Louisiana State University
LSU Digital Commons

LSU Doctoral Dissertations | Graduate School

10-22-2018

Reinforcement Learning in Robotic Task Domains with Deictic Descriptor Representation

Harry Paul Moore
Louisiana State University and Agricultural and Mechanical College, hmoor14@lsu.edu

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations

Part of the Robotics Commons

Recommended Citation

Moore, Harry Paul, "Reinforcement Learning in Robotic Task Domains with Deictic Descriptor Representation" (2018). *LSU Doctoral Dissertations*. 4738.
https://digitalcommons.lsu.edu/gradschool_dissertations/4738

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contactgradetd@lsu.edu.

REINFORCEMENT LEARNING IN ROBOTIC TASK DOMAINS WITH DEICTIC DESCRIPTOR REPRESENTATION

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

Division of Computer Science and Engineering
The School of Electrical Engineering and Computer Science

by
Harry Paul Moore
B.S., Louisiana State University, 1974
M.S., Louisiana State University, 1975
December 2018

To my darling grandchildren Ruby, Charlee, and Cohen

Table of Contents

Abstract v

Chapter 1. Introduction 1
1.1. Thesis Statement 1
1.2. Thesis Outline 13

Chapter 2. Background 14
2.1. Reinforcement Learning and Markov Decision Processes 14
2.2. Hierarchical Task Decomposition and Semi-Markov Decision Processes 16
2.3. Partially Observable Markov Decision Process 18
2.4. Transfer Learning 19
2.5. Environment Domains and Domain Tasks 20
2.6. Representing the Environment 23
2.7. Physical and Mental Environments 30
2.8. Optimal and Satisficing Behavior 31

Chapter 3. Related Works 32

Chapter 4. Learning Deictic Descriptor Task Policies 40
4.1. Introduction 40
4.2. The Basic Deictic Descriptor Task Policy Learning Algorithm 43

Chapter 5. Testing the Deictic Policy Learning Algorithm 52
5.1. Metrics and Assumptions for Task Learning 52
5.2. Name Tags World: Alphabetize Name Tags 54
5.3. Blocks World: Stack Block A on Block B 65
5.4. Warehouse World: Pick Up the Green Box 77
5.5. Whiteboard World: Add Binary Numbers 85
5.6. Grid World: Learn a Universal Policy for Room Navigation 94
5.7. Warehouse World: Move a Size-Ordered Stack of Boxes 104
5.8. Manufacturing World: Inspect Box Contents 111

Chapter 6. Improving the Basic Learning Algorithm 118
6.1. Basic Algorithm Deficiencies 118
6.2. Robot Experience Retention 118
6.3. Basic Algorithm vs Experience Retention Memory Algorithm 119
6.4. Reversing Actions 120

Chapter 7. Conclusions and Future Work 124
7.1. Summary 124
7.2. Conclusions 125
7.3. Future Work 126

References 129

Vita 138

Abstract

In the field of reinforcement learning, robot task learning in a specific environment with a Markov decision process backdrop has seen much success. But, extending these results to learning a task for an environment domain has not been as fruitful, even for advanced methodologies such as relational reinforcement learning. In our research into robot learning in environment domains, we utilize a form of deictic representation for the robot's description of the task environment. However, the non-Markovian nature of the deictic representation leads to perceptual aliasing and conflicting actions, invalidating standard reinforcement learning algorithms. To circumvent this difficulty, several past research studies have modified and extended the Q-learning algorithm to the deictic representation case with mixed results. Taking a different tact, we introduce a learning algorithm which searches deictic policy space directly, abandoning the indirect value based methods. We apply the policy learning algorithm to several different tasks in environment domains. The results compare favorably with value based learners and existing literature results.

Chapter 1. Introduction

1.1. Thesis Statement

In this thesis, we claim that a robot utilizing a small number of training episodes and employing a deictic representation in conjunction with a Markov state representation, can learn a deictic satisficing task policy applicable to an environment domain. This long-winded statement may be interpreted in a number of ways. To make it more concrete, in this introductory chapter we present an example and relate it to our claim. The following subsections cover the questions: What is a robot? What is meant by an environment and an environment domain? What is a state representation and a deictic representation? What is a deictic satisficing task policy? How can a robot learn? What are training episodes? Hopefully, answering these questions helps the reader better understand the boundaries and meaning of our claim. Note that the example is intended for the uninitiated in machine learning. Also, we cover these topics in more detail in later chapters. So, we urge the experienced reader to skip or skim the following sub-sections.

1.1.1. The Robot

The word "robot" has a colorful beginning in Karel Čapek's play R.U.R. (1920). In the play, robots are manufactured humans forced to work in servitude, eventually revolting and destroying humanity. Our robots are not so sensational.

In the non-fiction world, robots come in many flavors and varieties. Although the word "robot" is used for online software agents such as chatbots and robot financial advisors, we limit the meaning to an embodied agent, restricted in movement by the laws of physics. So for

our meaning, robots are mechanical agents operating in the physical world, viewing their surroundings with sensors and applying forces through effectors.

Sophisticated robots, capable of performing tasks requiring mobility and object manipulation, are expensive. For example, the iCub robot, used for research such as ours, costs about $250,000. Unfortunately, this price is beyond reach, so we experiment with simulated robots. In the following paragraphs, we describe a simulated research robot in more detail.

The simulated robot has two arms with hands capable of picking up and putting down objects. The hands have enough precision for writing text on a whiteboard. The independent motorized propulsion of the robot's wheels allow for a zero degree turning radius. The robot has batteries for powering the motors and actuators and executing actions is the major component of battery depletion. The detailed physics of the robot's movements are not simulated but this could be added if needed for the problem.

For all of our research and experiments, the robot utilizes high level actions. The control of the robot at lower action levels is assumed to be flawless, making all high level actions deterministic. Examples of high level actions are moving forward one unit and turning in-place either left or right by 45 degrees.

High level actions are built from lower level actions such as the primitive actions of the robot's motors and actuators. For example, the high level action of picking up an object uses lower level actions to make the robot's arm move the hand toward the object, open the fingers, and grasp the object.

Lower level actions, in many cases, use sensor feedback to compensate for the inaccuracy of the primitive actions. The sensors used for feedback may be different from the ones used for

the task being learned. An example is accurately controlling the robot wheels. To accomplish this, a shaft mounted encoder disc with evenly spaced markings are detected with an led/photocell arrangement to form a feedback signal measuring actual angular position of the motor shaft. Non-primitive actions are similar to tasks and in fact, a learned task may later be an action in another task. In this way, the robot has a hierarchy of actions using a variety of feedback sensors, all constructed from lower level actions in the hierarchy with primitive actions at the bottom.

For sensors, the robot has vision, audio, touch, ultrasonic, and RFID (radio frequency identification) capabilities. The touch sensors are for collision detection during motion. The RFID sensors have a multitude of uses including location detection and object type identification. For converting raw vision and audio sensor inputs to something more meaningful, the robot is equipped with a supervisory machine learning visual classification system and a speech recognition system. The ultrasonic sensors are used in conjunction with the visual classification system to determine object distance.

1.1.2. An Example Environment

The physical surroundings of the robot are termed the robot's environment. In many cases, the robot's physical body is also considered part of the environment. Later in this document, we discuss the robot's mental environment. All environments we consider are discrete time environments (as opposed to continuous time environments). A common simple environment is a grid world environment consisting of rooms or mazes located on a grid of tiles. Our example is a two-room environment where the tile size is larger than the robot (overhead view). The rooms, shown in Figure 1.1, share a wall with a single tile width doorway. In our

discussions, we use N, S, E, W (north, south, east, and west) and combinations NE, NW, SE, and SW for describing directions. As depicted, the robot is located in the west (left) room and is facing east. The eastern (right) room has a charging station tile where the robot may recharge its battery while located on top of that tile. Each floor tile has a RFID tag with a unique information value readable by the robot's RFID sensor.

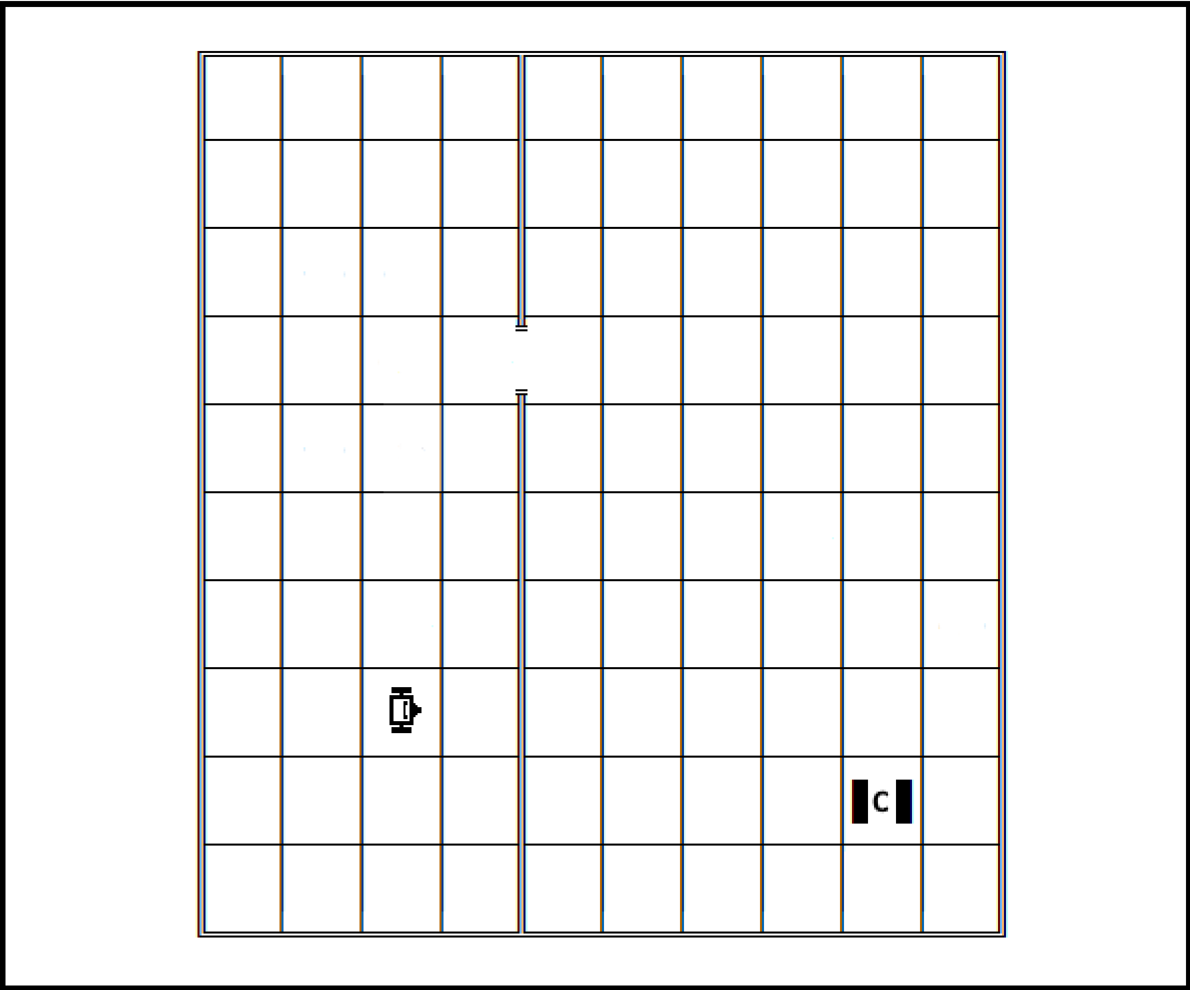

Figure 1.1. A two-room grid environment with a robot in the left room and charging station tile in the right room. The robot's task is to find the charging station from any starting tile.

A blinking LED in the NW corner of each room is used by the robot to discern its orientation. For the west room, the LED color is red and is blue for the east room.

In the thesis statement, we use the snippet “environment domain”. We define this to mean a set of similar environments that differ in one or more specification parameters. For our two-room example environment, we define seven specification parameters: length and width of each room, location of the doorway between the two rooms, the location of the charging station tile, and the orientation of the rooms (the rooms could be oriented north and south for example). Obviously, we are able to specify a very large number of “two-room” environments using various combinations of parameter values.

For the two-room environment, we limit the robot’s movement to the tile directions N, S, E, and W and along the diagonals of NE, NW, SE and SW, moving one tile per movement action. The robot has three high-level actions: forward, rotate left 45 degrees, and rotate right 45 degrees. We abbreviate these action by F, L, and R. After a rotation action, the robot is facing in a new direction and a forward action (F) would propel the robot in the new direction. If the robot is facing a wall either head on or diagonally and tries to execute a F (forward) action, the action fails and the robot does not move but may receive a negative reward from the environment. By using these actions, we say we are taking the robot’s point of view. We could have used actions of “go north”, “go north-west”, “go south”, etc. We call the latter choice the observer’s point of view.

1.1.3. State and Deictic Representations

We define state to mean a description of the robot’s situation in an environment at a certain time point. In any instance of our example environment, the robot is always located on a specific tile and facing in one of eight directions. These two pieces of information, the tile identification and robot’s facing direction, are one way to express the current state. If the

robot performs an action successfully and either moves to a different tile or rotates to a new direction, the current state transitions to another state. In summary, our state representation for this environment consists of two features, tile tag id and robot facing direction. This type of representation is called propositional and is discussed in more detail in Chapter 2.   Note that we have other choices for features to represent state. For example, we could use the robot's ultrasonic distance sensors and define a North-distance feature and an East-distance feature. Combining those two features with a LED color feature would allow us to replace the tile tag id. Getting more sophisticated, we could use the robot's relationship to other objects like the doorway and the charging station.

For any instance of our example environment, there are a certain number of possible states. If we change the environment (like specifying larger rooms), the number of states may change. In the environment shown in Figure 1.1, there are 100 tiles and the robot can face in 8 directions which results in 800 states. By doubling the room dimensions, the number of states quadruples to a total of 3200 states. Changing the doorway location and/or charging station location does not change the number of states but may change state descriptions.

Now we examine deictic representations. A deictic representation, unlike a state, is a partial description of the robot's situation. This partial description focuses on the objects important to the task being performed and is normally taken from the robot's point of view. To illustrate the difference between a state and a deictic description, let's consider the "exit room" task for the situation shown in Figure 1.1. Suppose the RFID under the robot has a value of 'T37'. Since the robot is facing east, the state is ("Tile"='T37',"Facing"='east'). For the deictic description, the important object is the doorway since the robot's task is to exit the room.

From the robot's point of view, the doorway is to its left. A deictic description for this situation could be ("DoorwayDirection"='Left'). Notice if the robot is facing east on another tile close to the current tile, the state is changed but the deictic description is still ("DoorwayDirection"='Left'). This is typical for states and deictic descriptions – many states may have the same deictic description. As we discuss later, this can be a blessing and a curse.

1.1.4. Deictic Satisficing Task Policy

We use the word policy to mean a mapping from situations to actions. A robot has a policy for each task it knows how to perform. The task's policy dictates the actions the robot takes in response to the situations it encounters from the start of the task until the task is completed. For our robot, a task is performed as follows, repeated as needed:

1. The robot reads its sensors and determines the current situation.
2. If the current situation is the "task complete" situation then stop.
3. The robot uses the task policy to map the current situation to an action.
4. The robot performs the action.

For any given task, there may be more than one way to perform the task, some better than others. Each of these "ways" corresponds to a different policy. Given some measure of "best" or "optimal", we say the task policy is optimal if following this policy results in the best measure of performance. For example, our measure may be completing the task in the shortest period of time. Then, a policy that allows the robot to accomplish this would be an optimal policy. Note, that there may be more than one optimal policy.

The word "satisficing" was coined by Herbert Simon in 1956 [44] to mean a way of acting that may not be optimal but is "good enough". So, we say a satisficing task policy is a good

policy but not necessarily an optimal policy. Why would our robot not use an optimal policy for all of its tasks? Well, it may be the case that learning an optimal task policy in a certain environment would take a long time, not worth the effort – a "good enough" policy would do. Another reason is the satisficing policy may be usable in a number of similar environments and an optimal policy would only apply to a single specific environment.

A deictic task policy is a policy where the situations are encoded into deictic descriptions and the task policy maps deictic descriptions to actions. As an example, in the previous sub-section our robot translated its current situation into the deictic description ("DoorwayDirection"='Left'). Since the robot's task is to exit the room, a reasonable task policy mapping of the deictic description to action is ("DoorwayDirection"='Left') ==> L.

1.1.5. Robot Learning

A useful robot has capabilities. We divide these capabilities into two broad categories: perception and performance. Perception deals with the robot's ability to translate sensor readings into information. A perception capability could be simple such as reading an RFID tag or more complicated like face recognition. Although perception is extremely important, we do not study it in this work. We assume the robot has the perception functions we need. The second category, performance, is about taking actions to achieve an objective. How does the robot acquire performance capabilities? The obvious answer is to have a software engineer program the robot to do all its tasks. Another answer, the one we are interested in, is to have the robot learn these capabilities. Hence, our work focuses on the performance category, specifically robots learning to perform tasks. Basically, a robot learning to perform a task

translates to (somehow) filling in the task policy with a good (or possibly optimum) action for each situation the robot may find itself in.

We, as humans, typically learn by being "told" where we use the word "told" somewhat loosely. Some examples of being "told" are watching a how-to video, reading a textbook, and being taught by another person. Sometimes though, we learn a task on our own. An example of this is mastering a new video game. This type of learning utilizes trial and error – we try different actions and see what happens. We play the video game over and over again, gradually making improvements in our techniques, until we have learned to play the game proficiently. This is how our robot is to learn tasks – by trial and error. We call "playing the game" a training episode. The robot is started in an initial situation and it then tries to perform the task. At first it is not very good – either failing or taking many more actions than needed. But, as it gathers more experience, the robot's performance improves. Finally, after some number of training episodes, it reaches a satisficing level of performance in the task.

1.1.6. Example Environment Task

We have completed each piece of the thesis statement. Let's now give a preview of what's to come – the proverbial "where's the beef?". At the core of "the beef" is the notion of learning to perform the task in a number of similar environments (the environment domain). To explain, we continue the example and cover how a robot uses a well-known learning algorithm to learn a task policy (using states) for a specific two-room task. We follow that with a discussion of a task policy using deictic descriptions where the deictic policy is applicable to a range of environments.

The example task the robot learns is the “go to charging tile” task. As the task name indicates, the robot, starting in either of the two rooms, executes actions to move to the charging station tile. For scoring the robot’s performance, the robot begins the task with a battery charged to 100 energy units and is debited 1 energy unit for each action it takes. The best performance is completing the task with the most energy units. The robot fails the task if it runs out of battery energy before reaching the charging station tile.

To learn a task, the robot must have a learning algorithm. The learning algorithm controls the robot’s behavior during training. As the robot moves around, directed by the learning algorithm, the situations it encounters, the actions it takes, and the scores it receives are used to update the learning algorithms internal data structures. At some point, the learning algorithm completes and the end result is a policy for doing the task. After that, the robot uses the task policy to perform the task, no longer needing the training algorithm.

We train the robot by giving it a training algorithm and letting it attempt the task over and over again. As stated previously, each attempt is called a training episode. At the start of each episode, we give the robot the 100 energy units and place it in a random location in one of the two rooms (but not on the charging tile). The training algorithm takes over the robot and starts moving the robot. The episode ends either when the robot reaches the charging station tile or it runs out of energy by taking an excessive number of actions.

For an example training algorithm, we use the Q-learning algorithm [3]. The algorithm maintains a Q-value (a real number) for each state-action pair. For our example environment there are 100 tiles and the robot can face in 8 directions which results in 800 states. Note that for 8 of these states, the robot is on the charging station, leaving 792 possible starting states.

Since there are 3 actions, the total number of Q-values to learn is 2376. Each state/action pair's Q-value is the maximum score the robot would expect to receive if it started in the state and took the action and then completed the task in the best possible manner. The Q-learning algorithm starts with some initial Q-value (perhaps 0.0 or a random number) for each state-action pair. Then as the robot takes actions, repeatedly transitioning from a current state to a next state, the current state/action Q-value is refined with an update rule based on the transition. Under certain assumptions, the Q-values will converge and stop changing. The task policy is filled in by examining the Q-values for each state/action and using the action with the largest Q-value for that state. The resulting task policy is guaranteed to be an optimal policy if the Q-values have converged.

To give a feel for Q-learning performance on the "go to charging tile" task, we carried out an experiment with Q-learning using a typical setup and tuning. The robot was started in all 792 initial states and then allowed to concentrate on the ones it was having trouble with. After 922 episodes, the robot learned a task policy for the "go to charging tile" task which it could use for all 792 starting states. This policy was a satisficing policy but was optimal for 556 of the starting states. Comparing the policy to the optimal policy, the learned policy was non-optimal for only 6% of the states. The total number of actions taken by the robot during learning was 32736. Not counting time to set up episodes, this translates into about 9 hours of training time if each action takes 1 second to complete. Note, a completely optimal policy could have been learned if we had allowed the training to continue for a longer time.

Our robot, after using Q-learning, has a satisficing task policy for the specific two-room environment shown in Figure 1.1. But, if we change to a two-room environment with a

different doorway location or a different charging tile location or different size rooms, our task policy will no longer work. The robot must go through the learning process for the changed environment.

Now, we look at the case of using deictic descriptions instead of states. The deictic descriptions are encoded using features and then mapped to deictic descriptors which we cover in more detail in Chapter 2. For the task, the two important features from the robot's point of view are 1) "where is the charging tile?" and 2) "where is the doorway?" In fact, if the robot sees the charging tile, it does not care where the doorway is. The robot's perception functions must recognize the doorway, the charging station tile, and their direction relative to the robot. These features are utilized to form the seven deictic descriptors shown in Table 1.1. The first three descriptors, D1, D2, D3, are encountered in the left room of the example where there is no charging station tile signified by "ChargeTileDir=None. Descriptor D3 is translated into English as "there is no charging tile in this room and the doorway is to my left or behind me". The remaining four are encountered in the right room. So, every situation in the example two-room environment will be perceived as one of these seven descriptors. In fact, these descriptors are independent of the domain's 7 specification parameters and therefore usable in any instance two-room environment.

For training, the robot was started in 2 locations in the west (left) room. The first was in the upper left corner facing east and the second along the west wall 6 tiles down facing south. The robot learned the deictic task policy shown in Table 1.1 using our learning algorithm which we cover in Chapter 4. In the table, the robot knows the task is complete when it perceives deictic descriptor D7 (charging tile is under the robot). The learned deictic task policy is optimal

for all 792 starting states. During training, the robot used a total of 1015 actions, taking about 17 minutes at 1 second per action. This is a large reduction compared to Q-learning which took 9 hours for training. Directionally, this makes sense. Q-learning had 2376 values to learn by gathering data from results of many episodes. The deictic policy has only 6 physical actions to learn. Also note that the deictic policy can be used for many different 2-room environments whereas the Q-learning policy is specific to this 2-room environment.

Table 1.1. The robot's deictic descriptor policy for the "go to charging tile" task.

Deictic Descriptor	ChargeTileDir	DoorwayDir	Action
D1	None	Front	F
D2	None	Left	L
D3	None	Right	R
D4	Front		F
D5	Left		L
D6	Right		R
D7	Under		Q

1.2. Thesis Outline

The outline of the thesis is as follows. Chapter 2 covers background information such as reinforcement learning, Markov decision processes, transfer learning, hierarchical task decomposition, and partially observable environments. Related works are covered in Chapter 3. In Chapter 4, we introduce our basic learning algorithm. Following that, in Chapter 5, we test our algorithm on seven example environment domains and compare the performance with Q-learning, R-max and existing works if available. In Chapter 6, we improve our basic algorithm with experience retention memory and reversible actions. We end with a summary, conclusions, and future work in Chapter 7.

Chapter 2. Background

2.1. Reinforcement Learning and Markov Decision Processes

Reinforcement learning is a branch of machine learning where an agent (in our case a robot) learns how to act optimally in an environment with rewards (or costs) as its only feedback on performance. A robot deciding on what actions to take at each time step from a starting position to an ending position is often modeled as a Markov decision process (MDP). In Sutton's and Barto's introductory text on reinforcement learning [6] they comment about MDPs: "they are all you need to understand 90% of modern reinforcement learning". In a more recent text [7], Otterlo and Wiering state: "In fact MDPs have become the de facto standard formalism for learning sequential decision making." So, in the following, we briefly describe the nomenclature and learning algorithms for MDPs as normally encountered in the reinforcement learning literature [7].

A MDP is a tuple $M = (S, A, T, R)$ where S is a finite set of states $\{s_1, \dots, s_n\}$, A is a finite set of actions $\{a_1, \dots, a_k\}$, T is the state transition probabilities from the current time step i to the next time step $T(s, a, s') = \Pr(s_{i+1} = s' \mid s_i = s, a_i = a)$, and R is the scalar reward function $S \times A \times S \to \mathbb{R}$.

The action taken by a robot in a state is determined by its policy. A robot policy can be deterministic: $\pi(s): S \to A$ or stochastic: $\pi(s, a): S \times A \to [0,1]$. We only utilize deterministic policies in our work.

When a robot learns a task in an environment modeled as a MDP, the state transition probabilities and reward function are typically not known to the robot. In situations where the robot is expected to learn by trial and error (i.e., reinforcement learning), the methods are

generally divided into value function approaches and policy search [4] with value function-based approaches being by far the most prevalent.

The value to the robot of being in a particular state of an environment modeled as a MDP is related to the robots policy and the discounted rewards it receives by following that policy [6]. The value to the robot of a state s under a policy π is defined recursively by the Bellman Equation [1]:

$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')\big(R(s, \pi(s), s') + \gamma V^{\pi}(s')\big)$$

where γ is the reward discount rate, $0 \leq \gamma \leq 1$. This value is unique when either the discount rate is strictly less than one or the policy leads eventually to the task completion state.

The value of an action a taken by a robot using policy π in a state s is the Q function defined as:

$$Q^{\pi}(s, a) = \sum_{s'} T(s, a, s')\big(R(s, a, s') + \gamma V^{\pi}(s')\big)$$

Here, the robot takes the action (which may not be its policy) and thereafter follows its policy. The maximum value of a state is given by the Bellman [1] optimality equation:

$$V^{*}(s) = \max_{a} \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V^{*}\,(s'))$$

$V^{*}(s)$ is short hand for $V^{\pi^{*}}(s)$ where π^{*} is the optimal policy π^{*} which maximizes the sum of the rewards received by the robot while performing a task.

There are basically two types of learning algorithms for the case of unknown state transition probabilities and reward function: model based (where some sort of model of the transitions is developed during exploration) and methods where no transition model is used.

An important learning method of the latter type is Q-learning [3] which updates the Q function incrementally by taking actions and sampling the environment. Take for example a robot in state s_i which takes an action a_i and then receives a reward r_j and transitions to state s_j. The Q function is updated by:

$$Q(s_i, a_i) = Q(s_i, s_i) + \alpha \left[r_j + \gamma \max_{a_j} Q(s_j, a_j) - Q(s_i, a_i) \right]$$

where α is the learning rate parameter, $0 < \alpha \leq 1$. Watkins [3] shows that with enough exploration, the Q function updating will converge to the optimal Q^*. Then, the optimal policy for any state s is to choose the action a which maximizes $Q^*(s, a)$.

An example model-based method is the R-max algorithm [8]. This method maintains a model which is optimistically initialized. The agent or robot always follows an optimal policy obtained from the model (no exploration phase is needed) and updates the model as the environment is observed. States move from unknown to known as the robot gains experience. Once all states are labeled known, learning is complete. As a side note, in most of our experiments, the number of actions required to learn a policy favors a model-based algorithm such as R-max (orders of magnitude reduction over Q-learning in some cases). Obviously, this is very important for robot task learning time and user time.

A key observation is policies learned by the above methods, if given enough samples, are optimal. For the given specific environment, the robot will receive the maximum reward when following the policy from a start state to a goal state.

2.2. Hierarchical Task Decomposition and Semi-Markov Decision Processes

Learning large complicated tasks is a difficult undertaking for robots. As humans, we commonly break such problems into smaller chunks which are easier to learn. In school, our

teachers guided us in this way, instructing on, for example, addition and subtraction followed by multiplication and division. This strategy of task decomposition has been used by many researchers for reinforcement learning [20,32,38,40,42,43,50,60,71,88]. Sutton, Precup, and Singh [9] brought together many of the earlier works into a general framework for temporal abstraction with an extended action called an option. An option is a closed-loop policy for taking actions over a period of time and can be treated in the MDP framework as a kind of super action with primitive actions being a special case option. The motivation for developing options was for improved learning performance. Note that using options does not give the agent/robot any additional "powers" as explicitly stated in the text "one can ultimately do just as well with primitive actions as one can with options" [9].

An option is defined as a tuple (I, π, β). The first component, I, is a subset of the MDPs non-terminal states and is called the the initiation set. An option may only be selected when the current state is a member of the initiation set. The second component is the option policy, π, defined as a MDP policy. The third component is a termination condition, β, which gives the probability of ending the option for any encountered state.

Learning a task with options is accomplished in steps. The first step is dividing the task into smaller pieces. This is most often done by the teacher [17,19,32] but there are works [97] for automating the decomposition. The next step is learning a policy for each piece to develop each option. This learning may be accomplished with any of the standard reinforcement learning algorithms. The option learning starts at the lowest level and progresses hierarchically to the top level options. Finally, the overall task is learned using an action set which includes the previously learned options and perhaps primitive actions.

2.3. Partially Observable Markov Decision Process

A partially observable Markov decision process (POMDP) is frequently used to describe environments where the robots view of the current situation is not complete or is inaccurate with respect to the task being performed. In other words, a POMDP is a MDP where the robot, for some reason, does not obverse the true current state of the environment but some part/distorted version of it. For example, the robot's sensors could be noisy or some object in the environment is hidden from the current view of the vision system.

Mathematically, a POMDP adds two elements to the MDP formalism [72]: a set of observations, $\Omega = \{o_1, \dots, o_m\}$, and an observation function $O: S \times A \rightarrow \Pi(\Omega)$. This function gives a probability distribution over the set of observations for each action a and resulting state s'. So, as the robot takes actions, it perceives a series of observations, each of which depend on the true state with probability $O(o \mid s', a)$.

This uncertainty of the current state results in the robot being unsure of the proper action to take. Obviously, POMDPs are more difficult to analyze than MDPs and reinforcement learning techniques for MDPs do not extend to POMDPs. Crook, in his dissertation [67], presents a lengthy discussion of the many approaches researchers have taken to ease the robot's predicament of learning a task policy in partially observable environments. Many of the methods rely on recovering or estimating the underlying MDP.

For our case, the robot has the ability to observe the complete current state but chooses to focus on only part of the task environment. This seems counterintuitive at first glance. Why put yourself in a more difficult learning situation? Later in this work, we show that there are advantages to taking this approach.

2.4. Transfer Learning

Learning tasks from scratch can be difficult. The idea of "transfer learning" is to build on knowledge of previously learned tasks to help in learning new tasks, especially for speeding up the learning process. Transfer learning for tasks covers not only tasks in the same environment domain (multi-task learning) but also learning from tasks in entirely different environment domains. In this work, we only consider multi-task learning where task environments are drawn from a common environment domain and each task has a final state (a goal). Most research in multi-task learning assumes each task has the same state-space and action set. We relax this and only require the same action set.

As previously discussed, Q-learning may be used to learn a MDP (task) policy tabula rasa. Let us suppose we want our robot to learn policies for ten MDPs, all taken from the same environment domain. The straightforward approach is to train the robot using Q-learning on each of the MDPs, learning 10 policies, and be done with it. But, suppose we want to teach the robot how to complete a task for any MDP from the environment domain. How can we do that? This cannot be done with Q-learning – a policy is learned for each MDP as it is needed. So, we turn to transfer learning to make learning a policy for a new MDP quicker.

Taylor and Stone [5] produced a transfer learning framework which we briefly cover here. Their work differentiates RL transfer learning methods categorized using five basic dimensions. These are: task difference assumptions, source task selection, task mappings, transferred knowledge, and allowed learners. Their survey assumes the RL problems are Markov decision processes. We are interested in task differences as they are the most applicable to our work. Also note that the meaning of the word "task" is different from a layperson's conventional

definition. Its meaning includes not only what is to be accomplished but also the environment and situation in which the task is performed. Taylor and Stone state this explicitly by equating task and MDP.

The task difference assumptions divide methods by differences of source MDPs and target MDPs in transition functions, reward functions, state spaces, goal states, start states, action sets and/or state features. They also add the number of objects in source and target tasks to cover relational reinforcement knowledge transfer. Some of these differences interact. For example, if the action sets are different then the transition functions must be different to account for the dissimilar actions. The opposite does not need to be true, the transition function can change with equal action sets. Our allowable task differences include all the above except the action set.

Taylor and Stone also layout 5 transfer learning metrics which compare the benefits of using a transfer method over not using the method. These are named “Jumpstart”, “Asymptotic Performance”, “Total Reward”, “Transfer Ratio” and “Time to Threshold”. The “Jumpstart” metric measures the improvement in the initial performance, the “Asymptotic Performance” in the final performance, and the “Time to Threshold” somewhere in between. The “Total Reward” and “Transfer Ratio” measure improvements in total reward.

2.5. Environment Domains and Domain Tasks

In the transfer learning section above, task and Markov decision process were used interchangeably. A robot performing an episodic task in an environment modeled as a MDP begins in an initial state and, after some number of actions, ends in a final state completing the task. Learning the task is the process of learning a policy that the robot uses to go from the

initial state to the final state. We, however, are interested in a more general definition of a task, one where the robot learns a single policy that covers all MDPs drawn from an environment domain. So, for us, a domain task policy applies to all the MDPs as opposed to each instance MDP having its "instance" task policy. Once the domain task policy is learned, the robot may use it to perform the task in any instance MDP. We presented an example of this in Chapter 1 where the robot learned a single deictic descriptor policy which covered a multitude of 2-room environments.

For our work, an environment domain consists of a prototype environment and a set of specification parameters. Each environment instance is a result of a particular setting of parameter values applied to the prototype environment and is modeled by a MDP. An environment has a set of configurations which are arrangements of elements drawn from a world. Each state of the instance MDP corresponds to a configuration of the instance environment. A total configuration is the arrangement of elements and a robot in a particular pose. We sometimes, when discussing configurations, leave out the word "total" if in the context the meaning is clear. The robot has a set of physical actions applicable to the domain environments. When the robot performs a legal action for the current configuration, the current configuration changes deterministically to a new configuration. Certain configuration and action pairs are not legal. In these cases, if the robot attempts the action, the configuration does not change and the robot may receive a negative reward. The user is able to initialize an environment to any configuration of the environment instance.

As a recap, we offer this listing of meanings of terms:

- World: A (perhaps stylized) subset of the "real" world. Examples are the blocks world and the grid world. We do not define these rigorously but use them as a backdrop for describing what kind of environment we are discussing.
- Environment domain: A family of environments defined by a prototype environment and a set of specification parameters.
- Prototype environment: The environment that forms the basis for all instance environments (and MDPs) in an environment domain. The prototype environment has elements from a world, a set of physical robot actions, and a set of configurations. Every configuration can be reached from any other configuration by applying actions from the action set.
- Specification parameters: The parameters used to make an instance environment. For example, consider a grid world environment domain where the prototype environment is a single room and an exit door. The specification parameters could be the room size and door location. Then, any environment instance could be created by specifying values for these parameters.
- Instance environment: One of the environments of an environment domain. All instance environments of an environment domain have the same set of robot actions. The set of configurations for an instance environment depend on the prototype environment and parameter values used to define the instance.
- Environment configuration: A complete description of an instance environment situation. A MDP state (or deictic description) of any desired class (atomic, propositional, relational, etc.) may be derived from a configuration.

- Domain task: A task defined for all instance environments of an environment domain. The initial and final task situations are spelled out in a global manner relative to elements in the prototype environment and independent of the environment domain's specification parameters. There exists a domain task final situation function which maps any instance environment configuration to a boolean value where True signifies the configuration is the final task situation.

2.6. Representing the Environment

Robots typically map their sensor readings (percepts) into some sort of internal representation. These representation methods can be divided into 4 basic classes: atomic, propositional, deictic, and relational [16]. In the atomic state class, each percept is mapped to a distinct symbol. For the propositional state class, percepts are commonly represented as a vector of attribute/value pairs. The relational representation class is more structured, using descriptions similar to first-order logic to encode the environment objects and relations between objects. The deictic representation, in some ways, bridges the gap between propositional and relational representations [12]. Unlike the other classes, the robot takes part dynamically in what is perceived. Using a focusing mechanism, the robot centers its attention on what is important to the task at hand and essentially ignores the rest.

To be concrete, we look at a configuration from a warehouse environment and present example representations for the 4 classes. Our coverage is weighted towards the deictic representation since it is a central component of our work. The interested reader is directed to Mitchell [113] for an excellent treatment of propositional attribute/value representations and to van Otterlo [16] for the relational representation.

Consider a robot in a warehouse having 4 pallets with 5 colored boxes stacked as shown in Figure 2.1. The robot can move from pallet to pallet and pick up and put down a box. The robot's task is to pick up the green box. So, for each class, we define a representation framework and then use it to instantiate a state (or description) of this particular environment configuration.

2.6.1. Atomic Representation

For the atomic state, our approach is to assign a distinct numeric symbol to each possible configuration. To accomplish this, the image of the scene from the vision sensor becomes the input to a neural net which produces a unique scene number. The neural net is sophisticated enough to ignore minor variations in the image so as not to yield false changes in scene. The scene number is the state representation. Perhaps this configuration is state '1057' or maybe state '4821' – it doesn't matter as long as each configuration is mapped to a distinct numeric symbol.

2.6.2. Propositional Representation

The propositional representation is more expressive. We specify a set of features (attributes) to capture the essence of every possible configuration. In our example representation, each pallet is represented by a feature so we have features "Pallet1", "Pallet2", "Pallet3", and "Pallet4". The value of a pallet feature is a string of the box names on the stack, separated by spaces and bottom box first. For example, "B2 G1" is the string representing the stack of 2 boxes on pallet3 so feature "Pallet3" has the value "B2 G1". The robot position is conveyed by the feature "RobotPos". This feature has the integer value of the pallet the robot is currently facing. The robot can be holding a box which is represented by the feature

“HoldingBox” which has a value of the box name or None if not holding a box. Table 2.1 shows the propositional state representing the environment configuration of Figure 2.1. Note, we did not represent the box colors but this would be easy to add.

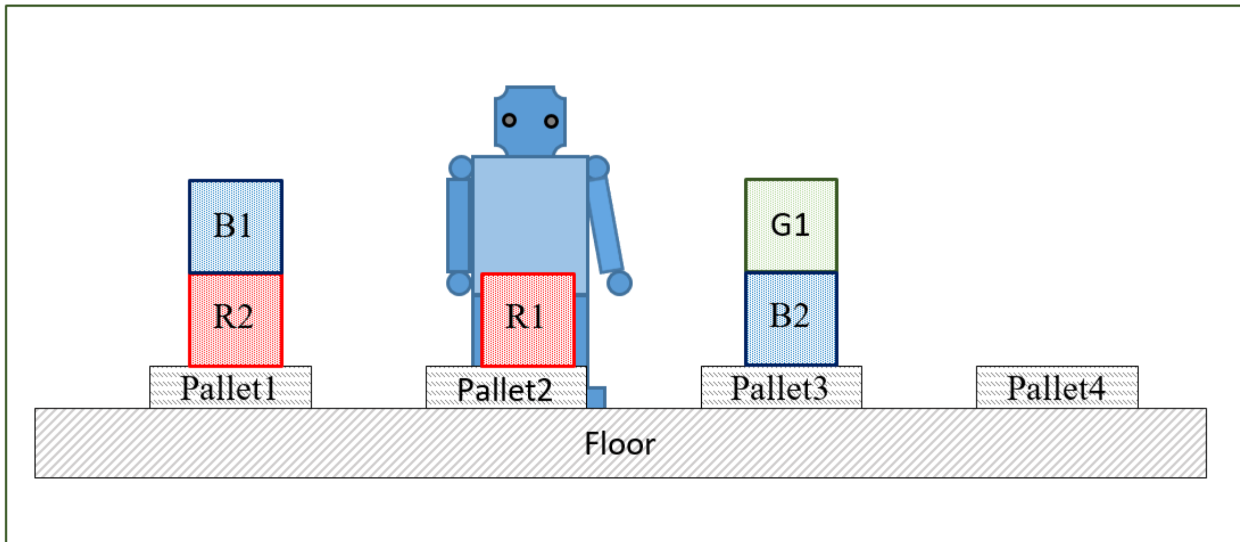

Figure 2.1. A warehouse environment with pallets and colored boxes. The robot’s task is to pick up the green box.

Table 2.1. A propositional state for the warehouse environment with a robot, 4 pallets, and 5 boxes. The robot is facing pallet2 and is not holding a box.

Feature Number	Propositional Feature	Feature Value
1	Pallet1	“R2 B1”
2	Pallet2	“R1”
3	Pallet3	“B2 G1”
4	Pallet4	“ “
5	RobotPos	2
6	HoldingBox	None

2.6.3. Relational Representation

The relational representation defines the scene in terms of objects with properties and relations between objects. Typically, a scene configuration is described as a list of ground facts [18]. For the example, we could use the predicates ‘on’, ‘clear’, and ‘facing’. The scene is then

encoded as the list [on(R2,Pallet1), on(B1,R2), clear(B1), on(R1,Pallet2), clear(R1), on(B2,Pallet3), on(G1,B2), clear(G1), clear(Pallet4), facing(Robot,Pallet2)].

2.6.4. Deictic Representation

In the deictic representation, a deictic description's meaning is relative to the entity that uses it and the setting in which it is used [12]. Although the other 3 representations are silent about what particular task the robot is undertaking, the deictic representation description of a situation is constructed with the task in mind and depends on what the robot is focused on. Note, we do not call this description a deictic state as we reserve the word "state" to denote a Markov representation of the configuration. We use the word "description" to signify a deictic representation of an environment configuration. We further logically map the deictic descriptions to what we call the deictic descriptors. So, continuing with our example, suppose the robot's task is to pick up the green box and it is currently focused on red box R1. From the robot's point of view and its task, the scene could be described by the features "Holding" and "Focus". The "Holding" feature is obviously the object in the robot's hands. In a propositional representation, this would have a value like the name of the box. But, for the deictic representation, we take the task into account. Since the robot is interested in the green box, we give the "Holding" feature 3 possible values, 'GreenBox', 'NonGreenBox', and 'None'. The "Focus" feature is what the robot is currently looking at. This could be the pallet, a box, or the robot could be looking above the stack. Therefore, we give the "Focus" feature values of 'Pallet', 'GreenBox', 'NonGreenBox', and 'AboveStack'. Since the robot is currently focused on the red box R1, the deictic feature values would be ("Focus"='NonGreenBox', "Holding"='None'). Table 2.2 shows the table of deictic descriptors derived from the deictic

description features. Notice that we only have 9 descriptors as opposed to 12 possible combinations. This is because the descriptors take the task into account and once the robot has the green box, the task is over so there is no need to worry about the focus feature.

Table 2.2. The deictic descriptors for the "pick up the green box" task.

Deictic Descriptor	**Focus**	**Holding**
D1	Pallet	None
D2	AboveStack	None
D3	notGreenBox	None
D4	GreenBox	None
D5	Pallet	notGreenBox
D6	AboveStack	notGreenBox
D7	notGreenBox	notGreenBox
D8	GreenBox	notGreenBox
D9		GreenBox

So what are these deictic descriptors and how do they come about? As the robot changes its focus, different parts of the environment come into view. For each of these environment sub-sections, the robot reads its sensors and internally maps the objects in the view to names.

Deictic Name: A deictic name is a name (chose by the designer of the RL method) that can refer to an object or a set of objects in the environment without using the object's name. For example, a deictic name could be "the_box_I_am_looking_at".

Deictic values are defined as follows.

Deictic Value: A deictic value can be a boolean, a number, a string, or a deictic name.

For each task, the robot has a set of features tailored for the task. We call these the deictic features.

Deictic Feature: A deictic feature is a deictic name that takes a deictic value as its value.

Examples of deictic features are “RobotPosition” and “DoorDirection”.

Deictic Description: A deictic description is a list (conjunction) of (deictic feature, deictic value) pairs.

An example of a deictic description is [(“RobotPosition”, 2), (“DoorDirection”, ‘South’)].

Finally, a deictic descriptor is the same as a deictic description except it may contain a “don’t care” true term in place of a deictic feature/value pair in the deictic description. These two notions are used somewhat interchangeably.

As mentioned above, the number of descriptors is less than the number of configurations which results in perceptual aliasing conditions. This is easily illustrated in our example for the three cases where the robot is facing different pallets and observing the bottom box on the pallet. In each case, an atomic, propositional, or relational representation would result in three distinct states. However, for the deictic representation, each case results in the same deictic descriptor D3 as shown in Table 2.2. The robot is not holding a box and is observing a non-green box.

Now let’s cover the robot’s focusing mechanism in more detail. The robot, for each task and associated environment domain, has methods for changing its focus of attention. The first method and simplest is to have the focus pointer move when the robot physically moves. The second method is a focus pointer action. The robot performs the focus pointer actions mentally, there is no physical movement. As such, the focus pointer actions are not included in the environment domain physical action set. To illustrate each of these methods, let’s look again at Figure 2.1. For this environment, the robot has a physical action to move from one stack to the next. So, in this case, we elect to have the robot’s focus pointer move with the

robot. When the robot stops at a stack, the focus pointer is on the pallet of that stack. If the robot moves to a new stack, the focus pointer moves to the pallet of the new stack. This is an example of the focus pointer movement tied to a physical movement. Once the robot is facing a stack, it can perform a mental focus pointer action to move the focus up the stack to the first box on the pallet. Then, if desired, the robot could perform the mental action again to focus on the next box on the stack. Each time the focus pointer moves, the perceived deictic features are updated which may result in a descriptor change. Note that the robot's focus pointer may also include some different parts of the environment such as, in the example, what it is holding.

For our learning methods, the descriptors must be independent of the environment's specification parameters and pertinent to the robot's task. We require the maximum number of descriptors to remain constant when the environment's size changes. Again looking at Figure 2.1, suppose each environment instance can have a different number of named boxes, more colors than just green, red, and blue, and any number of pallets. The robot's task is to pick up the green box (there is only one green box in any environment instance). Forming descriptors from the features "Holding" and "Focus" described above meets our requirements. No matter how many boxes there are or their names or their color, they each belong to either the 'NonGreenBox' or 'GreenBox' category. The number of pallets does not matter either since the "Focus" feature only returns the generic name 'pallet'.

Another requirement for deictic descriptors is that there must be at least one descriptor which signifies the task is complete. In our example, when the feature "Holding" has the value of 'GreenBox', the task is complete.

2.7. Physical and Mental Environments

The physical environment can include the robot but not always. For example, in the blocks world, the gripper is typically not included in the environment description. In contrast, a grid world environment almost always includes the robot. In any case, the robot is the only agent and any change in the environment is due to a robot action. The only exception to this is when the user initializes the environment during episodic training.

Besides the physical environment, we utilize a nonconcrete environment we call a mental environment. This environment contains the deictic focus pointer discussed in the previous sub-section and the abstract machinery for managing the current policy which we discuss shortly. Like the physical actions used in the physical environment, the robot has mental actions at its disposal. The robot is able to "move" the focus pointer and switch the currently active policy with mental actions.

Using a deictic representation for a task policy brings along the blessing and curse of many environment configurations mapping to a single deictic descriptor. We saw this in the 2-room example in Chapter 1. The blessing is the reduction in the size of the policy. The curse is the partial-observability problem - in this case commonly called perceptual aliasing. For our algorithm, perceptual aliasing becomes a problem when performing the task requires the robot to executed different physical actions in response to the same deictic descriptor. To handle this situation, we give the robot the ability to switch to a different policy. This adds a hierarchical capability but makes learning more difficult for the robot. It must figure out when it needs to use an alternate policy and also learn the alternate policy.

Having a mental environment forces us to include it in the description of the states in a MDP. In Chapter 4, we discuss in more detail how this is handled with a Z-state, a combination of the state of the mental environment and the state of the physical environment. Also, the robot's starting point for a task includes how the mental environment is initialized. We designate one policy to be the "main" policy. When a task is started, the main policy is the current policy. The initialization of other pieces of the mental environment are handled on a task by task basis.

2.8. Optimal and Satisficing Behavior

As discussed in the introductory section of Chapter 1, the phrase "satisficing behavior" means a way of acting that may not be optimal but is "good enough". Of course, we would like our robot to learn a domain task policy that is optimal for each instance environment. But, if this is not possible, we may nonetheless choose a satisficing domain task policy to avoid training the robot optimally on every instance MDP.

In the 2-room example in Chapter 1, the robot learned an optimal deictic policy for "go to charging tile" task. This is not the case for all the remaining examples. For the test examples where our learning algorithm finds a less than optimal policy, we compare the robot's performance to the optimal performance and to a robot taking random actions to reach the task's goal.

Chapter 3. Related Works

Most all of our representation methods are relative to the robot and use a variation of the deictic state representation class. The robot does not name objects directly but uses indirect naming. For example, the top block on a stack of blocks could be "the-top-block-on-the-stack-of-blocks-I-am-facing". The deictic representation class was introduced by Agre and Chapman [14]. They used deictic representation in a video game to show how much simpler it was as compared to the relational representation. Their work did not involve learning.

Whitehead and Ballard [15] were the first to develop a reinforcement learning algorithm using a deictic representation. They realized the major problem of deictic representations which they named "perceptual aliasing" now more commonly called partially observable. They performed experiments and observed that even if the agent is started at the optimal policy, the classical reinforcement learning system becomes unstable and actually moves away from the optimal policy. Their learning algorithm, Lion, is based on Q-learning but includes a component that suppresses action-values of inconsistent decisions, in effect, avoiding aliased states. They tested Lion on blocks world tasks which were divided into difficulty classes of "easy", "intermediate", "difficult", and "most difficult". Their results showed the Lion algorithm could learn to solve all the tasks perfectly except for the "most difficult" class of tasks for which it failed about 10% of the time.

McCallum, a student of Ballard, developed a series of reinforcement learning algorithms [13] for agents with "selective perception" and "hidden" states. His idea is that an agent needs to narrow its focus to the items related to the task (which we also follow). A real-world agent would, in practice, not be able to always perceive the entire environment, resulting in hidden

state. To neutralize the partial observability of selective perception and hidden state, his algorithms use short-term memory to remember percepts/features of previous time periods. Not all of his experiments ended with success, including the blocks world example of Whitehead and Ballard [15].

Muñoz [37] developed an algorithm called CANDID for embedded agents. His agent uses "deictic sensors" and can only observe a fraction of the environment. The deictic sensors are:

- Foveal sensor, the visual focus of the agent, gives information about the environment (like block, ground, and color) at the location of the focus.
- Peripheral sensors (4 total) give partial details of the visual field around the focus. Each sensor indicates the presence or absence of an object. The peripheral sensors are located above, below, left, and right of the foveal sensor.

The agent has 4 actions to move the visual focus. It can move the sensors as a group one visual position left, right, up, or down.

The CANDID algorithm has two tables, one related to successful trials and the other to failed trials. For successful trials, each state-action pair's entry in the successful table is incremented. Similarly, for failed trials (when too many actions are taken), each state-action pair's entry in the failed table is incremented. These two tables are used to select actions during an episode. Specifically, for a situation, the algorithm chooses the action with the highest possibility of obtaining a positive reward. The CANDID algorithm learns reliable and general behaviors with acceptable (as opposed to optimal) performance. It can also handle a moderate level of perceptual aliasing.

The CANDID algorithm has 4 main tuning parameters:

- N: This parameter determines the initial value of the statistics and a weighting parameter. N is in the range of 50 to 200.
- Maximum Actions: This parameter is the number of actions allowed in a trial and determines when a failed episode occurs.
- Maximum Repetitions: This parameter determines how many times a specific task is attempted and is helpful for learning the task in a difficult environment instance.
- Maximum Episodes without a Policy Change: This parameter is the stopping criteria for CANDID.

To speed up learning, five heuristics are also used (which we will not discuss here).

The author presents an example task (similar to Whitehead and Ballard [15]) where the agent must find a specific block in a group of from 5 to 8 blocks in a visual 12x10 scene with the bottom two rows being the “ground”. The performance of CANDID is compared empirically to Q-learning, backward Q-learning (updates are done after each trial), and SARSA(1). The results showed that CANDID is better than the other algorithms for this task. Later in this document, we compare our algorithm to CANDID.

Deictic state representation was revisited by Finney, Gardiol, Kaelbling, and Oates [12]. They used a history of observations to combat the deictic representation’s partial-observability problem per McCallum [13]. A standard Q-learning algorithm with a neural net function approximator was chosen for the experiments. Using a similar blocks world setup as Whitehead and Ballard [15], they compared the learning performance of the deictic

representation to a fully observable propositional representation. As the title of their paper suggests, their empirical results showed deictic representation worsens learning performance.

Ravindran [17] studied MDP minimization by exploiting structure and symmetries of the problem. He was able to use deictic representation in some restricted settings and was able to use the deictic representation in a hierarchical manner.

Learning a large MDP can be very difficult and time consuming. To ease this problem, hierarchical reinforcement learning breaks a large MDP into smaller chunk MDPs. The smaller MDPs are easier/faster to learn. Sutton et al. [9] defined the term options for temporally extended actions which combined concepts from previous research into a simpler reinforcement learning framework.

Croonenborghs et al. [19] extended options to the relational domain to aid in transfer learning. Their method first learns a task utilizing primitive actions and a relational reinforcement learning algorithm (like relational Q-learning). The relational options are then induced from a dataset extracted from the Q-function of the task. Their experiments showed that using the induced options in the task improved learning performance over the primitive actions case. The following discussion describes their grid world example in more detail.

Each of the grid world environments has a sequence of rooms connected by colored doors with colored keys on the floor. In each room, the robot must navigate using actions up, down, left and right to a key whose color matches that of the exit door. Once the robot is at the proper key, it must pick up the key and then navigate to and through the exit door. If, when entering a room, the robot already possesses the correct colored key, it can bypass getting another key and proceed directly to the exit door. The robot's goal is reached when it

exits the last room in the sequence. To make this task difficult and not consistently solvable by relational reinforcement learning (or in fact any other reinforcement learning method), the researchers specified that the robot must be able to perform the task (without relearning) when presented with new, never seen before environments where the number of rooms is varied, the sizes of the rooms are varied, the door locations are varied, and the location and number of keys is varied.

This environment/task specification is not amenable to a propositional version of Q-learning or R-max and is difficult for relational Q-learning. So the researchers approached the problem with a hierarchical method using skills (options). Their method of learning skills involves 4 steps. In the first step, the agent learns (or partially learns) the task using a relational reinforcement learning algorithm [18]. For the next step, they create a binary policy-based labeled dataset of state-action pairs extracted from the Q-function of the first step. Then, from this dataset, they induce a relational decision tree that predicts whether the action will be executed by the policy. Finally, they extract a relational policy for the option. Because options induced in this way may not always terminate, all non-goal states are given a non-zero probability of terminating the option. For the grid world skill/option learning, the agent explores for 200 episodes with a maximum of 500 actions per episode. If the goal is reached, the agent receives a reward of 1 and 0 otherwise. Two options, “pickup_key” and “find_door”, are learned from a dataset created from 100 of the episodes. Note that the agent always perceives the complete description of the configuration consisting of the dimensions of the different rooms, the locations and colors of the doors, the location and colors of the keys, the

keys the agent possesses, the agent's location and the goal location. The agent also uses the typical up, down, left, and right movement actions and an action for picking up keys.

Croonenborghs et al. [19] performed experiments using their grid world and showed results for using only primitive actions and cases using options. Using relational reinforcement learning with primitive actions resulted in a policy that could perform the task about 70% of the time but required a very large number of actions. Using relational reinforcement learning with relational options, the policy was able to perform the task successfully more than 90% of the time and achieved an order of magnitude reduction in primitive actions.

Fern et al. [11] describe learning large relational MDPs by generating multiple trajectories using the current policy structure from a MDP simulator as samples for a supervised learning task. The improved policy structure is induced from the trajectory state-action pairs. They use approximate policy iteration to learn in policy space instead of the more common value-function methods. The downside of this approach is the required simulator for sampling from any state at any time step.

The blocks world has been a favorite testing environment for relational reinforcement learning [18, 21, 22, 28, 65, 74, 91, 96, 109]. A recent and perhaps best performing algorithm of this group is Sarjant's CERRLA algorithm [28, 109]. The CERRLA algorithm produces policies directly, using the Cross-Entropy Method (CEM) to control policy creation, from a set of learned relational condition-action rules. The Cross-Entropy Method assigns a sampling probability to each condition-action rule. These probabilities are gradually modified by CEM to produce policies (randomly sampled) with improved rules. An inferred partial model of the environment is used to help in rule creation resulting in human comprehensible policies. Sarjant evaluated

CERRLA on four environments, the blocks world and three computer games. His results show that CERRLA is able to learn, in less time, equal or better behavior compared to existing relational reinforcement algorithms. For the blocks world case, when trained on environments with differing number of blocks, CERRLA does not consistently find an optimal policy for the "stack block A on block B" task, but still maintains a relatively high performance. In Chapter 5, we compare our algorithm to CERRLA and other relational reinforcement learning algorithms.

The blocks world was also tackled using a propositional representation by Irodova and Sloan [112] and Langlois and Sloan [110]. These use essentially the same approach so we describe the latter, more recent research. In their approach, they represent each state by a fixed number of features:

1. Number of blocks.
2. Number of stacks.
3. Number of blocks in the highest stack.
4. Number of highest stacks.
5. Number of blocks in the second highest stack.
6. Number of second highest stacks.
7. Number of stacks of height one.
8. Number of blocks above block A.
9. Number of blocks above block B.
10. Number of blocks in stack with block A.
11. Number of blocks in stack with block B.
12. Is the top of block A clear? (Boolean).

13. Is the top of block B clear? (Boolean).

14. Is block A on the floor? (Boolean).

15. Is block B on the floor? (Boolean).

16. Are blocks A and B in the same stack? (Boolean).

They use a novel method for actions (instead of the move(X,Y) type actions) which allows them to have a constant number of actions independent of the number of blocks. These actions are connected to the set of features. They define symbolic names to form actions which are:

1. RestOne – stack with one block that is not a or b.
2. RestMiddle – any stack of height different from tallest, and greater than one that does not contain blocks a or b.
3. Blocka – stack that contains block a.
4. Blockb – stack that contains block b.
5. Tallest – tallest stack, may contain blocks a and/or b.
6. Floor

There are 27 actions of the form $SymbolName_i$-to-$SymbolName_j$. For example, the Tallest-to-Floor action moves the top block of the tallest stack to the floor. They learn a Q-function for each of the 27 actions where each Q-function is a linear combination of the above 16 features. For training, they ran 60,000 episodes, 9000 for 3 block cases, 15,000 for 4 block cases, and 36,000 for 5 block cases. They then tested on cases with 3 to 10 blocks. For the on(A, B) task, the performance was good, reaching the goal in about 93% of the test trials.

Chapter 4. Learning Deictic Descriptor Task Policies

4.1. Introduction

This chapter begins discussion of our deictic descriptor task policy learning algorithm where the robot perceives the environment configurations as Markov states and deictic descriptors. After learning the task policy, the robot needs only the deictic description to perform the task.

Our objective is to create an algorithm which learns a universal deictic task policy applicable to any environmental drawn from an environment domain. Can we use a modified/extended state based algorithm for our deictic policy learning? Let's briefly discuss this question. For a state and a task, we can calculate a value for being in the state using a value based learning algorithm like Q-learning. Then, the policy can be derived from the values. Not so for the deictic descriptors. The deictic representation suffers from perceptual aliasing [15] and thus when more than one state maps to the same descriptor, the descriptor value function is not defined. We could try to augment our descriptors to make them have a one-to-one mapping with states. Like chemical elements, we could have descriptor isotopes, each isotope mapping to a state. Then, after learning the state values, the assigned action for a descriptor would somehow be chosen using the values of the descriptor isotope collection. At a high level, this value-based approach is the direction researched in the past with unfavorable results. So, we take the "path less traveled" and learn the deictic policy directly, without learning an intermediate value function.

The algorithm we describe is similar to other RL algorithms in that the robot learns by exploring the environment. How this exploration is done is part of the learning algorithm. For

example, a popular exploration policy for Q-learning is ε-greedy in which ε fraction of the time a random action is chosen and at all other times, an action is chosen using the current Q-function derived policy. Many other exploration methods have been used with Q-learning such as upper confidence bound (UCB) and soft-max action selection [6]. In contrast, the R-max learning algorithm always follows its currently calculated optimal policy [8]. For our algorithm, like R-max, we also always use the current policy for choosing the next action. However, for us this strategy has a downside. The incomplete/imperfect policy may lead the robot in "circles", repeating the same action or sequence of actions over and over again. For example, suppose for the current situation the policy says rotate left a half turn. After performing the rotation, for the new situation the policy says rotate right a half turn. After this the robot is back where it started and ends up rotating left and then right forever. For the R-max algorithm, this kind of phenomena was reported by Grześ and Hoey [80]. We have also observed this in tests with R-max but in our experience, the problem occurs infrequently. A simple solution is to add a limit to the number of actions allowed in an episode. Then, when a repeated sequence of actions is encountered, the episode ends when the action count reaches the limit. As an example, the CANDID algorithm [37] uses this method. Unfortunately, for our deictic policy learning algorithm, looping behavior is much more frequent. For example, when learning the name-tags task, more than 82% of episodes end because of a repeated sequence situation. In this case, using the simple solution adds many unneeded robot actions to the learning process. Obviously, a better solution is desired. So, to detect the repeated sequence situation at the earliest possible time, we allow the robot to observe not only the deictic descriptors but also the Markovian states. The algorithm keeps a path of full state information and as soon as a

repeat is discovered, the algorithm realizes it must repair the flawed deictic policy. Note that after learning the deictic descriptor policy, the states are no longer needed and performing the task may proceed without the Markovian states.

A second issue encountered when learning deictic descriptor policies is the conflicting actions problem which is related to perceptual aliasing [15]. The conflicting actions problem occurs when the Markovian world has 2 (or more) situations mapping to the same deictic descriptor and each situation requires a different action. Our learning algorithm manages this conflicting action problem by using sub-policies. Sub-policies are alternative policies to the main policy. The algorithm uses special actions we call "mental" actions to select and return from an alternate policy (we limit alternate policy selection to be non-recursive). This feature adds a hierarchical quality to the algorithm but also adds complexity. Although a sub-policy permits multiple actions per deictic descriptor, the algorithm must decide when to switch to and return from the alternate policy. Also, the overall size of the policy is increased.

Adding mental action results in a mental environment. The robot can always view its mental environment but it still must be kept up with. Our algorithm utilizes what we call a Z-state for this purpose. A Z-state is the combination of the Markovian physical state and the mental state. At a minimum, the mental state consists of a current alternate policy index and a call stack of the previous switched-to alternate policies. The mental state may also include attention pointers like the focus pointer. Z-states are used during deictic policy learning. After learning is complete, the Z-states are no longer needed but the mental environment must be maintained during task performance.

4.2. The Basic Deictic Descriptor Task Policy Learning Algorithm

Let's move on to the algorithm. Note that we describe the algorithm with pseudocode somewhat similar to the python programming language and use terms like procedure and routine. As discussed above, the robot perceives the environment's configurations as Markovian states and deictic descriptors. In the following discussion, we use the terms state and descriptor in place of Markovian state and deictic descriptor.

The top-level procedure for our algorithm, named LearnPolicy, has the purpose of learning a deictic policy for a domain task. Roughly speaking a deictic policy P maps descriptors to actions; namely P consists of a set of rules of the form $DeicticDescriptor \rightarrow Action$. A simplified overview schematic of the procedure is shown in Figure 4.1.

The LearnPolicy procedure is given a list of training examples which are supplied by the user. Each training example is an instance environment along with an initial state. The initial state corresponds to an initial configuration of the instance environment. The procedure begins by creating an empty seed policy and initializing the totalCost variable to zero. Then, starting with the first environment/start-state pair, it calls the ExtendPolicy procedure (see overview in Figure 4.2). The ExtendPolicy procedure creates a list of policies which, when used by the robot, perform the domain task successfully for the given instance environment. The LearnPolicy then uses these successful policies as seed policies to the ExtendPolicy procedure for the second training example. These steps are continued for each training example. After the last pair has been processed, each policy in the final list of successful policies has worked in each of the environments. Each of these policies also carries a total cost value which is the sum of the costs debited the robot during task execution in each of the environments. The

LearnPolicy returns the policy (or policies) with the minimum total cost. Note that if at any step, none of the seed policies produce success lists, the LearnPolicy procedure returns an empty list indicating failure.

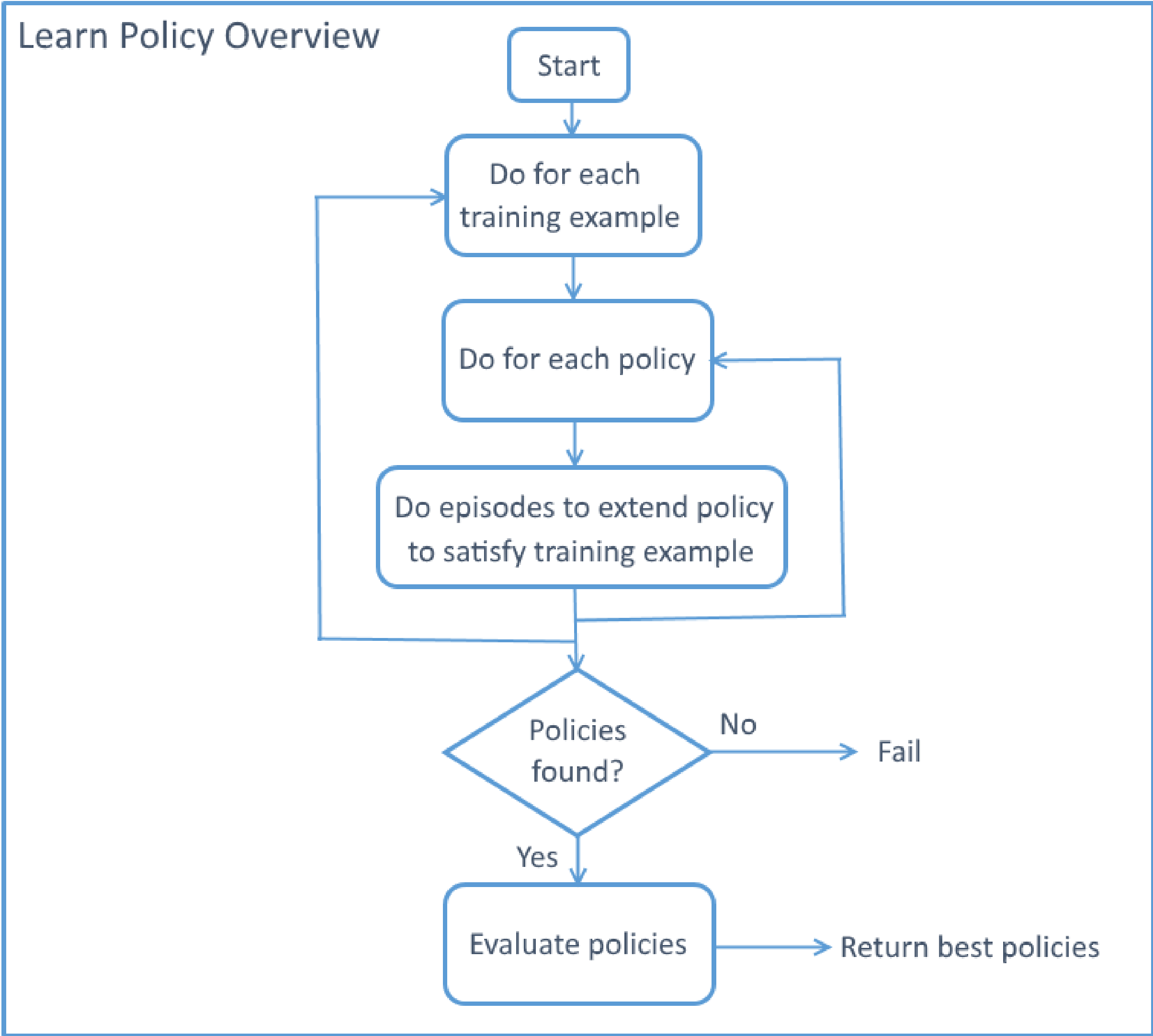

Figure 4.1. A simplified overview of the LearnPolicy procedure.

Now, given the above algorithm overview, let's delve deeper with a description in pseudocode. The top-level/main procedure of the algorithm processes the list of training examples in a sequential manner. The pseudocode for this top-level procedure, LearnPolicy, is shown in Figure 4.3. The LearnPolicy procedure requires three arguments:

- Env: The environment object used for initialization, perceiving states and descriptors, performing actions, receiving rewards, and communicating task completion status. Basically, this object is how the learning algorithm interacts with the world.
- s0List: This is the list of training examples. Each element is an initializer for the instance environment for the domain task.
- maxSize: This an integer representing the limit to the size of a deictic descriptor policy.

The procedure begins by creating a success list (successList0) initialized to a tuple with elements of 0 and policy (lines 2,3,4). The policy is actually an empty deictic descriptor seed policy and the 0 represents the incurred cost of the policy so far.   Line 5 starts the main loop for processing each of the training examples. A new empty success list (sucessList1) is created in line 6 followed by a loop which calls the subroutine ExtendPolicy for each tuple in successList0. The ExtendPolicy subroutine is described in detail below. It returns a list of successful policies and costs for the training example and trial policy.   If this list is not empty, it is added to the successList1. After all the policies in successList0 are tried, it may be the case that no policies were extended. This is a failed situation and is tested in line 12. Once all the training examples have been processed, lines 16-25 determine the best policy (or policies) which is then returned.

The heart of the processing, in its most basic form, is the ExtendPolicy procedure (see pseudocode in Figure 4.4). The purpose of the ExtendPolicy procedure is to find extensions to a

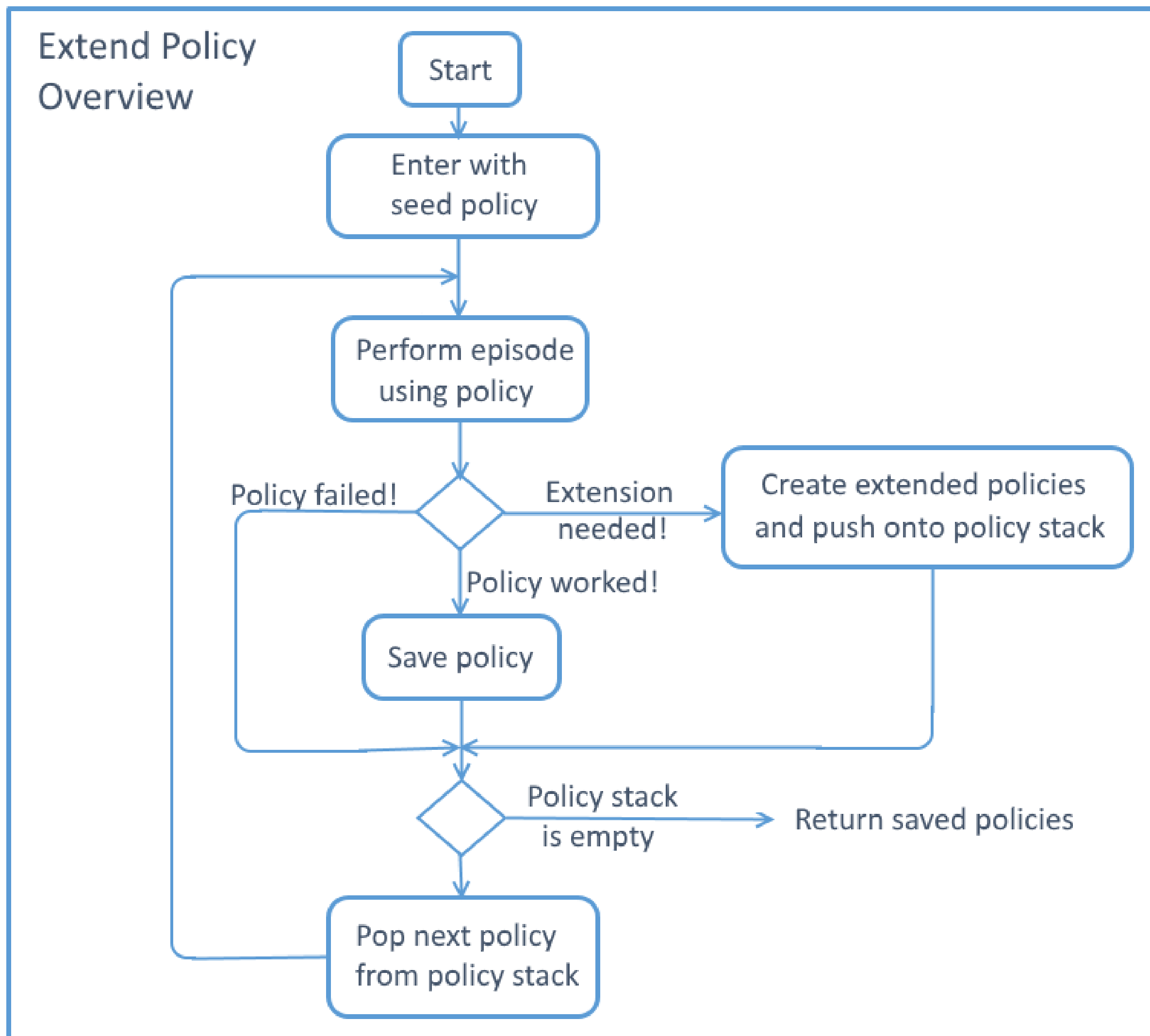

Figure 4.2. A simplified overview of the ExtendPolicy procedure.

seed policy (which may be empty) where the extended policies allow the robot, starting in a given initial situation of a given environment, to complete the domain task. The inputs to this procedure are an environment, a deictic policy, an action set, an initial state, a total cost and a maximum policy size. The environment may be any instance environment from the domain of the domain task. The deictic policy is a seed policy which may be empty. Actions from the action set are candidates for policy extension entries. The initial situation is the state corresponding to an initial configuration of the instance environment. The maxSize is an

integer representing the maximum number of entries allowed in an extended policy. We discuss the totalCost argument later. The ExtendPolicy procedure extends the seed policy by adding new action entries from the given action set to the seed policy, perhaps creating multiple extended policies. The new entry positions are found by exploring the environment. The procedure has 2 sections: the episode section and the policy-extending section. In the episode section (lines 6-28), the robot starts in the initialized environment and repeatedly takes actions using the current deictic policy. Note that the policy is composed of a main policy and zero or more sub-policies which are indexed by the z-state's 'm' element (z.m) where the z-state is the combined mental and physical state (the Markovian state). The episode ends in one of 4 ways. The first is when the task completes successfully and the policy either confirms this (the 'Q' mental action) or has no entry yet for the descriptor (lines 22-27). Either way, the current policy is updated with the mental 'Q' action and added to the success list. In the second case, the task is complete but the policy has a non-quit entry for the descriptor. This indicates a flawed policy and the policy is discarded. The third case occurs when a repeated sequence is detected (line 18). This also indicates a flawed policy and the policy is discarded. In the final case, the robot has reached a point in the environment where the deictic policy has no entry for the current descriptor (lines 14-16). This is a learning situation and the policy needs extending to cover the newly seen situation. This brings us to section 2 of the procedure. Before extending the policy with a new action entry, section 2 (lines 30-33) first checks that the policy size is less than the maxSize parameter. If the policy is deemed extendable, extended versions of the policy are created and added to a policy stack. Each new policy has an applicable candidate action inserted to cover the encountered situation. The next trial policy is then

popped off the policy stack and another episode is start (lines 34 and 35). Once the policy stack is empty, the ExtendPolicy procedure returns the success list to the main training example processing procedure. Note that if the seed policy fails (the first episode tests the seed policy) then an empty list is returned indicating a flawed seed policy. Also, if the seed policy is successful, no extensions are made, and the seed policy is returned as the only element of the success list. Thus, we see that the only case where extended policies are returned is when the robot encounters a learning situation during the first episode using the seed policy.

The ExtendPolicy procedure uses several subroutines to perform its duties. These subroutines are not described in detail but the following briefly outlines their meaning and role:

- Env.Init(s0): This function initializes the environment to the configuration corresponding to the Markovian state s0. This may involve the robot signaling the user and request the initialization. The user would respond by manually arranging the environment to the requested state and then instructing the robot to continue. Obviously, this is undesirable but sometimes unavoidable. In Chapter 6 we improve the algorithm to minimize environment initializations.
- Env.Descriptor: This function returns the deictic descriptor for the current environment configuration. For the robot, learning this function is a prerequisite to learning the deictic descriptor task policy.
- DoAction(Env, z, action): This is the interfacing procedure which directs the robot to perform a mental or physical action. DoAction returns the new z-state, the cost of performing the action, and a flag signifying the task completion status.

Detecting the task completion is feedback from the environment Env which may or may not be automated by the user.

- PolicySize(P): This function returns the number of action entries in policy P.
- CreatePolicyExtensions(P, actionSet, m, d): This procedure creates extended policies from policy P, each with an action entry from the action set for the sub-policy indexed by m (m=0 means the main policy) and the descriptor d. The extended policies are returned in a list.

```
01 Procedure LearnPolicy(Env,s0List,maxSize)
02     successList0 = EmptyList()
03     policy = EmptyPolicy()
04     successList0.append((0,policy))
05     Do For s0 in s0List
06         successList1 = EmptyList()
07         Do For cost,policy in successList0
08             successList = ExtendPolicy(Env,policy,s0,cost,maxSize)
09             if successList is not empty then
10                 successList1.extend(successList)
11         End Do
12         if successList1 is empty then exit Do
13         successList0 = successList1
14     End Do
15     if successList1 is not empty then
16         minCost = System.MaxNumber
17         Do For cost,policy in successList0
18             if cost < minCost then
19                 minCost = cost
20         End Do
21         successList1 = EmptyList()
22         Do For cost,policy in successList0
23             if cost == minCost then
24                 successList1.append(policy)
25         End Do
26     return successList1
```

Figure 4.3. The pseudocode for the LearnPolicy procedure of the basic deictic policy learning algorithm.

```
01 Procedure ExtendPolicy(Env,policy,s0,actionSet,totalCost,maxSize)
02     z0 = CreateZstate(s0)
03     polStack = EmptyStack()
04     successList = EmptyList()
05     Do
06         Env.Init(s0)
07         dix = Env.Descriptor()
08         z = z0
09         cost = 0
10         zHistory = List(z)
11         updatesRequired = False
12         Do
13             action = policy[z.m,dix]
14             if action == None then
15                 updatesRequired = True
16                 exit Do
17             z1,cost1,taskComplete = DoAction(Env,z,action)
18             if z1 in zHistory then exit Do
19             z = z1
20             cost = cost + cost1
21             dix = Env.Descriptor()
22             if taskComplete == True then
23                 action = policy[z.m,dix]
24                 if action and action != 'Q' then exit Do
25                 policy[z.m,dix] = 'Q'
26                 successList.append((totalCost+cost,policy))
27                 exit Do
28             zHistory.append(z)
29         End Do
30         if updatesRequired == True then
31             nEntries = PolicySize(policy)
32             if nEntries < maxSize then
33                 polStack.push(CreatePolicyExtensions(policy,actionSet,z.m,dix))
34         if polStack is empty then exit Do
35         policy = polStack.pop()
36     End Do
37     return successList
```

Figure 4.4. The pseudocode for the ExtendPolicy procedure of the basic deictic policy learning algorithm.

As mentioned above, the user is responsible for choosing domain training instance environments and initial conditions for learning the domain task. This choice could be random but this may not be the best strategy. For example, suppose our user wants to show off her robot's learning ability by having it learn the blocks world on(A,B) task for instance environments of up to 100 blocks starting with any arrangement of these blocks. For training purposes, should the user start with an environment of 100 blocks, arrange them randomly in stacks, and instruct the robot "Learn the task"? Remember, the robot doesn't even know what

the goal of the task is and even a 10 block environment has over 58 million starting configurations. Humans, not knowing what the goal is, would find the 100 block environment difficult. For our algorithm, just like humans, starting with small environments and then graduating to larger environments results in a quicker, less costly learning experience for the robot. As we discuss later in the blocks world example, our first training example has the A and B blocks on the table. The second is block B stacked on A. We continue with a few 4 block examples, a couple of 5 block, a 6 block, a 7 block, and end with a 10 block example. After processing this list of training examples, the robot has learned how to do the on(A,B) task for any number of blocks. If we assume it takes a couple of seconds to move a block to a new location, 30 seconds to set up a configuration, and we use the best version of our algorithm (see Chapter 6), the robot learns the task in less than an hour. In contrast, using the basic algorithm discussed above would result in a 4 day learning time. Reversing the order of the environments for training causes the best algorithm to take over 4 hours and the basic algorithm 18 days.

In the next chapter, we use a set of 7 worlds/tasks to test learning algorithms. We describe each world environment and its sizing dimension or dimensions along with one or more tasks applicable to the environment. The details of starting configurations, determining the task completion configuration or configurations, action list, and how actions produce valid and invalid transitions are included in the description.

Chapter 5. Testing the Deictic Policy Learning Algorithm

5.1. Metrics and Assumptions for Task Learning

As discussed in previous chapters, the robot's objective during task learning is to create a task policy. To do this, the robot utilizes a learning algorithm. Since some learning algorithms are better in some sense than other algorithms, we outline how we evaluate learning algorithms when these algorithms are used for robots. In all cases, we assume the robot is learning online as opposed to learning, for example, in a simulation and then transferring the policy to the robot. The emphasis here is the robot's physical actions take time (on the order of seconds) to perform and time is valuable. Also, we assume the user/trainer may not be a roboticist but does understand how to use tools for teaching the robot such things as object recognition and feature categories and how to set up training scenarios. So, we assume for robot task training, the user's time is valuable and setting up training trials takes time.

Training a robot to perform a task has some pre-task-training costs and some during-task-training costs. For now, we ignore the pre-task-training costs and consider only the training costs incurred while actually training to perform the task. Using the maxim "time is money", the two largest time consuming components during task training are the number of robot physical actions taken and setting up (or resetting) the environment configuration for episodes. For example, consider a blocks world environment of ten blocks arranged in an initial configuration, C_0, of stacks on the table. The user wants the robot to learn the task of arranging the blocks until block A is on block B. The robot learns the task by trial and error where each trial is setup by the user to the initial configuration, C_0. This process can take a lot of trials and each trial may end in a different configuration making it more difficult for the user.

We could make an accurate timing calculation that included times for each action and times to reset to the initial configuration from any other configuration but, we won't. We opt to use a simple metric using a single time, Ta, for each action and a single time, Te, for setting up an episode (trial). Our simple metric is the sum of the number of actions taken by the robot during all trials times Ta plus the number of trials times Te. For the example, let's assume each robot action of moving a block takes about 4 seconds to complete and the user spends about 30 seconds to reset the block configuration to C_0. If training takes a total of 6000 actions and 280 episodes then our simple metric has a value of 6000*4 + 280*30 --> 9 hours. Wherever possible, we use this metric to compare our task learning algorithms to other task learning algorithms.

Now, we spend the rest of this chapter looking at different environments and tasks:

- Name tags world: A robot learns to alphabetize name tags on a table.
- Blocks world: A robot learns to arrange blocks so that block A is on block B.
- Whiteboard world: A robot learns how to add binary numbers.
- Warehouse world: A robot learns how to find a green box and pick it up.
- Grid world: A robot learns to navigate an unfamiliar sequence of rooms.
- Warehouse world: A robot learns to move a size-ordered stack of boxes.
- Manufacturing world: A robot learns about inspecting hidden box contents.

For each of these, we describe the learning difficulty, what robot actions are used, the propositional and deictic representation, and the training examples. Then we compare our basic deictic task policy learning algorithm to the Q-learning algorithm, the R-max algorithm,

and any available prior works. In Chapter 6, we substantially improve the basic algorithm and present a summary comparison with results of this chapter.

5.2. Name Tags World: Alphabetize Name Tags

For this environment and example task, imagine a table with a number of distinct name tags on it. A small robot on the table starts at the left-most name tag and its task is to arrange the tags in alphabetical order, ending up at the right-most name tag. The robot has actions for moving left/right along the table. Using its left or right hand, it may pick up the name tag it is facing and it may put down a name tag in an empty position it is facing. At the user's discretion, the robot may have a left or right hand preference (which hand it uses first when both hands are empty). Each action has an associated cost of 1 unit of energy. In some situations, an action is illegal and attempting the action results in no change to the environment. An example is the robot attempting to pick up a name tag when the robot is facing an empty tag position. When the robot is positioned at the right-most tag with the name tags in alphabetical order, the robot receives a status from the environment signaling the task is complete.

5.2.1. Environment Domain

We want the robot to learn to perform this task for any number of name tags and for any names on the tags. So, the environment's size varies with the number of name tags.

Making the example specific, let's suppose we have six name tags on the table with the robot facing the left-most tag as shown the top of Figure 5.1. For 6 tags, there are 720 of these initial/starting configurations. The complete physical configuration of the robot/environment is

described by the position of the robot and the position of each name tag on the table or in the left or right hand of the robot.

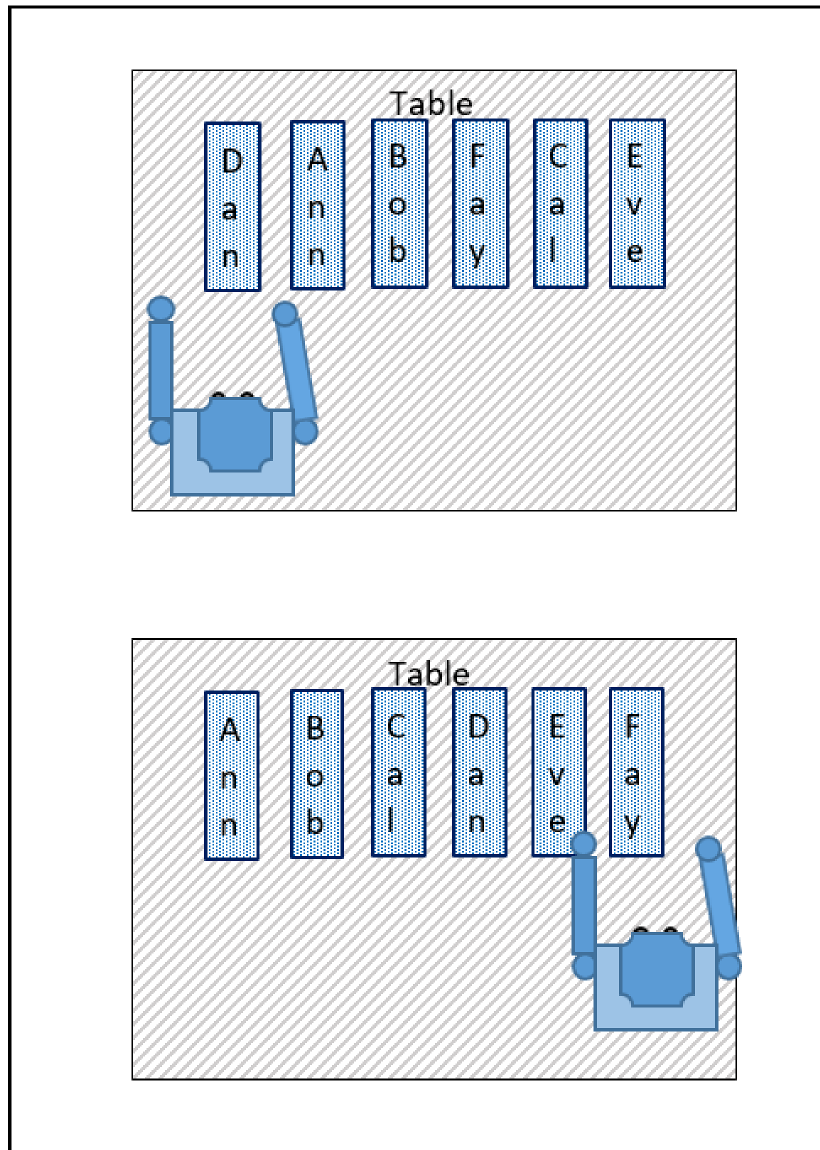

Figure 5.1. A name tag task example. The top is the initial configuration. The bottom is the final configuration with the name tags alphabetized.

Note that the number of distinct legal configurations (robot and name tags) is surprisingly large at 120,960. Of these 120,960 possible configurations, only 1 is the task complete configuration (bottom of Figure 5.1). After task learning, the robot can start in any initial configuration and using its learned policy, complete the task without performing any illegal

actions. Note, the number of environmental configurations (robot and tags) increases rather quickly with the number of name tags (Table 5.1) making this a very challenging learning task for the robot. For the 6 name tags task, a robot acting randomly takes about 300 million total actions on average (about 420,000 per initial configuration) to complete the task for all the starting configurations. This would take the robot over 9 years if it could perform 1 action per second.

Table 5.1. The number of name tag initial configurations, environment configurations (includes the robot), and task goal configurations for alphabetizing name tags task.

No. of name tags	No. of initial configurations	No. of environment configurations	No. of task complete configurations
2	2	24	1
3	6	180	1
4	24	1 440	1
5	120	12 600	1
6	720	120 960	1
8	40 320	14 515 200	1
10	3 628 800	2 395 008 000	1
15	1 307 674 368 000	2 667 655 710 720 000	1

5.2.2. Actions

To transition from a starting configuration to the completed task configuration, the robot has 5 actions:

- R: the robot moves right to the next name tag position. The robot cannot move past the right-most name tag and attempting this results in no movement.
- L: the robot moves left to the previous name tag position. The robot cannot move past the left-most name tag and attempting this results in no movement.

- G: the robot picks up the name tag it is facing with its left hand or right hand if it has a name tag in its left hand (left-handed robot). If the robot is facing a position with no name tag, nothing changes in the configuration.
- PL: the robot places the name tag in its left hand on the table. The robot must be facing an empty name tag position for this to succeed.
- PR: robot places the name tag in its right hand on the table. The robot must be facing an empty name tag position for this to succeed.

5.2.3. Propositional Representation

The propositional attributes and their values we use for describing the 6 name tag environment configurations are (Table 5.2) the position of the robot, “RobotPos”, with an integer value of 1 to 6, what the robot’s left hand is holding, “LHolding”, with a string value of the name or ‘None’, what the robot’s right hand holding, “RHolding”, and the 6 name tag position, “NtagPi”, content name strings or ‘None’ for an empty position.

Table 5.2. The propositional representation for the initial configuration of 6 name tags as shown in the top of Figure 5.1.

Feature Number	Propostional Feature	Feature Value
1	RobotPos	1
2	LHolding	None
3	RHolding	None
4	NtagP1	Dan
5	NtagP2	Ann
6	NtagP3	Bob
7	NtagP4	Fay
8	NtagP5	Cal
9	NtagP6	Eve

For each learning episode, the robot is placed in a starting configuration and either finishes the task or is stopped after a large number of actions have been taken. This episodic learning is repeated until the robot can perform the task for all 720 initial configurations.

5.2.4. Deictic Representation

For the deictic descriptors, the robot has two different modes of attention, one when it is not holding any name tags and the other when it is holding 1 or 2 name tags. So, when the robot is not holding a name tag, its focus pointer is on the cards in front of it. This makes sense since it is trying to determine if the name tags are in the correct order. But, when it is holding a name tag, it is interested in the relation of that name tag to the one in front of it. To differentiate these modes, the robot has a feature, “Holding”, with a value equal to the number of name tags it is holding, 0, 1, or 2. When its hands are empty, the robot’s focus is on the name tag it is facing and on the name tags on either side of the “facing” name tag. We call these the left tag, the facing tag, and the right tag. The robot wants to know if these name tags are out of order so we use 2 boolean features, “LeftNotOk” and “RightNotOk”. The “LeftNotOk” feature has a value of False unless the left and facing tags are not in the correct order. If robot’s position is at the left-most name tag then there is no “left tag” so “LeftNotOk” is False. The “RightNotOk” feature is like the “LeftNotOk” feature except it compares the right tag with the facing tag. The robot also has a boolean feature for detecting if it is facing the right-most name tag, “PosRight”, where a value of True indicates the robot is facing the right-most name tag. With these 3 features, “LeftNotOk”, “RightNotOk”, and “PosRight”, we have 6 realizable deictic descriptors (two of the eight combinations are not possible when the robot is at the right-most name tag). Note that the features do not depend on the number of name

tags or the actual names on each name tag. This means that these deictic descriptors are independent of the environment domain's specification parameters.

The comparisons change when the robot has picked up a name tag. Now it is focused on the alphabetical relation of the name tag in its hand to the name tag it may be facing. There are 3 cases for comparisons. The first is right after the robot picks up a name tag and is facing an empty name tag position. There is no comparison for this case. The second and third cases are when the robot is facing a name tag. So, to cover these different cases, we have a feature "Compare" with values 'None', 'Ok', and 'NotOk'. The value is 'NotOk' when the name tag in the robot's hand is not in alphabetical order with the name tag it is facing. Note that if the robot is holding 2 name tags, no comparison is made. Again, in this mode, the descriptors do not depend on the number of name tags or the actual names on each name tag. Also note that the robot's focus is related to the physical actions and no mental focus pointer actions are required for this task.

Table 5.3 shows the 10 descriptors which follow our deictic mantra "focus on what is currently important to the task". The robot must be taught to recognize these 10 descriptors before deictic descriptor task policy learning begins.

5.2.5. Training

As discuss in Chapter 4, the deictic descriptor task policy learning algorithm requires a list of training examples. Since the learned deictic descriptor task policy will work on any number of tags, we have many choices of training examples. So, to make learning faster, we give the robot small examples, one with 2 name tags, one with 3 name tags, and the last with 4 name tags. For the 2 name tags, the robot learns how to swap the name tags. The 3 name tags is a

little more difficult with no name tag ending in its starting position. In the 4 name tag case, the order starts out completely reversed. Here are the 3 initial arrangements of name tags:

1. [Bob, Ann]
2. [Bob, Cal, Ann]
3. [Dan, Cal, Bob, Ann]

Table 5.3. The deictic descriptors formed from the 5 features of the alphabetize name tags task. Blanks are "don't care" entries.

Deictic Descriptor	Holding	LeftNotOk	RightNotOk	Compare	PosRight
D1	0	False	False		False
D2	0	False	True		False
D3	0	True	False		False
D4	0	True	True		False
D5	0	False			True
D6	0	True			True
D7	1			Ok	
D8	1			NotOk	
D9	1			None	
D10	2				

The robot, using the above examples, learns the deictic policy shown in Table 5.4. The two right-most columns are the main-policy and a sub-policy. When the robot perceives the D2 descriptor (facing tag and right tag out of order), it uses the mental action 'x1' to switch to the sub-policy and then executes the get action G. The sub-policy remains in effect until the robot perceives the D7 descriptor. It then drops back into the main-policy with the mental quit action, 'q', and executes a left action 'L'. Examining this policy closer, we see that the sub-policy and the latter part of the main-policy form a "swap" sequence of two adjacent name tags. The deictic descriptor D9 is also interesting in that it has different physical actions in the main and sub-policies. In the main policy, the robot puts the name tag in its right hand back

Table 5.4. The deictic descriptor task policy for alphabetizing name tags. The right two columns are the action entries.

Deictic descr	Holding	LeftNotOk	RightNotOk	Compare	PosRight	Main-policy	Sub-policy
D1	0	False	False		False	R	
D2	0	False	True		False	x1	G
D3	0	True	False		False	L	
D4	0	True	True		False		
D5	0	False			True	Q	
D6	0	True			True		
D7	1			Ok		L	q
D8	1			NotOk			G
D9	1			None		PR	R
D10	2						PL

on the table whereas in the sub-policy, the robot moves right. This represents a perceptual aliasing situation where the Markov state actions are different. Also note that two of the descriptors have no action entries. The learning algorithm discovered that these two situations can never occur during task execution using this deictic policy. This is not an unusual consequence since the user is responsible for coming up with the deictic descriptors in the first place and cannot completely foresee how the learning algorithm will use them.

Now, let's use the deictic policy on the 6 name tags environment shown in Figure 5.1. The execution steps are show in Table 5.5 with time step 0 being the initial configuration with the robot facing the left-most name tags (we have abbreviated the name tags and the robot is shown as a left hand and right hand contents). Since the D and A tags are not in order, the robot starts a swap sequence in time step 0. It uses this same swap sequence 4 more times on its march to the right-most name tag. In one case (see time step 21), the robot has to move back to the left to fix the D and C tags. This example makes use of every descriptor which has

an associated action (or actions). Also, the environment does not need to signal the robot that it has completed the task. The robot knows the task is complete when it perceives the D5 descriptor. The reader probably recognizes that the robot has learned the "insertion sort" algorithm. This is the algorithm that people often use to sort bridge hands [108].

Table 5.5. The robot's task execution sequence for alphabetizing the name tags in Figure 5.1. The robot is using the deictic descriptor task policy of Table 5.4.

Time step	Deictic descriptor	No. of mental, physical configurations	Action
0	D2	D A B F C E (_,_)	x1, G
1	D9	_ A B F C E (D,_)	R
2	D8	_ A B F C E (D,_)	G
3	D10	_ _ B F C E (D,A)	PL
4	D7	_ D B F C E (_,A)	q, L
5	D9	_ D B F C E (_,A)	PR
6	D1	A D B F C E (_,_)	R
7	D2	A D B F C E (_,_)	x1, G
8	D9	A _ B F C E (D,_)	R
9	D8	A _ B F C E (D,_)	G
10	D10	A _ _ F C E (D,B)	PL
11	D7	A _ D F C E (_,B)	q, L
12	D9	A _ B F C E (_,B)	PR
13	D1	A B D F C E (_,_)	R
14	D1	A B D F C E (_,_)	R
15	D2	A B D F C E (_,_)	x1, G
16	D9	A B D _ C E (F,_)	R

(table cont'd.)

Time step	Deictic descriptor	No. of mental, physical configurations	Action
17	D8	A B D _ C E (F,_)	G
18	D10	A B D _ _ E (F,C)	PL
19	D7	A B D _ F E (_,C)	q, L
20	D9	A B D _ F E (_,C)	PR
21	D3	A B D C F E (_,_)	L
22	D2	A B D C F E (_,_)	x1, G
23	D9	A B _ C F E (D,_)	R
24	D8	A B _ C F E (D,_)	G
25	D10	A B _ _ F E (D,C)	PL
26	D7	A B _ D F E (_,C)	q, L
27	D9	A B _ D F E (_,C)	PR
28	D1	A B C D F E (_,_)	R
29	D1	A B C D F E (_,_)	R
30	D2	A B C D F E (_,_)	x1, G
31	D9	A B C D _ E (F,_)	R
32	D8	A B C D _ E (F,_)	G
33	D10	A B C D _ _ (F,E)	PL
34	D7	A B C D _ F (_,E)	q, L
35	D9	A B C D _ F (_,E)	PR
36	D1	A B C D E F (_,_)	R
37	D5	A B C D E F (_,_)	Q

5.2.6. Q-learning and R-max Comparisons

For example comparisons, we ran the robot with Q-learning and R-max algorithms on the small 6 name tag environment of Figure 5.1 and using the propositional representation. For

each learning episode, the robot is placed in a starting configuration and either finishes the task or is stopped after a large number of actions have been taken. This episodic learning is repeated until the robot can perform the task for all 720 initial configurations. For the performance metric, we using 1 second per physical action and 10 seconds to set up an episode. For the deictic learning, we did a learning session with the small training set environments of 2 tags, 3 tags, and 4 tags and one with the 6 name tag environment. In all cases, each action costs the robot 1 unit. Also, for Q-learning and R-max, the robot received a reward of 10 units when completing the task. The results (Table 5.6) show the robot learns a task policy quicker with the deictic algorithm than with either Q-learning or R-max. Also, using smaller environment instances for training examples improves the deictic learner's metric.

Table 5.6. The algorithm performance results for the task of alphabetizing 6 name tags.

Task: Alphabetize 6 name tags	Q-learning (6 tags)	R-max (6 tags)	Deictic with small training examples	Deictic (6 tags)
No. of physical actions	9191263	799369	12474	70322
No. of training episodes	32446	750	2094	8230
Training time metric (Days)	110	9.34	0.39	1.77

For the deictic descriptor policy, the robot has learned how to use "insert sort" [108] and can alphabetize any number of name tags with any names on them. In contrast, when using either of the other two policies (Q-learning, R-max), the robot can only alphabetize these 6 name tags with the specific 6 names. But, it is very, very good at doing these specific 6 name tags. In essence, the robot has memorized what to do for any situation. On average, the robot only requires 42% of the time required by the deictic policy to perform the 6 name tag task.

5.3. Blocks World: Stack Block A on Block B

The blocks world environment we use consists of two or more blocks on a table with each block labeled with a distinct name. The blocks are arranged on the table in stacks of one or more blocks and there is room on the table to accommodate the largest number of stacks (all stacks having a single block). An environment configuration captures the block/block and block/table situation with the on(X,Y) relation, independent of where blocks are physically located on the table. For example, the configuration for the blocks/table in Figure 5.2 using relational ground facts is:

{ on(C, Table), on(E, Table), on(A, E), on(G, A), on(B, Table), on(H, B) }

Note that moving a whole stack from one place on the table to another place on the table does not change the environment configuration.

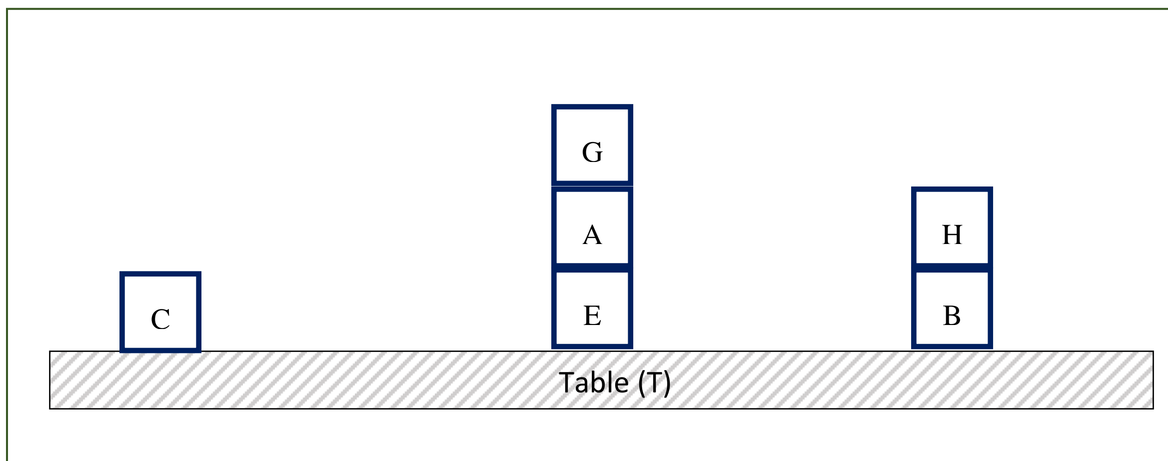

Figure 5.2. A blocks world example configuration for the on(A,B) task. The physical location of the blocks on the table is not captured, only the relationship between the blocks and the table.

In the relational representation setting, a popular blocks world task is the on(A,B) task where blocks are moved around until block A is on block B. For example, Dzeroski, Raedt, and Driessens [18] used relational-RL (Q-RRL and P-RRL) to learn this task and we use their example as our guide in the following description.

The environment configurations they used are as we describe above with one block named A, another block named B and zero or more remaining blocks also having distinct names. The robot consists of a single gripper and has a parameterized high-level action move(X,Y) where X is a variable for a block name and Y is a variable for a block name or the table. For our work, this high-level action is implemented using the robot's underlying control system, vision system, and primitive actions. The robot's vision system knows the physical position of all the blocks and their names and the control system, using this information, plans a list of primitive actions to move block X to the top of block Y. If either of the blocks is not clear, the plan is infeasible and the high-level action fails resulting in no moved block. If Y is the table then only block X must be clear since the table can always accept another block. Note a significant difference between our implementation and Dzeroski's is that our learning algorithm must learn which actions are valid for each block configuration. In Dzeroski's description, the robot is given the actions applicable for each configuration it encounters.

5.3.1. Actions

In a propositional representation of the blocks world, specifying actions for moving blocks from stack to stack or to the table requires some innovation. The relational action, move(X,Y), has the action parameterized with variables X and Y. Since variables are not allowed in the propositional blocks world, some other actions must be used. A straightforward solution is to have N^2 actions of the form MOV.X.Y where N is the number of blocks, X is a block name, Y is a block name or Table, and X <> Y. The underlying control system parses the action to obtain the block names and with its knowledge of block locations, makes the required move (if the move is valid for the current situation). We cannot use this approach since our goal is to learn a single

policy applicable to a blocks world independent of the number of blocks in the environment. So, in our implementation, we have 2 move actions. The first, MOVS, moves a block from one stack to another and the second, MOVT, moves a block from a stack to the table. To implement these actions, we add functionality to the robots underlying vision system. As described above, the robot's vision system sees all the blocks and their names and knows the complete block object layout. We add to this vision system a robot controllable object "pointer" used to focus attention on a single block and the stack it is part of. We call the "pointed to" block the focus block and its stack the focus stack. The robot can move the object pointer with mental actions NEXT and UP where NEXT moves the object pointer to the next stack's bottom block and UP moves the object pointer up a block in that stack. If the robot has moved the pointer to a block, it can issue the PUT mental action which results in a "virtual" sticker being put on that block (only one block can be stickered at a time). This sticker distinguishes the block as the block to be moved. The robot can then move the object pointer to a different stack and issue the action MOVS which commands the gripper to move the "stickered" block to the top of that stack. Of course this action fails if the "stickered" block is not clear. The MOVS action also removes the virtual sticker from the "stickered" block, allowing the virtual sticker to be reused. The robot has an additional action, MOVT, which commands the gripper to move the top block of the focus stack to the table. Thus, with these 3 mental actions and 2 physical actions, the robot can find specific blocks and move them from place to place without the need of parameterized actions. Here is a synopsis of the actions:

- UP: Mentally move the focus pointer to the block above the current focus block. If the current focus block is the top block in the stack then nothing happens.

- NEXT: Mentally move the focus pointer to the next stack's bottom block. This makes this stack the focus stack for physical moves. If there is only one stack then nothing happens.
- PUT: Mentally put the virtual sticker on the focus block. This mental action fails if the virtual sticker is in use on a block.
- MOVS: Physically move the stickered block to the top of the focus stack. This action fails if there is no stickered block, or if the stickered block has a block on it, or if the stickered block is also the top block in the focus stack.
- MOVT: Physically move the top block of the focus stack to the table. This action fails if the focus stack is a single block stack.

5.3.2. Environment Domain

Although having a small action set is nice, it comes at a significant price in terms of the number of environment configurations. The total configuration now includes not just the physical configuration but also the mental configuration of where the focus block is and where (if any) the stickered block is. This method increases the number of environment configurations by (roughly) a factor of N^2 when N is the number of blocks (see Table 5.7). Also, each physical move action is accompanied by some number of mental actions, making the sequence of actions longer and the determination of applicable actions more complicated.

5.3.3. Propositional Representation

There are a number of ways to propositionally represent the mental and physical aspects of our block world configurations. For our scheme, each block has a feature and the object pointer has 2 features. There is also a feature naming the stickered block. The block feature is

“On<BlockName>” where BlockName is replaced with the block name. The “OnX” feature for the block named X has a value equal to the name of the block on block X or “None” if block X is clear. The “Stickered” feature has the value of the name of the stickered block or ‘None’ if no block is stickered. The focus pointer features are “Stack” and “Position”. The “Stack” feature has the value equal to the name of the bottom block of the focus stack. The “Position” feature has the value 1 if the focus block is the bottom block, 2 for the next block up, and so forth.

Table 5.7. The number of configurations for different numbers of blocks.

No. of blocks	No. of physical configurations	No. of mental, physical configurations
3	13	156
4	73	1 460
5	501	15 030
6	4 051	170 142
7	37 633	2 107 448
8	394 353	28 393 416
9	4 596 553	413 689 770
10	58 941 091	6 483 520 010

Figure 5.3 shows an example block world configuration of 8 blocks and Table 5.8 the propositional representation. The action MOVS is valid for this configuration and would move block D onto block G changing block C “OnC”=’None’, and block G “OnG”=’D’. The MOVT action is also valid and would move block G to the table. In our implementation, the NEXT action uses alphabetical order so NEXT will move the focus pointer to the block F stack. The UP action will move the focus pointer to block G. Note that the focus pointer is restricted to point to a block so if it is pointing to the top block in a stack and the MOVT action is executed, the pointer will move down to the new top block in the stack. Note that the focus pointer features are part of

the mental environment and not needed for a robot using the move actions MOV.X.Y as described in Section 5.3.1.

Table 5.8. The propositional representation for the block configuration of 8 blocks as shown in Figure 5.3.

Feature Number	Propostional Feature	Feature Value
1	Stack	E
2	Position	2
3	Stickered	D
4	OnA	G
5	OnB	H
6	OnC	D
7	OnD	None
8	OnE	A
9	OnF	None
10	OnG	None

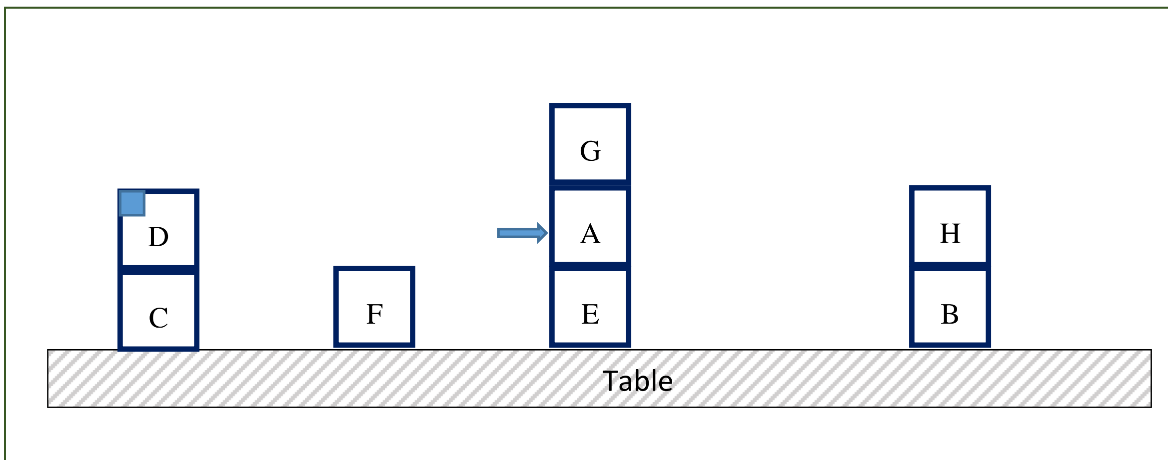

Figure 5.3. A blocks world example configuration for the on(A,B) task. Block D is shown "stickered" and the deictic focus pointer is on block A.

5.3.4. Deictic Representation

The robot uses its focus pointer not only for move and put actions but also for perceiving the environment with deictic descriptors. For this task, the robot is interested in several things: where is block A, where is block B, have I placed a virtual sticker on a block, what's on block A,

what’s on block B, etc. The robot does not care about the names of other blocks, only their relation to blocks A and B like if they are on block A or B. So, it classifies non-A-or-B blocks as “other” blocks, bucketing them into one large group. This is important because it makes the descriptors independent of the number of blocks in the environment. So, we give the robot 2 deictic features associated with the focus pointer and one related to the virtual sticker. The first is “FocusBlock”. This feature has 3 possible value, ‘A’, ‘B’, and ‘O’ where ‘A’ indicates the focus is on block A, ‘B’ indicates the focus is on block B, and ‘O’ indicates the focus is on some other block, not A or B. The second feature is “OnFocusBlk” which in words asks “What is on the focus block”. It has the same 3 possible values as the “FocusBlock” feature plus ‘None’ indicating the top of the focus block is clear. So, for example, if the focus block is block B and block A is directly on top of B then “FocusBlock”=’B’ and “OnFocusBlk”=’A’. The third feature is a boolean feature associated with the virtual sticker, “VSinUse” which is True if a block has been stickered and False otherwise. This feature is also independent of the number of blocks in the environment. The combinations of these features results in 20 (not 24) deictic descriptors since block A cannot be on itself and the same for block B. Table 5.9 shows these 20 deictic descriptors. The 20 descriptors allow the robot to focus its attention on the objects important to completing the task. Also, the robot does not care how many blocks are in the environment when performing the task. It does, however, care about the number of blocks it has to deal with while learning the task. Since, at the beginning of the learning process, the robot does not even know what the task is, it behooves the user to start with a small number of blocks in the environment.

Table 5.9. The deictic descriptor features for the blocks world on(A,B) task.

Deictic Descriptor	FocusBlock	OnFocusBlk	VSinUse
D1	O	A	False
D2	O	B	False
D3	O	O	False
D4	O	None	False
D5	O	A	True
D6	O	B	True
D7	O	O	True
D8	O	None	True
D9	A	B	False
D10	A	O	False
D11	A	None	False
D12	A	B	True
D13	A	O	True
D14	A	None	True
D15	B	A	False
D16	B	O	False
D17	B	None	False
D18	B	A	True
D19	B	O	True
D20	B	None	True

5.3.5. Training

For teaching the robot the on(A,B) task, we start with small environments and then move on to larger ones. The 11 initial configurations are shown in Figure 5.4. The focus pointer starts at the bottom block of the left-most stack in each of the configurations shown. As outlined above, the deictic policy learning algorithm uses both the propositional states and the deictic descriptions.

Let's take the starting configuration 3 in Figure 5.4 as an example. The robot's focus begins on block B. The robot converts its sensors readings and focus pointer information into values for the propositional features and deictic features as shown in Table 5.10 and Table 5.11.

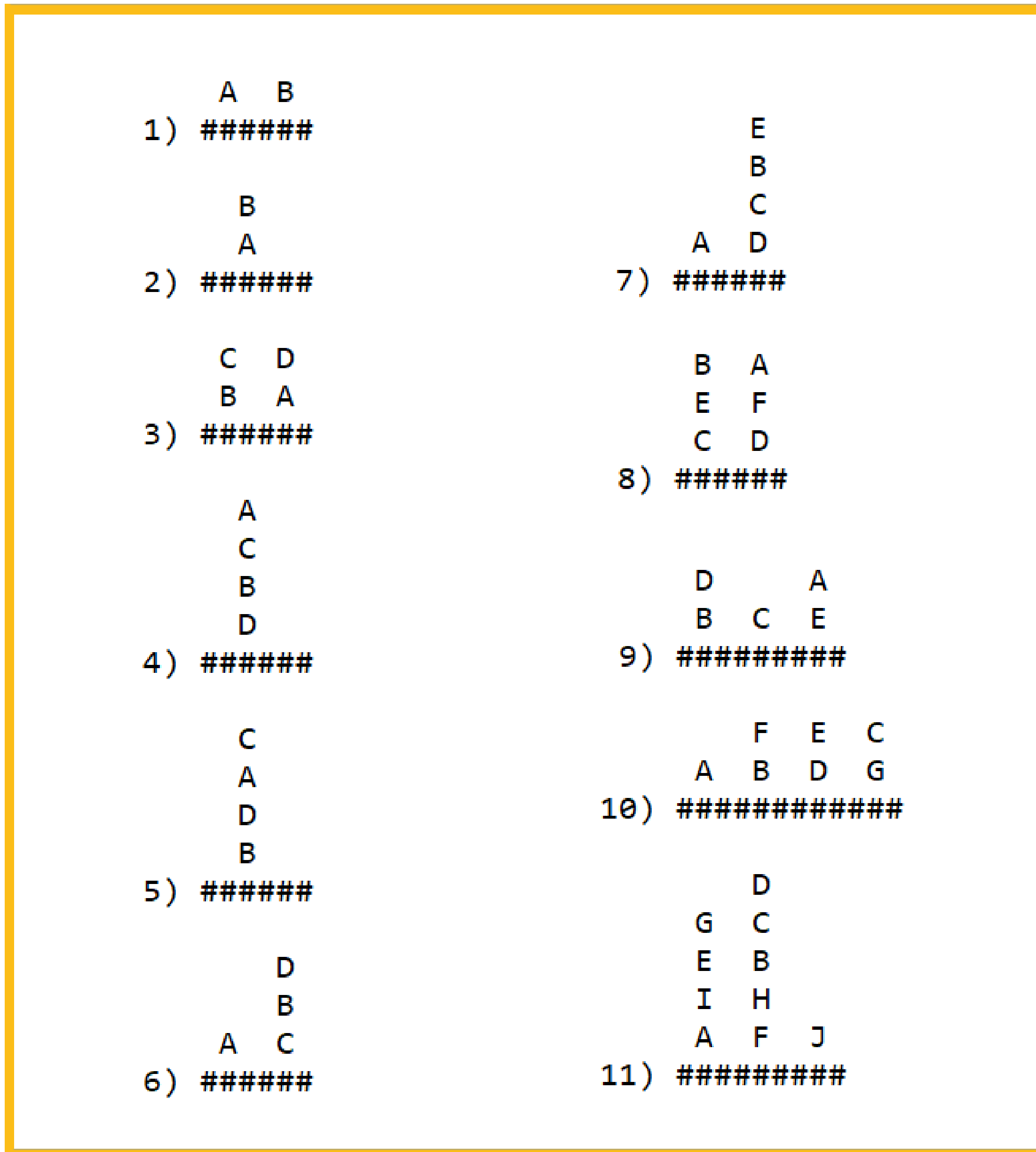

Figure 5.4. The eleven training examples for learning a deictic policy for the on(A,B) task. The hash marks represent the table. Training example 1 is block A and block B on the table. Training example 2 is block B on block A and block A on the table.

The deictic features for this starting configuration correspond to deictic descriptor D16 from Table 5.9.

As the robot explores, each of these is updated and used for deictic policy learning. After the deictic descriptor task policy is learned, it is used for performing the task. The focus pointer

is still used but the rest of the propositional state is not needed. Table 5.12 displays the deictic descriptor policy. The Q-learning and R-max algorithms use the propositional features for both learning and task execution. Again, we emphasize that the features and values used for the deictic descriptors do not change as the number of blocks change which means the deictic task policy is universally applicable to any size blocks environment for the on(A,B) task.

Table 5.10. The propositional features and values for training example #3 (see Figure 5.4).

Feature Number	Propositional Feature	Feature Value
1	Stack	B
2	Position	1
3	Stickered	None
4	OnA	D
5	OnB	C
6	OnC	None
7	OnD	None

Table 5.11. The deictic features and values for training example #3 (see Figure 5.4).

Feature Number	Deictic Feature	Feature Value
1	FocusBlock	B
2	OnFocusBlk	O
3	VSinUse	False

In contrast, the number of propositional features and values change as the number of blocks varies. For Q-learning and R-max, actions must be learned for the new propositional states if the deictic block move mechanism is not used. Another interesting point concerns the propositional versus deictic features with respect to relations. In the name tags world of the previous example, we required the robot to know an alphabetical relation between name tags for some of the deictic features. The robot didn't require any such relational information for the propositional features because the physical layout of the name tags could be used. In this

blocks world with no physical block locations, relations are natural for both propositional and deictic features.

Table 5.12. The deictic descriptor policy for the on(A,B) task.

Deictic Descriptor	FocusBlock	AboveFocus	VSinUse	Action
D1	O	A	False	UP
D2	O	B	False	UP
D3	O	O	False	UP
D4	O	None	False	NEXT
D5	O	A	True	
D6	O	B	True	UP
D7	O	O	True	UP
D8	O	None	True	NEXT
D9	A	B	False	MOVF
D10	A	O	False	MOVF
D11	A	None	False	PUT
D12	A	B	True	
D13	A	O	True	
D14	A	None	True	NEXT
D15	B	A	False	Q
D16	B	O	False	MOVF
D17	B	None	False	NEXT
D18	B	A	True	
D19	B	O	True	MOVF
D20	B	None	True	MOVS

5.3.6. Q-learning and R-max Comparisons

The robot learns the deictic task policy show in Table 5.12 using the 11 training configurations of Figure 5.4. As in the name tags world, some of the deictic descriptors are not used or needed. Using this learned deictic descriptor task policy, we tested the robot on 100,000 randomly generated blocks world environments where we varied the number blocks from 2 to 25 and the number of initial stacks from 1 to 5. The robot reached the on(A,B) task goal in every case. We ran experiments to compare learning times for our deictic policy

learner, Q-learning, and R-max. For the value-based learners, we used the robot's total time required to learn a policy to perform the on(A,B) task for all 5 block and 6 block environment configurations. These results are compared with the time required for the robot to learn a universal deictic descriptor task policy. These experiments all used the virtual sticker mechanism for moving blocks. Since the value-based learners can use the propositionalized relational actions move(X, Y) --> MOV.X.Y, we also included experiments using these actions instead of the virtual sticker actions. Here (Table 5.13) are the results (1 second per physical action, 10 seconds to setup an episode):

Table 5.13. The comparison results for the on(A,B) task learning.

Task = on(A,B)	Q-learning VS moves	R-max VS moves	Deictic	Q-learning MOV.X.Y	R-max MOV.X.Y
No. of physical actions	2019895	512626	22583	128811	157638
No. of episodes	385551	65762	9777	4230	4845
Training metric (Days)	68	13.5	1.39	1.98	2.39

The results show that the deictic learning algorithm outperformed the Q-learning algorithm and the R-max algorithm. Also, the Q-learning algorithm slightly outperformed the R-max algorithm when using the MOV.X.Y actions. In all our other experiments, R-max is normally much better than Q-learning.

5.3.7. Comparisons with Existing Works

The blocks world on(A,B) task has been used for testing in many relational reinforcement learning settings. Sarjant [109] compiled results of many of these algorithms including CERRLA [28], P-RRL [18], RRL-TG [87], RRL-RIB [21], RRL-KBR [74], TRENDI [22], TreeNPPG [91], MARLIE [65], and FOXCS [96]. The results are shown in his table [109] which I reproduce in Table 5.14. The results show that for the on(A,B) task, the relational reinforcement learning algorithms can

learn a general policy for the task anywhere from 90% to 99% of the time. The required number of episodes for the best algorithms ranges from 2000 to 10000. In contrast, our basic deictic algorithm learns a general policy 100% of the time and takes about 10000 episodes. Our improved deictic algorithm (see Chapter 6) learns a general policy for this task in less than 500 episodes.

Table 5.14. RRL algorithm results (from Sarjant [109]).

Algorithm	Average Reward		# of Training Episodes ($\times$1000)	
	stack	*onAB*	*stack*	*onAB*
CERRLA	**1.0**	**0.99**	**1.6**	**10.3**
P-RRL	1.0	0.9	0.045	0.045
RRL-TG	0.88	0.92	0.5	12.5
RRL-TG (P learning)	1.0	0.92	30	30
RRL-RIB	0.98	0.9	0.5	2.5
RRL-KBR	1.0	0.98	0.5	2.5
TRENDI	1.0	0.99	0.5	2.5
TREENPPG	—	0.99	—	2
MARLIE	1.0	0.98	2	2
FOXCS	1.0	0.98	20	50

In Chapter 3, we covered the propositional approach of Langlois and Sloan [110]. Their results were good, reaching the on(A,B) task goal about 93% of the time for 3 to 10 block cases.

5.4. Warehouse World: Pick Up the Green Box

For this example, we use a variation of the blocks world task introduced by Whitehead and Ballard [15]. In their original version, the agent is presented with blocks arranged in stacks on a conveyor belt. One of the blocks is green and the others may be red or blue in any combination. The agent has a gripper with an action for picking up a clear block and an action for putting down a block either on the belt or on a clear block. The task is complete when the gripper picks up the green block. When the agent performs the task successfully, the blocks

disappear and a new configuration of blocks comes down the conveyor belt. But, if the robot takes too many actions before picking up the green block, the blocks fall off the end of the belt and again a new configuration of blocks comes down the conveyor. The agent has the ability to move "focus of attention" markers. There are two types of markers, overt and perceptual. The agent uses an overt marker in conjunction with actions. For example, the agent may use action PUT_AT_M to place the block the gripper is holding to a place pointed to by overt marker M. The perceptual markers are for collecting focused information about the current block configuration. Our world is slightly different with a movable robot instead of a gripper and boxes on pallets in a warehouse instead of blocks on a conveyor belt. We also allow additional colors for the boxes (although we start with green, red, and blue like the original specification). The robot has physical actions for moving to the next stack of boxes or the previous stack of boxes. While facing a stack, it can move its "focus of attention" from the pallet to each box in the stack to above the top box. We call this the robot's focus pointer as we did in the blocks world. The difference here is that moving the focus may be accomplished with physical actions as well as mental actions. The focus pointer moves to a new stack when the robot moves to a new stack so in this case, a physical movement action is also moving the focus. Once facing a pallet of boxes, the focus pointer is moved up the stack using a mental action. The robot has physical actions to get the top box on a stack and to put the box it is holding on a stack or on an empty pallet. Here we add another nuance to the task. We want the robot, when moving boxes, to preferentially put the box on an existing stack, not on an empty pallet. The robot's task is to find the green box and pick it up, perhaps having to move other boxes out of the way.

5.4.1. Environment Domain

For this warehouse world, the environment specification parameters are N and M where N is the number of boxes (which is also the number of pallets) and M is the number of colors. In this world, unlike the blocks world, the physical location of the boxes comes into play. A pallet may be empty or have a stack of 1 or more boxes on it. This greatly increases the number of configurations compared to the blocks world with N distinct blocks. For example, in the blocks world with 5 distinct blocks there are 501 physical configurations. In a warehouse world with 1 green box, 4 other boxes (either blue or red), and 5 pallets, there are 10080 configurations, a 20 fold increase. Adding in the robot's position, focus, and holding a box or not holding a box increases the number of configurations to 160720. Adding colors also increases the number of configurations. In the preceding example, adding yellow as a color choice increases the number of configurations from 10080 to 51030.

5.4.2. Actions

In this warehouse world, the robot's actions are divided into 3 categories. The first is actions for moving from stack to stack. The second category is actions for moving its focus. Picking up and putting down boxes makes up the final category. The following describes each action available to the robot in this box world:

- NXT: The robot moves to the next stack of boxes. The focus moves to the new stack of box and is on the pallet.
- U: The robot moves its focus up. This is a mental action. This action fails if the robot is holding a box.

- G: The robot picks up the top box of the stack of boxes it is facing. The focus tracks the box. The robot can only hold a single box so this action fails if it is holding a box.
- PP: The robot places the box it is holding on an empty pallet. The focus tracks the box. This action fails if the robot is not holding a box.
- PS: the robot places the box it is holding on the top box of the stack of boxes it is facing. The focus tracks the box. This action fails if the robot is not holding a box.

5.4.3. Propositional Representation

The propositional representation for this example is modeled like the physical layout of the stacks on pallets. The number of pallets (and maximum number of stacks), Np, is equal to the number of boxes, N, so we define features, "Palleti" for i = 1...Np, one for each pallet. The value of a pallet feature is a string representing a stack with each symbol in the string representing a box. For example, "BRG" is a stack of 3 boxes and the top box is the green box. An empty string signifies the pallet is empty. The robot position feature, "RobotPos", has the integer value of the pallet the robot is currently facing. The robot can be holding a box which is represented by the feature "HoldingBox". This feature's value is the color of the box the robot is holding or 'None' if the robot is not holding a box. The last feature, "FocusPos", specifies the robot's focus pointer location. The "FocusPos" feature has a value of -1, 0, 1 to M, or M+1 where M is the number of boxes in the stack. The following defines the different meanings:

- FocusPos = -1: Robot is focused on the box it is holding.
- FocusPos = 0: Robot is looking at the pallet.

- FocusPos = 1..M: Robot is focused on box i in the stack where i = 1 is the bottom box.
- FocusPos = M+1: Robot is gazing above the stack of boxes.

Figure 5.5 depicts a warehouse world of 5 boxes with the robot facing a red box on pallet2. From the figure, it is not possible to obtain the robot's focus but let's assume it is on the red box. For this environmental configuration, our propositional representation is shown in Table 5.15.

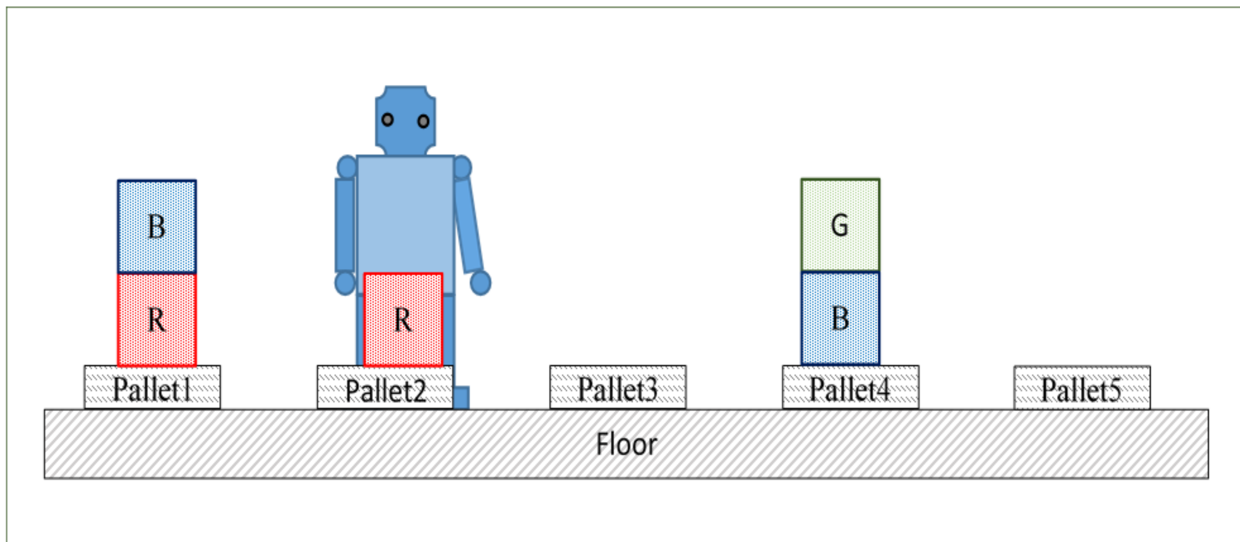

Figure 5.5. A robot world of boxes and pallets for the "pick up the green box" task.

5.4.4. Deictic Representation

Now we turn to the deictic features. For this task, the robot should be looking for the green box, clearing it and then picking it up. We want the learned deictic policy to be universal, independent of the number of boxes and the number of colors. Also, we want the robot to preferentially not put boxes on an empty pallet. So, we give the robot 3 features, "FocusObj", "Holding", and "OneStack". The "FocusObj" feature gives information about the focus and has values of 'Pallet', 'AboveStack', 'Green', and 'notGreen'. The 'Pallet' value indicates the focus is

on the pallet. The 'AboveStack' value denotes the focus is above the stack. The 'Green' and 'notGreen' values signify the focus is on a box with either a green or non-green color. The "Holding" feature has values of 'Green', 'notGreen', or 'None' depending on whether the robot is holding a green or non-green box or not holding a box. The boolean feature "OneStack" is True when there is only one stack. The robot needs to know this because it cannot use the action of moving a box from one stack to another stack unless there is more than a single stack. Table 5.16 displays the 18 deictic descriptors formed from these features.

Table 5.15. The propositional features and values for the configuration show in Figure 5.5.

Feature	Feature name	Feature Value
1	RobotPos	2
2	HoldingBox	None
3	Pallet1	"RB"
4	Pallet2	"R"
5	Pallet3	""
6	Pallet4	"BG"
7	Pallet5	""
8	FocusPos	1

5.4.5. Training

In these "Pick up the green box" task learning experiments, we change the cost of the two put actions so the robot would find it less costly to move a box to another stack than to the floor. For the experiments with the value-based method, we removed the "FocusPos" feature and the action for changing the focus (U). This was done because the mental action and focus are not needed or useful to the learning algorithms and doing so lowers the learning size of the task, improving their performance. Even so, both Q-learning and R-max took a large number of

actions and episodes to learn the 5 box, 3 color environment "Pick up the green box" task. The Q-learning tuning parameters were adjusted (but not optimized) to get better performance.

Table 5.16. The deictic descriptor features for the "pick up the green box" task.

Deictic Descriptor	FocusObj	Holding	OneStack
D1	Pallet	None	False
D2	AboveStack	None	False
D3	notGreen	None	False
D4	Green	None	False
D5	Pallet	notGreen	False
D6	AboveStack	notGreen	False
D7	notGreen	notGreen	False
D8	Green	notGreen	False
D9	Green	Green	False
D10	Pallet	None	True
D11	AboveStack	None	True
D12	notGreen	None	True
D13	Green	None	True
D14	Pallet	notGreen	True
D15	AboveStack	notGreen	True
D16	notGreen	notGreen	True
D17	Green	notGreen	True
D18	Green	Green	True

For our algorithm, we again took advantage of the deictic learner's ability to find a universal policy and used 4 training examples: a 2 box, a 3 box, a 4 box, and a 7 box task. These training examples are displayed in Figure 5.6. The robot, using our deictic descriptor policy learning algorithm, learned the universal policy shown in Table 5.17. The deictic policy is fairly simple. If, at the start, there is only one stack, the robot gets the top box from the stack. If that's the green box, then it has finished. If not, it puts the box on an empty pallet and now there are two stacks. This then starts the normal task execution with more than one stack. Now, the robot moves its focus up the stack it is facing. If it doesn't find the green box, it

moves to the next stack. If it does find the green box in the stack it is facing, it gets the top box from the stack. If it is the green box then the task is complete. If it is not the green box, it moves to the next stack and puts the box on that stack. It then moves to the green box stack and repeats moving non-green boxes to the next stack until it picks up the green box. Then the task is complete.

5.4.6. Q-learning and R-max Comparisons

The learned deictic task policy was tested on all 10080 5 block, 3 color initial configurations and completed all with no problems. Then, 10000 randomly generated configurations of 6 to 15 boxes and 5 colors were tested. The robot finished each of the 10000 tasks flawlessly. The robot used the Q-learning algorithm and the R-max to learn the "pick up the green box" task for the 5 box, 3 color environment. The Q-learning algorithm learned Q values for 280,000 states and the R-max modeled over 45,000 states. The performance for each is shown in Table 5.18. Also, the deictic learner's performance was optimum.

5.4.7. Comparison with Existing Works

In Whitehead and Ballard [15], their agent was tested in random configurations of 4 blocks. They categorized easy configurations where the green block was the only block in a stack to the most difficult where the green block was at the bottom of a stack of 4 blocks. They ran tests of 1000 episodes with the agent failing after 30 physical actions on any episode. They found the agent learned to reliably solve all but the most difficult problems and then only failed 10% of the time. They ran experiments with increased number of blocks and found performance degraded. In contrast, our robot, after learning a deictic descriptor policy, can perform the task flawlessly for any number of boxes.

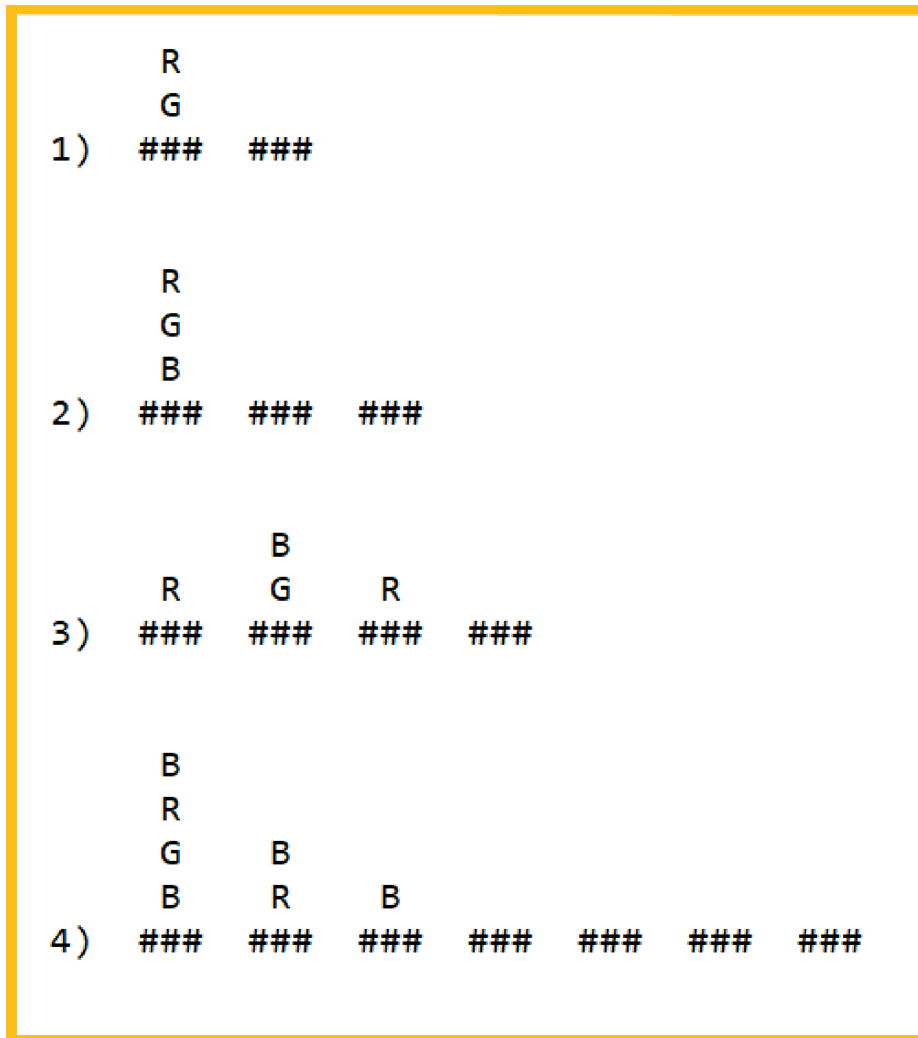

Figure 5.6. The four training examples for the "pick up the green box" task. A group of hash marks represents a pallet. Training example 1 has 2 pallets with a red box on a green box on the first pallet.

Muñoz [37] used his CANDID algorithm (see Chapter 3 for details) on a "find a specific block" task. The CANDID algorithm could find a general policy almost 100% of the time after 12000 episodes compared to our basic algorithms 3000 episodes.

5.5. Whiteboard World: Add Binary Numbers

The whiteboard world consists of a large dry-erase board, an eraser, and various colored dry-erase markers. In its initial state, the whiteboard may be blank or contain symbols. The robot stands in front of the whiteboard with a marker in its left hand. The robot has high-level

actions for writing/erasing symbols on the board, for moving its hands to different positions with respect to the board, and exchanging markers for a different color. The high-level write actions come in 2 types – overwrite and non-overwrite. In overwrite mode, performing a write-symbol in a position containing an existing symbol first has the robot erase the existing symbol and then write the new symbol all as one high-level action. The non-overwrite mode forces the robot to perform 2 actions: an erase action and then a write action of the new symbol.

Table 5.17. The deictic descriptor task policy table for the "pick up the green box" task.

Deictic Descriptor	FocusObj	Holding	OneStack	Action
D1	Pallet	None	False	U
D2	AboveStack	None	False	NXT
D3	notGreen	None	False	U
D4	Green	None	False	G
D5	Pallet	notGreen	False	PS
D6	AboveStack	notGreen	False	
D7	notGreen	notGreen	False	NXT
D8	Green	notGreen	False	
D9	Green	Green	False	Q
D10	Pallet	None	True	G
D11	AboveStack	None	True	
D12	notGreen	None	True	
D13	Green	None	True	
D14	Pallet	notGreen	True	
D15	AboveStack	notGreen	True	
D16	notGreen	notGreen	True	PP
D17	Green	notGreen	True	
D18	Green	Green	True	Q

Table 5.18. The learning metrics for Q-learning, R-max, and deictic learner for the "pick up the green box" task.

Pick up the green box	Q-learning (5 box, 3 color)	R-max (5 box, 3 color)	Deictic (universal)
No. of physical actions	8860357	402673	14939
No. of training episodes	1239840	24319	3004
Training metric (Days)	246	7.48	0.52

Our robot task for the whiteboard world is adding binary numbers. The robot is initially positioned at the whiteboard which contains two binary numbers, one above the other as shown in Figure 5.7. The robot's hand holds a black marker which it can use to write symbols in overwrite mode. Starting at the least significant pair of digits, the robot's task is to write the answer to the addition problem using its actions of moving its hand left or right and to write symbols in the results location and optionally above the addends. The robot's hand is limited to writing symbols in the single line below the addends or the single line above the addends.

An initial environment configuration is the addends as strings of length equal to the length of the resulting sum (leading 0's are added if needed), a blank-filled result string, and the robot's hand at the right-most result position. A legal configuration includes an initial configuration and any configuration where the robot has moved its hand and/or written one or more symbols in the result line or above the addends. Note that the robot may write a symbol anywhere in the result or above the addends by moving its hand left or right. In other words, it is not limited to writing the result in right-to-left order. The task goal is considered reached when the result string is equal to the sum of the addends and the robot's hand is to the left of the sum. Note that any symbols written above the addends are ignored in determining a correct result.

5.5.1. Environment Domain

The environment instances for this whiteboard world are distinguished by a single specification parameter, the number of bits used in the addends. So, for example, a size 5 environment instance would allow addends values up to decimal 31 or binary 11111. Table 5.19 displays the number of initial environmental configurations and the number of possible

legal configurations for various size addends assuming the robot only writes ‘0’, ‘1’, anywhere in the result line and does not write anything above the addends. Also, these configurations include the robots hand position. We see that even for small size environments, the robot has plenty of room to make mistakes.

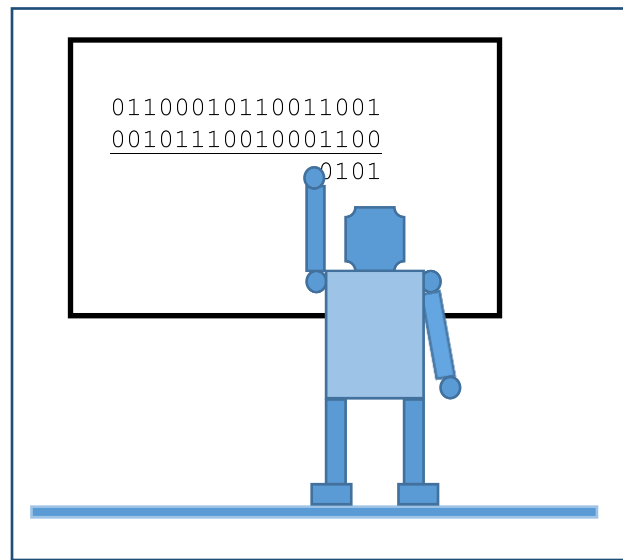

Figure 5.7. A whiteboard environment with a robot adding two binary numbers.

5.5.2. Propositional Representation

The propositional representation of the binary addition environment has 4 features. The first is the robot’s hand position, “HandPos”, with an integer value representing the distance in digits from right most addend digit. The remaining three are the first addend (“Addend1”), second addend (“Addend2), and the sum (“Sum”). Each of the addends has a value which is a string of the digits 0 and 1. The sum feature value is a string (initially blank) of length equal to the length of the answer containing whatever symbols the robot has written so far. The task is considered complete when the addition of the 2 addend values results in the sum value and the robot’s hand is positioned to the left of the left-most results digit. Table 5.20 shows an example of the propositional representation of the 4-bit problem 1010 + 1101 where the

robot's hand is over the third digit position of the sum and the robot has written (somewhat incorrect) symbols.

Table 5.19. The number of initial configurations and legal configurations for different numbers of addend bits for the binary addition task assuming the robot only writes in the result line. The robot's hand position is considered part of the configuration.

No. bits in addends	No. of initial configurations	No. of possible legal configurations
3	64	14 355
4	256	218 475
5	1 024	3 146 139
6	4 096	43 806 843
7	16 384	596 540 475
8	65 536	7 996 088 763
9	262 144	105 916 021 947
10	1 048 576	1 389 945 502 395

Table 5.20. An example propositional representation features and values for the 4-bit binary addition problem 1010 + 1101.

Feature	Feature name	Feature Value
1	HandPos	2
2	Addend1	"01010"
3	Addend2	"01101"
4	Sum	"011 1"

5.5.3. Actions

For the binary addition task, we present two deictic representation approaches based on two action sets. The first action set uses "compound" actions where a compound action is a joined physical action and a mental policy switch action. The second action set allows two physical actions in the same perception cycle. Neither action set includes mental actions for the focus pointer. The focus pointer is always located at the robot's hand position and moves with the hand.

Let's first look at the compound actions approach. In the previous name tags example, the robot, after perceiving a deictic descriptor, performed any mental policy switch action from the current policy, switched policies and retrieved the physical action from the new policy. Here, we turn the order around and perform the physical action before the mental policy switch action. This makes a single deictic descriptor policy entry contain a physical action and a mental action which we specify as a single compound action. Here are the actions for this representation version:

- W0.s, W1.s: The robot writes the symbol '0' (or '1') at the current sum hand position and then switches to the sub-policy (compound action). This action fails if the robot is already in the sub-policy.
- W0.q, W1.q: The robot writes the symbol '0' (or '1') at the current sum hand position and then returns to the main-policy (compound action). This action fails if the robot is not in the sub-policy.
- W0, W1: The robot writes the symbol '0' (or '1') at the current sum hand position. No policy switch is performed.
- L: The robot moves its hand left one digit. The focus moves with the hand. This action fails if the hand is already past the left most digit position.
- R: The robot moves its hand right one digit. The focus moves with the hand. This action fails if the hand is already at the right-most digit position.

Now, let's look at the case where we allow two physical actions per perception cycle. The two actions can occur only when the robot is writing a '0' or '1' digit (not when moving left or right). The second action following a '0' or '1' write is to make a "carry" mark similar to how

school children are taught to add. The actions for moving left and right do not change. Here are the "write" actions:

- W0C, W1C: The robot writes the symbol '0' (or '1') at the current hand position and then makes a "carry" mark above the next addend column to the left. The hand position is returned to the focus pointer position. These all occur in the same perception cycle but cost the robot more than the single action situation.
- W0, W1: The robot writes the symbol '0' (or '1') at the current sum hand position.

5.5.4. Deictic Representation

The mental focus pointer for the binary addition task is initialized to the robot's starting hand position at the right-most digit column. The left, 'L', and right, 'R', physical actions move the hand and the focus pointer left and right.

For the compound action set, the robot perceives the addend digits in the focus pointer column and also whether or not it has written a symbol in the result sum for that column. Note that when the robots hand/focus position is to the left of the result, it perceives blank addend digits and a blank result. So, taking a simple approach, we use only one deictic feature, "AddInfo". The "AddInfo" feature has 6 possible values. If the robot has written a symbol or if there is already a symbol in the result for the column, "AddInfo" has the value 'Written'. Otherwise, the "AddInfo" value is a string containing the 2 addend symbols which is one of "00", "01", "10", "11", or " ". With only a single feature, each possible value translates into a deictic descriptor giving a total of 6 descriptors. For the dual physical action set, we still use the "AddInfo" feature but add a boolean "Carry" feature. The "Carry" feature is True if the column has a carry mark above it. This doubles the number of deictic descriptors to 12.

5.5.5. Training

Learning the binary addition task is not easy. So, as in other tasks, we choose to start with small training examples and move on to larger ones. For the binary addition task, we use the 7 problems: "0+0", "0+1" , "1+0" , "01+01", "001+011", "011+001", and "0011+1011".

The robot, using our deictic descriptor policy learning algorithm and the action set with compound actions, learns the binary addition policy shown in Table 5.21. The policy consists of a main policy and a sub-policy. The main policy performs the additions "0+0 --> 0", "0+1-->1", and "1+0-->1" with non-compound writes. But when it encounters the D4 ("11") descriptor, it uses a compound action W0.s to write a '0' and switch to the sub-policy. While in the sub-policy, the additions are made as if there is a "carry" bit until it perceives the D1 ("00") descriptor. In this situation, it uses the compound action to write a '1' and switch back to the main policy. Note that 3 of its available actions, W1.s, W0.q, and R, are not used. With this deictic descriptor task policy, the robot can add any length binary numbers.

Table 5.21. The deictic descriptor policy for the binary addition task using compound actions.

Deictic Descriptor	AddInfo	Main Policy Action	Sub-Policy Action
D1	"00"	W0	W1.q
D2	"01"	W1	W0
D3	"10"	W1	W0
D4	"11"	W0.s	W1
D5	" "	Q	
D6	Written	L	L

The robot, using our deictic descriptor policy learning algorithm with the dual actions, learns the binary addition task policy shown in table 5.22. This policy uses all the actions except the right, R, action.

Table 5.22. The deictic descriptor policy for the binary addition task using "carry" action.

Deictic Descriptor	AddInfo	Carry	Policy Action
D1	"00"	False	W0
D2	"01"	False	W1
D3	"10"	False	W1
D4	"11"	False	W0C
D5	" "	False	Q
D6	Written	False	L
D7	"00"	True	W1
D8	"01"	True	W0C
D9	"10"	True	W0C
D10	"11"	True	W1C
D11	" "	True	
D12	Written	True	L

5.5.6. Q-learning and R-max Comparisons

The two deictic descriptor task policies learned by the robot are universal policies for the binary addition environment domain. The policy using the compound action set is optimal. The policy using the dual physical action (carry mark) is not optimal but some would consider it satisficing.

The robot using the Q-learning or R-max algorithm perceives the complete state of the environment and therefore must learn the proper sequence of actions for each binary addition problem. Learning one 4-bit addition problem does not help in learning a different 4-bit addition problem (or any other larger bit addition problem). But, by adding leading zeroes, any 1-bit, 2-bit, or 3-bit number can be made to look like a 4-bit number. So, using this technique, we can say the policies are universal up to a 4-bit size.

We trained the robot using the Q-learning and R-max algorithms on the 4-bit environment for which there are 256 starting configurations. Table 5.23 summarizes the results

for the robot's learning performance using our deictic task policy learning algorithm versus the Q-learning and R-max algorithms. As previously indicated, the deictic descriptor task policies allow the robot to add any size numbers. The results indicate the significant benefit of training on smaller numbers. Also, we ran a 2-bit case for the R-max algorithm. The results show that even with this small a problem, the deictic learning algorithm had better performance metrics.

Table 5.23. The comparison results for the algorithms on the binary addition task.

Adding binary numbers	Q-learning (4-bit)	R-max (4-bit)	R-max (2-bit)	Deictic (compound)	Deictic (carry write)
No. of physical actions	4159986	1808754	5686	1222	1727
No. of episodes	125440	47904	255	324	364
Training metric (Days)	62.7	26.5	0.0953	0.0516	0.0621

5.6. Grid World: Learn a Universal Policy for Room Navigation

In Chapter 1, we used a simple grid world environment with 2 rooms. Here, we use more complicated environments from the grid world as described in Croonenborghs et al. [19] for their work using skills (options) for reinforcement transfer learning in a relational representation setting. Their research was covered in Chapter 3 but we expand the description here to blend in with our work.

The robot's task is to learn a universal policy for a finite set of similar grid room environments. By universal policy, we mean a policy that the robot can use successfully on any environment instance drawn from the set of environments. Also, the robot is only allowed to use a small subset of environments for learning – the unseen environments are used for testing the robot's (hopefully) universal policy. The task is considered complete if for every environment E in the set of environments, the robot, starting in E's initial configuration, reaches

the goal configuration using the universal policy.   This is very similar to what we have been doing with our deictic policies.

Each instance in the grid world environment set has a sequence of rooms connected by colored doors. Every room has colored keys on the floor with at least one that matches the color of the exit door. To reach the goal, the robot, starting in the first room, must navigate to a key whose color matches that of the room's exit door. Once the robot is at the proper key, it must pick up the key and then navigate to and through the exit door to the next room. If, when entering a room, the robot already possesses the correct colored key, it can bypass getting another key and proceed directly to the exit door. The goal is reached when the robot exits the last room in the sequence. The environment instances are generated by varying the parameters of a grid world specification. These parameters are the number of rooms, the room sizes, exit door location, entry door location, and the location, color, and number of keys.

5.6.1. Environment Domain

For this problem, we use the specification parameters from the original paper. There, they used rooms with dimensions of between 3 and 5 units and each room could have from 1 to 3 colored keys. The number of rooms in the sequence of rooms varied between 1 and 5. For each room, the exit door can be located anywhere on any wall. The entry door can be located on any wall except the exit door wall. Of course, these choices must be physically realizable with the 2-d layout of the rooms. With this specification parameters, the number of initial configurations can be very large.   For example, consider the smallest room (size 3 by 3) with from 1 to 3 keys, each of a color red, blue, or green. By varying entry and exit door locations, exit door color, key and key locations and colors, 884,520 partial environments can be

specified. For the next largest room, a 3 by 4, the number of partial environments jumps to 2,721,074. Now, if we make environments from a sequence of such rooms, the set of environments becomes very, very large.

5.6.2. Actions

Before discussing the propositional and deictic representation features, let's specify the robot's available physical actions. The action set is chosen to be "deictic friendly" meaning relative to the robot's point of view. So, the robot actions for our version of the grid room environment are different than the up, down, left, and right of the original paper. These original actions are relative to a person viewing the room from above on a computer screen. The robot may think the action 'up' means to jump! Our robot has a facing direction and so the actions are relative to this direction. The robot's physical actions are:

- F: The robot moves one tile forward in the facing direction unless it is facing a wall or an exit door for which it has no key.
- L: The robot rotates left 90 degrees of facing direction. The robot stays on the same tile, only its facing direction changes.
- R: The robot rotates right 90 degrees of facing direction. The robot stays on the same tile, only its facing direction changes.
- P: The robot picks up the key or keys on the tile it occupies.

So, the robot can only move in the direction it is facing and must rotate its body to move in a different direction. The downside of this action set is that it increases the number of configurations. For each tile position, the facing direction of the robot must be included.

5.6.3. Propositional Representation

Now let's cover our propositional features for the rooms and for the robot. For the rooms, there are 3 static features and one dynamic feature (the keys). These features are:

- RoomSize<i>: The ith room's size, L by W. L and W can be 3, 4, or 5.
- ExitLocCol<i>: The ith room's exit door location and color, (x, y, d, c). The x,y coordinates are the location of the tile in front of the door. The direction d is direction robot travels to pass through the door. The color of the door is c.
- EntryLoc<i>: The tile location of the entry door and direction of the robot when it enters this i^{th} room from the exit door of the previous room. This, in effect, links this room to the previous room. There is no entry door for the first room
- Key<i>.<k>: The ith room's k^{th} key (k = 1, 2, or 3) represented by a tuple (x, y, c) where x, y is the grid location of the key and c is its color. If the robot picks up the key, the value becomes (-1, -1, c).

There is a feature associated with the robot. This feature is:

- RobotLocDir: This feature's value is the robot's location and facing direction. The value is a tuple (r, x, y, d) where r is the room number, x, y is the robot's grid coordinate for the room and d is its facing direction. When the robot exits the last room, the value is (0,0,0,0) signifying the robot has exited the last room.

Figure 5.8 shows an example environment with 2 rooms. The robot is in the first room, facing east. There are 3 keys in the room (yellow, green, and red). The exit door is red. To reach the goal, the robot must get the red key and exit room 1 through the red door. Then, in

the second room, get the yellow key and exit through the yellow door. Alternatively, the robot could get the red and yellow keys from the first room and skip picking it up in the second room. Table 5.24 shows the propositional representation of the rooms.

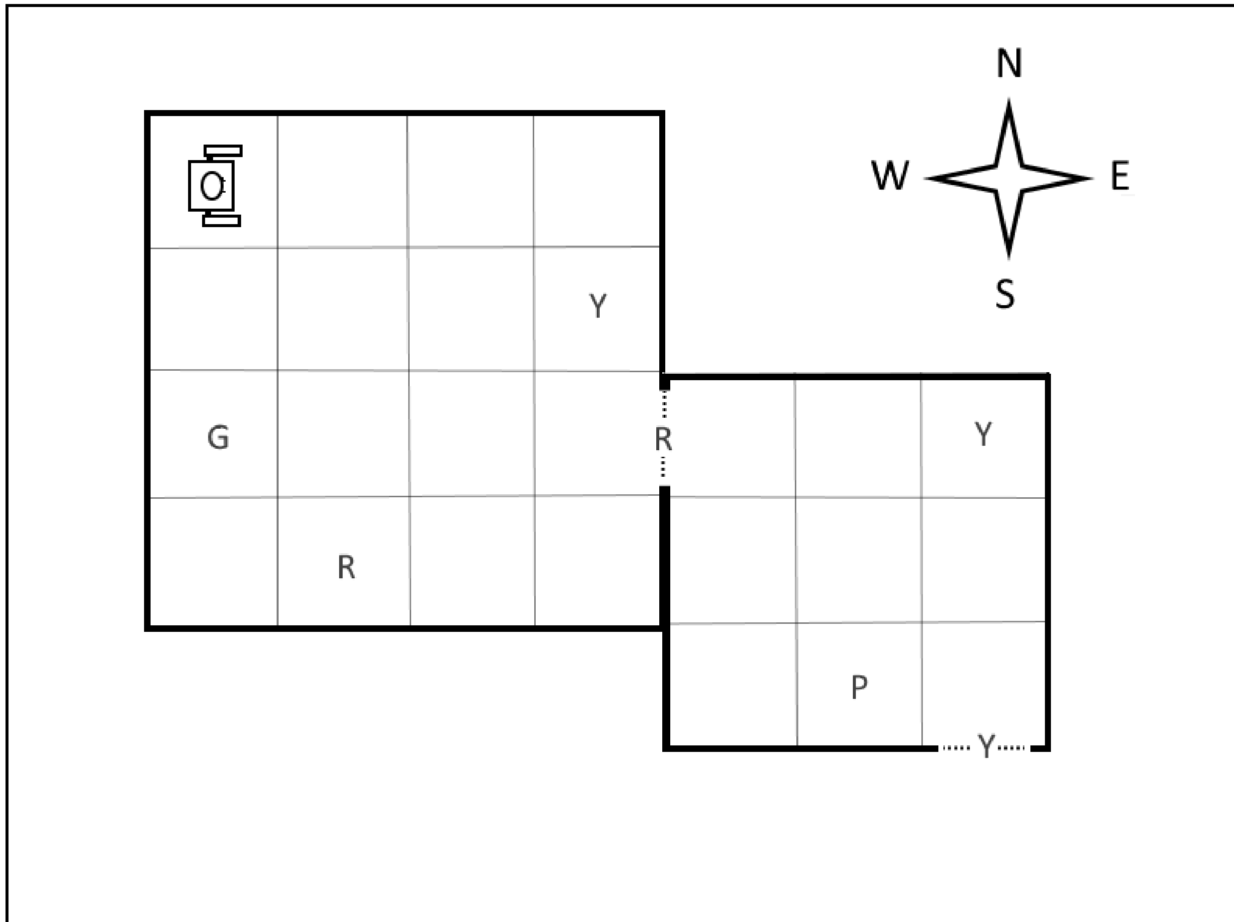

Figure 5.8. An grid world environment for the "navigate rooms with keyed doors" subtask. The robot is in the upper left corner of the west room. The west room contains 3 colored keys (yellow, green, red) and has a red exit door. Only the red key will open the red door. The subtask goal is reached when the robot exits the second room's yellow door.

5.6.4. Deictic Representation

Turning to the deictic descriptors, we consider what is the robot focusing on to reach the environments' goal. The robot is concerned with the exit door and the keys and does not care at all about the size of the room. So, when the robot enters a room, its focus pointer is on the

exit door and one of the keys on the floor (call it the focus pointer key). It has a single mental focus pointer action, NK, which is to move the focus pointer to the next key.

Table 5.24. The propositional representation for the 2 rooms environment (Figure 5.8)

Feature No.	Feature name	Feature Value
1	RobotLocDir	(1,1,1,E)
2	RoomSize1	(4,4)
3	ExitLocCol1	(4,3,E,R)
4	Key1.1	(4,2,Y)
5	Key1.2	(1,3,G)
6	Key1.3	(2,4,R)
7	RoomSize2	(3,3)
8	ExitLocCol2	(3,3,S,Y)
9	EntryLoc2	(1,1,E)
10	Key2.1	(3,1,Y)
11	Key2.2	(2,3,P)

If the robot needs a key, it is concerned with the location of the key in relation to itself. If the robot has the correct key then it is concerned with the location of the exit door in relation to itself. With this in mind, we make the deictic descriptors a function of several features:

- RobotOutside: This boolean feature is True if the robot has exited the last room.
- HoldingMkey: This boolean feature is True if the robot is holding a key matching the exit door color.
- KeyDir: This feature's value is the direction of the key the robot is focusing on relative to the robot's facing direction. The values are 'forward', 'left', 'right', or 'over'. The 'over' value means the robot is over the key.
- KeyMatch: This boolean feature is True if the color of the key that the robot is focusing on matches the color of the exit door.

- ExitDir: This feature's value is the direction of the exit door relative to the robot's facing direction. The values are 'forward', 'left', or 'right'.

The deictic descriptors are a function of these features as shown in Table 5.25.

Table 5.25. The deictic descriptor features for the "navigate rooms with keyed doors" subtask.

Deictic Descriptor	RobotOutside	HoldingMkey	KeyDir	KeyMatch	ExitDir
D1	False	False	forward	False	
D2	False	False	left	False	
D3	False	False	right	False	
D4	False	False	over	False	
D5	False	False	forward	True	
D6	False	False	left	True	
D7	False	False	right	True	
D8	False	False	over	True	
D9	False	True			forward
D10	False	True			left
D11	False	True			right
D12	True				

5.6.5. Training

We used a single environment for training the robot. The environment (see Figure 5.9) has 5 rooms with three of size 3x3 and one each of size 4x4 and 5x5. There a total of 14 colored keys spread around the rooms. The robot is started in the center of the north room facing west. To reach the outside (the goal), it must collect the correct key for each door while navigating through each room. For this example, an optimal path requires 48 actions to exit the navy door in the fifth room. The robot, using this training example with the deictic descriptor task policy learning algorithm, learns the policy shown in Table 5.26.

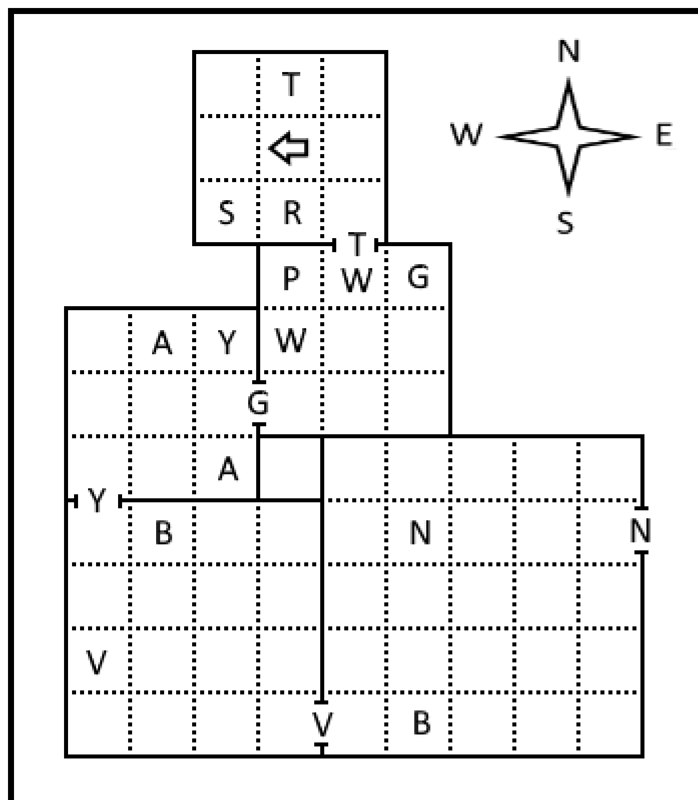

Figure 5.9. Training environment containing 5 rooms connected by keyed doors. The robot, shown as an arrow, is located in the north room facing west. To reach the last room and exit to the outside, the robot must navigate through the rooms, unlocking doors as it goes by collecting the Teal, Gold, Yellow, Violet, and Navy keys on its way.

5.6.6. Q-learning and R-max Comparisons

For learning time comparisons, the same 5 room example (Figure 5.9) was used for training the robot with Q-learning and R-max as was used by the deictic policy learning algorithm. Although the value-based algorithms cannot provide a universal policy, the training times reflect, at least qualitatively, the difficulty of learning in this environment. The Q-learning algorithms was the poorest performer (Table 5.27), taking 124 days of elapsed time to train the robot assuming 1 second per robot physical action and 10 seconds to setup the environment for each episode. The robot training time for R-max was about 14 days and the for deictic about 1.3 days.

Table 5.26. The deictic descriptor task policy table with features and actions for the “navigate rooms with keyed doors” subtask.

Deictic Descriptor	RobotOutside	HoldingMkey	KeyDir	KeyMatch	ExitDir	Action
D1	False	False	forward	False		NK
D2	False	False	left	False		NK
D3	False	False	right	False		NK
D4	False	False	over	False		NK
D5	False	False	forward	True		F
D6	False	False	left	True		L
D7	False	False	right	True		R
D8	False	False	over	True		P
D9	False	True			forward	F
D10	False	True			left	L
D11	False	True			right	R
D12	True					Q

The robot learned an optimal policy for each of the methods but of course, only the deictic policy is universal. In fact, the deictic policy is optimal for all such environments when all key colors are distinct. When there are keys with the same color, there are situations where the deictic policy may not lead the robot optimally. The first situation is when a non-door key in an earlier room may be less costly to collect in the earlier room than the same color key for a door in a later room. The second situation is when there are duplicate door colored keys in a room. The deictic policy leads the robot to the first door key it focuses on which may not be the best key from a navigation perspective.

Table 5.27. The comparison results for the algorithms on the “navigate rooms with keyed doors” subtask.

Navigate rooms with keyed doors	Q-learning	R-max	Deictic
No. of physical actions	10508649	1204321	74945
No. of episodes	16749	2794	3698
Training metric (Days)	124	14.3	1.30

5.6.7. Comparisons with Existing Works

Chapter 3 summarized the approach of Croonenborghs et al. [19], the originators of this learning task. As previously covered, they performed two sets of experiments using relational reinforcement learning. Note, they used the phrase "primitive actions" for non-option actions and sometimes the word "skills" for option type actions. In the following, we will follow their nomenclature for actions.

The first experiments, the primitive action case and the one most comparable to ours, used movement actions of up, down, left, and right for navigating to keys and exit doors. These experiments were done mainly to compare with the options (skills) based tests. We use them for comparisons with our deictic policy learning algorithm. The agent learned a policy using relational reinforcement learning with a training set of instance environments. The agent always perceives the complete description of the environment consisting of the dimensions of the different rooms, the locations and colors of the doors, the location and colors of the keys, the keys the agent possesses, and the agent's location. The resulting learned policy was not universal, allowing the agent to reach the task goal only about 70% of the time. Also, when the agent did reach the goal, the number of actions taken was very large and far from optimal. In contrast, a robot using our algorithm and primitive actions learns a deictic policy which always reaches the task goal and in many cases is optimal.

To improve the performance of the agent, they devised a method for learning skills (options). Using the learned skills "pickup_key" and "find_door", the learned relational policy performed better, solving over 90% of the environments. Also, the number of primitive actions used was greatly reduced over the non-options case. They did not report if the completed tasks

were performed optimally. In any case, the task completion results are closer to our deictic policy results.

5.7. Warehouse World: Move a Size-Ordered Stack of Boxes

In the warehouse world, the robot is stacking/unstacking boxes, moving them from one pallet to another. The robot has actions for moving the top box on the stack it is facing to another stack or an empty pallet. We visited a similar world in the previous “Pick up the green box” task. For this example of the warehouse world, we add constraints to box movements. The environment consists of 3 pallets and a number of different size boxes. The pallets, located in 3 corners of the warehouse, are labeled in clockwise order “Source”, “Temporary”, and “Destination”. The initial configuration of the environment has all the boxes stacked on the “Source” pallet in size order with smallest on top and the robot facing the “Destination” pallet. The robot’s task is to move a box at a time until all the boxes are on the “Destination” pallet with the constraint that no box can ever be placed on a smaller box. The robot may use the “Temporary” pallet as a holding spot if needed. The reader may have recognized this task/environment as the Tower of Hanoi puzzle with boxes and pallets replacing the rings and poles.

5.7.1. Actions

The move actions for the environment are at a higher-level than the previous “Pick up the green box” task. In this warehouse world implementation, the robot has macro actions for moving a box from one stack to another. For example, let’s suppose the robot is facing the “Destination” pallet and it wants to move the top box of the “Source” pallet to the empty “Temporary” pallet. It moves to the “Source” pallet, gets the top box, moves to the

“Temporary” pallet and puts down the box. The robot ends up facing the “Temporary” pallet. These 4 “less higher-level” actions form a macro action we could name MST for move “Source” to “Temporary”. This is what we do except we take the robot’s point of view. Using the robot’s viewpoint, the robot calls the stacks “the stack I am facing”, “the stack to my left (counter-clockwise)”, and “the stack to my right (clockwise)”. As in the example, the robot has actions for moving a box from one stack to another stack (or empty pallet) and after the move, the robot ends up facing the stack it moved the box to. There are 6 move actions:

- MLR: The robot moves the top box from the left stack onto the right stack or empty pallet. The right stack is now the facing stack.
- MRL: The robot moves the top box from the right stack onto the left stack or empty pallet. The left stack is now the facing stack.
- MLF: The robot moves the top box from the left stack onto the facing stack or empty pallet. The facing stack remains the facing stack.
- MRF: The robot moves the top box from the right stack onto the facing stack or empty pallet. The facing stack remains the facing stack.
- MFL: The robot moves the top box from the facing stack onto the left stack or empty pallet. The left stack is now the facing stack.
- MFR: The robot moves the top box from the facing stack onto the right stack or empty pallet. The right stack is now the facing stack.

Note that for all move actions, attempting to move a box onto a smaller box fails and neither the box nor the robot moves. Also, a careful examination of cases reveals that the last two move actions, MFL and MFR, are not strictly needed to perform the task. The first case is the

initial situation where the robot is facing the empty “Destination” pallet. Obviously, with an empty pallet it cannot use MFL or MFR. The robot’s first action must either move the top “Source” box to the “Temporary” pallet or to the “Destination” pallet using MRL or MRF respectively. In either situation, the robot ends up facing the box it has just moved. In fact, the robot always ends up facing the box it has just moved. So, using a MFL or MFR, the box it is facing is either 1) moved back to where it came from, cancelling the previous move or 2) moved to a stack which could have been reached using a single move instead of a double move. So why include these moves? They can be useful for undoing “mistakes”. The robot sometimes gets itself into a situation where it has no legal move except a MFL or MFR. For example, take the 2 box case, a large box and a small box. If the robot performs the sequence MRF, MRL, MRF, it ends up facing the temporary stack with the small box on the large box. Now the robot has no legal action except a MFL or MFR.

5.7.2. Environment Domain

For N boxes, there are 3^N legal ways to stack them on the 3 pallets. When we include the robot’s three possible positions, the number of legal environment configurations increases to 3^{N+1}. But because of the box order constraint and the fact that the robot always faces the last box moved, the number of “reachable” configurations is smaller. Figure 5.10 is an example of this. Here, the number of boxes is N = 3. The starting configuration (not shown) had the 3 boxes on the source stack, smallest on top. Presumably, the robot has moved boxes and is now facing the largest box on the temporary stack. Since the robot is facing the largest box, the robot’s last action moved this largest box from either the source or destination stack to the temporary stack. But the largest box could not have been on either the source or destination

stack – this violates the size ordering constraint of no box may be placed on a smaller box. Therefore, this configuration could not have been reached by the robot. The number of reachable configurations is $2(3^N-1)$. Table 5.28 shows the number of legal and reachable configurations for N = 3 to 10. We see from the table that the number of configurations is not large. The reason we include this task in our examples is because the minimum number of actions to perform the task is large at 2^N-1.

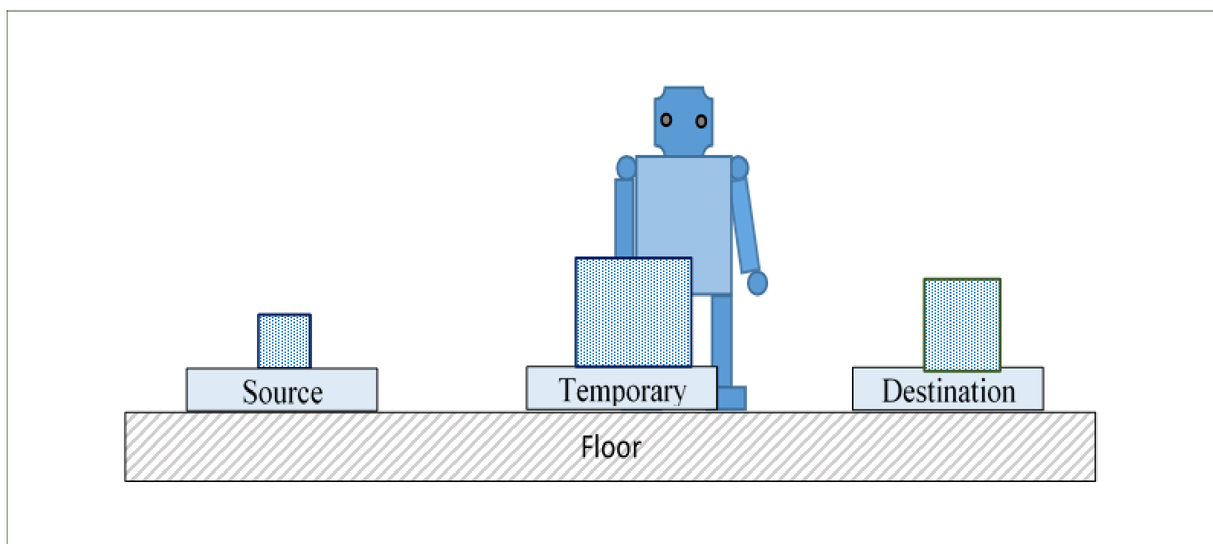

Figure 5.10. An unreachable configuration of boxes.

5.7.3. Propositional Representation

For the propositional representation, we use similar features as we used in the "Pick up the green box" task. Here, the boxes are named numerically by size like 1 through 5 for five boxes. The environment configurations are specified with 4 features. The first is the robot position "RobotPos", the pallet the robot is facing. This feature has one of 3 possible values S, T, or D meaning "Source", "Temporary", and "Destination". The remaining 3 features describe the pallet stacks. These features are "PalletS", "PalletT", and "PalletD". The value of each is a list of box names representing the boxes stacked on that pallet. The first box name in the list is

the largest box. An example configuration is shown in Table 5.29 where we have 5 boxes with 3 boxes, 5, 4, 3 on the "Source" pallet, 2 boxes, 2 and 1, on the "Temporary" pallet, and the "Destination" pallet is empty. The robot is facing the "Temporary" stack.

Table 5.28. The number of legal and reachable configurations for different numbers of boxes. The optimal (minimal) number of actions to move the stack is shown in the last column.

No. of boxes	No. of legal configurations	No. of reachable configurations	Minimum number of actions required
3	81	52	7
4	243	160	15
5	729	484	31
6	2 187	1 456	63
7	6 561	4 372	127
8	19 683	13 120	255
9	59 049	39 364	511
10	177 147	118 096	1023

5.7.4. Deictic Representation

We want the robot to perform the task regardless of the size of the environment (the number of boxes) or the size of each box but still adhere to the box stacking constraint. So, for the deictic descriptor representation we use features which do not include these sizes.

Table 5.29. An example propositional representation of 5 boxes with 3 boxes on the source pallet and 2 boxes on the temporary pallet. The robot is facing the temporary pallet.

Feature	Feature name	Feature Value
1	RobotPos	T
2	PalletS	[5,4,3]
3	PalletT	[2,1]
4	PalletD	[ ]

The number of boxes is reflected in the boolean feature "IsEven" which is True if the number of boxes is even. For this task, the robot is interested in the relation of the top-of-stack

box sizes, not in their absolute sizes. So, we use 3 relational boolean features which compare the boxes on the top of each stack. These features are “F>R”, “F>L”, and “R>L” which in words mean “the top box I am facing is larger than the top box on the right stack”, “the top box I am facing is larger than the top box on the left stack”, and “the top box on the right stack is larger than the top box on the left stack”. Note that for these comparisons, a pallet is considered larger than any box and no pallet is larger than any other pallet. For these 3 features, there are 8 combinations of True/False assignments but only 4 of these are possible situations. First, the assignments (F>R=False, F>L=True, R>L=False) and (F>R=True, F>L=False, R>L=True) are not logically possible. Continuing, except for the initial configuration, the robot is facing a stack because it just moved a box from some other stack. Therefore, the top box of the stack the robot is currently facing (the F stack) cannot be larger than both the right and left top boxes. But two of the 8 combinations have F>R and F>L and so are not possible. This eliminates two more combinations leaving 4 possible cases as shown in Table 5.30. Note that the initial environment configuration is case 2. The last feature, “EmptyRL”, is True when the pallets to the robot’s right and left are both empty. This could be the task complete configuration if the robot is actually facing the destination pallet (but not so if the robot is facing the source or temporary pallet). Also, this feature can only be True for Case 1 of Table 5.30. So, all total, this results in 10 deictic descriptors as shown in Table 5.31.

5.7.5. Training

For learning, the robot needs at least 2 environments, one with an odd number of boxes and the other with an even number of boxes. Following our maxim “use small examples”, we chose a 3 box and a 4 box environment. The robot, using these 2 small examples, learns the

deictic descriptor task policy shown in Table 5.32. This policy, when used by the robot, performs the "move a size-ordered stack of boxes" task with any number of boxes utilizing the optimal (minimum) number of actions.

Table 5.30. The 4 possible cases for the 3 size relation features.

	F>R	**F>L**	**R>L**
Case 1	False	False	False
Case 2	True	False	False
Case 3	False	False	True
Case 4	False	True	True

Table 5.31. The 10 deictic descriptors for the "move a size-ordered stack of boxes" task.

Deictic Descriptor	IsEven	F>R	F>L	R>L	EmptyRL
D1	False	False	False	False	False
D2	False	True	False	False	False
D3	False	False	False	True	False
D4	False	False	True	True	False
D5	False	False	False	False	True
D6	True	False	False	False	False
D7	True	True	False	False	False
D8	True	False	False	True	False
D9	True	False	True	True	False
D10	True	False	False	False	True

5.7.6. Q-learning and R-max Comparisons

The Q-learning and R-max algorithms comparison cases were done using 3, 4, 5, and 6 box environments. After training, the robot performed the task in each environment using the optimal (minimum) number of actions. The results are displayed in Table 5.33. When using the

deictic task policy learning algorithm, the robot learned a universal policy 165 times faster than Q-learning and 15 times faster than R-max.

Table 5.32. The deictic descriptor task policy table for the "move a size-ordered stack of boxes" task.

Deictic Descriptor	IsEven	F>R	F>L	R>L	EmptyRL	Action
D1	False	False	False	False	False	MRL
D2	False	True	False	False	False	MRF
D3	False	False	False	True	False	MLR
D4	False	False	True	True	False	MLR
D5	False	False	False	False	True	Q
D6	True	False	False	False	False	MRL
D7	True	True	False	False	False	MRL
D8	True	False	False	True	False	MLR
D9	True	False	True	True	False	MLF
D10	True	False	False	False	True	Q

Table 5.33. Training results for the "move a size-ordered stack of boxes" task.

Move a size-ordered stack boxes	Q-learning	R-max	Deictic
No. of physical actions	135203	12848	256
No. of episodes	330	49	58
Training metric (Days)	1.6	0.15	0.0097

5.8. Manufacturing World: Inspect Box Contents

In the previous examples, the robot, in McCallum's terminology [13], chooses to use selective perception. During task execution, the robot perceives the complete configuration of the environment, but chooses to concentrate on certain features and ignore others. This environment is designed to show how a robot can learn a deictic descriptor task policy in a fully observable environment but then use the policy in a partially observable environment where the robot is forced to contend with hidden state during task execution. The environment

consists of 3 conveyer belts, one for incoming boxes and two for outgoing boxes. The robot's task consists of 5 steps:

1. Lift the lid of the box on the incoming conveyer belt.
2. Inspect the contents – is the contents OK or not OK?
3. Close the lid on the box.
4. Place the box on either the OK outgoing conveyer or the NotOK conveyer.
5. Push a button to move the conveyer belts to get another box and send the inspected box on its way.

These steps continue until there are no more boxes to process.

The robot has 2 difficulties in this task. In steps 1 and 4, before taking an action, the robot is looking at a closed box on the incoming conveyer belt. Taken in isolation, the two steps/configurations appear identical to the robot. Should it lift the lid of the box or move the box to the outgoing belt? Also in step 4, since the box is closed, how does the robot know whether to place the box on the OK belt or the NotOK belt? Because of the hidden contents of the box, value-based reinforcement learning algorithms will not work for this task. If we enable the robot to perceive the closed box contents, then the robot could acquire a policy using a value-based approach like Q-learning or R-max. The learned policy by these methods would not use steps 1, 2, and 3. The robot, being able to "see" the box contents without lifting the box lid, would optimize and just use steps 4 and 5.

For the deictic descriptor task policy learner, the fully observable states are used for detecting looping behavior (not for learning the policy). The trick we use is to allow the robot to have the information needed for the states but deny this information for deictic descriptors.

Then, the algorithm can detect looping behavior and also learn the deictic descriptor task policy for the partially observable problem. So, for this "inspect box contents" task, we make the task learnable by temporarily equipping the robot with an X-ray camera. After the task is learned, we remove the camera and use only the deictic descriptor task policy.

5.8.1. Actions

For the "inspect box contents" task, the robot has 5 actions:

- O: The robot opens the box. This will only succeed if there is a box on the incoming conveyor belt and the box is closed.
- C: The robot closes the box. This will only succeed if there is a box on the incoming conveyor belt and the box is open.
- MOV_OK: The robot moves the box from the incoming conveyor belt to the outgoing Ok belt. This action succeeds only if their is a box on the incoming belt and its lid is closed.
- MOV_NOK: The robot moves the box from the incoming conveyor belt to the outgoing NotOk belt. This action succeeds only if their is a box on the incoming belt and its lid is closed.
- NEXT: The input conveyor belt increments to the next box to be inspected. The output conveyor belts move the checked box down the line.

5.8.2. Propositional Representation

In the fully observable environment, we define features for each box. These features describe the location of the box, the box lid status, and the box content status. Each conveyor's possible box positions are numbered. For the input conveyor, the box inspection position is

position 0. The next box to be inspected is position 1 and so on. The output conveyors are similar with the location for a newly inspected box being position 0. When the NEXT action is executed, the inspected box in position 0 is now in position 1, leaving position 0 open for another newly inspected box. Here is the description of each feature:

- Box<i>Loc: This is the location of box i. The value of this feature is a tuple (Cnvr, Pos) where Cnvr is either 'inbelt', 'okbelt', or 'nokbelt' and the Pos is the position on that belt.
- Box<i>Lid: This is the status of the lid for box i with a value of either 'open' or 'closed'.
- Box<i>Content: This is the status of the contents of box i with a value of either 'ok' or 'nok'.

5.8.3. Deictic Representation

With the deictic policy learning algorithm, the robot overcomes the hidden state by using multiple policies. During the task learning stage, we allow the robot to perceive the full configuration (including the box contents) for detecting looping situations. The deictic descriptors, however, only know the box content status when the box lid is open. This forces the robot to open the box to determine the correct output conveyor belt. After the robot has learned the deictic descriptor policy, the robot can perform the tasks in the partially observable environment and does not need the Markov states.

The boolean features for the deictic descriptors are:

- InEmpty: The boolean status of the inspection position of the input conveyor belt. A value of True means the position is empty.

- OutBox: The boolean status of the output conveyor belts. This feature is True if either of output conveyor belts have a box. Once the NEXT action is executed, the box (if any) is moved down the line and this feature becomes False.
- BoxOpen: The boolean status of the lid of the box being inspected with True meaning the box is open.and "
- ContentOk: This feature is the status of the box contents and has 3 possible values. If the box is closed, the value is 'unkown'. If the box is open, it has either 'ok' or 'nok' meaning the contents inspected ok or not ok.

5.8.4. Training

Using 2 examples with a single box and an example with 2 boxes, the robot learns the deictic policy shown in Table 5.34. The policy has a main policy and 2 sub-policies and performs as follows for a box with "ok" contents (assuming the initial configuration with the newly uninspected box on the input conveyor belt and no inspected box on the output conveyors):

- The robot is mentally in the Main policy and perceives descriptor D1. The Main policy action is open the box, O, so the robot opens the lid on the box.
- After executing the open box, O, action, the contents of the box is revealed. The contents are ok the robot perceives descriptor D3.
- The Main policy action for the D3 descriptor is a mental policy switch action, x2, which causes the robot to switch to Sub2 policy which contains the close box, C, action. The robot performs the C action and closes the box. With the box closed and still on the input conveyor belt, the robot again perceives the D1 descriptor.

- The robot is still mentally in the Sub2 policy so the action for descriptor D1 is to move the box to the “ok” output belt, MOV_OK. Now, with the inspected box on an output belt, the robot perceives descriptor D4.
- The action in the Sub2 policy for descriptor D4 is a mental action, q, which causes the robot to mentally switch back to the Main policy and execute the NEXT action. The NEXT action brings in a new box on the inspection position of the input belt (or not if there are no more boxes to process) and moves the box on the output belt down the line. The robot now either perceives descriptor D1 (a new box) or D5 if the inspection position of the input belt is empty. The latter case results in the task completion status mental action Q.

The steps above are the same for a box with “not ok” contents except the robot would mentally switch to Sub1 policy after opening the box and end up putting the box on the “not ok” output belt. This policy is universal and can be used by the robot for any number of boxes on the input belt.

Table 5.34. The deictic descriptor task policy for the “inspect box contents” task.

Deictic Descriptor	InEmpty	OutBox	BoxOpen	ContentOk	Main Policy	Sub1 Policy	Sub2 Policy
D1	False	False	False	unknown	O	MOV_NOK	MOV_OK
D2	False	False	True	nok	x1	C	
D3	False	False	True	ok	x2		C
D4	True	True	False	unknown	NEXT	q	q
D5	True	False	False	unknown	Q		

5.8.5. Q-learning and R-max Comparisons

Learning the “inspect box contents” task falls into the class of partially observable problems so is not learnable by the Q-learning or R-max algorithms. The difficulty arises

because the robot must close the box before it can move it to the output belt. With the box closed, the robot does not know which output belt to put it on. For the deictic learner, the algorithm figures out (on its own) that it must "remember" the status of the box contents and creates sub policies for each box content situation.

The robot took 6416 actions and 2255 episodes to learn this simple task. Using 1 second per action and 10 seconds for setting up an episode, this required over 8 hours of elapsed time. The basic deictic descriptor policy learning algorithm performed its duty but seems deficient in the time efficiency department! In the next chapter, we confront this issue with algorithm improvements and reduce the learning time for this task from 8 hours down to 4 minutes.

Chapter 6. Improving the Basic Learning Algorithm

6.1. Basic Algorithm Deficiencies

The basic algorithm introduced in chapter 4 met our requirements of learning a task policy applicable to any instance environment of an environment domain. When compared with the value-based learning algorithms, Q-learning and R-max, the basic algorithm was competitive. But, taking a close look at the paths taken during the learning process, we find that many action sequences are repeated. Also, every dead end in the search for policy entries results in a new episode. So to improve the performance, we borrow from the model-based reinforcement learning algorithms (like R-max) and include a memory for transition experiences. The next section discussion how the basic algorithm is modified to include a memory for robot experience in the task learning process.

6.2. Robot Experience Retention

At each time step during the task learning phase, the robot samples the current environment configuration, translate it into both a Markovian state and deictic descriptor and then takes an action. This action results in a (perhaps) new configuration, state, and descriptor. In the basic algorithm, this transition may be repeated many times, possibly executing many unneeded actions. So, to remedy this deficiency, we add a memory to retain these transition experiences.

Reusing experience requires the robot to have 2 modes of operation. The first mode is when the robot is performing actions in the “online” environment. This is the mode the robot uses in the basic algorithm. The second mode, the “offline” mode, is when the robot is performing actions from its retention memory. For managing these modes, the robot employs

a “path” list. As the robot performs actions, it keeps track of the configurations and actions it has taken by updating the path list. The robot always begins in the “online” mode, updating retention memory and the path list as it goes. If the robot encounters a situation it has seen in the past, it transfers to “offline” mode and begins taking “virtual” actions. In “offline” mode, it continues to update the path list. If at anytime while in “offline” mode, the robot comes upon a situation not in its retention memory, it must return to “online” mode. The problem is that the robot is physically still in the position and pose it was in when it entered “offline” mode. This requires the robot to use the path list to actually perform the actions it did not take while in “offline” mode until it is up-to-date with the unseen situation.

In the basic algorithm version, each time a new extended policy is tried, the robot must restart from the initial configuration (a new episode). This is very inefficient in physical action usage and episode counts. With retention memory, we keep status information along with the extended policies. Then, when we start a new extended policy, the robot is backed up to the untried part of the policy. If the robot stays in the “offline” mode during this back up, then an initialization is not required. This saves on physical action usage and episode counts.

6.3. Basic Algorithm vs Experience Retention Memory Algorithm

We tested the improved algorithm on the 7 environment domains of the previous chapter. In each case, the same training examples and costs are used. As before, the timing is 1 second to perform a physical action and 10 seconds to initialize the environment for a new episode. The comparison results are in Table 6.1 (actions), Table 6.2 (episodes), and Table 6.3 (training metric). Note that the training times are now in minutes instead of days. The results show significant reductions in physical actions and the required number of training episodes.

Table 6.1. The physical actions taken during learning for each example task using the basic deictic learning algorithm and the enhanced algorithm with experience retention memory.

Task vs Actions taken	Basic (Actions)	Memory (Actions)
Alphabetizing name tags	12474	341
Put block A on block B	22583	1068
Pick up the green box	14939	980
Adding binary numbers	1222	298
Adding binary numbers (carry)	1727	535
Navigating rooms with keyed doors	83681	6973
Move a size-ordered stack of boxes	256	136
Inspecting box contents	6416	84

6.4. Reversing Actions

The retention memory algorithm much improves the basic algorithm but still requires a large number of episodes. For example, the blocks world on(A,B) task required over 400 episodes. For a user, setting up the blocks for each of these episodes would not be much fun! So, to help with this difficulty, we add "reversing" actions to the learning algorithm. As the name implies, a "reversing" action undoes an action and the reversing actions are only used during task learning. So, for any environment configuration Ci and action Ak where taking action Ak while in Ci results in environment configuration Cj, we propose a reversing action Akr which when executed in Cj will result in a transition back to Ci. To use reversing actions, the robot learns an "undo" function U(Ci,Ak,Cj): Ci x Ak x Cj -> Akr. Updating the deictic learning algorithm with reversing actions reduces the number of episodes to one per training example. For the on(A,B) task, this reduces the episodes from 444 to 11, a substantial reduction.

Table 6.2. The number of episodes used during learning for each example task using the basic deictic learning algorithm and the enhanced algorithm with experience retention memory.

Task vs Episodes used	Basic (Episodes)	Memory (Episodes)
Alphabetizing name tags	2094	37
Put block A on block B	9777	444
Pick up the green box	3004	127
Adding binary numbers	324	70
Adding binary numbers (carry)	1727	535
Navigating rooms with keyed doors	4018	334
Move a size-ordered stack of boxes	58	14
Inspecting box contents	2255	16

Learning the undo function may be easy or difficult. For example, the left and right movement actions in the binary addition task are inverses of each other, hardly any learning required. But, the get and put actions in the alphabetizing name tags task require the robot to look at which hand it used, making learning the reversing actions more difficult.

Table 6.3. The total time used during learning for each example task using the basic deictic learning algorithm and the enhanced algorithm with experience retention memory. The assumptions are 1 second per physical action and 10 seconds to set up each episode.

Task vs Learning time (1 second/action, 10 seconds/episode)	Basic (Minutes)	Memory (Minutes)	Ratio Basic/Mem
Alphabetizing name tags	557	11.9	47
Put block A on block B	2006	91.8	22
Pick up the green box	220.2	19.2	11
Adding binary numbers	74.4	16.6	4.5
Adding binary numbers (carry)	89.5	25.4	3.5
Navigating rooms with keyed doors	1865	191	9.8
Move a size-ordered stack of boxes	13.9	4.60	3.0
Inspecting box contents	483	4.07	119

In all our results, we have used a default of 10 seconds to set up an episode. Suppose it took 100 seconds or 10 minutes instead of 10 seconds. This may entice the user to consider

reversing actions. The number of actions required for each version is roughly equivalent so, in Table 6.4, we show the additional time required when not using reversing actions assuming 3 different episode setup times of 10 seconds, 100 seconds, and 1000 seconds. The delta time increase proportionally with the setup time as expected.

Table 6.4. The total time used during learning for each example task using the enhanced experience retention memory algorithm and the reversing action algorithm episode set. The assumptions are 1 second per physical action and 10 seconds to set up each episode.

Task vs Additional time required of retention memory to reversing action	10sec setup, delta hours	100sec setup, delta hours	1000sec setup, delta hours
Alphabetizing name tags	0.1	0.9	9.4
Put block A on block B	1.1	11.9	120.2
Pick up the green box	0.1	1.9	19.4
Adding binary numbers	0.1	1.7	17.5
Adding binary numbers (carry)	0.2	2.5	25.5
Navigating rooms with keyed doors	1.1	10.1	100.4
Move a size-ordered stack of boxes	0.02	0.3	3.3
Inspecting box contents	0.03	0.4	3.6

In Table 6.5 and Figure 6.1 we show the performance metric results for the different versions of the deictic descriptor policy learning algorithm.

Table 6.5. The performance metric in minutes for each example task and each algorithm version. The assumptions are 1 second per physical action and 10 seconds for episode setup.

Task vs Learning Metric (1 second/action, 10 seconds/episode)	Basic (Minutes)	Memory (Minutes)	Reversing (Minutes
Alphabetizing name tags	557	11.9	6.2
Put block A on block B	2006	91.8	25.7
Pick up the green box	220	19.2	10.7
Adding binary numbers	74.4	16.6	7.7
Adding binary numbers (carry)	89.5	25.4	11.5
Navigating rooms with keyed doors	1865	191	125
Move a size-ordered stack of boxes	13.9	4.6	3.5
Inspecting box contents	483	4.1	2.1

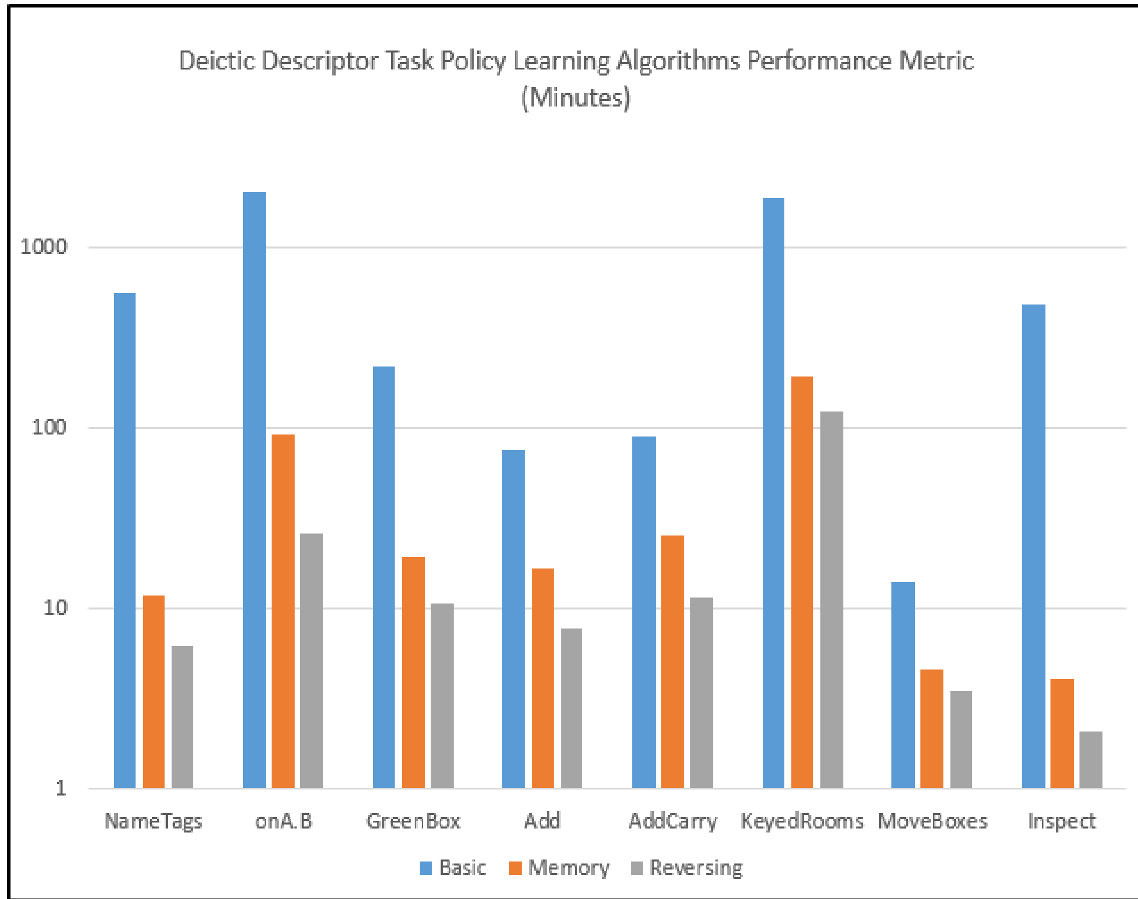

Figure 6.1. The performance metric in minutes for each example task and each algorithm version. The assumptions are 1 second per physical action and 10 seconds for episode setup.

Chapter 7. Conclusions and Future Work

7.1. Summary

The first chapter of this work introduces the thesis statement and follows with a layman's description of the concepts of a robot, an environment, state and deictic representations, satisficing behavior, and learning. A small example is included to set the stage for learning universal task policies applicable to more than one specific environment. Chapter 1 ends with a brief outline of the dissertation.

Chapter 2 covers background information. Like most other works in this field, it reviews reinforcement learning, Markov decision processes, POMDPs, and hierarchical decomposition. The area of transfer learning is also covered due to its goal of reusing policies from one environment to another environment. Importantly, the chapter outlines our meanings for environment domains and domain tasks, physical and mental environments, and representation methods.

Chapter 3 looks at related works. Past research in learning with the deictic representation is emphasized. It also briefly examines works related to the test examples and relational reinforcement learning.

Chapter 4 describes the basic algorithm for learning deictic descriptor task policies. Instead of using a value-based method, the algorithm searches for a policy directly using a small number of examples. The algorithm details are described with pseudocode and supporting methods are summarized.

In Chapter 5, the policy learning algorithm of Chapter 4 is tested on seven example tasks. For each example, the propositional and deictic representation is presented. The environment

specification parameters are reviewed along the environment domain's sizes in terms of configurations and varied parameters. The training examples used for learning are discussed, emphasizing that, because of the universal nature of a deictic descriptor policy, smaller is better in terms of learning times. Where possible, the Q-learning algorithm and R-max algorithm are employed to produce comparisons in learning times.

Chapter 6 improves the basic deictic descriptor task policy learning algorithm by adding experience retention memory to reduce repeated action sequences and by allowing reversing actions to reduce the number of episode setups required. Results are given, showing large reductions in the performance metric over the basic algorithm of Chapter 4.

7.2. Conclusions

The deictic descriptor task learning algorithms developed from our research allow a robot to learn a deictic task policy for an environment domain and domain task when the instance environments are modeled by MDPs. These policies are universal in nature and can be learned by using a small number of training examples.

The deictic descriptor task policy algorithms were tested using a simulated robot in 7 example environment domains. The robot learned a satisficing and sometimes optimal deictic descriptor task policy for each example domain. Where applicable, Q-learning and R-max comparisons were made. The performance of each learning algorithm was based on the elapsed time needed by the robot to learn a task policy for a relatively small instance environment of the domain. The timing used for the metric in all cases assumed 1 second per physical robot action and 10 seconds to set up an episode. The chart in Figure 7.1 summarizes the results of the comparisons. Note, these comparisons are meant to be a qualitative measure

of the relative learning performance of the deictic descriptor task policy learning algorithm. No transfer learning methods were used to help improve the Q-learning and R-max algorithms task learning performance. Also, typical tuning parameter values were used for the Q-learning algorithm.

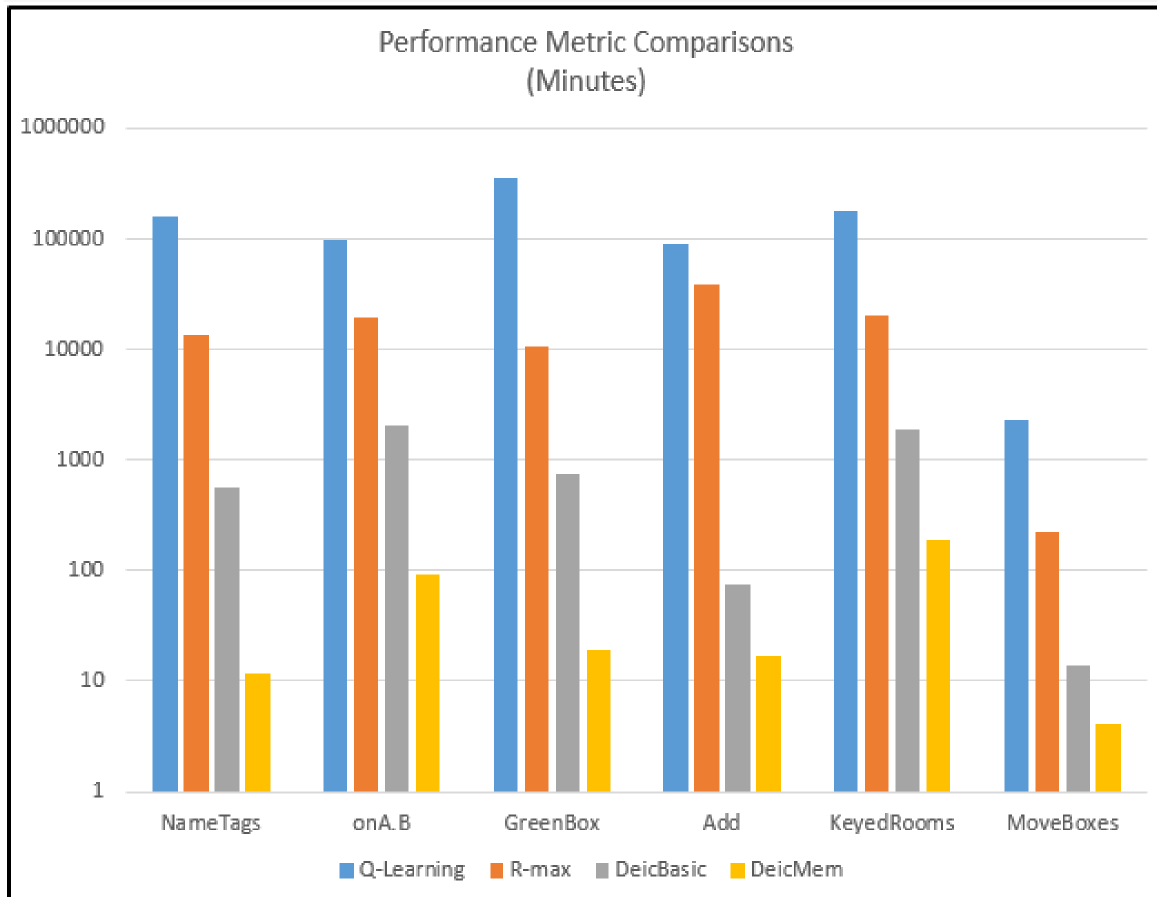

Figure 7.1. A performance metric comparison of deictic descriptor task policy learning, Q-learning, and R-max.

7.3. Future Work

The use of the deictic representation in reinforcement learning is not as well explored as other representation methods. For us, there seems to be a large amount of research opportunities to be had. Here, we give a few ideas of some of these we are interested in pursuing.

7.3.1. Deictic Descriptor Functions with Variables

In relational reinforcement learning settings, it is natural to use variables as task arguments. This feature would be nice for deictic descriptor task learning. For example, the deictic descriptor function for the "stack block A on block B" task has the actual block names hardcoded into the function. The same goes for the "pick up the green box" task. Bringing these out as variables would not be difficult. But just replacing the A and B names or the color Green attribute with new names or colors would not be enough because the functions contain references to "other" blocks and "other" boxes. The robot may need to understand relations and logical expressions. Using variables in tasks would also be beneficial for using these tasks as high-level actions in a hierarchical manner. Using the variable arguments would allow the high-level actions to be customizable in the new learning task.

7.3.2. Learning Deictic Descriptor Functions from Imitation

In imitation learning, someone, perhaps an expert, demonstrates the task to the robot, helping the robot learn. For the propositional Markovian state case, the robot observes the states as the demonstration progresses, seeing the path from start state to ending state. In the deictic case, the descriptors are task specific so the robot could also learn the descriptors from the sequence of configurations seen along the path. How to learn these deictic descriptors is an interesting area of study.

7.3.3. Interruptible Deictic Descriptor Policies

The robot, when performing a task, should be able to respond to abnormal events. The simplest kind of event would not change the actual task sequence, just put it on hold while the other task is performed. This would be fairly easy to implement by saving the current situation

and perhaps restart information. The more difficult case, where the task actions need to reordered or change is an interesting problem and requires more study.

References

[1] R. Bellman, *Dynamic programming*. Princeton, NJ: Princeton University Press, 1957.

[2] R. Bellman, *Adaptive Control Processes: A Guided Tour*. Princeton, NJ: Princeton University Press, 1961.

[3] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D., King's College, Cambridge, Cambridge, England, 1989.

[4] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement Learning in Robotics: A Survey," in *Reinforcement LearningState-of-the-Art*. vol. 12, M. Wiering and M. van Otterlo, Eds., ed: Springer Berlin Heidelberg, 2012, pp. 579-610.

[5] M. E. Taylor and P. Stone, "Transfer learning for reinforcement learning domains: A survey," *The Journal of Machine Learning Research,* vol. 10, pp. 1633-1685, 2009.

[6] R. S. Sutton and A. G. Barto, *Introduction to reinforcement learning* vol. 135: MIT press Cambridge, 1998.

[7] M. van Otterlo and M. Wiering, "Reinforcement Learning and Markov Decision Processes," in *Reinforcement Learning State-of-the-Art*, ed: Springer, 2012, pp. 3-42.

[8] R. I. Brafman and M. Tennenholtz, "R-max-a general polynomial time algorithm for near-optimal reinforcement learning," *Journal of Machine Learning Research,* vol. 3, pp. 213-231, 2002.

[9] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence,* vol. 112, pp. 181-211, 1999.

[10] M. P. Deisenroth, P. Englert, J. Peters, and D. Fox, "Multi-task policy search for robotics," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, 2014, pp. 3876-3881.

[11] A. Fern, S. Yoon, and R. Givan, "Approximate policy iteration with a policy language bias: Solving relational Markov decision processes," *Journal of Artificial Intelligence Research,* vol. 25, pp. 75-118, 2006.

[12] S. Finney, N. H. Gardiol, L. P. Kaelbling, and T. Oates, "The thing that we tried didn't work very well: deictic representation in reinforcement learning," in *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence*, 2002, pp. 154-161.

[13] A. K. McCallum, "Reinforcement learning with selective perception and hidden state," PhD dissertation, Computer Science, University of Rochester, 1996.

[14] P. E. Agre and D. Chapman, "Pengi: an implementation of a theory of activity," in *Proceedings of the sixth National conference on Artificial intelligence-Volume 1*, 1987, pp. 268-272.

[15] S. D. Whitehead and D. H. Ballard, "Learning to Perceive and Act by Trial and Error," *Machine Learning,* vol. 7, pp. 45-83, 1991.

[16] M. van Otterlo, "Solving Relational and First-order Logical Markov Decision Processes: A Survey," in *Reinforcement Learning State-of-the-Art*, ed: Springer, 2012, pp. 253-292.

[17] B. Ravindran, "An algebraic approach to abstraction in reinforcement learning," PhD dissertation, Computer Science, University of Massachusetts at Amherst, 2004.

[18] S. Džeroski, L. De Raedt, and K. Driessens, "Relational reinforcement learning," *Machine learning,* vol. 43, pp. 7-52, 2001.

[19] T. Croonenborghs, K. Driessens, and M. Bruynooghe, "Learning Relational Options for Inductive Transfer in Relational Reinforcement Learning," Berlin, Heidelberg, 2008, pp. 88-97.

[20] M. B. Ring, "Continual learning in reinforcement environments," PhD dissertation, University of Texas at Austin, 1994.

[21] K. Driessens and J. Ramon, "Relational instance based regression for relational reinforcement learning," in *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 2003, pp. 123-130.

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

[23] A. Lazaric, "Knowledge transfer in reinforcement learning," PhD dissertation, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano, Italy, 2008.

[24] T. J. Walsh, "Efficient learning of relational models for sequential decision making," PhD dissertation, Computer Science, Rutgers University, 2010.

[25] H. B. Ammar, E. Eaton, M. E. Taylor, D. C. Mocanu, K. Driessens, G. Weiss*, et al.*, "An automated measure of MDP similarity for transfer in reinforcement learning," in *Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[26] L. D. Raedt, K. Kersting, S. Natarajan, and D. Poole, "Statistical relational artificial intelligence: Logic, probability, and computation," *Synthesis Lectures on Artificial Intelligence and Machine Learning,* vol. 10, pp. 1-189, 2016.

[27] S. Sarjant, B. Pfahringer, K. Driessens, and T. Smith, "Using the online cross-entropy method to learn relational policies for playing different games," in *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, 2011, pp. 182-189.

[28] S. Sarjant, "Policy search based relational reinforcement learning using the cross-entropy method," PhD dissertation, University of Waikato, 2013.

[29] M. L. Littman, "Inducing Partially Observable Markov Decision Processes," in *ICGI*, 2012, pp. 145-148.

[30] A. Lazaric and M. Restelli, "Transfer from multiple MDPs," in *Advances in Neural Information Processing Systems*, 2011, pp. 1746-1754.

[31] H. B. Ammar, K. Tuyls, M. E. Taylor, K. Driessens, and G. Weiss, "Reinforcement learning transfer via sparse coding," in *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, 2012, pp. 383-390.

[32] O. Kozlova, "Hierarchical & Factored Reinforcement Learning," PhD dissertation, Computer Science, Université Pierre et Marie Curie-Paris VI, 2010.

[33] H. B. Ammar, D. C. Mocanu, M. E. Taylor, K. Driessens, K. Tuyls, and G. Weiss, "Automatically mapped transfer between reinforcement learning tasks via three-way restricted boltzmann machines," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2013, pp. 449-464.

[34] M. E. Taylor, "Autonomous inter-task transfer in reinforcement learning domains," PhD dissertation, The University of Texas at Austin, 2008.

[35] A. Boularias, "Predictive Representations For Sequential Decision Making Under Uncertainty," Citeseer, 2010.

[36] C. Devin, A. Gupta, T. Darrell, P. Abbeel, and S. Levine, "Learning modular neural network policies for multi-task and multi-robot transfer," in *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, 2017, pp. 2169-2176.

[37] M. M. Muñoz, "Reinforcement learning for embedded agents facing complex tasks," PhD dissertation, Universitat Politècnica de Catalunya, 1998.

[38] M. Wiering and J. Schmidhuber, "HQ-Learning: Discovering Markovian subgoals for non-Markovian reinforcement learning," *Technical report IDSIA-95-96, IDSIA,* pp. 1-13, 1996.

[39] M. Wiering and J. Schmidhuber, "Solving POMDPs with Levin Search and EIRA," in *Machine Learning: Proceedings of the thirteenth International Conference*, 1996, p. 534.

[40] M. Wiering and J. Schmidhuber, "HQ-learning," *Adaptive Behavior,* vol. 6, pp. 219-246, 1997.

[41] P. E. Agre, "The dynamic structure of everyday life," MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB AI-TR-1085, 1988.

[42] R. E. Korf, "Macro-operators: A weak method for learning," *Artificial Intelligence,* vol. 26, pp. 35-77, 1985/04/01/ 1985.

[43] S. Amarel, "On Representations of Problems of Reasoning about Actions," in *Machine Intelligence*. vol. 3, D. Michie, Ed., ed New York: American Elsevier, 1968.

[44] H. A. Simon, "Rational Choice and the Structure of the Environment," *Psychological Review,* vol. 63, pp. 129-138, 1956.

[45] S. D. Whitehead, "Reinforcement learning for the adaptive control of perception and action," ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE TR-406, 1992.

[46] S. D. Whitehead and L.-J. Lin, "Reinforcement learning of non-Markov decision processes," *Artificial Intelligence,* vol. 73, pp. 271-306, 1995/02/01/ 1995.

[47] H. Yim and D. Kim, "Evolving internal memory strategies for the woods problems," in *Control, Automation and Systems (ICCAS), 2012 12th International Conference on*, 2012, pp. 366-369.

[48] P. Poupart, "Exploiting structure to efficiently solve large scale partially observable Markov decision processes," PhD dissertation, Computer Science, University of Toronto, 2005.

[49] S. P. Singh, T. Jaakkola, and M. I. Jordan, "Learning without state-estimation in partially observable Markovian decision processes," in *Machine Learning Proceedings 1994*, ed: Elsevier, 1994, pp. 284-292.

[50] B. Bakker and J. Schmidhuber, "Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization," in *Proc. of the 8-th Conf. on Intelligent Autonomous Systems*, 2004, pp. 438-445.

[51] R. A. McCallum, "First results with utile distinction memory for reinforcement learning," University of Rochester Technical Report 446, 1992.

[52] R. A. McCallum, "Overcoming incomplete perception with utile distinction memory," in *Proceedings of the Tenth International Conference on Machine Learning*, 1993, pp. 190-196.

[53] R. A. McCallum, "Instance-based utile distinctions for reinforcement learning with hidden state," in *Machine Learning Proceedings*, 1995, pp. 387-395.

[54] M. L. Littman, "Memoryless policies: Theoretical limitations and practical results," in *From Animals to Animats 3: Proceedings of the third international conference on simulation of adaptive behavior*, 1994, p. 238.

[55] M. L. Littman, "Algorithms for sequential decision making," PhD dissertation, Computer Science, Brown University, 1996.

[56] E. Parisotto, J. L. Ba, and R. Salakhutdinov, "Actor-mimic: Deep multitask and transfer reinforcement learning," in *International Conference on Learning Representations*, 2016.

[57] L. Peshkin, N. Meuleau, and L. P. Kaelbling, "Learning Policies with External Memory," in *Proceedings of the Sixteenth International Conference on Machine Learning*, 1999, pp. 307-314.

[58] B. Ravindran and A. G. Barto, "Relativized options: Choosing the right transformation," in *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 2003, pp. 608-615.

[59] B. Ravindran, A. G. Barto, and V. Matthew, "Deictic option schemas," in *IJCAI'07: Proceedings of the 20th international joint conference on Artifical intelligence*, San Francisco, CA, USA, 2007, pp. 1023–1028.

[60] D. Wierstra, "Statistical models for non-markovian control tasks," PhD dissertation, Utrecht University, 2004.

[61] D. Wierstra and M. Wiering, "Utile distinction hidden Markov models," in *Proceedings of the twenty-first international conference on Machine learning*, 2004, p. 108.

[62] J. Schmidhuber, "Optimal ordered problem solver," *Machine Learning,* vol. 54, pp. 211-254, 2004.

[63] S. Sanner and K. Kersting, "Symbolic Dynamic Programming for First-order POMDPs," in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[64] M. A. Wiering, "Explorations in efficient reinforcement learning," PhD dissertation, University of Amsterdam, 1999.

[65] T. Croonenborghs, J. Ramon, H. Blockeel, and M. Bruynooghe, "Online Learning and Exploiting Relational Models in Reinforcement Learning," in *IJCAI*, 2007, pp. 726-731.

[66] P. Crook and G. Hayes, "Learning in a state of confusion: Perceptual aliasing in grid world navigation," presented at the Towards Intelligent Mobile Robots, 2003.

[67] P. A. Crook, "Learning in a state of confusion: employing active perception and reinforcement learning in partially observable worlds," PhD dissertation, University of Edinburgh, 2006.

[68] P. A. Crook and G. Hayes, "Consistent exploration improves convergence of reinforcement learning on POMDPs," in *AAMAS 2007 workshop on adaptive and learning agents*, 2007.

[69] H. B. Ammar, E. Eaton, J. M. Luna, and P. Ruvolo, "Autonomous cross-domain knowledge transfer in lifelong policy gradient reinforcement learning," in *International Joint Conference on Artificial Intelligence*, 2015, pp. 3345-3351.

[70] E. Çilden, "Abstraction in Reinforcement Learning in Partially Observable Environments," PhD dissertation, Middle East Technical University, 2014.

[71] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research,* pp. 237-285, 1996.

[72] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial intelligence,* vol. 101, pp. 99-134, 1998.

[73] L. P. Kaelbling, T. Oates, N. Hernandez, and S. Finney, "Learning in worlds with objects," in *Working Notes of the AAAI Stanford Spring Symposium on Learning Grounded Representations*, 2001, pp. 31-36.

[74] T. Gärtner, K. Driessens, and J. Ramon, "Graph kernels and gaussian processes for relational reinforcement learning," in *International Conference on Inductive Logic Programming*, 2003, pp. 146-163.

[75] W. Dabney and A. McGovern, "The thing we tried that worked: Utile distinctions for relational reinforcement learning," in *Proceedings of the ICML-06 Workshop on Open Problems in Statistical Relational Learning*, 2006.

[76] W. Dabney and A. McGovern, "Utile distinctions for relational reinforcement learning," in *Proceedings of the 20th international joint conference on Artifical intelligence*, 2007, pp. 738-743.

[77] W. Dabney and A. McGovern, "Multi-modal utile distinctions," University of Massachusetts Technical Report UM-CS-2010-065, 2010.

[78] S. S. Koenig, Reid G., "Complexity Analysis of Real-Time Reinforcement Learning," in *International Conference on Artificial Intelligence*, 1993.

[79] A. Lazaric, "Transfer in reinforcement learning: a framework and a survey," in *Reinforcement Learning*, ed: Springer, 2012, pp. 143-173.

[80] M. Grześ and J. Hoey, "Efficient planning in R-max," in *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 3*, 2011, pp. 963-970.

[81] M. Hutter, "Feature reinforcement learning: Part I. unstructured MDPs," *Journal of Artificial General Intelligence,* vol. 1, pp. 3-24, 2009.

[82] V. F. Farias, C. C. Moallemi, B. Van Roy, and T. Weissman, "Universal reinforcement learning," *IEEE Transactions on Information Theory,* vol. 56, pp. 2441-2454, 2010.

[83] D. Chapman. (1989) Penguins can make cake. *AI Magazine*. 45-50.

[84] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274,* 2017.

[85] B. Bakker, "Reinforcement Learning with Long Short-Term Memory," in *Advances in Neural Information Processing Systems*, 2002, pp. 1475-1482.

[86] S. Kalyanakrishnan, "Learning methods for sequential decision making with imperfect representations," PhD dissertation, Computer Science, The University of Texas at Austin, 2011.

[87] K. Driessens, J. Ramon, and H. Blockeel, "Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner," in *European Conference on Machine Learning*, 2001, pp. 97-108.

[88] N. K. Jong and P. Stone, "Hierarchical model-based reinforcement learning: R-max+ MAXQ," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 432-439.

[89] H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling, "Learning symbolic models of stochastic domains," *Journal of Artificial Intelligence Research,* vol. 29, pp. 309-352, 2007.

[90] K. Kersting and L. De Raedt, "Logical Markov decision programs and the convergence of logical TD (λ)," in *International Conference on Inductive Logic Programming*, 2004, pp. 180-197.

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

[92] T. J. Walsh, L. Li, and M. L. Littman, "Transferring state abstractions between MDPs," in *ICML Workshop on Structural Knowledge Transfer for Machine Learning*, 2006.

[93] S. Jodogne and J. H. Piater, "Task-driven discretization of the joint space of visual percepts and continuous actions," in *European Conference on Machine Learning*, 2006, pp. 222-233.

[94] S. M. Kakade, "On the sample complexity of reinforcement learning," PhD dissertation, University of London, 2003.

[95] L. Torrey, J. Shavlik, T. Walker, and R. Maclin, "Relational macros for transfer in reinforcement learning," in *International Conference on Inductive Logic Programming*, 2007, pp. 254-268.

[96] D. Mellor, "A learning classifier system approach to relational reinforcement learning," PhD dissertation, School of Electrical Engineering and Computer Science, The University of Newcastle, Australia, 2008.

[97] S. Girgin, F. Polat, and R. Alhajj, "Improving reinforcement learning by using sequence trees," *Machine Learning,* vol. 81, pp. 283-331, 2010.

[98] T. Lang, M. Toussaint, and K. Kersting, "Exploration in relational worlds," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2010, pp. 178-194.

[99] P. Nguyen, P. Sunehag, and M. Hutter, "Feature reinforcement learning in practice," in *European Workshop on Reinforcement Learning*, 2011, pp. 66-77.

[100] J. Piater, S. Jodogne, R. Detry, D. Kraft, N. Krüger, O. Kroemer*, et al.*, "Learning visual representations for perception-action systems," *The International Journal of Robotics Research,* vol. 30, pp. 294-307, 2011.

[101] M. T. Spaan, "Partially observable Markov decision processes," in *Reinforcement Learning*, ed: Springer, 2012, pp. 387-414.

[102] G. Konidaris, I. Scheidwasser, and A. Barto, "Transfer in reinforcement learning via shared features," *Journal of Machine Learning Research,* vol. 13, pp. 1333-1371, 2012.

[103] J. MacGlashan, "Multi-source option-based policy transfer," PhD dissertation, University of Maryland, Baltimore County, 2013.

[104] E. Brunskill and L. Li, "Sample complexity of multi-task reinforcement learning," in *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence*, 2013, pp. 122-131.

[105] Y. Yu, P.-F. Hou, Q. Da, and Y. Qian, "Boosting nonparametric policies," in *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, 2016, pp. 477-484.

[106] M. Hashemzadeh, R. Hosseini, and M. N. Ahmadabadi, "Exploiting generalization in the subspaces for faster model-based learning," *arXiv preprint arXiv:1710.08012,* 2017.

[107] E. B. Baum and I. Durdanovic, "Evolution of cooperative problem solving in an artificial economy," *Neural Computation,* vol. 12, pp. 2743-2775, 2000.

[108] R. Sedgewick and K. Wayne, *Algorithms*: Addison-Wesley Professional, 2011.

[109] S. Sarjant, B. Pfahringer, K. Driessens, and T. Smith, "A Direct Policy-Search Algorithm for Relational Reinforcement Learning," in *International Conference on Inductive Logic Programming*, 2013, pp. 76-92.

[110] M. Langlois and R. H. Sloan, "Reinforcement learning via approximation of the Q-function," *Journal of Experimental & Theoretical Artificial Intelligence,* vol. 22, pp. 219-235, 2010.

[111] J. Ramon, K. Driessens, and T. Croonenborghs, "Transfer learning in reinforcement learning problems through partial policy recycling," in *European Conference on Machine Learning*, 2007, pp. 699-707.

[112] M. Irodova and R. H. Sloan, "Reinforcement learning and function approximation," in *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference (FLAIRS-2005)*, 2005.

[113] T. M. Mitchell, *Machine Learning*. New York, NY: McGraw-Hill, Inc., 1997.

Vita

Harry Paul Moore, born in Fort Bragg, North Carolina, worked as an engineer for several years in Louisiana after receiving his master's degree from Louisiana State University. During his career at a large petrochemical company in Baton Rouge, he was responsible for controlling a large manufacturing facility and became interested in making computers think like humans. His interest in the fascinating field of artificial intelligence grew to the point where he decided to return to school and pursue a doctoral degree in computer science. Upon completion of his doctoral degree, he will continue his quest for a "thinking machine".