM) to predict in advance the direction in which a sentence may be going and to begin constructing a quicker response. A HMM according to Eddy *et al* [17] is "a finite model that describes a probability distribution over an infinite number of possible sequences". In other words, if we know the value of the state at time t-1 then we can make a prediction about future states. The basic ideas behind this are relevant to our research albeit to a different domain. Episodic memory involves using an observation at a given point in time t, to reason about what might have occurred in the past, time t-N or what might occur in the future, time t+N.

We have reviewed the material that is most relevant to this dissertation. In particular, we are interested in the work on dynamic memory networks [13, 14] as these take an alternative look at the way in which episodic memory can be applied. However, natural language processing is not a focal point of our research and so we have only very briefly reviewed some of this work that has similarities with episodic memory concepts.

2.2 Topological Mapping

Topological mapping is relevant to our work since it is necessary for the episodes in memory to have a location that has semantic meaning, rather than coordinates on a map. For example, the robot may need to know if it is the kitchen versus the dining room. Topological maps create regions to which we can attach such labels.

Bormann *et al* [2] provide a detailed and comprehensive survey of methods for room segmentation. They note how, like our approach, the most popular technique for room segmentation is to start from a Generalised Voronoi Diagram (GVD). However, all these techniques make assumptions about room structures, such as rooms being convex. As will become clear in Chapter 3, segmenting an environment often leads to multiple sub-regions being created due to clutter. Sub-regions that are members of the same larger region must be merged. Work reviewed in the survey by Bormann *et al* [2] also identifies the need to merge smaller Voronoi regions that

may have been created as a result of tables and chairs being interpreted as doorways, however, the techniques provided rely on ad-hoc mathematical operations for determining when certain regions should be merged, for example assuming the size of doorways, which is not generalisable to all environments.

Crespo *et al* [18] survey work on using semantic information for robot navigation. The survey focuses on both human-assisted and autonomous techniques for acquiring semantic information from a metric map. The survey details the main reasons why semantic mapping is necessary, specifically focusing on how it relates to high-level reasoning about an environment similar to how people reason about the environment. It also details how semantic or topological mapping improves human-robot interaction, autonomy, localisation and efficiency. We have also identified additional areas where topological mapping is relevant such as in maintaining a world model that can be used for high-level planning and episodic memory.

2.2.1 Building Topological Maps

Some of the earliest work on topological mapping is by Thrun and Bücken [19]. In this work they introduce the concept of critical points for partitioning a metric map into isometric regions of interest. Our work on topological mapping also makes use of the idea of critical points in that we attempt to identify doorway-like points on a metric map. Thrun and Bücken initially generate a Voronoi diagram and then select a number of points within the Voronoi diagram as potential critical points. Points are selected as members of the Voronoi diagram if they are free and equidistant from two occupied cells elsewhere in the map. The metric that they use to define a critical point is that is it in closer proximity to its respective occupied cell than any other Voronoi point within some ϵ-neighbourhood region. The ϵ-neighbourhood area however is hard coded and each region must have only one critical point. This means that there are multiple critical points defined that should not be critical points. For example, along a corridor there may be several critical points when in reality there should

be only two, assuming the corridor is clear.

Our method for defining critical points is more generalisable as it relies on a change in the distance between a Voronoi point and its respective occupied point as we traverse along a path of Voronoi points. This means that a critical point is exactly that. It is a doorway point or some other point of interest created by clutter. In using an ϵ-neighbourhood approach it means that there will inevitably be critical points where there should not be and it is also possible that other valid critical points may be missed. This is because each neighbourhood can and must have only one critical point. This explains why corridors have multiple when they should not have but it also means that actual critical points, like doorway points, may be missed if the distance between a Voronoi cell and that Voronoi cell's occupied point in a doorway is not the smallest distance for that ϵ-neighbourhood.

Beeson *et al* [20] look at another method for detecting points of interest or critical points on a topological map by identifying junction points on the Generalised Voronoi Diagram. These junction points are defined as points on the GVD that fork or branch into multiple other paths.

In an empty environment this would work well. However, as noted, a typical domestic environment contains furniture and so this does not solve the larger scale issue of place detection in the context of rooms and corridors.

As outlined in the survey by Bormann *et al* [2], the most common approach to generating topological maps is to start with a Voronoi diagram. Friedman *et al* [21] generate a Voronoi graph from an occupancy grid and then represent each Voronoi point as a node of a conditional random field. This they refer to as a Voronoi Random Field (VRF). From here they estimate the label of each node in the Voronoi graph. Using the labels of each node they can segment the environment into rooms, hallways and doorways. They use human-labelled training data to learn the parameters for the VRFs. This very nicely segments an environment into a collection of regions. Our work uses a similar GVD however we have simplified the classification problem as we are looking only at identifying doorway points. This also allows for more accurate reconstruction of regions within the

environment as we are less likely to suffer from mis-classification problems.

Wu *et al* [22] propose partitioning a map into a collection of Voronoi cells. These cells however contain no semantic information and the objective is not to determine the structure of an environment but to create a series of navigable zones.

In our work we focus on indoor robots that either have access to good quality sensors that can generate metric maps or that already have access to metric maps. However, some work has been done in the context of field robotics where high quality sensor data or metric maps are not available. Ramaithitima *et al* [23] look at generating topological maps using a swarm of robots. They use the relative position of robots to one another in a swarm to create a GVD. Each robot is equipped with a bearing sensor that can detect that robot's neighbours and a touch sensor for obstacle avoidance. The results show an accurate generation of a GVD in an unknown environment with limited metric information.

2.2.2 Room Segmentation

Liu *et al* [24] look at building a semantic map using a Markov chain that produces samples of probabilistic world models. Here they use a Bayesian framework to infer the most probable world W^*, from a space of possible worlds Ω given the map M, where $W^* \in \Omega$. This probability can be described as

$$W^* = \underset{W \in \Omega}{\operatorname{argmax}}\, p(W|M)$$

To solve this they construct a Markov chain that generates samples W^i from the solution space Ω. The method works very effectively but makes a number of assumptions. The main assumptions made are: rooms must be rectangular, have at least one door, each cell should belong to only one room and walls must have two main orientations. In contrast, our approach makes no assumptions about the environment.

Mura *et al* [25] present an approach that reconstructs complex indoor en-

vironments from cluttered point cloud scans. They extract candidate walls from the point cloud by applying a diffusion process to separate the candidate walls from clutter. Similarly Ochmann *et al* [26] automatically reconstruct building models from point clouds. They focus their work on indoor environments and initially filter the point cloud to remove points that lie outside of the house. They take multiple scans of each room generating a point cloud. The initial scans provide a rough segmentation of the environment into rooms however openings such as doors and windows lead to overlap between scans. Filtering noise from outside the environment gives a rough estimation of points inside the environment. This is then extended by generating potential wall candidates. To separate rooms using wall candidates, they pair wall surface lines that satisfy certain constraints and assume therefore that the two opposite surfaces of the walls separate the rooms.

The work on segmenting an environment using 3-D point clouds as presented by [26] and [25] is further extended by Ambrus *et al* [27]. They perform automatic room segmentation from unstructured 3-D data of indoor environments. They focus on the identification of walls and openings using point cloud data. While they provide accurate representations of environments their work again makes some significant assumptions. Firstly, they define an opening or a door as being an opening in a wall that matches some size criterion. This criterion means that it is less generalisable to some more obscure environments. They also make assumptions that rooms must be mostly convex. This is something that we avoid. Finally, they require a point cloud representation of the environment. This is something that is not easily generated. By focusing with 2-D SLAM generated maps we can be more certain that our approach is usable by almost all standard robots.

An alternative method to segmenting a topological map is presented by Mielle *et al* [28]. In this instance they do not rely on a GVD but rather create a free space image where each pixel in the image is assigned a value based on its nearest obstacle. Neighbouring pixels with the same values are grouped into regions. This results in a slight problem however, as typically areas such as corridors, where each corridor should be a single region are

often not identical in length and so result in multiple regions being created along a corridor. By using a GVD and monitoring the rate of change of distances of the points along the GVD our method is significantly more robust to this type of noise.

They do address this issue by defining a metric upon which two sub-regions should be merged. This metric is loosely based around the idea that two regions that have similar values should still be merged. However, this appears not to be generalisable as they also must define a second metric to merge regions that should be merged but were not covered by the first.

2.2.3 Semantic Mapping

As already noted, topological mapping is the process of segmenting an environment into meaningful regions. This is related to semantic mapping, which is the process of assigning high-level labels to areas of an environment. The key difference between the two is one of emphasis: topological mapping is the process of segmenting an environment. Semantic mapping involves assigning high-level labels to regions of an environment. Furthermore, topological mapping seeks to identify connections between regions, for example to be used in path planning, while semantic mapping is concerned with identifying meanings within a region.

Buschka and Saffiotti [29] propose a two-part method for room detection. They first use segmentation to isolate various spaces in an environment that may represent a room and detect when a robot has entered a new room. Then they use feature extraction to determine the type of that room. Their work relies on the assumption that a room is rectangular. A further extension of this work by Galindo *et al* [30] proposes a multi-hierarchical approach, maintaining two map representations in parallel. They maintain a spatial and a semantic map. The two maps are linked by a concept called anchoring. For example, they would link bed-1 to an image of a bed. Again their work largely focuses on inferring types of rooms given perceptual inputs. The work is relevant to our work however, in that inferring a room's type given perceptual inputs is necessary, but as it is not

the primary focus of our work we will only briefly review some of the other research that has been conducted in this area.

Dellaert and Bruemmer [31] propose an extension to fastSLAM that includes semantic information and Limketkai *et al* [32] add semantic information about objects on a grid using Relational Markov Networks to represent the relationships between objects on a metric map allowing a robot to reason about the map from a spatial perspective.

Rottmann *et al* [33] use a standard classifier to determine objects surrounding a robot and from there infer the room type. Martínez Mozos *et al* [34] extend this work by using an adaptive boosting method to improve performance over more standard classifiers. Other such work that covers adding semantic information to a topological map and that we will review only very briefly is that of Nieto-Granda *et al* [35]. They create a semantic map in cooperation with a human guide. The result is a probabilistic classification of the metric map into a set of labels given by the guide.

There has been some more recent work in this field also. Sünderhauf *et al* [36] use a convolutional network paired with a series of one-vs-all classifiers. This enables the robot to recognise place types and to learn new semantic classes online. Similarly Brucker *et al* [37] extends this work by applying it to a three-dimensional domain.

2.2.4 Maintaining a World Model

As we noted, the objective of generating a topological map is to form the basis for a world model. The world model is a collection of predicates explaining the current known state of the world and those predicates are organised within the topological map. Some earlier work focused on this idea, in particular maintaining and refining incomplete domain models for planning systems, Gil [38]. The paper looks at refining knowledge required for a planning agent through direct interaction with the environment. While we did not explore this work in great detail, it has applications to our research. Most noticeable is how this system handles planning with

incomplete preconditions or effects due to missing information in the domain knowledge. In Section 4.2.4, we note one of the limitations of our work is assuming actions effects are observed immediately after an action observation. This has the potential to miss potentially crucial information and result in a partially complete domain. In an extension to this research, we aim to look at how adopting this model to our system could assist in limiting the effects of this limitation.

Sridharan *et al* [39] explore a similar concept, interactively learning domain knowledge in the context of human-robot collaboration. They use Answer Set Prolog to represent and reason about incomplete knowledge in the domain and combine this reasoning system with advanced probabilistic models to learn actions that can be used to solve subsequent problems.

In recent years a good deal of work has been done in this field. Herrero *et al* [40] propose a relational database model where both objects and rooms have physical and conceptual or semantic properties. They link an object described by its physical properties to that same object described by its conceptual properties. The proposed model is good in that it provides a generalised means to represent objects in a world so that they can be reasoned about on different levels of abstraction. Their use of a relational database however, means that queries can be quite complicated and we feel that a NoSQL database works better for maintaining a world model.

Bazcuoğlu *et al* [41] present a model for maintaining a symbolic knowledge base of action costs for robot manipulation tasks on the assumption that robots can use previously updated probabilistic models for improving their actions. To each action they record both symbolic and sub-symbolic data. They demonstrated their method on a door opening task. The symbolic data contains information such as the action executed, the arm used, the goal and any failures. The sub-symbolic data was the metric data return from the robot's perception, trajectories, joint-states etc. They timestamped both sets of data so they could be matched. They generate positive and negative samples of data and fit a Gaussian Mixture Model (GMM) to both which was used as a predictive function for choosing action parameterisation. The model presented works well as a symbolic database

for storing knowledge regarding the execution of specific tasks. They are not proposing a world model in the sense of an understanding about what is true in the environment. However, they do propose a means for representing a robot's action capabilities symbolically which is relevant to a more generic world model.

Mason and Marthi [42] provide a semantic world model for long-term change detection and semantic querying. Their work primarily focuses on object segmentation, attempting to stay away from *a priori* objects and focus on generic object descriptions such as colour and size. They define an object as a geometrically distinct region above a plane. A plane in this case, is any large flat surface that is not the floor. This technique works well in that it is generalisable to a multitude of environments. However, we feel that in this instance, making use of some assumptions could have drastically improved the performance of the system. For example, they are focusing their work towards domestic robots. This means that there will be a significant collection of generic household objects that a classifier would be very effective at identifying. We are not saying that the semantic description of objects is not beneficial, in fact quite the opposite. However, focusing only on a semantic description can lead to a lot of errors. For example, there are many objects that would have roughly the same size and colour in any given environment. Their technique of detecting if two observed objects are the same is to check if their convex hulls overlap. This will inevitably lead to false detections and could be massively improved by running a classifier as additional confirmation. The querying method is not unique and simply involves querying on fields in a noSQL database. Their technique for determining if change has occurred is to check the convex hulls of two observed objects and see if they are the same at time t and $t^{'}$. This is fine as long as the object is in roughly the same area and will collapse if it is not present at all as you cannot compare the convex hull of an object to empty space. It also means that changes in object locations relative to one another cannot be represented or even behavioural relations in objects cannot be represented as there is no information as to what that object actually is.

Elfrin *et al* [43] proposes a much more complex and complete world model. They identify four main components that a world model should contain:

1. Must contain semantically rich objects for anchoring;
2. Must have good data association - Need to update objects based on measurements;
3. Model based object tracking - Know when an object has changed position;
4. Real time execution.

The main contribution that the paper made was an extended anchoring algorithm using multiple hypothesis tracking. Each object is represented by a symbol. For example, symbol l_1 might represent cup-12. Predicate symbols represent properties of objects and are mapped in the predicate attribute space. An anchor then consists of the following properties:

$$\alpha_a = (l_1, z_i^k, M_a^k)$$

l represents the actual object, k is the time stamp, z are the set of observations at time k and M is the behaviour model of that anchor. The behaviour model probabilistically measures the behaviour of the item at time k. By using this form of behavioural modelling they are able to perform object tracking in the world model. For example, a cup placed on a table may move. The behavioural model of that cup states that it may move around that table and there is a high probability that it may move within a certain region. Therefore, if the cup does move the behavioural model will tell the world model that it is the same cup and not a new one. Their work on maintaining a world model was subsequently extended in [44]. In this work the model is updated based on the expected information gain obtained by the update, the action cost of the update and the task that the robot is performing. In this way the updates are handled using more conceptually relevant information.

2.2.5 Conclusion to Topological Mapping

Building on the existing work in topological and semantic mapping, we identify a number of key issues that we address in this dissertation:

1. **Limiting environmental assumptions**. Almost all of the existing techniques on topological mapping make strong assumptions about the environment that limit their broader applicability. For example, assumptions are made in relation to the shape of regions (e.g., regions must be convex), or the number or size of doors, or whether the environment must be empty. While it is true that all approaches to topological mapping must make some assumptions, our method is able to segment regions without making such significant assumptions that it makes the approach difficult to scale. Instead we make only minor assumptions, for example, a room must have at least one entry point and the environment must be an indoor environment, making it applicable to a very broad range of indoor environments.

2. **Semantic labelling based on topological maps**. Using object recognition to assign semantic labels is crucial to identifying the semantics of rooms and spaces. However, techniques that seek to classify regions only by identifying visible objects cannot accurately determine the structure of those underlying regions, nor how those regions are connected. While [21] and, more recently, [37] are able to produce excellent results they will still encounter the problem of mis-classifying parts of the environment. Our technique of identifying doorway points and creating regions around these, followed by merging regions with the same high-level label, mitigates this problem and determines a room's structure to a much higher degree of accuracy.

3. **Topological mapping as a rich information source for cognitive robots**. Our technique produces five distinct maps, each one containing unique information that is highly relevant to the needs of a cognitive robot. This provides information to the robot at different levels or abstraction that can be applied to a broad range of

cognitive robotic functions. The base two maps are the raw or low-level occupancy grid. These maps provide a robot with a low-level representation of the environment and are useful for localisation and navigation. The third map, a Generalised Voronoi Diagram provides a robot with knowledge about the safest routes through an environment. These routes are not strictly speaking optimal but show paths of maximum clearance, paths where a robot is least likely to damage itself by colliding with obstacles. The fourth and fifth maps allow a robot to reason at a high-level about the spatial structure of the environment. It is at this stage that we make the most significant contribution to the literature.

Some recent work has attempted to address the problems that we have raised also. Hou *et al* [45] propose an approach to topological mapping which is similar to ours in that they attempt to first segment an environment into regions of interest and then merge those regions into room and corridor structures. Their approach however has several methodological flaws. They firstly do not provide any quantitative results to show the success of their methods. They also use open source libraries to detect alpha shapes and merge polygonal regions that are within the same alpha region. This means that they are limited in the shapes of the environments that their method is applicable to. By not using some sort of critical point to measure start points of polygons they are also obliged to split single polygons across more than one region and having to arbitrarily split them.

Much of the work reviewed in this section, particularly the research on topological mapping, is work that is either directly relevant to the field of research and is work that we make significant contributions to, or it is work that is necessary to implement for our research such as the work on maintaining a world model by Elfring *et al* [44, 43]. A world model is a collection of predicates organised within a topological map. It is an essential component of an episodic memory system and this is why we have extensively reviewed all of the relevant research in this field.

2.3 Spatial Reasoning

In addition to knowing the location of objects in a map, it is also necessary to know the spatial relationships between objects and to be able to reason about those relationships. There are two types of spatial reasoning, qualitative (QSR) and metric (MSR). We return to the example of the fallen glass to demonstrate the importance of spatial reasoning in episodic memory. the falling glass. To detect that a glass has fallen the robot must be able to reason about the glass' position relative to the environment. For example, a glass is more likely to fall if it is placed on the edge of a table rather than in the middle.

Thippur *et al*[46] compare QSR and MSR and show how they can improve object classification in a table-top scene. They start by creating a series of point cloud clusters of the objects on the table. In the case of an office table these objects are computer monitors, keyboards or mice for example. They then use a pre-trained classifier to label each object. By then comparing the spatial relations of all of the clusters they try to improve the confidence in the object classification. For example, a keyboard is more likely to be in front of a monitor.

Kunze *et al* [47] focus on how spatial reasoning can help to improve object recognition when a pre-trained classifier selects a point cloud cluster as being two of the possible objects that may be in the scene in question. They refer to the *relatum* as the reference object, or the object against which a spatial relation measurement is made. They refer to the object that a spatial relation is measured with respect to the relatum as the *referent*. So for example, if the relatum is the keyboard and the referent the mouse then we would say that the mouse is found to the right of the keyboard or the referent is found to the right of the relatum.

Metric Spatial Reasoning is when we describe a spatial relation in terms of continuous values, such as the distance between two objects measured in centimetres, or the angle between two objects measured in degrees. With QSR these quantitative representations are replaced by logical relations. To

define a spatial relation using QSR Thippur *et al* [46] proposed 12 different predicates:

- 4 directional relations
    1. behind-of
    2. in-front-of
    3. left-of
    4. right-of
- 3 distance relations
    1. very-close-to
    2. close-to
    3. distant-to
- 3 size relations
    1. shorter-than
    2. narrower-than
    3. thinner-than
- 2 projective relations
    1. overlapping
    2. non-overlapping

They found that, for low perceptual accuracy, both MSR and QSR improved classification. As the perceptual accuracy increased, only MSR showed an improvement. This is because it relies on specific measurements and when a sufficiently large training data set is provided, it can yield very accurate results. They also found that when there is a small training data set, QSR performs better, which is where the advantage of generalisation comes in. Kunze *et al* [47] also found similar results in that spatial reasoning did increase the performance of object classification compared to

just using a pre-trained classifier alone. Hu *et al* [48] proposes something similar in that they attempt to improve object recognition using spatial understanding.

Another interesting area where spatial reasoning is of use is described by Young *et al* [49]. Here QSR is used to train robots by observation. The specific robots in question are simulated in the RoboCup 2D soccer simulation. The robot watches a human attempt to score a goal with another human defending the goal and observes a qualitative action at each time step. It then discards any continuous information so that it is left only with an action class e.g. *kick, run, turn* etc. These actions are then turned into a set of QSR features. These features are defined as a combination of star calculus features, that provide a binary representation of the qualitative directions between entities in the environment, relative to one another and the qualitative trajectory calculus, which provides information about the relative motion. The QSR features are then combined with a pre-trained classifier to make a prediction on what the action is. The robots do not always replicate the humans perfectly due to mis-classifications or the robots mimic mistakes.

Gemignani *et al* [50] demonstrate how QSR can assist in long-term navigation for a robot. They use QSR in conjunction with a representation of the environment to execute commands given verbally by a user. Similar Kumar *et al* [14], they want to be able to reason about the command to extract crucial information such as the command type, object location, reference object location and the spatial relation between the object and reference object. The NLP used however is slightly different from Kumar *et al* [14] in that it relies a more on specification. Kumar *et al* [14] can extract information from abstract commands. In Gemignani *et al*, the agent has a knowledge base (KB) of frames. These frames contain information about the environment in which the agent and user are operating. When a command is given verbally by the user, the agent begins to parse it. It first isolates the command type, which will ordinarily be the verb in question. It places the remaining tokens (parts of the sentence) into an array, which will contain information about the object in question , the reference object

and the spatial relation between them. So if, for example, a command like *"go to the cupboard to the right of the sink"* were given to the robot, then the command type would be *go to*, the object in question would be the *cupboard*, the reference object would be *sink* and the spatial relation would be *right-of*. Once the command is correctly interpreted the agent generates the necessary messages which will allow the command to be executed.

What we would ideally like to do is to enable the spatial reasoner proposed by Gemignani *et al* [50] to work in conjunction with that proposed by Thippur *et al* [46] and Kunze *et al* [47]. So the robot would have a KB of both the map but within that KB it would have "sub-KBs" which would contain information about the more confined locations within a room such as a table-top. This is the representation we believe is most appropriate for episodic memory.

Qualitative Spatial Reasoning has also been used in robot learning. For example, Wolter *et al* [51] use Qualitative Spatial Reasoning to learn manipulation tasks, throwing a piece of paper into a bin. They define a spatial logic that combines qualitative abstraction with linear temporal logic. This allows them to represent relevant information about the learning task. Their architecture contains the following main components:

- A QSL or qualitative task description (throw the paper in the bin);
- A controller;
- A learning element;
- A planning module.

They input training data to the learning element, which trains a forward model (a model which allows the robot to predict event outcome based on its controller inputs). From there, the forward model sends an action to a planner. The planner creates a plan and then executes it. The result of the execution is then compared to the task description. If it does not meet the desired requirements then adjustments are made and the forward model is

retrained. To restrict the amount of learning data to be gathered they also use spatial logic to augment the selection process for gathering data. For example, they only gather training data which leads to the robot throwing the rubbish in front of itself.

They adapt Allen's temporal calculus [52] to represent spatial relations. Allen defines 12 predicates for representing temporal intervals. The general form of such a predicate is r(X, Y) and is known as a qualitative constraint. For example, $O(A, B)$ means that A overlaps B. A scene can be described by a the conjunction of qualitative constraints.

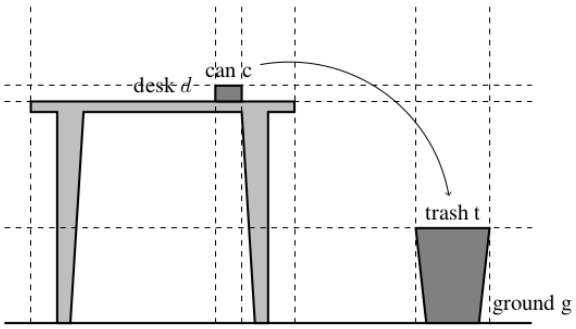


Figure 2.1: An example scene taken from [51]

In Figure 2.1, the state of the world can be represented as a conjunction of qualitative constraints as:

$$ABOVE(c, d) \wedge ABOVE(d, g) \wedge ABOVE(t, g) \wedge$$
$$ABOVE(c, g) \wedge LEFTOF(c, t) \wedge LEFTOF(d, t)$$

They need to be able to reason about both time and space as the act of throwing something is dynamic and therefore its scene description changes over time. For this, they define a Krike structure

$$\langle N, I \rangle$$

.

Here N is a set of time points and I is an interpretation function which assigns truth values to the constraints at various time points. For example, at time 0 the Krike model would interpret the constraint $TOUCHES(b, g)$ to be false or:

$$0 \notin I(TOUCHES(b, g))$$

At time 1 however it would interpret it as touching or

$$1 \in I(TOUCHES(b, g))$$

By using this combination of spatio-temporal logic, they are able to compare the current state of the environment with the goal state to see if the robot is performing the task as required. If not then they can adjust parameters accordingly to try and achieve the goal state. For example, in a situation where the robot is about to throw the paper, it might predict that with an impulse A, the paper will land with position (O_x, O_y). The robot's position is R and the goal position is P. The following relation shows how the parameters are to be updated to train the robot:

$$S_o(R, P) \wedge S_o(P, (O_x, O_y) \rightsquigarrow A^x < A$$

This says that if the robot throws in direction S_o and the paper lands in (O_x, O_y) with respect to the goal P, then the robot has over thrown and the next impulse A^x must be less than the previous impulse. By using this logic, they are able to train the robot to complete manipulation tasks.

So far, all of the work that we have reviewed in spatial reasoning involves ways to qualitatively represent the environment. This is most relevant to our work with regard to maintaining a world model.

Jan Oliver Wallgrün [53] uses spatial reasoning to learn the topological map of an unknown environment. He imagines a topological map as being a collection of hallways and junctions. When the robot arrives at a junction, it observes the hallways that are connected via the junction. All junction observations are made using qualitative cardinal coordinates. For example

a junction observation may be represented as:

$$J_1 = \left\{ southeast(l_1^{J_1}), south(l_2^{J_1}) \right\}$$

This means that the junction J_1 connects two hallways, one running south-east and the other running south. When a new observation is made they update their representations of their un-directed graph

$$G_H = (V_H, E_H)$$

where V_H represents the junctions of the environment and E_H represent the hallways. Using these representations, the system can learn a model of the environment represented in this qualitative format.

Representing an environment in this way is interesting but needs to be extended. Simply viewing an environment as a collection of hallways and junctions provides only a fraction of the information required to reason about the environment. No mention is made of the rooms, the connections between rooms, the spatial relations of the rooms to each other, not to mention the lack of information about any objects that may be present in the environment.

For much of our work, we make use of the region connection calculus by proposed by Cohn and Renz [54], called RCC-8. They define eight Jointly Exhaustive Pairwise Disjoint (JPED) relations, hence the 8 in the RCC-8. For any two regions exactly one of the JPED relations holds. Similar to Allen [52], Cohn and Renz use spatial regions rather than spatial points as a primitive. The regions that they define are as follows:

1. Disconnected - DC;
2. Externally Connected - EC;
3. Partially Overlapping - PO;
4. Equal - EQ;

5. Tangible Proper Part - TPP;
6. Non-Tangible Proper Part - NTPP;
7. Inverse Tangible Proper Part - TPP^{-1};
8. Inverse Non-Tangible Proper Part - $NTPP^{-1}$.

From a visual point of view it looks as follows.

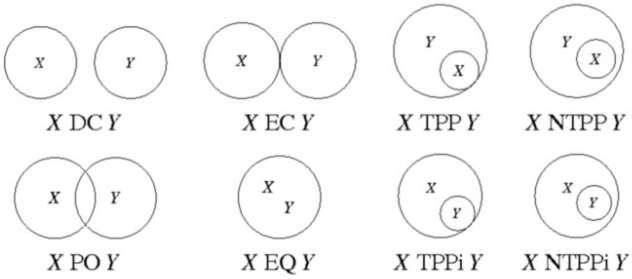


Figure 2.2: The RCC-8 relations visualised

Each of these relations is extendable to an n-dimensional domain.

2.3.1 Conclusion to Spatial Reasoning

We have described some of the most relevant research in spatial reasoning. In this work, we show how spatial reasoning contributes to the efficiency and success of long-term learning through episodic memory. We also show how using spatial reasoning with episodic memory has applications in the field of explainable AI. Looking at one potential use case it is easy to see why. Suppose Paul and Michael always have a meeting in Michael's office at two o'clock on a Wednesday evening and suppose someone were to ask a concierge robot where they might find Paul. The episodic memory would tell the robot that because it is two o'clock on a Wednesday they will be able to find Paul in Michael's office. The ability to reason about the world spatially, i.e. the spatial connectivity of different rooms and their orientations relative to one another, as provided by the topological map, means that a robot can naturally describe the path to Michael's office in a

way that a person can understand. This is similar to work done by Krieg-Brückner *et al* [55] and Matuszek *et al* [56] although they focus mainly on the interpretation of directions rather than providing information.

2.4 Temporal Reasoning

In 1983 Allen [52] proposed a calculus for temporal reasoning. His representation uses a temporal interval as a primitive and describes ways of representing the relationships between these temporal intervals. For this reason the method is often known as Allen's Interval Calculus. Kautz and Ladkin[57] proposed the integration of metric and qualitative temporal reasoning. This is an extension of Allen's interval calculus, in which intervals of time are expressed in both a qualitative and metric manner. They illustrate this with the following example. Suppose there is a time interval that begins at time x and ends at time y. Let that interval be denoted {x, y}. In metric reasoning one might write:

$$2 \leq \{x, y\} < 3$$

This means that the interval is greater than or equal to two and less than three of whatever the relevant units are (seconds, minutes, hours etc). Metric reasoning deals with these exact relations between an interval and time points. Qualitative reasoning however deals with the relations between two time intervals as denoted by the Allen predicates:

1. P = Precedes;
2. M = Meets;
3. O = Overlaps;
4. S = Starts;
5. D = During;

6. F = Finishes.

So if, for example, we have two intervals i and j, $i(P)j$ means that the interval i precedes the interval j, or the latest point in i is before the earliest point in j. The representation proposed by Kautz and Ladkin [57] combines both of these notations into one. They also provide a method for converting from metric-to-Allen and vice versa. The idea of reasoning about specific points in time in a purely metric sense was addressed in a paper by T. Allen *et al* [58]. Here, they design and implement an algorithm for qualitative point based temporal reasoning. The difference between what they propose and what James Allen proposed is that James Allen takes time *intervals* as a primitive whereas T. Allen takes time *points* as a primitive. They developed four reasoners, two basic reasoners using standard graph operations and another which uses ranking of nodes to improve query-answering times. They have a further two reasoners, one based on *series parallel* graphs and the other a re-implementation of the chain based approach proposed by Gerevini and Schubert [59].

2.4.1 Conclusion to Temporal Reasoning

In episodic memory an event happens at a specific time in a specific location. However, for that event to happen, something had to have changed. To recognise that change, we must be able to reason about the world temporally. For example, an action may be defined by preconditions and effects. The effects must succeed the preconditions and to recognise this the agent must apply its knowledge of temporal reasoning.

2.5 Episodic Memory

Episodic memory was first described by Tulving in 1972 [1] when he distinguished between episodic and semantic memory. The definition was extended in 1983 [60]. He conjectured that declarative memory was split

into two parts. Episodic memories were collections of events experienced by a person. They are highly contextualised meaning they all have a time and a location associated with them and, for the memories to be created, something significant has to happen. For example, you may recall loosing your wallet at a shopping centre last Tuesday.

Semantic memories consist of knowledge that is not dependent on context. For example, you may know that Abraham Lincoln was the president of the United States in the 1860s and yet you did not experience the event.

The definition proposed by Tulving is in line with that of Wheeler and Ploran [61] in that episodic memories are personal experiences. They do however provide a little more scientific backing to the theory by presenting studies that show how people can be episodically but not semantically impaired. The studies provide more insight into how declarative memory is split but still, they still do not definitively prove what episodic memory is.

Tulving and Markowitsch claimed in 1988 [62] that episodic memory was a trait unique to people. Griffiths *et al* [63] however believe that other animals exhibit episodic memory systems and that it is not possible to prove that they don't because it is not possible to prove a universal negative. They demonstrate their findings by designing experiments around how Jay birds cache their food for winter. They show that Jay birds can remember where and when they stored certain types of food and will always collect the most perishable food first. They also exhibit the ability to update information regarding the current status of the cache.

Baddeley [64] describe an episodic buffer as being one of the components of working memory. They claim that it is a limited capacity, temporary storage system that is capable of integrating information from a variety of sources and if necessary, manipulating and modifying that information. The module is episodic in that is holds episodes, whereby information is integrated across time and space. Like the work reviewed so far, this is episodic memory in the context of human psychology and not an AI agent. It is an extension of the theory proposed by Tulving in that Baddeley

explicitly addresses the issue of temporary storage. Up to the time of his paper, there had been no mention of how or why certain information was forgotten. This is an essential component of episodic memory in an artificial agent, as computers like people do not have endless storage and managing which memories are to be kept or thrown away is of great significance.

From the perspective of cognitive psychology, what is largely agreed is that episodic memories must have three fundamental components: time, location and an action.

In AI, episodic memory has been studied in various areas. As we saw in section 2.1, episodic memory has been introduced into dynamic memory networks for use in question answering [13, 14]. Similarly in neural networks, Lopez-Paz and Ranzato [65] look at gradient episodic memory for continual learning. Their system learns over a sequence of tasks, noting the limitations of classical neural network approaches to learning. Firstly, the agents are constrained to learning a small number of tasks and suffer from forgetting previously learnt material when a new task presents itself. They also require large numbers of training examples per task and must be shown these examples multiple times. They address these issues by presenting Gradient Episodic Memory (GEM). The GEM can store memories of a given task. These memories can then be used to constrain the loss function of the succeeding task so that it does not increase. In this way, previously learnt information is being used in a new task.

Much of the work that uses episodic memory for learning either in a neural network or through reinforcement learning has focused on the issue of sample inefficiency. Botvinick *et al* [66] look at how through using episodic memory, traditional reinforcement learning techniques can learn much more quickly and with significantly less data. They combine episodic RL with meta-RL which is the concept of learning to learn. The episodic component allows for more efficient sampling by addressing the small step size required during learning while the meta-RL addresses the issue of weak inductive bias. A system with a weak inductive bias will be able to learn a wide range of patterns but will in general be sample inefficient. The episodic memory element allows the learning algorithm to match the current state

to a previously learned behaviour which means that the previously learnt behaviour influences the current policy.

Lin *et al* [67] apply a similar technique to RL called Episodic Memory Deep Q-Networks. The principle behind this work is that the episodic memory can supervise an agent during training. The episodic control remembers experiences during training that returned a high reward and replays these experiences during evaluation. Their results show that they need only 20% of the interactions of standard Deep Q-Networks to achieve state of the art results on the Atari games.

The term episodic memory has become quite widely used in recent years in artificial intelligence. However, the term is used very inconsistently. In a neural network, an episodic memory is some event that returned a positive reward at an earlier stage during training. The definition we assume in this dissertation is very different to those in the neural networks or some other machine learning literature.

Given any observation, our aim is to be able to reconstruct an event of the same type if a similar event has been previously observed and learn an associated action. Thus, our definition of an episodic memory is closer to that of cognitive psychology: it is personal experience (or experience of a robot) that is associated with a context, usually a time and location, and has an associated action. The context may be specific, e.g. a particular time and location or may be generalised, e.g. every Friday afternoon.

Cognitive robotics is one of the areas where episodic memory has been most prevalent. One of the earliest mentions of episodic memory for cognitive robots is EPIROME [68]. This is a framework for investigating high level episodic robot behaviour. However, there is no mention of a retrieval mechanism nor of how an event is represented. EPIROME differs from some other systems, such as SOAR [69], in that EPIROME events are typed, similar to classes in an object orientated programming language. This is also a representation that we make use of in our work.

Even earlier than EPIROME, Laird *et al* [69] proposed the SOAR cognitive

architecture. In 2004 Nuxoll and Laird [7] extended the architecture to include an episodic memory. In this paper they listed the main components of an episodic memory that we have already noted. In later work [8, 70], discuss their implementation in more detail. One thing that remains a constant in the work on SOAR and indeed in most work on episodic memory is the point at which an event is created. In SOAR an event is created every time an agent performs an action. This is a reasonable strategy in their test domains, which are games such as tankSOAR or Pacman. As we shall see, a domestic environment presents problems that require a different approach..

The memory retrieval mechanism used in SOAR is based on a two-phase, nearest neighbour, cue matching algorithm. This is very similar to the way most case-based reasoning systems work, as we will discuss later. Retrieval involves first identifying possible episodes based on surface cue analysis and then performing structural cue analysis on episodes that were returned from the first phase. This method works well for a small number of stored episodes but degrades as that number increases, With nearest neighbour cue matching algorithms, there may be many episodes very close to the cue, any of which could be possible matches. Even if there are no episodes near to the cue, one will still have to be selected. SOAR addresses many of the problems associated with nearest-neighbour methods, such as providing agents with meta-data to detect sub-perfect matches. However, on a large enough database this will still have difficulties accurately detecting the correct match. We propose a retrieval method that reduces the possibility of selecting the wrong episode given an observation by learning type-specific retrieval policies. The algorithm is described in Chapter 4.

Wallace *et al* [71] address the efficiency of SOAR on large data sets of episodes. As an agent's sequential collection of episodes increases it takes an agent significantly longer to trace back through its history to find the correct episode cue. Instead they assign hashcodes to episode sequences, reducing an episode sequence to an integer. This meant that they can quickly recognise episodes but not recall them as the entire episode is discarded.

Vanderwerf *et al* [72] extend this work by examining possible hashcodes that can allow at least partial recall of the episodes also. Again in the

context of single agent games, tracing through an agent's history is adequate to eventually find the correct match. There are a limited number of actions that an agent can take and so the correct action will likely be in a recent trace of episodes. As our test domain is an embodied agent in an unstructured environment, a rule-based retrieve mechanism gives us greater flexibility and oes not assume that the relevant event is in a recent trace.

Nuxoll *et al* [9] compare the most common approaches for forgetting episodes. While there are various techniques that can be used to improve the efficiency of an episodic retrieval mechanism, one of the most effective ways is to remove episodes that are no longer relevant. Their comparison concluded that an activation approach, in which episodes are selectively removed from the database based on criteria relating to a certain type of events frequency and recency of recall has the best performance.

In 2012 Nuxoll *et al* [8] further extended the work on episodic memory in SOAR and showed how it can enhance an agent's cognitive capabilities. Up to this point all work had primarily focused on the structure of an episodic memory and how it can fit into a cognitive architecture. Here, they evaluate how episodic memory contributes to the cognitive capabilities of an agent in an environment. Their evaluation is on two different games: tankSOAR and Pacman. The evaluation consists of showing how the agent's action modelling and decision making improve when enabled by an episodic memory. They show how, after multiple runs of each game, passing episodic memories down through the runs, the agents are able to learn to outperform the hand coded control agent by predicting the outcome of future actions.

Operating in an unstructured domestic environment, episodic memories require a rich representation. Tecuci and Porter [73], represent generic events as a triple: $\langle context, contents, outcome \rangle$, where *context* is the setting that an episode took place in, *contents* are the set of events that make up an episode and the *outcome* is the episode's effect. We employ a similar representation as described in Chapter 4. However, we make some slight adjustments. Each generic event is represented as a frame [74] in a graph

database. The *contents* in Tecuci and Porter's representation are replaced by a *compound* action, whose effect results in a significant state change. This is something that will be explained in more detail later. The *context* is separated into *time* and *location* variables. We also include links to other events that may be connected to the current event. The event representation and justification for it is explained in Chapter 4.

Stachowicz *et al* [75] proposes an episodic-like memory (ELM) for cognitive robots. They use the term episodic-like memory as, once again, their model is based on a hypothesis of what episodic memory is, as opposed to a proven definition. The term episodic-like memory was coined by Clayton *et al* [76, 77]. Stachoiwcz *et al* design their system with the aim of being able to connect experiences over time and space and to be able to efficiently recall them.

Stachowicz *et al* identify a number of components that they claim are essential to an ELM module. They first state that previously experienced events must be recollected in their spatio-temporal context. They deal more specifically with quantitative spatio-temporal measurements as opposed to qualitative. They note that a complex event is made up of several sub-events and that the retrieval of any one component of an event means a retrieval of all of the event. The information present in an episode or event must be flexible, by which they mean that they should be able to manipulate the information to relate it to similar but not exact replicas of the event.

Learning from an event must be done after a single observation. They note that events can be both nested and overlap with one another and that the ELM module must act independently of any other software modules. In other words, no other software module should be designed specifically to cater for the ELM.

Although this covers a number of the key elements of an Episodic Memory Module (EMM) it misses several crucial points. For example, they fail to take into consideration the significance of an event. The significance of an event is one of the main factors in determining whether or not an

event is added to the EMM in the first place. They also do not take into consideration the degradation of a memory over time. The EMM has a limited capacity and so it is crucial that old and possibly irrelevant memories are removed to make room for more recent or more relevant episodes. Another serious limitation, which is common to most of the work that we have reviewed, is that they put constraints on the data that an episode can have. This means that an agent is limited in what it can learn and remember and therefore also limited in what it can retrieve.

Constraining data types has the advantage that events can be indexed and retrieved in nLog(n) time. However, speeding up retrieval does not remove matching errors. Constraining the data types also means trying to fit a single retrieval cue to all events. We have already discusses why this is not suitable for unstructured environments, as encountered by a domestic robot. So while this approach may retrieve events quickly, it is not guaranteed that the correct event will be retrieved and it does not generalise to unstructured environments. We describe our solution to these problems using Ripple Down Rules in Chapter 4.

Liu *et al* [78, 79] present some interesting uses for an Episodic Memory Module (EMM). They use the EMM to improve behaviour planning. Even though all states of the environment are not observable to a robot it must still make a decision on its behaviour. They note how using a partially observable Markov decision process for planning under uncertainty can produce some good results but they also note that it can also become computationally inefficient when the number of unobservable states becomes too large. By using the episodic memory they can reduce perceptual aliasing. In other words, they try to reduce the errors of the current observed state determined purely by sensor data by trying to match the current state to one of the previously observed states. They note in their paper how the spatio-temporal information is relevant to constructing an event but again discuss it in a purely quantitative way.

Similar to Stachowicz *et al* they also note the importance of one-shot learning. Like with Stachowicz, however, they do not take into consideration the significance of an episode, nor do they address the degradation of an episode

as its repetition throughout time becomes less frequent or non-existent. Liu *et al* also constrain their data types.

Other methods for event retrieval, have been proposed. We have already discussed the technique used in SOAR [70]. Lim *et al* [80] use a method called a compound cue to retrieve events. This is problematic, however, as it relies on retrieving events based on a number of matches in the attributes between two events. This, in a large enough data base will almost certainly lead to multiple matches with no way of finding the best match given the current context.

The general associative memory model of Shen *et al* [81] also proposes a method that can retrieve events even with partial or noisy retrieval cues. Chang and Tan [82] improves on this, based on a generalised self-organising neural network known as fusion adaptive resonance theory. They evaluate their method on the CAVIAR data set. While their model is complete and works well with both noisy and partial retrieval cues it makes assumptions about what information is relevant to any given event. This will make it difficult to generalise over a complete continuous spectrum of events as some information is relevant in certain contexts and not others. Their retrieval is also based on obtaining a match score between two events and therefore risks losing valid matches as it returns the best match score as the correct match every time. As it is a neural network like approach they normalise all of the input data thereby losing all contextual information.

We address many of these issues by embedding ripple down rules [4, 6, 3] into the generic frame of an episode.This allows our system to learn a customised matching algorithm for events of any type. That is, the best match will be returned based on a set of rules for that particular event. This means that partial or noisy event cues can be handles by rules that can discard missing or noisy information. It also means that we are not confined to learning a matching algorithm for events of any one type.

Berlin and Motro [83] propose an alternative approach to event retrieval. They use Bayesian learning to match two client schema. An attribute dictionary is a collection of attributes consisting of possible values and their

probabilities. The database acquires knowledge about attributes through inputs from domain experts. They determine a matching score which is a numeric value determining to what extent the client X matches to the dictionary A. They do this for pairs of schema and the total matching score between schema A and schema B is the sum of the matching scores between schema A and the dictionary and schema B and the dictionary. As this requires a lot of training data, we chose not to investigate this any further.

Other machine learning techniques for data matching have been proposed by Leordeanu [84] and Caetano [85]. While machine learning techniques can be very powerful for event retrieval as we have noted, they can also have major drawbacks in that they very often attempt to fit a single cue to all events. One of the advantages of Leordeanu and Caetano is that they move away from handcrafted solutions to a retrieval cue, and learn as new data are presented. However, as with a lot of machine learning techniques these methods required large data sets.

Vernon *et al* [86] present a case for a joint episodic-procedural memory. Their procedural memory is a means by which an agent anticipates future events based on sensory experience. They propose a framework that allows internal simulation to be conditioned by sensory inputs and episodic memory. The simulation hypothesis, as they refer to it, makes three assumptions:

1. Regions of the brain can be excited without causing physical movement;
2. Perceptions can be caused by internal brain activity and not just external stimuli;
3. Motor behaviour or perceptual activity can evoke other perceptual activity.

They claim that episodic memories can evoke these three assumptions and can therefore be used to anticipate future action. In their research, they focus strongly on the neurobiological aspect of episodic memory. They

note how episodic memories can be fuzzy or incomplete representations of an experience and retrieval of an episodic memory may create a different reconstruction each time.

2.5.1 Case Based Reasoning

Case Based Reasoning (CBR) attempts to solve problems based on solutions to similar, previously encountered problems, and so is very closely related to episodic memory. Kolodner [87] and Sharma and Sharma [88] review recent work on CBR.

Some of the earliest work related to CBR was presented by Schank and Abelson [89] in 1977. They proposed that situations can be recorded as scripts, where each script provides information about an event that allows the system to create expectations or perform inferences. Watson and Marir [90] give an extensive review of this work.

One of the main challenges facing CBR is how to retrieve the most relevant case given an observation. One of the most common methods is the two-phased approach described earlier. The first phase typically involves an inexpensive, shallow retrieval to select some candidate matches. The candidate matches are typically quite broad in nature and most of the candidates will not be valid matches of the observation. The second phase applies a more fine-grained method to rank the matches, selecting the best one. Kendall-Morwick and Leake [91] compare some of the most common two-phased retrieval algorithms and note the design considerations necessary for effective and efficient retrieval. One of the more common methods is to window the phase-one retrieval so that the number of candidate matches returned is limited, thereby improving the time complexity for the second phase. Our retrieval algorithm is also based on a two-phased approach, although we do not limit the number of candidate matches in the phase-one retrieval as this risks excluding a valid match.

Kendall-Morwick and Leake's compare phase-two retrieval algorithms where the candidate matches are ranked. No algorithm was conclusively found

to be superior. This is because CBR can be applied to such a wide diversity of environments that it is impossible to definitively state the "best" method for ranking candidate matches. Many of the algorithms implement a qualitative similarity function. This is usually a simple calculation based on the quantity of common information between two cases or some other hand crafted similarity measure, such as, a geometric distance calculation. Veloso and Aamodt [92] and Riesbeck and Shank [93] are examples of the latter where the observed features serve as dimensions in the event space.

Homem *et al* [10] employ qualitative learning through CBR and apply it to learning "keep-away" soccer in the RoboCup 2D soccer simulation. To match cases, their qualitative similarity function is based on a Conceptual Neighbourhood Diagram (CND). A CND is defined as a spatial region containing each of the 49 spatial relations in $\mathcal{EOPRA}_6$ [94]. They calculate the qualitative distance between objects in the CND to compute the qualitative similarity between previous cases and the current case.

The problem with qualitative similarity functions is that while they retrieve cases that are similar, based on the provided metric, they do not necessarily retrieve the most relevant cases for the task at hand. Smyth and Keane [95] note that instead of simply retrieving the case that is most similar, the system should instead retrieve the case that is most easily adaptable to achieve the goal of the current task. They use a look ahead method where they estimate the cost of adapting a particular case to achieve the goal of the current task before retrieving it. This is computationally expensive in the short run, but it does achieve better results. Typically, cases that are easier to adapt to solve the problem are best suited to the task, following an Occam's razor assumption. However, there is a limitation to this approach too. They evaluate it in a warehouse domain that has a finite number of possible or likely states and therefore a finite number of executable actions. Thus, it is possible to estimate the cost of adapting a case to a particular situation. This is not so suitable for an unstructured environment where an agent may have to perform the same kinds of tasks in a variety of situations or contexts and so modelling the cost of adaptation of particular cases is very difficult and often impossible.

In our work we address the problem of how episodes can be retrieved based on contextually relevant information where the type of event is not known in advance. It is also not known in advance what types of information might be relevant to a given type of event and why. Thus, our system is built on an adaptive, incremental learning algorithm where rules for why one type of event matches an observation are learnt.

There have been many practical applications of CBR. A review of applications of CBR by Maher and Pu [96] details how CBR is has been used in design. In particular, they review applications in architectural design, car design, communications design and even in such niche areas as fire engine design. CBR has also been used in medical applications (Holt *et al* [97]). In more recent years, neural networks have been favoured in this area [98, 99], but due to the increasing demand for explainable AI, CBR methods are making a return. CBR enables a physician to narrow down the scope of possible diagnoses and treatments. Much of this work again uses on qualitative similarity measures where the more symptoms that match, the more likely two cases match. As noted, CBR has largely been applied to domains where qualitative similarity functions are suitable or in the case of Smyth and Kean [95], a cost estimation approach is instead used. In the review by Sharma and Sharma [88], they note the four steps in CBR as *retrieve, reuse, revise and retain* and the *retrieve* step is responsible for recalling the most similar case using some variation of a nearest-neighbour algorithm.

Our work can be considered a form of case-based reasoning where each type of event has its own similarity measure, which is learned incrementally with each observation of a new instance of an event type. Using incremental learning, a policy for a particular type of event can be acquired based only on what is relevant to that type of event. If a particular type of event is incorrectly recalled then the policy is updated, in either a supervised or an unsupervised manner, using domain specific knowledge acquired from a human expert and induced through observations of that type of event. We describe this process in Chapter 5.

2.5.1.1 Never Ending Learning

Never Ending Learning (NEL) is very closely related to episodic memory in that it involves agents learning from their environment over extended periods of time. Never Ending Language Learner (NELL) [100, 101] is a machine learning system designed to read blocks of text from the web, extracting relevant information. Each day the learner is supposed to improve on its reading ability, which means to improve on the accuracy of its interpretation of the text. Carlson *et al* [100] describe the learner in its earlier stages after just 67 days, explaining the architecture and goals of the system. The basic design consists of a knowledge base (KB), which is a collection of facts and a knowledge integrator (KI). This basic design decides on which facts to promote to the KB. Each day the KB is consulted and external sources check to verify if the facts are still relevant. The facts are then weighted with a confidence rating and presented to the knowledge integrator, which then makes a final decision on the level of confidence in the proposed fact and the facts with the highest rating are restored to belief status. In this way, the current base of facts is constantly being questioned to ensure that they are still valid. This would be of benefit in a domestic robot since the environment can change frequently.

Carlson *et al* describes the design as working like an expectation-maximisation (E-M) algorithm. The E step adds facts to the KB and the knowledge integrator then checks these facts and makes a final decision on them. The M step then retrains the software modules so that they are up to date with the new knowledge. Further design principles that Carlson *et al* claim are necessary for a system like this to work are:

- Using subsystem components that make uncorrelated errors - If we train multiple classifiers to classify the same thing then they can verify each other, reducing the overall error rate;
- Learning multiple types of inter-related knowledge. This provides multiple independent sources of the same types of beliefs;
- Using couple semi-supervised learning. Coupling allows one predicate

to teach another;

- Distinguishing high confidence beliefs from low confidence constraints
- Using a uniform KB representation to capture candidate facts.

After the first 67 days the NELL system achieved a precision of 74%.

Mitchell *et al* [102, 101] extended NELL, collecting over 120 million confidence weighted beliefs. They discuss the similarities between NELL and the way in which a human learns over time. People, as they note, learn to perform numerous different functions over years of self-supervised training, where previously learned knowledge enables the learning of new knowledge and self reflection ensures that plateaus in performance do not occur.

The NELL provides a good case for where long-term learning is useful. Long-term Autonomy has also been a focus for research in robotics. Hawes *et al* [103] study the performance of service robots over long-term usage. Similar work was also conducted by Willow Garage [104] and TANGY [105]. Hawes *et al* define long-term usage as a number of weeks of non-stop use, with the obvious exception of allowing the robot time to recharge. They focus mainly on software robustness and how it can improve over time through learning.

Nicolescu *et al* [106] teach robots how to perform new tasks through a combination of demonstration and verbal cues from a human expert. The aim is to develop a flexible mechanism that allows a robot to learn and refine representations of high-level tasks from interaction with a human through a set of underlying capabilities already available to the robot. One ability that humans possess when learning new tasks is to differentiate between relevant and irrelevant information. Therefore, they propose to perform multiple demonstrations to the robot, in different environments and to further refine it thereafter with feedback from the teacher. In this way, the robot can more easily distinguish relevant and irrelevant facts. As well as purely visual training during the demonstration, the human teacher gives the robot verbal cues . For example, if the teacher were to say "Here" it indicates to the robot that the current environment is relevant to the

task. “Take/Drop” are instructions that provoke the robot into performing certain actions and “Start/Stop” indicate when training has started and finished.

Calinon *et al* [107] similarly look at differentiating between relevant and irrelevant information. The main focus of the paper is on how to generalise the information that has been presented to the robot during training. The aim is to teach the robot some simple manipulation tasks, specifically, picking and placing a bucket, moving a piece on a chess board in a certain configuration and picking up a piece of sugar and bringing it towards the mouth. The procedure is first to find the relevant features of the task then decide on how the task should be performed and finally attempt to find an optimal controller to generalise the task by the most efficient method. In training the robot they use a slightly different approach from Nicolescu [106]. Instead of getting the robot to watch and repeat before giving it some verbal cues to refine the task reproduction performance, they instead manually move the various hardware components of the robot in a process they refer to as kinesthetics. They argue that by training the robot in this fashion they can more accurately and more quickly collect the relevant data than if the robot were to observe a human performing the task.

CoBot [108, 109, 110] was a robot operating at CMU alongside people over an extended period of time. The idea is that CoBot could assist people in the lab with things that they may need but could also request help from people when it encounters a problem that it could not execute due to hardware or software constraints. The research is conceptually similar to ours in that it is a robot operating in the real world with people. If the robot became mis-localised it asked a person to confirm to it where it was. The research also focused on knowing which people were likely to be able to help, for example they may know that Dave in room B is a good candidate for assistance. This is episodic in that the information is spatio-temporally aligned, but otherwise the system did not make as general use of episodic memory as our system.

Kunze *et al* [111] conducted a review of work in Long-Term Autonomy. The survey covers in depth the application of LTA to different fields of

robotics such as Navigation and Localisation, Knowledge Representation, HRI, Perception and Learning. Most of the work presented in the review has also been reviewed in this literature survey.

2.5.2 Plan Recognition

An important aspect of our work is in being able to recognise the state of the world as a result of some action or set of actions. This share some problems with plan recognition. Here, we only review the work most closely related to ours.

Blaylock and James [112] goal recognition system tries to predict the goal of a plan from a sequence of observation. The work is related to ours in that it must recognise if a new observation is a new event or an extension of the currently observed events. In our work, we are concerned with being able to determine the point at which an episode terminates.

Abductive reasoning is often used to infer a plan given an observation. Singal and Mooney [113] note some of the issues that occur with classical abductive reasoning techniques such as, for example, that they are typically constructed of hard clauses and as such cannot handle uncertainty. By contrast, probabilistic methods handle uncertainty well but cannot cope with structured data representations. They introduce an approach that uses Markov Logic Networks. The approach mitigates the drawbacks associated with both first-order logic approaches and probabilistic methods and retains the benefits of both.

An example best illustrates the relevance of plan recognition to our work. If a robot is requested to pick up a cup there are several actions that are executed in the process. The first is to navigate to where the cup is. Using abductive reasoning, combined with a probabilistic model, we could abduce that no plan resulted in this state of the world, i.e. the robot being at the cup location. It is therefore likely that this action is an intermediate action and part of a plan that will result in a significant change in the world's state, such as the robot holding the cup.

This is essential as will become clear when we detail how an event is created in Section 4.2. We create events based on a significant state change in the world, which is the result of a compound action being executed. Recognising compound from intermediate actions means that we do not create events for every single action but only for significant ones.

Meadows *et al* [114] address a similar issue. They focus on *plan understanding* which is different to *plan recognition* in that the objective is to provide an explanation for an agent's behaviour in terms of the inferred goals, beliefs and intentions of the agent based on observed actions. To achieve this, their system incorporates a knowledge base of rules expressed in first-order logic and a working memory that is a set of ground literals. Their representation has a hierarchical structure, in which a task is subdivided into conditions, invariants, subtasks and effects. This structure relates closely to our work because, as previously noted, we require a technique to differentiate intermediate and compound actions, similar to how Meadows *et al* segment their task structure into compound-and sub-tasks.

However, they rely on *a priori* domain specific knowledge. This is means that facts must be explicitly added to the knowledge base, which reduces the generalisability of their technique. In contrast to this, we use a statistical model, combined with abductive reasoning to learn whether certain types of actions are in compound actions or if they serve only as an intermediate action. This is complicated by the fact that this may depend on the context in which the action is executed. Consider two cases where a robot must leave a room. In one case, it is to get to another location where a more significant task awaits it. In another it is because it has been requested to leave the current room because it is causing an obstruction. In the former case, the action of traversing to another room is an intermediate action whereas in the latter it is a compound action as its outcome has achieved a significant state as the action was carried out on the instructions of a person.

Kautz and Allen [115] introduce an action hierarchy where actions are decomposed into sub-actions. This enabling them to make inferences about a plan without exact matches being required. For example, a *makePasta*

action may be subdivided into two sub-actions, *makeSpaghetti* and *makeGnocci*. Therefore, on observing either of these actions a conclusion could be made that making pasta is the objective and as this action is tied to a boil action it can be inferred that boiling will occur at some point in the future.

This action hierarchy representation is relevant to our work in that we require a similar understanding of how actions can be decomposed into compound actions that achieve a significant goal and intermediate actions that are executed along the way.It should be noted however, that Kautz and Allen make a closed world assumption where all actions and their decompositions are known in advance. We make no such assumptions. Moreover, in our work the same action executed in different contexts may be an intermediate action or a compound action and so we cannot use an *a priori* world representation like Kautz and Allen.

Chapter 3

Topological Mapping

The work described in this chapter was published in the Australasian Conference for Robotics and Automation 2020 [116]

3.0.1 Statement of Acknowledgement

This research was jointly conducted with Dr. David Rajaratnam. Dr. Rajaratnam's original research on topological mapping was extended so as to capture, as best as possible, the true ground truth representation of each room and corridor within an environment allowing us to achieve the state of the art results that are presented here and allowing us to build a semantic ontology of an environment for use in an episodic memory system. .

3.1 Introduction

We already noted in the literature survey the relevance of topological mapping, particularly its function in providing us with a basis for a world model, which is essential for episodic memory. This is our primary use for a topological map in the context of an episodic memory system. While we mention the advantage of a topological map to a navigation task, par-

ticularly how some of the maps that we create can be used to generate paths of maximum safety, we do not explore those use cases in this thesis. Instead, we explore how a topological map allows us to ground objects to regions of an environment so that a robot can more effectively reason about state changes of those objects. This is handled through a combination of the world model and live observations. We also reviewed the most relevant work to date in the field. In this chapter, we present the basis for our method. We first briefly summarise the method and explain the significance of the research and we then provide a detailed description of the theory, implementation, evaluation and results. Finally, we conclude with proposed extensions to the work.

Topological mapping is the process of segmenting a metric map into a collection of meaningful regions. What is regarded as meaningful is context dependent. For example, sometimes it may be relevant to know only the outer boundaries of an environment so that an agent knows the limits to which it is permitted to travel. In other contexts, it may be necessary to segment the environment into a collection of regions that an agent can access, or a collection of regions that a team of rescue robots can access [117].

It is worth noting the distinction between topological mapping and semantic mapping as these two are often confused. As noted, a topological map is a map that has been segmented into different regions of space. In our approach, as will become clear, these regions are linked by *critical points*. A semantic map provides a high-level or qualitative description of the environment, often using features to generated meaning to a region of space. Our approach primarily focuses on topological mapping in that we are presenting a novel approach to segmenting a metric map. We do however use semantic features within the environment to assist in a semi-autonomous labelling process to provide logical names to regions. For example, we might use the fact that a robot identified a sink and fridge to assign the label *kitchen* to a particular region. Our main contribution however is in the field of topological mapping rather than semantic mapping.

In our work, we are interested in regions that provide an accurate description of the structure of rooms and corridors within an indoor environment

such as an apartment, house, or office. In robotics, there are several applications for topological maps based on this type of segmentation, for example, high-level path planning, localisation [118] and spatial reasoning [119] to name but a few.

As noted already, our primary motivation for generating accurate topological maps lies in their application to the development of episodic memory, which includes building associations between information of many different types, such as times, locations and actions. In other implementations of episodic memory in cognitive robots, such as the SOAR architecture [69, 7, 120], the availability of this information is often assumed at a high-level of abstraction. It remains an important and open challenge, how such abstract information can be derived from low-level sensor data. In particular, because episodic memories contain spatio-temporal information, the ability to determine the location, in qualitative terms, at which an event took place is essential. Accurate topological maps fulfil this function.

Our work addresses some key issues in the construction of accurate topological maps from the occupancy grid produced as part of a robot's Simultaneous Localisation and Mapping (SLAM) process [121, 122, 123]. The main issues that we aim to address are assumptions that previous approaches make in relation to the environment. While not all approaches that we have reviewed make all these assumptions, the main ones that we have identified are, environments must be empty, environments must be convex or relatively simple in shape, or the number or size of doorways in the environment. Our approach softens many of these assumptions while still achieving state of the art results. In particular, we make three major contributions. Firstly, we provide a method to build accurate topological representations of an environment. Secondly, our method makes no assumptions about the semantics of the environment.

Finally, our approach produces five maps at distinct levels of abstraction, all with different yet highly relevant information. At each stage, we create a more generalised representation of the environment for the robot to use, removing low-level of metric features at each step. The different levels of abstraction are important to note as they provide the robot with maps

that can be used in different contexts. For example, the third map that we create is a Generalised Voronoi Diagram. This informs the robot about paths of maximum clearance throughout the environment. By following these paths, a robot may not find an optimal route but will find one of maximum safety where it is least likely to damage itself or anything else by colliding with objects or people. This is essential if we are to adhere to Azimov's first and second laws of robotics. The final maps provide the robot with high-level descriptions of the spatial connectivity of different regions of the environment. The robot is then able to communicate at a level that people can comprehend. This abstract representation of an environment is crucial for a robot to co-exist with people.

The maps are produced as part of a five-stage pipeline. We summarise this pipeline in Section 1.3.

Our method is evaluated using a precision and recall metric as proposed by Bormann *et al* [2] over ground truth. We achieve 98% precision and 96% recall in both empty and cluttered environments of arbitrary shapes and sizes, which to the best of our knowledge is the most accurate segmentation of an environment to date. We use a combination of open source maps, provided by Bormann *et al* [2], Google's open source SLAM software, Cartographer[124] and maps that we generated in our laboratory using a Toyota Human Support Robot [125].

3.2 Methodology

As noted, our region segmentation algorithm is a five stage pipeline where each stage of the pipeline produces a map at a different level of abstraction to the previous stage. The five stages of the pipeline are as follows:

- Clean the occupancy map;
- Create a Generalised Voronoi Diagram (GVD);
- Determine doorway points;

- Create regions;
- Merge regions.

Note that we use the term *regions* as opposed to Voronoi regions. While it is true that we start from a GVD, the regions themselves are not Voronoi regions. A Voronoi region is a polygon in which all points enclosed within the polygon are closer to a one Voronoi point than they are to any other Voronoi point. It will become clear later as to why this is not the case in our work.

The first four stages of the pipeline can be computed dynamically by extending the dynamic variant of the Brushfire algorithm for computing a Generalised Voronoi Diagram, proposed by Lau *et al* [11]. While the dynamic variant of the Brushfire algorithm is itself not an original contribution of our work, as it is a re-implementation of Lau *et al* [11], we do extend the algorithm by applying it to stages 1 through to 4 of our pipeline.

3.2.1 The Brushfire Algorithm

The Brushfire algorithm is a means of calculating the attractive potential of cells in an occupancy grid. It is often categorised as a planning algorithm, but it has a broader application. It is particularly useful when trying to find a path of least resistance rather than the shortest path. In using the algorithm on an occupancy grid, the initial step is to assign all occupied cells a zero distance value, the next step is to assign all free cells that touch an occupied cell a value of one. Cells that touch a cell with a value of one but are themselves free are assigned the value two and so on. The method is summarised in algorithm 2. A graphical representation of how the algorithm is applied to a static map is given in figure 3.1.

Let $c_x \in R^{i \times j}$. Where $R^{i \times j}$ is the occupancy grid and c_x is a cell with coordinates (i, j)

Extending the basic brushfire algorithm, Lau *et al* [11], develop a dynamic variant that is able to efficiently deal with updates to the occupancy grid

Algorithm 2 brushFire: Brushfire Algorithm

Input: $R^{i \times j}$

brushfire($R^{i \times j}$)

```
1: for all c_x ∈ R^{i×j} do
2:    if occupied(c_x) then
3:       c_x.value ← 0
4:    else if notOccupied(c_x) then
5:       lowestNeighbour = findLowestNeighbour(c_x, R^{i×j})
6:       c_x.value ← lowestNeighbour + 1
7:    end if
8: end for
```

Algorithm 3 findLowestNeighbour: Finding a cell's lowest valued neighbour

Input: Cell c_x with coordinates (i,j), Occupancy grid $R^{i \times j}$

Comments:

1: *Gets the 8 cells that touch the current as an array.*

3: *Remove the zeroth element of the Adj_8 array* 5: *It is possible the nearbouring cells have not yet been assigned a value either*

findLowestNeighbour($c_x, R^{i \times j}$)

```
1: Adj_8 = getAdjacent(c_x, R^{i×j})
2: lowestValue = getValue(Adj_8.at(count))
3: Adj_8.removeAt(0)
4: while lowestValue is null do
5:    lowestValue = getValue(Adj_8.at(count))
6:    Adj_8.removeAt(0)
7: end while
8: for all a ∈ Adj_8 do
9:    value = a.value
10:   if value is null then
11:      continue
12:   else if value < lowestValue then
13:      lowestValue = value
14:   end if
15: end for
16: return lowestValue
```

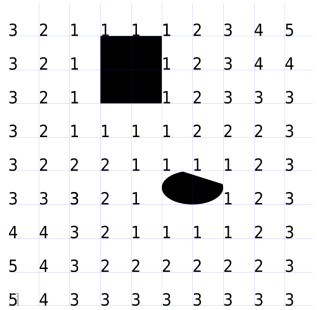


Figure 3.1: Graphical representation of the Brushfire algorithm. The cells in the grid that are coloured black are occupied. Any cell, whether fully or partially occupied is assigned a value of zero. Neighbouring cells of a cell whose value is zero, and free, get a value of one, cells that are neighbouring a cell of value one and are free are assigned a value of two and so on.

by tracking any changes and only recalculating cells that are modified as a result of these changes. They use this algorithm to calculate Voronoi points as part of a GVD. Here, we apply the algorithm to each of the first four stages of our pipeline.

When creating a map, a new occupancy grid is generated at whatever frequency the LIDAR publishes its data. This new map contains information about previously unknown cells or else updated information about the state of previously known cells. When a cell changes its state to *occupied*, Lau *et al* [11] initially call one of two functions: **setObstacle** for when the value of the new cell is known and occupied or **removeObstacle** for when the value of the new cell is known and unoccupied. Both of these functions update the value of the cell and insert it into a priority queue that sorts the queued cells by distance. They then iterate over this queue in the **updateDistanceMap** function and call either a "lower" or "raise" wavefront as required. A "lower" wavefront is called when a cells state changes to occupied. The "lower" wavefront updates the closest obstacle distance of affected cells. Similarly, a "raise" wavefront is called when the value of a cell changes to unoccupied. The "raise" wavefront clears the distance value of all cells whose closest obstacle was the newly unoccupied cell. When calling the **updateDistanceMap** function the changes are propagated to all affected cells which completes the update.

In the implementation by Lau *et al* [11], the "lower" wavefront is responsible for determining if a cell is in the GVD. In our case, the algorithm is has more than one purpose. The first job of the "lower" wavefront is to determine the value of unknown occupancy grid cells. We are assuming that each cell should have a definite value, either occupied or unoccupied, but due to noisy lidar readings this may not be the case. Thus, as we will demonstrate in Section 3.2.2, we develop a method to decide the value of unknown occupancy grid cells. The second job is the to determine if a cell should be included in the GVD, the third is to determine if a cell in the GVD is a *critical* cell and lastly if a cell is not in the GVD then we determine which region it belongs to.

The pseudo code for these algorithms is seen in algorithms 4 to 7.

In algorithms 4 to 7, *obst* refers to an array, which for each cell in the occupancy grid, stores the coordinates of the nearest occupied cell. If a cell, s, is occupied then its value in *obst* is itself. The variable $dist_s$ refers to the distance from s to its nearest occupied cell. The function **isOcc(s)** returns true if s is an occupied cell and **clearCell** resets the value of s. The function **insert**, inserts s into the priority queue with distance d or updates its priority if s is already in the queue. Comparing these algorithms to Lau *et al* [11], one might only see a small difference in algorithm 7 where, instead of only trying to establish if a given cell is a member of the GVD, we execute all functions that make up the first four stages of our pipeline. This is what the function **callAllMapUpdates** is responsible for.

Algorithm 4 setObstacles/removeObstacle: Lau *et al*'s dynamic variant of the Brushfire algorithm

Input: s - a cell on the grid

setObstacle(s)

1: $obst_s \leftarrow s$
2: $dist_s \leftarrow 0$
3: **insert**(*open*, s, 0)

removeObstacle(s)

1: clear(s)
2: $toRaise_s \leftarrow true$
3: **insert**(*open*, s, 0)

Algorithm 5 updateDistanceMap: Updating the distance map

updateDistanceMap()

1: **while** $open \neq empty$ **do**
2: $s \leftarrow$ **pop**($open$)
3: **if** $toRaise_s$ **then**
4: **raise**(s)
5: **else if isOcc**($obst_s$) **then**
6: **lower**(s)
7: **end if**
8: **end while**
9: **return** updatedMap

Algorithm 6 raise: A raise wavefront

Input: s - a cell on the grid

raise(s)

1: **for all** $n \in Adj_8(s)$ **do**
2: **if** $(obst_n \neq cleared \wedge \neg toRaise_n)$ **then**
3: **if** $\neg$**isOcc**($obst_n$) **then**
4: **clearCell**(n)
5: $toRaise_n \leftarrow true$
6: **end if**
7: **insert**($open, n, dist_n$)
8: **end if**
9: **end for**
10: $toRaise_s \leftarrow false$

Algorithm 7 lower: A lower wavefront

lower(s)

1: **for all** $n \in Adj_8(s)$ **do**
2: **if** $\neg toRaise_n$ **then**
3: $d \leftarrow \|obst_s - n\|$
4: **if** $d < dist_n$ **then**
5: $dist_n \leftarrow d$
6: $obst_n \leftarrow obst_s$
7: **insert**(open, n, d)
8: **else**
9: **callAllMapUpdates**(s, n)
10: **end if**
11: **end if**
12: **end for**

In the following explanation of each of the five stages of the pipeline, we assume that we are working with a static map. Our main reason for doing this is that it makes the explanation of what is happening at each of the stages clearer.

Algorithm 8 callAllMapUpdates: Calling the relevant functions

Input: Cell s and adjacent cell n

callAllMapUpdates(s, n)

1: **correctMap**(s, n)
2: **selectVoronoiPoints**(s, n)
3: **selectDoorwayPoints**(s, n)
4: **createRegions**(s, n)

3.2.2 Cleaning the Occupancy Map

An occupancy grid is a map generated using SLAM [121, 122, 123]. The value of a cell in an occupancy grid can be either *occupied*, *unoccupied* or *unknown*, where the last case indicates the unexplored areas of the environment. Deciding what to do with unexplored areas is important in building a topological map, in particular because frequently, the laser scanner used to generate the occupancy map incorrectly identifies a particular cell as unknown when in fact the value of that cell should be known.

This can produce a spray paint effect that leads to generating regions around features that do not correspond to any actual features in the environment. This can also have adverse effects on other common low-level algorithms such as an A^* path planning algorithm.

To solve this, we propose a simple, yet effective, algorithm that corrects or cleans the occupancy grid of these cell misrepresentations. If a cell is unknown but touches an occupied cell then its value becomes occupied. If it is unknown but touches only unoccupied cells then its value becomes unoccupied. The algorithm is summarised in algorithm 9. Let the state or value each cell, $c_{i,j} \in R^{i \times j}$ where $R^{i \times j}$ is the two dimensional occupancy grid, be represented as $c_{i,j}.state$.

This algorithm makes the task of generating a topological map significantly more feasible as it means that at the very least, the outer boundary of the map and any other walls within the map are now unbroken and do not contain any missing or incomplete data.

On occasions however, it is possible that we remove real artefacts from the

Algorithm 9 correctMap: Cleaning the Occupancy Map

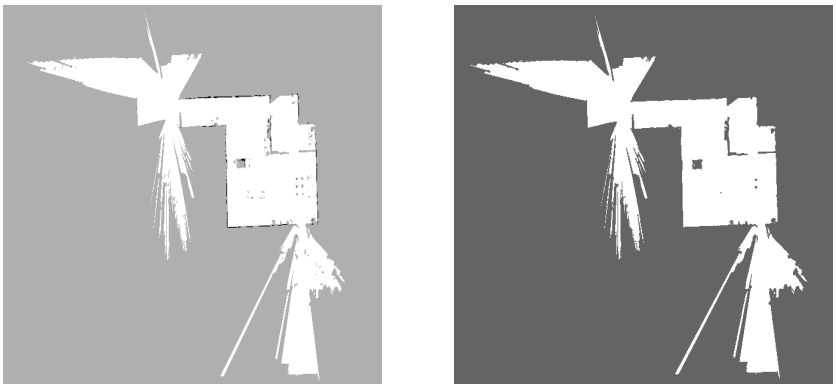
Input: Occupancy map $R^{i \times j}$
Comments:
9: *If any neighbouring cell was occupied, the state would not be unknown and so this would only be assigned occupied in that event*
correctMap(R)

```
1: for all c_{i,j} ∈ R do
2:     Adj_8 = getAdjacent(c_{i,j}, R^{i×j})
3:     for all a ∈ Adj_8 do
4:         if c_{i,j}.state is unknown ∧ a.state is occupied then
5:             c_{i,j}.state ← occupied
6:             break
7:         end if
8:     end for
9:     if c_{i,j}.state is unknown then
10:        c_{i,j}.state ← unoccupied
11:    end if
12: end for
```

occupancy grid. For example, the legs of a table or chair are often narrow, which can confuse a SLAM algorithm as the LIDAR data are not as reliable as data from a wall or some other larger feature.

When this happens, the legs of tables can often be represented as unknown points when they should be known and occupied. Moreover, it is possible that they are represented as unknown points that touch only unoccupied cells. By algorithm 9, they are then interpreted as being themselves unoccupied cells and are removed from the map. This can create difficulties for a navigation algorithm. However, we have found that it does not cause any serious issues for our region segmentation algorithm. Such an effect can be seen in figure 3.2. Importantly, as the robot keeps exploring the space around it, the LIDAR data will become more complete and any errors in the map will be updated.

A before and after image is shown in figure 3.2. The occupied cells are black, the unoccupied cells are white and unknown cells are grey. Note that in the figure on the right there are only two colours present after cleaning because all cell states are considered to be known.



(a) Before Correcting (b) After Correcting

Figure 3.2: Before and After Correcting the Occupancy Grid

3.2.3 Creating the Generalised Voronoi Diagram

The Generalised Voronoi Diagram (GVD) is a collection of points in free space where the two closest obstacles to each point have the same distance [126, 127]. Typically, this applies only to a continuous space and not to a discretised cell space. A more complex algorithm is needed to compute the GVD for an environment represented by a map with discretised cells, as usually generated by SLAM algorithms.

In selecting which cells are candidates for the GVD, we use the method proposed by Lau *et al* [11], which is itself an extension of the technique presented by Kalra *et al* [128]. We choose this algorithm as it very accurately and quickly generates a GVD and it is easy to implement with a Brushfire algorithm to work with dynamically changing environments.

Let a cell be denoted $c_{i,j}$ and a neighbouring cell be n_{i^*,j^*}, the asterisk representing plus or minus zero or one. If both $c_{i,j}$ and n_{i^*,j^*} are potential Voronoi cells in the continuous sense with the distance to the nearest occupied cell, occ_c and occ_n being $dist_c$ and $dist_n$ respectively, and if $dist_n$ cannot be lowered by using occ_c as the new occupied cell for n_{i^*,j^*} then we must check which of the two cells is a candidate for the GVD. This can occur because we are dealing with cells as atomic variables rather than single points. Having two neighbouring cells in the GVD can cause complications so it is generally considered good practice to avoid this.

We first check that neither $c_{i,j}$ nor n_{i^*,j^*} is adjacent to its closest obstacle. Lau *et al* [11] then state the following: If n_{i^*,j^*} has a valid closest obstacle that is different and not adjacent to the valid closest obstacle of $c_{i,j}$ then the cell that is chosen as a candidate for the GVD is whichever cell violates the continuous Voronoi condition to a lesser extent. In other words, if we were to swap the obstacles around so that the closest obstacle to $c_{i,j}$ is now the obstacle of n_{i^*,j^*} and vice versa, then whichever has the smaller distance increase is selected as a candidate for the GVD. Algorithm 10 clarifies the process. Let voroGraph represent the GVD. When calling this algorithm, it is assumed that we have computed the GVD in the continuous space already. This algorithm is then called when two of the cells in the GVD are neighbours to each other.

Algorithm 10 selectGVDCell: GVD Cell Selection [11]

Input: Two cells in the GVD, $c_{i,j}$ and n_{i^*,j^*} and the two cells that each of these are nearest too, $c_{nearest}, n_{nearest}$.

selectGVDCell$(c_{i,j}, n_{i^*,j^*}, c_{nearest}, n_{nearest})$

```
1: if dist_c > 1 or dist_n > 1 then
2:   if c_nearest ≠ n_nearest ∧ ¬touching(n_nearest, c_nearest) then
3:     if distance(c_{i,j}, n_nearest) < distance(n_{i*,j*}, c_nearest) then
4:       voroGraph.insert(c_{i,j})
5:     else
6:       voroGraph.insert(n_{i*,j*})
7:     end if
8:   end if
9: end if
```

The final result can be seen in figure 3.3. Figure 3.4 shows the Generalised Voronoi Graph being created dynamically using the dynamic variation of the Brushfire algorithm as already discussed. Note that figure 3.3 and figure 3.4 are not of the same map.

3.2.4 Selecting Critical Points

The final three stages of the pipeline are where we make the most significant contribution. We have already detailed a minor contribution in how we clean the occupancy map, which, given the reduction in error allows us

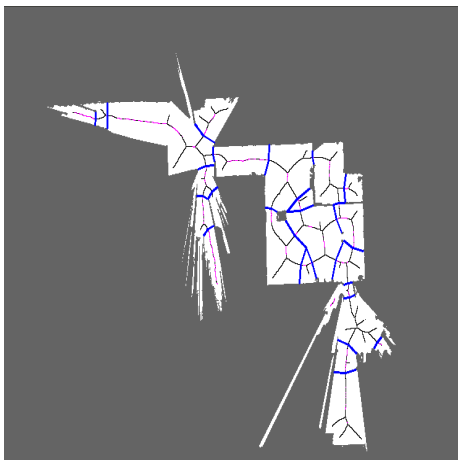
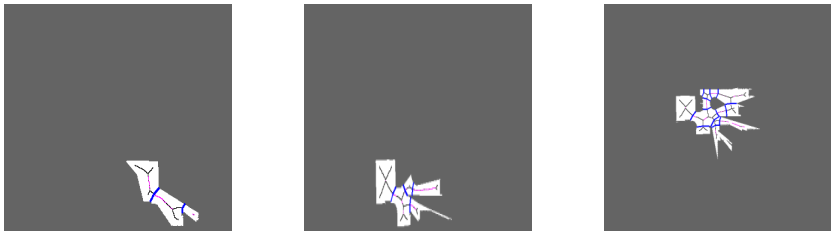


Figure 3.3: Voronoi diagram of @Home arena of RoboCup 2018 in Montreal. The blue lines represent critical points that we cover in Section 3.2.4. The black points are the cells that are selected as members of the GVD and pink cells are cells in the GVD that are also corridor points.



(a) (b) (c)

Figure 3.4: GVD being created dynamically

to generate significantly more accurate environment representations than previous techniques.

In this section we detail how select what we refer to as *critical points*. In ideal environments, all *critical points* would only correspond to *doorway points*. However, as environments are rarely ideal, often times other points are selected because they have been misinterpreted as being *doorway points*. Thus, it is more appropriate to refer to these points as *critical points* rather than *doorway points*.

As already noted in the literature review, there have been other approaches that have looked at the concept of selecting critical points in a generalised Voronoi diagram. Most notably by Thrun and Bücken [19] and

also by Beeson *et al* [20]. These approaches however lack the robustness of our method. Thrun and Bücken rely on selecting critical points inside ϵ-neighbourhood regions. Each ϵ-neighbourhood region is a circle of fixed radius, inside which there must be at least one critical point. This critical point is defined as being the point in the GVD with the shortest distance to its respective occupied cell. Using this approach, there will be multiple critical points defined in locations that they should not be. For example, in our approach a corridor will have two critical points, one at each end of the corridor, assuming the corridor is empty, which, for arguments sake, we will in this instance. However, Thrun and Bücken would have multiple critical points, as each ϵ-neighbourhood region must have at least one.

Beeson *et al* [20], define critical points as being junctions in the topological map. Like many methods that we have reviewed so far, this only works when the environment is completely empty and so does not generalise to more common domestic environments that are typically filled with clutter.

Our definition of a critical point means that our approach is extendable to many types of environments. Our goal is to identify points that correspond to *doorway points*. We can accurately selects doorways as being *critical points* but also we may select other points that are not doorways due to clutter.

Our approach to identify critical points is as follows. Given any cell in the GVD, v_n, there are at least two obstacles in the environment, whose distances to the cell are the same. We denote this distance as d_n. The change in d_n between two consecutive Voronoi cells, is $d_n - d_{n^*}$. Where n^* is the Voronoi cell next to cell n in the GVD, assuming that they are ordered by distance to one another. We will refer to this distance as d_{diff}.

We compute d_{diff} for each Voronoi cell pair, (v_n, v_{n^*}), which we represent as v_p. Assuming there are i cells in the GVD there will be *i-1* pairs as each cell will belong to two pairs except the first and last.

We compute the running average for each $d_{diff}(v_p)$ as follows:

$$d_{avg} = \forall n \in \{0, \ldots, i-1\} \left(\frac{\left\| d_{diff}(v_{p(i)}) \right\| + d_{total}}{n} \right)$$

Where d_{total} is the current total of all distance differences summed. Algorithm 11 shows the pseudo code for this algorithm.

We try to find a $d_{diff}(v_p(i))$ that is significantly less than d_{avg}. That is,

$$d_{diff}(v_p(i)) \ll d_{avg}$$

In most environments, LIDAR provides a detailed enough occupancy grid that the rate of change of distances between any cell in the GVD to its nearest occupied cell and that same distance measure for a consecutive cell in the GVD, is subtle. This is, of course, until an irregularity occurs which causes a rapid change in this distance. Such an irregularity might be a door. Therefore, we assume that all points in the GVD have the potential to be *critical points* but that the initial value for a point n is false. Let the value for a *critical point* for any point n be cp_n. By default then

$$cp_n \leftarrow false$$

If we observe a rapid change in the distance from one Voronoi cell to its nearest occupied cell then this value becomes true.

$$cp_n \leftarrow true$$

Algorithm 11 describes how we select a critical point. Let V be the collection of Voronoi points in the GVD and v_n be Voronoi point n.

We have already noted how it is possible for us to get *false positives* using this algorithm.

These points are *false positives* for doorways, instead picking up legs of chair

Algorithm 11 selectCriticalPoints: Critical Point Selection

Input: the GVD **as** V
selectCriticalPoints(V)

1: **for** $v_n \in V$ **do**
2: **if** $v_n \neq v_{end}$ **then**
3: $d_{diff} = d_{v_{n^*}} - d_{v_n}$
4: $sum = sum + d_{diff}$
5: $curr_{avg} = sum/n$
6: **if** $d_{diff} \ll curr_{avg}$ **then**
7: $cp_n \leftarrow true$
8: **end if**
9: **end if**
10: **end for**
11: **return** V

and tables etc. However, they still provide useful information about the environment, particularly in that one constraint for identifying a *doorway point* is that the robot must be able to navigate through the area that was responsible for creating this point. Thus, a path planner might find these points useful as it is known in advance which areas are accessible to the robot.

3.2.5 Creating Regions

The previous stage of the pipeline returns the GVD, along with a vector of critical points. We take a subset of the vector where each point in the subset has the value of cp_n as *true*. Let this vector be denoted by:

$$CP = \langle cp_1, \ldots, cp_n \rangle$$

assuming that there are n *critical points*. Each *critical point* is also a Voronoi point and between two consecutive *critical points* are a vector of other Voronoi points. Let two consecutive *critical points* cp_n and cp_{n^*} be denoted by P_n.

$$P_n = \langle cp_n, cp_{n^*} \rangle$$

CP can then be represented as:

$$CP = \langle P_1, \ldots, P_{n-1} \rangle$$

For each P_n, there are a set of Voronoi points. The set of Voronoi points are the points in between each of the *critical points* that construct the *critical point* pair P_n. This set is called V_n.

Each *critical point* pair P_n and the set of associated Voronoi points V_n make up the collection of Voronoi points that are contained in region R_n.

The triple:

$$< cp_n, V_n, cp_{n^*} >$$

represents the Voronoi points between two critical points. Starting at either end of this vector, that is, one of the two *critical points*, we create a circle. The radius of this circle is the distance from each *critical point* to its respective occupied cell, in the case of a *critical point* that is also a *doorway point*, this is the distance from the centre of the door to the edge of the door. This then defines a maximum limit for how far this region, which is the region being created between cp_n and cp_{n^*} can expand into the next region, that is, the region that is created between cp_{n^*} and $cp_{n^{**}}$. As we assume that doorways are relatively small in width this ensures that no region expands too far into a connecting region but also guarantees at least some overlap. See figure 3.5

Taking the vector V_n, we start at the first Voronoi point and we repeat the same process as we did for each doorway until we reach the last Voronoi point in V_n. That is, we create a circle with each Voronoi point as the centre of that circle and the radius of each circle equal to the distance of

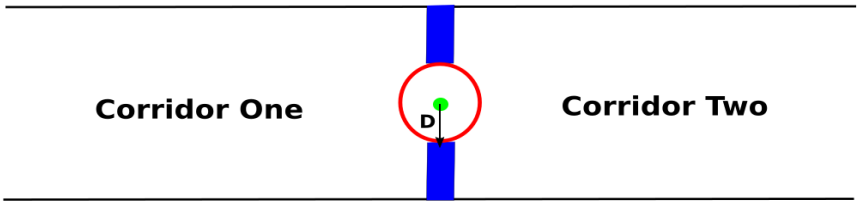


Figure 3.5: The blue blocks represent a doorway dividing two corridors. The green dot is the Voronoi point at the centre point of the doorway. The distance from the Voronoi point to the door is D, the red circle is the circle created around this doorway point and is of radius D. The circumference of the circle is the limit that the region representing *corridor one* can expand into *corridor two* and vice versa for the region representing *corridor two* into *corridor* one.

each Voronoi point to its closest occupied cell. Remember that V_n itself does not contain either *critical point* but the points in between. We have one condition when creating these circles. No circle created around a Voronoi point that is not a *critical point* can expand past the maximum point of expansion as determined by the circles created around the *critical points*. We let this condition be denoted by γ and a violation of this condition is γ^*. As circles are just regions we, can use the RCC-8 relations [54] to check if this condition has been violated.

Let C_{cp_n} be the circle created around one of the *critical points* and let C_{v_n} be the circle created around Voronoi point n. We can say that if either of the following two RCC-8 relations holds then the condition has been violated.

$$TPP(C_{cp_n}, C_{v_n}) \vee NTPP(C_{cp_n}, C_{v_n}) \rightarrow \gamma^*$$

This means the if C_{cp_n} is a Tangential Proper Part of C_{v_n} or if C_{cp_n} is a Non-Tangential Proper Part of C_{v_n} then we have violated the condition for maximum expansion. In theory, there is nothing wrong with C_{cp_n} being a TPP of C_{v_n}. However, it is very close to a violation and so we do not allow it for safety. See figure 3.6.

Figure 3.6: Tangential and Non-Tangential Proper Part

If there is a violation then we reduce the radius of the circle C_{v_n} until there is no longer a violation. This can be more easily understood through algorithm 12. Let the function **createCircle**, generate a circle with centre at point, v and radius R.

Algorithm 12 generateCircles: Expansion of Voronoi Points to Occupied Cells

C_{cp_n} : Circle around doorway point n
$C_{cp_{n+1}}$: Circle around doorway point n+1
C_v : Circle around Voronoi point v
R_{C_v} : Radius of C_v
generateCircles()

1: **for** $v \in V_n$ **do**
2: C_v = **createCircle**(v, R_{C_v})
3: **while** γ^* **do**
4: **reduce**(R_{C_v})
5: **end while**
6: **end for**

When determining the region we calculate the point of intersection of each of the circles. The intersection points are the vector of points that make up the polygon that represents the region. When we have enough Voronoi points, that is, a high enough resolution in the GVD, the points of intersections of each of the circles are a negligible distance away from the occupancy grid wall. This means that we can very accurately trace along the edges of the occupancy grid. See figure 3.7.

We show an example of this in Figure 3.8. Figure 3.9 shows how this can be done dynamically using the dynamic invariant of the Brushfire algorithm.

Note that this region is not strictly speaking a Voronoi region. To explain

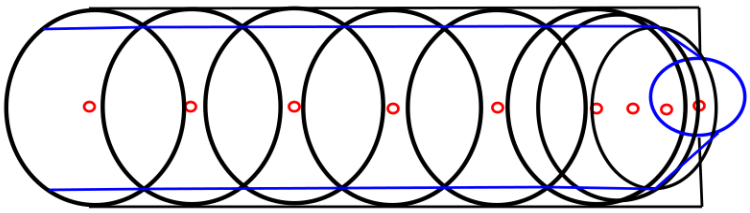


Figure 3.7: Key: Voronoi points can be seen in red, circles are in black and the region created is in blue

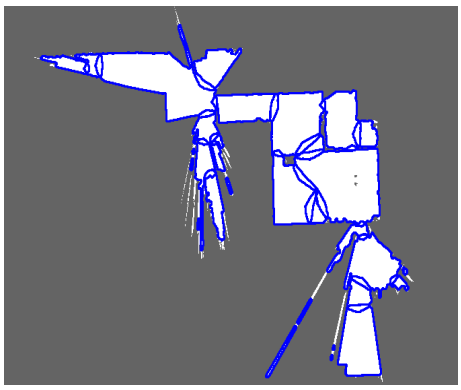
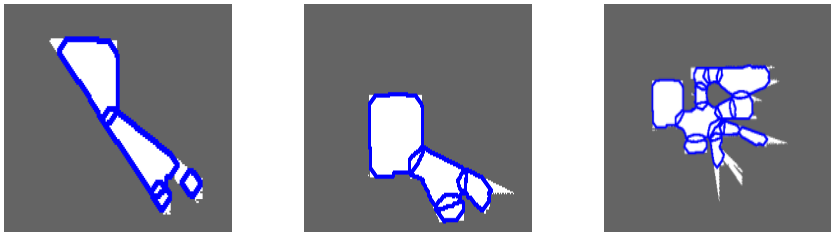


Figure 3.8: Regions Diagram of @Home arena in RoboCup 2018, Montreal



(a) (b) (c)

Figure 3.9: Regions being created dynamically

why this is the case, let us consider the strict definition of a Voronoi region. In a given n dimensional plane, a collection of points can be selected based on some chosen metric. Each of these points is known as a Voronoi point. In a Generalised Voronoi Graph that metric is a coordinate whose distance to its two closest obstacles is the same. The metric however can be anything. A Voronoi region is a polygon with the same number of dimensions as the plane itself that contains all coordinates that are closest to a given Voronoi

point than to any other Voronoi point in that plane.

If we look only at *critical points* as Voronoi points, then our generated regions are not strictly Voronoi regions. Take two *critical points*, α and β. The region extending from α to β is denoted by R_α and the points within that region are P_α. We can say that there will be some points within P_α that will be closer to *critical points* β than to *critical point* α, more formally expressed as follows:

$$\forall x \in P_\alpha, \exists x : [distance(x, \beta) < distance(x, \alpha)]$$

From this it is clear that the regions created by our proposed method are not strictly Voronoi regions.

3.2.6 Merging Regions

The final stage in our pipeline is merging each of the regions created in the previous stage. As noted in section 3.2.5, an environment that is not completely empty or convex will give rise to sub-regions within rooms. This is due to clutter creating false doorway points. While these sub-regions are useful for localisation and navigation, it is also possible to generate a higher level map by labelling and merging sub-regions into larger regions that more closely match our understanding of rooms.

We extend the previous map by attaching a label to each of the sub-regions based on the larger region that they create. We use a semi-autonomous procedure to do this. There are two related reasons for this. Firstly, in general, a room's type (kitchen, bedroom, etc) cannot be determined purely by its geometry. Additional information is required about the nature of the objects within the room. One way of obtaining this information it to use object recognition, but this is prone to recognition errors.

Secondly, we assume that the robot interacts with its human operators. For these interactions to be natural and intuitive, the regions in the environment require meaningful names. These names can be very specific to

individual people. For example, a typical house may have several bedrooms some of which may be assigned to a specific person while others may be designated as a visitor bedroom or study.

So even if the robot has an accurate vision system, it must still ask its human operators to assign meaningful names to regions. Consequently, it is reasonable for the purposes of this research to adopt a semi-autonomous region labelling process requiring human operator involvement.

The semi-autonomous labelling system has two steps. Initially the robot uses its vision system, in combination with YOLO [129, 130, 131], to recognise and place different objects around the environment. Then using a look-up table containing a list of objects likely to be found in a room, it labels regions based on those objects, or if no such objects were identified then it labels the region based on the objects that were closest to it.

The systems then returns the labelled map to a human, who corrects or updates the region labels. This is only possible for maps for which we have vision data. Many of the maps that we evaluated were open source and we do not have vision data. In these cases we had no choice but to manually label the maps.

Each region that is generated in stage four of this pipeline, is assigned a unique number, starting at one and ranging to however many regions are created, R $= \langle r_1, \ldots, r_n \rangle$. For each $r_n \in R$ we assign a descriptive label depending on what room or larger region that region, r_n is a member of. If two regions are connected and have the same label then they are merged.

$$connected(r_n, r_m) \wedge same\_label(r_n, r_m) \rightarrow merge(r_n, r_m)$$

As regions are just polygons made up of a collection of two dimensional coordinates, we can use the CGAL computational geometry library [132] to perform the merging of two regions. When two regions are merged, a new region is created. This new region is connected to all of the individual regions to which the original two regions used in the merge were connected.

Assuming all regions have been labelled correctly, we group them based on those labels. We then iterate over each group and merge the connecting regions together.

Algorithm 13 summarises the process. The **replace** function takes the original region r, the region that it was merged with r_{next} and replaces both of those regions, that are in the group of regions g with the newly generated, merged region r_{new}.

Algorithm 13 mergeRegions: Merging regions

Input: G = Groups of regions by name
mergeRegions(G)

```
1: for g ∈ G do
2:     for r ∈ g do
3:         r_next = r.next
4:         if connected(r, r_next) then
5:             r_new = merge(r, r_next)
6:             replace(r, r_next, r_new, g)
7:         end if
8:     end for
9: end for
```

After algorithm 13 has completed, all regions with the same label are merged into one. Figure 3.10 shows the end result for the RoboCup 2018 @Home arena.

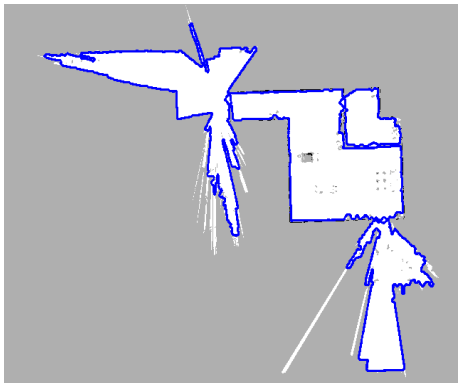


Figure 3.10: Regions having been merged diagram of @Home arena Montreal RoboCup 2018

The results of the evaluation of the methods described in this chapter are presented in Section 6.1.

3.3 Conclusion

This chapter, presented a novel approach to segmenting a metric map into a topological map that achieves state of the art accuracy in determining an environment's structure. The empirical evaluation presented in Section 6.1, show that the method achieves 98% precision and 96% recall. Segmentation of a metric map into a collection of rooms and corridors has a number of uses in robotics, navigation, localisation, spatial reasoning, maintaining a world model, human robot interaction and episodic memory.

In this thesis, we are interested in how the topological map can provide an abstract description of the spatial information pertaining to a particular domain entity. For certain types of events, this information may be needed to provide a contextual understanding of the event. For example, a meeting that takes place at the same time in a particular room. Another example of this type of information might be that the robot has an understanding that a wallet was last seen next to the fridge in the kitchen. If the robot learns that this is a common location for a person to leave a wallet then it can use that information to infer where a wallet might be should it be asked to retrieve it. Our contribution is not focused on the language used to describe these spatial relations but rather how our topological map can be used to provide the most accurate representations of these spatial relations between objects within the environment.

We have already addressed some of the limitations of this work. The main limitation relates to how regions are labelled. Manual labelling is not ideal, however there is a large body of work on autonomous labelling, most notably by Brucker*et al* [37] that compliments our approach very well and could be used to extend it.

The first method clusters objects together. Often sub-regions are created around items such as fridges or tables (looking only at a domestic environment). Therefore, if certain objects appear within sub-regions or clusters of certain objects appeared within sub-regions, this leads to a reasonable expectation as to the type of room to which the region belongs. For exam-

ple, if two regions are next to one another and one region contains a kettle and a toaster and in the other, a fridge, this a good indication that both regions belonged to the kitchen.

Another technique is to use a method similar to that proposed by Kong *et al* [133]. They use natural language descriptions of scenes to improve symbol grounding, that is, to more accurately assign a label to a particular object or scene. The method involves having an understanding of the typical spatial relations of objects to one another in a particular room. Having determined where objects are relative to one another they can apply a label to a scene. For example, if the robot observes a television in front of a couch in a sitting room, then it is safe to assume that any regions in-between the two objects belong to the sitting room.

This may seem unnecessary given the previous suggested technique, however, consider for example a studio apartment where there may not be clearly defined separations between different rooms. To one side of a couch you might have a dining table and to the other side a television. Using the logic proposed by Kong *et al* [133] you could say that any regions that lie between the couch and dining table belonged to the dining room and any regions between the couch and television were the sitting room.

Chapter 4

Creating and Retrieving Events in Episodic Memory

The work described in this chapter was accepted as a poster presentation in the Advances in Cognitive Systems conference in 2020 [134].

4.1 Introduction

In this chapter, we address the following questions:

- Why was an event recalled?
- How was the event recalled?
- Why was the event stored in the first place?

Figure 4.1 shows an example of a type of event that a person is likely to have stored in their episodic memory. In recalling events, there are two main points of interest that need to be considered. The first is the accuracy and the second is the efficiency with which events can be recalled. Our approach is most concerned with the accuracy of the recall but we also develop extensions that address the issue efficiency.

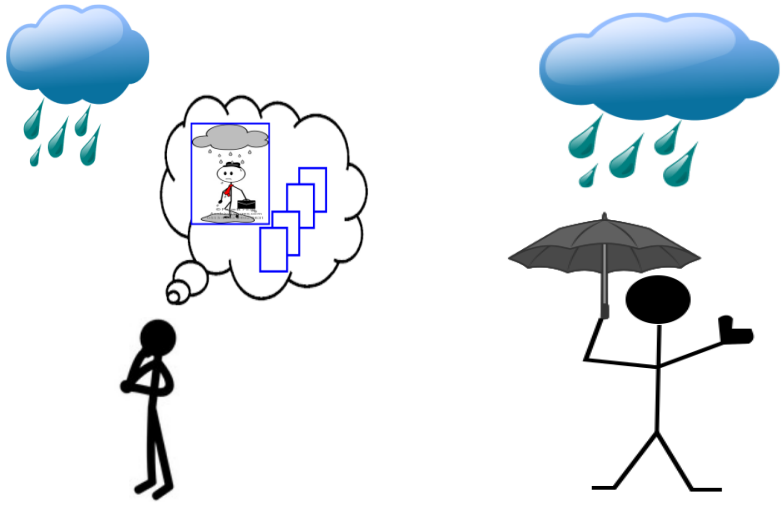


Figure 4.1: The person sees rain which recalls an event depicting a time when they were caught outside and reminds them to bring an umbrella. The blue boxes represent other memories that may be banked in a person's episodic memory. The goal of the retrieval algorithm is to recall the correct type of event from the bank of memories.

As described in the literature survey, many different approaches to event recall have been proposed. In the context of episodic memory for cognitive robots, the most notable is the SOAR cognitive architecture [69, 7, 8, 70, 9]. To retrieve memories, Nuxoll and Laird [7, 8] use a nearest neighbour procedure. Nearest neighbour methods work well in domains with a finite number of realistically probable states. For example, in evaluating their system, Nuxoll and Laird [7, 8] test the accuracy of retrieval in two different board games, the more complex of these is TankSOAR which is a variation on "first person shooter" games. The world is a two-dimensional grid and the agent is a tank that can move about the grid. The tank is equipped with hardware that can change the state of the world, for example, a device that can fire missiles. To fire a missile, the tank must collect missiles that are randomly scattered throughout the environment. The objective of TankSoar is to score as many points as possible. Points are awarded for hitting an enemy tank with a missile and more points are awarded if the enemy tank is destroyed. A tank loses points if it is hit.

Nearest neighbour approaches will work well in such cases, however they are

not as effective in unstructured, partially observable environments because they struggle to distinguish between two apparently similar types of events that only have subtle differences.

Event recall or "case retrieval" has also been studied extensively in Case Based Reasoning (CBR). As already noted, the objective of CBR is to find a solution to a newly observed problem by retrieving a previously encountered case that best matches the current observation. This research has the same retrieval problem.

CBR often retrieves cases using a two-phased approach. The first phase is typically a simple and inexpensive search for a set of candidate matches. The requirements for retrieval in the first phase are very general so that it doesn't exclude any cases that are valid matches. The second phase attempts to find the best match amongst the candidates.

Our approach differs from previous methods in how it identifies the best match. In this chapter, we detail this approach and explain why it is a contribution to the field. In Section 2.5.1 we described the most relevant CBR research. Our conclusion is that CBR retrieval often relies on a simple qualitative similarity metric that does not consider contextual information that is unique to each type of event. We handle this by using Ripple Down Rules (RDR) to acquire context-specific recall policies. This is essential in rich, unstructured environments because subtle differences in cases may require substantially different responses. Because RDRs can be learned incrementally, if two events are identified as equally good matches, when one is actually a better match, we can update the recall policies to make the correct distinction, based on the current context. To our knowledge, no other approach allows this.

4.2 Creating Events in Episodic Memory

4.2.1 Event Representation

Many of the systems that we have reviewed represent events as an agent's action in a particular situation. However, it is also possible that events occur due to some exogenous actions and may not be seen by the agent at the time that the event happened. For example, on leaving and returning to a room, the agent may see that the position of a chair has changed. From that, the agent infers that an action must have been performed, creating a new event. The ability to infer such events is needed in a multi-agent environment, such as a home, since other occupants, human or machine, have the ability to act independently of the robot.

We represent actions in the same way as the PDDL task-planning language [135]. That is, each action has preconditions and effects and both are conjunctions of predicates. The agent is endowed with apriori information regarding the generic structure of an action. That is to say, the agent is aware that an action has parameters, preconditions and effects. However, we do not provide the agent with specific instances of action models in advance. For example, we do not inform the agent that a *move to waypoint* action has two parameters, the *from* waypoint and the *to* waypoint, or that the precondition is that the agent is at the *from* waypoint and the effect is that the agent is at the *to* waypoint. Instead, each instance of a new type of action model is learned by observations and by taking advice from a trainer.

The system stores the current state of the world as a conjunction of predicates, organised within a topological map. The arguments of these predicates refer to things in the world, for example, people, objects or the robot and the topological map allows us to ground these arguments to qualitative locations needed for the episodic memory and for the agent's planning, reasoning and communication. Locations can also be arguments of a predicate. So, from now on we refer to all people, objects, the robot, locations and anything else that can exist in the world as *objects*.

We recognise that an action has taken place and therefore, that an event has taken place using the world model. If at some point in time the preconditions of an action are satisfied and if at a later point in time the effects are satisfied, then it is assumed that the action must have occurred.

Let a predicate, P, represent a belief in the world. Let, A, be an action and let the world at time, t, be represented by Γ_t, such that:

$$\Gamma_t = \langle P_1 \wedge \cdots \wedge P_n \rangle$$

The following is the PDDL representation of a robot moving from one location to another.

$$
\begin{array}{l}
(: action \quad goto\_waypoint \\
\quad : parameters(?r - robot \;\; ?from \;\; ?to - waypoint) \\
\quad : precondition((robot\_at(?r, \;\; ?from))) \\
\quad : effect(and \\
\qquad (not(robot\_at(?r, \;\; ?from))) \\
\qquad (robot\_at(?r, \;\; ?to)) \\
\quad) \\
)
\end{array}
$$

An alternative way of describing an action uses Allen's temporal logic [52]. Let, X, be a state that occurs at time t, Y, is a successor state at time $t+1$ and, A, is the associated action:

$$\forall X, Y : X \neq Y \wedge \{X \succ Y\} \implies A$$

where $X \succ Y$ means X occurs before Y, or in terms of the world model:

$$(\Gamma_t \neq \Gamma_{t^*}) \wedge (\Gamma_t \succ \Gamma_{t^*}) \implies A$$

This tells us that action A can be deduced if we see state X, followed by state Y provided states X and Y are not the same. Making the closed world assumption, the world model tells us everything that is currently believed to be true in the robot's environment. If one of those beliefs changes then something must have happened.

In addition to the action definition, an event has other information such as the time, location, other events that it is connected to and also the recall policy for that type of event. The agent is provided with a means to create a timestamp and also a topological map apriori. Furthermore, it is also provided with the generic structure of the frames representing each of these types of data and a generic frame representing objects within the environment. The robot must use this apriori information and populate each slot in these frames with information that is has observed to learn the specific instance representation of the type of event that is taking place. We are assuming that the robot has perfect sensing for this. That is to say, it correctly identifies objects using computer vision and that it has perfectly localised itself and the relevant objects within the environment. We are aware that this assumption will not hold true in all cases in the real world. While this is a limitation of this work, it is partially handled through inductive reasoning. Here we use multiple observations of a type of event to infer what information remains consistent. In the event of a false positive, where the robot idenitifies something that it should not, this is quite effective at removing the incorrect information. However, in the case of a false negative it is less effective. We have provided some possible solutions to this in Chapter 5.

Events and all other data associated with events including the matching rule for a given type of event are stored as frames in a graph database. Before going into detail about this, we address some concerns regarding when we choose to remember an event.

As already noted, the system creates an event when there is a change in

the world model. While this allows us to recognise actions that the agent is and is not responsible for, it can also mean that every change, however inconsequential, can trigger event creation. This is clearly undesirable. There are several ways that this can be dealt with. Below, we describe some of these methods, along with our own solution, which is covered in more detail in Section 4.4.3.2. The first approach is to have a pre-defined set of action models, but we have already said that we want to avoid this since we want our robot to be able to learn new actions. For robots to co-exist with people, it must be possible for the robot to learn in a form of Never Ending Learning (see Section 2.5.1.1).

A better approach, for our purposes, is to learn that some events can be ignored as remembering them typically proves not to be useful. This method is discussed in Section 4.4.3.2. Previous research, such as Nuxoll and Laird [8], use an attention mechanism and a threshold value. If the event does not excite the attention mechanism above a threshold value then the event is not remembered.

Another way this can be done is to build policies for types of actions, similar to how we build recall policies for different types of events, and learn in what contexts a particular type of action should lead to an event being created. This is the preferred approach as it makes no prior assumptions about an action and whether or not that action is relevant and therefore makes no assumptions about whether that event is relevant. Some contexts may mean that one type of action is relevant and others may not be. Take for example, the action of moving from one room to another. Sometimes, this is only intermediate, serving as a node in a sequence of actions where a *critical* result is achieved. For example, if the robot is required to get a glass of water, then leaving the room that it is currently in and going to the kitchen is not a significant part of the event and remembering it is not useful.

In contrast, if the robot is being a hindrance and the person directly asks the robot to leave, then the action of the robot going from one room to the other is significant and, in this context, should be remembered.

We refer here to types of actions being relevant rather than types of events being relevant. This is because actions are responsible for creating events and therefore we are concerned whether a certain type of action is relevant in a given context.

We define an action hierarchy in which a *critical* action may consist of a sequence of intermediate, low-level actions. We represent this critical action as $\mathcal{A}$ and the intermediate actions as α_n, where $n \in \{0, \ldots, t\}$, assuming there are t intermediate actions we can then say the following:

$$\mathcal{A} \leftarrow \alpha_t \succ \alpha_{t-1} \succ \cdots \succ \alpha_1 \succ \alpha_0$$

For simplicity, when we refer to an action in the context of an event, we are referring to the *critical* action that created this event.

We can therefore create a new event when a *critical* action is observed. That is, a *critical* action acts like a trigger, indicating that we should create a new event. Note that this new event may or may not be a new type of event. It is the job of the recall policy associated with each type of event in memory to determine if this observed event is an instance of a type of event already in memory or a new type of event.

Note that an event that is constructed from a sequence of actions (among other information), can also be part of an event sequence. To establish if it is a part of an event sequence, we keep a record of the most recently observed events in an *episodic buffer*. When a new event is observed, we check with the other events in the buffer to see whether the current event is connected to any of them and if so, we calculate the temporal relation of the two events to one another and link them to each other.

For example, consider three types of events, the first of which is a person sitting on a sofa, the second is a person switching on the television and the final event is getting a beer from the fridge. All three events are recorded as separate events but they are all part of a single event sequence.

In this chapter, we focus only on the recall policies and how successfully

they perform and so we are assuming that every action is a *critical* action. This allows us to create a much larger database on which to evaluate the effectiveness of RDRs as recall policies.

4.2.2 Frames

As already noted earlier, each event is stored as a frame in a graph database and in this sub-section we explain this particular implementation of frames and why they are well suited to this application.

Frames are data structures used to segment knowledge into smaller sub-structures of knowledge. They were proposed by Minsky [74] in 1975. The reason we use frames is that events or episodes are collections of multiple different types of data, which can themselves be a collection of multiple different types of other data. For example, *critical* actions are responsible for creating events. Thus, one of the frames that make up an event is an *action* frame which has as members *predicate* frames and so on.

Frames permit embedded data structures with no limits on the types of data that construct an event and therefore frames allow for the most generalised representation of an event in episodic memory. They also allow us to easily access the individual data that make up the episode which is essential when it comes to event matching and generalisation.

Our system is implemented using the FrameScript language [136]. Frames can be *generic*, representing classes of objects, or instance frames, each representing an individual object. Data are held in slots, each slot having a unique name. A generic frame defines the base structure of the frame, all of the slots, any default values that those slots may have when a new instance of this generic frame is created and any procedures associated with those slots.

A procedure attached to a slot must be defined within the generic frame. Figure 4.2 shows a simple example of a generic frame.

Frames support inheritance, which is essential for our system. Since events

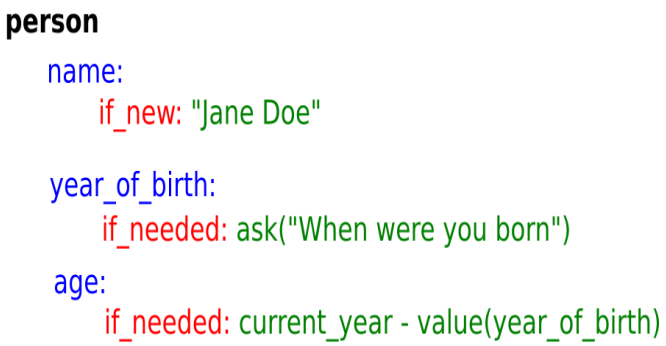


Figure 4.2: This figure shows a simple example of a generic frame describing a person. Slot names refer to properties associated with the frame type. For example, each person has a name and therefore we declare a slot called *name*. We can also apply procedures to these slots. In the example shown here we declare a procedure called *if_new* which is called when a new instance of this frame is created. The purpose of this procedure is to initialise the slot value of the new instance. This value can be overwritten later.

are typed, each event type is represented by its own generic frame. In addition, each event type inherits properties of the generic *event* frame.

As an example of an event instance, consider when a person requests a cup of coffee. The action for this event is to get the coffee, the time may be in the morning, and the location might be an office or kitchen. All of these times and locations are instances of other generic frames and all are linked from the instance frame for the event. How all these frames are used will become clear when we explain our recall pipeline in Section 4.4.

Tecuci *et al* [73], represent events as a triple,

$$\langle context, contents, outcome \rangle$$

where *context* is the setting that an event took place in, *contents* are the set of actions that make up an event and the *outcome* is the event's effect. Our representation is similar, where the *contents* are represented by a *critical* action, if the event involves a sequence of sub-actions.

In the next section we explain the generic frames that are the core of our

event representation.

4.2.3 Generic frames Used in Event Representation

After several observations of a particular type of event, it may become apparent that some kinds of information are not relevant. However, initially all of the data represented by the frames below must be considered potentially relevant.

Generic Frame *event*:
This frame defines the top-level event structure. All other event data are linked to this frame.

A generic **event** has slots: *action, start time, end time, start location, end location, references to any other information and connected events.*

Generic Frame *action*:
The core of each event is an action. A generic **action** frame has slots: *name, parameters, preconditions and effects.*

Preconditions and effects are conjunctions of predicates, so we create a frame structure to represent a predicate.

Generic Frame *predicate*:
A generic **predicate** has the slots: *name, arity, arguments.*

Predicates have arguments and the values of these arguments are objects. The word *object* refers to anything that is in the environment. This can be a physical object such as a glass, phone, television, etc. but it can also represent people, the robot, locations and just about anything else that is a grounded concept in the environment. So we define a fourth generic frame, *object*.

Generic Frame *object*:
A generic **object** has slots: *name, class.*

We also require generic frames for *time* and *location*.

Generic Frame *time*:
A generic **time** frame has slots: *year, month, day (1-31), weekday (Mon-Sun), hour, minute, part of day (morning, afternoon, evening, night).*

Generic Frame *location*:
A generic **location** has the slots: *room name.*

Each event may also have connected events. For example, if one were to break a glass then a connecting event would be to clear it up. We therefore define a sixth generic frame, *connected event*

Generic Frame *connected event*:
A generic **connected event** has slots: *event, temporal relation.*

Connected events have two slots, the first slot is the event itself which links to an *event* frame which has already been defined. However, a connected event also has a temporal relation to the event that it is connected to. This is represented by one of Allen's temporal relations[52]. Therefore we define one last generic frame.

Generic Frame *temporal relation*:
A generic **temporal relation** has all of Allen's temporal relations[52]: *precedes, meets, overlaps, finished by, contains, starts, equals, started by, during, finishes, overlapped by, met by, preceded by.*

Note that the *connected event* slot may be empty if there are no other associated events.

Before explaining how a new event is created, we note that recall policies for events are represented as RDRs. The purpose of the recall policy is to establish if something that an agent has observed is an instance of the type of event to which the policy applies. Therefore, we extend generic frames to include an RDR recall policy. RDRs can be used to recall any kind of frame, not just event frames.

4.2.4 Creating New Types of Events

In Section 4.4, we explain how we determine if an event belongs to a new event type or if it is a type seen before. However, for now, let us assume that all observed events are new types. The system creates a new instance of an *event* frame, with an associated *action* frame and all the other required frames for time, location, etc.

If the event is a person sitting down, then the action is an instance of an *action* frame with the precondition: $not(sitting(Person))$ and effect: $sitting(Person)$. The time is an instance of the *time* frame and the location being an instance of the *location* frame type.

The job of the recall policy is to determine if an observed event is an instance of a known event type. If it is not, new generic frames for the new event type must be created. A new type of event may also have a new type of action, which becomes an instance of a new sub-class of the generic *action* frame. This is needed to create a unique matching policy for that new type of action and the same for all other data. New generic frames may also be needed to represent specialisations of other frames, e.g. sub-classing time to add morning and afternoon. Each new generic frame has an associated RDR for recognising instances of that generic frame.

The need for this can be better understood by looking at the generic *object* type. Remember that the slot names associated with this are, *name* and *class*. In an event involving a breaking glass, the *object* in question would be a glass, with some unique ID in the environment. The value of the name slot for this object is *glass_one* and the value of the class slot is *glass*. When trying to recall this event at a future point in time, we are not concerned with the fact that it was *glass_one* that broke but rather that the class of the *object* was a glass. We therefore create a rule for this type of *object*. The rule should state that if the class of the object is a glass, then it is a valid instance of this type of object:

if $match(class)$ **then** $match$

However, consider a friend, *John Smith* comes over to visit. The *object* now in question is one with class *person* and name *John Smith*. However, in this type of event the name of the person is what is relevant and not just the fact that it was any person. So here we train a policy that states the following:

$$\textbf{if } match(class) \wedge match(name) \textbf{ then } match$$

This policy is then used along with all of the other policies for the other types of frames in this type of event to establish if another observation of *John Smith* coming to visit is an instance of this type of event. If the policies have been trained correctly then it should determine that the next time *John Smith* comes to visit it is another instance of this event and a new type of event does not need to be created. One of the benefits of using RDRs to express policies, however, is that if the agent makes a mistake and incorrectly concludes that either the second observation is a new type of event or that it is an instance of a different type of event, the RDR can be incrementally updated to account for the difference between the two instances so that the same mistake is not made in the future.

These policies are very basic and only serve to demonstrate the necessity for the hierarchical structure of the data that we adopt for our model. When the robot observes a new type of event we create sub-classes for all frames that are linked to that event, starting with the event itself so that individual matching procedures can be defined. Algorithm 14 details the process that is used for every object in the event. In creating a new generic frame that is a sub-class of an existing generic frame, the slot values of the initial observation frame are the slot values of this new generic frame.

The first step to creating a new type of event is to create a new sub-class of the event and all other data. Algorithm 14 specifies this process.

It is not only generic events that have unique recall policies assigned to them but all other types of data, such as the *action* and *time* frames within an event too.

Algorithm 14 createFrameSubclass: Frame inheritance for all frames in an event

I_i is an instance of some generic frame. The function **put** takes three arguments. The frame that we are trying to put a value in, the name of the slot in the frame that that value is being put in and finally the actual value itself. The function definition is therefore: **put**(Frame, Slot name, Slot value). The **createFrameSubclass** takes as input an instance frame, gets the parent of that instance frame and creates a new generic frame that is a sub-class of this frame's parent.

Comments:

6: *Instance is a data type, this line says that if the slot value is an instance frame of some other data type then the if statement is true*

Input: I_i

createFrameSubclass(I_i)

1: $G_i \rightarrow$ parent of I_i
2: $G_{new} \leftarrow$ subclass of G_i
3: **for all** slot $\in$ *slots of* I_i **do**
4: sname = name of slot
5: svalue = value of slot
6: **if** svalue ***instanceof*** *Instance* **then**
7: **put**($G_{new}, sname,$ **createFrameSubclass**($svalue$))
8: **else**
9: **put**($G_{new}, sname, svalue$)
10: **end if**
11: **end for**
12: **return** G_{new}

Thus, we must evaluate all relevant recall policies in order to determine if an observation is an instance of a given type of event. For example, consider an event type whose rule is:

$$\textbf{if } match(action) \textit{ then } match$$

This means that if we compare two frames, one being the generic event type stored in memory and the other being an event that has just been observed and if the action slot values are the same then we conclude that the observation is also an instance of this type of event. To determine if the *action* slot values are the same we must evaluate the rule for this type of action. This might also involve evaluating the rules for some other types of data too and so on. Of course we only create new types of actions when we need to create a new type of event.

When one type of event has not yet had a sufficient number of observations so that unique recall policies can be trained, we still need some way to recall that event from memory. Therefore, if we have only seen one instance of a type of event, then the recall policies for every slot are inherited from the generic frames for which that frame is a sub-class. For example, the recall policy for each type of event is initially the recall policy that has been assigned to the generic *event* frame and the same for the *action*, *time*, *location*, etc. These are what we refer to as the default policies.

The default policies have been chosen in such a way that they are guaranteed to recall an event that matches the observation, however, because they lack specificity, they can also recall many other events that are distant matches to the observation. This is why we must then specialise the recall policy for the type of event that is a close match so that it can be more accurately recalled at a future time. This is explained in more detail in Section 4.4.

Before checking if an observed event is an instance of a new type of event or if it is an instance of an event already in memory, we need to collect and organise the information from the observation within the event frame.

We then use this frame and compare it against the generic events already in memory to determine if it is a new type of event or not. Algorithms 15 through 20 specify the process for collecting and organising the information that an agent obtains through observing an event into an event instance frame.

The first stage is to check if anything has changed in the agent's world. As already explained, an agent has a world model which is a collection of facts, represented by predicates, organised within a topological map. When one of these facts changes, for example the agent moves to a new location, this informs the agent that something has happened and therefore it is possible that an event has occurred.

We update the world model using information that the agent can observe. If the agent observes a cup on a table when there was not one there before, it adds the fact: $on(cup, table)$ to the world model. We periodically update the world model and compare two succeeding instances of the world model to check if anything has changed. If something has changed, we create an action using the differences between the two observations as *preconditions* and *effects* of that action. So if the change in the world model was the location of the robot then we would create the following action model:

$$
\begin{aligned}
&: precondition((robot\_at(robot,\ location_x))) \\
&: effect(and \\
&\quad (not(robot\_at(robot,\ location_x))) \\
&\quad (robot\_at(robot,\ location_y)) \\
&\ )
\end{aligned}
$$

We then check to see if this action is a *critical* action and if it is we create a new event instance. Using this new event instance and the generic event types that are stored in memory that have unique recall policies defined, we check to see if this is a new type of event or if it is an instance of one

of the types of events already observed and stored in memory.

Before describing the pipeline for organising information from an observation into an event it is worth noting the way that we treat predicates in the world model. Suppose we represent the state of a glass being on the table as:

$$on(glass_x, table_y)$$

with $glass_x$ being an instance of *glass*. We assume that the *on* predicate cannot apply to that same glass in more than one situation at the same time. Therefore, if we observe that same glass on the floor then the agent should add the predicate $on(glass_x, floor)$ to the world model and remove the predicate $on(glass_x, table_y)$.

Algorithm 15 handleWorldUpdates: Handle World Model Updates

This algorithm is only ever called once, when the robot is started. Initially, lastWorld will be obtained from some database where we stored the last known state of the world on shutting down the system. While the robot is in operation, the while loop to get new states of the world is called continuously

Input: $lastWorld \leftarrow \{P_1 \wedge \cdots \wedge P_n\}$: {Last known state of the world}
$allActions \leftarrow [\ldots]$: {A *critical* action can be preceded by several *non-* actions so declare an empty array to store them}
Struct Difference **contains**
  precondition: **List** of **predicates**
  effect: **List** of **predicates**
  wasDifferent: **boolean**
end
Comments:
3: *Returns a set of predicates*
11: *This is where we check if it is a new type of event or an instance of a previously observed type of event. This is a multi-stage pipeline clarified in Section 4.4*
handleWorldUpdates*(lastWorld)*

```
 1: diff = new Difference
 2: while true do
 3:     currentWorld ← getCurrWorld(lastWorld)
 4:     diff ← compare(currentWorld, lastWorld)
 5:     if diff.wasDifferent then
 6:         action ← buildAction(diff)
 7:         isCritical ← checkIsCritical(action)
 8:         if isCritical then
 9:             allActions ← allActions.append(action)
10:             event ← makeEvent(allActions)
11:             passToEpisodicMemory(event)
12:             lastWorld ← currentWorld
13:             allActions ← [...]
14:             continue
15:         else
16:             allActions.append(action)
17:             lastWorld ← currentWorld
18:             continue
19:         end if
20:     end if
21: end while
```

Algorithm 1, specifies how an agent updates its world model. In this al-

gorithm we have not detailed how the *observableWorldStates* variable is obtained. An agent can obtain information from a variety of perceptual inputs. Depending on the robot in question and the domain of application there may be different sensors that return information about the world. Most robots are equipped however with a camera and a lidar. We use the camera to give the agent information about the objects that it can observe and the lidar returns information about the map.

Algorithm 16 getCurrWorld: Get the Current World

Input: *lastWorld*: {the last known state of the world}
Comments:
6: *Each predicate has a primary argument. If the action of a predicate applied to the primary argument is in the world model and is also something that the agent can observe then we found it easier to simply replace that predicate in the world model rather than check the secondary arguments.*
getCurrWorld(*lastWorld*)

```
1: currentWorld ← lastWorld
2: states ←observableWorldStates as predicates
3: for all s ∈ states do
4:     if s ∉ currentWorld then
5:         currentWorld.append(s)
6:     else if s ∈ currentWorld then
7:         currentWorld.replace(s)
8:     end if
9: end for
10: return currentWorld
```

Algorithm 17 details how an *action* is constructed. Actions form the base of every event and are essential to event recall in almost every case. Thus, this is a crucial step in our pipeline.

In algorithm 18, we check if an action is a *critical* action or not. If an action is not *critical* then we do not create a new event instance. Rather we note this action as being an intermediary action in a sequence of actions that will conclude with a *critical* action.

Algorithm 17 buildAction: Building an Action

Input: *difference* ← instance of *Difference* struct: {predicates that have changed between two world model observations}

Struct Action **contains**
  precondition: **List** of **predicates**
  effect: **List** of **predicates**
  parameters: **List**
end

Comments:
5: *Gets all the arguments in all the predicates of the precondition*

buildAction(*difference*)

```
1: action = new Action
2: action.precondition ← difference.precondition
3: action.effect ← difference.effect
4: for all precondition ∈ action.precondition do
5:     thisPreconditionArguments ← get(arguments of precondition)
6:     action.parameters.append(thisPreconditionArguments)
7: end for
8: for all effect ∈ action.effect do
9:     thisEffectArguments ← get(arguments of effect)
10:    action.parameters.append(thisEffectArguments)
11: end for
12: return action
```

Algorithm 18 checkIsCritical: Check if action is significant

Input: *action* ← instance of Action struct:
Comments:
1: *We have assigned an RDR policy to the generic* action *frame. This function evaluates those rules to check if they are significant.See Section 4.4.3.2*
checkIsCritical(*action*)

```
1: isCritical ← evaluateSignificantRules(type of action)
2: if isCritical then
3:     return true
4: else
5:     return false
6: end if
```

Actions are constructed from preconditions and effects. On an observation of the world, it is possible that changes will have occurred. Preconditions are states that were true before the current observation and effects are states that are true after the current observation. Algorithm 19 details how we compare two succeeding observations of the world to establish if there is any observable difference.

Algorithm 19 compare: Compare Worlds

Input: *current* ← current state of world
Input: *last* ← state immediately before current state
Comments:
3: *"names of last" is an array with the names of all predicates contained in the last state of the world. It is one of the simplest checks we do to see if a new state has been observed*
8: *If the predicate's action applied to a primary argument are in both the current and last state of the world we check to see if the secondary arguments have changed.*

compare$(current, last)$

```
1: difference.wasDifferent ← false
2: for all P_n ∈ current do
3:    if name of P_n ∉ names of last then
4:       difference.wasDifferent ← true
5:       difference.precondition.append(¬P_n)
6:       difference.effect.append(P_n)
7:    else if name of P_n ∈ last then
8:       if arguments_sec of P_{n_current} ≠ arguments_sec of P_{n_last} then
9:
10:         difference.wasDifferent ← true
11:         difference.precondition.append(P_n of current)
12:         difference.effect.append(P_n of last)
13:      end if
14:   end if
15: end for
16: for all P_n ∈ last do
17:   if name of P_n ∉ current then
18:      difference.wasDifferent ← true
19:      difference.precondition.append(P_n)
20:      difference.effects.append(¬P_n)
21:   end if
22: end for
23: return difference
```

Algorithm 20 makeEvent: Make a new event

Input: *allAction* ← List of Actions: {all intermediary actions and the critical action}

Comments:

9: *The episodic buffer stores all of the events in this particular sequence. For example, breaking a glass and cleaning it up consists of two separate events. For this example we would create an instance of a connected event frame and include the breaking glass as the event with a "precedes" temporal relation. This connected event instance would then be put into the connected_event slot of the final event, which is cleaning it up.*

11: *see algorithm 14*

makeEvent(*allActions*)

```
1: startTime ← time of allActions[0]
2: endTime ← time of allActions[length(allActions - 1)]
3: startLocation ← location of allActions[0]
4: endLocation ← location of allActions[length(allActions - 1)]
5: event.startTime = startTime
6: event.endTime = endTime
7: event.startLocation = startLocation
8: event.endLocation = endLocation
9: event.connectedEvents = establishEventConnectivity(episodicBuffer)
10: event.action = allActions[length(allActions - 1)]
11: event ← createDataTypeSubclasses()
12: return event
```

Before continuing we wish to address a limitation of this work and some possible implications of it. We also would like to discuss a possible extension to this research to address this limitation. One may note that these algorithms refer only to a single type of action occurring at any given time. Thus, from these algorithms one can infer that the agent is incapable of handling different actions happening simultaneously. This is correct and it is one the limitations to this work. One of the main reasons for this limitation is that we have chosen not to provide the agent with specific action representations apriori. Consequently, an agent is unaware of the specific precondition and effect predicates that are relevant to any given type of action and must assume, unless instructed otherwise, that all observed predicate state changes are relevant to a given type of action. Thus, if two actions occur simultaneously and the agent does not know anything about either action, it must assume that the state changes that occurred as a result of both actions are relevant to only one action. While all of the

information relating to both actions will be captured, on a future observation of either of these actions, the information relating to the other action will be removed as per our learning pipeline outlined in Chapter 5.

One of the main implications of this is that we risk missing important information and not learning how to solve potential problems that arise as a result of some types of events. If a person had expected the agent to learn this information, it may be very difficult to explain why the agent has not and this will have an impact on the level of trust instilled in the autonomous agent.

One possible solution to this problem is to provide the agent with action models apriori. However, this creates further problems and requires more hands on training from a human expert. We have explicitly stated that this is something that we wish to minimise in this thesis.

4.3 Ripple Down Rules

In this chapter we explain how Ripple-Down Rules RDR are usd to create policies for event types.

RDRs were introduced by Compton in 1990 [3] as a knowledge acquisition method for knowledge based systems. An RDR is learned incrementally by interaction with a human trainer. Initially, the RDR consists of a very general default rule, which is progressively specialised as new cases are encountered. The RDR methodology can be thought of as a general-to-specific search for the best fitting set of rules. Thus, the initial RDR represents the most general hypothesis. When a new case is covered by the RDR, when it should not be, the RDR is specialised by adding an exception rule. When a new case if not covered when it should be, a new alternative rules is added. This is illustrated in Figure 4.3

We use RDRs as event recall policies in an episodic memory system. As we have noted throughout this thesis, different types of events will have different types of information that are relevant to them. Thus, when checking

if a particular event stored in memory matches an observation of an event, it is logical that we should use a policy that is specialised for that type of event. When training a Ripple Down Rule policy, we have the option to discard information that is not relevant to the system. Therefore, the irrelevant information will not be considered when making a decision as to whether the candidate event matches the observation. This is in comparison to other learning methodologies where all data that are relevant to the system must be defined in advance and used across every decision.

As the job of the RDR in our case is to determine if an observation matches an event in episodic memory, the default rule is:

if *true* **then** *no_match*

This is because it is most likely that an observation does not match an event in memory, as there may be many different types of events and only one valid match.

To further explain how Ripple Down Rules work, we use a simple example of identifying fruit. Later, we give examples of how they are used as recall policies for episodic memory, however, as the structure of episodes can be quite complex, a simpler example serves as a better introduction. We start with the default rule:

if *true* **then** *unknown*

If the first training example is a banana we add an exception rule. A generic frame for fruit may have properties like colour, shape, size, etc. When trying to generate a new exception rule, the system asks whether the values of these properties of the new example are relevant to the conclusion. In this case, we discard all properties except the colour. Thus, the new rule is:

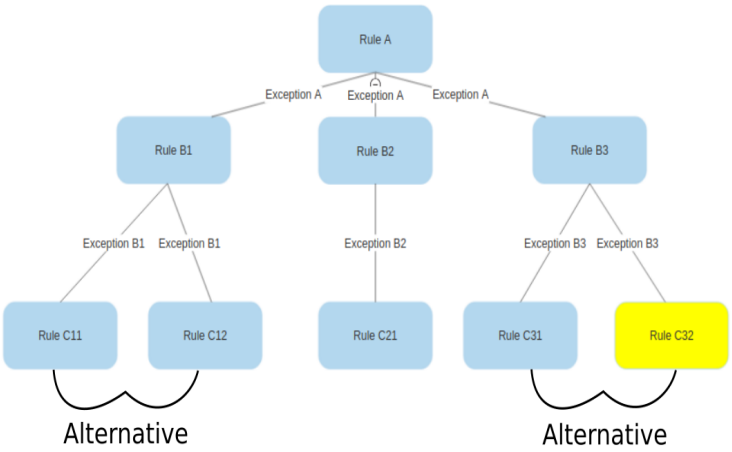


Figure 4.3: Each of the nodes in the tree represents a new RDR that is an exception to its parent node. Rule A is the default rule and in this example it has three exceptions. Rules that are exceptions to another rule but are themselves at the same level in the tree are alternatives to one another. In this example, Rule B1, Rule B2 and Rule B3 are alternatives to one another and exceptions to Rule A. Similarly, Rule C11 and Rule C12 are alternatives to one another and exceptions to Rule B1. Rule C32 is the rule that fired in this case. That is, its conditions were satisfied. If it fired incorrectly, then an exception can be added to this rule creating another level to the entire RDR. This is done dynamically without any need to rebuild the RDR tree.

if *colour* **is** *yellow* **then** *banana*

which is added to the existing RDR producing the new RDR:

if *true* **then** *unknown* **except**
  if *colour* **is** *yellow* **then** *banana*

A case that causes the creation of a new rule is called a "cornerstone case" and is stored with the rule.

The next training example that we provide is an apple. In this case the banana rule is not satisfied, so the conclusion from the default rule holds. To update the RDR, we add an alternative to the exception rule that was not satisfied. Again, we discard all properties except colour and the RDR is now:

if *true* **then** *unknown* **except**
  if *colour* **is** *yellow* **then** *banana*
  else if *colour* **is** *red* **then** *apple*

The condition for the new rule is obtained from the differences between the cornerstone case of the rule that fired incorrectly and the new case. In this example, the system will ask, *"is it because the colour is red"?*

Suppose now that a lemon is presented. One of the rules states that if the colour is yellow, the conclusion is *banana*, which is incorrect. Therefore, we now add an exception to the banana rule. The prior case of the banana and the new case have the same colour, so this cannot be used to discriminate the training examples, however, they do differ in their shape. So a new exception rule is added:

if *true* **then** *unknown* **except**

if *colour* **is** *yellow* **then** *banana* **except**

if *shape* **is** *round* **then** *lemon*

else if *colour* **is** *red* **then** *apple*

Algorithms 21 to 24 and 25 detail how an RDR is updated and evaluated respectively.

Algorithm 21 updateRDR: Update a Ripple Down Rule

Input: oldCase ← instance of some generic frame type, e.g. fruit
Input: newCase ← instance of some generic frame type, must be the same as oldCase
Input: rule ← instance of RDR and is the rule to update
Comments:
1: *This is the last rule to fire in the rule that we are updating. This is the rule that fired incorrectly*
2: *Get the differences between the old case and the new case for which the rule needs to be specialised to account for*
3: *RDRs are recursive structures. Exceptions or alternatives that are added to an RDR are themselves instances of RDRs*
4: *Adds the new rule as an exception to the lastRule. If there are already other rules as exceptions to the lastRule then this is an exception to those rules*
updateRDR(oldCase, newCase, rule)

```
1: lastRule ← rule.lastRule
2: List conditions ← differences(oldCase, newCase)
3: RDR newRule ← createNewRule(conditions)
4: addRule(newRule, lastRule)
```

If the trainer makes a mistake and answers a question incorrectly, the RDR can be corrected when a later case is misclassified due the error. This will create a more complex RDR than is necessary, but it will yield the correct conclusion.

Algorithm 22 Differences between two cases

Input: oldCase ← instance of some generic frame type, e.g. fruit
Input: newCase ← instance of some generic frame type, must be the same as oldCase
Struct Slot **contains**
name
value
end

Struct Condition **contains**
name
value
end
allconditions ← [. . .]
Comments:
4: *We get the name of the slot from the slot variable and use this to get the value from the same slot for the newCase instance*

differences(oldCase, newCase)
1: **for all** s ∈ slots of oldCase **do**
2: svalueOld ← s.*value*
3: snameNew ← s.*name*
4: svalueNew ← **getValue**(newCase, snameNew)
5: **if** svalueOld *neq* svalueNew **then**
6: condition = **new** Condition
7: condition.*slot name* ← s.*name*
8: condition.*value* ← s.*value*
9: allConditions.**put**(condition)
10: **end if**
11: **end for**
12: **return** allConditions

Algorithm 23 createNewRule: Creating a new rule

We pass a list of potential conditions to this function. RDRs training is guided by a human however and the human has the option to discard information that they believe to be irrelevant. Therefore, when training RDRs using this manual approach we need to run an additional check over the conditions to see if they should be included in the new rule and to find out what the conclusion to the new rule should be. However, when we train an RDR policy using techniques borrowed from Inductive Logic Programming as we do in Chapter 5, we do not take this last step.

Input: conditions ← List of Condition
createNewRule(conditions)
  return_rule = **new** RDR
  return_rule.*conclusion* ← **ask**(What is the conclusion to this new rule)
  for all c ∈ conditions **do**
    isRelevant ← **ask**(Is c.*name* equal to c.*value* relevant?)
    if isRelevant **then**
      return_rule.**put**(c)
    end if
  end for
  return return_rule

Algorithm 24 addRule: Adding a new rule to an RDR

RDRs also contain a *wasTrue* variable. This is true if the rule fired when it should not have and so an exception is added. It is false if a case for a rule was not covered when it should have been and so an alternative is added.
Input: newRule ← instance of RDR
Input: lastRule ← last rule to fire
addRule(newRule, lastRule)
  if lastRule.*wasTrue* **then**
    lastRule.*exception* ← newRule
  else
    lastRule.*alternative* ← newRule
  end if

Algorithm 25 evalRDR: Evaluate a Ripple Down Rule

Input: rule ← instance of RDR
Input: case ← an instance of some generic frame. In this example, an instance of fruit
evalRDR(rule, case)

```
while rule ≠ null do
  bool didFire = checkFire(rule.condition, case)
  if didFire then
    if rule.exception ≠ null then
      var result = evalRDR(rule.exception, case)
      if result ≠ null then
        return result
      else
        return rule.conclusion
      end if
    end if
  else
    if rule.alternative ≠ null then
      rule = rule.alternative
    end if
  end if
end while
return null
```

4.3.1 Ripple Down Rules as Event Recall Policies

Recall from Section 4.2 that an event is represented by a frame with slots whose values can be other frames representing different types of information. For example, there is an *action* frame, a *time* frame, a *location* frame etc. Each of these frames may also have slots that refer to other frames, for example, the action frame has two slots, *precondition* and *effect* and these refer to frames of type *predicate* etc.

The following sections describe events used in our evaluation and how we use RDRs to construct specialised recall policies.

Before we explain how we are using RDRs as event recall policies, we describe the differences between how we use RDRs and how they are more commonly used. When adding an RDR to a frame it is usual for one to add the RDR to a slot in the generic frame. The RDR is then used to determine the value that that slot should have under certain conditions.

As always, each rule in the RDR will have conditions and a conclusion. The conditions refer to the name and the expected value of one of the other slots in the frame. The conclusion to a rule is the value that the slot should receive if the rule fires.

This is the situation that we descried in Section 4.3. It is likely that we would declare a generic *fruit* frame where the slots are: *name*, *shape* and *colour*. We would add the RDR described to the *name* slot and depending on the values of the *shape* and *colour* slots we would use the RDR to determine the name of the fruit.

In our case however, we do not add an RDR to a specific slot in a generic frame but rather to the generic frame itself. The conditions of each rule refer to slot names, however, we do not include specific slot values or ranges of values within the conditions. Rather, each condition in a rule is a boolean with the same name as the slot to which it applies. The condition states whether that slot value in the generic frame should match the value of the same slot in the observation in order for the rule to fire. For example, a

rule such as:

$$\textbf{if } match(time) \textbf{ then } match$$

means that if the value of the *time* slot in the generic frame is the same as the value of the *time* slot in the observation, then the rule should fire and the frames match.

We do this for a number of reasons. The first reason for this variation on RDRs is because we often have slot values that are instances of other frames. This means that for these cases we cannot perform a simple equality as a condition. We instead need to use the policy associated with the action frame to determine if the action frame in the observation is the same as that of the generic.

Lastly, we explained in Section 4.4 how we handle the comparison of two lists. Recall that slots can hold lists in two different ways. The first is that the slot is a single valued slot but has a list as a single value. If our matching policy for the frame dictates that this slot is relevant then all terms in the list of the generic frame must match all terms of the same slot in the observation.

The second way that a slot in a generic frame can have a list value is if the slot is multivalued. This is the more common way for slots to contain lists and we treat the terms in these lists as alternatives. Thus, if the slot is relevant to the match policy then only one term in the generic frame needs to match one of the terms in the observation.

Thus, it is possible for us to capture equalities, inequalities and ranges of values depending on the how the slot contains the list. If new values of a list present as relevant then we can update the list in the generic frame rather than the policy in the generic frame. This means that we only need to update a matching policy when information about which slots are relevant to the policy change and not when values of those slots change. This has proven to be more efficient than directly updating the RDRs.

4.4 Event Retrieval

So far we have only shown how observed events are represented as frames. In Section 4.2, we made the assumption that all observations were of new types of events and therefore, when creating an event, we created a new sub-class of event and also created new sub-classes of all frame types contained within that event. It is not necessary to do this every single time. In fact, as will quickly become clear, we see a lot of repetition in day-to-day life meaning that the same types of events are observed many times and new types of events become less common.

In this section, we explain how the recall policies are evaluated against an observation, we evaluate the effectiveness of this approach to event recall and also detail extensions to this approach that can make it more efficient in both the time and space.

We explain how recall policies are trained in Chapter 5.

4.4.1 Recall Policies for Common Generic Frames

Recall from Section 4.2 that we defined eight generic frames that are relevant to all events. Also remember that, after a number of observations, it may become clear that these frames may need to be specialised for particular types of events. When a type of event has not had a sufficient number of observations to train its own unique recall policy, that type of event and all frames contained within it inherit the recall policies of the parent frames. This is to ensure that we have some means to recall events that have not yet had a sufficient number of observations so that a more unique and accurate policy can be trained.

These policies were coded manually and have been written in such a way that they remain general enough to make sure that if an observation is not a new type of event, then the observation will be assigned to the correct type of event. However, they are so general that it is likely that an observation may also be assigned to event types to which it should not be assigned.

When inheriting policies, the child frame adopts the recall policy of the parent frame. For example, we have defined a generic *event* frame and we have manually coded a basic recall policy for the generic *event* frame. When we observe a new type of event, we create a sub-class of this generic *event* frame. However, as the new type of event has not yet had sufficient number of observations to train a unique policy, it takes on the policy that we manually coded for the generic *event* frame. It should also be noted that each event is made up of several different types of frames, the *action*, *location*, *time*, etc. and we also create sub-classes of each of these in our new type of event. Each of these frames will also adopt the policy of their respective parent frame.

While the default policies are manually trained or coded, the policies for the individual types of events are learned autonomously and incrementally.

Here we explain these policies and show that while they are effective at recalling events that should be recalled, they also recall many other events that should not be recalled. This explains the need to specialise the policy to narrow the retrieval to the most relevant events.

In the following, we present rules without including the default rule, which is always:

$$\textbf{if } true \textbf{ then } no\_match$$

To evaluate the rule we use a generic frame and compare this to an observation of something that we believe might be an instance of that frame type. We compare the values of slots that have the same name. If the slot values are the same then the slot's *match value* value is *true*, otherwise it is *false*.

$$\textbf{if } slot_{generic}.value = slot_{instance}.value \textbf{ then } slot_{generic}.match\_value \leftarrow true$$

The conditions of the rules refer to the *slot match value* which is a boolean value. In the context of a rule, we refer to this value by the name of the

slot. For example, refer to the rule assigned to the generic *event* data type below.

event:

$$\textbf{if } match(action) \textbf{ then } match$$

When trying to establish if an observation is an instance of an event type, we compare the value of the *action* slot in the observation with the value of the *action* slot in the generic. If the values are the same then we conclude that the observation is an instance of an event.

action:

$$\textbf{if } match(precondition) \wedge match(effect) \textbf{ then } match$$

predicate:

$$\textbf{if } match(name) \wedge match(argument) \wedge match(arity) \textbf{ then } match$$

object:

$$\textbf{if } match(type)\ then\ match$$

time:

$$\textbf{if } true \textbf{ then } no\_match$$

location:

$$\textbf{if } match(name) \textbf{ then } match$$

temporal relation:

$$\textbf{if } match(all\ slots) \textbf{ then } match$$

connected_event:

$$\textbf{if } match(event) \wedge match(temporal\ relation) \textbf{ then } match$$

The rule for the *temporal relation* frame may seem confusing. The condition, $match(all\ slots)$ means that for the system to conclude that something is an instance of a *temporal relation*, all slots in the instance must have the same value as all slots in the generic. We present this rule with the condition, $match(all\ slots)$, because each slot in a *temporal relation* frame is one of Allen's temporal intervals [52], and as there are quite a few of these it more compact to write the rule as above.

4.4.2 Distinguishing Between Episodic and Semantic Memories

In Section 2.5, we described some of the theories surrounding episodic memory in cognitive psychology. Recall that Tulving [60] distinguished between episodic and semantic memories. He stated that episodic memories are recollections of personally experienced events and that each episodic memory is contextualised. This means that episodic memories have a time and a location associated with them. In contrast, semantic memories are known facts and entities, the events surrounding which happened independently of the person's experience.

Therefore, Tulving proposed that declarative memory should be split between episodic memories and semantic memories. Similarly our collection of memories is also split. Semantic memories are non-episodic declarative concepts. All types of events that are semantic memories have been observed more than once. As more observations of a given type of event are noticed, we are able to induce a more general representation of an event type. This reasoning may determine that the spatio-temporal aspects of the event that were a part of the event in some of the earlier observations are no longer considered relevant and so are removed and the remaining information is stored in the semantic memory.

This is not to say that episodic memories "become" semantic memories. Some information from the episodic memory is retained because it is relevant information and this part forms a semantic memory. For example, when learning that Edmund Barton was the first prime minister of Australia, the experience of learning that information remains an episodic memory, however, some of the information learnt through that event is stored as a semantic memory.

Separating memories into episodic and semantic stores has a practical advantage. As our collection of events grows over time, the recall time also increases. By splitting events into two separate collections we can increase the efficiency of recall. This is our primary justification for splitting the events into two separate collections. There are no other additional benefits to be gained from making this distinction apart from improved efficiency. Thus, our reasoning for calling the two collections of events episodic memories and semantic memories is largely due to historical reasons. The types of events stored in the semantic memory collection will likely have less episodic qualities than those events stored in the episodic memory collection. Historically, in cognitive psychology, events with these qualities have been called semantic events. Thus, we have chosen to call this collection the semantic episode collection.

4.4.3 Event Recall Using Ripple Down Rules

In Section 4.3.1 we gave examples of how an RDR can represent a recall policy and explain how they are used to recall those events.

In Chapter 2.5, we reviewed the most common approaches to event retrieval in case-based reasoning (CBR). Event or case retrieval is most commonly implemented in two phases. The first phase usually involves some form of *shallow* query that returns candidate matches for a new observation. The second phase selects the case from the candidate matches that is the best match to the observation.

We similarly adopt this model of a two-phased retrieval pipeline.

4.4.3.1 The Retrieval Pipeline

In this section we describe the retrieval pipeline. We show how recall policies are evaluated against an observation and how to determine if the observation is a new type of event or if it is an instance of a type of event already in memory. We also give examples to explain the advantages of this method over other approaches to event recall in a setting where an agent must co-exist with other agents in a dynamically changing environment. However, before proceeding, we explain how we differentiate between *critical* and *non-critical* actions.

4.4.3.2 The Difference Between *Critical* and *Non-critical* Actions

Section 4.2.4 explained that an event is created when the agent notices a change in the state of the environment. There are a couple of problems associated with this approach that we would likely to address. First is that the effects of an action sequence that has been learned are assumed to be the instantaneous observation of a state change. In reality, there may be multiple effects to any given action and they may be observed at different temporal points. Assuming that the effect of the action is only what was instantaneously observed could have potential implications. The most significant is that important information relating to the action may not be included in the action model. However, as our learning method allows for human intervention, it is possible to manually correct this in the event that it does occur. It is also possible to include new information in an action model on subsequent observations of that type of action.

The second and arguably more significant problem is that even inconsequential state changes can lead to events being created when they should not. To prevent an agent from creating spurious events, the robot must have an understanding of the significance of the end state of an observed action. We can achieve this by defining a set of conditions that describe states of the environment that, if observed, constitute a significant state,

which we refer to as a *critical* state. If we observe a *critical* state then we conclude that the action that resulted in the observed state is a *critical* action. A simple example of one of these states is:

$$on(floor, broken\_glass)$$

We assume that unless explicitly told otherwise, the world is in equilibrium. That is, there is no reason to assume that an event has occurred.

Once a *critical* state in the world has been observed by the agent, an event is created. Note, however, that just because a state change doesn't result in a *critical* state, it doesn't mean that we do not take note of it. In fact, this is how we learn new action models. The following example clarifies this.

In planning, the objective is to achieve a goal and it can be assumed that the goal is a *critical* state. In the PDDL planning language [135] we specify a domain that informs the agent about the actions available to achieve that goal. Assume that we have a goal where a person should be holding a glass of water.

$$holding(Person,\ water)$$

An action model should specify an action whose effects satisfy the goal. As actions are described by their preconditions and effects, the preconditions setup sub-goals that must be satisfied first. Consider the following sequence of actions that achieve the goal:

$$holding(Person,\ water)$$

as illustrated in figure 4.4.

To achieve the goal, five actions must be executed, where only the last one achieves the desired effect. However, we only wish to create a single event to represent this sequence. However, suppose the agent has not been endowed with the domain knowledge of how to accomplish a given

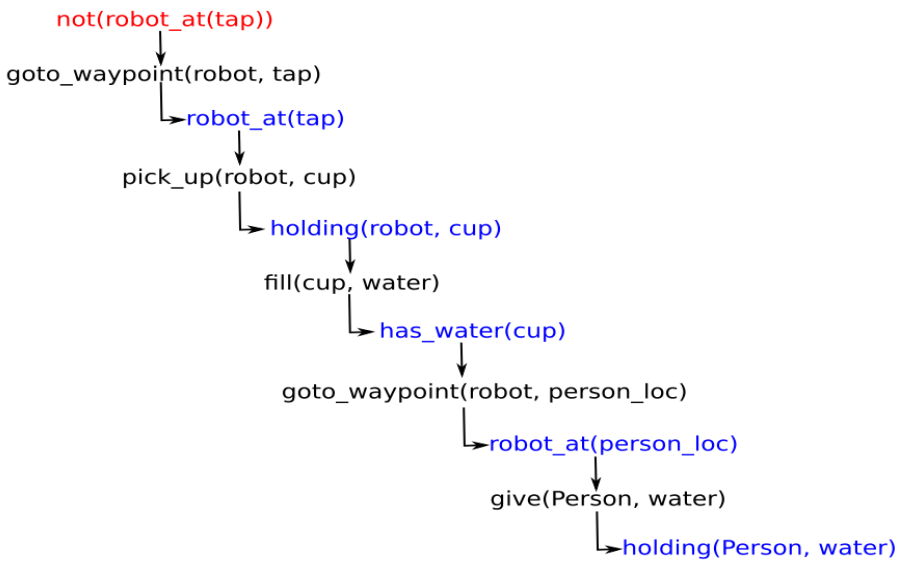


Figure 4.4: The initial state is shown in red. Actions are in black and the effects of an action are pointed to by a hooked arrow. The preconditions of an action are indicated by a straight arrow. Preconditions and effects can be conjunctions as well as single predicates.

task. Through observations of state changes, it is possible for the agent to build this action sequence and so it is able to learn how new goals can be achieved. If we assume noiseless observations then algorithm 26 describes this process. Note that noisy observations can be handled by generalisation over multiple instances of the action sequence.

Note that algorithm 26 only shows a high level description of what happens when actions are or are not considered *critical* actions.

Also note that what is considered a *critical* action is not only something that results in a *critical* state but any succeeding action that resolves this state. This is because a *critical* state is one that is often undesirable and therefore the robot must attempt to address it. For example, if the *critical* state is that broken glass is on the floor then a plan that resolves this state is to clean the broken glass.

In reality, determining if an action is a *critical* action or not is quite challenging. *Critical* states in an unstructured environment must be prioritised.

Algorithm 26 newActionSequence: Learning new action sequences

Variable *HolderAction* stores the *non-critical* actions that form part of the action sequence observed prior to observing the *critical* action. This is stored as the last *HolderAction* in an array of *HolderActions.* Record both the time and the location of these actions as this is relevant to the start and end time/location of the event that will eventually be created when the agent observes a *critical* action.

struct HolderAction
    action: Instance of *action*
    time: Instance of *time* {*The time this action was recorded*}
    location: Instance of *location* {*The location this action was recorded*}
end

newActionSequence()

```
1: action_sequence = new List of HolderAction
2: observation = action that was observed
3: for all e ∈ effect of observation do
4:    if e is critical state then
5:       critical_action = new HolderAction(observation, current_time,
         current_location)
6:       action_sequence.put(critical_action)
         comment: This is where we call Algorithm 20
7:       makeEvent(action_sequence)
8:    else
9:       this_action = new HolderAction(observation, current_time, cur-
         rent_location)
10:      action_sequence.put(this_action)
11:   end if
12: end for
```

For example, the agent may observe two actions happening at the same time. Let us assume that both actions are *critical* actions and therefore we create events for both actions. Recalling these events recalls the entire sequence of events that leads to a behaviour that the robot can execute and that attempts to address the *critical* states.

Assume that we label the actions *action* $\mathcal{A}$ and *action* $\mathcal{B}$ and that the effect of $\mathcal{A}$ is that there is dirt on the floor and the effect of $\mathcal{B}$ is that there is broken glass on the floor.

Even if we observe *action* $\mathcal{A}$ first, *action* $\mathcal{B}$ must take priority because it is of greater significance. Therefore, we must order our *critical* states based on their significance.

While this may seem like a clear solution to the issue of which types of actions should take priority, it is further complicated by the fuzzy nature of people's psyche. For example, consider the action sequence in figure 4.4. Here we have two actions that are *goto_waypoint* actions. It is reasonable to assume that these are not actions around which individual events would normally be created but rather they would be considered intermediary actions. However, consider a situation where a person explicitly asks the robot to leave. Our model may tell us that a person who is frustrated is a *critical* state, the solution to which is to leave the room. Given the context of the situation, the *goto_waypoint* action is now relevant as it has resolved our *critical* state, or at least improved it!

This problem can be solved by once again introducing Ripple Down Rules. Recall that the main advantage of RDRs is that they can learn incrementally as new evidence presents to the system. For matching policies, RDRs are used in both the sub-class and the super-class of the frame type. However, when deciding if an action should be considered a *critical* action or not, we apply the rule only to the super-class, i.e. the *action* frame type.

We choose the default rule to be:

if *true* **then** *non_critical*

We then add an exception to this default rule so that the rule is now:

if *true* **then** *non_critical* **except**
if *state* $\in$ *critical state vector* **then** *critical*

However, an action can also resolve a previous *critical state*. Therefore, another rule is added as an exception to the default and the rule is now:

if *true* **then** *non_critical* **except**
if *state* $\in$ *critical state vector* **then** *critical*
else if (*critical state*) $\wedge$ (*action resolves*) **then** *critical*

This policy states the following:
If a *critical* state is observed then the action is a *critical* action. However, if a *critical* state has already been observed and it has also been observed that that *critical* state has been resolved, then the action is also a *critical* action.

To date we have found that only these two rules are necessary. However, as we have highlighted throughout this dissertation, the advantage of using RDRs as policies is that the policy can be incrementally updated. So, if another case were to present whereby an action should be considered a *critical* action, then that rule can be added to the policy.

Previous work, such as Nason and Laird [137], uses numerical values to judge the significance of state-action pairs. Their work is an example of a hybrid-AI model where reinforcement learning techniques are crossed with symbolic practices. Depending on the state, the same action may return a different value to the state-action pair. This is something that we feel could be complimentary to the work that we are conducting here. However, by using RDRs rather than numerical values we have some key advantages,

the main one being that we can more aptly capture the contextual setting under which actions should be considered *critical*.

4.4.3.3 What Are *Critical* States and How Are They Arranged in a Hierarchy

A *critical* state is something that may but should not physically occur and therefore must be resolved.

Using an RDR policy applied to the super-class *action* frame, we can determine whether or not an action is a *critical* action. In this section we explain what a *critical* state is and how they are arranged in a hierarchy so that they can be prioritised.

As the world is represented as a collection of predicates, it is logical for us to also represent *critical* states as a collection of predicates. We also group certain *critical* states together that are similar to one another. For example, we might define a type of *critical* state called *hazard* and group *critical* states that represent hazardous states under this type, such as, for example, $on(floor, broken\_glass)$ and $on(floor, water)$, as both of these are dangerous states that could lead to a person getting hurt.

Types of *critical* states are arranged within a hierarchy in order of priority. We also order all of the *critical* states that are grouped under the same type in a hierarchy. This means that within a given class of *critical* state we have an order of priority. For example, one may consider a broken glass to be more significant than a slippy floor and so under the *hazard* class, a broken glass would be prioritised over a wet floor.

We represent a type of *critical* state as $\mathcal{C}$, where $\mathcal{C}$ is a tuple consisting of a set of members, $\mathcal{M}$, a level of significance, $\lambda \in \mathbb{N}$ and a set of children, $\mathcal{T}$, where $\forall t \in \mathcal{T}, t \rightarrow \mathcal{C}'$, such that $\mathcal{C}'$ represents a sub-type of $\mathcal{C}$.

The set of members, $\mathcal{M}$ are the *critical* states, represented as predicates, that are all of the same type. Referring to the same two examples from before, $on(floor, broken\_glass)$ and $on(floor, water)$, these are both mem-

bers of the *hazard* class.

However, the *hazard* class can also have sub-classes, for example, a *trip hazard* or a *slip hazard*. Each one of these sub-classes can also have members, so in the examples that we have used so far it is likely that we would in fact include the state $on(floor, water)$ as a member of the *slip hazard*.

This is a recursive structure meaning that each $t \in \mathcal{T}$ is also a type of *critical* state that can also have sub-classes. We therefore represent the *critical* state tuple as follows:

$$\mathcal{C} ::= \langle \lambda, \mathcal{M} | \mathcal{T} \rangle$$

It is possible that there can be multiple different types of *critical* states that are at the same level of priority. When this happens it is because we could not prioritise one over the other and so when the agent observes two members of different types but at the same level it deals with them on a first come first served basis.

Critical states that are sub-classes of other *Critical* states are ordered hierarchically within the parent state. See figure 4.5 for clarity.

Initially, and for the purpose of evaluation, we chose the *critical* states based on the environment that we tested in. We have also manually encoded the hierarchy of these *critical* states. Manually encoding a hierarchy is not ideal and is something that we would like to address in future research. For example, it should be possible to add to and learn new *critical* states as the robot observes the environment and automatically learn the significance of that *critical* state with respect to other *critical* states. However, what we were trying to prove was that by using this model an agent could ignore an event that is not relevant, thus it is adequate to select the *critical* states that we believe are likely to be observed. We are aware of this limitation however and have suggested it as an extension to this research.

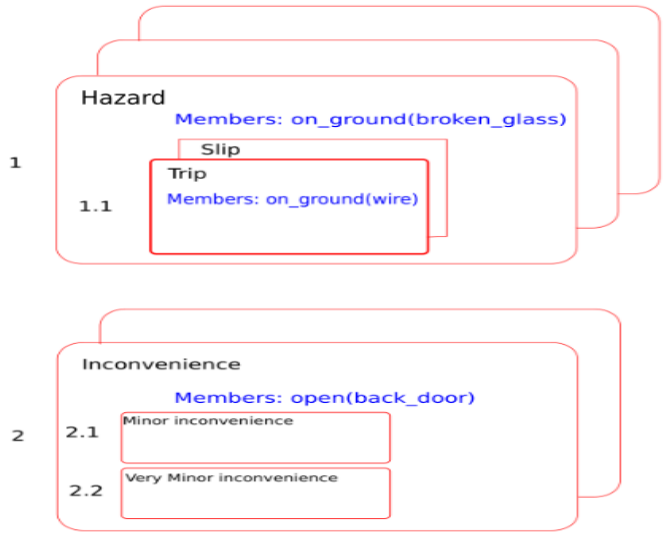


Figure 4.5: Each box represents a type of constraint. The hierarchy of types is organised from top to bottom. Shadow boxes represent other types of *critical* states at that level and boxes within boxes represent sub-classes of the *critical* state type that they are embedded in. For example, here we have a type called hazard and it is our most significant state, along with some others not named in this diagram. The only member of hazard is the broken glass predicate and so this takes priority. Other types of hazards are trip and slip hazards that we consider to be of equal significance. We prioritise all members of the parent *critical* state type before we handle any of the members of the children. We must prioritise all hazards before we deal with any inconveniences.

4.4.3.4 When Should an Agent Recall or Create an Event

Before returning to the main subject of this section, namely how events are recalled from memory, we want to very briefly explain when it is that an event should be recalled from memory or when a new type of event should be created and added to memory. Recall that in Section 4.2 we stated that a new type of event is created when we fail to recall any other type of event either in episodic or semantic memory.

While this is true, the point at which we create the new type of event is not at the exact time when the agent fails to recall any of the other types of events. In Section 4.4.3.2 we noted the policy for determining whether an action is a *critical* action or not and the policy is that if the effect of the action is a *critical* state or rather if at least one of the effects of the action is a *critical* state then the action is a *critical* action.

At this point, we attempt to recall an event from memory. This is a deliberate constraint that we place on our recall pipeline. We include this constraint for the same reason we constrain the point at which a new type of event is created in memory. Recall that we only create new types of events when an agent observes a critical action to prevent the system from creating multiple different types of seemingly irrelevant events. For similar reasons, we do not want to attempt to recall events on every minor state change in the environment. This would be inefficient and time consuming. Thus, we constrain our recall policy to only recall events when a *critical* state has been observed. If we are successful then we recall the entire event sequence which concludes with an event, the action of which is an action that the agent can execute to resolve the *critical* state.

However, in order for that final event to be a part of that event sequence, we need to wait for it to be observed. Therefore, when we observe a *critical* state and we cannot recall any event from memory that relates to that *critical* state, then we must assume that at some point the agent will observe another event that resolves this *critical* state and thus we must wait until this event is observed before we can create a new type of event in memory using the process outlined in Section 4.2.

Returning once again to the recall process. Like in CBR, our algorithm is a two-phased approach and depending on the necessity may be run twice. This is because initially we aim to match the observation to an event in semantic memory. The reason for this is that each type of event in semantic memory has been observed a sufficient number of times for a unique recall policy to be defined for that type of event. Therefore, if the observation is in fact an instance of a type of event in semantic memory then we are more likely to recall the correct type of event from semantic memory and that type of event only.

In querying the events in episodic memory, we are using the default recall policies, which are the policies that are inherited from the default generic frames that we outlined in Section 4.4.1.

First Phase Recall:
The first phase involves a very simple *shallow* query where we return any events that have at least one or more slot values that match the same slots in the observation. These matches are called candidate matches. When initially trying to determine if the observation matches an event in semantic memory this can be quite an effective step to take. This is because the types of events in semantic memory are the most generalised versions of those types of events and so there is only a minimal amount of irrelevant information. Thus, we can filter out quite a large sample of semantic memories immediately.

We have found this to be less effective however when querying episodic memories. This is because the events in episodic memory have not yet had a sufficient number of observations for us to logically induce a more generic representation of that type of event. Thus, they typically contain a lot of random or irrelevant information, some of which is likely to match something in the observation.

It is also true that as our database of episodes grows, the first phase recall is likely to return a significant number of candidate matches. The first phase recall is largely an implementation detail and we mention it only because it is a common practise within Case-Based Reasoning recall. Returning a

large number of candidate matches is not ideal however it will limit to an extent the number of matches that the second phase recall must evaluate. Thus, it provides some relief to the more complex part of this recall policy.

Second Phase Recall:
The second phase of our recall pipeline is where we make the most significant contribution to this field. It is at this stage that we are using RDRs to establish if the current observation event is an instance of any of the events either in semantic or episodic memory.

In algorithm 27 we have left out some of the details such as **handleManyValidMatches** function. This is because the details of this function are more aptly covered when we explain the recall policy training procedure.

The most important function in algorithm 27, is the **evaluateRules** function. This is where an observation is evaluated against the recall policy for each type of event that is returned as a candidate match.

Algorithm 28 evaluateRules: Evaluating Recall Policies for Episodic Memory Retrieval

---

Input: observation, candidateMatches {We are assuming that we have already executed the shallow query.}
evaluateRules(*observation*, *candidateMatches*)

```
for all match ∈ candidateMatches do
  eventMatch ← doFramesMatch(match, observation)
  if eventMatch then
    return true
  end if
end for
return NULL
```

When evaluating the rules, we compare all common slots in the observation to the candidate match and compare the results of the comparison against the condition in the match policy for the particular type of frame that we are dealing with. Figure 4.6 shows a very rudimentary example of this for a simple event. Algorithm 28 through 36 clarifies this process for the **evaluteRules** method.

Algorithm 27 recallEvents: Recall Pipeline Summary

Input : event_collection ← *semantic* {This is the name of the collection that we initially query. We always start by querying the semantic collection and so this variable is initialised to *semantic*}
Input : observation
Comments:
6: *This is the point that we know a new type of event has been observed but we are waiting to observe the event that resolves the critical state that has been observed before creating this new type of event in memory*
11: *Nothing returned from semantic memory so repeat the process on episodic memory*
14: *We have recalled an event from memory and only one event so we return it. We use the observation to induce a more specialised policy as will be explained in Chapter 5*
16: *At this point we need to specialise the policies for each returned event because only one should be correct*

recallEvents(event_collection, observation)

```
1: candidates ← shallowQuery(observation, event_collection)
2: validMatches ← evaluateRules(observation, candidates)
3: if validMatches is NULL then
4:     if event_collection is episodic then
5:         while ¬ resolveEventObserved do
6:             doNothing
7:         end while
8:         createNewTypeOfEvent(observation)
9:     else
10:        event_collection ← episodic
11:        recallEvents(event_collection, observation)
12:    end if
13: else if validMatches ≠ NULL ∧ length(validMathces) is 1 then
14:    return validMatches
15: else
16:    handleManyValidMatches()
17: end if
```

The first step in this algorithm is to determine if the two frames that we are comparing, match each other. If we are comparing two event frames, one being a generic event in memory and the other being an observation event, and the **doFramesMatch** method return *true*, then we conclude that the observation is an instance of the candidate event type.

Algorithm 29 doFramesMatch: Checking if Two Instances Match

Input: *candidatematch* **as** *cm*
Input: *observation* **as** *o*

Comments:
1: *Returns all slots. Refer to algorithm 22 for slot structure*
5: *We don't have the slots for this frame so we need to call an external function to get the value for slot called sname in the observation instance*
6: *If slot values are null, then we can not do a comparison*
14: *Returns a tuple. The didEvaluate is a boolean which is true if a rule fired and the conclusion is the conclusion of that rule. Candidate match stores slot match value for each slot which is used when evaluating the rules*
16: *The conclusion will be either match or no_match*
doFramesMatch(cm, o)

```
 1: slots = cm.getSlots()
 2: for all slot ∈ slots do
 3:    sname ← slot.name
 4:    svalue_candidate ← slot.value
 5:    svalue_observation ← o.getValue(sname)
 6:    if svalue_candidate is NULL ∨ svalue_observation is NULL then
 7:       continue
 8:    end if
 9:    if doSlotsMatch(svalue_candidate, svalue_observation) then
10:       slot.match_value ← true
11:    end if
12: end for
13: parent ← cm.getParents()
14: (didEvaluate, conclusion) ← evalMatchRDR(cm, parent.matchRule)
15: if didEvaluate then
16:    return conclusion
17: end if
```

In comparing two frames, we can only compare them to each other if they have at least one common ancestor. For example, an event in memory may be represented as a generic frame type that is a sub-class of the generic

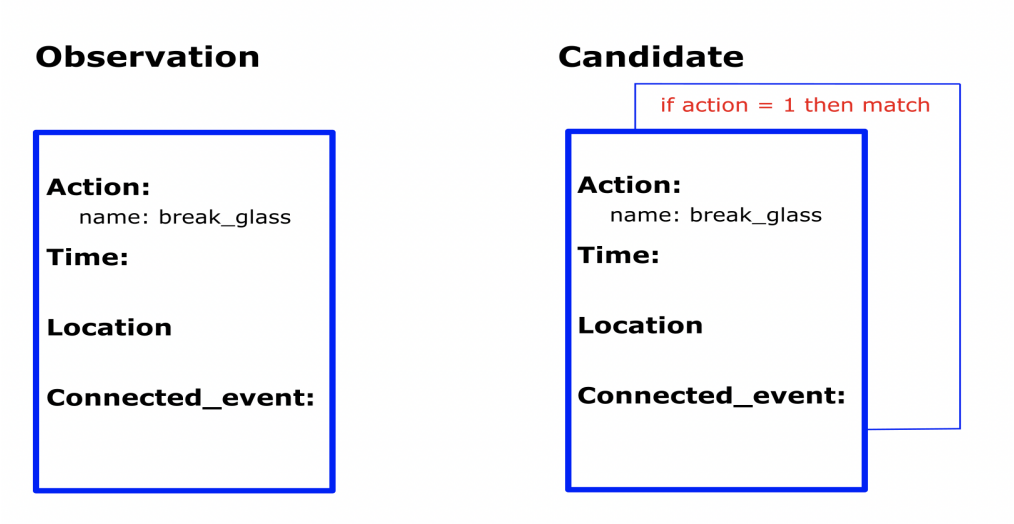


Figure 4.6: Both boxes are *pseudo-representations* of a very simple type of event. The box on the left is the observation. The box on the right is a candidate match. The box on the right is the exemplar for the generic *event* type that has at least a partially trained recall policy. The shadow box represents that generic event with the RDR in red. The RDR in this case states that if the value of the action of an event that has been observed matches the value of the action in the generic frame for this event then the observation is also an instance of the candidate event type. In this case that would be true and so the candidate event would be returned as a valid match to the observation

event frame. Whereas the observation is just an instance of the generic *event* frame. Even though the immediate parent of both instances is not the same, we can still compare them against each other as they have a common ancestor in *event*. We compare only slots that have the same name as each other and check whether the data in those slots match, if they do then the slot *match value* variable for that slot is assigned *true*. We use this value when evaluating the matching policy for this type of frame.

Algorithm 30 doSlotsMatch: Checking if the Slot Values Match

Input: candidate value ***as*** cv
Input: observation value ***as*** ov
Comments:
1: *cv/ov are the values associated with the two slots that we are currently comparing*
2: *If values are null then they can't be compared and so we assume they do not match*
5: *If the variable types are not the same then they can't be equal*
8: *Slot values can be instances of other frames and so we recursively call the process for these two values*
15: *By now we have established that the slot values are lists. Slot values that are lists can be two types, multivalued or not multivalued and they must be handled separately. We clarify this further below*
doSlotsMatch(cv, ov)

```
 1: if cv.isNull ∨ ov.isNull then
 2:    return false
 3: end if
 4: if type of cv ≠ type of ov then
 5:    return null
 6: end if
 7: if type of cv is Instance ∧ type of ov is Instance then
 8:    return doInstsMatch(cv, ov)
 9: end if
10: if type of cv is not List then
11:    return cv.equals(ov)
12: end if
13: if ¬multivaluedSlot then
14:    if cv.containsClass(instance) then
15:       return doListsWithFramesMatch(cv, ov)
16:    else
17:       return cv.equals(ov)
18:    end if
19: else
20:    return doListsIntersect(cv, ov)
21: end if
```

Algorithm 30 requires further clarification. In this algorithm we check if the value of two slots match. Although this may seem trivial, the values of a slot can be another frame, in which case we recursively start the process again. Slots can also contain *lists* as their values. This is again more complicated because a slot can contain a *list* in two ways. The first is that the slot can be multivalued, the second is that the slot can be single valued

but have a list variable as the only member. We specify the process for confirming if two lists that have frames as member match in algorithms 31 to 34.

Algorithm 31 doListsWithFramesMatch: Check if Two Lists With Instances Match

Input: candidate list ***as*** cl
Input: observation list ***as*** ol
Comments:
5: *One of the main reasons we avoid lists is we should be sure of the order of elements in the list also*

doListsWithFramesMatch(cl, ol)

```
1: if cl.length ≠ ol.length then
2:     return false
3: end if
4: for all (index, element) ∈ cl do
5:     if ¬doInstsMatch(element, ol.at(index)) then
6:         return false
7:     end if
8: end for
9: return true
```

Algorithm 32 checks if two lists intersect. Lists can either contain instance frames or not. If two lists contain instance frames then checking if those two lists intersect is more complicated as we have already clarified. We outline the process for checking if two lists with instance frame intersect in algorithm 33.

Algorithm 32 doListsIntersect: Check if Two Lists Intersect

Input: candidate list ***as*** cl
Input: observation list ***as*** ol
Comments:
4: *Assuming that no term is an instance then we check if any of the other terms are common to both lists. If they are then the commonTerm function returns true*

doListsIntersect(cl, ol)

```
1: if cl.containsClass(Instance) then
2:     return doListsWithInstancesIntersect(cl, ol)
3: else
4:     return cl.commonTerm(ol)
5: end if
```

Algorithm 33 doListsWithInstancesIntersect: Check if Two Lists With Instances Intersect

Input: candidate list ***as*** cl
Input: observation list ***as*** ol
Comments:
1: *This returns the instance pairs that we referred to earlier. Both of the instances in each pair are of the same type and so we compare the instance frames in each pair to see if they match*
3: *the first of the pairs is always from the candidate and not the observation*
doListsWithInstancesIntersect(cl, ol)

```
1: instPairs ← pairInstances(cl, ol)
2: for all pair ∈ instPairs do
3:     if doInstsMatch(pair.first, pair.second) then
4:         return true
5:     end if
6: end for
7: return false
```

Algorithm 34 takes two lists that have instance frames as members and groups the instance frames that are of the same type or have the same parents, into pairs. Recall that we mentioned that instance frames can only be compared for similarity if they have at least one common ancestor. However, a list within the context of a frame is not constrained to have only one type. Nor are lists constrained to have types ordered in any particular way. Consequently, any given list might contain frames that have different parents. Thus, if we were to apply a brute force comparison of two lists, we will likely be trying to compare two frames that do not have a common ancestor. As we have already stated, this is not possible and so this approach would be inefficient. To maximise efficiency we group instance frames that have the same parents and compare each pair of instance frame. We can then find the intersection of each of the individual instance pairs.

Algorithm 34 pairInstances: Pairing instance frames together

Comments:
5: *If the parents are the same then we create an instance pair* **Input**: candidate_list **as** cl
Input: observation_list **as** ol
pairInstances(cl, ol)

```
 1: return_list = new List
 2: for all c ∈ cl do
 3:    this_pair = new List
 4:    for all o ∈ ol do
 5:       if c.getParents().equals(o.getParents()) then
 6:          this_pair.put(c)
 7:          this_pair.put(o)
 8:          return_list.put(this_pair)
 9:       end if
10:    end for
11: end for
12: return return_list
```

When comparing the items of a *multivalued* slot to the items of another *multivalued* slot, we only look for some intersection between the two. In other words, to be able to decide that two *multivalued* slots match one another, we only need there to be one common term between the two slots and that common term does not necessarily need to have the same index in both lists. Thus, in the context of a rule, one can think of the terms of a multivalued list as being disjunctions of one another. For example, consider we had the following slot called *drinks* and the value of this slot in the generic was:

$$whiskey,\ beer$$

If the value of the drinks slot is relevant to the recall of the event and if the agent observes either whisky or beer as being a member of the drinks slot in the observation it will conclude that the two events match. Another way that this can be written is:

if $whisky \vee beer$ **then** $match$

On the other hand, when comparing the values of two single valued slots when those values are lists, we compare the two lists in the same way that we compare any other variables. Thus, in order to conclude that two lists match, all of the values in both lists must be the same. In the context of a rule we can consider all values of these types of lists to be conjunctions of each other. Thus, if both whisky and beer were relevant to the event above then this rule would be:

if $whisky \wedge beer$ **then** $match$

There is a reason for differentiating between these two different types of lists and we should also note that when dealing with lists it is most common for us to be dealing with items from a multivalued slot rather than a list from a single valued slot. Consider the representation that we have for an action, which has preconditions and effects. The preconditions and the effects of an action are conjunctions of predicates: $\langle P_1 \wedge \cdots \wedge P_n \rangle$, and in the frame representation, each predicate is an item in a multivalued slot. It is unlikely, when comparing an observation event to an event in memory that the state of the world will be the same even though the events might be the same. Other information, previously unknown, will almost certainly have been added or removed in between observations of this type of event and so an exact match of the preconditions and effects will be unlikely even though the events might be the same.

Thus, if we wish to see whether preconditions between an observation of an action and a generic action in memory match, we look only for an intersection between the items in the slots. Similarly for the effects. More formally speaking:

$$\forall A, B\{list(A) \wedge list(B) \wedge multival(A) \wedge multival(B) \wedge intersect(A, B) \rightarrow match(A, B)\}$$

However, there may also be a case where all elements of a list must match for the slot to match. For example, consider the event is of a meeting and before that meeting can take place all attendees must be present. We might then have a type of object called *attendees* whose value is everyone involved in the meeting. Hypothetically, it might be the case where if all attendees are not present then a different meeting is taking place. In this case, we would have a single valued slot, whose value is the list of attendees. If the observation of this does not match exactly then we conclude that it is a different meeting.

One further difficulty arises when we have multivalued slots when one or more of the items is another instance. When this happens we must split the multivalued slot into sub-lists, each sub-list containing instance frames of the same type. We then compare the sub-lists that have frames that are of the same type with one another to see if there is a common term between the sub-lists. We do this as per algorithm 34.

For example, consider we had two frames and one of the slots contained information about objects that had been observed in the environment. It might be a case that in one of the events we saw two chairs, a dog and a cat and in the other we saw three chairs but no dog or cat. In the former frame, we would split the list into three sub-lists with the first sub-list containing the instance frames of the two chairs, the second containing the instance frame of the dog and the third containing the instance frame of the cat. In the latter, we would split the list into a collection of sub-lists however this collection size would be one and contain the instance frames of the three chairs. Only the chair frame type is common to both and so we would check for any intersection between these two sub-lists. If there is a common chair to both then we would assign the slots match each other.

When running this comparison it is easier for us to create another list of *instance pairs*. For example, consider the two *chair lists* we have just discussed. If the first list is represented as:

$$\langle C_a, C_b \rangle$$

and the second as:

$$\langle C_c, C_d, C_e \rangle$$

then we create another list:

$$\langle\langle C_a, C_c\rangle, \langle C_a, C_d\rangle, \langle C_a, C_e\rangle, \langle C_b, C_c\rangle, \langle C_b, C_d\rangle, \langle C_b, C_e\rangle\rangle$$

When then iterate over this list of *instance pairs* and check if any two instances in each pair match one another.

Before detailing the algorithm to evaluate the match rules for each data type, recall that we record whether the slots in the candidate and observation matched. We use this value to evaluate the matching policies.

Algorithm 35 evalMatchRDR: Evalute the Match Policy for a Given Type of Data

Input: candidate instance **as** ci
Input: matchRule
Comments:
2: *This is a boolean variable. If the rule fires this is set to true*
5: *Checks if a condition was satisfied*
9: *If there is no exception then set the rule to be the alternative to the current rule. Eventually this will be null and the while loop will end*
12: *return whether any rule fired and the conclusion of the last rule to fire*
evalMatchRDR(ci, matchRule)

```
 1: RDR rule = matchRule
 2: conditionSatisfied ← false
 3: while rule ≠ null do
 4:     lastRule ← rule
 5:     conditionSatisfied ← evaluateCondition(ci, rule.condition)
 6:     if conditionSatisfied ∧ rule.exception ≠ null then
 7:         return evalMatchRDR(ci, rule.exception)
 8:     else
 9:         rule ← rule.alternative
10:     end if
11: end while
12: return ⟨conditionSatisfied, lastRule.conclusion⟩
```

A condition is a tuple consisting of a functor, f and two arguments, v_1, v_2. The functor can be one of three types: it can either be a boolean comparison (==), a logical *and* ($\wedge$) or a logical *or* ($\vee$). In the event that the functor is a boolean comparison, the second variable in the tuple is the slot name and the third is the boolean value indicting if the compared slots matched. In the event that it is a logical *and* or a logical *or*, the remaining two values are other conditions.

For example, if the rule were:

$$\textbf{if}\ match(action)\ \textbf{then}\ match$$

that is another way of saying:

$$\textbf{if}\ match(action)\ ==\ true\ \textbf{then}\ match$$

Therefore, the condition tuple in this case would be:

$$\langle ==,\ action\ match\ value,\ true \rangle$$

However, if the rule were:

$$\textbf{if}\ match(action) \wedge time\ \textbf{then}\ match$$

then condition tuple in that case would be:

$$\langle \wedge, action\ match\ value\ ==\ true, time\ match\ value\ ==\ true \rangle$$

Algorithm 36 evalCondition: Evaluate the Condition for a Given Rule

Input: candidate instance **as** ci
Input: condition
Comments:
1: *Caters for the default rule*
18: *At this point the condition is a tuple with a functor, ==, a slot name and an expected match value. If the actual match value of the slot is equal to this then return true*
19: *We record whether the slots matched when compared*
evalCondition(ci, condition)

```
1: if condition is true then
2:    return true
3: end if
4: if condition.functor is and then
5:    if evalCondition(ci, condition.v1) ∧ evalCondition(ci, condition.v2)
      then
6:       return true
7:    else
8:       return false
9:    end if
10: end if
11: if condition.functor is or then
12:    if evalCondition(ci, condition.v1) ∨ evalCondition(ci, condition.v2)
       then
13:      return true
14:   else
15:      return false
16:   end if
17: end if
18: sname = condition.v1
19: didMatch ← ci.getMatchValue(sname)
20: if didMatch.equals(condition.v2) then
21:    return true
22: else
23:    return false
24: end if
```

4.5 Conclusion to Event Retrieval

We have presented a novel approach for event retrieval using Ripple Down Rules as recall policies. The method is intended for robots that operate in an unstruc-

tured, partially observable environment with people or other robots and which have a wide variety of tasks where it is impossible to know in advance what those tasks might be.

This is in contrast to almost all other domains for which research in either case or event retrieval has been evaluated. As we will demonstrate when we present the evaluation of this method in Section 6.2, most other approaches work best in domains where there is either a single objective (goal) or a finite list of known goals. In our work however, this is not the case, and no assumptions are made regarding the expected behaviour of a robot. In the context of our research, the purpose of episodic memory is so that a robot can better recognise situations in which it is required to assist people. In an environment like a game, where the robot is the only agent and episodic memory only serves to better the robot's performance, an approach in which the qualitative similarity between an observation and an event in memory, exciting an attention mechanism above some threshold, may be more effective. This is because a game can typically rely on the same information being relevant to every situation and the number of goals that the agent needs to achieve is finite and known.

In many domains this is not the case. We require an approach where the information that is pertinent to a particular type of event is prioritised in the recall policy for that type of event. Furthermore, it is essential that the recall policy for any type of event can be incrementally updated as new and more relevant information regarding that type of event is presented. This will become clearer when we describe how we train recall policies.

We have demonstrated an approach to episodic recall that is capable of capturing the contextual information that is relevant to different types of events. As our method uses a different representation of context to other methods in the literature a direct comparison is difficult. However, we will show in our evaluation that our approach is effective at differentiating between different types of events in different contexts and can consequently accurately recall the correct event from memory.

That is not to say however that our approach is without flaws. An agent must initially be provided with apriori information relating to the environment and to the structure of the data representing the various components of an event. We also hand encode the hierarchy of *critical* states which must be addressed.

While these assumptions and constraints are not insignificant, we believe that they can be addressed. In Chapter 7 we suggest some possible extensions to this research that may address these problems.

Another notable draw back is time complexity. Recall policies are evaluated in linear time $O(n)$, so as the number stored episode grows, the run time grows linearly. We have, to some extent, already addressed this issue. By first splitting our memories between episodic and semantic memories there is less of a demand on each individual query and by using a two-phased approach, we reduce the number of episodes for which we must evaluate the recall policies.

However, we can further improve the time complexity by compiling our recall policies into a structure similar to a RETE network [138]. Such a network would evaluate an RDR only when the values referenced in the condition are present. This means that our retrieval algorithm could be much improved in time complexity as we would not need a linear traversal of each event, evaluating the RDRs one at a time.

Chapter 5

Training Recall Policies

5.1 Introduction

So far we have explained how recall policies can be used to retrieve an event from memory, with specific reference to how different recall policies can be assigned to different types of events, capturing only the information that is relevant to that type of event. In this chapter, we explain the process by which these recall policies are learned.

Before beginning, we provide some further clarification on our terminology. We use the phrases "recall policy" and "matching policy" interchangeably. There is a reason for this. Matching policies are applied to generic frames and the objective of the matching policy for a given type of frame is to establish whether an observation is an instance of that generic frame. Events are just other types of frames and so an event recall policy is no different from an event matching policy. However, as we are trying to recall events from memory we often refer to the policy associated with a given type of event as that event type recall policy. With other types of frames, we refer to the matching policy of that type of frame.

Training policies through induction has gained some attention in recent years. For example, Law *et al* [139] learn Answer Set Programs, including normal rules, choice rules and constraints using Inductive Logic Reasoning. The primary difference is the type of policy that we are training. We have chosen to explore the possible benefits afforded by using Ripple Down Rule policies. Our motivation

for choosing this type of policy is that it can be trained easily and if necessary can be guided by a human to correct any errors in the learning.

There are two ways in which recall policies can be trained and each method has advantages and disadvantages. The first way is to train policies manually using a human trainer to guide the learning, as is consistent with how Ripple Down Rules are traditionally trained. As will be seen, in Section 5.2.1, there are few situations in which one would want to manually train a recall or matching policy.

Manual training involves asking the trainer a series of questions. Dealing with a single complex event or with a sequence of events can often lead to a lot of questions being asked and can prove vexing to the trainer. The advantage of this type of training, however, is that we can often train a policy to a high degree of accuracy with only a single observation of the type of event for which we are training. This is arguably the only advantage to this style of training.

The second way to train a recall policy for an event is by induction. Through this process, a policy is induced from a number of observations that are known to be a specific type of event. We do not bias the process in anyway, so it is possible that a policy can contain irrelevant information or it may be affected by noisy observations.

There is a difference between irrelevant information and noise. Irrelevant information is information that has been correctly observed but bears no relevance to the event. An example may be the time of day that a glass broke. Irrelevant information is rarely a cause for concern and our recall policies are usually not affected by this information. Noise on the other hand comes from poor observations and noisy observations can at times lead to relevant information not being observed correctly and thus not being included in the recall policy for that type of event. This is more problematic but as we will explain in the following sections it is still possible for us to manage these situations.

5.2 Training Policies

RDRs are most commonly trained by a human expert. However, if the system must ask the trainer many questions, this can be labour intensive, and inefficient. For example, in the case that we presented in Section 4.3, we use RDRs to

recognise fruit and initially a human trains the system by answering questions about the properties of the fruit. As there is typically a small number of properties, like *size*, *colour*, *shape*, etc., this is quite manageable for the trainer and an accurate classifier can be learned with little effort.

In our case, however, the data structures that we are working with can be quite complex. While it is possible to train event recall policies using only a human expert, that approach can lead to hundreds of questions being asked, which for obvious reasons, is not practical. Therefore, the system learns a recall policy by observing a series of events. This still involves using a human expert to assist, however it typically only requires about two or three questions per event. Thus, the learning system requires more examples than one driven entirely by a trainer, but it requires less human intervention.

We explain both techniques and outline the advantages and disadvantages of each. The main advantage of RDRs is their ability to learn incrementally without any need to rebuild a model. Whether a human expert explicitly trains the recall policy or the policy is learned from examples, amendments to the policy can be easily made with succeeding observations of an event type.

In the conclusion of this dissertation, Chapter 7, we compare our approach with common methods for training recall policies both in episodic memory and case-based reasoning. We outline the advantages and disadvantages of our approach and give examples of why the approach is well suited to robots operating in domestic environments.

5.2.1 Training Policies Through Human Guidance

In this section, we explain how an event recall policy can be trained using a human to guide the learning. The main advantage is that a recall policy can be trained with only a few observations, and often only one. An example of where this approach might be necessary or preferred to a more autonomous learning method, is if the type of event is very rare and therefore there are very few training examples. For example, if the type of event is an annual one, this might mean that it could take several years for a robot to properly learn how to recall this type of event. Therefore, strongly guided human training is preferred for rare cases.

However, as will become clear, when applied to training event recall policies, this style of training is generally discouraged. The reason that we discourage this type of learning is because an event is a complex, large data structure. When training an RDR policy on simple cases, like the example where our system learns to recognise fruit, a human-guided approach is acceptable. This is because only a small number of questions will be asked to train the policy. For events, this is not the case and it is possible that a significant number of questions will be asked to train a recall policy. This is the main disadvantage to this training method.

The default RDR covers the most common cases. If a new case is covered by the RDR when it should not be, an exception is added. That is, the RDR is *specialised*. When a case is not covered when it should be, an alternative is added, resulting in a *generalisation*. If a branch in the RDR tree is created due to noisy data, this can be corrected when further examples are observed.

Consider again the example that we presented in Section 4.3, where the objective is for the system to learn to identify fruit. The default rule is:

if *true* **then** *unknown*

When a new example of fruit is presented, the default rule fires but returns an incorrect conclusion. Because the rule fired and produced an incorrect conclusion, it must be specialised by adding an exception rule. The conclusion of the exception rule is the correct classification, provided by the trainer. The condition of the exception rule is obtained by examining the differences between the conditions in the rule that fired and the new case. For the default rule, the values associated with all properties of the new case are potentially relevant since the condition of the default rule is just *true*.

The system generates a set of questions for each property of the fruit. In the case of a banana the questions might be:

is it because **colour** = **yellow**?

is it because **shape** = **long**?

The trainer may answer *yes* to the first question and *no* to the second. Therefore, the following exception is added to the default rule and the RDR now becomes:

if *true* **then** *unknown* **except**
if *colour* = *yellow* **then** *banana*

Suppose the system is now presented with an orange. The default rule fires, but since it has an exception rule, it is evaluated but does not fire since its condition is not satisfied by the new example. The difference between the properties required by the last evaluated rule and the new case is that the colour is orange, so the trainer will be asked if the ruled failed because of this difference. If the answer is yes, then an alternative rule is added to generalise the RDR:

if *true* **then** *unknown* **except**
if *colour* = *yellow* **then** *banana*
else if *colour* = *orange* **then** *orange*

While this method is very effective for training systems where typically the number of questions being asked is small, this is not a practical solution for complex data, where a large number of questions may be needed to determine the conditions in a new rule.

This is the case for our episodic memory since an event frame can have many properties, including embedded frames of other types of data. Each generic frame has its own recall or matching policy and these policies are represented by Ripple Down Rules. Trying to train the recall or matching policy for an event type could require hundreds of questions. In the next section, Section 5.2.1.1, we present an example to show how even a simple event can require many questions to construct the recall policy.

Nevertheless human-assisted training may be needed in three circumstances:

1. In rare cases, as described earlier, this approach might be preferable to an

autonomous one.

2. If an event's recall policy has already had extensive training then it is likely that only a small number of cases will not be covered by the recall policy. Thus, the number of questions that must be asked to update the policy is likely to be quite small and manageable.

3. We can more effectively handle noisy observation. If the agent incorrectly observes something that it should not have or fails to correctly observe something that it should have, we can instruct the agent that this difference is not relevant and it should be ignored.

5.2.1.1 Manual Training of Episodic Recall Policies

When manually training policies, we compare two observations of an event against each other and determine if the values in each of the slots that have the same name are equal. Recall that a slot stores the values of a specific property of a given type of frame. So for example, if the type of frame that we are comparing is a *time* frame, then one of the slots might be called *hour* and its value might be 8.

Suppose we say that a given event is of type $\mathcal{X}$. If a slot, has as its value, an instance of another generic frame $\mathcal{Y}$, then we can say that $\mathcal{X}$ embeds $\mathcal{Y}$ or $\mathcal{X} \hookrightarrow \mathcal{Y}$. When this is the case, we train the matching policy for the generic frame $\mathcal{Y}$ first. This way, we have a policy that we can use to determine if the slot values in $\mathcal{X}$, that contain $\mathcal{Y}$ as a value, match one another. We do this recursively for every frame that contains other embedded frames.

For example, when determining if two events match one another, we use the recall policy associated with one of those events, for example:

$$\textbf{if } match(action) \textbf{ then } match$$

To conclude that the events match, we check that the values of the *action* slots match. We do this by evaluating the rule associated with the type of action contained in the *action* slot. Before we can do this, however, we must have a rule defined for that type of action. Thus, when training or updating event recall

policies we use a bottom-up approach. That is, we first identify frames whose slots contain only atomic values or values that are not instances of other frames and thus can be compared directly without needing to learn a new policy.

As already noted, when manually training RDRs, a user is asked questions that are generated by the agent, based on the differences between the case for which the current rule incorrectly fired and the new case for which the rule must be updated. By using the difference between the current rule and the case for which the rule fired incorrectly, the system is intelligently constructing questions. By doing this, we are not asked about irrelevant information that had no effect on the system's incorrect decision. This is one of the key properties of RDRs and one of the main reasons that we chose to use them. When training matching policies, it is not the values of the slots but rather whether the values match of two slots that are being compared that matters. Therefore, we also record whether the last time we compared two slots, if the values of those slots matched.

If we are training policies manually, we can begin to specialise the default policy after only the first observation. If training through induction, as we will explain at a later stage in this chapter, we must wait for at least a second observation before we can begin specialising the policy.

While it is possible to train a policy for an event after only the first observation, we do not recommend this. This is because we will be asked a significant number of questions. The reason for this is that the trainer is asked questions regarding the differences between the initial case and the new case that resulted in the rule being specialised. When specialising the default rule, everything is considered to be different as we have had nothing previously to compare it against. In other words, we must answer questions about every single slot and whether the values of those slots are relevant.

It is for this reason that, in very complex situations, a lot of questions can be generated to train the recall policy for an event type. Therefore, we use this style of training only in a very small number of situations.

The following algorithms, algorithms 37 to 39 outline the process for manually training a recall policy. Succeeding these algorithms is an example presented to demonstrate why this is an approach one would want to avoid.

If one chooses to train policies only by hand, we recommend that the trainer

waits for at least the second observation of that type of event. This is because it is possible that the number of questions asked to train a policy manually might be reduced dramatically if one were to wait for a second observation of that type of event. However, this might also not be the case and it depends on the nature of the event in question. For example, if the two observations of the event had much the same information, then we would only be asked questions on the differences of which, there would be few. If, however, the two observations had only a small amount of common information then we would be asked a large number of questions to generalise the policy. Thus, as a default, we avoid the manual approach to policy training.

Algorithm 37 trainRDR: Manual Training of Event Recall Policies

Input: candidate
Input: observation instance ***as*** observation
Comments:
1: *The generic frame of the candidate*
2: *We keep track of the last rule to fire. This is the rule to which we add an exception*
3: *The new rule to be added*
4: ***ask*** *is a function that takes a question as input, asks that question and returns true if the answer is yes*
11: *Need to train the policy for any embedded frames*
13: *Lists also contain frames that have policies that need to be trained*
trainRDR(candidate, observation)

```
 1: generic_to_train ← candidate.getParent()
 2: lastRule = generic_to_train.matchRule.lastRule
 3: exception = new RDR
 4: if ask(Do events candidate and observation match) then
 5:    exception.conclusion ← match
 6: end if
 7: for all s ∈ slots of cand do
 8:    svalue_candidate = s.value
 9:    svalue_observation = observation.getValue(s.name)
10:    if svalue_candidate instanceof Instance then
11:       trainRDR(svalue_candidate, svalue_observation)
12:    else if svalue_candidate instanceof List then
13:       trainListRDR(svalue_candidate, svalue_observation)
14:    else
15:       slot_match_value ← svalue_candidate.equals(svalue_observation) ?
          true : false
16:    end if
17: end for
18: exception.condition ← askQuestions(slots of candidate)
19: lastRule.exception.add(exception)
```

The *trainListRDR* function takes two lists as input and checks to see if either list contains instance frames. If they do then they must train the policies for the generic frames from which these instances inherit their properties.

Algorithm 38 trainListRDR: Manual Training of Data Types Contained Within Lists

Input: candidate list **as** candidate
Input: observation list **as** observation
Comments:
4: ***inheritsFrom*** *returns all of the ancestors of a particular instance frame*
trainListRDR(candidate, observation)

```
1: for all t_candidate ∈ candidate do
2:    for all t_observation ∈ observation do
3:       if t_candidate instanceof Instance ∧ t_observation instanceof Instance then
4:          if inheritsFrom(t_candidate).intersects(inheritsFrom(t_observation)) then
5:             trainRDR(t_candidate, t_observation)
6:          end if
7:       else if t_candidate instanceof List ∧ t_observation instanceof List then
8:          trainListRDR(t_candidate, t_observation)
9:       end if
10:   end for
11: end for
```

In algorithm 38, on line 4, we check that two instance frames have a common ancestor before we use them to update a policy for the candidate instance frame. This is so we don't try and compare, for example, an apple and an car in order to train the policy for the apple. However, if we were comparing an apple and an orange, they would have a common ancestor in *fruit* and so we would assume that there might be some common information between the two that could help train a policy for the apple.

Algorithm 39 askQuestions: Generating and Asking Questions for Training

Struct Condition **contains**
slot_name: *String*
slot_match_value: *Boolean*
end
Input: Slots of the candidate **as** slots
Input: The rule that fired incorrectly **as** rule ← Instance of RDR
Comments:
6: *We check if the current condition is already in the rule. If it is not then it hasn't been covered and so could be relevant to the new rule*
5: *We keep track of whether this slot's value was the same as the same slot in another instance, the last time they were compared*
7: *This function generates the question based on the current case and the rule that fired incorrectly and waits for a response. The question might be:* ***is the fact that the action slots match relevant***

askQuestions(slots, rule)
1: conditions = **new** List⟨Condition⟩
2: **for all** s ∈ slots **do**
3: currentCondition = **new** Condition
4: currentCondition.*slot_name* ← s.*name*
5: currentCondition.*slot_match_value* ← s.*lastMatchValue*
6: **if** currentCondition ∉ rule **then**
7: question = **createQuestion**(currentCondition)
8: **if ask**(question) **then**
9: conditions.**put**(currentCondition)
10: **end if**
11: **end if**
12: **end for**
13: **return** conditions

In the following example, we show how a simple event of a glass falling off a table is represented using frames and further demonstrate why you want to avoid manually training recall policies using this event as an example.

In this example, if one trains the events recall policy after only the first observation, the total number of questions asked to train this policy is 31. However, even if one were to wait for a second observation of this type of event it is likely that this number would not decrease by much. This is due to the nature of this event. Only the action is relevant and so it is likely that all other information, such as, the time, the location, etc., would be different on a second observation.

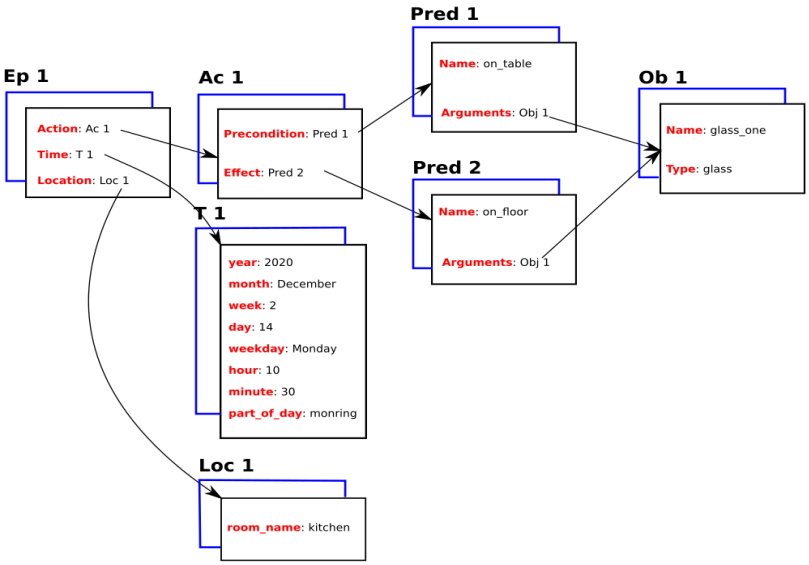


Figure 5.1: This is the first observation of the event for which we are manually training a recall policy. The event is depicting a glass falling off a table. The shadow boxes represent the generic frames of which the boxes containing the information are the exemplars. It is these generic frames for which we are training matching policies. Slot names align along the left hand side of these boxes with the values of these slots immediately to the right of each slot name. A value like *Action 1* for example is the unique ID of another instance frame of some action type. The time of this event is 10:30 am on December 14th.

Thus, demonstrating why this approach is to be avoided.

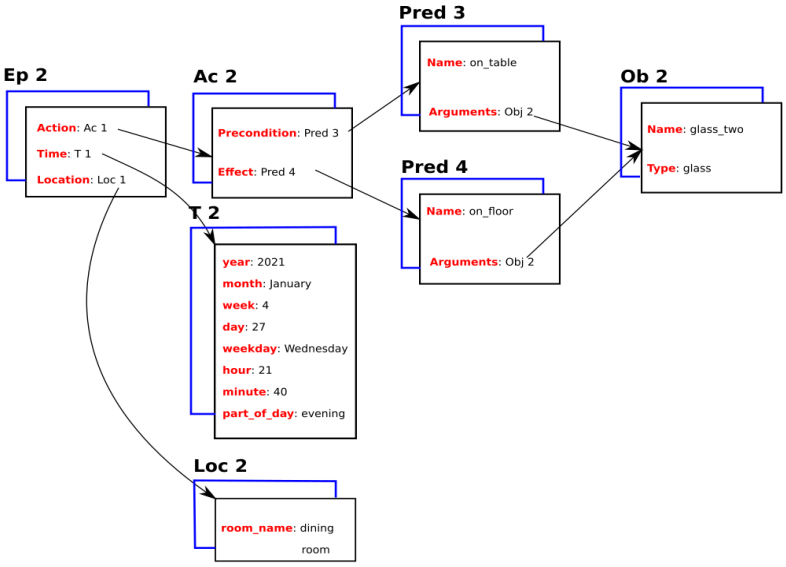


Figure 5.2: This is the second observation of the same type of event depicted in figure 5.1. It contains largely the same information except one might notice that the time of the event is different. For this type of event the time should be irrelevant

While it is possible to specialise a recall policy after only the first observation, it is almost never something that we would recommend pursuing. Thus, we present here an example where we are generalising a policy after the second observation of the event. The first and second observations of this event type are depicted in figures 5.1 and 5.2 respectively. Because we have already one observation of this type of event, the first specialisation of the default rule serves as the template rule for this type of event. That is to say, the policy for this type of event is as follow:

if *true* **then** *no_match* **except**

if $match(action) \wedge match(time) \wedge match(location)$ **then** *match*

Each of the *action*, *time* and *location* frame type contained within their respective slots also have rules of their own that we do not show here. However, one should note that the objects names are different, *glass_one* and *glass_two* respectively and that all of the information in each of the slots of the *time* frames are different and also the *location* frames.

Because all of this information is different, the rule:

$$match(action) \wedge match(time) \wedge match(location)$$

will fail and consequently the default rule will fire and return, *no_match*. Thus, it must be specialised and this specialisation will be a generalisation of the first rule that we had. We generalise this rule by discarding conditions that no longer hold true in each of the rules as we see necessary.

The values of the *action* slots were different, however, the value of the *action* slot is an action frame type and it also has a matching rule. This rule has also returned the conclusion *no_match* because the names of the objects further down the line are different and initially the action policy would have assumed these names to be relevant. Thus, the first question that we are asked by the agent is the following:
Question one: The values of the *name* slot in **Ob 1** are different. Can I discard this condition?

We would of course answer yes to this and the new policy for that type of object would be the following:

if *true* **then** *no_match* **except**
if $match(name) \wedge match(type)$ **then** *match*
else if $match(type)$ **then** *match*

This generalisation allows us to conclude that the *action* slots of the two observations of this event type do in fact match and thus we do not need to discard the $match(action)$ condition from the event policy and we are consequently not asked about it. However, the *time* slots and the *location* slots also have differing information. In the interest of brevity, we will not describe every question asked, however, the *time* frame has 8 slots all with different information, all of which needs to be discarded and the *location* frame has 1 slot with differing information. We are then finally asked if we can discard the *time* and *location* conditions from the event recall policy and this brings the total question count to 12.

The final policy would then be:

if $true$ **then** no_match **except**
  if $match(action) \wedge match(time) \wedge match(location)$ **then** $match$
  else if $match(action)$ **then** $match$

We should note that we have shown only a very simple example and in reality the structure of an event is even more complex with start and end times and locations, as well as other connected events. It is also likely that there will be more than 1 object. Thus, with very complex events, the question count could run into the hundreds and this is why we avoid this approach where possible.

5.2.2 Training Policies by Induction

We have so far detailed how a policy can be trained using a human guide. However, it is clear that this is not a practical way to train event recall policies in most situations. A more efficient way, however, is that policies can be trained by induction over several examples.

To learn a policy we use some background knowledge and a collection of facts to induce an RDR. The collection of facts are events that we know to be of the same type. The background knowledge consists of a pre-defined ontology of generic frames.

5.2.3 Adopting a Hybrid Training Model

Suppose we have a situation where the policy for a certain type of event is:

if $match(action) \wedge match(time)$ **then** $match$

We are ignoring the default rule here. Consider however, that in the case that the *location* values are the same we want to conclude that the events are in fact not the same. On another observation of the event, our recall policy will not consider the location of the event because the recall policy has not indicated

that the location is relevant and so if the *action* and *time* values are the same then the rule will fire and return the conclusion *match*. At this point we must specialise the rule to include the following:

$$\textbf{if } match(location) \textbf{ then } no\_match$$

Because the *action* and *time* slots are covered by the original rule the system only asks a question about the location. The new rule then becomes:

$$\begin{aligned}&\textbf{if } match(action) \wedge match(time) \textbf{ then } match \textbf{ except}\\ &\quad \textbf{if } match(location) \textbf{ then } no\_match\end{aligned}$$

Because we are only asked a single question or a very small number of questions to update this policy it might make more sense, in this case, to train the policy manually.

In reality, a situation like this is unlikely to occur and would in fact only be possible if we were to keep track of every observation of a given type of event rather than only storing the generic. There is good reason for this and it is to do with how inductive reasoning works. If the location at which an event took place is not relevant then it is not likely to be consistent across observations of that type of event and so we will induce a generic representation for that type of event that does not include the location. Thus, when we compare an observation of an event to a generic event in memory, the *location* slot values will never match because the *location* slot will have been removed from our generic representation of that type of event.

If coincidentally the location remains consistent across a number of observations of the event then it will be captured by the first rule. That is, our first rule would in fact be:

$$\textbf{if } match(action) \wedge match(time) \ \wedge match(location) \textbf{ then } match$$

Thus, when an observation of this event presents itself where the *location* slot

values are not the same, then this rule will not fire and the *default* rule will fire instead. Thus, we must add another exception to the default rule. The entire rule will therefore be:

if $true$ **then** no_match **except**
if $match(action) \wedge match(time) \wedge match(location)$ **then** $match$
else if $match(action) \wedge match(time)$ **then** $match$

However, it might also be desirable for us to keep a record of all observations of any type of event and not just the generic frame for that type of event. This is so that we can more effectively handle noisy observations. This we explained in Section 5.2.2. If we adopt this approach then we can use a voting system to determine which information should be included in the generic frame of that event. It is then possible for the location of the event to be included in the generic frame of that type of event but not be explicitly captured by an earlier rule. Consider the following example by way of explanation.

Workers in an office will often times organise weekly meetings. Typically these meetings will be at a specific time and a specific location. However, if for example the first two observations of a meeting happened in a different location then the agent would induce the following rule for this type of event.

if $match(action) \wedge match(time)$ **then**$match$

This is because the location of this meeting has been inconsistent across the two observations and thus would not be included in the recall policy for this type of event. How this is induced is detailed in the next section. As will become clear when we detail the process for inducing a generic representation of an event, the location information in the generic frame of the event will be null. If we only ever store one example of each event in memory, then this will never change. This is because we are storing what the agent believes is the most generalised version of that type of event and as it was previously proven that the location did not matter, then it will never be considered as part of the event.

However, if we store all observations of an event and use a voting system to decide which information is included in the generic frame of an event then this will not be the case. For example, consider that the same meeting was coincidentally held in the same room for the next five weeks in a row. The voting system should inform the agent that the location information should be included in the generic frame of the event. The policy however remains the same because the current policy:

$$\textbf{if}\ match(action) \wedge match(time)\ \textbf{then} match$$

has so far entailed all positive cases of this type of event.

Consider now that we are told we can no longer use that room because it has been booked for another meeting but that we are free to use any other room that we like. We must now update this policy to explicitly inform the agent that if we observe a meeting in this room in the future then it is a different meeting and so we should not recall this type of event from memory. In this case, because only a small number of questions need to be asked to generate the update, it is preferable to manually inform the agent about the update and so the new policy becomes:

$$\textbf{if}\ match(action) \wedge match(time)\ \textbf{then}\ match\ \textbf{except}$$
$$\textbf{if}\ match(location)\ \textbf{then}\ no\_match$$

It would however, be more effective to generate the initial updates to this policy using induction over several examples. In the next section, Section 5.2.4, we outline the process for updating a recall policy using induction.

While it is unlikely to occur, as we have demonstrated, there are few disadvantages to employing a hybrid-training method should it be required. One of the main disadvantages to using a human-guided approach is the number of questions that a user is asked. However, this is only the case if the event is new. Recall that RDRs construct questions based on differences between cases. When no policy has been trained, all observed information is perceived as different to the default policy. However, if the policies have been trained to a degree, prefer-

ably through induction, then it is likely that there will only be small differences between the observed case and the rule that fired incorrectly. Thus, a human can quickly update the policy to reflect the correct information, with relatively few questions.

5.2.4 Fully Automated Approach to Updating Policies for Event Recall

We have explained how in some contexts it might be preferable to update a policy using a hybrid approach where the first updates to a matching policy are learnt by induction over several examples and later updates are done manually. In almost all cases, with the exception of very rare types of events, it is preferable to initially train a recall policy using induction. In the last section we describe more formally the process for updating policies by induction.

The initial step to autonomously update a policy using induction is to find the most general representation of the event that we are training the policy for. This involves finding the intersection between the generic event from memory and the event instance that the agent observed. We are making the assumption that we have already established at this point that the observed event is the same type as the event recalled from memory. The agent is also endowed apriori with the pre-defined ontologies representing the relevant components of the event. That is to say, the agent is aware of the structure of the frames that we defined in Section 4.2.3. We also assume that each of the slots in each of the frames has been populated with the correct information, i.e. the *time* slot contains an instance of a *time* frame. This apriori information and these assumptions are important aspects of our automated learning pipeline.

Finding the intersection between two frames involves removing information that does not remain consistent across succeeding observations of that type of frame. Thus, in doing this we find the minimal generalisation of these two frames.

Algorithm 40 formalises the process for finding the intersection between two frames. The inputs to this algorithm are the generic frame in memory and the observation frame. The algorithm initially receives two event frames.

Algorithm 40 frameIntersection: Finding the intersection of two instance frames

We have previously referred to the events memory as candidates. However, when calling this algorithm we have already established the observation is an instance of the generic event in memory. Thus, we in this case we refer to this frame simply as the referent.

Comments:

7: *If the slots values do not have the same type then they cannot be the same and so we remove them*

9: *In this line we check to see if the slot values intersect one another. Slot values can be other instances or lists. Therefore we must check for intersections of these instances or lists. If they are atomic, strings, integers, etc. then we compare the values*

13: *If the slot values intersect then we replace the value of the slot with the value returned from the* ***slotIntersection*** *function. If there has been no further generalisation then there will be no change to the slot value of the generic*

Input: referent, observation

frameIntersection(referent, observation)

```
1:  slots = referent.getSlots()
2:  for all s ∈ slots of generic do
3:      slot_name = s.name
4:      slot_value = s.value
5:      observation_value = observation.getSlot(slot_name)
6:      if slot_value.getType() ≠ observation_value.getType() then
7:          referent.removeSlot(slot_name)
8:      end if
9:      slot_intersection = slotIntersection(referent_value, observa-
        tion_value)
10:     if slot_intersection.isNull() then
11:         referent.removeSlot(slot_name)
12:     else
13:         referent.replaceSlot(slot_name, slot_intersection)
14:     end if
15: end for
16: return referent
```

Algorithm 41 slotIntersection: Finding the intersection of two slot values

This algorithm takes the values in two slots and finds the intersection between them. If the values are not list or instance types then we check only that the values are the same and return null if not. If they are of list or instance types then it is slightly more complicated. We are assuming that we have already established that both values are the same type and so we do not check for that here.

Comments:

4: *Lists can also have instances and this further complicates finding the intersection between two lists. Algorithm 42 clarifies the process*

Input: referent_value **as** rv

Input: observation_value **as** ov

slotIntersection(rv, ov)

```
1: if rv instanceof Instance then
2:    return frameIntersection(rv, ov)
3: else if rv instanceof List then
4:    return listIntersection(rv, ov)
5: else
6:    return rv.equals(ov) ? rv : null
7: end if
```

Algorithm 42 finds the intersection of two lists. This is more complicated than it might appear since the elements of the lists can have different types: instance frames, other lists and atoms (strings, integers, etc).

Algorithm 43 takes lists with elements that have different types and sorts the elements by type into three sub-lists, the first element of which is a list containing instance frames, the second is a list containing other lists and the third is a list containing atoms.

$$\langle\langle inst_1, \ldots, inst_n\rangle, \langle list_1, \ldots, list_n\rangle, \langle atom_1, \ldots, atom_n\rangle\rangle$$

Recall that the objective is to find the intersection between the two original lists that are now sub-divided into three other lists,the first of which contains instance frames. Thus, we first find the intersection between the two lists that contain instance frames.

To find the intersection between two lists containing only instance frames we

must first group the instance frames into pairs depending on their parents. If two instance frames have the same parent then we must assume that there might be some commonality between them. The number of pairs of instances of a given frame will be equal to the number of instances of that frame in the first list multiplied by the number of instances of that same frame in the second. For example, consider the following two lists:

$$\langle dog_1, dog_2, dog_3, cat_1, cat_2 \rangle, \langle dog_4, cat_3 \rangle$$

These two lists are then grouped as follows:

$$\langle \langle dog_1, dog_4 \rangle, \langle dog_2, dog_4 \rangle, \langle dog_3, dog_4 \rangle, \langle cat_1, cat_3 \rangle, \langle cat_2, cat_3 \rangle \rangle$$

Algorithm 34 groups frames by their parents. We then find the intersection of each pair of frames per algorithm 40.

We then repeat this process for the list types by pairing lists together and recursively calling algorithm 42 on each pair. We do this because we must also assume that it is possible for any two lists in either of the original lists to have a common term. Thus, we group the list types as follows:

$$\langle list_1, list_2, list_3 \rangle, \langle list_4, list_5 \rangle$$

$$\langle \langle list_1, list_4 \rangle, \langle list_1, list_5 \rangle, \langle list_2, list_4 \rangle, \langle list_2, list_5 \rangle, \langle list_3, list_4 \rangle, \langle list_3, list_5 \rangle \rangle$$

Finally, we find the common terms between the atomic values and include these in the final result also.

Algorithm 42 listIntersection: Finding the intersection of two lists

Comments:

13: *We refer the reader to algorithm 34*

Input: referent_list **as** rl
Input: observation_list **as** ol
listIntersection(rl, ol)

```
1: return_list = new List
2: referent_split = splitList(rl)
3: observation_split = splitList(ol)
4: all_ref_instances = referent_split.atIndex(0)
5: all_obs_instances = observation_split.atIndex(0)
6: all_inst_pairs = pairInstances(all_ref_instances, all_obs_instances)
7: for all p ∈ all_instance_pairs do
8:     temp = frameIntersection(p.atIndex(0), p.atIndex(1))
9:     return_list.put(temp)
10: end for
11: all_ref_lists = referent_split.atIndex(1)
12: all_obs_lists = observation_split.atIndex(1)
13: all_list_pairs = pairLists(all_ref_list, all_obs_lists)
14: for all p ∈ all_list_pairs do
15:     temp = listIntersection(p.atIndex(0), p.atIndex(1))
16:     return_list.put(temp)
17: end for
18: for all t ∈ referent_split.atIndex(2) do
19:     if t ∈ observation_split.atIndex(2) then
20:         return_list.put(t)
21:     end if
22: end for
23: return return_list
```

Algorithm 43 splitList: Split a list into different frame types

Comments:

2-4: *Create lists for each type that we are dividing the original list into*

Input: list

splitList(list)

```
 1: return_list = new List
 2: instance_list = new List
 3: list_list = new List
 4: atom_list = new List
 5: for all val ∈ list do
 6:    if val instanceof Instance then
 7:       instance_list.put(val)
 8:    else if val instanceof List then
 9:       list_list.put(val)
10:    else
11:       atomic_list.put(val)
12:    end if
13: end for
14: return_list.put(instance_list)
15: return_list.put(list_list)
16: return_list.put(atomic_list)
17: return return_list
```

It is possible that the intersection of two events can be null. This happens when observations have noise. It is most common for this to happen when the agent fails to observe something that it should have. For example, if an object in the environment is relevant to an event and if, say, it is the only object that is relevant to that event, then it is likely that after a sufficient number of observations of that type of event, the generic frame will contain only that object, assuming that we have had noiseless observations. However, if on one of the observations of that type of event the robot fails to identify the object properly, it is going to assume that it is not relevant and that object will be removed from the generic event after the noisy observation. Because it is the only relevant information it is likely that no information will be contained in the generic event and it will thus be null.

We have already proposed a solution for addressing this in that we store all observations of each type of event in memory and use a voting system to decide which information is kept. While this would largely solve the issue, it comes at the cost of requiring significantly more storage.

The voting system that we could employ would be akin to a confidence interval used in machine learning. The confidence interval is an interval statistic that is a quantified measure of the uncertainty of an estimate. The challenge that we will face in employing this system is in determining how many observations of a given type of frame we require before we can get a meaningful estimate of the relvant information that should be kept.

There are also other situations where a voting system is preferable to decide which information is contained in the generic frame. Consider a situation where there is altering information for one type of event, for example, an event where a friend comes over to visit and in some cases the two friends drink whisky and on other occasions the two friends drink beer. If the first two observations of this type of event were to have different drinks then the agent would logically induce that the drinks are not relevant to the event even though they are.

For obvious reasons this would cause issues. By using multiple instances of the event however the agent can apply a voting system to determine which information should be included in the event's generic frame. After a sufficient number of observations it would conclude that both whisky and beer are relevant to this type of event. It would then put both drinks in the generic frame as objects of a multi-valued list. One should recall from algorithm 32 that when comparing two multi-valued lists, the terms within those lists are disjunctions of one another. Thus, we can inform the agent that the drinks are relevant and if the agent then observes the two friends drinking either whisky or beer on a future observation, then it will correctly recall this event.

Alternatively, we could adopt a hybrid model and directly inform the agent that both drinks are relevant to the event.

The space complexity of storing each instance of every event, however, is an enormous constraint. Each event contains many different frames, consuming a significant amount of storage. Thus, this is not the approach that we adopted for this work. In Chapter 7, we provide a more detailed explanation of the benefits of storing each instance of every type of event and how this can be implemented in future work.

On each pass of algorithm 40, the frame that is passed to the algorithm becomes more general, assuming that there are further generalisations to be made. Using the information contained in the generic frame for the event we can update the

matching policy as detailed in algorithm 44.

In analysing algorithm 44, recall how a Ripple Down Rule policy is structured and updated. An RDR begins with a default rule that returns the most likely conclusion when no further information has been provided that allows us to specialise the policy. In the case of a matching policy that rule is:

if *true* **then** *no_match*

When this rule fires incorrectly, it is specialised to account for the new case as has been described in Sections 4.3 and 4.3.1. Therefore, when updating a policy we assume that some rule has fired incorrectly or else there would be no reason to update the policy. The structure of an RDR captures the last rule that fired, the rule to which we are adding an exception.

There is one catch to this however. When we induce a policy for a type of event we induce this policy using the generic frame of that event type. Embedded within that generic frame will be other generic frames each with their own unique matching policy as has been outlined.

It might not be the case where all matching policies fired incorrectly, however the only information that the agent is aware of is that some rule incorrectly fired. Therefore, it must initially assume that all are incorrect and that each rule must now be specialised or generalised further.

If it is the case where one of the matching policies of one of the generic frames contained within the event fired correctly however, the newly induced rule will be identical to a rule already in the RDR matching policy for that generic frame. Thus, a quick check tells us that this policy does not need further specialisation.

The input to the **induceMatchPolicy** function is the generic frame that we are training the policy for.

Algorithm 44 induceMatchPolicy: Inducing a matching policy

Struct Condition **contains**
    slot_name: *String*
    slot_match_value: *Boolean*
end
Struct RDR **contains**
    conditions: *List*⟨ Condition ⟩
    conclusion: ⟨ *match, slot_match_value* ⟩
end

Comments:
7: *If the two slot value is a list then we need to induce a policy for all frames in the list*
Input: generic
induceMatchPolicy(generic)

```
 1: new_conditions = new List⟨ Condition ⟩
 2: slots = generic.getSlots()
 3: for all s ∈ slots of generic do
 4:    if s instanceof Frame then
 5:       induceMatchPolicy(s)
 6:    else if s instanceof List then
 7:       induceMatchPolicyList(s)
 8:    end if
 9:    temp_condition = createCondition(s.name)
10:    new_conditions.put(temp_condition)
11:    new_rule = new RDR
12:    new_rule.conditions = new_conditions
13:    new_rule.conclusion = match
14:    addRule(intersection, new_rule)
15: end for
```

Algorithm 45 induceMatchPolicyList: Inducing policies from lists

We induce policies for all frames in a list

Input: list

induceMatchPolicyList(list)

```
1: for all t ∈ list do
2:    if t instanceof Frame then
3:       induceMatchPolicy(t)
4:    else if t instanceof List then
5:       induceMatchPolicyList(t)
6:    end if
7: end for
```

Algorithm 46 createCondition: Creating a new condition

Here we create a new condition to add to the new rule that is being added to the policy.

Input: slot_name

createCondition(slot_name)

```
1: return_cond = new Condition
2: return_cond.slot_name ← slot_name
3: return_cond.slot_match_value ← true
4: return return_cond
```

Algorithm 47 addRule: Adding a new rule

It is worth noting that technically it is possible to have multiple parents. In reality this rarely proves to be the case with our work

Input: data_instance, new_rule

addRule(data_instance, new_rule)

```
1: data_type = data_instance.getParents()
2: for all d ∈ data_type do
3:    this_rule = d.matchRule
4:    last_rule = this_rule.lastRule
5:    if new_rule ∉ last_rule.exception then
6:       last_rule.exception.put(new_rule)
7:    end if
8: end for
```

5.3 Conclusion

In this chapter we outlined the process by which we are training a data matching policy. We present the results of this in Section 6.2.3 where we show how accurate recall policies can be trained after only a small number of observations which is

why this approach to learning has such significant contributions in the field of episodic memory in cognitive robots. We present 3 ways that recall policies can be trained and have outlined the advantages and disadvantages of each.

We require a system that can be updated dynamically, incrementally, as relevant information appears and captures only the information that is relevant to the type of event to which the policy is assigned. Ripple Down Rules provide us with all of these relevant capabilities and it is why we choose to engage them as recall policies.

Our approach also differs in that we do not attempt to fit a single retrieval cue or mechanism to all events in memory but rather assign individual policies to each type of event. Consequently it was necessary that these policies could be trained with relative ease and perform well over a large data set. The results, presented in Section 6.2.3 prove this to be the case.

Chapter 6

Evaluation and Results

6.1 Topological Mapping Evaluation and Results

Chapter 3 explained how we generate topological maps from two-dimensional occupancy grid representations of environments. The occupancy grid only shows free and occupied space and this does not necessarily correspond to a logical understanding of areas. A topological map provides the foundation on which a world model can be built and the world model provides the system with the necessary information required to model episodic memories.

Our approach to generating a topological map achieves state of the art accuracy. To evaluate our approach we use the precision and recall metric as proposed by Bormann *et al* [2]. This is the standard metric on which topological maps are evaluated. Our results show that, on average, we achieve 98% precision and 96% recall which, to the best of our knowledge, is the highest level of accuracy that has been achieved.

Before explaining this in further detail we briefly explain what these evaluation metrics mean and how it is possible to achieve different results in both categories.

6.1.1 Evaluation Metrics

When segmenting a topological map, the result is compared against a *ground truth*. Depending on the application, the environment may be segmented into different types of regions. In our case, as we deal primarily with domestic robots, we segment the environments into a collection of rooms and corridors and each room or corridor is given its own *ground truth*. In determining the *ground truth* of a region, people are asked what they believe to be the most accurate representation of a given room. For our evaluation we use twenty open source maps provided by Bormann *et al* [2] which already have an associated *ground truth* and an additional six maps that we have created ourselves that we believe show more complex environments and therefore more aptly demonstrate the capabilities of our approach.

Precision is defined as the maximum overlapping area of a segmented region with a ground truth region divided by the area of the segmented region. *Recall* is the maximum overlapping area of a ground truth region with a segmented region divided by the area of the ground truth region [2]. Precision is high if the segmented region is fully contained within the ground truth region and recall is high if the ground truth region is fully contained within the segmented region.

Let the area of the ground truth region be A_{gt} and the area of the segmented region be A_{seg}. The area of the overlap between the two regions is $A_{\{gt \cap seg\}}$. Therefore,

$$precision = \frac{A_{\{gt \cap seg\}}}{A_{gt}}$$

$$recall = \frac{A_{\{gt \cap seg\}}}{A_{seg}}$$

While our precision and recall figures vary only slightly, others have found that these figures can vary widely. For example, Bormann *et al* [2], achieve 93% precision, with 85% recall. Figures 6.1 and 6.2 show graphically how this is possible.

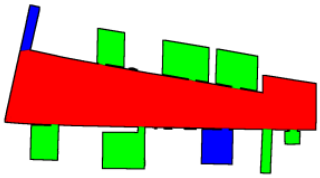


Figure 6.1: This shows a ground truth representation of the environment as determined by Mielle *et al* [28]

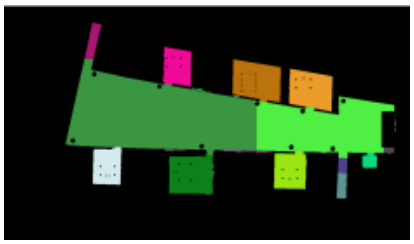


Figure 6.2: This shows the segmentation as determined by Bormann *et al* [2]. If one looks at the largest, centre room it is clear to see why they have achieved high precision but relatively low recall. Both segmented regions are fully contained within the ground truth and hence high precision is achieved. However, the ground truth is not even nearly contained within either segmented region and hence recall is low.

6.1.2 Results

We hypothesised that our approach to topological mapping would achieve state-of-the-art results using the precision and recall metric. In turn, our approach would produce the most accurate segmentation of an environment, giving a clear representation of the individual rooms and corridors that make up the environment. We also hypothesise that we achieve these results while softening many of the assumptions that previous attempts have made, as we have already outlined. We present our results in comparison with the most accurate attempts to date, using the same metric for evaluation.

On average, we achieve 98% precision and 96% recall and we evaluate this on a total of 28 maps, 6 of which we created either using a robot in a real environment or through an open source ROS bag file and the other 20 were provided as part of the open source data set by Bormann *et al* [2].

This compares with Mielle *et al* [28] who achieve 96% precision and 95% recall, Bormann *et al* [2] who achieve 93% precision and 85% recall and Fermín-Leon *et al* [140] who achieve 93% precision and 81% recall. The results were achieved on the open source data set provided by Bormann *et al* [2]. As will be clear, we also show results on some additional maps to show the capabilities of our system in more complex environments. However, we have not tested the other approaches on these maps.

One might compare our results to those of Mielle *et al* and conclude that we make only minor improvements. However, Mielle *et al* make a number of significant assumptions that make their work less generalisable to more complex environments. The first assumption is that environments must be empty. This is arguably the most significant assumption as unstructured environments are rarely empty and thus it would fail in almost any furnished house or office. They also make the assumption that environments must be convex or be relatively simply shaped. As can be seen in figures 6.3 to 6.11 where we graphically present the results of nine maps that we evaluated, this is not an assumption that we make and thus we can conclude that we achieve state of the art results while making no assumptions about the environment's contents or structure.

Each figure from 6.3 to 6.11 shows five of the same map at different levels of abstraction. The map on the far left is the raw occupancy grid, the second from the left is the occupancy grid after it has been cleaned as outlined in Section 3.2.2, the centre map shows both the Generalised Voronoi Diagram as well as the selected doorway (critical) points. The map that is second from the right shows all of the regions that we generate before they are merged and the last map shows the end result which is the fully segmented map into the rooms and corridors that construct that environment.

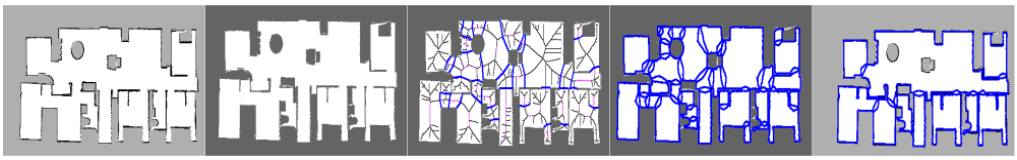


Figure 6.3: This is a map that we manually created to represent as accurately as possible a real house with multiple different rooms and obstructions

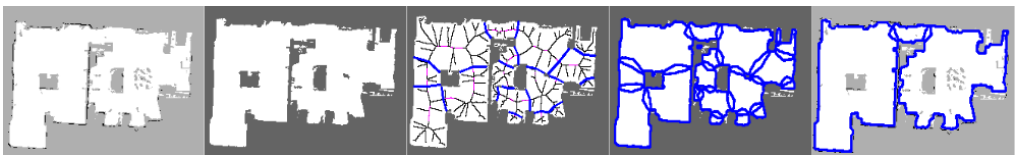


Figure 6.4: This is a map of our @Home testing arena at the University of New South Wales

Figure 6.5: This is a map of the arena for the roboCup@Home league in Montreal, 2018

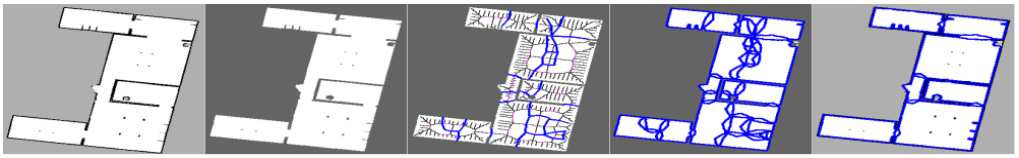


Figure 6.6: This is a map of one of the Gazebo environments provided by Robotis[141]

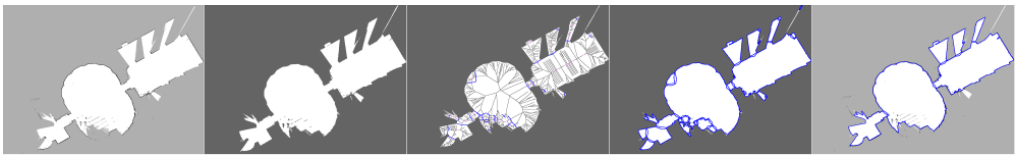


Figure 6.7: This is the first of two partial maps that we made using open source ROS bags files from Google cartographer [124]

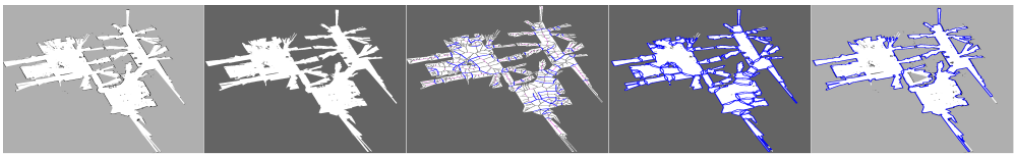


Figure 6.8: This is the second map taken from open source ROS bag file provided by [124]

.

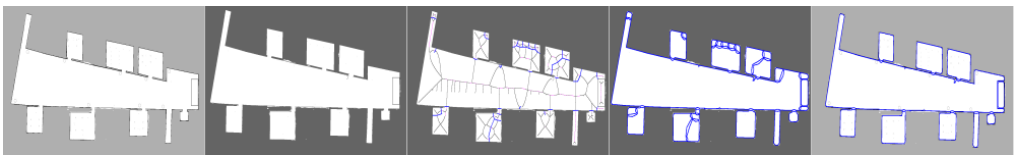


Figure 6.9: This is one of the maps provided in the open source data set from Bormann *et al* [2]. These have become the standard for evaluating approaches in topological mapping.

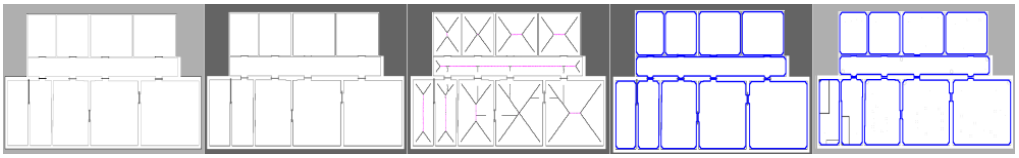


Figure 6.10: This map shows that in some, very rare cases, it is possible that the final stage of our pipeline has only minimal improvement over the second to last stage. As noted already, in a completely empty and convex environment, stages one to four of our pipeline are sufficient to accurately segment an environment into rooms and corridors. We have included this map, which is empty and convex to show this.

Figure 6.11: This map can also be found in the data set provided by Bormann *et al* [2]

In conclusion, we have shown how our approach to topological mapping achieves state of the art results while addressing some of the key assumptions that we identified in previous work.

6.2 Event Recall Results

One of the main claims of this thesis is that different types of events have different types of information that are relevant to those types of events and therefore require different recall policies. We also claim that a recall policy should only

capture the information that the system believes is relevant to an event type. We also note that an event is just one type of frame and that these claims extends to all types of frames.

We are hypothesising that by assigning unique recall policies to individual types of events we can successfully capture the contextual information relevant to that type of event. In turn, when recalling instances of this type of event from memory we can correctly recall instances of the type of event that we are observing and discard instances of other types of events, even ones that are qualitatively similar in nature. We further hypothesise that by using RDRs as recall policies we can train them to recall events correctly with only a minimal number of training examples. This is an important feature of our system as in order for robots to effectively co-exist with people, they must be able to learn quickly and effectively.

We use the term *believe* here intentionally. One can never be sure that one has accurately captured only relevant information and so we must assume that on some future observation we may need to generalise or specialise the policy to account for this. Our choice of policy type allows for this.

We have devised an approach that can correctly distinguish between different types of events given only subtle contextual cues. We believe our method to be more suitable in an unstructured, partially observable environment such as a house, compared with other methods that we have reviewed that are more suited to a game environment, as we will explain later in this chapter. It is true that other approaches may be effective in domains where the number of goals and the types of information presented are known and finite. However, we evaluate our system in a domestic robot environment that is unstructured and partially observable. In these types of environments the number of potential goals or types of information that are presented to the agent can be assumed to be finite.

Our evaluation consists of two main components. We must evaluate how successfully we can recall a given type of event or frame using the process outlined in Chapter 4. Secondly, we measure how many observations of an event type we require to train a recall policy for that type of event.

To evaluate our approach we must first create training and test data (see Section 6.2.1 below). We create these data by synthesising different instances of an event type. This process, as outlined in Section 6.2.2, allows us to create large, varied test and training sets from a small number of observations of any type of event

6.2.1 Creating Data Sets

Collecting data involves observing the environment and waiting for things to change. In collecting data we collect different types of events that an agent observes. In Section 4.2, we described the structure of an event and how it is constructed from other types of data such as *time*, *location*, etc. Thus, when collecting instances of events, we are also collecting a host of other types of data.

To collect these data, we need to run experiments in which an agent operates in a realistic environment and wait for changes to occur. We use a simulation rather than a physical environment for the following reasons.

Firstly, simulations allow us to obtain data sets in a much shorter time than using a physical robot. This means that we were able to collect a larger and more varied data set. It also means that we can re-run experiments far more effectively when needed.

Secondly, the objective of this thesis is to develop an system whereby an agent can effectively distinguish between the different types of events in memory using unique recall policies assigned to those individual types of events. Thus, providing a more effective approach to episodic memory in unstructured, partially observable environments. Evaluating our approach in a real world environment would require us to have a more complete and structured low-level robot architecture. For example, in using a simulated environment we are less affected by things like changing light conditions which on a real robot requires a recalibration of the vision system and delays experimentation. We also simplify computer vision problems by using QR codes to identify objects, as object recognition is outside the scope of this project. However, we do simulate sensor noise. The simulation allows us to evaluate our approach effectively, as it contains much of the complexity of a real world environment but also allows us to avoid time consuming tasks or problems outside the scope of the thesis.

The simulation that we use is an open source TurtleBot3 simulation in ROS/Gazebo provided by Robotis [141]. We have been careful to make sure that the environment is as varied as possible and we regularly move objects to different areas, mimicking as much as possible, a real world setting. Figure 6.12 shows an example of a setting that we created.

Figure 6.12: This figure shows a simple setting in our simulated environment. The robot is observing a table with a glass placed on top of the table. In the background it can also see a bookshelf.

We also note that for the purpose of collecting events for training and testing, we do not employ *critical* actions for determining whether an event should be created or not. This is to assist with creating as large a data set as possible. Thus, even noisy observations are noted as events and treated as such. Events that are created due to noisy or irrelevant observations are represented exactly the same as events created due to valid observations. Therefore, for the purpose of evaluation, we can treat noisy or irrelevant events as real events. Furthermore, we show towards the end of this chapter, by employing a method to detect *critical* actions we can significantly reduce the length of event sequences where several of the events in the sequence were created as a result of a noisy or irrelevant observation.

There is a difference between noisy and irrelevant observations. A noisy observation is something that has been incorrectly observed, for example, a false positive in object recognition. An irrelevant observation is something that has been correctly observed but it present or absence does not affect the observer event. For example, one may observe a pencil being moved from one side of a keyboard on a desk to the other side, but the side in which the pencil appears may not be relevant..

The final data set that we created has 331 types of frames, of which 40 are event frames, with each of these being a sub-class of a generic *event* frame.

This provides us with a highly varied data set to evaluate our approach. We tried as far as possible to create a data set where all of the event types had differentiating features that indicated why one was more suitable to be recalled than another. In some cases the differentiating features were only slight and in other cases they were more noticeable.

For example, consider two separate observations of a robot moving from room A to room B. These may appear to be two observations of the same type of event. However, it is likely that events of this nature would be part of a larger sequence and can be discarded as they are intermediary events. We have already explained why we do not discard them for the purpose of collecting data. Therefore, even though instances of both types of events have much the same information, a slight change in context, such as, for example, the time of day, can result in one recall policy firing and another not and so one of these events is recalled as well as all of the events in the sequence of which it was a member, and the other is not.

Of all of the events that we collected, the shortest event sequence was 1. That is, that events that are part of an event sequence of 1 are stand alone events and have no connected events associated with them. The longest sequence was 11, however, many of the types of events in this sequence were in fact either created through noisy observations or through intermediary events such as moving from one room to another. We will not explain all of the different types of events in detail but rather provide a brief explanation as to how some of these events were collected. For example, one of the events that we observed was that of a person coming over to visit. In fact, in our data set we have two types of events of this nature but we change the time of the week so as to trigger different recall policies. For example, a friend coming over on a Monday morning might mean that the robot should make two cups of coffee, whereas a friend coming over on a Friday evening might mean that the robot should get two beers.

Recall that the agent is looking for is a change in the state of the world. The agent is endowed with a world model that represents the agent's current beliefs about the state of the world. On an initial survey of the environment, the agent does not observe any other person, with the exception of the occupants of the

house. Therefore, if the agent, on a subsequent observation, sees another person, it concludes that something has changed and a new event is created. A human would refer to that something as a visit from a friend. However, to a robot, it is represented as an action with the preconditions of the action stating that the person-*Jack*-was not in the house, and the effects of the action stating that *Jack* is now in the house.

$$precondition \leftarrow not(at\_house(Jack))$$

$$effect \leftarrow at\_house(Jack)$$

When collecting events, we make sure that some of them are very similar, like the two events where people come to visit, but we change some of the contextual information. The purpose of this is to show how our recall mechanism is capable of distinguishing between two seemingly similar types of events based on some subtle changes in context. We also make sure to keep other types of events very different, to show that we are not constrained by the types of information presented and our system is capable of generalising to a wide variety of types of events.

For example, with the two events that represent friends coming to visit, we recorded one of these event sequences on a Monday morning and the other on a Friday evening. What the agent noted was that in the former case, the succeeding event was of two cups of tea being made and in the latter it was two bottles of beer being fetched from the fridge. The different recall policies for the visiting events should indicate therefore that the time of the week is important information to both events as very different succeeding behaviours are required.

Another example of an event that has been mentioned previously is a glass falling off a table. For this case, the recall policy should inform the agent that neither the time nor the location are relevant to and only the action of the glass falling and breaking should be taken into consideration when recalling the event.

6.2.2 Synthesising a Training and Testing Set

To synthesise training and test sets, we construct an generic frame for each type of event, a pre-defined recall policy for that type of event and a set of rules governing the kind of information that can be contained in an instance of that type of event. We then construct instances of the event type by randomly generating slot values that are consistent with these constraints. Since events are constructed from other types of data, the procedure is applied to all frames when generating slot values. Thus, to explain how we synthesise events, we refer more generally to how we synthesise frames.

The aim of the learning system is to be able to reverse engineer the hand-crafted policies from the data.

Before outlining the process of synthesising a data set we wish to clarify one point. Synthesising data was not a focal point of our research and consequently we have not investigated the extent to which this would scale to other domains. We chose to synthesise data so that we could quickly and effectively generate a training and testing set. We describe the process to show the reader that we generate a sufficiently random data set while still maintaining legal entries in each of the synthesised datum points. For example, we will not generate a datum point where the *time* slot has the value *dog* as this is not a legal value for time. We cannot say with any certainty that this approach will scale to other domains as synthesising data was outside of the scope of our research.

6.2.2.1 Pre-defining Recall Policies

We hand-craft policies that define the relevant information for each type of event. Each generic frame has its own recall or matching policy that determines whether an observation is or is not an instance of that generic frame. Therefore, we overwrite the default recall policy for each generic frame

if *true* **then** *no_match*

with one that defines the intended recall policy.

A simple example is a rule that we might append to a specific type of object that states:

$$\textbf{if } match(name) \wedge match(type) \textbf{ then } match$$

Policies for all 331 generic event frames can be created by manually training RDRs, as described in Section 5.2.1.1. These polices can then be used to generate training and test instances for learning, which attempts to reconstruct the manually created policies from the data.

6.2.2.2 Rules for Generic Frames

The *object* generic frame is at the top of our ontology. This type of frame and all sub-classes of it contain at least two slots, the object name and the object class. The name must be a unique identifier for this object and the class is the type of object. For example, a glass positioned somewhere in the environment is an instance of a sub-class of type *object* and the slots might be populated as follows: $name : glass\_one, isa : glass$.

We might then define a policy for this sub-class of *object* as follows:

$$\textbf{if } match(class) \textbf{ then } match$$

By this we mean that for this type of frame we consider it to be a valid match if the values in the *class* slots are the same. To state this more clearly, any glass in the environment is an instance of this sub-class of *object*. In order for us to synthesise other instances of this type of object we create what we refer to as a *rule frame*.

A generic frame specifies the legal values of the slots in the corresponding instance frame. For example, the a generic frame with slots,

$$\{name : [glass\_one, apple\_one, pear\_one], type : [glass, apple, pear]\}$$

specifies the legal values of the *name* and *type* slots. Type constraints can also

be numeric. For example, assume that we were dealing with a *time* type and the *hour* slot within that type. Only values in the range $0, \ldots, 23$ are legal. The slot values of synthesised training data are randomly selected from the legal values contained in the generic frames.

If the value of a slot is an instance of another frame, then the value of the slot in the generic frame is a generic frame for the type of frame contained in that slot. For example, when synthesising instances of an event, the time slot contains a frame of type *time*. Thus, in the *time* slot of the generic frame we will have another generic frame of type *time*.

Using the generic rule frame and the generic frame for the specific type of data, the manually created matching or recall policy can synthesise a new set of instances of that generic frame. The process is clarified in algorithms 48 to 53. We accompany these algorithms with relevant explanations.

Algorithm 48 synthesiseInstances: This function is executed until the required number of instance frames have been synthesised

We pass three arguments to this function: the generic frame from which we are trying to synthesise other instances; the generic "rule frame" specifying the permitted slot value ranges that we refer to earlier; and the number of instances we require.

Input: generic, rule frame **as** rf
Input: required number **as** size
Comments:
1: *Declare and empty list to store all the synthesised instances*
synthesiseInstances(generic, rf, size)
1: result = [. . .]
2: **for** $i = 0; i < size; i \leftarrow i + 1$ **do**
3: synth = **synthesiseInstance**(generic, rf)
4: result.**push**(synth)
5: **end for**
6: **return** result

Algorithm 49 is accompanied by comments to clarify some of the statements. We call the **getParents** function on the first line to gain access to the generic frame which, in turn, gives us access to the matching policy that we have pre-defined for that frame type.

Algorithm 49 synthesiseInstance: Synthesising events using matching policies and a set of rules

Input: generic frame and rule frame **as** rf. **Comments:**

2: *We create a new instance of this generic frame*
4: *This splits up the RDR into a list of individual rules with the conclusion "match"*
6: *Choose one of these rules to use at random*
9: *We don't have the slots of the rule frame but we have the name of the slot we are looking for so we can use this function*
10: *We call this function which will synthesise a value for this slot using the value of the generic, the policy and the rule indicating what we can include in this slot*

synthesiseInstance(generic, rf)

```
1: parent = generic.getParents()
2: synthesised_instance = new parent
3: match_rule = parent.match_rule
4: split_match_rule = splitRule(match_rule, [])
5: slot_names = generic.getSlots()
6: chosen_policy = chooseRandom(split_match_rule)
7: for all s ∈ slots do
8:     generic_value = s.value
9:     rule_value = rf.getSlotValue(s.name)
10:    synthesised_slot = synthesiseSlot(s, chosen_policy, generic_value,
       rule_value)
11:    put(synthesised_instance, s, synthesised_slot)
12: end for
13: return synthesised_instance
```

Algorithm 49, details how an one instance frame is synthesised. We must initially establish the type of instance frame that we are synthesising and create a new instance of that frame type. The generic frame stores the matching policy which we then use to choose values for each of the slots in the new instance. The first step is to split the matching policy to get all of the rules that conclude with match. Algorithm 50, describes this process.

Algorithm 50 splitRule: Split a matching policy

Input: matching policy **as** policy, all valid rules **as** rules
splitRule(policy, rules)

```
1:  while policy ≠ null do
2:    if policy.conclusion = match then
3:      rules.put(policy)
4:    end if
5:    if policy.exception ≠ null then
6:      policy ← policy.exception
7:      splitRule(policy, rules)
8:    else if policy.alternative ≠ null ∧ policy.alternative instanceof
      RDR then
9:      policy ← policy.alternative
10:     splitRule(policy, rules)
11:   else
12:     policy ← policy.alternative
13:   end if
14: end while
15: return rules
```

Algorithm 50, specifies the process of splitting an RDR into the individual rules that conclude with "match". This is one of the most important steps to this process. A matching policy is represented as a Ripple Down Rule, Since an RDR is a recursive structure of rules with exceptions and alternatives, for a given generic frame there may be several constraints defining conditions for which an observation is considered to be an instance of the generic frame. For example, the following policy for a *time* frame has two conditions that conclude with *match*.

if *match*(*hour*) **then** *match*

else if *match*(*weekday*) **then** *match*

A situation where this might be the case is if the event in question happened at a specific hour or range of hours every day, e.g 3-5pm, with the exception of Monday where it didn't matter what time of day it was. The generic frame would have a range of hours in the *hour* slot and Monday as the value of the *weekday* slot.

We would therefore use both of these rules when synthesising instances of this *time* type. We do this by synthesising values for each slot, one-at-a-time as per algorithm 51.

Algorithm 51 synthesiseSlot: Synthesise a value for a slot

Input: slot name **as** sname
Input: policy **as** p, *One of the rules from the matching policy that concludes with "match"*
Input: generic slot value **as** svalue
Input: rule slot value **as** rvalue, *The permissible values for this slot*
Comments:
4: *If we have a list of instances we call this function*
6: *If the slot contains neither an instance or a list then choose a value. If the slot is not relevant then a value is choosen at random from one of the permitted values defined in rvalue*
synthesiseSlot(sname, policy, svalue, rvalue)

```
1: if svalue instanceof Instance then
2:     return synthesiseInstance(svalue, rvalue)
3: else if svalue instanceof List then
4:     return synthesiseList(sname, policy, svalue, rvalue)
5: else
6:     return chooseSlotValue(sname, p, svalue, rvalue)
7: end if
```

Algorithm 51, outlines the process for synthesising a slot value, given the chosen rule on which to synthesise. In each rule that concludes with *match* we have a set of conditions that lead to that conclusion. Recall that when we use a matching policy to determine if an observation is an instance of a frame type, we compare all values in the slots of the observation with the values of the same slots in the generic frame in question. The conditions of a rule therefore state whether two slot values should or should not match in order for the rule to fire and return the conclusion *match*.

Using these conditions it is possible to synthesise a new instance frame that, although different, is still a valid instance of the frame type. For example, take the following rule applied to the *glass* object from earlier:

if *match*(*type*) **then** *match*

This rule tells us that any object in the environment that has the same *type* slot

value as the generic frame for this type of object, is also an instance of this type of object. However, as the *name* slot was not explicitly mentioned in the rule, we can conclude that the value of this slot does not matter. Therefore, we select at random one of the legal values from the *name* slot in the generic frame for our new instance.

In algorithm 52, we choose a value for the slot. We are assuming at this point that the slot value is atomic as all other types have been handled by this point.

Algorithm 52 chooseSlotvalue: Choosing a slot value

Input: slot name **as** sname
Input: policy **as** p
Input: slot value **as** svalue
Input: rule value **as** rvalue
Comments:
1: *If the slot is not included in the policy then we assume it is irrelevant and therefore choose a value at random*
4: *This statement checks if the name is in the policy and that the condition is that it should match the slot in the generic. This is what $\wedge true$ means in this context*
6: *Else the name is included in the policy but the policy states that it should not match the value of the slot in the generic. If this is the case then the chosen value cannot be the same as the generic*
chooseSlotvalue(sname, p, svalue, rvalue)

```
 1: if sname ∉ policy then
 2:     random_number = randomInt(length(rv))
 3:     return rvalue.at(random_number)
 4: else if sname ∈ policy ∧ true then
 5:     return svalue
 6: else
 7:     random_number = randomInt(length(rvalue))
 8:     temp = rvalue.at(random_number)
 9:     if temp = svalue then
10:         chooseSlotValue(sname, p, svalue, rvalue)
11:     else
12:         return temp
13:     end if
14: end if
```

In algorithm 53, we synthesise values for a list of instances. This is where we create varied slot values. If the slot is in the policy and the condition is it should match then we must synthesise at least one that matches. Otherwise we

synthesise any random number and are not too concerned if they do or don't match to each other.

Algorithm 53 synthesiseList: Synthesising values for a list of instances

Input: slot name **as** sname
Input: policy **as** p
Input: slot value **as** svalue
Input: rule value **as** rvalue
Comments:
1: If the slot is included in the policy then we select some of the instances in that slot at random to synthesise off.
6/7: *The special instance rule frame must be at the same index in the list as the frame to which it applies*
9: *If the slot is not included in the policy then we completely randomise this slot*
16: *This is the step that truly randomises what happens. By resetting the matching policy to the default policy for this generic frame, our system has no indication of how two instances of this frame type match. Therefore, all slot values are now randomised. Some coincidentally may match and others may not*

synthesiseList(sname, p, svalue, rvalue)

```
 1: return_list = [...]
 2: if sname ∈ policy then
 3:     length = randomInt(length(svalue))
 4:     for i = 0; i < length; i ← i + 1 do
 5:         random_number = randomInt(length)
 6:         temp_inst = svalue.at(random_number)
 7:         temp_rule = rvalue.at(random_number)
 8:         return_list.put(synthesiseInstance(svalue, rvalue))
 9:     end for
10: else
11:     length = randomInt(length(sv))
12:     for i = 0; i < length; i ← i + 1 do
13:         random_number = randomInt(length)
14:         temp_inst = svalue.at(random_number)
15:         temp_rule = rvalue.at(random_number)
16:         resetMatchRule(svalue.parent)
17:         return_list.put(synthesiseInstance(svalue, rvalue))
18:     end for
19: end if
20: return return_list
```

The final algorithm that we present in this process, algorithm 53, synthesising values for a list of frames. When the value of a slot is a list of frames, the process for synthesising values for that slot is more complex. On an initial observation of a particular generic frame, we assume that all information that was presented

as part of that observation is relevant. The reason for this is that each event is unique and we cannot know or make assumptions about the type of information that is relevant to any given type of event.

Let us assume that a particular observation of a given type of event had a list of objects that were present at the time of the observation. For example, on observing a glass fall from a table the agent may also note there are some apples, some oranges and a banana on the table. These may not be relevant and after subsequent observations of this type of event we would expect the robot to draw that conclusion. However, initially we cannot assume this.

If a slot is multi-valued slot and if the matching RDR states that this slot is relevant, then we must assume that one or more of slot values is relevant. Thus, when we synthesise the value of a slot that contains a list of other frames, we choose a number at random in the range of one to the number of frames in the slot. We then create that number of instance frames to go into the slot.

Using this method, we can also synthesise noisy observations. In the example above, only the glass object is relevant. However, we do not inform our algorithm of this fact. Thus, when selecting instance frames from the slot that holds those objects, it might select one of the other objects to include in the synthesised instance of this event rather than the glass. This simulates an observation of an event where there is missing data, often as a result of sensor noise.

6.2.3 Results

Our evaluation focuses on how successfully we can recall events from memory and how quickly event recall policies can be trained. When collecting data for training and evaluation, we assumed all actions to be *critical* in nature. Therefore, we also show how, when a method to differentiate *critical* actions is employed, we can reduce the size of large event sequences.

The results show that even with very complex event structures, where there are multiple different types of events as part of an event sequence, we can achieve high accuracy in our event recall with error rates typically ranging between 0.2% and 2%. By assigning a unique policy to each type of event and iteratively generalising and specialising each policy so that it captures, as best as possible,

the information that is unique to that type of event, we can specialise each policy so that it is in some way unique from every other policy and thus will be able to accurately distinguish instances of the type of event to which the policy belongs from every other type of event.

These results are very encouraging and demonstrate the capabilities of RDRs as event recall policies. Even as our database of events grows, it is possible to accurately capture the information that is relevant to each type of event, in that event's recall policy so that it is distinguishable from every other event in memory and thus we can recall events from memory with a very low error rate.

We present our results in two ways. First, we show how our recall improves on each training instance by demonstrating that the false positive and false negative rate decreases on each training instance.

Second, we show how quickly an event recall policy can be created from data. In Section 6.2.2.1, we explain how we hand-craft recall policies for each of the events in memory so that we can use these policies to create a synthesised data set. These are the policies that we want to recreate and thus, one of the metrics that we use for evaluation is how many training instances are required to recreate these policies on average.

We present the latter results in increasing order of event complexity. That is, we initially show the results for simple events, events that are not connected to any other event, and progress to the most complex results, those for events that are part of very long sequences of larger events.

6.2.3.1 Event Sub-class Results

In our first evaluation, we show the improvement of our recall after each training instance. We also check how many training instances are required for the recall policies to converge, which is the point at which there is no improvement in the recall on our test set. The aim of the system is to learn a policy with only a small number of training examples. The results presented below are promising as they show that, in fact, only a small number of observations of a type of event is required to accurately recall that type of event.

The first evaluation is done under three different conditions. Initially, we look at

events that have distractions. A distraction is a correct observation of something that is not relevant to the event. For example, if the agent initially sees a rubbish bin in a room while observing two people having a conversation, then that rubbish bin is a distraction. We present our results by showing how effectively our policies can be trained and used with varying amounts of distractions.

We then look at how well our system performs with missing or incorrect data. Missing data is information that is relevant to the event but is either not included in the event or it is included incorrectly. The most common reason for this is sensor noise, however, it can also be due to the relevant information not being in the robots field of view when the event instance was recorded.

An example of where the information is included but incorrect might be a case where the robot receives noisy LIDAR data and consequently is mis-localised. Thus, if the location were relevant to the event in question, it would be recorded incorrectly. An example of data that are missing might be if the robot should have observed three pieces of fruit but instead only observed two. Thus, the event will have incomplete or missing information.

Similarly, we present the results showing the effects of different levels of missing data.

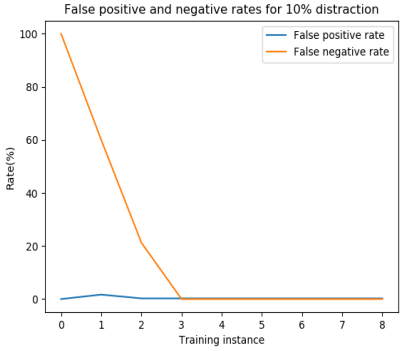
Lastly, we evaluate our system on a range of misclassification noise. Misclassification noise is where the trainer incorrectly classifies an event in the training set. For example, if we were trying to train a policy for event type $\mathcal{A}$ and some of the examples are incorrectly labelled event type $\mathcal{B}$.

In our case, because the goal of the policy for each type of event is to determine whether an observation is an instance of that type of event or not, misclassified data in the training set is given the label, *no_match*. Thus, we incorrectly inform the agent about conditions under which is should reject instances as being members of the event type to which the policy belongs. We show our results for a range of misclassification noise from 1% to 5%.

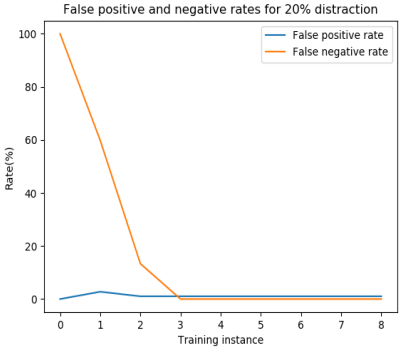
Distraction Results

The first set of results are for "distractions". Distractions are different from

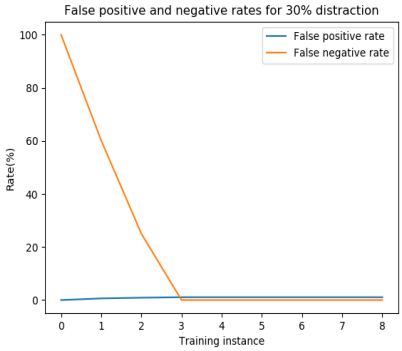
noise in that a distraction is something that is a correct observation but not relevant. The evaluation includes differing amounts of distraction. To calculate the number of distractions in a training instance, we calculate the amount of additional information that training instance has over the generic frame for that type of event. Our results are presented in figures 6.13 to 6.14.



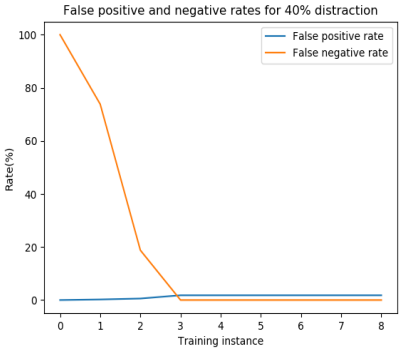
(a) 10%



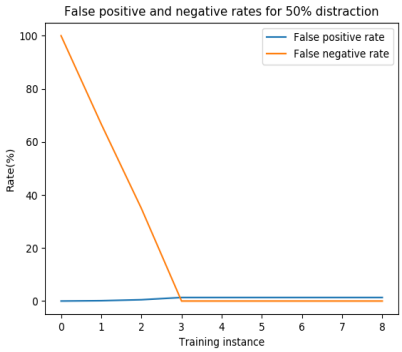
(b) 20%



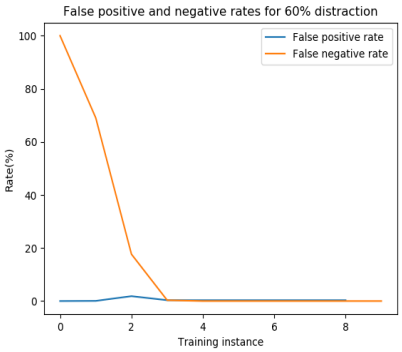
(c) 30%



(d) 40%

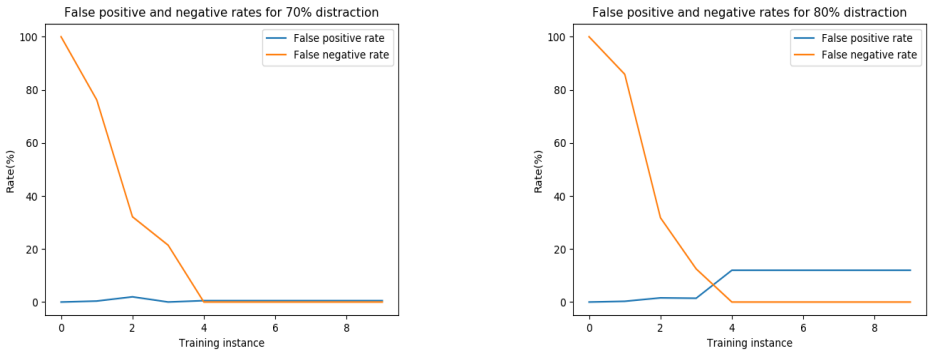


(e) 50%



(f) 60%

Figure 6.13: False positives and false negatives after each training instance for levels of distraction between 10% and 60%



(a) 70%

(b) 80%

Figure 6.14: False positives and false negatives after each training instance for levels of distraction between 70% and 80%

One can see from these results that the number of distractions present in the training examples has little impact on how effectively the event recall policies are trained and used. The false negative rate always starts at 100% and the false positive rate starts at 0%. This is because the default policy is:

if *true* **then** *no_match*

This policy tells the agent that if no other rule fires, it should conclude that the observation is not an instance of the event type to which the policy is assigned. Therefore, with no training, and thus no generalisation or specialisation to the policy, the agent correctly rejects all events that are not instances of the event type to which the policy is assigned but it also incorrectly reject all events that are.

For the majority of cases, except where the distraction rate increases above 80%, the false positive rate increases slightly after the first or second training instance before either decreasing again or plateauing. This can again be expected. As a policy is specialised or generalised, it is likely that instances of some other events types will be captured by the updated policy. We did not intervene in the training process, however, at the discretion of the trainer, it is possible to interrupt training and specialise the recall policy to discard the incorrect instances that were originally captured.

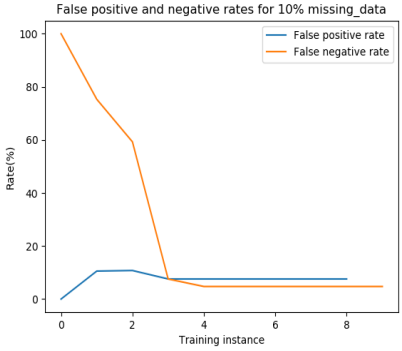
Regardless of event complexity, the recall policies can be trained quickly and can accurately recall events.

Missing Data

Missing data is a problem for any learning system. For the most part, missing or incorrect data are mainly due to failures in the robot's perception, including misclassifying objects, failing to identify objects, incorrectly identifying a person and being mis-localised.

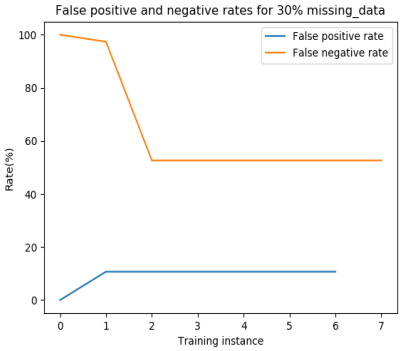
To evaluate the amount of missing or incorrect data in a training instance we use much the same approach as calculating the amount of distraction in a training instance in that we compare it to the known generic frame for the type of event we are training. We assume that all information contained in the generic is relevant. In the case of missing data however, we are less concerned with the amount of additional information and more concerned with the amount of missing information. For example, if an event has three pieces of fruit that are relevant and the robot observed only two, then the amount of missing data is 33%.

We present our results in figure 6.15. We found that with more than 40% missing data, we struggled to obtain anything meaningful. While this is a problem, we also found that in reality it is not common to have more than 40% missing data and for the purpose of training we intentionally added noise into the training instances to try and push our system to the limit. The average amount of missing data is between 5% and 15%. With less than 10%, the system is still able to achieve accurate recall, 4%, after only a small number of observations.



(a) 10%

(b) 20%



(c) 30%

Figure 6.15: False positives and false negatives rafter each training instance for varying levels of missing data

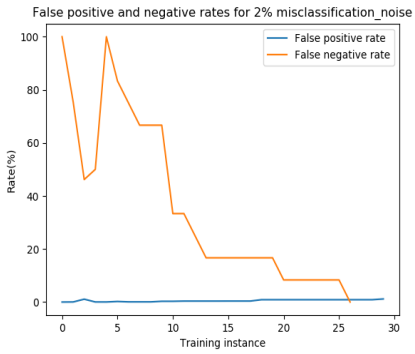
Misclassification Noise

Finally, we present the results for how the system is affected by misclassified training instances. When training a policy for an event type, the training set generally consists of positive only examples that are labelled, *match*.

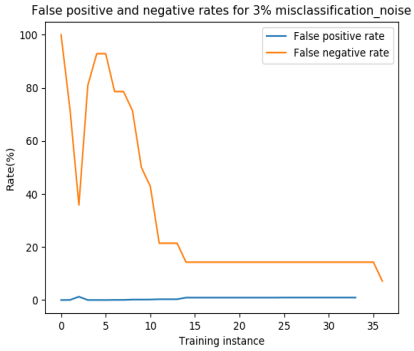
However, sometimes a training instance will be incorrectly labelled as a *no_match*, resulting in misclassification noise. We show the effect of this for varying levels of misclassification noise in figure 6.16.

As was expected, the policy learning still converges, but takes longer depending on the amount of misclassification noise. Note that the false negative rate fluctuates. When a policy generalises enough, it captures only the information

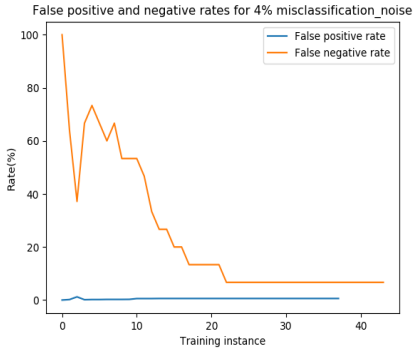
(a) 1%



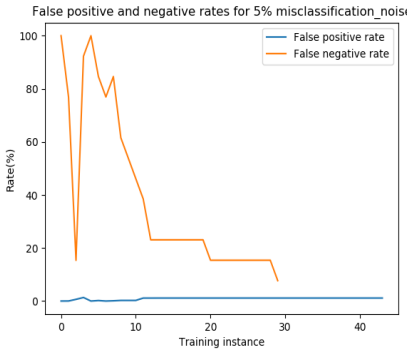
(b) 2%



(c) 3%



(d) 4%



(e) 5%

Figure 6.16: False positives and false negatives after each training instance for varying levels of misclassification noise

that is relevant to an event type. If a misclassified example is presented at a late stage in the training, no further generalisation to the policy is made, with the exception that the label has been inverted. Consider for example that an event's policy was as follows:

$$\textbf{if } match(action) \textbf{ then } match$$

This policy says that if the values of the *action* slots in the generic and the event instance are the same then the instance is a member of this event class. If we show it a training example that has has been labelled as *no_match* however, then an exception will be added to this rule. This is because the rule fired but returned the wrong conclusion, or at least it believed to return the wrong conclusion.

Because no further generalisation can be done to the event, the exception that is added is the following:

$$\textbf{if } match(action) \textbf{ then } match \textbf{ except}$$
$$\textbf{if } match(action) \textbf{ then } no\_match$$

Therefore, a correctly labelled training example is required to correct the mistake.

Policy Recreation

The final set of results that we present show the average number of observations of an event type that are required to recreate the hand-crafted policies that we described in Section 6.2.2.1. The following table, table 6.1, details those results. We present the results in incremental order of complexity. The Event Sequence Length (ESL) refers to the number of individual events in a sequence. We do not show the results for each individual type of event that we collecting but rather the average for all events of the same sequence length.

Table 6.1: Recreating event recall policy results

The number of training examples required to recreate the hand-crafted policies			
ESL	ANSC	AISC	ANTE
1	7	1061	2.03
2	7	545	2.83
3	4	421	4.02
4	2	412	4.665
5	2	412	4.35
6	2	113	6.5
7	1	106	5.27
8	1	85	4.375
9	1	105	5.5

ESL = Event Sequence Length
ANSL = Average Number of Sub-Classes
AISC = Average Instances per Sub-Class
ANTE = Average Number of Training Examples

Because we are training a robot to operate in a domestic setting, our goal is to keep the required number of training examples to a minimum. These results show that our system does perform well with very little training

It is often the case that with large event sequences in this training and testing set, many of the events that we created could have been discarded if we employed some means to detect *critical* actions.

6.2.3.2 The Effect of *Critical* Actions

In collecting the data above we did give the agent any means of differentiating between *critical* and *non-critical* actions (see Section 4.4.3.2). While we only touch briefly on the subject in this dissertation we think that it is an important aspect which merits further research.

Even on a relatively small number of events we can demonstrate how beneficial and important a feature this is. We also show that when agents are endowed with this functionality, they can often discover solutions to problems other than the ones we expect them to learn.

Here, we demonstrate that by replaying the events that we have collected but differentiating between *critical* and *non-critical* actions, we can reduce the size

of large event sequences. The first step is to define some *critical* states. We initially keep this simple and define a set of states that we want the robot to avoid. For example, one of the many states that indicates a *critical* state is:

$$on\_ground(broken\_glass)$$

When a *critical* state is noted, we begin recording an event sequence. When that same *critical* state has been resolved, we finish recording the event sequence. This can reduce the size of large event sequences. For example, two of the event sequences of over 7 events can be reduced to an event sequence of only 3 events. This is because within this event sequence we recorded several intermediary events and at least one event that was created due to a noisy observation. On one occasion the agent believed that a bookshelf that was leaning against the wall was placed on the other side of that wall. Thus, that perceived state change recorded a new event.

We also made an interesting discovery in one of the event sequences where an agent clears up rubbish. When collecting the data, we indicated to the agent that the event terminated when the fallen fruit had been placed in the bin. However, our agent instead learned that by simply picking up the fallen fruit it was no longer on the ground and so the *critical* state had been resolved. In theory, the agent was correct but for obvious reasons it is less than ideal for an agent to carry around indefinitely something that should be in the bin.

One possible solution is to inform the agent that that too is a *critical* state and so in resolving one *critical* state it observes another which in turn triggers a new sequence of events. This is something, however, that we suggest as an extension to this research and do not explore it in any further detail at this stage.

6.2.4 Discussion

Our hypothesis was as follows: By assigning unique recall policies to individual types of events we can successfully capture the contextual information relevant to that type of event. In turn, when recalling instances of this type of event from memory we can correctly recall instances of the type of event that we are observing and discard instances of other types of events, even ones that are

qualitatively similar in nature. We further hypothesise that by using RDRs as recall policies we can train them to recall events correctly with only a minimal number of training examples. This is an important feature of our system as in order for robots to effectively co-exist with people, they must be able to learn quickly and effectively.

Our quantitative results show that with minimal training examples we can recall events from memory with high accuracy. The quantitative results would appear to support the hypothesis. However, there was considerable structure placed on the information contained within an event and it is unclear how effective this approach would be on an unstructured data set. We also only evaluated our results in one particular setting, that of a domestic environment. In this type of environment we believe that our approach is an effective means at learning and distinguishing between context in different types of events. It is also an effective approach at learning and reasoning about which types of behaviours should be applied in a given situation.

However, this method would be significantly less effective at learning how to execute those types of behaviours and we believe that a more traditional reinforcement learning algorithm would be more effective in this case. In other types of environments were Case-Based Reasoning approaches have been applied, such as, learning to play a game of soccer, it may be preferable to engage an approach similar to the one discussed in the SOAR cognitive architecture [69, 120, 70, 7, 9, 8].

To conclude, in an unstructured environment such as a house, our approach is an effective means of learning to distinguish between different types of events and recalling specific types of events on future observations. However, it is not without limitations and has considerable scope for future research. We provide some suggestions for future research in Chapter 7.

Chapter 7

Conclusion

7.0.1 Thesis Summary

This dissertation presents a novel approach to how events are created and recalled from episodic memory in cognitive robots. Episodic memory is largely agreed to be one of two main components of human declarative memory, the other being semantic memory. We do not claim to "replicate" human-like episodic memory. However, taking inspiration from the concepts of episodic memories are and how they are stored, we have formulated some key points that we regard as essential elements of an episodic memory, when integrated with an artificial cognitive agent.

We assume that the information stored in an episodic memory is mostly qualitative. For example, if a person were to be asked where something happened, they would respond with the name of the place, a town, a city or in our case, operating as we do in a domestic environment: a room name. Therefore, a topological map is required so that a robot can locate objects, and itself, within the environment at the qualitative level required for episodic memory. Previous methods for generating topological maps have several limitations that needed to be addressed. In Chapter 3, we presented our approach to topological mapping and demonstrated that we achieve state of the art results in precision and recall, while making none of the assumptions limit the previous work.

For episodic memories to be of use, there must be some means by which we

can recall those memories. Other research into episodic recall lacks a number of crucial elements. The most significant limitation is that they have no means of addressing the recall of events based only on the specific contextual cues that are relevant to the event, often relying on some kind of distance metric to determine the qualitative similarity between two cases or events. By not typing events, the domain of application is limited to environments with a finite number of goals and types of information. Typically, partially observable, unstructured environments do not adhere to these constraints and so a different approach to episodic recall is required.

By distinguishing between different types of events we are able to define specific recall policies that are unique to each type of event. Separating events into different sub-classes has not been commonly adopted elsewhere and the only other mention of it that we are aware of is in EPIROME [68]. However, our motivation for distinguishing between different types of events becomes clear when one thinks of the domain of application that we are investigating, namely a robot operating in partially observable, unstructured environment, such as a home. The second major contribution of our research is an episodic recall mechanism in which unique policies are are acquired for different types of events. Thus, a policy can more effectively capture the information that is relevant to the type of event that it is associated with. This was presented in Chapter 4.

If we are to have individual recall policies for each type of event we need an efficient way to train these policies. Batch style machine learning is not suitable for our purpose, since recall policies must be learned incrementally, using as few training examples as possible. The final contribution that we make is an efficient, hybrid approach to episodic recall policy training based on incremental learning that can also be guided by a human expert. This approach is an extension to how Ripple Down Rules are traditionally trained, where only a human guides the learning. We presented our method in Chapter 5.

Before discussing possible extensions and further work, we summarise the advantages of our approach over previous methods for implementing episodic recall or case retrieval.

7.0.2 Research Comparison

Evaluating systems that employ episodic memory is difficult because there are no standard test scenarios for comparison. However, we present a set of situations where one system might be preferable to another. We assume that agents have no prior understanding of the goals that they will be expected to achieve or the information they are likely to be presented with. It is likely that different types of events will have completely different types of relevant information. Thus, we require a system that can ignore irrelevant information so that it can accurately recall episodes. To the best of our knowledge, our method is the only one that provides this capability.

Apart from SOAR or case based reasoning, one of the earliest mentions of episodic memory for cognitive robots was EPIROME by Jockel *et al* [68]. It appears to be the only other approach, apart from ours, that distinguishes between different types of events. EPIROME incorporates the concept of typed events but it does not address what we believe is the most essential element of an episodic memory system, namely, the retrieval mechanism.

We have already discussed SOAR in the literature review. Depending on the domain of application, it may sometimes be preferable to use SOAR's approach to episodic memory over ours. For example, SOAR uses nearest neighbour retrieval [7, 8] and quantitatively weights an event's significance [137]. This is efficient and much simpler to implement and may be preferable when events are not typed, the goals are finite and known and the input data are finite and known.

Case based reasoning applies similar logic to matching cases and much of the same reasoning applies as, in most systems that use case based reasoning such as Homem *et al* [10], the goals are finite and known in advance. Thus, a system that quantitatively evaluates the similarity between cases is appropriate. However, for the same reasons as already outlined, when differentiating between different types of events, a simple distance metric lacks the capacity to capture the different types of information relevant to the different types of events.

The use of RDRs as recall policies enables our system to create customised recall policies that can be trained incrementally, and which are able to ignore irrelevant information. They can also be trained in a hybrid of human-guided

and self-guided learning.

7.0.3 Extensions and Future Work

When generating topological maps, we have a five-stage pipeline, where the fifth stage uses a semi-autonomous process for labelling regions that are part of the same larger region. This can be improved by employing a fully autonomous process to label regions. The system may have semantic information about the contents of an apartment that could be used to assist in labelling regions. For example, if the vision system recognises a fridge, the region is likely to be the kitchen. Spatial relations between different objects can also be used. In a studio apartment, the separations of different regions or rooms may not be clearly defined. For example, the gap between a couch and a wall may separate the dining room from the living room. In a living room however, a couch will be likely to face towards a television and so all regions in between these can also be assumed as part of the living room. In doing this it could be useful to employ research conducted by Fidler *et al* [142] and Kong *et al* [133]. They use an understanding of spatial relations to improve an agent's scene understanding and help to classify other objects in the environment.

There are also extensions that can be made to our work in episodic recall. A recall policy is a Ripple Down Rule and RDRs are built using contextual, semantic information. Therefore, it should be possible for an agent to explain the reasons why it has recalled one event over another. However, we have not yet implemented an explanation mechanism.

The efficiency of the recall algorithms could be improved significantly. At present, evaluating episodic recall policies is linear in time complexity. While we have, to an extent, addressed the inefficiency of the recall process by employing a two-stage pipeline, there are still further improvements that could be made. For example, compiling policies into a network similar to RETE [138] would evaluate an RDR only when the values referenced in the condition are present. This means that our retrieval algorithm could be much improved in time complexity as we would not need a linear traversal of each event, evaluating the RDRs one at a time.

Forgetting events that are no longer relevant would further improve efficiently

by reducing the size of the database. Nuxoll and Laird [9] compare the most common algorithms used for forgetting episodes and conclude that an activation based method, in which episodes are selected for removal from memory based on frequency and recency of a particular episode's recall has the best performance.

Differentiating between *critical* and *non-critical* actions is also important. It should be possible to autonomously add new *critical* states into the environment. At present we inform the agent, in advance, of the constraints that it is likely to encounter or at least a class of constraint that an agent is likely to encounter. However, this is not flexible and thus not scalable or generalisable. Therefore, we need to either be able to learn new constraints or else assign quantitative values to events based on their perceived significance.

The most significant extension that we propose with regard to training recall policies is how we handle misclassification and perception noise. Currently, the only means to correct mistakes due to noise is for the trainer to intercept and manually guide the learning when a mistake has been made. We have proposed simple solutions to this, namely a voting system to decide which information is and is not relevant however, a more sophisticated approach would be preferable.

To deal with misclassification noise a method similar to Smith and Martinez [143] may be used. They introduce a filtering method called PRISM to identify instances in a data set that may be misclassified. Our approach would have to be altered slightly, as we may need to identify misclassified instances from a very small data set.

To handle perception noise we must address the low-level issues, namely inaccurate vision and localisation. Both of these are outside the focus of this research, however, and so we have, as yet, not explored this in much detail.

Bibliography

[1] E. Tulving *et al.*, "Episodic and semantic memory," *Organization of memory*, vol. 1, pp. 381–403, 1972.

[2] R. Bormann, F. Jorda, W. Li, J. Hampp, and M. Hägele, "Room segmentation: Survey, implementation, and analysis," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 1019–1026.

[3] P. Compton, G. Edwards, B. Kang, L. Lazarus, R. Malor, P. Preston, and A. Srinivasan, "Ripple down rules: Turning knowledge acquisition into knowledge maintenance," *Artificial Intelligence in Medicine*, vol. 4, no. 6, pp. 463 – 475, 1992, representing Knowledge in Medical Decision Support Systems. [Online]. Available: http://www.sciencedirect.com/science/article/pii/093336579290013F

[4] B. R. Gaines and P. Compton, "Induction of ripple-down rules applied to modeling large databases," *Journal of Intelligent Information Systems*, vol. 5, no. 3, pp. 211–228, Nov 1995. [Online]. Available: https://doi.org/10.1007/BF00962234

[5] P. Compton and B.-H. Kang, *RIPPLE-DOWN RULES: THE ALTERNATIVE TO MACHINE LEARNING*, 1st ed. Crc Press Llc, 2021.

[6] B. H. Kang, "Multiple classification ripple down rules : Evaluation and possibilities," 2000.

[7] A. Nuxoll and J. E. Laird, "A cognitive model of episodic memory integrated with a general cognitive architecture," in *ICCM*, 2004.

[8] A. M. Nuxoll and J. E. Laird, "Enhancing intelligent agents with episodic memory Action editor : Vasant Honavar," *Cognitive*

Systems Research, vol. 17-18, pp. 34–48, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.cogsys.2011.10.002

[9] A. Nuxoll, D. Tecuci, W. C. Ho, and N. Wang, "Comparing forgetting algorithms for artificial episodic memory systems," in *Proc. of the Symposium on Human Memory for Artificial Agents. AISB*, 2010, pp. 14–20.

[10] T. P. D. Homem, P. E. Santos, A. H. R. Costa], R. A. [da Costa Bianchi], and R. L. de Mantaras], "Qualitative case-based reasoning and learning," *Artificial Intelligence*, vol. 283, p. 103258, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370218303424

[11] B. Lau, C. Sprunk, and W. Burgard, "Improved updating of euclidean distance maps and voronoi diagrams," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2010, pp. 281–286.

[12] Z. Kasap and N. Magnenat-Thalmann, "Towards episodic memory-based long-term affective interaction with a human-like robot," in *19th International Symposium in Robot and Human Interactive Communication*, Sep. 2010, pp. 452–457.

[13] C. Xiong, S. Merity, and R. Socher, "Dynamic memory networks for visual and textual question answering," 2016.

[14] A. Kumar, O. Irsoy, P. Ondruska, M. Iyyer, J. Bradbury, I. Gulrajani, V. Zhong, R. Paulus, R. Socher, J. Bradbury, and R. Com, "Ask Me Anything: Dynamic Memory Networks for Natural Language Processing," *Proceedings of The 33rd International Conference on Machine Learning, PMLR 48*, vol. 48, pp. 1378–1387, 2016. [Online]. Available: http://proceedings.mlr.press/v48/kumar16.html{%}0Ahttp://proceedings.mlr.press/v48/kumar16.pdf

[15] J. Li, W. Monroe, A. Ritter, and D. Jurafsky, "Deep Reinforcement Learning for Dialogue Generation," *arXiv*, vol. 2, no. 2, pp. 1192–1202, 2016. [Online]. Available: http://arxiv.org/abs/1606.01541

[16] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc.,

2014, pp. 3104–3112. [Online]. Available: http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf

[17] S. R. Eddy, "Hidden Markov models," *Current Opinion in Structural Biology*, vol. 6, no. 3, pp. 361–365, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0959440X9680056X

[18] Crespo, Castillo, M. Oscar, and Barber, "Semantic information for robot navigation: A survey," *Applied Sciences*, vol. 10, p. 497, 01 2020.

[19] S. Thrun, "Integrating Grid-Based and Topological Maps for Mobile Robot Navigation Arno Bucken Grid-Based Maps," *Proceedings of the Thirteenth Nation Conference on Artificial Intelligence Artificial Intelligence*, no. August, 1996.

[20] P. Beeson, N. K. Jong, and B. Kuipers, "Towards autonomous topological place detection using the extended voronoi graph," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, 2005, pp. 4373–4379.

[21] S. Friedman, H. Pasula, and D. Fox, "Voronoi random fields: Extracting topological structure of indoor environments via place labeling." in *IJCAI*, vol. 7, 2007, pp. 2109–2114.

[22] L. Wu, M. A. Garcia, D. Puig, and A. Sole, "Voronoi-based space partitioning for coordinated multi-robot exploration," *Journal of Physical Agents*, vol. 1, no. 1, pp. 37–44, 2007.

[23] R. Ramaithitima, M. Whitzer, S. Bhattacharya, and V. Kumar, "Automated creation of topological maps in unknown environments using a swarm of resource-constrained robots," *IEEE Robotics and Automation Letters*, vol. 1, no. 2, pp. 746–753, July 2016.

[24] Z. Liu, D. Chen, and G. von Wichert, "2d semantic mapping on occupancy grids," in *ROBOTIK 2012; 7th German Conference on Robotics*, May 2012, pp. 1–6.

[25] C. Mura, O. Mattausch, A. J. Villanueva, E. Gobbetti, and R. Pajarola, "Automatic room detection and reconstruction in cluttered indoor environments with complex room layouts," *Computers & Graphics*, vol. 44, pp. 20 – 32, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0097849314000661

[26] S. Ochmann, R. Vock, R. Wessel, and R. Klein, "Automatic reconstruction of parametric building models from indoor point clouds," *Computers & Graphics*, vol. 54, pp. 94 – 103, 2016, special Issue on CAD/Graphics 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0097849315001119

[27] R. Ambruş, S. Claici, and A. Wendt, "Automatic room segmentation from unstructured 3-d data of indoor environments," *IEEE Robotics and Automation Letters*, vol. 2, no. 2, pp. 749–756, April 2017.

[28] M. Mielle, M. Magnusson, and A. J. Lilienthal, "A method to segment maps from different modalities using free space layout maoris: Map of ripples segmentation," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4993–4999, 2018.

[29] P. Buschka and A. Saffiotti, "A virtual sensor for room detection," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 1, no. October, pp. 637–642, 2002.

[30] C. Galindo, A. Saffiotti, S. Coradeschi, P. Buschka, J. A. Fernandez-Madrigal, and J. Gonzalez, "Multi-hierarchical semantic maps for mobile robotics," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Aug 2005, pp. 2278–2283.

[31] F. Dellaert and D. Bruemmer, "Semantic SLAM for Collaborative Cognitive Workspaces," *AAAI Fall Symposium Series 2004: Workshop on The Interaction of Cognitive Science and Robotics: From Interfaces to Intelligence*, 2004. [Online]. Available: http://frank.dellaert.com/pub/Dellaert04ss.pdf

[32] B. Limketkai, L. Liao, and D. Fox, "Relational object maps for mobile robots," *IJCAI International Joint Conference on Artificial Intelligence*, pp. 1471–1476, 2005.

[33] A. Rottmann, O. Mozos, C. Stachniss, and W. Burgard, "Semantic place classification of indoor environments with mobile robots using boosting." *Proceedings of the National Conference on Artificial Intelligence*, vol. 3, pp. 1306–1311, 01 2005.

[34] Óscar Martínez Mozos, R. Triebel, P. Jensfelt, A. Rottmann, and W. Burgard, "Supervised semantic labeling of places using information

extracted from sensor data," *Robotics and Autonomous Systems*, vol. 55, no. 5, pp. 391 – 402, 2007, from Sensors to Human Spatial Concepts. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S092188900600203X

[35] C. Nieto-Granda, J. G. Rogers, A. J. B. Trevor, and H. I. Christensen, "Semantic map partitioning in indoor environments using regional analysis," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2010, pp. 1451–1456.

[36] N. Sünderhauf, F. Dayoub, S. McMahon, B. Talbot, R. Schulz, P. Corke, G. Wyeth, B. Upcroft, and M. Milford, "Place categorization and semantic mapping on a mobile robot," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 5729–5736.

[37] M. Brucker, M. Durner, R. Ambruş, Z. C. Márton, A. Wendt, P. Jensfelt, K. O. Arras, and R. Triebel, "Semantic labeling of indoor environments from 3d rgb maps," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018, pp. 1871–1878.

[38] Y. Gil, "Learning by experimentation: Incremental refinement of incomplete planning domains," in *Machine Learning Proceedings 1994*, W. W. Cohen and H. Hirsh, Eds. San Francisco (CA): Morgan Kaufmann, 1994, pp. 87–95. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558603356500192

[39] M. Sridharan and B. L. Meadows, "Knowledge representation and interactive learning of domain knowledge for human-robot interaction," 2018.

[40] J. C. Herrero, R. I. B. Castaño, and O. M. Mozos, "An inferring semantic system based on relational models for mobile robotics," in *2015 IEEE International Conference on Autonomous Robot Systems and Competitions*, April 2015, pp. 83–88.

[41] A. K. Bozcuoglu, Y. Furuta, K. Okada, M. Beetz, and M. Inaba, "Continuous modeling of affordances in a symbolic knowledge base," in *IEEE/RSJ International Conference on Intelligent Robots and Systems(IROS)*, Macau, China, 2019, accepted for publication.

[42] J. Mason and B. Marthi, "An object-based semantic world model for long-term change detection and semantic querying," *Proceedings of the ...*

IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 3851–3858, 10 2012.

[43] J. Elfring, S. van den Dries, M. van de Molengraft, and M. Steinbuch, "Semantic world modeling using probabilistic multiple hypothesis anchoring," *Robotics and Autonomous Systems*, vol. 61, no. 2, pp. 95 – 105, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0921889012002163

[44] J. Elfring, M. Molengraft, and M. Steinbuch, "Semi-task-dependent and uncertainty-driven world model maintenance," *Autonomous Robots*, vol. 38, pp. 1–15, 01 2014.

[45] J. Hou, Y. Yuan, and S. Schwertfeger, "Area graph: Generation of topological maps using the voronoi diagram," 10 2019.

[46] A. Thippur, C. Burbridge, L. Kunze, M. Alberti, J. Folkesson, P. Jensfelt, and N. Hawes, "A Comparison of Qualitative and Metric Spatial Relation Models for Scene Understanding," *Aaai*, no. Section 4, pp. 1632–1640, 2015. [Online]. Available: https://pdfs.semanticscholar.org/af7b/83dca60af4ab7d96a2fb7bb0d6f757493f5e.pdf

[47] L. Kunze, C. Burbridge, M. Alberti, A. Thippur, J. Folkesson, P. Jensfelt, and N. Hawes, "Combining top-down spatial reasoning and bottom-up object class recognition for scene understanding," *IEEE International Conference on Intelligent Robots and Systems*, no. Section III, pp. 2910–2915, 2014.

[48] R. Hu, H. Xu, M. Rohrbach, J. Feng, K. Saenko, and T. Darrell, "Natural language object retrieval," *CoRR*, vol. abs/1511.04164, 2015. [Online]. Available: http://arxiv.org/abs/1511.04164

[49] J. Young and N. Hawes, "Learning by Observation Using Qualitative Spatial Relations," *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015)*, 2015.

[50] G. Gemignani, R. Capobianco, and D. Nardi, *Approaching Qualitative Spatial Reasoning About Distances and Directions in Robotics*. Cham: Springer International Publishing, 2015, pp. 452–464. [Online]. Available: https://doi.org/10.1007/978-3-319-24309-2{_}34

[51] D. Wolter and A. Kirsch, "Leveraging qualitative reasoning to learning manipulation tasks," *Robotics*, vol. 4, pp. 253–283, 2015.

[52] J. F. Allen, "Maintaining knowledge about temproal intervals," *Readings in qualitative reasoning about physical systems*, vol. 26, no. 11, pp. 361–372, 1990.

[53] J. O. Wallgrün, "Qualitative spatial reasoning for topological map learning," *Spatial Cognition & Computation*, vol. 10, no. 4, pp. 207–246, 2010. [Online]. Available: https://doi.org/10.1080/13875860903540906

[54] A. G. Cohn and J. Renz, "Chapter 13 qualitative spatial representation and reasoning," in *Handbook of Knowledge Representation*, ser. Foundations of Artificial Intelligence, F. van Harmelen, V. Lifschitz, and B. Porter, Eds. Elsevier, 2008, vol. 3, pp. 551 – 596. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574652607030131

[55] B. Krieg-Brückner and H. Shi, "Orientation calculi and route graphs: Towards semantic representations for route descriptions," in *Geographic Information Science*, M. Raubal, H. J. Miller, A. U. Frank, and M. F. Goodchild, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 234–250.

[56] C. Matuszek, D. Fox, and K. Koscher, "Following directions using statistical machine translation," in *2010 5th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, March 2010, pp. 251–258.

[57] H. A. Kautz and P. B. Ladkin, "Integrating Metric and Qualitative Temporal Reasoning," *Ninth National Conference on Artificial Intelligence. AAAI'91*, pp. 241–246, 1991.

[58] T. Allen, J. Delgrande, and A. Gupta, "Point-based approaches to qualitative temporal reasoning," *Proceedings of the National Conference on Artificial Intelligence*, pp. 305–316, 11 2006.

[59] A. Gerevini and L. Schubert, "Efficient algorithms for qualitative reasoning about time," *Artificial Intelligence*, vol. 74, no. 2, pp. 207–248, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/000437029400016T

[60] E. Tulving, *Elements of Episodic Memory*. Oxford University Press, 1983.

[61] M. E. Wheeler and E. J. Ploran, "Episodic Memory," in *Encyclopedia of Neuroscience*, L. R. Squire, Ed. Oxford: Academic Press, 2009, pp. 1167–1172. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780080450469007609

[62] E. Tulving and H. J. Markowitsch, "Episodic and declarative memory: Role of the hippocampus," *Hippocampus*, vol. 8, no. 3, pp. 198–204, 1998. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291098-1063%281998%298%3A3%3C198%3A%3AAID-HIPO2%3E3.0.CO%3B2-G

[63] D. Griffiths, A. Dickinson, and N. Clayton, "Episodic memory: what can animals remember about their past?" *Trends in Cognitive Sciences*, vol. 3, no. 2, pp. 74–80, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1364661398012728

[64] A. Baddeley, "The episodic buffer: a new component of working memory?" *Trends in Cognitive Sciences*, vol. 4, no. 11, pp. 417 – 423, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1364661300015382

[65] D. Lopez-Paz and M. A. Ranzato, "Gradient episodic memory for continual learning," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 6467–6476. [Online]. Available: http://papers.nips.cc/paper/7225-gradient-episodic-memory-for-continual-learning.pdf

[66] M. Botvinick, S. Ritter, J. X. Wang, Z. Kurth-Nelson, C. Blundell, and D. Hassabis, "Reinforcement learning, fast and slow," *Trends in Cognitive Sciences*, 04 2019.

[67] Z. Lin, T. Zhao, G. Yang, and L. Zhang, "Episodic memory deep q-networks," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 7 2018, pp. 2433–2439. [Online]. Available: https://doi.org/10.24963/ijcai.2018/337

[68] S. Jockel, D. Westhoff, and Jianwei Zhang, "Epirome - a novel framework to investigate high-level episodic robot memory," in *2007 IEEE Interna-*

tional Conference on Robotics and Biomimetics (ROBIO), Dec 2007, pp. 1075–1080.

[69] J. E. Laird, A. Newell, and P. S. Rosenbloom, "SOAR: An architecture for general intelligence," *Artificial Intelligence*, vol. 33, no. 1, pp. 1–64, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0004370287900506

[70] N. Derbinsky and J. E. Laird, "Efficiently implementing episodic memory," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5650 LNAI, pp. 403–417, 2009.

[71] S. Wallace, E. Dickinson, and A. Nuxoll, "Hashing for lightweight episodic recall," *AAAI Spring Symposium - Technical Report*, pp. 56–61, 01 2013.

[72] E. Vanderwerf, R. Stiles, A. Warlen, A. Seibert, K. Bastien, A. Meyer, A. Nuxoll, and S. A. Wallace, "Hash functions for episodic recognition and retrieval," in *FLAIRS Conference*, 2016.

[73] D. Tecuci and B. Porter, "A generic memory module for events." *Proceedings of the Twentieth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2007*, pp. 152–157, 01 2007.

[74] M. Minsky, "Minsky's frame system theory," in *Proceedings of the 1975 Workshop on Theoretical Issues in Natural Language Processing*, ser. TINLAP '75. Stroudsburg, PA, USA: Association for Computational Linguistics, 1975, pp. 104–116. [Online]. Available: https://doi.org/10.3115/980190.980222

[75] D. Stachowicz and G.-J. Kruijff, "Episodic-like memory for cognitive robots," *Autonomous Mental Development, IEEE Transactions on*, vol. 4, pp. 1–16, 03 2012.

[76] N. Clayton, T. Bussey, and A. Dickinson, "Can animals recall the past and plan for the future?" *Nature reviews. Neuroscience*, vol. 4, pp. 685–91, 09 2003.

[77] N. S. Clayton and J. Russell, "Looking for episodic memory in animals and young children: Prospects for a new minimalism," *Neuropsychologia*, vol. 47, no. 11, pp. 2330 – 2340, 2009, episodic Memory and the

Brain. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0028393208004132

[78] D. Liu, M. Cong, and Y. Du, "Episodic Memory-Based Robotic Planning under Uncertainty," *IEEE Transactions on Industrial Electronics*, vol. 64, no. 2, pp. 1762–1772, 2017.

[79] D. Liu, M. Cong, Y. Du, Q. Zou, and Y. Cui, "Robotic autonomous behavior selection using episodic memory and attention system," *Industrial Robot: An International Journal*, vol. 44, no. 3, pp. 353–362, 2017. [Online]. Available: http://www.emeraldinsight.com/doi/10.1108/IR-09-2016-0250

[80] M. Y. Lim, R. Aylett, P. A. Vargas, W. C. Ho, and J. a. Dias, "Human-like memory retrieval mechanisms for social companions," in *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 3*, ser. AAMAS '11. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2011, pp. 1117–1118. [Online]. Available: http://dl.acm.org/citation.cfm?id=2034396.2034446

[81] F. Shen, Q. Ouyang, W. Kasai, and O. Hasegawa, "A general associative memory based on self-organizing incremental neural network," *Neurocomputing*, vol. 104, pp. 57 – 71, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S092523121200834X

[82] P.-H. Chang and A.-H. Tan, "Encoding and recall of spatio-temporal episodic memory in real time," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI'17. AAAI Press, 2017, pp. 1490–1496. [Online]. Available: http://dl.acm.org/citation.cfm?id=3172077.3172094

[83] J. Berlin and A. Motro, *Database Schema Matching Using Machine Learning with Feature Selection*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 315–329. [Online]. Available: https://doi.org/10.1007/978-3-642-36926-1_25

[84] M. Leordeanu, R. Sukthankar, and M. Hebert, "Unsupervised learning for graph matching," *International Journal of Computer Vision*, vol. 96, no. 1, pp. 28–45, Jan 2012. [Online]. Available: https://doi.org/10.1007/s11263-011-0442-2

[85] T. S. Caetano, J. J. McAuley, L. Cheng, Q. V. Le, and A. J. Smola, "Learning graph matching," *CoRR*, vol. abs/0806.2890, 2008. [Online]. Available: http://arxiv.org/abs/0806.2890

[86] D. Vernon, M. Beetz, and G. Sandini, "Prospection in cognition: The case for joint episodic-procedural memory in cognitive robotics," *Frontiers in Robotics and AI*, vol. 2, p. 19, 2015. [Online]. Available: https://www.frontiersin.org/article/10.3389/frobt.2015.00019

[87] J. Kolodner, *Case-based reasoning*. Morgan Kaufmann, 2014.

[88] M. Sharma and C. Sharma, "A review on diverse applications of case-based reasoning," in *Advances in Computing and Intelligent Systems*, H. Sharma, K. Govindan, R. C. Poonia, S. Kumar, and W. M. El-Medany, Eds. Singapore: Springer Singapore, 2020, pp. 511–517.

[89] R. C. Schank and R. P. Abelson, *Scripts, Plans, Goals and Understanding: an Inquiry into Human Knowledge Structures*. Hillsdale, NJ: L. Erlbaum, 1977.

[90] I. Watson and F. Marir, "Case-based reasoning: A review," *The knowledge engineering review*, vol. 9, no. 4, pp. 327–354, 1994.

[91] J. Kendall-Morwick and D. Leake, *A Study of Two-Phase Retrieval for Process-Oriented Case-Based Reasoning*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 7–27. [Online]. Available: https://doi.org/10.1007/978-3-642-38736-4_2

[92] M. Veloso and A. Aamodt, Eds., *Case-Based Reasoning Research and Development: Proceedings of the First International Conference on Case-Based Reasoning*. Berlin: Springer Verlag, 1995.

[93] C. Riesbeck and R. Schank, *Inside Case-based Reasoning*. Northvale, NJ: Erlbaum, 1989.

[94] R. Moratz and J. Wallgrün, "Spatial reasoning with augmented points: Extending cardinal directions with local distances," *Journal of Spatial Information Science*, vol. 5, no. 2012, pp. 1–30, 2012, cited By 21. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84889008275&doi=10.5311%2fJOSIS.2012.5.84&partnerID=40&md5=6f3f3bde69a7fb0a59358d57da03ff8d

[95] B. Smyth and M. T. Keane, "Retrieving adaptable cases," in *Topics in Case-Based Reasoning*, S. Wess, K.-D. Althoff, and M. M. Richter, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 209–220.

[96] M. L. Maher and P. Pu, *Issues and applications of case-based reasoning to design*. Psychology Press, 2014.

[97] A. HOLT, I. BICHINDARITZ, R. SCHMIDT, and P. PERNER, "Medical applications in case-based reasoning," *The Knowledge Engineering Review*, vol. 20, no. 3, p. 289–292, 2005.

[98] F. Amato, A. López, E. M. Peña-Méndez, P. Vaňhara, A. Hampl, and J. Havel, "Artificial neural networks in medical diagnosis," 2013.

[99] Q. K. Al-Shayea, "Artificial neural networks in medical diagnosis," *International Journal of Computer Science Issues*, vol. 8, no. 2, pp. 150–154, 2011.

[100] A. Carlson, J. Betteridge, and B. Kisiel, "Toward an Architecture for Never-Ending Language Learning." *In Proceedings of the Conference on Artificial Intelligence (AAAI) (2010)*, pp. 1306–1313, 2010. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/download/1879/2201

[101] T. Mitchell, W. Cohen, E. Hruschka, P. Talukdar, B. Yang, J. Betteridge, A. Carlson, B. Dalvi, M. Gardner, B. Kisiel, and et al., "Never-ending learning," *Commun. ACM*, vol. 61, no. 5, p. 103–115, Apr. 2018. [Online]. Available: https://doi.org/10.1145/3191513

[102] T. Mitchell and E. Fredkin, "Never-ending language learning," in *2014 IEEE International Conference on Big Data (Big Data)*, Oct 2014, pp. 1–1.

[103] N. Hawes, C. Burbridge, F. Jovan, L. Kunze, B. Lacerda, L. Mudrova, J. Young, J. Wyatt, D. Hebesberger, T. Kortner, R. Ambrus, N. Bore, J. Folkesson, P. Jensfelt, L. Beyer, A. Hermans, B. Leibe, A. Aldoma, T. Faulhammer, M. Zillich, M. Vincze, E. Chinellato, M. Al-Omari, P. Duckworth, Y. Gatsoulis, D. Hogg, A. Cohn, C. Dondrup, J. Pulido Fentanes, T. Krajnik, J. M. Santos, T. Duckett, and M. Hanheide, "The STRANDS Project: Long-Term Autonomy in Everyday Environments," *IEEE Robotics and Automation Magazine*, 2017.

[104] W. Meeussen, E. Marder-Eppstein, K. Watts, and B. P. Gerkey, "Long term autonomy in office environments," in *ICRA 2011 Workshop on Long-term Autonomy*, IEEE. Shanghai, China: IEEE, 05/2011 2011.

[105] T. T. Tran, T. Vaquero, G. Nejat, and J. C. Beck, "Robots in retirement homes: Applying off-the-shelf planning and scheduling to a team of assistive robots," *J. Artif. Int. Res.*, vol. 58, no. 1, p. 523–590, Jan. 2017.

[106] M. N. Nicolescu and M. J. Mataric, "Natural methods for robot task learning: Instructive demonstrations, generalization and practice," in *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, ser. AAMAS '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 241–248. [Online]. Available: https://doi.org/10.1145/860575.860614

[107] S. Calinon, F. Guenter, and A. Billard, "On learning, representing, and generalizing a task in a humanoid robot," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 37, no. 2, pp. 286–298, 2007.

[108] S. Rosenthal, J. Biswas, and M. Veloso, "An effective personal mobile robot agent through symbiotic human-robot interaction," *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, pp. 915–922, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1838329

[109] S. Rosenthal and M. Veloso, "Mobile Robot Planning to Seek Help with Spatially-Situated Tasks." *Aaai*, vol. 15213, pp. 2067–2073, 2012. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/viewPDFInterstitial/5141/5373

[110] S. Rosenthal, J. Biswas, and M. Veloso, "An effective personal mobile robot agent through symbiotic human-robot interaction," *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, pp. 915–922, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1838329

[111] L. Kunze, N. Hawes, T. Duckett, M. Hanheide, and T. Krajník, "Artificial intelligence for long-term robot autonomy: A survey," *CoRR*, vol. abs/1807.05196, 2018. [Online]. Available: http://arxiv.org/abs/1807.05196

[112] N. Blaylock and J. Allen, "Hierarchical instantiated goal recognition," in *Proceedings of the AAAI Workshop on Modeling Others from Observations*, 2006, pp. 8–15.

[113] P. Singla and R. J. Mooney, "Abductive markov logic for plan recognition," in *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[114] B. Meadows, P. Langley, and M. Emery, "Seeing beyond shadows: Incremental abductive reasoning for plan understanding," *AAAI Workshop - Technical Report*, pp. 24–31, 01 2013.

[115] H. A. Kautz and J. F. Allen, "Generalized plan recognition." in *AAAI*, vol. 86, no. 3237, 1986, p. 5.

[116] C. Flanagan, D. Rajaratnam, and C. Sammut, "Topological Mapping for Cognitive Robots," *Proceedings of the Australasian Conference on Robotics and Automation*, 2020.

[117] V. Govindarajan, S. Bhattacharya, and V. Kumar, "Human-robot collaborative topological exploration for search and rescue applications," in *Distributed Autonomous Robotic Systems*, N.-Y. Chong and Y.-J. Cho, Eds. Tokyo: Springer Japan, 2016, pp. 17–32.

[118] E. Garcia-Fidalgo and A. Ortiz, "Vision-based topological mapping and localization methods: A survey," *Robotics and Autonomous Systems*, vol. 64, pp. 1 – 20, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0921889014002619

[119] J. O. Wallgrün, "Qualitative spatial reasoning for topological map learning," *Spatial Cognition & Computation*, vol. 10, no. 4, pp. 207–246, 2010. [Online]. Available: https://doi.org/10.1080/13875860903540906

[120] J. Laird, K. Kinkade, S. Mohan, and J. Xu, "Cognitive Robotics Using the Soar Cognitive Architecture," *8th International Workshop on Cognitive Robotics*, pp. 46–54, 2012. [Online]. Available: http://www.aaai.org/ocs/index.php/WS/AAAIW12/paper/download/5221/5573

[121] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part i," *IEEE Robotics Automation Magazine*, vol. 13, no. 2, pp. 99–110, June 2006.

[122] H. Choset and K. Nagatani, "Topological simultaneous localization and mapping (slam): toward exact localization without explicit localization," *IEEE Transactions on Robotics and Automation*, vol. 17, no. 2, pp. 125–137, April 2001.

[123] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "FastSLAM: A factored solution to the simultaneous localization and mapping problem," *Proceeding Eighteenth national conference on Artificial intelligence*, vol. 68, no. 2, pp. 593–598, 2002.

[124] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-time loop closure in 2d lidar slam," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 1271–1278.

[125] T. Yamamoto, K. Terada, A. Ochiai, F. Saito, Y. Asahara, and K. Murase, "Development of human support robot as the research platform of a domestic mobile manipulator," *ROBOMECH Journal*, vol. 6, no. 1, p. 4, 2019. [Online]. Available: https://doi.org/10.1186/s40648-019-0132-3

[126] H. Choset, S. Walker, K. Eiamsa-Ard, and J. Burdick, "Sensor-based exploration: Incremental construction of the hierarchical generalized voronoi graph," *The International Journal of Robotics Research*, vol. 19, no. 2, pp. 126 – 148, February 2000.

[127] H. Choset and J. Burdick, "Sensor based motion planning: The hierarchical generalized voronoi graph," in *Workshop on Algorithmic Foundations of Robotics*, January 1996.

[128] N. Kalra, D. Ferguson, and A. T. Stentz, "Incremental reconstruction of generalized voronoi diagrams on grids," in *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS)*, March 2006.

[129] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2015.

[130] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," 2018.

[131] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once : Unified , Real-Time Object Detection."

[132] "Cgal computational geometry library," https://www.cgal.org/.

[133] C. Kong, D. Lin, M. Bansal, R. Urtasun, and S. Fidler, "What are you talking about? text-to-image coreference," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, June 2014, pp. 3558–3565.

[134] C. Flanagan and C. Sammut, "Adaptive Event Retrieval for Episodic Memory," *Advances in Cognitive Systems*, 2020.

[135] M. Ghallab, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. Smith, Y. Sun, and D. Weld, "Pddl - the planning domain definition language," 08 1998.

[136] M. McGill, C. Sammut, and J. Westendorp, "FrameScript: A Multimodal Scripting Language," 2008. [Online]. Available: http://www.cse.unsw.edu.au/{~}claude/research/papers/framescript.pdf{%}5Cnpapers3://publication/uuid/114CFE5B-0EC8-4347-B7CC-514AFB8B1FA0

[137] S. Nason and J. E. Laird, "Soar-rl: integrating reinforcement learning with soar," *Cognitive Systems Research*, vol. 6, no. 1, pp. 51 – 59, 2005, special Issue of Cognitive Systems Research - The Best Papers from ICCM2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389041704000646

[138] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, vol. 19, no. 1, pp. 17 – 37, 1982. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0004370282900200

[139] M. Law, A. Russo, and K. Broda, "The ilasp system for inductive learning of answer set programs," 05 2020.

[140] L. Fermín-Leon, J. Neira, and J. A. Castellanos, "Incremental contour-based topological segmentation for robot exploration," *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2554–2561, 2017.

[141] "Robotis manual for the turtlebot3," https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.

[142] S. Fidler, A. Sharma, and R. Urtasun, "A sentence is worth a thousand pixels," pp. 1995–2002, 06 2013.

[143] M. R. Smith and T. Martinez, "Improving classification accuracy by identifying and removing instances that should be misclassified," in *The 2011 International Joint Conference on Neural Networks*, 2011, pp. 2690–2697.

.1 Topological Map Results Extended

Map ID	Precision	Recall	Bormann
1	98	97	no
2	99	94	no
3	94	96	no
4	95	94	no
5	96	95	no
6	90	90	no
7	98	97	no
8	99	96	no
9	100	97	yes
10	97	98	yes
11	97	92	yes
12	98	95	yes
13	99	95	yes
14	92	96	yes
15	97	97	yes
16	98	94	yes
17	99	94	yes
18	100	98	yes
19	100	98	yes
20	95	95	yes
21	96	95	yes
22	99	97	yes
23	99	96	yes
24	98	96	yes
25	97	92	yes
26	96	93	yes
27	99	93	yes
28	99	97	yes