UTILIZING NEGATIVE POLICY INFORMATION TO ACCELERATE REINFORCEMENT LEARNING

A Thesis
Presented to
The Academic Faculty

by

Arya John Irani

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Interactive Computing

Georgia Institute of Technology
May 2015

UTILIZING NEGATIVE POLICY INFORMATION TO ACCELERATE REINFORCEMENT LEARNING

Approved by:

Charles L. Isbell Jr., Committee Chair
School of Interactive Computing
Georgia Institute of Technology

Karen M. Feigh
School of Aerospace Engineering
Georgia Institute of Technology

Andrea L. Thomaz
School of Interactive Computing
Georgia Institute of Technology

Doina Precup
School of Computer Science
McGill University

Mark Riedl
School of Interactive Computing
Georgia Institute of Technology

Date Approved: April 6, 2015

ACKNOWLEDGEMENTS

One consequence of my long tenure at Georgia Tech has been the opportunity to get to know a large number of exceptional people. I'm thankful for tremendous support from some particularly exceptional people: Melissa Eriksen, Peng Zang, Jeremy Brudvik, Misha Novitzky, David Roberts, Chip Mappus, Sheila Isbell, Karthik Raveendran, Drew Taylor, Raffay Hamid, Maya Cakmak, Svetlana Yarosh, Chris and Caroline Simpkins, Matt Bonner, Heather Muñoz.

A huge thank you to my thesis committee: Karen Feigh; Mark Riedl; Doina Precup; my advisor Charles Isbell, a gentleman and a scholar, who gave me the opportunity to think outside the box; and my adoptive advisor Andrea Thomaz, who helped me see it's also quite a good idea to stop and focus on the box long enough to get a bow onto it.

Thank you to PFUNK gen2 and gen3 members "Luisca" Cobo Rus, Jon Scholz, Kaushik Subramanian, Ashley Edwards, Himanshu Sahni, Jesse Rosalia, Ryan Curtin, Pushkar Kolhe; and to coworkers John Matthews, Brad Lee, Stefan Roth, Stephen Lee-Urban, David Skoog, Lance Gatlin, Jack Wallace, Anwar Reddick, Margaret Loper, Betty Whittaker, Miriam Pierce, Ron Bohlander, Matt Moyer, Mark Kindl, Webb Roberts.

Thank you to Dana Randall (my first friend at Georgia Tech); my first advisor, Thad Starner; my second advisor Tucker Balch; my AAAI mentor Manuela Veloso, Mike Stilman, Elaine May Huang, Kate Wagner, Bradley Knox, G. J. Halfond, Catherine Grevet, Hilary Nichols, Kenny St. Clair, Betsy Richter, Matt Conley, Christine Westberg, Mark Nelson, Christina Strong, Zsolt Kira, Colin Emerson, Kalin Ambrose, Eric Caspary, Chris Lee, Carlos Nieto, Sooraj Bhat, Julie Kientz, Shan Shan Huang, Steve Webb, Tammy Clegg, George Baah, Monirul Sharif, Grant Schindler, Valerie and Jay Summet, Dan Ashbrook, Tracy Westeyn, Helene Brashear, David Minnen, Ben Wong, Marty McGuire, Keith O'Hara, Nick DePalma, Maithilee Kunda, Josh Jones, Mike Helms, Josh Zaritsky, Santi Ontañón, Attila-Giovanni Gabor, Corey Myers, Chad Stolper, Jessica Pater, Maia Jacobs,

Mark Luffel, Alex Trevor, Crystal Chao, Martin Levihn; CJJT's Jeremy Lafreniere, Iman Castañeda; TRT's Roberto Traven, Craig Shelton, Chris Sorensen, Creed Perry, Gabelaia Levan, Cameron Ayer, Pete Snider, and Ron Demeritt.

Thank you for support from afar from: Aren Worley, Jessica Takayama, Dawn Velasquez, Jenny Lee, Chris and Nohea Runnells, Ben Berkey, Nathan Albin, Kelly Hutson, Abe Robison, LeJenna Wilton, Tiffany Anderson, Marcelo Jimenez, Carole Miura, Judith Gersting, Eric Jeschke, Pi Chun Chuang, Sherryll Mleynek, Robert Fox, Bill Chen, Nancy Schein, Bert Sambueno, Birch and Ruth Robison, Michael Lukson, Mel Manuel, Daniel Churach, Masashi Kishimoto, Rob Norris, Stephen Compall, and Randall Schulz, Erik Osheim, Pedro Furla, and Tony Morris.

Still, I'm probably forgetting someone. I'm sorry and I love you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF PROGRAM LISTINGS

SUMMARY

A pilot study by Subramanian et al. [33] on Markov decision problem task decomposition by humans revealed that participants break down tasks into both short-term subgoals with a defined end-condition (such as "go to food") and long-term considerations and invariants with no end-condition (such as "avoid predators"). In the context of Markov decision problems, behaviors having clear start and end conditions are well-modeled by an abstraction known as options [27], but no abstraction exists in the literature for continuous constraints imposed on the agent's behavior.

We propose two representations to fill this gap: the state constraint (a set or predicate identifying states that the agent should avoid) and the state-action constraint (identifying state-action pairs that should not be taken). State-action constraints can be directly utilized by an agent, which must choose an action in each state, while state constraints require an approximation of the MDPs state transition function to be used; however, it is important to support both representations, as certain constraints may be more easily expressed in terms of one as compared to the other, and users may conceive of rules in either form.

Using domains inspired by classic video games, this dissertation demonstrates the thesis that explicitly modeling this negative policy information improves reinforcement learning performance by decreasing the amount of training needed to achieve a given level of performance. In particular, we will show that even the use of negative policy information captured from individuals with no background in artificial intelligence yields improved performance.

We also demonstrate that the use of options and constraints together form a powerful combination: an option and constraint can be taken together to construct a constrained option, which terminates in any situation where the original option would violate a constraint. In this way, a naïve option defined to perform well in a best-case scenario may still accelerate learning in domains where the best-case scenario is not guaranteed.

CHAPTER I

INTRODUCTION

Because it's widely understood that people have insight into many tasks we might like to automate, it's become popular in the past decade to look for ways to utilize those human insights in reinforcement learning. A pilot study by Subramanian et al. [33] on Markov decision problem task decomposition by humans revealed that participants break down tasks into both short-term subgoals with a defined end-condition (such as "go to food") and long-term considerations and invariants with no end-condition (such as "avoid predators"). In the context of Markov decision problems, behaviors having clear start and end conditions are well-modeled by an abstraction known as options [27], but no abstraction exists in the literature for continuous constraints imposed on the agent's behavior.

We propose two representations to fill this gap: the state constraint (a set or predicate identifying states that the agent should avoid) and the state-action constraint (identifying state-action pairs that should not be taken). State-action constraints can be directly utilized by an agent, which must choose an action in each state, while state constraints can only be used given some approximation of the MDPs state transition function; however, it is important to support both representations, as certain constraints may be more easily expressed in terms of one as compared to the other, and users may conceive of rules in either form.

This dissertation describes work in domains inspired by classic video games, demonstrating that the use of constraints from people does provide improved learning efficiency. We also demonstrate that the use of options and constraints together form a powerful combination: an option and constraint can be composed to produce a constrained option, which terminates in any situation where the original option would violate a constraint. In this way, a naïve option defined to perform well in a best-case scenario may still accelerate learning in domains where the best-case scenario is not guaranteed.

This dissertation demonstrates the following thesis:

> Explicitly modeling negative policy information accelerates reinforcement learning by decreasing the number of learning episodes needed to reach a given level of performance.
>
> In particular, even the use of negative policy information captured from individuals with no background in artificial intelligence yields improved reinforcement learning performance.

The rest of this document is organized as follows: Chapters 2 and 3 provide background information about reinforcement learning and state representations. In Chapter 4, we discuss related work and revisit our thesis statement and contributions with this greater context. We elaborate our solution in Chapters 5 and 6, and describe our experimental domains, procedures, and results in Chapters 7 through 10. Finally, Chapter 11 presents conclusions and avenues for future work.

CHAPTER II

REINFORCEMENT LEARNING

Reinforcement Learning (RL) [39] is a general learning framework with roots in behavioral science. The learner, or *agent*, interacts with an *environment* at sequential times $t = 0, 1, 2, \ldots$ by observing the environment state s_t and performing an action a_t of its choosing. The action results in some numerical reward r_{t+1} and an altered environmental state s_{t+1}. The agent will observe the new state and act again. This cycle is illustrated in Figure 1. In the general case, the cycle will continue forever; the agent's goal is to learn to maximize the reward it receives.

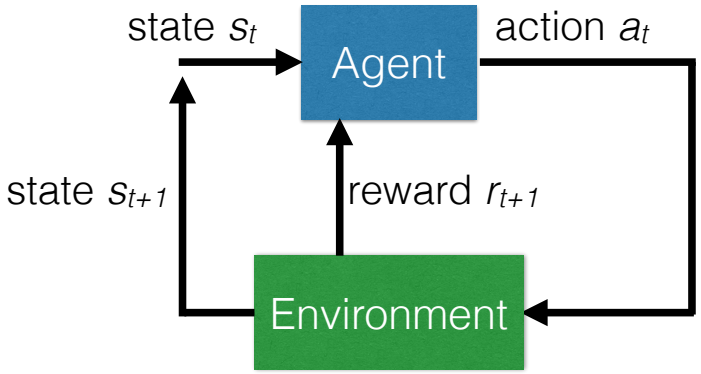


Figure 1: The reinforcement learning framework loop.

Example 1: Playing chess The state includes the positions, color, and types of each of the pieces on the board. The agent receives +1 reward for choosing a move (action) which wins the game, −1 reward for choosing a move that causes it to lose before its next turn, and 0 reward at all other times. After the agent chooses its move, the resulting state reflects its own move as well as the opponent's.

Example 2: Commodities trading The state represents the current price of gold. At each time-step, the agent can choose to buy an ounce of gold, sell one, or neither. The reward at each time-step is equal to the money spent (negative reward) or earned (positive

reward), if any. The agent is thus incentivized to buy low and sell high.

Example 3: Solving a maze The state includes the agent's position in a rectangular maze with a single exit. At each time-step, the agent can choose to move in any direction not blocked by an obstacle. It receives -1 reward for each time-step it spends inside the maze, and receives 0 reward once outside of it. By acting to maximize the reward received, the agent is incentivized to exit the maze as quickly as possible.

2.1 Markov Decision Processes

Reinforcement learning problems are often framed in terms of Markov decision processes (MDPs). Markovian systems (including Markov decision processes) are characterized as being "memory-less", meaning that all the information needed to predict an outcome or to make an optimal decision, including any necessary history, is encapsulated within the current state; knowing the history of past states does not provide any additional relevant information.

A Markov decision problem is defined by a tuple $\langle S, A_s, T, R, \gamma \rangle$, where S is a set of possible states that the agent can find itself in. A_s is the set of actions an agent can take in state s, subject to $(\forall s) A_s \neq \emptyset$. T is the transition probability function, specifying the probability of ending up in state s' when taking action a from state s:

$$T(s, a, s') = \Pr(s_{t+1} = s' | s_t = s, a_t = a) \tag{1}$$

If $T(s, a, s') = 1$ for exactly one s' for each (s, a) and 0 for all other s', the environment is called *deterministic*; otherwise it is called *stochastic*.

The reward function $R(s, a, s')$ gives the expected immediate reward received when transitioning from s to s' via a. Reward functions which mostly evaluate to zero (as in our chess example) or are otherwise uniform (as in our maze example) are considered *sparse*; most actions appear equally good or bad. The agent won't really learn anything until it observes some differences in reward; it's only at this point that the agent can begin trying to maximize its likelihood of continuing to receive improved reward.

Some states may be *terminal*, meaning that the chess game has ended, the agent has escaped the maze, or that for whatever reason, future actions are irrelevant. Although the MDP's $\langle S, A_s, T, R, \gamma \rangle$ tuple doesn't explicitly designate terminal states, they can be modeled as inescapable states that yield only zero-valued future rewards.

The discount factor $\gamma \in [0, 1]$ is a multiplier by which future anticipated rewards are discounted when choosing the best action at the current time. If $\gamma = 1$, a reward in the distant future is exactly as desirable as an immediate reward of the same magnitude—the chess-playing agent would have no incentive to win quickly, so long as it does win eventually. If $\gamma = 0$, rewards that are not immediate should not taken into consideration when making a decision—if the chess-playing agent cannot "mate in one", then all moves will appear equally bleak to it.

In the context of MDPs, a reinforcement learning agent's goal is to maximize the long-term discounted reward it receives while acting in environment defined by that MDP. It does so by estimating, through experience or other information sources, which available actions will most likely lead to reward in the long term. The MDP's γ dictates how long-term the agent's planning should be. This problem of determining which near-term actions will lead to the greatest reward over time is called the *temporal credit assignment* problem: Choosing any particular move on this turn will affect the chess-playing agent's probability of collecting reward (by winning) later in the game. Which move will create the most favorable future situation?

An agent's behavior can be defined by a Markov policy $\pi : S \times A_s \to [0, 1]$, which specifies the probability that the agent will choose a certain action when in a given state (Equation 2).

$$\pi(s, a) = \Pr(a_t = a | s_t = s) \tag{2}$$

Every policy π is subject to $(\forall s) \sum_{a \in A_s} \pi(s, a) = 1$, and so where notationally convenient, this dissertation will also use the equivalent formulation for specifying probabilities, $\pi : S \to \text{Dist}[A_s]$, a mapping from state to probability distribution over the set of actions.

If the chess-playing agent's first move is always to advance a uniformly arbitrary pawn by one position, then $\pi(\texttt{newGame}, \texttt{advancePawn}_i) = 1/8$, where $\texttt{advancePawn}_i$ is an action

representing advancing the ith of eight pawns.

Given a policy, we can define the *state value function* (often *value function* or *utility function*), $V^\pi : S \to \mathbb{R}$, which gives the expected discounted long-term reward when following π, using the Bellman equation:

$$V^\pi(s) = \sum_{a \in A_s} \pi(s,a) \sum_{s' \in S} T(s,a,s') \left[R(s,a,s') + \gamma V^\pi(s') \right]$$

Essentially, the utility of being in a certain state s is an expectation which depends on the action the agent will choose (with probability $\pi(s,a)$), the new state the action will lead to (with probability $T(s,a,s')$), the reward $R(s,a,s')$ that will be received on entering the state, plus the discounted utility of that new state, which itself is defined in the same way: dependent on the possible future outcomes. In the chess example, with its simple reward schedule of $+1$ for a win, -1 for a loss, and 0 at all other times, if we assume a discount factor $\gamma = 1$, $V^\pi(s)$ represents the probability of winning minus the probability of losing, if following policy π from the current board configuration. For example, if based on the current board state s, the agent's policy π has a probability of winning of 0.3, a probability of losing of 0.5 and a probability of a ending in a draw of 0.2, the utility would be

$$\begin{aligned} V^\pi(s) &= 0.3 \cdot r_{win} + 0.5 \cdot r_{lose} + 0.2 \cdot r_{draw} && (3) \\ &= 0.3 \cdot (+1) + 0.5 \cdot (-1) + 0.2 \cdot 0 && (4) \\ &= 0.2 && (5) \end{aligned}$$

If the policy and state are such that the agent is guaranteed to win with its next move, $V^\pi(s) = R(s,a,win) + \gamma V^\pi(win) = +1$ where a is the winning move dictated by π, *win* is a terminal winning state, $R(s,a,win) = +1$, and $V^\pi(win) = 0$. The reinforcement learning agent should choose actions to maneuver itself into states with the greatest utility.

We can similarly define the *state-action value function* or *Q function*

$$\begin{aligned} Q^\pi(s,a) &= \sum_{s' \in S} T(s,a,s') \left[R(s,a,s') + \gamma V^\pi(s') \right] && (6) \\ &= \sum_{s' \in S} T(s,a,s') \left[R(s,a,s') + \gamma \sum_{a'} \pi(s',a') Q^\pi(s',a') \right] && (7) \end{aligned}$$

which specifies the value of taking action a in state s and following policy π thereafter. You can see that definition of Q is just the same as the definition of V, except that the first action has been pre-determined.

The *optimal Q function* is denoted by Q^*:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \tag{8}$$

This is the maximum possible value for Q under any policy: the expected long-term discounted reward, given that action a is used now in state s, and the best possible actions are taken subsequently. Similarly, the optimal state value function is denoted by V^*:

$$V^*(s) = \max_\pi V^\pi(s) \tag{9}$$

$$= \max_{a \in A_s} Q^*(s,a) \tag{10}$$

It represents the expected long-term discounted reward given that the agent is in state s and chooses the best possible actions now and going forward. For any given state s, an action a is considered optimal if

$$a \in \arg\max_a Q^*(s,a) \tag{11}$$

The solution to an MDP is any optimal policy π^*, which maximizes the value function for every state, by only taking optimal actions:

$$\pi^*(s,a) > 0 \iff a \in \arg\max_{a \in A_s} Q^*(s,a) \tag{12}$$

By definition, this policy will yield the highest expected discounted long-term reward.

2.2 Q Learning

Q learning [39] is an algorithm for approximating Q^* without explicitly modeling the S, A_s, T, or R components of the MDP. The Q learning algorithm maintains an approximation of the Q-function, and the approximation is updated with each new experience. Q learning is called an *off-policy* algorithm, because it doesn't dictate what an agent's policy should be; it is simply a method for learning the Q function based on whatever the agent's choices

```
type Utility = Real
type Probability = Real

// Probability distribution over As
trait Distribution[A] {
  def pdf(a: A): Probability
  def sample: A // draw an A from the distribution
}

// The interface to our environment
trait GenerativeModel[S,A] {
  def actions(s: S): Set[A]
  def isTerminal(s: S): Boolean
  def sample(s: S, a: A): (Reward,S)
}

// Using representation-specific helpers 'readQ' and 'writeQ',
// update a QFunc, given experience from taking action 'a' from state 's'
def qUpdate[QFunc,S,A](qfunc: QFunc,
                       experience: (S,A,Reward,S)
                       nextActions: NonEmptySet[A],
                       learnRate: Real,
                       discount: Real)
                      (implicit
                       readQ: QFunc => S => A => Utility,
                       writeQ: S => A => Utility => QFunc => QFunc
                      ): QFunc = {
  val (s, a, r, s2) = experience
  val oldQValue = readQ(qfunc)(s)(a)
  val learnedQValue = r + discount * nextActions.maxBy(readQ(qfunc)(s2))
  val newQValue = learnRate * learnedQValue + (1-learnRate) * oldQValue
  writeQ(s)(a)(newQValue)(qfunc)
}

// Starting in state 's', loop and continue performing Q-learning until
// reaching a terminal state; then return the resulting QFunc.
def qlearningTrial[QFunc,S,A](s: S,
                              qfunc: QFunc,
                              explorationPolicy: QFunc => S => Distribution[A],
                              g: GenerativeModel[S,A],
                              discount: Real)
                             (implicit
                              getLearnRate: S => A => Real,
                              readQ: QFunc => S => A => Utility,
                              writeQ: S => A => Utility => QFunc => QFunc
                             ): QFunc = {
  if (g.isTerminal(s))
    qfunc
  else {
    val a = explorationPolicy(q)(s).sample
    val (r,s2) = g.sample(s,a)
    val updatedQfunc = qUpdate(q, s, a, r, s2, g.actions(s2), learnRate(s)(a), discount)
    qlearning(s2, updatedQfunc, explorationPolicy, g, discount)
  }
}
```

Listing 1: A simplified version of the Q learning algorithm.

and experiences happen to be. We will discuss some basic ways that a reinforcement learner can utilize Q learning, in Section 2.4.

The specific implementation of a Q learning algorithm will vary depending on the representation of the Q-function approximation, but each involves a Bellman-style update, which replaces a fraction of the current Q value estimate with a newly estimated value:

$$\underbrace{Q_{t+1}(s_t, a_t)}_{\text{new value}} := \underbrace{(1 - \alpha_t(s_t, a_t)) \times Q_t(s_t, a_t)}_{(1-\text{learning rate})\times\text{old value}} + \underbrace{\alpha_t(s_t, a_t)}_{\text{learning rate}} \times \underbrace{\left[r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a)\right]}_{\text{newlyestimatedvalue}} \quad (13)$$

A high-level algorithm emphasizing the Q function-representation-specific components `readQ` and `writeQ` is given in Scala in Listings 1.

2.3 Generative Models

Although the domains used in this dissertation are well-modeled by MDPs, we do not assume we know the MDP. In particular, we don't assume that we can enumerate all possible states. We will assume we have at least a *generative model* or simulator as described by Kearns et al. [14] to sample transitions from the underlying process, as this is the bare minimum capability needed to perform Q learning.

> Informally, [the Generative Model] is a "black box" to which we can give any state-action pair (s, a), and receive in return a randomly sampled next state and reward from the distributions associated with (s, a). Generative models have been used in conjunction with some function approximation schemes [34], and are a natural way in which a large MDP might be specified. Moreover, they are more general than most structured representations, in the sense that many structured representations (such as factored models [4, 24, 17]) usually provide an efficient way of implementing a generative model. Note also that generative models also provide less information than explicit tables of probabilities, but more information than a single continuous trajectory of experience generated according to some exploration policy, and so we view results obtained via generative models as blurring the distinction between what is typically called "planning" and "learning" in MDPs. —Kearns et al. [14]

In addition to the ability to sample next-states from the environment, the generative model gives us the set A_s of available actions, as well as a test to determine whether a particular state is a terminal state. The programmatic interface for the generative model is included in Listing 1.

2.4 The Exploration vs Exploitation Trade-off in Q Learning

Although the Q learning is a passive algorithm, a reinforcement learning agent can use Q learning to maximize its expected long-term reward. Recall that a *policy* dictates the probabilities with with an agent chooses actions. Given some estimated Q function $\hat{Q}$, a *greedy* policy chooses the action that the Q function indicates will yield the greatest expected long-term reward:

$$\pi_{\text{greedy}}(\hat{Q}, s) = \mathcal{U}\left(\arg\max_{a \in A_s} \hat{Q}(s, a)\right) \tag{14}$$

where the arg max operator yields a set and $\mathcal{U}(S)$ represents the uniform distribution over a set S. This policy of choosing the best action according to immediately available knowledge is referred to as *exploitation* of existing knowledge.

When $\hat{Q} = Q^*$, the greedy policy is an optimal policy, as given in Equation 12. When $\hat{Q} \neq Q^*$, the greedy policy generally will not be an optimal policy; in this case, the Q learner needs more experience to improve its Q function estimate. Taking the greedy action at each opportunity will provide the Q learner with data to improve its estimated Q value for that greedy action, but it won't necessarily do anything to help improve the estimated Q values for the other actions — even though it may be the case that one of these other actions *is* in fact the optimal action that the agent should ideally be using. To discover whether one of these actions, believed to be suboptimal according to the agent's limited past experience, is in fact an optimal action, the Q learning algorithm needs to see some data resulting from the use of that action. Taking these suspected-suboptimal actions to learn about their potentially great outcomes is referred to as *exploration*.

Now the agent (or the agent's designer) is faced with a difficult choice: Is it better to stick with the best known alternative (*exploiting*) or is it better to try something new (*exploring*), knowing that the outcome of exploring might easily be lousy? Is it a good idea to learn

by making mistakes? Choosing between these conflicting goals, information-gathering vs best-known performance, is known as the *exploration vs exploitation trade-off*.

A standard compromise that we will utilize is the *ϵ-greedy* policy, which exploits existing knowledge most of the time, but explores a fraction of the time as well. The ϵ-greedy policy explores with probability ϵ by choosing an action uniformly from the set of available actions, and exploits existing knowledge by choosing a greedy action with probability $1 - \epsilon$. Equations 15–16 define an ϵ-greedy policy by scaling the uniform and greedy policies by a factor of ϵ and $1 - \epsilon$ respectively:

$$\pi_{\epsilon\text{-greedy}}(\hat{Q}, s) = \epsilon \cdot \mathcal{U}(A_s) + (1 - \epsilon)\pi_{\text{greedy}}(\hat{Q}, s) \tag{15}$$

$$= \epsilon \cdot \mathcal{U}(A_s) + (1 - \epsilon)\,\mathcal{U}\left(\arg\max_{a \in A_s} \hat{Q}(s, a)\right) \tag{16}$$

The down side of using such a policy of course, is that if $\hat{Q}$ is sufficiently accurate, presumed-suboptimal actions will in fact be truly-suboptimal actions, and using them will needlessly hurt the agent's performance. The goal of reinforcement learning is to drive the probability of taking suboptimal actions to zero over time, and a standard strategy for accomplishing this is to decrease or *decay* ϵ in such a way that the agent can initially explore enough to get an acceptable approximation of the Q function, and then continue to perform well with little or no further exploration.

CHAPTER III

STATE REPRESENTATION

In the previous chapter, we discussed the agent's task of finding and utilizing the best action given the environmental state the agent finds itself in. It's worth understanding that the particular representation of the state determines how it will be presented to the agent, and what specific information (or *features*) might be available from it. In this chapter, we describe three standard classes of state representation and discuss the practice of feature engineering in AI systems and its relationship to our work.

3.1 Atomic State Representations

Like the name implies, an *atomic* state representation does not contain any deeper structure. (Or if it does contain some deeper structure, the structure is not exposed and cannot be utilized by the agent.) This is the most abstract state representation possible. We only assume the ability to tell whether or not two states are identical, since without this capability, the agent could not recall anything it had learned. A Q function estimate over atomic states can be represented by an hash-map, tree-map, array, or association list, depending on whether the state representation is hashable, orderable, mappable to a small unique non-negative integer, or none of the above, respectively. These are all loosely referred to as Q *tables*. Taking the Q learning algorithm in Listing 1 together with the read/write implementations in Listing 2 yields a *tabular* Q *learner* over atomic states.

3.2 Factored State Representations

A *factored* state representation is a tuple of scalar feature values $(f_1, f_2, ..., f_n)$. Each feature may have a Boolean, integer, real number, or other scalar type. It is common to design a state representation as a fixed-length vector of real numbers because this vector can then be used directly as numerical parameters to an approximator used by the agent. For example, many robots use laser range finders to "see" their surroundings. These range finders produce

```
type MapBackedQFunc[S,A] = Map[(S,A),Utility]

val defaultQValue: Utility = 0.0

def readQ[S,A]: MapBackedQFunc[S,A] => S => A => Utility =
  qfunc => s => a => m.getOrElse((s,a), defaultQValue)

def writeQ[S,A]: S => A => Utility => MapBackedQFunc[S,A] => MapBackedQFunc[S,A] =
  s => a => newQ => oldQfunc => oldQfunc.updated((s,a), newQ)
```

Listing 2: Reading and writing a Q function represented by a map from state-action pairs to Q values. The write operation delegates to the underlying map.

a vector in which each element represents the distance from the sensor to the nearest object at a given bearing relative to the range finder.

As compared to tabular Q learning, function approximation schemes (which are enabled by the use of these factored state representations) have a much greater capacity for generalization in learning [11], meaning that knowledge gained about one situation (e.g., cookies are delicious at home) can suggest knowledge about another situation (e.g., cookies are delicious in the car); in tabular Q learning, each state is considered to be completely unrelated to the others.

Taking the Q learning algorithm in Listing 1 together with the read/write implementations in Listing 3 yields a Q learner for a (much more compact) linear Q function approximation; however, a factored state representation can be treated as an atomic state representation, to learn an exact solution as previously described.

3.3 Structured (Domain-Specific) State Representations

Structured state representations contain an arbitrary combination of scalars, lists, vectors, trees, sets, relations, etc. For example, a gift-buying agent's state may consist of a person's set of hobbies, a list of $(gift, date, rating)$ triplets, and a mapping between gifts and costs, as in Listing 4.

Most real-world problems fall into this unrestricted category, but they can also be viewed trivially as atomic states; or they can be viewed as factored states, given some appropriate projection. For example, a set of hobbies can be viewed as a Boolean-valued vector, where

```
type LinApproxWeights = NonEmptyVector[Double]

// Produces a set of features from a given type of state and action
trait SAFeatExtractor[S,A] {
  def features: S => A => NonEmptyVector[Double]
}

def readQ[S,A](implicit fe: SAFeatExtractor[S,A]): LinApproxWeights => S => A => Utility =
  weights => s => a => {
    val features: NonEmptyList[Double] = fe.features(s)(a)
    (weights * features).sum // element-wise product, and then sum
  }

def writeQ[S,A](implicit fe: SAFeatureExtractor[S,A])
  : S => A => Utility => LinearApproxWeights => LinearApproxWeights =
    s => a => newQ => weights => {
      val features: NonEmptyVector[Double] = fe.features(s)(a)
      val oldQ: Utility = (weights * features).sum
      val deltaQ: Utility = newQ - oldQ

      // scalar * vector multiplication, and then element-wise sum
      weights + deltaQ * features
    }
```

Listing 3: Reading and writing a Q function represented by a list of linear function approximation weights.

```
case class GiftAgentState(hobbies: Set[Hobby],
                          ratings: List[(Gift,Date,Rating)],
                          costs: Map[Gift,Price])
```

Listing 4: A sample structured state representation.

each element corresponds to a particular hobby's presence in the underlying set. A projection from arbitrarily-structured representation to factored representation is called *feature extraction*, and it is an important first step for any learner which must use function approximation due to time and resource constraints; however, function approximation introduces an additional source of error [41].

3.4 Feature Engineering

Many learning algorithms only support a specific, restricted class of state representations. If the available state representation is not supported by a learning algorithm, then an AI expert must either select a different algorithm or derive a supported state representation from the available one. Additionally, an AI expert may alter an existing state representation

in an attempt to make the agent's learning task simpler.

In a live domain, the environment is the real world, and there is no "true" underlying digital representation.[1] An AI engineer uses a combination of raw sensor data and *feature extractors* (data transformations resulting in new synthetic features to supplement or replace raw sensor data) to produce a state representation he feels is suitable for the agent to learn with. By contrast, in a simulated, software-based domain, the simulated environment itself uses a particular state representation, generally a structured representation. The engineer may choose to simplify this structured representation by removing components, or augment it with the output of feature extractors, ultimately producing a factored state representation for function-approximation based Q learning, an atomic state representation for tabular Q learning, or a structured representation for use with a domain-specific learning algorithm.

These feature extractors can be quite sophisticated, and thus, the skill (or intuition) of the feature engineer has a tremendous impact on the overall effectiveness of the AI system, and thus one of our goals is to develop techniques which can improve performance without requiring users to have advanced degrees in artificial intelligence. Towards this goal, although the simulations used in this dissertation (described in detail in Chapter 7) use a structured state representation to simulate the environment, we only expose an atomic state representation to the Q learner. We deliberately avoid engineering a factored state representation for our domains, both to avoid attribution of our improved performance to feature-engineering by AI experts as well as to side-step negative issues inherent in function approximation [41].

[1] If the agent's sensors are taken as given, the raw sensor data may be viewed as the "true" underlying digital representation.

CHAPTER IV

PROBLEM DECOMPOSITION WITH INPUT FROM HUMANS

There have been a number of prior works which aim to reduce learning complexity through problem decomposition; through the use of human input to the learning process; or, like the work in this dissertation, through the use of human input for problem decomposition. This chapter discusses several past works which inspired ours.

We'll first discuss *options*, a powerful and popular abstraction for problem decomposition in reinforcement learning; highlight several prior examples of using human input for reinforcement learning (e.g., using human input for learning options); describe the motivation for our novel representation; and present our thesis statement and summary of contributions.

4.1 Problem Decomposition with Options

Options [35, 27] are a hierarchical abstraction for actions in reinforcement learning. Whereas the Markov policies we have considered so far produce a single *primitive* action to be taken in the current time-step, an option can specify actions to be taken over the next several time-steps. The option embeds its own full-fledged policy which the agent will follow until a given option termination criterion is met.

Options offer a means to specify a temporal task decomposition, e.g., "First perform subtask x, then perform subtask y, ..." Each option produces sequence of actions, the options themselves can be taken in sequences, just as you would think of taking sequences of primitive actions. If the agent's environment can be characterized as task-oriented, and if the task consists of a number temporally-disjoint subtasks, it is possible to learn or construct policies for those subtasks independently, construct options from each of the resulting policies, then learn a high-level policy over those options to solve the original problem. Options also provide transfer of learning, because an option for a particular subtask may be applicable to a broad range of higher-level tasks.

An option itself is defined by a tuple $o = \langle \mathcal{I}, \pi, \beta \rangle$, where $\mathcal{I}$ is an *initiation set* of states from which the option can be initiated, $\pi : S \to \text{Dist}[A]$ is the embedded, stochastic policy over low-level actions, and $\beta : S \to [0, 1]$ encodes the termination condition and represents the probability that the option will terminate upon reaching a particular state. Primitive actions can be lifted into options which take the underlying action and terminate immediately. An agent can then use an exploration policy over options $\pi : S \to \text{Dist}[O]$ to learn a Q function over options ($Q : S \times O_s \to \mathcal{R}$) or over low-level actions ($Q : S \times A_s \to \mathcal{R}$).

Learning with options has been studied in the literature within the field of Hierarchical Reinforcement Learning as semi-Markov decision processes [35]. Options can be automatically extracted under certain conditions [23, 31, 22, 32] and have been used to plan efficiently in complex problems [18]. Good options provide a coherent sequence of decisions for the agent; without them, the agent needs to learn each action in an optimal sequence independently. A set of very poor options can negatively impact learning, though: each additional action (or option, in this case) adds some burden to the agent's task of exploration. The options should ideally be good enough to make up for this added learning complexity.

4.2 Pilot Study

This dissertation is motivated by a pilot study by Subramanian et al. [33], which investigated the use of human input for task decomposition. They first described the rules and dynamics of the game of Pac-Man to each participant. (If needed, see Section 7.2 for specifics of our version of Pac-Man.) The low-level actions (i.e., moving one step left, right, up, or down) were presented as *buttons* that participants could use to control the agent. Participants were allowed to play the game until they were familiar with it. Then, each participant was asked to suggest new buttons that would make winning the game "easier" and "faster". They were restricted to suggesting at most four new buttons. Table 1 summarizes the participants' responses.

Some of the proposed buttons (e.g."go to the closest pac-dot" and "eat the ghost") represent subgoals which imply a specific course of action for the agent and clear end conditions. These are easily recognizable as options.

Table 1: Human-defined task decompositions for Pac-Man.

Button Name	# of participants
Go to the closest pac-dot	10
Avoid ghost	10
Go to the nearest energizer	10
Eat the ghost	4

The "avoid ghost" button is harder to formulate in the same way. It has a narrow but hard-to-scope initiation condition—does it only apply when Pac-Man is *too* near to the ghost? Should Pac-Man alternate between normal performance and ghost-avoidance mode? No: Pac-Man should always be avoiding inedible ghosts. Because of that reality, there is also no clear terminal configuration. These characteristics (hard-to-scope initiation condition, no clear terminal condition) were confirmed during the post-experiment interviews in which the participants were asked to describe the functionality of each modified button they had constructed.

4.3 Human Input in Reinforcement Learning

There are a number of related works which use input from humans to achieve improved reinforcement learning.

Argall et al. [3] provide a broad overview of Learning by Demonstration. Human demonstrations have been used to derive reward functions [25, 1], policies [2], and state abstractions [5]. There has also been work on extracting options from human demonstrations [42, 19]. In constructing these options, we are providing policy advice; essentially saying, "Here are some examples of things that you, Agent, will want to experiment with yourself." Argall et al. [2] demonstrate a framework that incorporates both human demonstration and specific motion-control advice operators. These are examples of learning from indirect and direct positive "do" information, but none of them utilize direct negative information (what not to do). Grollman and Billard [10] use failed task demonstrations by humans for robot motion planning by building a motion model for the failed demonstrations and generating new motion trajectories dissimilar to the failure model. They are able to provide indirect negative policy information: "Do something that is very different from

what I just did."

4.4 Reward Shaping

Dorigo [8] introduced a method of incorporating feedback from humans called *reward shaping*, which has been since been used broadly as a way to provide feedback to the reinforcement learner regarding its performance. In Chapter 2, we noted that when an environmental reward function is *sparse*, the lack of information can cause the learning process to be rather unfocused. Reward shaping is the idea that we may be able to help the learning algorithm by giving it additional *shaping* rewards in the hopes of guiding it towards an optimal policy more quickly.

The shaping rewards are introduced in the form of a *shaping reward function*, $F : S \times A \times S \to \mathbb{R}$ (just like the environmental reward function R). We can apply the shaping reward function to an MDP $M = \langle S, A, T, R, \gamma \rangle$ by replacing R with $R' = R + F$, yielding a new MDP $M' = \langle S, A, T, R', \gamma \rangle$, and applying the learning algorithm to M' instead.

To reuse our chess example, the agent does not receive any direct indication that losing a queen is bad. Over many many attempts, a Q learner will eventually infer a relationship between losing its queen, and reduced likelihood of reward through winning. Reward shaping would allow us to give the agent a small immediate negative reward whenever its queen is captured; in this way, the agent would immediately know that there is something bad about losing its queen.

But this raises a potential issue: once we've used shaping rewards to more efficiently learn an optimal policy for M', is resulting policy even optimal for the original M? Maybe not. For example, Randløv and Alstrøm [29] use shaping rewards to encourage a simulated cyclist to travel towards the goal. Unfortunately, the agent learns to ride in loops (Figure 2), because this allows the agent to travel towards the goal *more* than a direct route would.

Ng et al. [26] show that in order for an optimal policy of M' to be an optimal policy of M in the general case, it is necessary and sufficient for the shaping reward function to be *potential-based*, meaning that it is expressible in the form $F(s, a, s') = \gamma\Phi(s') - \Phi(s)$,

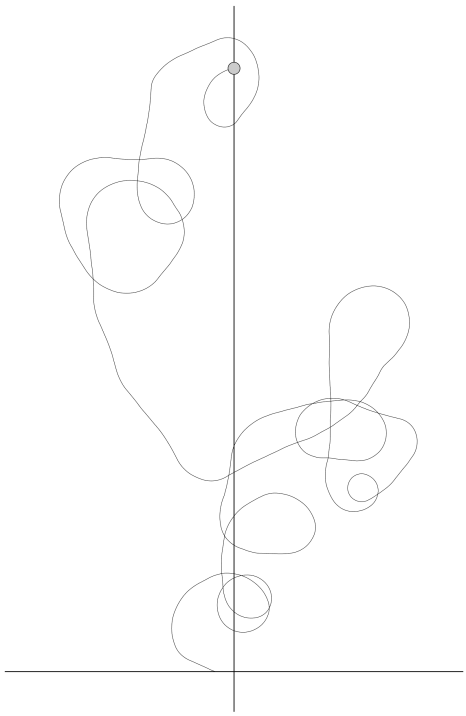


Figure 2: Reward shaping can alter the optimal policy.

dependent on a *potential* function of states $\Phi : S \to \mathbb{R}$. Wiewiora [40] notes that potential-based shaping and Q-value initialization are equivalent, and can be applied at the time the Q table is initialized, if they are already known; e.g., if it's not the case that they have yet to be collected interactively.

Reward shaping has been used successfully in contexts where an environmental reward function doesn't exist—there is no intrinsic optimal policy to deviate from—and one is instead trying to elicit a particular behavior from the agent [15] in an "I'll know good performance when I see it," kind of teaching. But as Ng et al. predict, the technique sees mixed results when evaluated against an environmental reward function [16].

4.5 A New Representation Is Needed

While positive instructions (or positive policy information) are easily described as options, negative instructions (or negative policy information) are naturally construed as *constraints* satisfying a never-ending goal. In the context of reinforcement learning, these constraints should place limitations on policies and equally on exploration in the learning task, keeping

the agent from wasting time or getting into trouble. Pac-Man's objectives can be viewed as achieving the participants' proposed options as many times as possible, while always respecting the constraints.

The "go to the closest pac-dot" option, which intuitively is a very useful behavior to have available, has a major flaw: if there is a ghost along the shortest path to the food, the option will lead Pac-Man to collide with the ghost and die. With enough exploration, the agent will learn the conditions under which it is advantageous to use the option, and under which conditions it isn't. Instead of "go to the nearest pac-dot", we would like to say "go to the nearest pac-dot *but* avoid ghosts while doing so". "Avoid ghosts" is not a difficult concept to learn, fortunately; however, learning "go to the nearest pac-dot *but* avoid ghosts while doing so" is more complex than learning both clauses in isolation. Recall that the participants in the pilot study discussed in the previous section described three potential options and one constraint ("avoid ghost"). It would be nice if we could learn "avoid ghost" just once, and apply that knowledge to each of the options, rather than needing to learn new, safe versions of each of the three options.

If optimal performance with respect to a particular environmental reward function is our goal (and in this dissertation, it is), reward shaping is not an appropriate solution for encoding policy-level information, positive or negative; therefore, a new representation is needed.

4.6 Thesis Statement and Contributions

This dissertation demonstrates the following thesis:

> Explicitly modeling negative policy information accelerates reinforcement learning by decreasing the number of learning episodes needed to reach a given level of performance.
>
> In particular, even the use of negative policy information captured from individuals with no background in artificial intelligence yields improved reinforcement learning performance.

This dissertation provides:

- A formalism for modeling negative policy information as *state constraints* and *state-action constraints* (Chapter 5);
- A method for composing an option and constraint to form a *constrained option* which will yield higher expected reward than its counterpart, provided that the constraint is not flawed (Section 6.2);
- An ϵ-greedy υ-defiant exploration strategy, which accelerates learning with accurate constraints while preserving optimality when learning with inaccurate constraints (Section 6.3.1);
- A greedy constraint-biased evaluation policy, which biases an agent to utilize knowledge embedded in a constraint, when no better information has been learned (Section 6.3.2);
- A scheme for learning state constraints in a semi-automated way, from examples constructed by laypeople (Chapter 8), and evaluation in two domains (Chapter 9);
- An evaluation of learning with software-engineered state-action constraints (Chapter 10).

CHAPTER V

CONSTRAINTS

In the previous chapter, we lamented the absence of a suitable representation with which to capture available domain knowledge about long-term policy constraints. Moreover, we would like a primitive that allows us to apply this captured domain knowledge in a concurrent way (*while* the agent does things), rather than exclusively in a sequential, *temporally disjoint* way, interleaved between other actions or tasks. This chapter defines and discusses our constraint representations, while Chapter 6 discusses how we can use constraints to achieve these goals.

5.1 State-Action Constraints

Within the context of MDPs, we define a *state-action constraint* ("actions to avoid") as a predicate:

$$c_{SA} : S \times A \rightarrow \{\texttt{Avoid}, \texttt{Unspecified}\} \tag{17}$$

indicating whether a given action should be avoided in a given state. With respect to the transition probability function $T(s, a, s')$ or reward function $R(s, a, s')$, state-action constraints only depend on the current state s and the proposed action a.

Using the example of the maze domain from Chapter 2, Example 3, one might specify:

$$c_{SA}(\texttt{(x=35,y=12)}, \texttt{West}) = \texttt{Avoid} \tag{18}$$

meaning: "When at position `(x=35,y=12)`, don't use the `West` action (because that will take you into a room with quicksand, which is difficult to get free from, and you will waste turns, all the while incurring negative reward)." (See Figure 3.)

5.2 State Constraints

We define a *state constraint* ("situations to avoid") as a predicate:

$$c_S : S \rightarrow \{\texttt{Avoid}, \texttt{Unspecified}\} \tag{19}$$

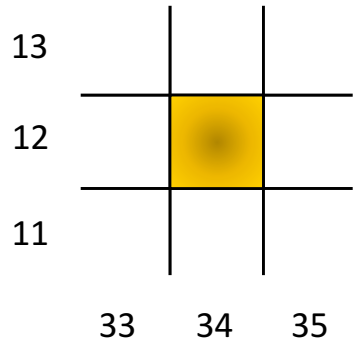


Figure 3: A small region of a maze with quicksand at `(x=34,y=12)`.

indicating that certain states are best avoided entirely, irrespective of the action or actions that may lead there. They can let an agent know that it has just made a mistake. In terms of the transition probability function $T(s, a, s')$ or reward function $R(s, a, s')$, state-action constraints only depend on the *next* state s'. If the agent has some ability to predict the effects of its actions, the state constraint can also let an agent predict when it is about to make a mistake.

An example of a state constraint would be:

$$c_S(\texttt{(x=34,y=12)}) = \texttt{Avoid} \tag{20}$$

meaning: "Avoid location `(x=34,y=12)` (because that is a room full of quicksand, and you shouldn't enter, no matter how)."

5.3 State Constraints vs State-Action Constraints

The choice of which constraint representation to use will depend on the particular constraint, with some being more easily expressed as state constraints, and others being more easily expressed as state-action constraints.

To demonstrate the tradeoff, we'll begin by assuming that the maze illustrated in Figure 3 is such that we can encode the previous state constraint avoiding position `(x=34,y=12)` using either of the two constraint representations. We see that the state constraint can be expressed more compactly by an order of magnitude:

state constraint	**state-action constraint**
$c_S(\texttt{(x=34,y=12)}) = \texttt{Avoid}$	$c_{SA}(\texttt{(x=34,y=11)}, \texttt{North}) = \texttt{Avoid}$
	$c_{SA}(\texttt{(x=34,y=13)}, \texttt{South}) = \texttt{Avoid}$
	$c_{SA}(\texttt{(x=33,y=12)}, \texttt{East}) = \texttt{Avoid}$
	$c_{SA}(\texttt{(x=35,y=12)}, \texttt{West}) = \texttt{Avoid}$

In the slightly different maze illustrated in Figure 4, there is a bridge that allows the agent to cross safely over the quicksand from the east or from the west; the only danger is in approaching from the north or south. In this case, a state-action constraint is more appropriate:

state constraint	**state-action constraint**
no equivalent	$c_{SA}(\texttt{(x=34,y=11)}, \texttt{North}) = \texttt{Avoid}$
	$c_{SA}(\texttt{(x=34,y=13)}, \texttt{South}) = \texttt{Avoid}$
	$c_{SA}(\texttt{(x=33,y=12)}, \texttt{East}) = \texttt{Unspecified}$
	$c_{SA}(\texttt{(x=35,y=12)}, \texttt{West}) = \texttt{Unspecified}$

If the state representation were rich enough to determine whether a given state poses a quicksand risk or not, a state constraint might again be the best choice. There's no compact state-action constraint equivalent because they are defined in terms of (s, a), whereas we are trying to make a statement about next-states s' having the `quicksand` property. The

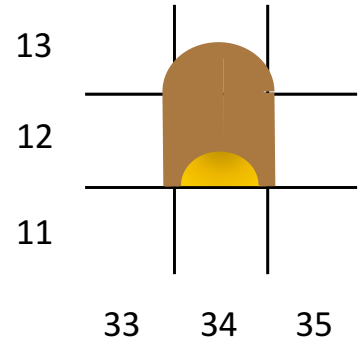


Figure 4: A small region of a maze with quicksand under a bridge at `(x=34,y=12)`.

state-action constraint equivalent would again need to enumerate all the possible ways to fail.

state constraint	**state-action constraint**
$c_S(\{s' : \texttt{quicksand=True}\}) = \texttt{Avoid}$ $c_S(\{s' : \texttt{quicksand=False}\}) = \texttt{Unspecified}$	no compact equivalent

5.4 Constructing State-Action Constraints from State Constraints

Although state constraints, specifying next-states to avoid, may offer a more compact representation for a particular concept, state-action constraints are more directly usable by an RL agent: they provide information based on current state and next action; an RL agent must make decisions based on the current state and next actions.

Given a transition probability function T or an approximation thereof, we can construct a state-action constraint c_{SA} from a state constraint c_S:

$$c_{SA}(s,a) = \begin{cases} \texttt{Avoid} & \text{if} \quad \left(\sum_{\{s' | c_S(s')=\texttt{Avoid}\}} T(s,a,s')\right) > tolerance \\ \texttt{Unspecified} & \text{otherwise} \end{cases} \tag{21}$$

This construction says to avoid a particular action if the probability of reaching any `Avoid` state when taking the action is greater than some tolerance. For example, if $tolerance = 0$, then any state-action pair with any probability of leading to an `Avoid` state will also be designated as `Avoid`. If $tolerance = 0.05$, the constructed constraint will tolerate this possibility of entering an `Avoid` state.

5.5 Composition of Constraints

Both state constraints and state-action constraints can be viewed as sets of elements for which the constraint specifies `Avoid` and as such, standard set operations can be applied to to them. For example, the union of two constraints specifies `Avoid` when either of the

component constraints do:

$$c_{\text{union}}(s,a) = (c_1 \cup c_2)(s,a) \tag{22}$$

$$= \begin{cases} \texttt{Avoid} & \text{if} \quad c_1(s,a) = \texttt{Avoid} \quad \vee \quad c_2(s,a) = \texttt{Avoid} \\ \texttt{Unspecified} & \text{otherwise} \end{cases} \tag{23}$$

and the intersection of two constraints specifies `Avoid` whenever both of the component constraints do:

$$c_{\text{intersection}}(s,a) = (c_1 \cap c_2)(s,a) \tag{24}$$

$$= \begin{cases} \texttt{Avoid} & \text{if} \quad c_1(s,a) = \texttt{Avoid} \quad \wedge \quad c_2(s,a) = \texttt{Avoid} \\ \texttt{Unspecified} & \text{otherwise} \end{cases} \tag{25}$$

Equations 22–25 define union and intersection operations for state-action constraints; they can be defined analogously for state constraints. The union of two constraints is *at least* as restrictive as its component constraints. The intersection of two constraints is *at most* as restrictive as its component constraints. Two examples are illustrated in the next section.

5.6 Optima-preserving Constraints

A given constraint may indicate to avoid any combination of optimal actions and suboptimal actions. We will say that a state-action constraint c_{SA} *allows* a policy π if the constraint doesn't specify `Avoid` for any of the actions taken by the policy:

$$\pi(s,a) > 0 \implies c_{SA}(s,a) = \texttt{Unspecified} \tag{26}$$

By satisfying Equation 27, an *optimum-preserving* state-action constraint allows *some* optimal policy. It will never avoid all of the optimal actions in any state, because for any action it specifies should be avoided, there's another permitted action that is as good or better in terms of expected long-term reward.

$$c_{SA}(s,a) = \texttt{Avoid} \implies \exists a' \left[c_{SA}(s,a') = \texttt{Unspecified} \ \wedge \ Q^*(s,a) \leq Q^*(s,a') \right] \tag{27}$$

An important subclass of optimum-preserving constraints allows *all* optimal policies; it only avoids suboptimal actions:

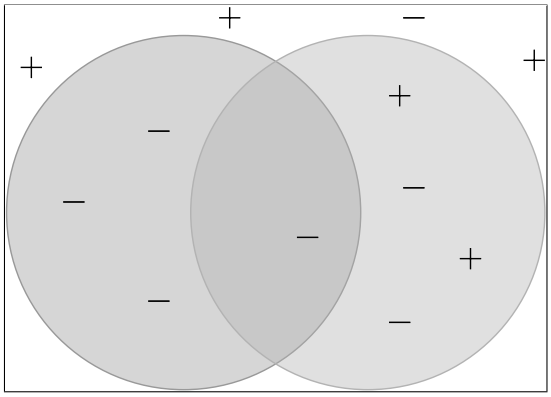
$$c_{SA}(s,a) = \texttt{Avoid} \implies \exists a' \left[c_{SA}(s,a') = \texttt{Unspecified} \ \wedge \ Q^*(s,a) < Q^*(s,a') \right] \tag{28}$$

or, noting that $V^*(s) = \max_a Q^*(s, a)$:

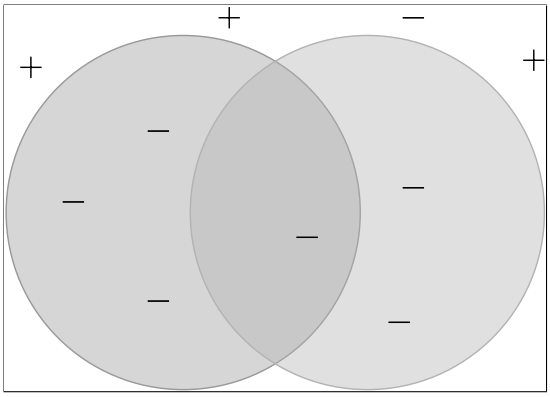
$$c_{SA}(s, a) = \texttt{Avoid} \implies Q^*(s, a) < V^*(s) \tag{29}$$

When we know that a constraint preserves all optimal actions we will refer to it as *all-optima-preserving*.

The intersection of two optimum-preserving constraints, being less restrictive, is still optimum-preserving. The union of two optimum-preserving constraints, being more restrictive, may not be optimum-preserving. The intersection of two all-optima-preserving constraints, being less restrictive of suboptimal actions, is still all-optima-preserving. The union of two all-optima-preserving constraints, being more restrictive of suboptimal actions, is also still all-optima-preserving. (See Figure 5.)



(a) An all-optima-preserving constraint (left) does not contain (avoid) any optimal actions; an optimum-preserving constraint (right) does not contain (avoid) all optimal actions. The intersection is all-optima-preserving; the union in this case is optimum-preserving.



(b) The two constraints are all-optima-preserving, because neither contains (avoids) any optimal actions. The union and intersection of the two constraints each satisfy this property as well.

Figure 5: Union and intersection of sample constraints viewed as sets. + symbols represent optimal actions, − symbols represent suboptimal actions. An ideal constraint contains all of the − and none of the +.

CHAPTER VI

ACTING WITH CONSTRAINTS

So far we have only presented constraints as an information source, and haven't specifically discussed how we might utilize them.

6.1 Constrained MDPs

If we are sure that we don't want our agent to ever take constrained actions, we can remove them from the MDP altogether: given an MDP $m = \langle S, A_s, p, r, \gamma \rangle$ and a state-action constraint c_{SA}, we can construct a *constrained MDP* $m' = \langle S, A'_s, p, r, \gamma \rangle$ where

$$A'_s = \{a \in A_s \mid c_{SA}(s, a) = \texttt{Unspecified}\} \tag{30}$$

The most ideal constraint preserves some optimal actions and no suboptimal actions; the solution to an MDP constrained by such a constraint is then trivial.

In degenerate cases however, we could find $A'_s = \emptyset$ for some s, which is not allowed, as it would cause our MDP m' to be ill-defined. To avoid this situation, one might rely on some other information source to derive a set of allowed actions. We don't have any such secondary information source, so our fallback clause is defined to simply ignore the constraint altogether in any state for which the constraint would cause every action to be removed.

$$okActions(s) = \{a \in A_s \mid c_{SA}(s, a) = \texttt{Unspecified}\} \tag{31}$$

$$A'_s = \begin{cases} okActions(s) & \text{if } okActions(s) \neq \emptyset \\ A_s & \text{otherwise} \end{cases} \tag{32}$$

6.2 Constructing Constrained Options

Constraints can be applied very naturally to options. In this section, we will show how to construct a *constrained option*, a modified version of an existing option, which is "safe" with respect to an applied state-action constraint, and discuss the benefits of doing so.

(a) Original option. (b) Constrained option.

Figure 6: Constructing a constrained option. The original option violates the "avoid red" constraint in seeking the target (green); the constrained option stops short of doing so.

When options are incorporated into a learning framework, it's typical to delegate control of the agent to the option's policy from the time the option is initiated until such time as the option terminates; however, in that time, the option may violate our constraint many times over (Figure 6a). We will construct a new option which doesn't (Figure 6b).

Given an option $o = \langle \mathcal{I}, \pi, \beta \rangle$ and state-action constraint c_{SA}, we can construct a *constrained option* $o' = \langle \mathcal{I}', \pi', \beta' \rangle$, which treats a constraint violation as an additional termination condition of the option. We do this by altering the option's initiation set so that it cannot be initiated if it would always begin with constrained action (Equation 34); by raising the termination probability for each state according to the probability that the original option would take a constrained action (Equation 35); and finally by altering the policy so that any probability mass associated with taking `Avoid` action is proportionately redistributed among the `Unspecified` actions (Equation 36).

$$o' = \langle \mathcal{I}', \pi', \beta' \rangle := \text{constrain}(o, c_{SA}) \tag{33}$$

where

$$\mathcal{I}' = \mathcal{I} - \{s \mid \forall a(\pi(s,a) > 0 \implies c_{SA}(s,a) = \texttt{Avoid})\} \tag{34}$$

$$\beta(s) = \beta(s) + (1 - \beta(s)) \Pr(c_{SA}(s, a_t) = \texttt{Avoid} \mid a_t \sim \pi(s, \cdot))) \tag{35}$$

$$\pi'(s,a) = \begin{cases} 0 & \text{if } c_{SA}(s,a) = \texttt{Avoid} \\ \pi(s,a)/z(s) & \text{otherwise} \end{cases} \tag{36}$$

where $z(s)$ is a normalizing constant chosen so that $\sum_a \pi'(s,a) = 1$. The constrained option o' is the same as o, except that it will terminate where the original would choose an action for which the constraint specified `Avoid`.

6.2.1 Acting with Constrained Options

Just as we defined A'_s in Section 6.1 to replace A_s in a reinforcement learner's problem definition, we can define O'_s to replace O_s for a reinforcement learner that knows how to use options. By analogy to Equations 31 and 32, let O_s represent the set of options available to the agent in state s, and let O'_s represent the set of constrained options resulting from applying this construction to each $o \in O_s$:

$$O_s = \{\langle \mathcal{I}, \pi, \beta \rangle \in O \mid s \in \mathcal{I}\} \tag{37}$$

$$okOptions(s) = \{\langle \mathcal{I}', \pi', \beta' \rangle = \text{constrain}(o, c_{SA}) \mid o \in O_s \wedge s \in \mathcal{I}'\} \tag{38}$$

$$O'_s = \begin{cases} okOptions(s) & \text{if } okOptions(s) \neq \emptyset \\ O_s & \text{otherwise} \end{cases} \tag{39}$$

6.2.2 Separation of Concerns

Sutton et al. [35] demonstrate composition of options by concatenation; we have demonstrated composition of constraints by union and intersection, as well as the composition of options with constraints.

Separation of Concerns, the idea that a software system must be decomposed into parts that overlap in functionality as little as possible, is considered by many to be the most important principle in Software Engineering [21]. It is so central that it appears in many different forms in the evolution of all methodologies, programming languages and best practices [13], and is the motivation behind modular reinforcement learning as well [30].

Decomposition of concerns into independent components allows combinatoric possibilities without combinatoric complexity. Our representation of constraints as first-class components allows options to be defined more simply: one can define an option to "do x", define a constraint to "avoid y", and compose the two to form the constrained option "do x while avoiding y," e.g., "find food while avoiding predator," "traverse terrain without falling into chasms," or "turn corner without hitting pedestrian." Options can be constructed for the general case, while constraints provide safeguards. This decomposition reduces the complexity of constructing options and constraints: x and y may utilize completely disjoint

subsets of features; halving the number of features will greatly simplify the factored learning problem as compared to the joint problem.

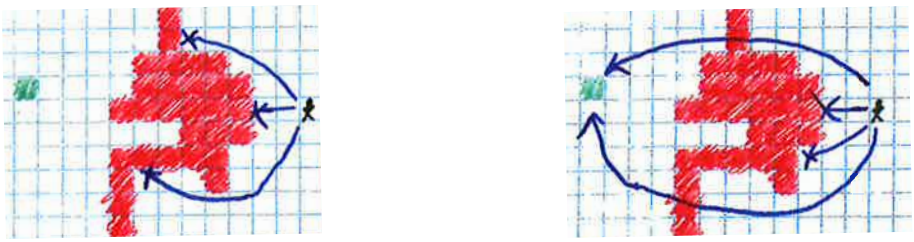
6.2.3 Improved Utility through Constrained Options

If we can make some nice assumptions about our constraints, we can draw some nice conclusions about the benefits of applying them to our options.

Sutton et al. [35, Theorem 2] show that it is advantageous to interrupt an executing option early, in any situation where the expected value of continuing to execute the option is less than the expected value of taking a different option from the current state instead. Granted, the Q values needed to make this determination are not known in advance: Sutton et al. offer the strategy of solving the MDP once normally, and then using the values learned through that initial process to construct new refined options which terminate early upon reaching any state for which the initial solution suggests an alternative option having a higher approximated Q value.

Instead of using an initial Q learning solution to construct the refined options, a state-action constraint could be used as a heuristic, suggesting early termination in cases when an option would violate the constraint. If a state-action constraint is all-optima-preserving, the conditions described by Sutton et al. [35, Theorem 2] will be satisfied, meaning that the Q value of the constrained option will be at least as great as that of the original option: it will only terminate early when it would otherwise take a suboptimal action.

Otherwise, if the state-action constraint is optimum-preserving, and O_s includes all of A_s, then the constrained options will still have utilities at least as great, if not greater than the original options. Even if the constraint incapacitates all of the temporally-extended options, there is at least one optimal primitive action still available to the agent, by virtue of the fact that the constraint is optimum-preserving. It should be noted, however, that although the agent will still be able to learn an optimal policy, the speed with which it can do so may be impacted; it will still have the additional learning burden of discriminating among the new options on top of the original action space, while potentially not benefitting from those options, due to the faulty constraint having been applied to them.



(a) Always obey the constraint. (b) Obey the constraint "whenever practical."

Figure 7: Obeying an ostensibly good constraint ("avoid red") can prevent completion of a task. We might instead like to enforce the constraint "whenever practical," provided some definition of "practical."

If the constraint is neither all-optima-preserving nor optimum-preserving however, then applying constraints in a hard and fast way described in Sections 6.1 and 6.2 will make it impossible for our agent to learn an optimal policy. Figure 7 illustrates a scenario in which strictly obeying a constraint completely prevents the agent from reaching a goal.

6.3 Constraint-Aware Policies

In Section 2.4, we discussed the exploration vs exploitation trade-off as well as the and greedy vs ϵ-greedy exploration; the constructions in Sections 6.1 and 6.2 allow these constraint-oblivious policies to be used with constraints, by altering the set of actions or options presented to them.

It should be unsurprising that good constraints can greatly speed up a learning task, because constraints that preserve one or more optimal policies reduce the amount of exploration that is needed. In the best case, a constraint may remove all but the optimal actions, solving the problem before learning begins. In the worst case, a constraint can prevent the agent from taking an optimal path. To enjoy our performance improvement in the best case while maintaining optimality guarantees in the worst case, we must allow some exploration of constrained actions.

Furthermore, it's not necessarily a simple matter of "good" vs bad constraints: along with the two forms of constraints we've discussed, "don't do x in situation y" and "avoid situation y," one can easily imagine more nuanced constraints, such as "don't do x unless you also shouldn't do y" or "don't do x unless it's a necessary step in accomplishing y."

Although these forms can be represented using state-action constraints (just as a state constraint can be modeled by a state-action constraint), it may feel clumsy or inconvenient to do so (just as the state-action equivalent of a state constraint may be much more verbose). One either needs additional models for these constraint forms and others — noting that there is a tradeoff between very expressive constraints vs computational feasibility — or to accept that a given constraint may be modeled imperfectly with a simpler form. With this in mind, exploring outside of the constraint is important not just to work around arguably flawed constraints, but to work around constraints that are arguably not flawed, but merely imperfectly represented due to a simple representation.

6.3.1 Maintaining Optimality Guarantees through ϵ-greedy υ-defiance

By analogy to the ϵ-greedy policy, which sometimes ignores the Q function, to learn about better alternative actions, we define the ϵ-greedy υ-defiant policy which sometimes ignores constraints, to learn about better alternative actions. This policy selects the best known action (with probability $1 - \epsilon$) or uniformly among the other actions (with probability ϵ), from either the constrained set of actions and options (with probability υ) or the set of unmodified actions and options (with probability $1 - \upsilon$). These four possibilities are summarized in Table 2.

As an example: suppose $\epsilon = 0.2$ and $\upsilon = 0.8$. The majority of the time ($(1 - \epsilon)\upsilon = 64\%$), the policy will choose a greedy action that respects the constraint. The rest of the probabilities are given in Table 3.

Just as ϵ represents the proportion of action choices that should be made with the assumption that our current Q approximation is good enough, rather than needing improvement; υ represents the proportion of action choices that should be made with the

Table 2: ϵ-greedy υ-defiant policy.

p	Respect Constraint	Ignore Constraint
Exploit	$(1 - \epsilon)\upsilon$	$(1 - \epsilon)(1 - \upsilon)$
Explore	$\epsilon\upsilon$	$\epsilon(1 - \upsilon)$

Table 3: ϵ-greedy υ-defiant policy with $\epsilon = 0.2$ and $\upsilon = 0.8$.

p	Respect Constraint	Ignore Constraint
Greedy Action	64%	16%
Random Action	16%	4%

assumption that our constraint is good enough, meaning that the optimal action can be found with then the constrained set of actions, rather than within the unmodified set. In this sense, υ represents our confidence in the constraint.

Besides ϵ-greedy, there are many strategies for choosing between exploration and exploitation. An ϵ-*first* strategy uses an initial period of pure exploration (ϵ fraction of the total learning time) followed by a period of pure exploitation ($1 - \epsilon$ fraction of the total learning time). An ϵ-*decaying* strategy is similar to ϵ-greedy, except that ϵ is decreased over time, resulting in more explorative behavior at the start and more exploitative behavior later on. An υ-selection strategy could be derived from any of these, and an optimal policy can still be found, provided that $\epsilon > 0$ and $\upsilon > 0$. Of course, if a given constraint is optimum-preserving, the most efficient learning will be achieved with $\upsilon = 1$.

In some of the experiments that follow, we use an υ-decreasing policy. This means that the agent will initially tend to obey the constraint, and most of its exploration will be among the actions that the constraint does not avoid. Over time, the agent will increasingly explore actions which the constraint does indicate to avoid. The rationale for this is that if the constraint is not optimum-preserving, an optimal policy will *only* be found by exploring outside of the constraint; because the agent has hopefully by this point internalized useful elements from the constraint through experience and Q function updates, it will not be too damaging to performance to rely on that Q function now.

6.3.2 Acting in Novel States: The Greedy Constraint-Biased Policy

In Section 2.4 we also discussed why a fully exploitative policy might be preferred during learner *evaluation*. Implicit in this preference for greediness is the assumption that the agent has learned some minimal amount of knowledge which can be exploited. In larger

stochastic domains however, many states are never encountered by the agent more than once, meaning that there is no past experience to exploit. In the case of some preliminary experiments we performed in the Pac-Man domain, 97–99% of states encountered were only encountered once.

In such cases, where an agent needs to choose an action in a totally novel state, or in any situation requiring a tie-break a, the agent could choose among its constrained actions, among its original actions, or from a combination of the two sets. If we want to obey the constraint in this situation, because we assume that the constraint is more likely to be informed than a totally inexperienced agent is, our greedy policy becomes a *greedy constraint-biased* policy:

Under the greedy constraint-biased policy, the agent will choose actions according to

$$\pi_{gcb}(\hat{Q}, s) = \mathcal{U}\left(\underset{o_i \in (O'_s \cup O_{\hat{Q}_s})}{\arg\max} \hat{Q}(s, o_i)\right), \tag{40}$$

where O'_s (from Equation 38) is the set of constrained options which can be initiated from s, and $O_{\hat{Q}_s}$ is a subset of the original, non-constrained options that includes an option o if and only if $\hat{Q}(s, o)$ has been previously used by the agent, meaning that it has received at least one Q table update so far. If a state s has not previously been visited, or if it has been visited but the agent has never used one of the original options there, $O_{\hat{Q}_s} = \emptyset$ and $O'_{s,c} \cup O_{\hat{Q}_s} = O'_s$; so we simply choose an option from O'_s under the presumption that an option which obeys the constraint will outperform one which doesn't.

6.4 Summary

In this chapter we saw how to compose options with constraints to create "safer" constrained options, as well as constraint-aware policies which allow us to work around *overly* protective constraints. The next chapters will lay out methods for producing constraints through human input; we will use the techniques discussed in this chapter for their evaluations.

CHAPTER VII

EXPERIMENTAL DOMAINS AND EVALUATION

In this dissertation, we evaluate our techniques in a simple synthetic domain called Radiation World, as well as two classic video game inspired domains, Pac-Man and Mario World. This chapter describes the three domains, including the dynamics, actions, options, and rewards, as well as our Q learner evaluation method.

7.1 Radiation World

In the Radiation World domain (see Figures 8– 10), the agent represents a search-and-rescue robot deployed in a nuclear disaster scenario to rescue humans. The episode ends when the agent has rescued every human, or if the agent moves onto a radiation source. There is an additional catch, being too near a radiation source affects the robot's ability to navigate precisely, which can mean delays in the best case, or mission failure by driving into a radiation source.

The agent receives +100 reward for each human rescued, and no other rewards. The discount factor $\gamma = 0.99999$. Radiation World was designed to exemplify the case where there are low-utility regions (the heavily irradiated areas) that are intuitively bad from a human's perspective, but neutral through the lens of immediate rewards.

The robot's primitive actions, `North`, `South`, `East`, `West`, have stochastic outcomes,

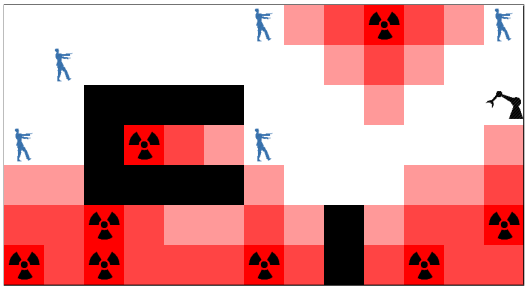


Figure 8: Radiation World board 1. The agent receives reward for each human rescued, but proximity to radiation centers increases entropy in movement. A human is rescued when the agent visits the human's cell.

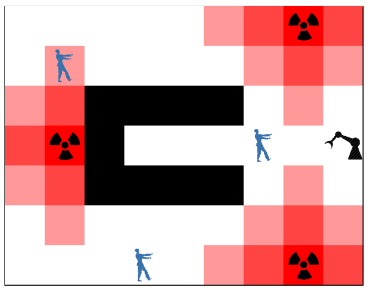


Figure 9: Radiation World board 2. On this board, it is necessary to traverse light radiation in order to rescue all three survivors.

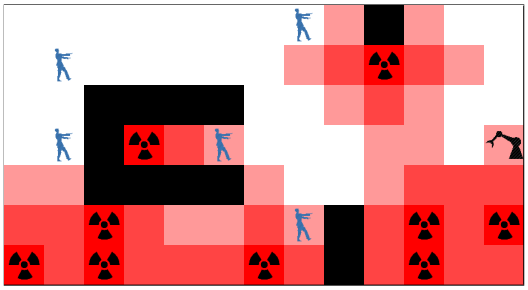


Figure 10: Radiation World board 3. On this board, it is necessary to traverse a greater amount of radiation in order to rescue any survivors.

and they are influenced by the environment. With each action the robot takes, there is a probability of moving in an unintended direction. This movement error increases with environmental radiation, modeled with geometric falloff based on distance from a radiation source. In the worst case, movement outcomes are uniformly random. The probability of moving in the intended direction is

$$\Pr(\text{move}(a) \mid \text{choose}(a)) = \max\left(\frac{1}{|A_s|}, 1 - \text{radiationStrength}(agentPos)\right) \tag{41}$$

where

$$\text{radiationStrength}(pos) = \sum_{radSrc} \left(\frac{1}{2}\right)^{\text{dist}(radSrc,pos)} \tag{42}$$

and dist represents path length between two points (radiation does not pass through these walls) and where *radSrc* and *agent* are positions of the radiation sources and the agent. For example, if there is a single radiation source with a distance of 2 from the agent, $radiationStrength = (1/2)^2 = 1/4$, and the probability of actually moving east when taking the `East` action is $\max(1/4, 1 - 1/4) = \max(1/4, 3/4) = 3/4$. The total probability

of moving in some other direction is therefore $1 - 3/4 = 1/4$; the actual direction the agent ends up moving in is chosen uniformly among the remaining three, so there will be a 3/4 probability of moving east, and a 1/12 probability of moving in any of the other directions. If the ultimately resulting move direction is one that is blocked by a wall, the agent's position does not change.

There is a single option, `gotoNearestHuman`, which attempts to take the most direct path to the nearest human, subject to the inexact movement described above. Moving onto a radiation source will terminate an episode with suboptimal total reward, and given that simply being near a radiation source can lead to unintended movement, it's best if the agent can avoid radiation whenever possible; however, there are no intermediate rewards to indicate this to a reinforcement learner.

Although we present an atomic state representation to our Q learner, the Radiation World simulator's structured state representation is given in Scala in Listing 5.

```
case class RadiationWorldState(width:     Int,
                               height:    Int,
                               walls:     Set[Position],
                               humans:    Set[Position],
                               agent:     Position,
                               radiators: Set[Position],
                               playState: PlayState,
                               aux:       AuxState)
{
  val radiationStrength: Map[Position, Double] = ...
  // map of radiation strengths across the board,
  // derived from wall and radiator positions
}

case class Position(x: Int, y: Int)
enumeration PlayState { Playing, Win, Lose }

case class AuxState(humanSaved: Boolean)
// used by simulator to remember whether to issue reward
```

Listing 5: Radiation World state structure and substructures.

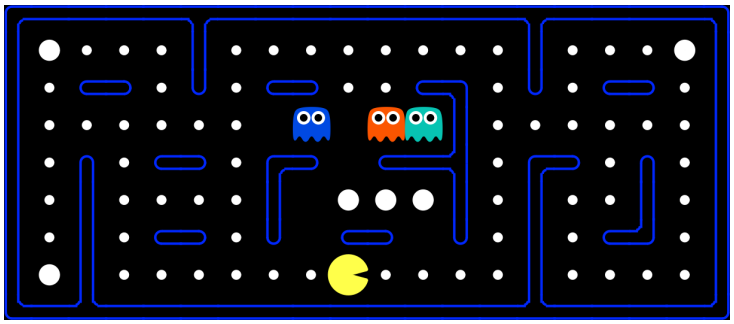


Figure 11: Pac-Man board 1. Pac-Man receives reward for eating pac-dots or ghosts, but dies with negative reward when coming into contact with an inedible ghost. Ghosts become temporarily edible when Pac-Man eats an energizer.

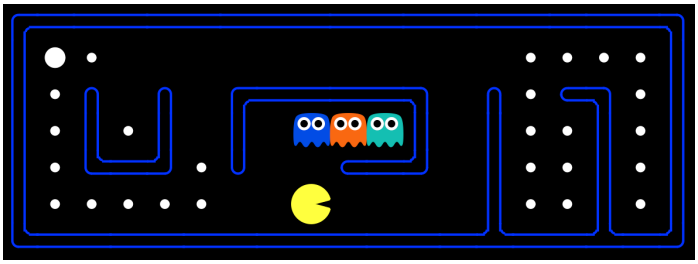


Figure 12: Pac-Man board 2.

7.2 Pac-Man

In this classic-game-inspired predator-prey scenario, the agent must collect *pac-dots* distributed throughout a maze while avoiding contact with one or more ghosts that roam the maze (see Figures 11–12).

Unlike in the classic arcade game, in which each ghost follows a particular strategy, these ghosts act identically. At each time step, each ghost will move in any of four directions, with certain limitations: it will never attempt to move in a direction that is blocked by a wall, and it will never reverse directions unless it has reached a dead-end. Given a choice between moving forward or turning left or right, relative to the direction of its previous move, it will choose among the unblocked directions with equal probability.

In addition to eating pac-dots, Pac-Man can also eat the larger dots called *energizers*, which are sparsely distributed around the maze. Eating an energizer will cause all of the ghosts to become edible and change color for a fixed time interval before becoming inedible again. If Pac-Man eats a ghost, the ghost re-spawns as an inedible ghost at its initial

location. If Pac-Man runs into an inedible ghost, he dies (terminal state with negative reward).

Pac-Man's primitive actions (`North`, `South`, `East`, `West`) result in deterministic movement. We have defined three options based on the pilot study discussed in Section 4.2: `gotoNearestPacDot`, `gotoNearestEnergizer`, and `gotoNearestGhost` (edible or not). Each of these options uses the A^* algorithm to take the most direct path to the nearest instance of its respective object type, taking walls into account but nothing else. (Specifically, it doesn't take the location of ghosts into account.)[1]

The schedule of rewards is taken from the popular UC Berkeley Pac-Man Projects [6]. The agent earns reward by eating dots (+10 reward), power pills (+0) and edible ghosts (+200) and for time elapsed (−1 per time-step). Additionally, the episode ends in a win when agent eats all the pac-dots in the maze (+500), and ends in a loss if Pac-Man runs into an inedible ghost (−500). The discount factor $\gamma = 0.9$.

Although we present an atomic state representation to our Q learner, the Pac-Man simulator's structured state representation is given in Scala in Listing 6.

7.3 Mario World

Mario World (see Figure 13) is a real-time domain based on a platform game, in which the agent (Mario) must jump between suspended platforms, over obstacles, and avoiding or stomping on enemies. The agent's actions are selected from the Cartesian product:

$$\left\{\begin{array}{c}\texttt{Up}\\ \texttt{Neutral}\\ \texttt{Down}\end{array}\right\} \times \left\{\begin{array}{c}\texttt{Left}\\ \texttt{Neutral}\\ \texttt{Right}\end{array}\right\} \times \left\{\begin{array}{c}\texttt{Jump}\\ \texttt{NoJump}\end{array}\right\} \times \left\{\begin{array}{c}\texttt{Shoot}\\ \texttt{NoShoot}\end{array}\right\}$$

for a total of thirty-six distinct actions in each state. No options are defined in our Mario World domain.

Mario can be in any of three power-up states: Small Mario, Super Mario, or Fire Mario. Touching a mushroom will turn Small Mario into Super Mario; touching a fire flower will

[1]The pilot study participants' third option was "eat ghost", but this felt like an unfairly smart and complex behavior to predefine for our experiments. The `gotoNearestGhost` option we implemented instead still requires the agent to learn the importance of eating energizers.

```
case class PacmanState(width: Int,
                       height: Int,
                       walls: Set[Position],
                       food: Set[Position],
                       capsules: Set[Position],
                       pacman: AgentConfig,
                       ghosts: List[GhostInfo],
                       edibleTime: Maybe[Int],
                       playState: PlayState,
                       aux: AuxState)
case class Position(x: Int, y: Int)
case class AgentConfig(position: Position,
                       facing: Direction)
case class GhostInfo(config: AgentConfig,
                     edible: Boolean,
                     paused: Boolean,
                     respawnPosition: Position)
enumeration PlayState { Playing, Win, Lose }
enumeration Direction { North, South, East, West }

case class AuxState(foodEaten: Boolean,
                    capsuleEaten: Boolean,
                    ghostsEaten: Int)
// used by simulator to remember how much reward to issue
```

Listing 6: Pac-Man state structure and substructures.

turn him into Fire Mario state. If Fire Mario gets hurt, which can happen in various ways dependent on what enemy he has run into, he will revert to Super Mario. If hurt in the Super Mario state, he will revert to Small Mario. If hurt while in the Small Mario state, Mario will die. The episode will also end in death with if Mario falls into a pit. The schedule of rewards is taken from Togelius et al. [38] and is given in Table 4. The discount factor $\gamma = 0.99999$.

If the `Shoot` sub-action is selected over multiple consecutive simulation states, the agent's speed will be increased until the `NoShoot` sub-action is selected. In the Fire Mario power-up state, the `Shoot` sub-action fires a projectile in the direction the agent is facing. The projectile will continue until it collides with an enemy (generally killing it), or collides with a wall. After the `NoShoot` sub-action is selected, then a new projectile can be fired by selecting the `Shoot` sub-action again in a subsequent time-step. Up to two fire projectiles can exist at

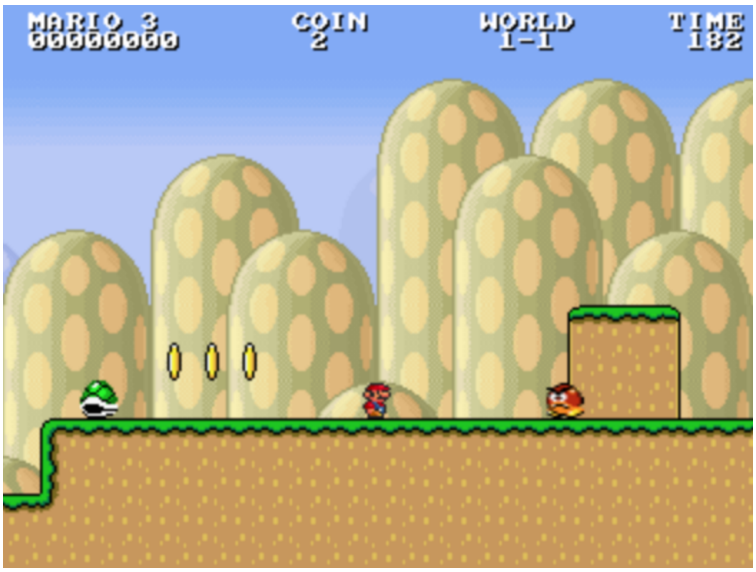


Figure 13: Mario World. The agent receives small rewards based on interaction with other entities on the screen, and +1024 for reaching the finish.

Table 4: Reward function for Mario World.

+1024	Reach Princess
−512	Die Trying (or timeout)
+64	Fire flower
+58	Mushroom
+24	Hidden Block
+16	Coin
+10	Stomp enemy
-42	Get hurt
-1	Each time-step

clearpage

any given time; additional `Shoot-NoShoot` cycles will not produce new projectiles until an existing one is eliminated through a collision as described previously or moves sufficiently outside of the visible screen area.

Although we present an atomic state representation to our Q learner, the Mario World simulator's structured state representation is given in Scala in Listing 7.

7.4 Evaluation Method

The evaluation process is divided into learning episodes and evaluation episodes. We evaluate the various learners by counting their cumulative, undiscounted reward over the course of each evaluation episode. Further, the learning episodes and evaluation episodes may use different policies. For example, in Chapters 8 and 9, we will use an ϵ-greedy policy during learning, and a greedy policy during evaluation, to allow the agent to demonstrate its best performance.

Each experiment has a particular number of independent trials τ, cumulative number of learning episodes per trial l, and an evaluation frequency f. In the following example, $\tau = 100$, $l = 10000$ and $f = 50$, meaning that in each independent trial, performance will be measured after every 50 learning episodes until 10,000 learning episodes have been completed for that trial. This is a total of $l/f = 10000/50 = 200$ evaluation episodes per independent trial, $\tau l/f = 20000$ episodes in total.

At the start of each trial, the agent is initialized with an empty map-backed Q table with a default Q value of 0, as in Listing 2. Next, the agent performs 50 episodes of Q learning using its training policy, and carrying its Q function approximation over from the end of the previous episode to the start of the next. After 50 learning episodes are completed, we perform one evaluation episode using the evaluation policy. The rewards received by the agent over the course of the evaluation episode are summed and recorded, yielding one data point: $(x = 50, y = r_{50,i})$ where i represents the current trial. The x-value represents the number of learning episodes the agent had experienced prior to this evaluation episode, and the y-value is the sum of rewards received during this evaluation.

Next, the agent continues learning using the training policy in episodes 51-100. The Q function approximation that was learned over the 1st-50th learning episodes is carried over to the 51st; the updates to the Q function which occurred during the *evaluation* episode are discarded at the end of the evaluation episode. In other words, the evaluation episodes do not impact the course of learning, which is particularly important if the evaluation policy is very different from the training policy. (Note that the agent can learn during the course of the evaluation episode, but the learning that occurs during the evaluation episode isn't carried forward to the next learning episode.) After the 100th learning episode is finished, the test harness performs another evaluation. This process is repeated until l learning episodes and l/f evaluation episodes have been performed.

This entire process (l learning episodes and l/f evaluation episodes) is repeated τ times independently, and the results are averaged, smoothed, and plotted. We compute $\bar{r_k}$ for each evaluation episode k, over all independent trials i,

$$\bar{r_k} = \frac{1}{\tau} \sum_{i<\tau} r_{k,i} \tag{43}$$

and plot the points $(x = k, y = \bar{r_k})$. To improve interpretability, the graph lines are smoothed using `gnuplot`'s Bezier algorithm.

We'll use this evaluation method in each of the experiments that follow.

```
trait MarioSimulationState {
  spriteSet: Set[Sprite]
  enemySet: Set[Enemy]
  mario: Mario
  score: Int
}

trait Sprite extends MarioObject {
  def getVelocity: MarioVector
  def getLabel: SpriteLabel
}

case class Enemy(getPosition: MarioVector,
                 getVelocity: MarioVector,
                 getSize: MarioVector,
                 getLabel: Label,
                 dead: Boolean) extends Sprite

case class MarioVector(getX: Float, getY: Float)

case class Mario(getPosition: MarioVector,
                 getVelocity: MarioVector,
                 getSize: MarioVector,
                 getLabel: SpriteLabel,
                 isCarrying: Boolean,
                 isGrounded: Boolean,
                 wasGroundedBefore: Boolean,
                 jumpInfo: JumpInfo,
                 mayJump: Boolean,
                 mayShoot: Boolean,
                 getMode: MarioMode,
                 getStatus: MarioStatus) extends Sprite

case class JumpInfo(jumpTime: Int, jumpSpeedX: Float, jumpSpeedY: Float)

enumeration SpriteLabel {
  Goomba, GoombaWinged, GoombaWave,
  KoopaRed, KoopaRedWinged, KoopaGreen, KoopaGreenWinged,
  BulletBill, Spiny, SpinyWinged, PiranhaPlant, Shell,
  Mushroom, MushroomGreen, FireFlower, Coin, Princess,
  Particle, Sparkle, Fireball, Mario
}
enumeration MarioMode { Small, Large, Fireball }
enumeration MarioStatus { Play, Win, Dead }
```

Listing 7: Mario World state structure and substructures.x

CHAPTER VIII

STATE CONSTRAINTS FROM DEMONSTRATED FAILURE: EXPERIMENTAL METHOD

In this chapter, we present an experiment designed to see how Q learning outcomes in the Radiation World and Pac-Man domains are affected by making use of (or not making use of) constraints from people. At a high level, we will collect examples of `Avoid` states as well as features from a layperson, use this data to train a decision tree to serve as a constraint, and test each derived constraint in learning, using the constraint-oblivious case as a control. The experiment will consist of seven distinct phases:

1. Orientation and "bad" state collection
2. "Bad" state selection criteria collection
3. Coding of user criteria into candidate state features
4. Feature extractor construction
5. Constraint decision tree training
6. State constraint to state-action state conversion
7. Q learning evaluation

The outputs and interdependencies of these seven phases are illustrated in Figure 14. Phases 1 and 2 involve collecting input from study participants; Phases 3 and 4 depend on the experimenter's interpretation of the outputs of Phase 2; Phases 5, 6, and 7 are fully automated. As shown in Figure 14, Phases 1–2 and 5–7 will be performed individually for each participant, yielding one constraint per participant per domain, while Phases 3–4 will be performed only once, in aggregate, and a single feature extractor will be shared in all instances of Phase 5 for a given domain.

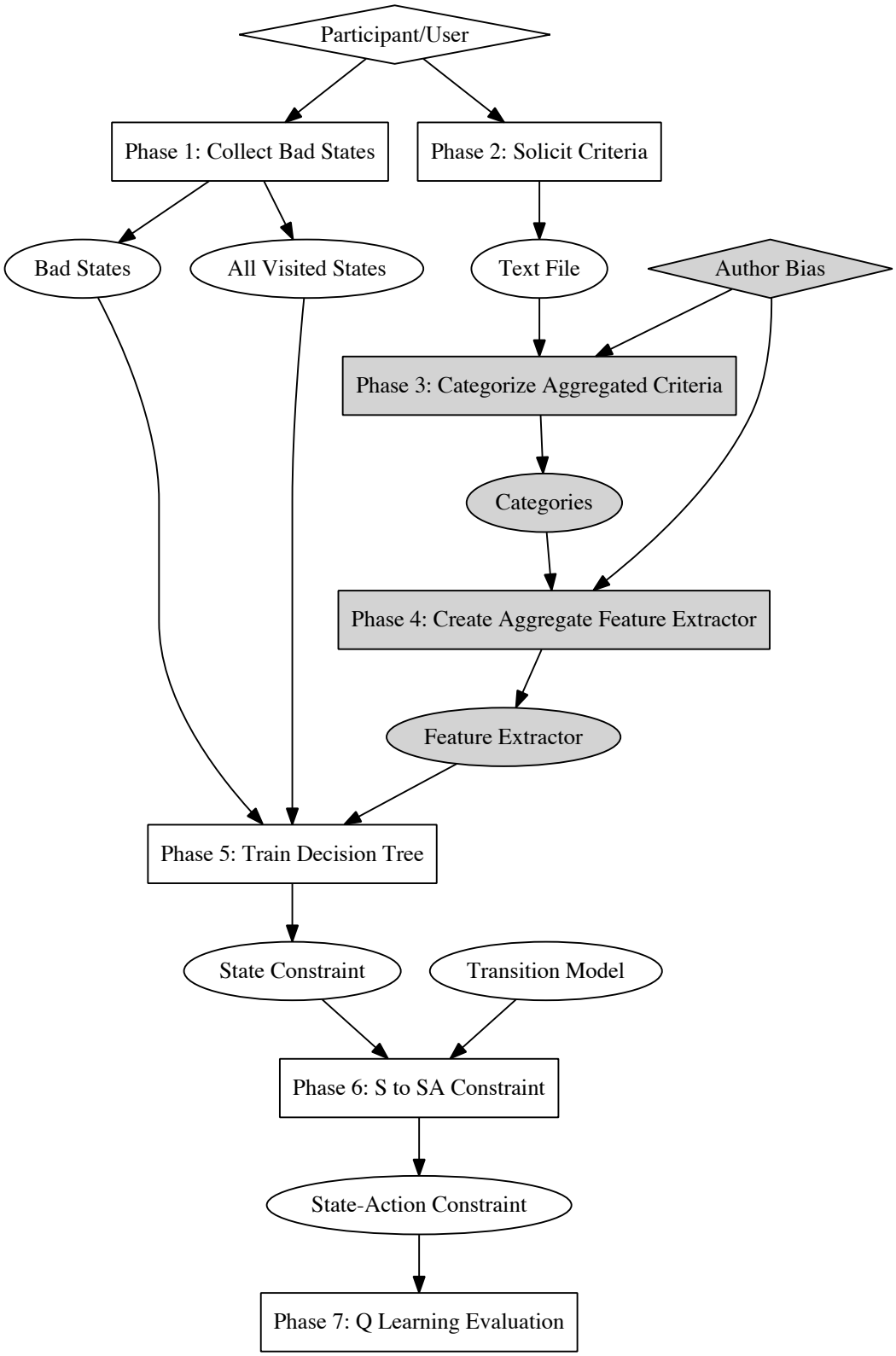


Figure 14: Inputs and outputs (circles), phases (rectangles), dependencies (arrows), and humans (diamonds), in the experiment with constraints from demonstrated failure states. Grey nodes represent components performed only once, in aggregate, for each domain; white nodes were performed for each (user,domain) pair.

8.1 Phase 1: Orientation and "Bad" State Collection

In this first phase, we will collect two quantities from each user: (1) a set of states the user designates as "bad" or `Avoid` and (2) the full set of states that the user witnesses while constructing the set of `Avoid` states. The difference between these two sets will yield a set of witnessed but non-designated states, which we will treat as `Unspecified` when constructing our classifier in Phase 5 of the experiment.

We begin by explaining to participants that we are trying to teach AI agents by providing examples of what a bad situation looks like. We will present the participants with the interface shown in Figures 15 and 16, and ask them to use the interface to construct examples. The user interface is made up of four primary components:

Game Area. The game area, in the upper left, is the largest component. It displays the current state of the game as the user takes actions.

Movement Controls. The user sends actions to the simulation using the keyboard. In our two domains, the arrow keys are mapped to primitive actions, `North`, `South`, `East`, `West`. In the current implementation, users do not have access to any options or macro-actions.

Simulation Controls. To supplement the movement controls, the interface also includes a "Reload initial configuration" button, which resets the simulation state; and an "Undo one action" button, which pops the most recently visited state off of the application's stack, reverting to the previous simulation state. The Undo button can be used repeatedly, to revert to a less recent state.

Snapshot collection and controls. Once the user has demonstrated a "bad" state, he can use the "Mark this configuration as Bad" button to retain the bad state in the snapshot collection, which displays thumbnails of the selected states below the game area. The user can use the mouse to select previously marked states; this loads the selected state in the Game Area. The ability to reload a state into the Game Area, in conjunction with the Undo functionality, facilitates additional exploration around previously marked "bad" states. A

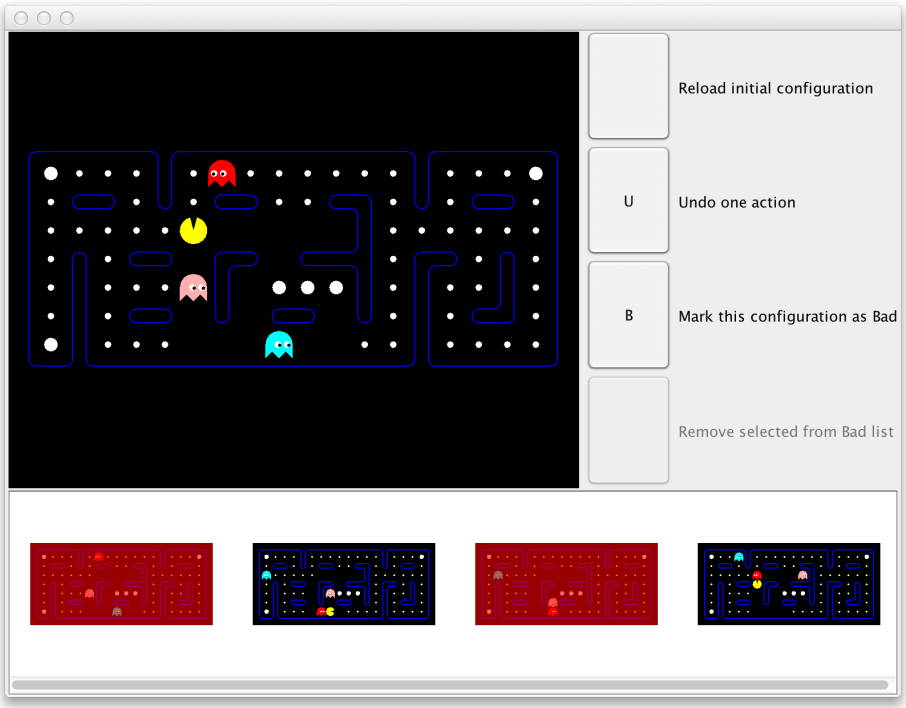


Figure 15: Capturing "bad" states in the Pac-Man domain.

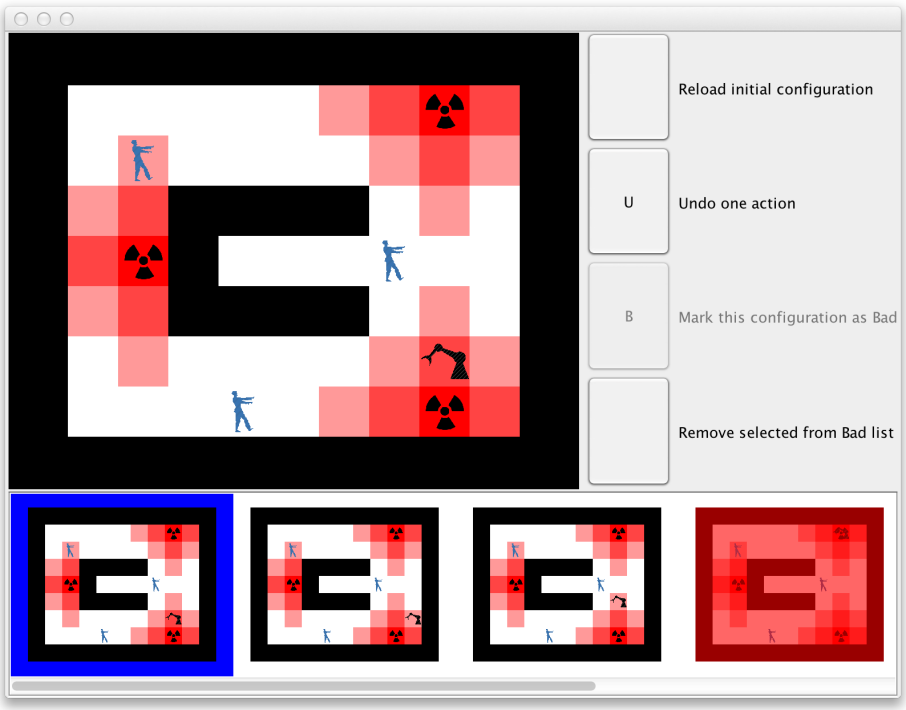


Figure 16: Capturing "bad" states in the Radiation World domain. The most recently marked state is selected.

selected state can be removed from the snapshot collection using the "Remove selected from Bad list" button.

After explaining the user interface components, we will give each participant an opportunity to play as many episodes of their first game as they need to feel familiar with the game. Next, we will guide each participant through a scripted practice run that utilizes all of the snapshot controls. We will instruct participants that they can even take a snapshot of a configuration in which they have already finished or lost. The full script of the user orientation is given in Appendix A.

Finally, we will begin recording data. We will allow users to control the agent and mark as many bad states as they feel is appropriate.

A side note: this user interface implementation can easily be reused for any turn-based domain for which there is a generative model, a mapping from keyboard keys to actions, the ability to render a given state to a `java.awt.Graphics2D`, and a codec mapping states to and from JSON.

8.2 Phase 2: "Bad" State Criteria Collection

Once the participants are satisfied with their selection of "bad situations", they will be asked to "write a few sentences describing how you selected the 'bad situations', or how you could tell if a situation was 'bad'." After a participant completes Phases 1 and 2 for both domains, they will be dismissed.

8.3 Phase 3: Coding Criteria into Potential Features

In this phase, we will interpret the participants' responses from Phase 2, and we will code them according to existing state features or simple functions of domain features. This phase has the potential to be very subjective, but we will attempt to be conservative in our interpretations: although one could potentially engineer a very complex feature extractor here, we choose to try to look for direct correspondences between ideas or quantities described by participants and existing state features or simple functions of existing state features. For example:

> "I selected each case where the robot died [`playState`]. I selected each of the darker-red squares [`radiationStrength`] since there is no reason the robot should try and go on those. I also marked the lighter red squares [`radiationStrength`] that weren't necessary for helping all of the zombies [`unavoidable`]."

8.4 Phase 4: Feature Extractor Implementation

In this phase, we must construct a feature extractor to be used in Phase 5. We want this to be a simple procedure, so we will take a conservative route and select the single most-suggested feature to back our decision-tree based constraint for each domain. We will see that this simple approach is sufficient for us to see improved learning outcomes.

At the end of Section 3.4, we hinted that feature engineering could make for an unfair comparison, but again here we will only use simple programming to select simple correspondences in the state representation based on layperson's criteria.

8.5 Phase 5: Constraint Decision Tree Training

Given the feature extractors constructed in Phase 4, we will use the WEKA J48 decision tree learning algorithm [12, 28], using the "bad" and unlabeled states provided by participants in Phase 1 to create a classifier which categorizes input states as `Avoid` or `Unspecified`. The states marked as "bad" by a given participant will be taken as positive examples of `Avoid`, while the unlabeled states visited the participant will be taken as examples of `Unspecified`.

Because we expect to have far fewer positive examples than unlabeled examples, and because the unlabeled examples, taken as negative examples, will actually contain a mix of unlabeled positive and negative instances, we will give less weight those unlabeled instances, such that the positive and negative sets each carry approximately the same total weight. (If we had a factored state representation, we could use a more principled weighting scheme such as those proposed by Lee and Liu [20], Denis et al. [7], Elkan and Noto [9].)

This procedure will result in one decision tree per domain, per participant. We can then interpret each or any of the resulting trees as a state constraint: the constraint returns `Avoid` for state instances that are classified as "Bad", and `Unspecified` for instances that are classified as "Ok".

8.6 Phase 6: From State to State-Action Constraint

We will use the construction from Section 5.4, Equation 21 with $tolerance = 0$ to derive state-action constraints for each state constraint we will have computed in Phase 5:

$$c_{SA}(s,a) = \begin{cases} \texttt{Avoid} & \text{if} \quad \left(\sum_{\{s' | c_S(s') = \texttt{Avoid}\}} T(s,a,s')\right) > 0 \\ \texttt{Unspecified} & \text{otherwise} \end{cases} \tag{44}$$

The state constraint c_S in is one learned in Phase 5, and the state transition function $T(s,a,s')$ is known for the Radiation World and Pac-Man domains. The resulting state-action constraint c_{SA} returns `Avoid` for any action that could result in a state for which c_S would return `Avoid`.

8.7 Phase 7: Q Learning Evaluation

To evaluate the effect of our user constraints on Q learning performance, we will perform 200 independent trials in each of 52 different experimental conditions (Figure 17), varying domain, board, primitive actions only or with options, with each distinct user-derived constraint as well as without constraints.

We will measure the performance as described in Section 7.4, with evaluation frequency $f = 50$. For learning episodes, we will use a ϵ-greedy v-defiant policy (described in Section 6.3.1); and for evaluation episodes, we will use a greedy constraint-biased policy (described in Section 6.3.2).

$$\left\{\begin{matrix}\text{Pac-Man} \\ \text{Radiation World}\end{matrix}\right\} \times \left\{\begin{matrix} board_{1_d} \\ \vdots \\ board_{N_d} \end{matrix}\right\} \times \left\{\begin{matrix}\text{primitive actions only} \\ \text{primitives and options}\end{matrix}\right\} \times \left\{\begin{matrix} \text{no constraints} \\ constraint_{1_d} \\ \vdots \\ constraint_{M_d} \end{matrix}\right\}$$

Figure 17: Cartesian product of conditions to be evaluated in this experiment. The subscript d references the selected domain, either Pac-Man or Radiation World.

CHAPTER IX

STATE CONSTRAINTS FROM DEMONSTRATED FAILURE: RESULTS AND DISCUSSION

We performed the experiment defined in Chapter 8; this chapter presents and discusses our results for each experimental phase, as well as some surprises we encountered.

Eight participants were recruited from the Georgia Tech student body, and they were selected for absence of past experience in artificial intelligence. The participant group consisted of one computer science Ph.D. student and seven students recruited from an undergraduate freshman-level English course. Odd-numbered participants worked first with Radiation World and then with Pac-Man; the even-numbered worked with the two domains in the opposite order.

9.1 Phase 1: Collect Bad States

We conducted the orientation for each participant, who then went on to use the user interface described in Section 8.1 to collect "bad" states for both Radiation World and Pac-Man domains. In doing so, our participants make use of all of the user interface's optional capabilities: reloading the initial configuration, undoing recent actions, reviewing the history of marked "bad" states, as well as loading previously-marked states back into the Game Area.

For Radiation World, the mean number of "bad" states collected by participants was 9.75 out of 52.75 states encountered. For Pac-Man, the mean number of selected states was 11.625 out of 516.75 encountered. Individual numbers for each participant are given in Tables 5 and 6.

Our first surprise was that several participants did not mark failure states (such as Pac-Man having been eaten by a ghost) as "bad", despite receiving instructions and reminders that it was legal to do so. Several participants would consistently navigate to a failure state, then press Undo to load the state just prior to failure, then marked the pre-failure state as "bad". They may have believed that the state was dangerous because it offered the

Table 5: "Bad" states selected in Radiation World.

Participant ID	States Selected	States Encountered
#1	22	75
#2	11	58
#3	6	41
#4	6	53
#5	12	64
#6	6	69
#7	9	34
#8	6	28
Mean	9.75	52.75

Table 6: "Bad" states selected in Pac-Man.

Participant ID	States Selected	States Encountered
#1	34	316
#2	11	1020
#3	5	434
#4	6	343
#5	9	312
#6	7	380
#7	16	1077
#8	5	252
Mean	11.625	516.75

opportunity of failure, even though there were alternative actions which could be taken to avoid failure. This is reminiscent of *anticipatory rewards*, described by Thomaz and Breazeal [37, 36], where participants would issue reward in anticipation of actions the agent hadn't performed. Perhaps what we observed is evidence of participants supplying *anticipatory constraints*. Alternatively, they may have believed that the failure state was so obviously bad that explicitly marking it as bad would be redundant; however, as described in Section 8.5, the classification algorithm will treat these visited-but-unmarked states as low-weighted `Unspecified`, potentially good states. We expect that this phenomenon can be avoided by explicitly telling participants that it is helpful (rather than merely "acceptable") to mark failure states as "bad".

9.2 Phase 2: "Bad" State Criteria Collection

The participants typed their answers to, "How could you tell if a situation was 'bad'," and were then dismissed. Some participants used the snapshot collection and controls to refresh their memories of their selections, while they were composing their answers. Two responses are given here, while the full annotated text of each participants response is given in Appendices B and C. We further characterize their responses in Section 9.3.

Participant #1, Radiation World

> I selected each case where the robot died. I selected each of the darker-red squares since there is no reason the robot should try and go on those. I also marked the lighter red squares that weren't necessary for helping all of the zombies.

Participant #1, Pac-Man

> Anytime I died I marked that as a bad state. Anytime I was within 1 or 2 squares of a ghost I marked that as a bad state. Anytime I was within 3 squares of a ghost but stuck in a corner I marked that as a bad state. Anytime I had just eaten a ghost, but it respawned within 2 squares, I marked that as a bad state."

9.3 Phase 3: Coding Criteria into Potential Features

In this phase, we interpret the participants' responses from Phase 2, and code them according to existing state features or simple functions of domain features. We describe one example here in detail, and summarize all of the results in Tables 7–9. Table 7 lists our interpretation of each of the coded criteria, and is divided into four sections: features which apply to both Pac-Man & Radiation World, Pac-Man-only features, Radiation World-only features, and finally, includes three classes of unsupported constraints which were implied by the participants' responses. The frequency with which each was represented in the participants' responses is given in Tables 9 and 8, and the full annotated text of each participant's response is given in Appendices B and C.

> "I selected each case where the robot died [`playState`]. I selected each of the darker-red squares [`radiationStrength`] since there is no reason the robot should try and go on those. I also marked the lighter red squares [`radiationStrength`] that weren't necessary for helping all of the zombies [`unavoidable`]."

In this example from Participant #1, we labeled the mention of the Radiation World robot dying as [`playState`], a reference to an internal variable present in both the Radiation World and Pac-Man domains' state representation indicating whether the game has been won, lost, or is still in progress. Two references to shades of red were labeled as [`radiationStrength`], a property of each grid cell in a Radiation World state. This property is used to determine the robot's movement accuracy (Equations 41 and 42), as well as to determine the redness of each grid cell as displayed in the user interface. By, "I selected/marked ... red squares," Participant #1 is referring to states which he had marked as "bad" after navigating the robot to a grid cell of a particular color. The last sentence refers to grid cells which are "necessary", meaning that they must be visited by any policy which rescues all the humans. This concept was labeled [`unavoidable`].

In addition to the the twelve state features we found in the participants' descriptions, we noticed three classes of features which were not supported by our implementation but which suggest avenues for future work:

Table 7: Interpretation of features indicated by participants.

Feature	Interpretation
`playState`	Win, Lose, or Playing.
`distanceToGhost`	The path distance to the nearest dangerous ghost.
`surrounded/ stuckInCorner`	Pac-Man is surrounded by ghosts or stuck in a corner.
`dotsRemaining`	Either the absolute count or percentage of dots remaining.
`energizersRemaining`	Either the absolute count or percentage of energizers remaining.
`dontWasteEnergizer`	In certain situations, eating an energizer would be wasteful.
`hardToReachEnergizer`	It would be difficult to reach an energizer.
`distanceToRespawnPoint`	Either the distance from Pac-Man to the ghost respawn point (if there is a single respawn point), or to the nearest ghost's respawn point, or the minimum distance from Pac-Man to any ghost respawn point.
`diagonalFromGhost`	A ghost's x and y coordinates are ± 1 of Pac-Man's.
`radiationStrength`	The "redness" of grid positions, or distance from a radiation center.
`humansSaved`	Either the absolute count or percentage of humans saved.
`unavoidable`	It's necessary to traverse a given cell, to reach all the humans.
non-Markov	The participant's description of the "bad" situation referred to his previous action.
state-action constraint	The participant described a situation where a particular action should not be used.
hierarchical constraint	The participant described a situation in which a particular *option* (as opposed to *primitive action*) should not be used.

Table 8: Radiation World features/criteria indicated by participants; criterion descriptions are given in Table 7.

Criterion	Participant IDs	% Participants Mentioning
`playState`	1, 2	25%
`radiationStrength`	1, 2, 3, 4, 5, 6, 7, 8	100%
`unavoidable`	1, 3	25%
`humansSaved`	2	12%
state-action constraint	5	12%

Table 9: Pac-Man features/criteria indicated by participants; criterion descriptions are given in Table 7.

Criterion	Participant IDs	% Participants Mentioning
`playState`	1, 4	25%
`distanceToGhost`	1, 4, 5, 7	50%
`surrounded/ stuckInCorner`	1, 2, 3, 4, 5, 6, 7, 8	100%
`dotsRemaining`	2	12%
`energizersRemaining`	2, 6	25%
`hardToReachEnergizer`	5	12%
`distanceToRespawnPoint`	6	12%
`diagonalFromGhost`	8	12%
state-action constraint	5, 6	25%
non-Markov	2, 5	50%
hierarchical constraint	5, 6	25%

State-action constraints. Although this dissertation defines and utilizes state-action constraints, the user interface used in this user study is only able to capture state constraints, not state-action constraints. An example is: "Avoid eating energizers when no ghosts are nearby," indicated by Participant #5.

Non-Markov constraints. A non-Markov constraint would be any which depends on a past sequence of states or state-actions, rather than only the current ones. An example is: "Any time I had just eaten a ghost, but it respawned within 2 squares, I marked that as a bad state," indicated by Participant #1. This constraint is defined over multiple time steps, something we haven't explicitly addressed.

Hierarchical constraints. These constraints are phrased in terms of avoiding taking a particular *option*, or avoiding an outcome we might associate with a particular option. An example is: "Avoid eating ghosts near the ghost respawn point, because they may eat you back." If "eating ghosts" is considered to indicate an option, then this constraint is defined over options, a straightforward extension to constraints over primitive actions, but not one that we have implemented here.

9.4 Phase 4: Feature Extractor Implementation

Our analysis of the criteria provided by participants yielded four potential features in the Radiation World domain (Table 8), and nine in the Pac-Man domain (Table 9).

For Radiation World, we implemented a feature extractor for the most popularly mentioned feature, `radiationStrength` (Listing 8), directly extracting it from Radiation World's structure state (Listing 5). For Pac-Man, the most popularly mentioned feature was `surrounded/stuckInCorner`; but given that Pac-Man exists in an enclosed space with the ghosts, we felt that any computable definition of what it means to be "surrounded" or "cornered" by ghosts could quickly become convoluted. On these grounds, we chose to disregard this criterion, for this experiment. It might have been the case that with additional discussion, the participants could have provided a operational definition of these terms, but because Phase 3 was conducted after they had left, they weren't given the opportunity. Instead of re-interviewing the participants to define these terms, we simply chose the next most popular feature: `distanceToGhost` (Listing 9). As we will see in Section 9.7, even this simplified approach (reliance on a state feature that was *not* the most-requested) yielded improved learning outcomes.

9.5 Phase 5: Constraint Decision Tree Training

The feature extractors we constructed in Phase 4 are given in Listings 8 and 9. These feature extractors use the states states visited by the participants in Phase 1 as inputs; their outputs, together with a training label chosen according to whether a given state had been marked as "bad" or not, are used as inputs to the WEKA J48 decision tree learning algorithm [12, 28] to create a classifier which categorizes input states as `Avoid` or `Unspecified`. Figures 19 and 18 show examples of trees learned, and Tables 11 and 10 summarize the decision trees learned for each participant.

```
def radiationStrength(state: S): Double = state.radiationMap(state.agent.position)
```

Listing 8: `radiationStrength` feature extractor for Radiation World.

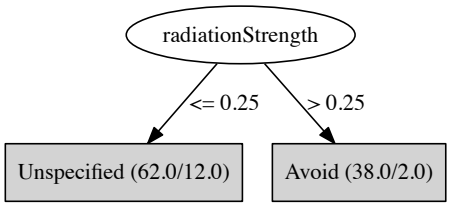


Figure 18: A learned Radiation World state constraint.

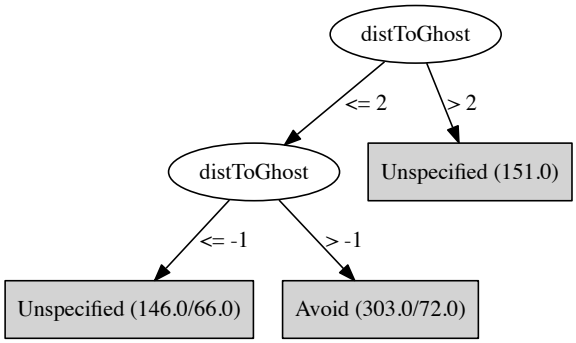


Figure 19: A learned Pac-Man state constraint.

Table 10: "Bad" `radiationStrength` values learned from participants' exemplars. A constrained range of "= 0.50" means that a state is classified as `Avoid` if the grid cell in which the agent is located has a radiation strength of exactly 0.50; the state is classified as `Unspecified` otherwise.

Constrained Range	Participant IDs
> 0.00	1
> 0.25	2, 4, 5, 6, 8
$= 0.50$	3, 7

Table 11: "Bad" `distanceToGhost` values learned from participants' exemplars. A constrained range of "0–2,10" means that a state is classified as `Avoid` if the distance to the nearest dangerous ghost is 0, 1, 2, or 10. A state is classified as `Unspecified` if there are no dangerous ghosts (distance of -1) or if the distance to the nearest ghost is in the range 3–9, or ≥ 11.

Constrained Range	Participant IDs
0	4
0–2	5
0–2,10	2
0–3	1
1–2	6, 7, 8
≤ 2	3

```
def distanceToGhost(state: S): Int = {
  val dangerousGhostPositions: Set[Position] = inedibleGhostPositions(state)

  // if there are dangerous ghosts on the board, then return the length of
  // the path to the nearest one
  if (!dangerousGhostPositions.isEmpty)
    pathToNearest(pacmanPosition(state), dangerousGhostPositions, state.walls).length
  else -1 // if there are none, there is no path; so return this sentinel value
}

// where pathToNearest is defined as:
def pathToNearest(start: Position,
                  targets: Set[Position],
                  walls: Set[Position]): List[Position] = // ...
  // A* search from `start` to any `target`, avoiding `walls`
```

Listing 9: `distanceToGhost` feature extractor for Pac-Man.

Two of the learned constraints stand out in particular: Participant 2's Pac-Man constraint, "avoid `distanceToGhost` $\in [0, 2] \cup \{10\}$", included the value 10, because the participant had demonstrated some "bad" states which were unrelated to `distanceToGhost`; specifically, he had demonstrated that consuming all of the energizers in quick succession was bad, a concept which could not be encoded using our interface. These exemplars are seen as noise with respect to our classifier's simple feature space. This might be improved by incorporating more features into the classifier, although there are tradeoffs there, in terms of feature space complexity versus training data set size.

Participant 3's Pac-Man constraint, "avoid `distanceToGhost` ≤ 2" stands out because it includes the sentinel value -1, which represents the absence of inedible ghosts. The constraint includes this sentinel value because the participant had marked several states in which all the ghosts were edible as "bad". This condition occurs whenever Pac-Man eats an energizer, so we can therefore expect that this constraint will bias Pac-Man against eating energizers, and reduce his possible reward as a result. In retrospect, we believe that we did not effectively explain the implications of Pac-Man eating energizers to this participant, and he therefore would not distinguish between edible and inedible ghosts. When we explained the user interface's Snapshot controls to each participant, we verified their understanding by asking them to explain the four buttons back to us; however, we did not verify understanding

of the game dynamics. Perhaps if we had verified participants' understanding of the game dynamics as well, Participant 3's constraint would not have included the sentinel value.

9.6 Phase 6: From State to State-Action Constraint

This phase does not yield a human-interpretable result, but is listed here for completeness.

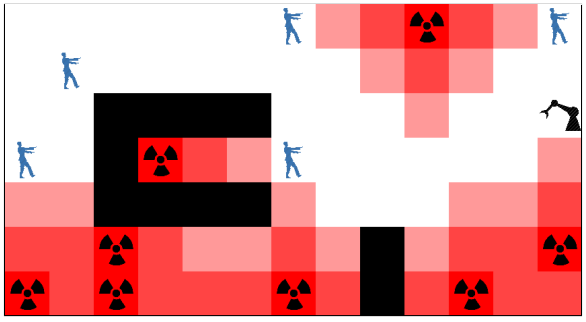
9.7 Phase 7: Q Learning Evaluation

To evaluate the effect of our user constraints on Q learning performance, we performed 200 independent trials in each of 52 different experimental conditions (Figure 17), varying domain, board, primitive actions only or with options, with each distinct user-derived constraint as well as without constraints.

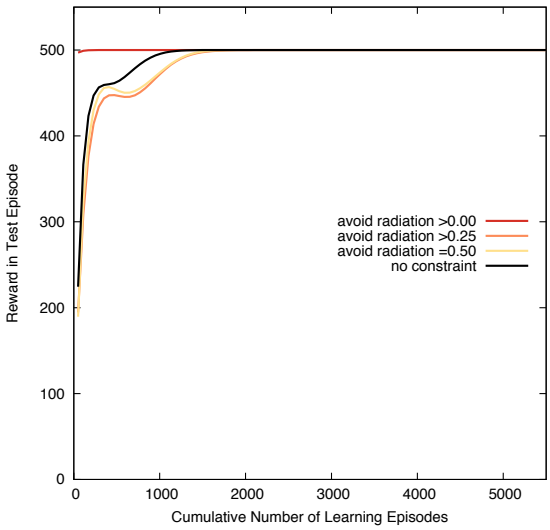
9.7.1 Q learning Evaluation in Radiation World

Figures 20, 21, and 22 illustrate the performance of the constraints learned from user exemplars, in three successively more complex Radiation World domain instances. These evaluations use a ϵ-greedy υ-defiant exploration policy where $\upsilon = [0.8 \text{ to } 0]$, $\epsilon = [0.4 \text{ to } 0]$ over 6000 episodes, and use a greedy constraint-biased evaluation policy.

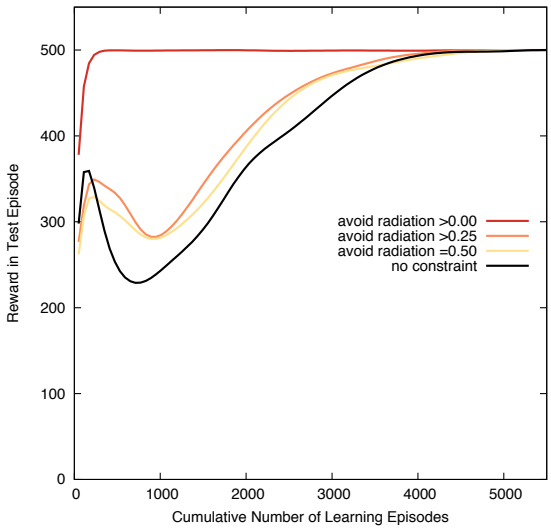
Passing through the irradiated areas increases the agent's risk of failure, so for the learners utlizing weaker constraints (e.g., avoid `radiationStrength` > 0.25, or avoid `radiationStrength` $= 0.50$) or using no constraints, early successes are more likely to involve passing through irradiated areas. A reinforcement learner generally doesn't have any way to recognize a poor best-known-solution (a poor best-known-solution is called a *local optimum*) until it has the experience of discovering a better one, and it aims to reproduce its earliest successes. The learner thus focuses exploration around this local optimum (rescuing humans using a path that takes it through radiation), causing performance to *decrease* over time, as the agent becomes increasingly efficient at finding the suboptimal path. The learner can eventually recover from this mistake and find an optimal policy, but only if it performs sufficient additional exploration. An effective constraint can prevent the learner from finding and wasting time near local optima, and we can see in Figures 20, 21, and 22, that the learners using the most conservative constraint (avoid `radiationStrength` > 0.00)



(a) Radiation World Board A

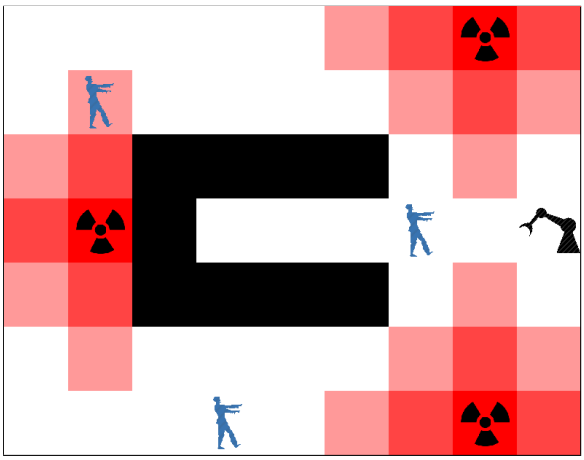


(b) Primitive actions only

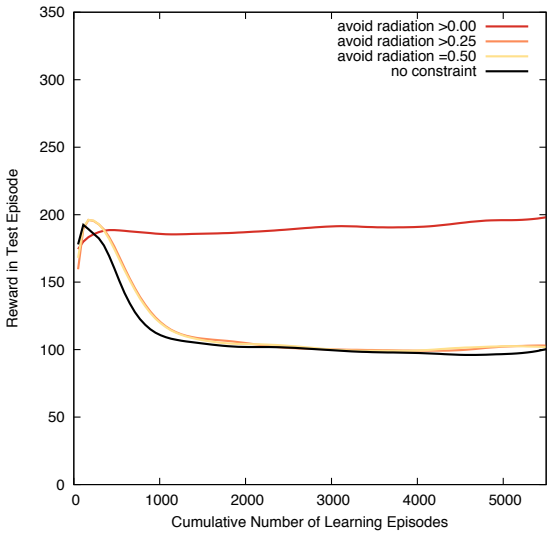


(c) Options + lifted primitive actions

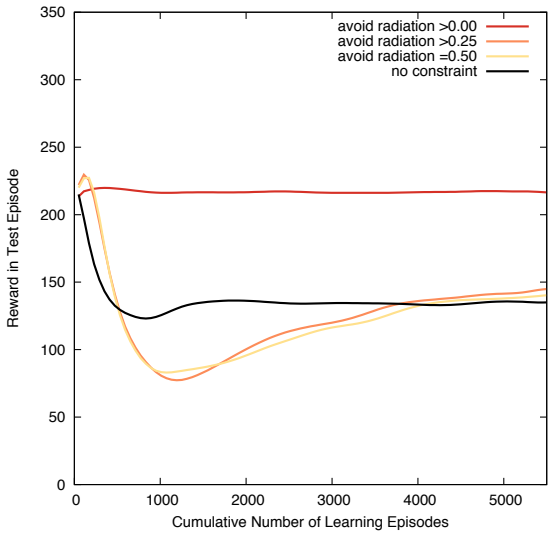
Figure 20: Average reward over 200 independent trials on Radiation World Board 1. Exploration policy: decaying ϵ-greedy υ-defiant (υ = [0.8 to 0], ϵ = [0.4 to 0] over 6000 episodes); Evaluation policy: greedy constraint-biased. The all-optima-preserving constraint "avoid `radiationStrength` > 0" helps the agent avoid local optima.



(a) Radiation World Board 2

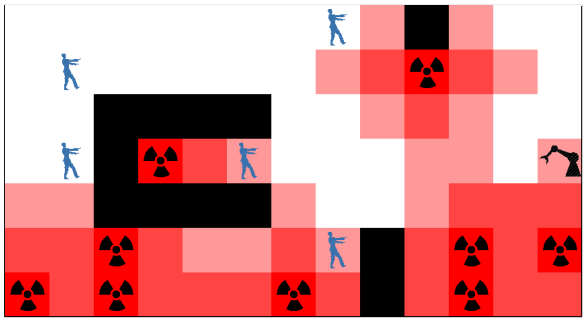


(b) Primitive actions only

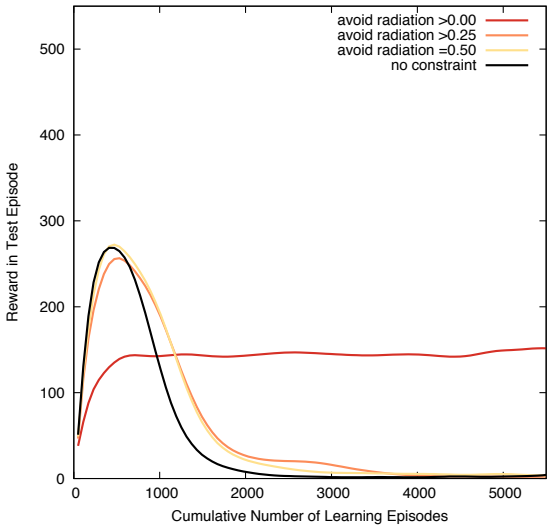


(c) Options + lifted primitive actions

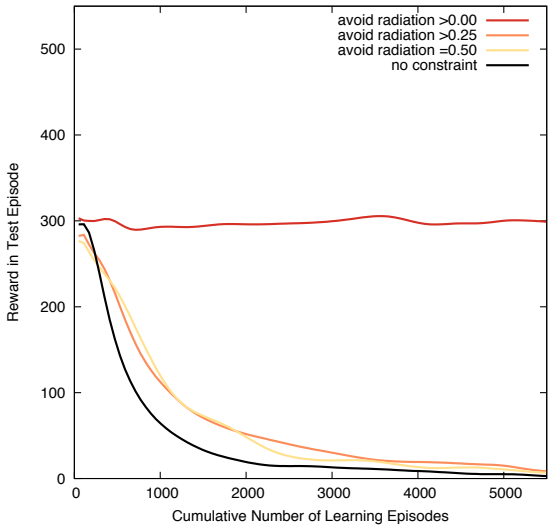
Figure 21: Average reward over 200 independent trials on Radiation World Board 2. Exploration policy: decaying ϵ-greedy υ-defiant (υ = [0.8 to 0], ϵ = [0.4 to 0] over 6,000 episodes); Evaluation policy: greedy constraint-biased. The "avoid `radiationStrength` > 0" constraint is not optimum-preserving but, in conjunction with a ϵ-greedy υ-defiant policy, still helps the agent avoid local optima.



(a) Radiation World Board 3



(b) Primitive actions only



(c) Options + lifted primitive actions

Figure 22: Average reward over 200 independent trials on Radiation World Board 3. Exploration policy: decaying ϵ-greedy υ-defiant (υ = [0.8 to 0], ϵ = [0.4 to 0] over 6,000 episodes); Evaluation policy: greedy constraint-biased.

are best protected from getting caught up in them.

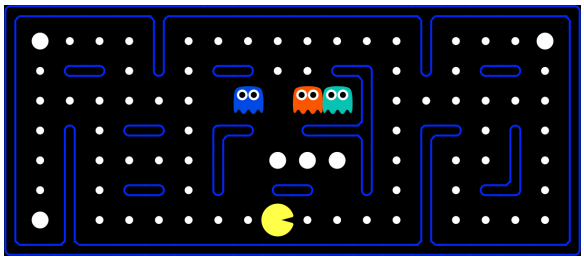
When options are used, this effect is even more dramatic (Figures 20c, 21c, 22c), for two reasons. First, recall that rescuing a human is the only source of reward in Radiation World. This means that although it won't necessarily have the highest long-term utility, the `gotoNearestHuman` option has the highest expected *immediate* reward of any action. This means that any early uses of `gotoNearestHuman` creates a local optimum. Second, when either of the weaker constraints is used to construct a constrained option from `gotoNearestHuman`, the resulting option is a bit dangerous: it may lead the agent into light radiation on the way to rescuing a human, and then terminate, stranding the agent in a dangerous spot without any guidance for how to extricate itself. The strictest constraint (avoid `radiationStrength` > 0.00) produces a safer constrained option, which stops the agent *before* it enters the radiation, instead of just afterward.

On Radiation World Board 3 with primitive actions only, the agent using the "avoid `radiationStrength` > 0.00" constraint is likely to be trapped by the light radiation in the top-right of the board (Figure 22a), leading to limited reward (Figure 22b). With the `gotoNearestHuman` option and υ-defiant exploration, the agent is able to discover the path through the radiation, leading to higher expected reward (Figure 22c).

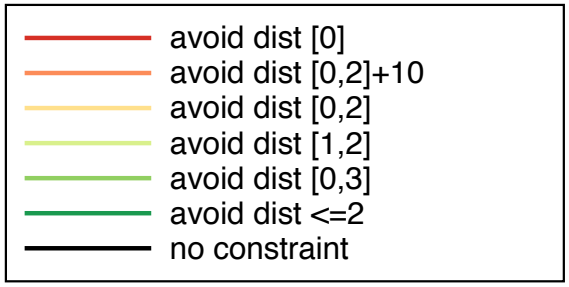
9.7.2 Q learning Evaluation in Pac-Man

The results of learning with user-derived constraints in Pac-Man are illustrated in Figures 23 and 24. These evaluations use the same decaying ϵ-greedy υ-defiant exploration policy where $\upsilon = [0.8 \text{ to } 0]$, $\epsilon = [0.4 \text{ to } 0]$ over 11000 episodes, and use the same greedy constraint-biased evaluation policy as in the Radiation World evaluations. We see that the most conservative constraint, "avoid `distanceToGhost` $= 0$" constraint, yields the highest performance in all cases.

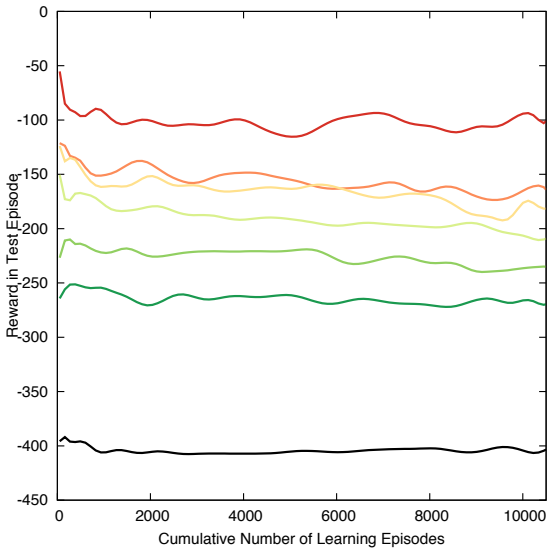
It is gratifying to see that the combination of constraints and options (Figure 23d and 24d, colored series) generally yields much higher performance than the use of constraints or options alone (Figures 23c and 24c, colored series; vs Figures 23d and 24d, black series.) Note the different scales of the y-axes.



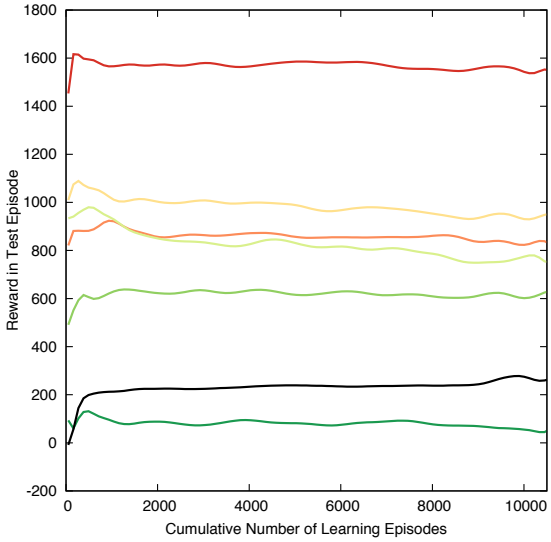
(a) Pac-Man Board 1



(b) Legend

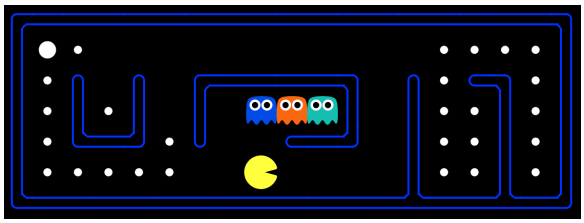


(c) Primitive actions only

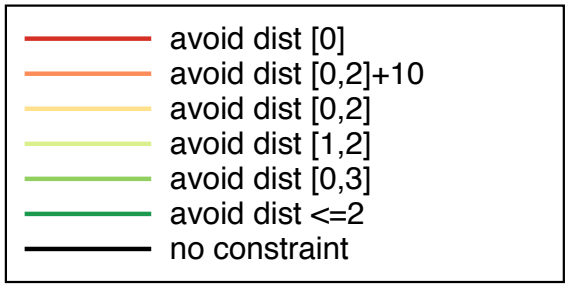


(d) Options + lifted primitive actions

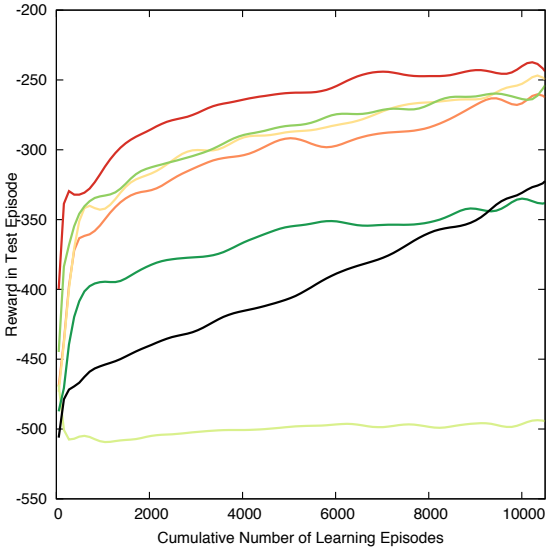
Figure 23: Average reward over 200 independent trials on Pac-Man Board 1.
Exploration policy: decaying ϵ-greedy υ-defiant (υ = [0.8 to 0], ϵ = [0.4 to 0] over 11,000 episodes); Evaluation policy: greedy constraint-biased.



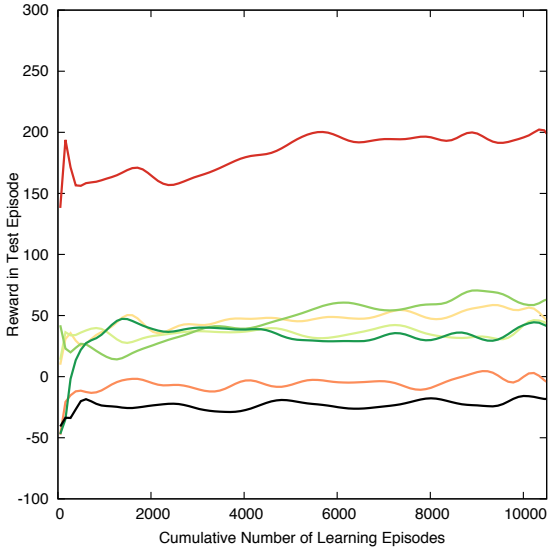
(a) Pac-Man Board 2



(b) Legend



(c) Primitive actions only



(d) Options + lifted primitive actions

Figure 24: Average reward over 200 independent trials on Pac-Man Board 2.
Exploration policy: decaying ϵ-greedy υ-defiant (υ = [0.8 to 0], ϵ = [0.4 to 0] over 11,000 episodes); Evaluation policy: greedy constraint-biased.

As we anticipated in Section 9.5, we can see in Figure 23d that Participant 3's "avoid `distanceToGhost` ≤ 2" constraint does severely limit performance by crippling the `goToNearestEnergizer` option. Figure 24d shows poor performance when using the "avoid `distanceToGhost` $\in [1, 2]$" constraint produced by Participants 6, 7, and 8. On Pac-Man Board 2, because the board is relatively small, the agent can get into situations where it has to choose between actions that would result in `distanceToGhost` $= 1$ vs 0 (a loss yielding -500 reward). The "avoid `distanceToGhost` $\in [1, 2]$" constraint mistakenly prefers the loss, leading to poor reinforcement learning performance. Again, we believe that both of these situations could be avoided with clearer instructions to participants; verifying their understanding of the domain in the former case, and explicitly encouraging them to label failure states as "Bad" in Phase 1 in the latter case.

9.8 Summary of Findings

We have seen through this chapter's experiment that even given imperfect instructions to participants; an imperfect ability to capture participants' desired constraints; quick and dirty feature extractors not relying on AI expertise, but on layperson recommendations; and an unsophisticated decision tree instance weighting scheme, we still see distinctly improved performance by modeling negative policy information in our Q learning.

Averaged over the Radiation World scenarios tested, 85% of user-derived constraints (1 out of 8 in Figure 20b, and 8 out of 8 in each of the other five scenarios) outperformed baseline within the 6,000 learning episode window, and none underperformed baseline. Averaged over the Pac-Man scenarios tested, 84% of user-derived constraints (7 out of 8 in Figure 23d, 4 out of 8 in Figure 24c, and 8 out of 8 in each of the other two scenarios) outperformed baseline within the 11,000 episode window, and 16% underperformed baseline.

In these domains tested, the strictest constraints, "avoid radiation > 0.00" and "avoid `distanceToGhost` $= 0$", yielded the highest performance. In Radiation World, which has sparse rewards, a strict constraint was helpful in avoiding local optima. For Pac-Man, the combination of options with constraints yielded much higher performance than either options or constraints alone. On the more cramped Pac-Man Board 2, learning was more

sensitive to overly-broad constraints such as the "avoid `distanceToGhost` $\in \{0, 1, 2, 10\}$" constraint. Data collected from laypeople could be more effective by instructing them that it is helpful to mark failure states as "bad", and by verifying their understanding of the task prior to data collection.

In the next chapter, we will see if we can reproduce a similar level of success in the more complex Mario World domain.

CHAPTER X

STATE-ACTION CONSTRAINTS FROM SOFTWARE ENGINEERS

Beyond the grid-based domains we investigated in the previous chapters, we want to know if we can get the same good results with larger systems. As an example, we will be attempting Q learning in the Mario World domain of Section 7.3.

In many cases, it may be more practical to take machine learning out of the process of constructing constraints, and instead implement a constraint (or option) using a general programming language. In terms of our previous experiment, we can directly write a state-action constraint using an appropriate programming language, effectively receiving the output of Chapter 8, Figure 14, Phase 6 directly from software engineers (Figure 25). We do not claim this as a universal solution, but it is clearly appropriate in certain cases. Structured state representations (Section 3.3) are currently more easily interpretable by humans than by machines. Whenever it would take less effort to communicate the constraint using a programming language, as compared to constructing and utilizing a means to capture examples or other data from humans and then providing the examples or other data, it is worth considering this direct approach.

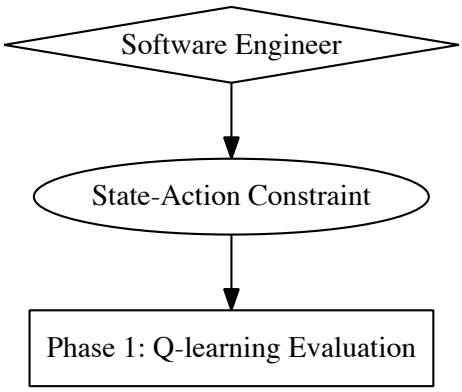


Figure 25: Inputs and outputs (circles), phases (rectangles), dependencies (arrows), and humans (diamonds), in the experiment with constraints from software engineers.

The software-engineered constraints presented here do not include any special knowledge of artificial intelligence, reinforcement learning, or anything beyond what is routinely covered in introductory programming courses. For example, the constraints learned in Chapter 8 are implemented very succinctly in Listings 10 and 11, without the rigamarole of building a specialized interface and scheduling appointments with freshmen.

```
def avoidRadiationOver(maxRadiation: Double) =
  StateConstraint[RadWorldState] { s =>
    if (s.radiationStrength(s.agent.position) > maxRadiation)) Avoid
    else Unspecified
  }

val avoidRadiationAbove025: StateConstraint[RadWorldState] =
  avoidRadiationOver(0.25)
```

Listing 10: Avoid radiation above a certain threshold (0.25 in this case).

```
def avoidGhostDist(badDistances: Set[Int]) = StateConstraint[PacmanState] {
  state =>
    val dangerousGhosts: Set[Position] =
      state.ghosts.filterNot(isEdible).map(ghostPosition).toSet

    if (dangerousGhosts.isEmpty)
      Unspecified
    else if (badDistances contains pathLengthToNearest(
                start = state.pacman.position,
                targets = dangerousGhosts,
                walls = state.walls))
      Avoid
    else
      Unspecified
}
```

Listing 11: Avoid dangerously inedible ghosts closer than a certain threshold.

Removing the supervised learning component from the equation simplifies system building, but obviously it does not guarantee perfect results. A programmer may have the same intuition as some of our study participants and believe that

```
avoidGhostDist(Set(1,2))
```

will be the most effective constraint (Listing 11), whereas Figures 23 and 24 showed that

constraint to be less effective than `avoidGhostDist(Set(0))` was. By directly encoding constraints using a general programming language, we do away with one source of error (everything entailed by supervised learning) but add another (conceptual or programmer error). Ideally, we can use whichever method is easier for a given domain on a constraint-by-constraint or concept-by-concept basis.

10.1 Engineering Challenges in Mario World

A fair bit of engineering goes into making experimentation possible in any domain, and this is acutely true in the case of Mario World. Using tabular Q learning without function approximation in the Mario World domain is quite challenging given finite computing resources. Within the first 2,000 learning episodes, we encounter up to three million unique states, even for very small problem instances. The size of the state representation is quite large as well: approximately 18 kilobytes per state, even on very small problem instances, for a total of 3000000 states $\times$ 18kB/state $=$ 54GB, excluding a number of types of overhead.

We modified the Mario AI Competition source code to eliminate global mutable data, to support multiple simultaneous simulations, and to support resuming a simulation from an arbitrary state. By performing extensive code analysis to reduce the memory footprint of the simulation and states and to detect and share certain identical and near-identical data structures within the state representation, we reduced the average state size by over 50% to 8.3 kilobytes without omitting any information. By memoizing repeated hashing operations, we are able to eliminate 100 minutes of hashing per 20 minutes of execution. By aggressively using interning (comparison and substitution) of commonly identical objects, we are able to share about 70% of Sprite objects (representing enemies or other moving entities in the Mario World simulation) and about 90% of Level objects (representing the map layout itself, as well as the position and condition of enemy spawn points), and were able to reduce the average state size to 1.4 kilobytes, at the expense of reduced parallelism, as the shared interning cache becomes a bottleneck. Now we are ready to do some experiments with very small Mario World problem instances.

All of this is simply meant to underscore the fact that large, interesting systems already

necessitate serious engineering. It is no problem at all to ask such competent developers to also build options, constraints, or simple feature extractors, using the arbitrarily expressive language they are already familiar with. Any of these is simply another function to write. Moreover, the software engineer can use his or her preferred language to bridge any conceptual gap between the desired constraint and the simple state or state-action representation. Exotic constraints such as rankings of action preferences, e.g., "take action a_1 unless x, in which case take action a_2 unless y, in which case take action a_3", can be programmatically adapted to the state-action format. By contrast, such constraints might be quite difficult to learn by from examples without substantial experience in machine learning and classification problems.

10.2 Developing Constraints as a Software Engineer

Our first step is to observe the behavior of a Q learning Mario agent without constraints. He doesn't seem to do anything useful. He mostly seems to jump around in place helplessly—which makes sense, since one half of the action space involves jumping. With this observation, we can begin designing constraints. It is worth noting that the definitions of these three constraints do not involve any concepts from artificial intelligence or machine learning!

Constraint #1. If there are no enemies nearby to interact with, avoid jumping, moving left, or staying still.

```
def constraint1 = StateActionConstraint[MarioState,MarioAction] {
  s => a =>
    if ((hasLeftComponent(a.direction) || a.jump || isIdle(a.direction))
      && {
      val mario = s.getMario
      val enemies = s.getEnemySet
      def distFromMario(sp: MSprite) =
        (mario.getPosition.asVector - sp.getPosition.asVector).magnitude
      !enemies.exists(e => distFromMario(e) < 50)
    }) Avoid else Unspecified
}
```

Listing 12: Constraint #1: Avoid exploring certain actions when no enemies are nearby.

Constraint #2. Don't explore the `Up` sub-action; we believe it will only have an effect if

Mario is at a ladder, but we have not observed any ladders in the Mario World simulation.

```
def constraint2 = StateActionConstraint[MarioState,MarioAction] {
  s => a => if (hasUpComponent(a.direction)) Avoid else Unspecified
}
```

Listing 13: Constraint #2: Don't use the `Up` subaction.

Constraint #3. Avoid colliding with enemies. This constraint has by far the most complex implementation. It uses a simplified version of the simulation's collision code to try to avoid fatal collisions that would occur on the very next frame.

A side note: One might expect, as we did, that Constraint #3 would make Mario effectively invincible, but it presently has little to no effect: most fatal collisions can't be avoided in the immediately preceding time-step, and our agent does not have the ability to reason about the implications of constraints multiple time-steps in the future. Of the three constraints, the one which in fact imparts the most apparent intelligence to the agent is Constraint #1. It causes Mario to walk along purposefully until encountering an enemy, and then clumsily attempt to stomp on it. The reality is that once an enemy is closer than 50 pixels from Mario, the constraint no longer applies at all; Mario reverts to his uninformed, random behavior which involves a lot of jumping.

Again, note that none of these constraint definitions involve any AI-specific concepts; they are basic, run-of-the-mill functions, which emit either `Avoid` or `Unspecified`.

10.3 Evaluation

To evaluate the effect of our software-engineered constraints on Q learning performance, we performed 24 independent trials in each of 6 different experimental conditions (Figure 26), varying domain instance and υ. Instance 1 has only flat terrain; there are no obstacles, apart from enemies. Instance 2 is the same, except that it includes uneven terrain. Our constraint was composed of the union of constraints #1, #2, and #3, above, and we measured the performance as described in Section 7.4, with `testFrequency` $= 1/20$ and a ϵ-greedy υ-defiant policy (described in Section 6.3.1) with fixed ϵ and υ.

$$\{\text{Mario World}\} \times \begin{Bmatrix}\text{instance 1}\\ \text{instance 2}\end{Bmatrix} \times \{\text{primitive actions only}\} \times \begin{Bmatrix}\upsilon = 0.0\\ \upsilon = 0.8\\ \upsilon = 1.0\end{Bmatrix}$$

Figure 26: Cartesian product of conditions evaluated in this experiment. All conditions use a fixed $\epsilon = 0.2$.

We performed far fewer learning episodes and far fewer independent trials in the Mario World domain, as compared to in the Radiation World or Pac-Man domains only because, despite our optimizations, the Mario World simulation state size was still so large that we exhausted the available memory (64GB) on our compute server after even this relatively low number of episodes. Furthermore, because the total number of learning trials was much lower, we wanted a higher evaluation frequency, in order to produce enough evaluation samples for a respectable graph; however, as we increase the ratio of evaluations, we are also rapidly increasing the total running time of an already slow experiment. As a compromise, we evaluated the outcomes of the learning episodes themselves instead. Finally, unlike in the previous experiments, we chose not to decay ϵ or υ; as to do so would indicate a confidence that we will likely already have found a near-optimal policy. Having no such confidence in this vastly larger domain, we chose to keep ϵ and υ constant. We would imagine, however, that a greedy constraint-biased policy would produce results similar to the $\upsilon = 1$ agent we do evaluate.

10.4 Results and Observations

Figure 28 illustrates the performance of three agents utilizing an ϵ-greedy υ-defiant policy for exploration and evaluation in Mario World instance 1, which is flat (no hills or pits). Again, the vast majority of states are not encountered more than once, so we are relying much more heavily on our constraints to provide good performance than on Q learning; this is why the learning curves are essentially flat within the window we were able to observe before running out of memory.

One agent uses $\upsilon = 0$, and thus always ignores our constraints, and consistently achieves

close to the minimum possible cumulative reward.[1] A second agent uses a fixed $\upsilon = 1$, and although this agent falls far short of the ideal score of approximately 1000, it achieves the highest performance of the three. A third agent uses a fixed $\upsilon = 0.8$, and thus uses mixed policy which follows constraints 80% of the time and ignores them 20% of the time. The $\upsilon = 0.8$ agent receives a score that falls between the $\upsilon = 0$ and $\upsilon = 1$ agents, indicating that following the constraint is more effective than not.

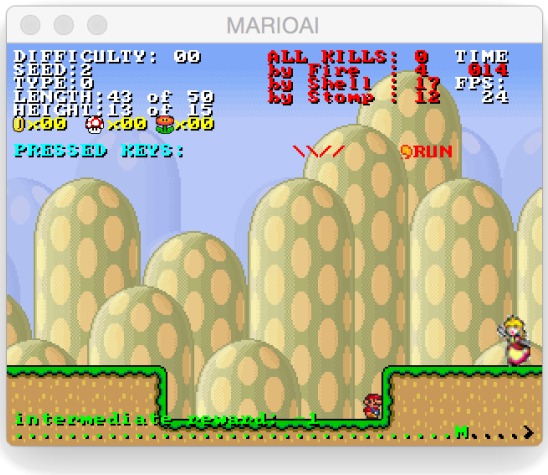
In Mario World instance 2, which contains pits, Constraint #1 is revealed to contain a liability — Mario can become trapped in a pit, because we believed that jumping was unproductive in the absence of enemies. (See Figures 27a, 27b, and 27c.) Figure 29 shows $\upsilon = 0.8$ outperforming $\upsilon = 1$, reaffirming that some deviation from the constraints must be allowed if there's any doubt as to whether a constraint is optimum-preserving.

It's ironically rewarding to see that even the known-suboptimal policy produced by always adhering ($\upsilon = 1$) to the non-optimum-preserving constraint in Mario World instance 2, the agent's performance still greatly outperforms the baseline Q learner.

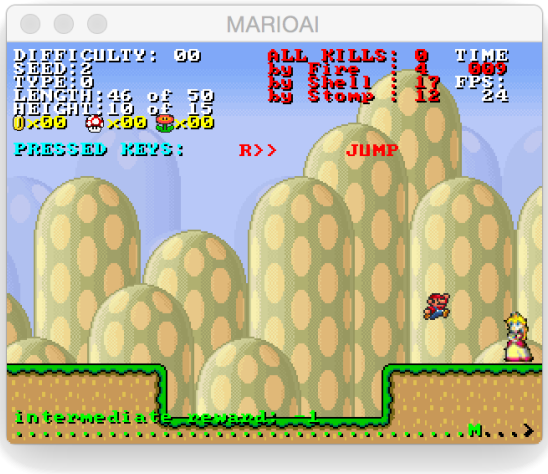
10.5 Summary of Findings

In this chapter's experiment, we saw that the use of constraints built through straightforward software engineering resulting greatly improved Q learning performance. In cases where a software engineer can envision a desired constraint, he can likely proceduralize it easily. Alternatively, a layperson may articulate a desired constraint to a software engineer, who can implement it quickly as in Listings 10 and 11, without building a user interface for capturing exemplars, and without knowing how to use those exemplars to train a classifier-based constraint.

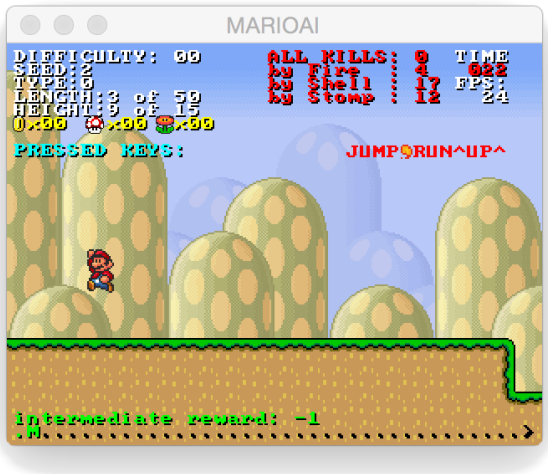
[1] The minimum possible cumulative reward is -512 (for timeout) + $(-1) * 750$ (step costs) -42 (if hurt on the final time-step) $= -1304$.



(a) Constraint #1 prevents the $\upsilon = 1$ agent, which always follows constraints absolutely, from escaping this pit.



(b) The $\upsilon = 0.8$ agent escapes the pit by defying the constraint.



(c) The constraint-ignoring $\upsilon = 0$ agent has not progressed beyond the start of the level.

Figure 27: Varying the value of υ in Mario World instance 2, which contains pits.

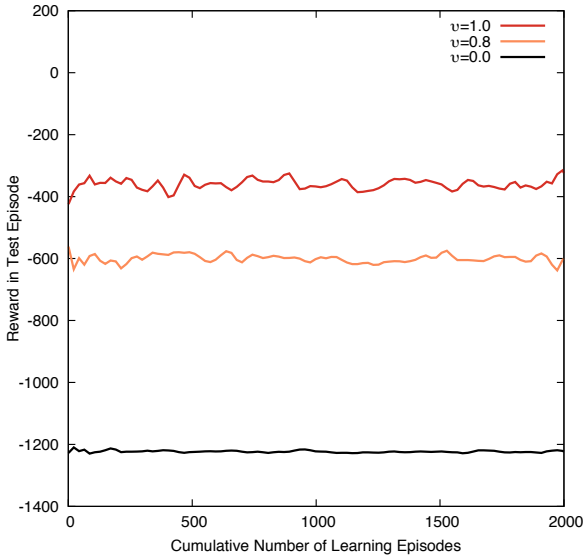


Figure 28: Mario World instance 1 (flat terrain). The $\upsilon = 0$ agent (black) ignores constraints, $\upsilon = 1$ (red) follows constraints absolutely, and $\upsilon = 0.8$ (orange) follows a mixed policy. In each case, the same policy is used for exploration and evaluation.

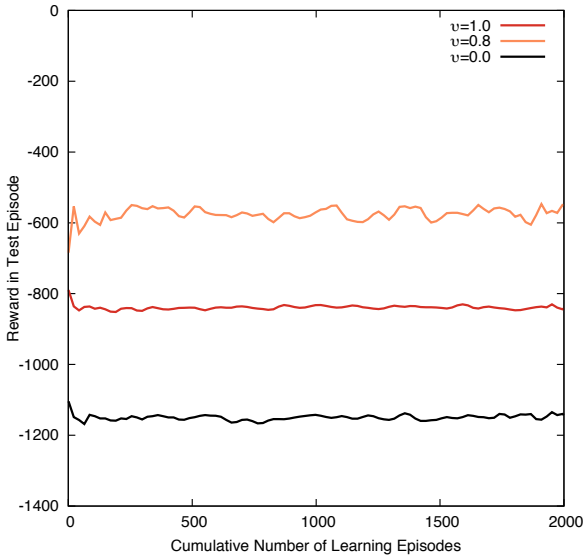


Figure 29: Mario World instance 2 (uneven terrain). The $\upsilon = 0$ agent (black) ignores constraints, $\upsilon = 1$ (red) follows constraints absolutely, and $\upsilon = 0.8$ (orange) follows a mixed policy. In each case, the same policy is used for exploration and evaluation.

```
def constraint3 =
  StateActionConstraint[MarioState,MarioAction] {
    state => action =>

      def marioNextPos(m: MMario) = m.position + nextVelocity(m, action)
      def enemyNextPos(e: MEnemy) = e.position + e.velocity

      def marioNextBB(m: MMario) = BoundingBox(marioNextPos(m), m.size)
      def enemyNextBB(e: MEnemy) = BoundingBox(enemyNextPos(e), e.size)
      def hasSafeTop(s: MEnemy) = s.getLabel match {
        case Spiny | SpinyWinged | PiranhaPlant => false
        case _ => true
      }

      def intersects(marioBB: BoundingBox,
                     enemyBB: BoundingBox): Boolean = {
        val Vector2(dx,dy) = marioBB.pos - enemyBB.pos
        // magic formula from simulation
        ((dx > -enemyBB.size.x * 2 - 4 && dx < enemyBB.size.x * 2 + 4)
          && (dy > -enemyBB.size.y && dy < marioBB.size.y))
      }

      def badCollide(m: MMario, e: MEnemy): Boolean = {
        val marioBB: BoundingBox = getMarioNextBB(m)
        val enemyBB: BoundingBox = getEnemyNextBB(e)
        if (e.dead) false // not bad, enemy is already dead
        else if (intersects(marioBB, enemyBB)) { // we have collided
          val dy = marioBB.pos.y - enemyBB.pos.y
          // dy < 0 means mario above enemy, because (0,0) is top left
          if (mario.getVelocity.getY > 0 // falling
                && dy <= 0 // above enemy
                && (m.isAirborne || m.wasJustAirborne)) {
            !hasSafeTop(e) // top not safe to stop on? --> bad
          }
          else true // side contact
        } else false // no collision
      }

      if (state.enemies.exists(e => badCollide(state.mario, e))) Avoid
      else Unspecified
  }
```

Listing 14: Constraint #3: Avoid fatal collisions.

CHAPTER XI

CONCLUSION AND FUTURE DIRECTIONS

In this dissertation, we have discussed the need for and introduced a new formalism for modeling negative policy information as *state constraints* and *state-action constraints* (Chapter 5), which provide learning speedups by avoiding unfruitful exploration, and which represent a novel form of problem decomposition to complement existing decomposition abstractions.

We have provided a method for composing options and constraints to form constrained options (Section 6.2) which we later saw could provide significantly improved performance as compared to the use of either constraints or options alone. We presented the ϵ-greedy υ-defiant exploration strategy, which preserves optimality guarantees during exploration (Section 6.3.1), and the greedy constraint-biased evaluation policy, a constraint-biased version of a greedy evaluation policy (Section 6.3.2).

We have presented a scheme and reusable user interface for learning state constraints from examples constructed by laypeople in a semi-automated way (Chapter 8) and evaluated the Q learning performance of agents utilizing the constraints learned from these laypeople in the Pac-Man and Radiation World domains, finding small to extreme improvement with user constraints in 84% of cases (Chapter 9). We also demonstrated of software engineered state-action constraints in the Mario World domain (Chapter 10.1) which showed greatly improved performance compared to learning without constraints, without requiring specialized AI knowledge.

Still, we envision a number of avenues for future work, grouped here under three umbrellas: representational generalizations, adaptation of existing constraints, capturing new constraints in other types of domains, and architectural improvements.

11.1 Representational Generalizations

Preferences and constraint ensembles. The constraints introduced in this dissertation are binary: each state or state-action is categorized as `Avoid` or `Unspecified`. We have begun work on real-valued *preferences* $\phi : S \times A \rightarrow [0, 1]$ which represent the likelihood/estimated probability that action a should be taken in state s.

A stochastic policy $\pi(s, a)$ can be viewed as a preference directly; the components $\phi(s, \cdot)$ will then sum to 1. A deterministic policy would correspond to a preference where a single component has a value of 1, and the rest have values of 0. A constraint c can be viewed as a preference:

$$\phi_c(s, a) = \begin{cases} 0 & \text{if } c(s, a) \\ 1 & \text{otherwise} \end{cases} \tag{45}$$

In Chapter 5 we discussed the composition of constraints via set operations; a set of constraints (such as the set constraints collected from participants in Chapter 9) could instead be composed into a preference which represents the weighted votes of each constraint. This would have the effect that states or state-actions that people universally agree should be avoided would be avoided much more strongly during an agents exploration than states or state-actions which a few outliers indicate should be avoided (perhaps due only to noisy data or to poorly executed user orientation).

Hierarchical and semi-Markov constraints. The constraints introduced in this dissertation are defined over primitive actions. Just as an agent's policy can be defined over primitive actions or over options, constraints over options would be a natural extension. Additionally, like semi-Markov options, a constraint could be be defined over histories rather than over states. This would enable us to define constraints such as, "don't eat multiple energizers in a row," (suggested for the Pac-Man domain by User #5), or generally to avoid useless action sequences.

11.2 Capturing Constraints in Other Kinds of Domains

Capturing state-action constraints in turn-based domains. In Chapters 8 and 9, we captured *state* constraints from users in the turn-based Radiation World and Pac-Man domains. Because some concepts are more naturally represented as *state-action* constraints, one can imagine that an interface to capture state-action constraints could be useful as well.

An interface similar to ours might work well, provided that the user has a means to specify, "the thing I *just* did was something you should avoid," or, "given this scenario that I have constructed, the *next* thing that I do is something you should avoid."

The interface that we used provided an easy way to review the selected "bad" situations before finalizing a user's selections. Our participants did utilize this feature during the state-selection phase as well as during the desiderata description phase, and we believe that it made the system more effective. To extend this to support state-action constraints, it might be sufficient to display a visualization of the specified action alongside the specified state.

Capturing constraints in real-time domains. In real-time domains such as Mario World, the task of capturing constraints is again more difficult. In domains with such high action rates (15 actions per second in Mario World), a user has very little time to consider his choices. It will likely be necessary to provide a mechanism to annotate recorded traces after the fact, such as by "scrubbing" the recording at user-controlled speeds and, once the appropriate timespan is located, marking specific frames as "bad" situations or specific frame pairs as illustrating the before-and-after effects of "bad" choices. Domain-specific adaptations may be required.

Although we believe that capturing *state* constraints in this way would be effective, it may be that users (non-programmers specifically) will not be able to conceive of and communicate *state-action* constraints in real-time domains using a freeze-frame paradigm. In this case, real-time recordings might be taken, and an entire trace annotated as "good" vs "bad" behavior.

11.3 Adaptation of Existing Constraints

Learning state-action constraints from state constraints. In evaluating the state constraints derived from user data in our study, we relied on known transition probability functions for Pac-Man and Radiation World. As shown in Section 5.4, some approximation of the transition probability function (specifically, knowing zero vs nonzero probability for a given transition) is needed to perform this construction.

It would be possible for an agent to instead incrementally discover transitions and update its constraint accordingly. If an action is found to lead to a state that should be avoided, that action can be avoided in future constraint-respecting decisions. Furthermore, any state for which all possible actions are to be avoided should itself be avoided if possible in the future. In this way, sequences of actions which inevitably lead to states best avoided can be detected and avoided themselves over time.

Refining constraints using expert traces. We have discussed and seen how overly-broad constraints can limit performance (Sections 5.6, 9.7, and 10.4). While our method of learning state constraints from demonstrated failure states relied primarily on positive examples of failure states, and assumed unmarked states to include a mix of positive and negative examples of failure, one could imagine using traces of users' attempts at very good behavior as explicit negative examples of failure states; in other words, states which are Ok. These non-failure states could either be used as input to the same classifier which creates the constraints, instead of or in addition to the unlabeled examples, or applied after constraint creation as overriding corrections.

Conversely, since some constraints may only apply to specific sub-tasks, constraints could be used to accelerate option learning.

Refining constraints through transfer learning. Along the same vein, it may be possible to learn the domain conditions under which a particular constraint is unlikely to be optimum-preserving. This could be done either by solving a small domain instance quickly, determining where optimal actions disagree with the constraint, generalizing over

those conditions, and then ignoring the constraint in such conditions; that said, the success of this scheme will depend on the accuracy of the generalization used. Given multiple constraints, they are unlikely to be wrong under the same conditions, so they should be treated separately here.

11.4 Architectural Improvements

Heterogeneous State Abstractions. In this work, we have used the same state representation in the constraints, options, and learning agent. Ideally, each component could use a state representation independently derived from the native state representation, allowing various abstraction schemes within in the learner while retaining expressiveness in other components. The simulator, agent, individual options, and individual constraints could each utilize their own independently derived state representation, permitting maximum compactness where desired without inflicting information loss or state aliasing on other components. This is not entirely straightforward, since a component generally can't query A_s or O_s if it is only aware of a derived state representation; at the same time, only being aware of the derived state representation is important to be a truly modular component.

LRU-evicting Q tables for large state spaces. In this dissertation, we have dealt with very large state spaces, as well as spaces with very large states. Finite memory (in relation to growing Q table size) becomes the primary limiting factor. Since the majority of states (97% or more, in our domains) have not been visited more than once, retaining their respective Q-values may be unnecessary. It is worth exploring whether forgetting many of these values would have a negative impact on learning outcomes or if it would simply enable longer learning learning trials (more episodes) and produce more effective policies.

APPENDIX A

USER STUDY SCRIPT

A.1 Introduction

My name is Arya Irani. I'm a grad student in the Interactive Machine Learning research group, which currently includes students and faculty members from the College of Computing and the College of Engineering. We're trying to teach AI agents by giving examples of what a bad situation looks like.

In Part A, you'll have an opportunity to play one of two games until you feel familiar with it In Part B, I'll ask you to try to set up situations that are bad for the agent, situations he should ideally never get into, and youll take snapshots of the bad situations, using the snapshot controls Ill explain later. Try to come up with as many scenarios as you think would be helpful for teaching the AI what it should avoid. In Part C, I'll ask you to type up a brief explanation about the situations you picked; what it was about them that made them good examples of situations to avoid. We'll go through Parts A, B, and C for two games.

[Go over movement controls for first of two randomly ordered domains.]

A.2 Pac-Man Movement Controls

Up/Down/Left/Right: Pac-Man will try to move in this direction.

If there's no wall blocking that direction, he'll move there; if there's a wall, he will stay in the same place. If Pac-Man eats all the small pellets, he's done — even if there are big pellets remaining. If he eats a big pellet, the ghosts become scared, and there will be a short period of time that Pac-Man can eat the ghosts. Otherwise, if he runs into a ghost, he loses.

Any questions about the movement controls?

[Explain snapshot controls (Section A.4) if this is the first domain.]

Lets do a practice run first. I'll pull up a board and then ask you to follow my instructions.

- Move in any direction 5 times, and then Reload Initial Configuration.
- Move three times, and Mark the configuration (1) as bad.
- Undo twice, and move in a different direction from before.
- Run into a ghost (lose), Mark that situation as bad (2).
- Select configuration (1) and delete it.
- Select configuration (2), undo once, and move in a different direction.

[Do real run.]

[Open text editor.]

Could you type up a few words about how you selected the "bad" situations, or how you could tell if a situation was "bad"?

A.3 Radiation World Movement Controls

Up/Down/Left/Right: The robot will try to move in that direction.

When the robot is very close to a radiation symbol, theres a chance that it will screw up and go in a random direction. The regions around the radiation symbol are colored red the darker the shade of red, the greater the chance that the agent will move in a random direction.

If there is a wall in the direction that the agent moves, the robots position will not change. Once the robot reaches all of the humans scattered around the map, its done. If it ends up at the radiation symbol, it loses.

Any questions about the movement controls?

[Explain snapshot controls (Section A.4) if this is the first domain.]

Let's do a practice run. I'll pull up a broad and then ask you to follow my instructions.

- Move in any direction 5 times, and then Reload Initial Configuration.
- Move three times, and Mark the configuration (1) as bad.
- Undo twice, and move in a different direction from before.
- Run into a radiation source (lose), Mark that situation as bad (2).
- Select configuration (1) and delete it.
- Select configuration (2), undo once, and move in a different direction.

[Do real run.]

[Open text editor.]

Could you type up a few words about how you selected the "bad" situations, or how you could tell if a situation was "bad"?

A.4 Snapshot Controls

Reload initial configuration This button will reset the board (but not the snapshot collection) back to the initial configuration.

Mark this configuration as Bad / B This will take a snapshot of the current situation, and it will appear in the snapshot area along the bottom. You can take a snapshot even of a configuration where you've already finished or lost.

Undo one action / U You can undo your last movement with this button. You can also think of it as Rewinding by one frame. You can rewind your last several movements by pressing this button multiple times. You can rewind even after you've finished or lost.

Snapshot collection The situations you've snapshot will appear here. You can click on them if you want to load them in the main screen, and continue working from that situation. This may be helpful during the second phase when you try to create situations for the AI to avoid.

Remove selected from Bad list If you Mark a configuration as Bad by accident, you can select it from the Snapshot area using the mouse, and remove it from the list using this button.

Verify Can you tell me how to [each of the four buttons]?

APPENDIX B

DESCRIPTIONS OF "BAD" RADIATION WORLD STATES

Participants' explanations of how they selected "bad" situations, or how they determined whether a situation was "bad" are given below. Typographical and orthographical errors have been preserved. The quotes have been annotated with features indicated by the participants' descriptions, in brackets; e.g. [`radiationStrength`]. These features are described in Table 7; the attribution of features to participants is summarized in Table 9.

User #1 "I selected each case where the robot died [`playState`]. I selected each of the darker-red squares [`radiationStrength`] since there is no reason the robot should try and go on those. I also marked the lighter red squares [`radiationStrength`] that weren't necessary for helping all of the zombies [`unavoidable`]."

User #2 The first three bad scenarios I marked were the three possible game-over scenarios [`playState`]. After those, I tried to mark the scenarios were the robot was very close to a game-over scenario (i.e. adjacent to the radiation [`radiationStrength`]) but hadn't saved many humans [`percentSaved`].

User #3 The first two ones: if the robots wants to get to the person from the starting point, it is unnecessary to be in the two grids[1]. A better way would be croing the upper-left grid of the radiation. 1: avoid higher risk [`radiationStrength`]. 2: do not go into the risky grids unless it is necessary [`unavoidable`].

User #4 The bad positions are that which put the robot in a dark red square that could lead the robot into the radiation and on the radiation squares themselves [`radiationStrength`].

[1]The participant is referring to two irradiated corner cells.

User #5 "1) There's no reason for the robot to be here at this time because the human on the other side of the radiation is already gone, and only makes it easier to get killed [`radiationStrength`]. 2)There's no reason to go through this way to get to the other side, concidering that the darker red squares cause the robot to not listen to you more often, which can make you die more easily than going through the ligher red squares twice [`radiationStrength`]. 3)There's no reason to purposefully go into this red square regardless of how many humans there are [`radiationStrength`]. 4-9)Again, there's no reason for the robot to purposefully more here regardless of how many humans. 10) There isnt any reason to go here because there aren't any humans within the wall, making you just waste extra moves [*state-action constraint*[2]]. 11) Too risky to cross from bottom to get to the top because the darker red squares are more likly to mess with your controls and get you killed, than going around and stepping on the lighter squares. Also the lighter squares are farther away from the radiation zone (2 moves away) compaired to the darker red zones (which are next to it). [`radiationStrength`] 12) Going here would be risky because you have to move into the darker red zone if you want to cross, which has a higher chance of getting you killed. Going around is much safer. If you didn't want to cross, its just a waste of a move and puts you into danger of falling into the radiation [`radiationStrength`]."

User #6 "Corner spots are dangerous because they are likely to randomely move you into the radiation symbol, while not being needed to complete the objective. Going through the far left to get to the human is bad because it forces the robot to go within one square of the radiation symbol, rather than going around. [`radiationStrength`]"

User #7 "because those 9 spots are right next to the radiation. [`radiationStrength`]"

User #8 "the immidiate neighbors of the radiation tiles are the most risky as they run the risk of death. The tiles surrounded by other red tiles, wvwn if they are not immidiately adjacent to the radiation tiles are also dangereous since the player needs to move through other red tiles to get out of there, increasing the risk of malfunction [`radiationStrength`]"

[2]In other words: do not move in a direction that cannot help you reach a human.

APPENDIX C

DESCRIPTIONS OF "BAD" PACMAN STATES

Participants' explanations of how they selected "bad" situations, or how they determined whether a situation was "bad" are given below. Typographical and orthographical errors have been preserved. The quotes have been annotated with features indicated by the participants' descriptions, in brackets; e.g., [`distanceToGhost`]. These features are described in Table 7; the attribution of features to participants is summarized in Table 9.

User #1 "Anytime I died [`playState`] I marked that as a bad state. Anytime I was within 1 or 2 squares of a ghost [`distanceToGhost`] I marked that as a bad state. Anytime I was within 3 squares of a ghost but stuck in a corner [`stuckInCorner`] I marked that as a bad state. Anytime I had just eaten a ghost, but it respawned within 2 squares [`distanceToGhost`], I marked that as a bad state."

User #2 "Most of the scenarios that are bad for pack man involve inescapable death. Examples of this include being trapped in the dead-end in the lower left by a ghost, getting caught in between two ghosts, or having all possible paths blocked by ghosts (this happens most often in loop on the lower right) [`surrounded`]. The scenario that I marked as bad even though there were no ghosts around I marked because all of the large pellets were gone [`energizersRemaining`] bot most of the small pellets were left [`dotsRemaining`]."

User #3 "I chose the bad configurations because it is possible that the robot has no way out [`surrounded`], meaning that if the monsters are intelligent(enough), it is going to be killed anyway."

User #4 "I could tell that the situations were bad either because pac man was cornered [`surrounded`] or had run into a ghost [`playState` or `distanceToGhost`]."

User #5 "1) Eatinf three of the circles that turn the ghosts white makes the game much hard to complete[1] concidering that eating the circles only refreshes the count [`dontWasteEnergizer`, *state-action constraint*, *non-Markov*], and doesn't add to the total number of spaces. 2)You are stuck, moving forward or back will get you eaten most likely [`surrounded`]. 3)Again you are stuck. 4) you are stuck again, and moving back will get you eaten most likely. 5)NO poinr in eating the circle to turn them white because they are far away and not a threat [`dontWasteEnergizer`, *state-action constraint*, *hierarchical constraint*]. 6) You are stuck again. 7)Too close to the ghosts and pretty hard to get to the circles that turn them white, which could turn out very bad [`distanceToGhost`, `hardToReachEnergizer`]. 8)Bad because you were surrounded by ghosts, which can geet you killed [`surrounded`]; a result of what might happen from the scenero above. 9)Surrounded [`surrounded`] and mostly likely will end up being eaten by the ghosts."

User #6 "Using the all the powerups shouild be avoided to prevent death from the ghosts [`energizersRemaining`][2]. Eating the ghosts near the center [`distanceToRespawnPoint`, *state-action constraint*, *hierarchical constraint*][3], after a powerup [*non-Markov*], should be avoided since they instantly respawn as a normal ghost. Situations where a ghost blocks the way to the last dot are bad as they allow additional chance to be killed by the other ghosts [`surrounded`]."

User #7 "the yellow one was cornered by the three monsters [`surrounded`], or just one step away [`distanceToGhost`]."

User #8 "getting cornered or studk in a tight corridor is dangereous as there are few ways of getting out of those situations which may be blocked by other ghosts [`surrounded`]. Also being in a situation where pacman is immidiately diagonal to a ghost is dangereous if he cant move away from the ghost, as there is a fifty percent chance of getting eaten

[1] The participant is referring to having just eaten the three energizers in series, which is wasteful. He is stating that because his recent action sequence was suboptimal, the resulting state is suboptimal.

[2] We note that it is not suboptimal to consume all the energizers; however, they should not be wasted.

[3] "Eating ghosts" (an option) should be avoided in states where PacMan is near the respawn point.

[diagonalFromGhost]”

REFERENCES

[1] P. Abbeel and A. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the 21st international conference on Machine learning (ICML 2004)*, 2004.

[2] Brenna D Argall, Brett Browning, and Manuela Veloso. Learning robot motion control with demonstration and advice-operators. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 399–404. IEEE, 2008.

[3] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

[4] Craig Boutilier, Richard Dearden, Moises Goldszmidt, et al. Exploiting structure in policy construction. In *IJCAI*, volume 14, pages 1104–1113, 1995.

[5] Luis C. Cobo, Zang Peng, Charles L. Isbell, and Andrea L. Thomaz. Automatic Abstraction from Demonstration. *International Joint Conference on Artificial Intelligence*, pages 1243–1248, 2011.

[6] John DeNero, Dan Klein, and Pieter Abbeel. The Pac-Man projects. URL `http://ai.berkeley.edu/project_overview.html`.

[7] François Denis, Rémi Gilleron, and Fabien Letouzey. Learning from positive and unlabeled examples. *Theor. Comput. Sci.*, 348(1):70–83, December 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2005.09.007. URL `http://dx.doi.org/10.1016/j.tcs.2005.09.007`.

[8] Marco Dorigo. *Robot shaping: an experiment in behavior engineering*. MIT press, 1998.

[9] Charles Elkan and Keith Noto. Learning classifiers from only positive and unlabeled data. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 213–220. ACM, 2008.

[10] Daniel H Grollman and Aude Billard. Donut as I do: Learning from failed demonstrations. In *International Conference on Robotics and Automation*, 2011.

[11] Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venkataraman. Efficient solution algorithms for factored mdps. *Journal of Artificial Intelligence Research*, pages 399–468, 2003.

[12] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009. ISSN 1931-0145. doi: 10.1145/1656274.1656278. URL `http://doi.acm.org/10.1145/1656274.1656278`.

[13] Walter L Hiursch and Cristina Videira Lopes. Separation of concerns. Technical report, Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, Massachusetts, 24 Feb, 1995.

[14] M. Kearns, Y. Mansour, and A.Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning*, 49(2):193–208, 2002.

[15] W. Bradley Knox and Peter Stone. Interactively shaping agents via human reinforcement: The tamer framework. In *The Fifth International Conference on Knowledge Capture*, September 2009. URL `http://www.cs.utexas.edu/users/ai-lab/?KCAP09-knox`.

[16] W Bradley Knox and Peter Stone. Reinforcement learning from simultaneous human and mdp reward. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 475–482. International Foundation for Autonomous Agents and Multiagent Systems, 2012.

[17] Daphne Koller and Ronald Parr. Computing factored value functions for policies in structured mdps. In *IJCAI*, volume 99, pages 1332–1339, 1999.

[18] George Konidaris and Andrew G. Barto. Efficient skill learning using abstraction selection. In *IJCAI*, pages 1107–1112, 2009.

[19] George Konidaris, Scott Kuindersma, Roderic Grupen, and Andre S Barreto. Constructing skill trees for reinforcement learning agents from demonstration trajectories. In *Advances in neural information processing systems*, pages 1162–1170, 2010.

[20] Wee Sun Lee and Bing Liu. Learning with positive and unlabeled examples using weighted logistic regression. In *ICML*, volume 3, pages 448–455, 2003.

[21] Hayim Makabee. Separation of concerns, 2 2012. URL `http://effectivesoftwaredesign.com/2012/02/05/separation-of-concerns/`.

[22] Shie Mannor, Ishai Menache, Amit Hoze, and Uri Klein. Dynamic abstraction in reinforcement learning via clustering. In *Proceedings of the twenty-first international conference on Machine learning*, page 71. ACM, 2004.

[23] Amy McGovern and Andrew G Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. *Computer Science Department Faculty Publication Series*, page 8, 2001.

[24] Nicolas Meuleau, Milos Hauskrecht, Kee-Eung Kim, Leonid Peshkin, Leslie Pack Kaelbling, Thomas L Dean, and Craig Boutilier. Solving very large weakly coupled Markov decision processes. In *AAAI/IAAI*, pages 165–172, 1998.

[25] Andrew Y Ng and Stuart J Russell. Algorithms for inverse reinforcement learning. In *ICML*, pages 663–670, 2000.

[26] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999.

[27] Doina Precup. *Temporal Abstraction in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, May 2000.

[28] Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[29] Jette Randløv and Preben Alstrøm. Learning to drive a bicycle using reinforcement learning and shaping. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 463–471, 1998.

[30] Kazuyuki Samejima, Kenji Doya, and Mitsuo Kawato. Inter-module credit assignment in modular reinforcement learning. pages 985–994, 2003.

[31] Özgür Şimşek and Andrew G Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 95. ACM, 2004.

[32] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *Abstraction, Reformulation, and Approximation*, pages 212–223. Springer, 2002.

[33] Kaushik Subramanian, Charles Isbell, and Andrea Thomaz. Learning options through human interaction. In *Workshop on Agents Learning Interactively from Human Teachers at IJCAI*, 2011.

[34] R.S. Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

[35] R.S. Sutton, D. Precup, S. Singh, et al. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999.

[36] A. L. Thomaz and C. Breazeal. Teachable robots: Understanding human teaching behavior to build more effective robot learners. *Artificial Intelligence Journal*, 172: 716–737, 2008.

[37] Andrea Lockerd Thomaz and Cynthia Breazeal. Reinforcement learning with human teachers: Evidence of feedback and guidance with implications for learning performance. In *AAAI*, volume 6, pages 1000–1005, 2006.

[38] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. The 2009 Mario AI competition. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010.

[39] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, 1989.

[40] Eric Wiewiora. Potential-based shaping and q-value initialization are equivalent. *Journal of Artificial Intelligence Research*, 19:205–208, 2003.

[41] Peng Zang. *Scaling solutions to Markov Decision Problems*. PhD thesis, Georgia Institute of Technology, 2011.

[42] Peng Zang, Peng Zhou, David Minnen, and Charles Isbell. Discovering options from example trajectories. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1217–1224. ACM, 2009.