2013

USE OF REINFORCEMENT LEARNING (RL) FOR PLAN GENERATION IN BELIEF-DESIRE-INTENTION (BDI) AGENT SYSTEMS

Jose L. Feliu
University of Rhode Island, jlfeliu@gmail.com

USE OF REINFORCEMENT LEARNING (RL) FOR PLAN GENERATION IN BELIEF-DESIRE-INTENTION (BDI) AGENT SYSTEMS

BY

JOSE L. FELIU

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2013

MASTER OF SCIENCE THESIS

OF

JOSE L. FELIU

APPROVED:

Thesis Committee:

Major Professor Lutz Hamel

Lisa DiPippo

Haibo He

Nasser H. Zawia
DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2013

ABSTRACT

The Belief-Desire-Intention (BDI) agent framework is a reactive agent framework based on the idea of intentionality. A known weaknesses of BDI is its lack of learning capabilities resulting from its dependence on an a-priori library of plans. BDI plans are designed by human experts on the domain to which BDI is being applied and are fixed. Any situation the BDI agent encounters which does not have a matching plan can result in erroneous agent operation and even agent failure.

Researchers have augmented the BDI framework with various learning frameworks including decision trees, self-organizing neural networks, hybrid-architectures using low level learners, and metaplans for plan hypothesis abduction and plan modifications. Other relevant research tackled the use of a-priori knowledge, previously learned knowledge and the learning of plans without a-priori knowledge on planning systems, and the integration of learning, planning and execution. These studies were, however, not investigated in relation to BDI systems.

This study explores the successful use of Reinforcement Learning (RL), a computational learning framework based on the idea of learning from repeated interactions with the environment, to generate plans in BDI systems without relying on a-priori knowledge.

ACKNOWLEDGMENTS

Back in the Fall of 2007 I started my "**official**" journey towards getting my master's degree. I had attempted it a few times but this time, this was the time. I was going to do it, no matter what. As I near completion, I look back with satisfaction, elated that I undertook the journey. I feel proud of the persistence and focus. I feel proud of all that I have learned and all of the personal and professional growth.

This thesis, as with any work of any significance, benefited from people that helped me with their advice, support and encouragement.

I would like to express my gratitude to the members of my Thesis Committee: **Dr. Lutz Hamel**, **Dr. Lisa DiPippo**, **Dr. Haibo He**, and **Dr. Bud Vincent**. Thank you Dr. Hamel for advising me and taking on a reinforcement learning thesis. Your advice about narrowing the scope of my research kept my overly enthusiastic expectations in check and allowed me to correct course when my initial coding attempts proved unsuccessful. Your courses on **Theory of Computation** and **Compiler Design** were always in the back of my mind as I tackled the thesis work. Some of the automatic agent generation code was inspired by what I learned during the compilers course. Thank you Dr. DiPippo for accepting to be a member of the committee even though I had only taken one course with you. Your emphasis on reading research papers as part of **Topics in Distributed Systems** prepared me to tackle the heavy job of doing the literature review and figure out what had been done before and what had not. Thank you Dr. He for agreeing to be on my thesis committee even though I had not taken any courses with you. I wish I had! I was inspired by the breadth of your research and the similarity of our interests.

Dr. Vincent, the **COEUT** program that you coordinated between **URI** and **NUWC** set me on my path to get my master's degree. I enjoyed taking your

Ocean System's Engineering course. I am very grateful that you stepped in last-minute to be my defense chair.

I would also like to express my sincere appreciation to **Dr. Peckham** for her support as I tackled some of the paperwork challenges with the Graduate School and to **Lorraine Berube** for providing tips to keep on track as I got close to the finish line.

Thanks to the **Graduate School**, **Carol Ward**, **Amy Tibbets**, and **Dean Nasser Zawia**.

Finally, I would like to thank somebody very special and close to my heart, **Dana A. Gionta**. Dana, you were there since the beginning. Since the first graduate course and through my multiple attempts at getting my degree. You were patient with the long journey, the busy schoolwork-filled weekends, and the late hours... You encouraged me, supported me, and cheered me. But most important of all, you believed in me. **Always!** For that and many more reasons I will forever be grateful and you will always have a special place in my heart!

DEDICATION

In loving memory of my late mother **Graciela M. Camacho Matos**. Wherever you are, I know you are smiling with pride.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

Introduction

1.1 Motivation

The lack of learning capabilities for BDI systems was recognized as far back as 2004 [1]. Researchers tackled this by augmenting the BDI framework with various learning frameworks including decision trees, self-organizing neural networks, hybrid-architectures using low level learners, and metaplans for plan hypothesis abduction and plan modifications. Other relevant research tackled the use of a-priori knowledge, previously learned knowledge and the learning of plans without a-priori knowledge on planning systems, and the integration of learning, planning and execution. These studies were, however, not investigated in relation to BDI systems.

Recent research relied on Markov Decision Processes (MDPs) to generate BDI plans from optimal policies for completely specified MDPs [2]. Pereira's work was augmented to work with Partially Observable Markov Decision Processes (POMDPs) [3]. These two studies come closest to the proposed study with the difference that for the proposed study neither fully specified MDPs nor POMDPs will be considered.

The problem selected for study is justified by the lack of research exploring the generation of plans in BDI systems using reinforcement learning that does not rely on a-priori knowledge.

1.2 Thesis Organization

Chapter 2 Planning and Learning discusses the close relationship between **planning and learning**. In this work, RL represents the learning aspect and BDI represents the planning aspect. **Chapter 3 Reinforcement Learn-**

ing introduces the computational RL framework and provides an introduction to the rigorous mathematical notions that underlie learning from repeated interactions with the environment. **Chapter 4 Belief-Desire-Intention (BDI) Agent Systems Framework** introduces the BDI framework and highlights its known weakness of relying on an **a-priori plan library**. The logic BDI programming language **AgentSpeak** is also examined as well. **Chapter 5 Differences Between Proposed Study and Previous Research** provides the previous research that justifies this thesis. The coding implementation for the RL and BDI parts is discussed in **Chapter 6 Experimental Implementation**. The results are discussed in **Chapter 7 Results**. Finally, a discussion of the results, limitations, and ways to enhance the ideas in this thesis in future work are part of **Chapter 8 Discussion**.

List of References

[1] A. Guerra-Hernández, A. E. Fallah-Seghrouchni, and H. Soldano, "Learning in bdi multi-agent systems," in *CLIMA*, 2004, pp. 218–233.

[2] D. R. Pereira and G. P. Dimuro, "Um algoritmo para extração de um plano bdi que obedece uma política Ótima," in *Workshop-Escola de Sistemas de Agentes para Ambientes Colaborativos, Pelotas, Anais do WESAAC 2007*, 2007.

[3] D. R. Pereira, L. Vargas, G. P. Dimuro, and A. C. R. Costa, "Constructing bdi plans from optimal pomdp policies, with an application to agentspeak programming," in *XXXIV - Conferencia Latinoamericana de Informtica, 2008, Santa Fe. Proceedings of CLEI 2008 (Lecture Notes in Computer Science)*, ser. CLEI 2008, vol. 1. Springer, 2008, pp. 1–10.

CHAPTER 2

Planning and Learning

2.1 Introduction

Planning and learning are closely connected. It is difficult to think of one without thinking of the other. Zimmerman considers them to be the "most broadly recognized hallmarks of intelligence" and defines them as:

planning - solving problems in which one uses beliefs about actions and their consequences to construct a sequence of actions that achieve one's goals.

learning - using past experience and precepts to improve one's ability to act in the future. [1]

2.2 Zimmerman's Model

Zimmerman introduces a 5 dimensional model to characterize automated planning systems that are augmented with a learning component, including an extensive survey of planning and learning [1]. The model allows us to compare, within the model limitations, the different approaches researchers have pursued to combine planning and learning in a graphical way.

The 5 dimensions of the model are:

1. **Problem Type**. The **problem type** is a function of the environment. Static, propositional, deterministic, fully observable environments, in which effects of actions are instantaneous fall under **classical planning**. Dynamic, continuous, partially observable environments, in which actions take time to be executed fall under **full-scope planning**.

2. **Planning Approach**. The **planning approach** describes how planning is achieved. There are two planning approaches: **search** and **model checking**. Search planners can search in **state space** or **plan space**. Model

checking planners transform the planning problem representation into one for which efficient algorithms exists, including satisfiability (SAT), constraint-satisfaction problems (CSPs), and integer linear programming (IP).

3. **Planning-learning Goal**. The **planning-learning goal** describes the aim of learning. There are 3 aims for learning:(a) **Planning speedup**, (b) **Improving planning domain theory** which is concerned with using failures related to deficiencies of the world model and domain theory through plan failures as feedback opportunities to improve the world model and domain theory for the planner, and (c) **Improving the quality of plans produced**.

4. **Learning Phase**. The **learning phase** describes when in the planning process the learning occurs. Learning can occur in 3 phases: (a) before planning, (b) during the process of finding valid plan, and (c) during the execution of plan

5. **Type of Learning**. The **type of learning** describes the learning used by the planner. There are 3 types of learning: (a) **inductive learning**, (b) **analytic learning**, and **(c) multistrategy learning** is the category for approaches that do not clearly fit in the two other types of learning. RL falls under multistrategy learning.

Besides providing a useful model to characterize planning and learning research, the survey discovered few research attempts that used RL for planning. These included a hybrid approach combining explanation based learning (**EBL**) with RL into an explanation base reinforcement learning (**EBRL**) algorithm and tested on chess endgames and synthetic maze tasks [2]. Incremental dynamic programming (**DYNA**) was proposed as an RL architecture that split learning and planning by having actions generated by a reactive system with a planning system acting ”independently and conceptually in parallel” [3]. Learning by Observation in Planning Environments (**LOPE**) was proposed as an architecture integrating learning, planning and execution [4]. LOPE’s learning is focused on learning operator definitions, plans using the operators, and executes plans that modify acquired operators.

List of References

[1] T. Zimmerman and S. Kambhampati, “Learning-assisted automated planning: Looking back, taking stock, going forward,” *AI Mag.*, vol. 24, no. 2, pp. 73–96, June 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=960150.960160

[2] T. G. Dietterich and N. S. Flann, “Explanation-based learning and reinforcement learning: A unified view,” in *Machine Learning*, 1995, pp. 176–184.

[3] R. S. Sutton, “Planning by incremental dynamic programming,” in *In Proceedings of the Eighth International Workshop on Machine Learning*. Morgan Kaufmann, 1991, pp. 353–357.

[4] R. Garcia-Martinez and D. Borrajo, “An integrated approach of learning, planning, and execution,” *Journal of Intelligent and Robotic Systems*, vol. 29, pp. 47–78, 2000.

CHAPTER 3

Reinforcement Learning

3.1 Introduction

Reinforcement learning (RL) is a computational learning framework based on the idea of learning from repeated interactions with the environment [1, 2, 3]. RL agents seek to maximize a reward signal from the environment. As the agent explores the environment it learns the actions that maximize the reward from particular states. The agent does this by selecting the action that will bring the greatest reward, known as the **greedy** action. Once an agent has completed a start-state to goal-state cycle, the agent has two choices: **exploitation** or **exploration**. An agent can **exploit** the knowledge it learned interacting with the environment by choosing the greedy action or it can **explore** the environment by choosing an untried action or trying again a sub-optimal action. This is known as the **exploration-exploitation** dilemma.

Choosing the greedy action all the time, however, is not an effective strategy. Immediate high rewards can be followed by low rewards that outweigh the initial high rewards. Immediate low rewards can, conversely, be followed by high rewards that outweigh the initial low rewards. To overcome myopic behavior, RL agents need to balance exploitation with exploration.

3.2 RL Problem

To apply reinforcement learning to a problem requires that the problem be characterized as a **RL problem**. RL problems can be characterized by four elements:

1. a policy

2. a reward function

3. a value function

4. a model

The **policy** maps actions to states. Given a state it will provide the action that the agent needs to perform if it wants to maximize the reward. The policy is created by the interaction of the agent and the environment. The policy answers the agent's question **What do I do in this state?** It is the map for getting what is good and avoiding what is bad. Policies are stochastic in general.

The **reward function** provides feedback from the environment in response to actions taken by an agent in particular states. Because immediate rewards only provide partial answers regarding what is the best course of action, the reward function has little to say about the value of particular states in the long run. Low rewards can follow high rewards and viceversa. Only by keeping track of this fact will the agent be able to prevent myopic behavior. The reward function answers the agent's question **What do I get if I do X in this state?** It is what is good in the short run. Reward functions are stochastic in general.

The **value function** is the expected rewards averaged over many exploration-exploitation trials. It provides an approximation of the value of particular states. It answers the agent's question: **What can I expect if I start in this state and follow what I have learned?** It is "what is good in the long run" [2].

The **model** provides a model of the environment and of how it reacts to specific actions by the agent. It is used for planning by simulating actions in particular states and observing the rewards, in effect experiencing the environment through a simulation. In many domains, however, a model is not available or unfeasible to obtain. Fortunately, RL can learn the world model empirically by interacting with the environment.

Figure 1 presents a notional diagram summarizing the process through which an RL agent learns a policy through its interaction with the environment.

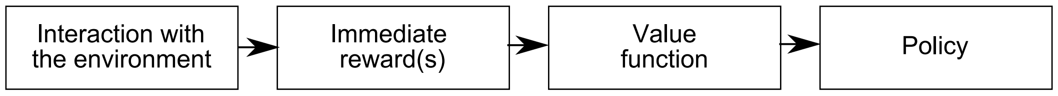

Figure 1. RL Notional Process

3.3 RL Problem Formalization

This section introduces a formalization of the RL problem.

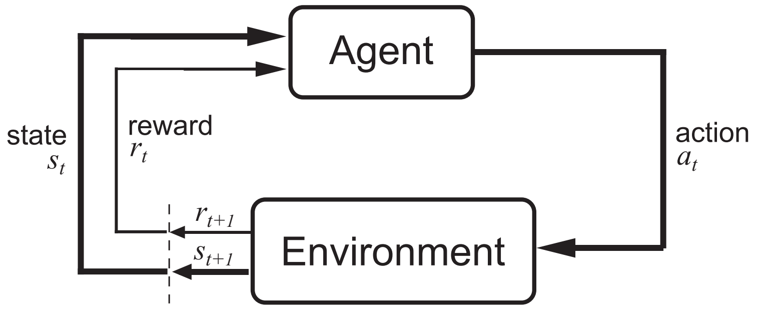

Figure 2. Agent-Environment Model [2]

In RL problems we will always have an agent situated in an environment. “An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon the environment through **actuators**.” [4]

agent: the RL agent, is the part of the system that is learning, the learner

action: $a_t \in A\left(s_t\right)$, the action chosen by the agent at a particular **state** and a particular time

environment: everything outside the agent or what the agent interacts with

state: $s_t \in S$, where the agent finds itself after choosing an **action** and receiving the corresponding **reward**

task: complete specification of the environment

reward: numerical value that the agent tries to maximize received after choosing an **action**

policy: $\pi_t(s, a)$, mapping from states to probabilities of selecting each possible **action**

The agent in state s_t selects action $a_t \in A(s_t)$, which results in reward r_{t+1} and state $s_{t+1} \in S$, leading to the following sequence:

$$s_0 a_0 r_1 \rightarrow s_1 a_1 r_2 \rightarrow s_2 a_2 r_3 \rightarrow \cdots \rightarrow s_t a_t r_{t+1} \rightarrow s_{t+1} a_{t+1} r_{t+2} \rightarrow s_{t+2} a_{t+2} r_{t+3} \rightarrow \cdots$$

As the agent moves from state to state it collects rewards. The sum of all rewards defines the **returns**.

3.4 Returns

RL agents seek to maximize the rewards received, the **returns**. Returns are defined differently depending on whether the task is **episodic** or **continuing**. Episodic tasks can be broken in discrete and finite episodes. In other words, they do not go on forever. In this case the return would be the sum of rewards from the beginning of the episode, at time $t + 1$ until time T that the episode finishes. In this straightforward case, the return R_t can be defined as:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T$$

Continuing tasks are not so easily broken into distinct episodes and can potentially go on for a long time, even "forever". In this case, rewards received immediately are more valuable than rewards received later. Multiplying each reward by a decreasing factor provides a way to specify the value of immediate rewards. This factor is called the **discounting factor** γ, for $0 < \gamma < 1$. The value for the discounting factor provides a way to specify how short-term or forward-looking we want the agent to be. The **discounted return** R_t is defined as:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

3.5 Markov Property

The **Markov property** is important in RL because it **defines systems that do not need to keep history**. In other words, what matters is the **current state of the system and not the history of how it got there**. For some systems there might be infinite ways of getting to a particular state. In problems where keeping history is necessary to predict future actions, the state space would be orders of magnitude bigger. In such cases the problem would be consequently much more complex and difficult to analyze, model and simulate.

Formally, the Markov property is exhibited by systems in which the following two equations are equivalent:

$$Pr\left\{s_{t+1} = s', r_{t+1} \mid s_t, a_t, r_t s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\right\} \tag{1}$$

$$Pr\left\{s_{t+1} = s', r_{t+1} \mid s_t, a_t\right\} \tag{2}$$

for all s', r_{t+1}, and histories, $s_t, a_t, r_t s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0$. [2] Equation (1) shows the probability of state s_{t+1} being s' and reward being r_{t+1} given the previous sequence of states, actions and rewards $s_t, a_t, r_t s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0$. Equation (2) shows probability of state s_{t+1} being s' and reward being r_{t+1} given only the previous state and previous action s_t, a_t. If these 2 equations are equivalent, it means that state, action and reward history is not a factor in determining the probability of the next state and its reward. This leads to **one-step dynamics** that allows to predict the next state and the expected next reward based on the current state and the current action.

A RL task that satisfies the Markov property is **Markov Decision Process** (**MDP**). For finite MDPs, the probability of each possible state, s', is:

$$P^a_{ss'} = Pr\left\{s_{t+1} = s' \mid s_t = s, a_t = a\right\}$$

The expected value of the next reward is:

$$R^a_{ss'} = E\left\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right\}$$

For the purposes of this research, a finite MDP is assumed.

3.6 Value Functions

The **value function** is the expected rewards averaged over many exploration-exploitation trials. Value functions have two equivalent representations: **value functions of states**, typically denoted V, and **value functions of state-action pairs**, typically denoted Q. These functions are evaluated for a given policy π.

State-value function for policy π

$$V^{\pi}(s) = E_{\pi}\left\{R_T \mid s_t = s\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\}$$

V^{π} is called the **state-value function for policy π** , it represents "the expected return when starting in s and following π thereafter".

Action-value function for policy π

$$Q^{\pi}(s,a) = E_{\pi}\left\{R_T \mid s_t = s, a_t = a\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\}$$

Q^{π} is called This equation represents the **action-value function for policy π** , it represents "the expected return starting from s, taking action a, and thereafter following policy π".

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}\left\{R_T \mid s_t = s\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\} \\ &= E_{\pi}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s\right\} \\ &= \sum_a \pi(s,a) \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma E_{\pi}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_{t+1} = s'\right\}\right] \\ &= \sum_a \pi(s,a) \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V(s')\right] \end{aligned}$$

One useful way of visualizing these equations is through the use of **backup diagrams**. In backup diagrams, states are denoted by white circles and actions are denoted by smaller black circles. Figure 3(a) shows the diagram for V^{π}. It is intuitive that the value for state s will depend on all possible actions a available at state s. These actions provide a reward r and take the agent to state s'. Since policies are stochastic and agents have to balance exploration-exploitation, the value for state s will be an approximation. The more the agent explores the environment, the closer the value for state s will come to its true value. Figure 3(b) shows the diagram for Q^{π}. In this case, similar to V^{π}, the value for action-value s, a will depend on the reward r and the value for s' and the next action a' constituting the next action-value s', a'.

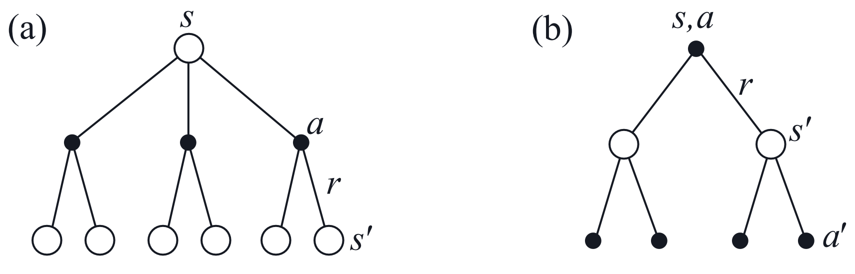

Figure 3. Backup Diagrams for (a) V^{π} and (b) Q^{π} [2]

3.7 Optimal Functions

Value functions are evaluated for particular policies. It should not be surprising that some policies perform better than others. Given two policies π and π', π is a better policy than π', $\pi \geq \pi'$, if and only if $V^{\pi}(s) \geq V^{\pi'}(s)$ for all $s \in S$. A policy that performs equal or better than all other policies is an **optimal policy**, denoted by π^*.

Using the optimal policy, an **optimal state-value function** can be defined as:

$$V^*(s) = \underset{\pi}{max}\, V^{\pi}(s)$$

for all $s \in S$.

The **optimal action-value function** can be defined as:

$$Q^*(s,a) = \underset{\pi}{max}\, Q^{\pi}(s,a)$$

for all $s \in S$, $a \in A(s)$.

The **optimal state-value function** equation above can be rewritten without referencing a policy:

$$\begin{aligned} V^*(s) &= \underset{a \in A(s)}{max}\, Q^{\pi}(s,a) \\ &= \underset{a}{max}\, E_{\pi^*}\left\{R_t \mid s_t = s, a_t = a\right\} \\ &= \underset{a}{max}\, E_{\pi^*}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s, a_t = a\right\} \\ &= \underset{a}{max}\, E\left\{r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\right\} \\ &= \underset{a}{max} \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V(s')\right] \end{aligned}$$

The last equation is the **Bellman equation** for V^*, or the **Bellman optimality equation**.

Similarly, the **optimal state-value function** equation can as the Bellman optimality equation for Q^*:

$$Q^*(s,a) = E\left\{r_{t+1} + \gamma \max_{a} Q^*(s_{t+1,}, a') \mid s_t = s, a_t = a\right\}$$
$$= \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma \max_{a'} Q^*(s', a')\right]$$

The backup diagrams forV^* and Q^* are shown in figure 4.

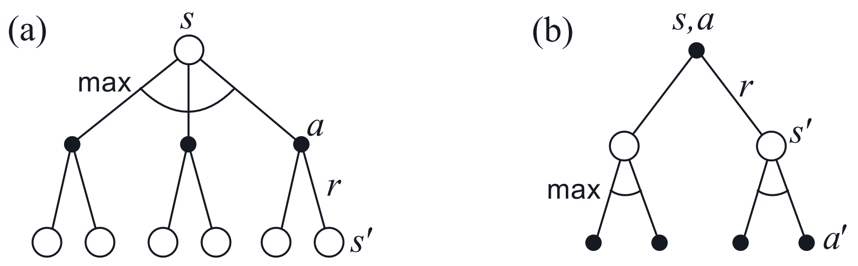

Figure 4. Backup Diagrams for (a) V^* and (b) Q^* [2]

3.8 Learning Tasks

Before RL and its algorithms can be used on a particular problem, the problem needs to be expressed as a **learning task**.

A **learning task** is a complete specification of the agent and the environment and its is defined by 3 sets.

1. The state set S.

2. The action set A.

3. The reward set R.

The environment is generally concerned with the state and the states $s_t \in S$. The state only changes as a result of actions selected by the agent.

In any learning task the goal is for the agent to learn what to do in any particular situation that it encounters, to learn a mapping from states to actions, a **policy**. The collection of all the situations that the agent may encounter define the **state-space** of the task. For example, an agent in a 2-dimensional **gridworld** has at least two pieces of information that it would have to track if we are to hope for any helpful and significant learning: where it is on the X coordinate and where it is on the Y coordinate. Furthermore, if the X coordinate has a maximum dimension of 10 and the Y coordinate has a maximum dimension of 10, then we know that there are 100 possible combinations of $< x, y >$ values. This is the state-space for the problem. The X and Y coordinate are **state-variables**. They define a state because we find it useful to distinguish $< x_t, y_t >$ from $< x_{t+1}, y_{t+1} >$ and they are variables because they change dynamically depending on the action chosen by the agent.

The agent is generally concerned with the action set and the actions $a_t \in A(s_t)$. When the agent chooses an **action** $a_t \in A(s_t)$ at a particular **state** $s_t \in S$ and a particular **time** t, the result will be a **reward** $r_{t+1} \in R$ and **new state** $s_{t+1} \in S$.

The reward is provided by the environment as a result of the agent selecting an action. Since many implementations of RL have synthetic, simulated environments, it is contingent upon the RL researcher to define a reward set in a manner that is conducive to learning. Usually this involves some amount of trial and error in selecting the reward structure in a way that will bias the learning task to learn the actual behaviors we want learned.

Once a problem is specified as a learning task, it is ready to learn but, how does the learning happen? The cross product $S \times A$ defines the **action values** or Q **values** for the learning task. These are state-action tuples. Action values are initialized arbitrarily since no learning has taken place and the true value is not yet known. As the agent interacts with the environment the RL algorithm updates the value of the corresponding state-action tuple by backing up a fraction of the value of latter states to preceding states to better reflect their true value. Selecting the state-action tuples with maximum value defines a **policy**.

3.9 RL Algorithms

A number of algorithms have been developed to implement RL. An important development in RL was the development of **Q learning** by Watkins in 1989 [2], [5]. In its simplest form the update rule for Q learning, also known as **one-step Q learning**, has the form:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_a Q(s',a') - Q(s,a)\right]$$

In 1992 Watkins and Dayan proved that **Q learning** "converges to the optimum action-values with probability 1 so long as all actions are repeatedly sampled in all states and the action-values are represented discretely" [6]. In other words "the learned action-value function, Q^π, directly approximates Q^*" [2].

Some of the benefits of Q learning include:

1. It is **model-free**. It does not need the probability distributions for transitions from state s to state s'.
2. It can handle stochastic transitions.

The algorithmic representation for Q learning is taken from [2] and is shown in table 1.

Q Learning Algorithm

```
Initialize $Q(s,a)$arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each episode):
        Choose $a$ from $s$ using policy derived from $Q$ ($\epsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma \max_a Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$
    until $s$ is terminal
```

Table 1. Q Learning Algorithm

3.10 RL and Planning

A detailed survey of RL and automated planning system is presented by Partalas [7]. Partalas argues that "there is a close relationship between those two areas as they both deal with the process of guiding an agent, situated in a dynamic environment, in order to achieve a set of predefined goals."

Because RL combines planning and learning the distinctions above blur in practice if not in theory. Since a relevant part of the research for this study will be the selection of feasible RL approaches to generate plans for BDI systems, only a quick summary of the research detailed by Partalas will be given here except in cases where research is directly related to the problem for this study. Partalas describes the possible approaches to combining planning with RL as:

1. **Planning for RL**. The approaches to planning for RL are categorized in 3 groups:

    (a) Based on **Dyna [8]** - Dyna is an architecture that extends reinforcement learning by including a **world model**. Sutton summarized some of the limitations of RL stating that "Reinforcement learning architectures are effective at trial-and-error learning, but no more. They can not do any of the things that are considered "cognitive," such as reasoning or planning. They do not learn an internal model of the world's dynamics, of what-causes-what, but only of what-to-do (policy) and how-good-is-it (return predictions). This is an important limitation because potentially much more can be learned in the form of a world model than can be learned by trial and error; the reward signal is just a scalar, while the sensory input signal is a much richer potential source of training information. And what if the goal changes? Typically, a world model can remain relatively intact over goal changes and can assist in achieving the new goal, whereas policy and return predictions must be totally changed." [9] Approaches based on Dyna include:

i. **Dyna-Q** Combines the Dyna architecture with Q-learning. It learns a world model to generate hypothetical experience and achieve planning. [10]

ii. **Queue-Dyna** Value function estimates are prioritized. The ones with the highest priority are put on a queue and performed. Places where the value function needs to be update are identified as **"update candidates**" . Two methods are proposed: **prediction difference** in which the priority depends on the magnitude of the predicted difference in value and **effect on start-state value** in which the priority depends on the contribution of update candidates to the the value of a fixed start-state. [11]

iii. **AHC-Relaxation Planning** Combines an **adaptive heuristic critic (AHC)** architecture with **relaxation planning**. AHC is an architecture to take into account the effect of delayed rewards. [12] "Relaxation planning, which is closely related to dynamic programming, is an incremental planning process that consists of a series of shallow (usually one-step look-ahead) searches and ultimately produces the same results as a conventional deep search." [13]

    iv. **Q-Relaxation Planning** Is "similar to Sutton's Dyna-Q architecture except that only the currently visited state is used to start hypothetical experiences." [13]

    v. **Exploration Planning [14]**

   (b) Based on **Prioritized Sweeping** [15] - The idea is to work backwards from states that have big changes in their value estimation. As state-action pairs are estimated, predecessing states are put on priority queue according to the size of their potential back-up value. This is repeated a number of times or until the queue is empty.

    i. **Generalized Prioritized Sweeping [16]** - Introduces the **Generalized Prioritized Sweeping Principle** (GenPS) which states "Update states where the approximation of the value function will change the most. That is, update the states with the largest **Bellman error**, $E(s) = \left|\hat{V}(s) - max_{a \in A}\, \hat{Q}(s,a)\right|$

    ii. **Structured Prioritized Sweeping** [17]

   (c) Based on **Other** - Other interesting approaches include **PLANQ-learning** [18], **Reinforcement Learnt-TOPs** [19], **Teleo-Reactive Q-Learning** [20, 21]. These, however, differ in the planning component which is different from the BDI model.

2. **RL for Planning**. The approaches to RL for planning are:

   (a) **Forward and bidirectional planning based on RL** [22] - Presents a

Dyna architecture implementing AHC using neural networks. A forward and backward planner are used. The backward planner functions in a similar fashion to prioritized sweeping, the main difference being the knowledge of specific goal states from which to plan backwards.

(b) **Extraction of Planning Knowledge from Reinforcement Learners** - A process for extracting plans is presented after using an RL algorithm to learn a policy [23, 24]. The process presented is "concerned with the ability to plan in an uncertain environment where usually knowledge about the domain is required. Sometimes is difficult to acquire this knowledge, it may impractical or costly and thus an alternative way is necessary to overcome this problem. [7]" Plans are extracted by successively calculating the probabilities of each action reaching the goal state from an initial state. The plan is selected greedily from the probabilities calculated.

(c) **RL Approach to Production Planning** [25] - RL through the use of Q-learning and Monte Carlo simulation are used to "solve a multi-period production planning problem in a two stage hybrid manufacturing process (a combination of build-to-plan with build-to-order) with a capacity constraint."

All references are as cited on [7].

Partalas ends the survey summarizing the approaches for combining learning and planning as "**first learn then plan**", "**first plan then learn**" or "**interchange learning and planning**".

List of References

[1] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[2] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[3] A. Gosavi, “Reinforcement learning: A tutorial survey and recent advances,” *INFORMS Journal on Computing*, vol. 21, no. 2, pp. 178–192, 2009.

[4] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall, 2003.

[5] C. J. C. H. Watkins, “Learning from delayed rewards,” Ph.D. dissertation, King’s College, Cambridge, UK, May 1989. [Online]. Available: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf

[6] C. J. C. H. Watkins and P. Dayan, “Technical note q-learning,” *Machine Learning*, vol. 8, pp. 279–292, 1992.

[7] I. Partalas, D. Vrakas, and I. Vlahavas, “Reinforcement learning and automated planning: A survey,” in *Advanced Problem Solving Techniques*, D. Vrakas and I. Vlahavas, Eds. IGI Global, 2008.

[8] R. S. Sutton, “Planning by incremental dynamic programming,” in *In Proceedings of the Eighth International Workshop on Machine Learning.* Morgan Kaufmann, 1991, pp. 353–357.

[9] R. S. Sutton, “Dyna, an integrated architecture for learning, planning, and reacting,” *SIGART Bulletin*, vol. 2, no. 4, pp. 160–163, 1991.

[10] R. S. Sutton, “Integrated architectures for learning, planning, and reacting based on approximating dynamic programming,” in *In Proceedings of the Seventh International Conference on Machine Learning.* Morgan Kaufmann, 1990, pp. 216–224.

[11] J. Peng and R. J. Williams, “Efficient learning and planning within the dyna framework,” in *Adaptive Behavior*, 1993, pp. 437–454.

[12] R. S. Sutton, “Reinforcement learning architectures,” in *Proceedings ISKIT’92 International Symposium on Neural Information Processing*, 1992.

[13] L. ji Lin, “Self-improving reactive agents based on reinforcement learning, planning and teaching,” in *Machine Learning*, 1992, pp. 293–321.

[14] G. Zhao, S. Tatsumi, and R. Sun, “Rtp-q: A reinforcement learning system with time constraints exploration planning for accelerating the learning rate,” *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 82, no. 10, pp. 2266–2273, 1999. [Online]. Available: http://ci.nii.ac.jp/naid/110003208162/en/

[15] A. W. Moore and C. G. Atkeson, “Prioritized sweeping: Reinforcement learning with less data and less time,” in *Machine Learning*, 1993, pp. 103–130.

[16] D. Andre, N. Friedman, and R. Parr, “Generalized prioritized sweeping,” in *Advances in Neural Information Processing Systems.* MIT Press, 1998.

[17] R. Dearden, “Structured prioritised sweeping,” in *ICML*, 2001, pp. 82–89.

[18] M. Grounds and D. Kudenko, “Combining reinforcement learning with symbolic planning,” in *Proceedings of the 5th , 6th and 7th European conference on Adaptive and learning agents and multi-agent systems: adaptation and multi-agent learning*, ser. ALAMAS’05/ALAMAS’06/ALAMAS’07. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 75–86. [Online]. Available: http://dl.acm.org/citation.cfm?id=1898681.1898687

[19] N. J. Nilsson, “Teleo-reactive programs for agent control,” *J. Artif. Intell. Res. (JAIR)*, vol. 1, pp. 139–158, 1994.

[20] M. R. Ryan and M. D. Pendrith, “Rl-tops: An architecture for modularity and re-use in reinforcement learning,” in *In Proceedings of the Fifteenth International Conference on Machine Learning.* Morgan Kaufmann, 1998, pp. 481–487.

[21] M. R. K. Ryan, “Using abstract models of behaviours to automatically generate reinforcement learning hierarchies,” in *In Proceedings of The 19th International Conference on Machine Learning.* Morgan Kaufmann, 2002, pp. 522–529.

[22] G. Baldassarre, “Forward and bidirectional planning based on reinforcement learning and neural networks in a simulated robot,” in *ABiALS*, 2003, pp. 179–200.

[23] R. Sun and C. Sessions, “Learning plans without a priori knowledge,” in *in Proceedings of WCCI-IJCNN’98.* IEEE Press, 2000, pp. 1–6.

[24] R. Sun, T. Peterson, and C. Sessions, “Beyond simple rule extraction: Acquiring planning knowledge from neural networks.” Berlin Heidelberg: Springer-Verlag, 2001.

[25] H. Cao, H. Xi, and S. F. Smith, “Supply chain planning: A reinforcement learning approach to production planning in the fabrication/fulfillment manufacturing process,” in *Winter Simulation Conference*, 2003, pp. 1417–1423.

CHAPTER 4

Belief-Desire-Intention (BDI) Agent Systems Framework

4.1 Introduction

BDI is a reactive agent framework based on the idea of intentionality [1]. BDI agents have beliefs, desires and intentions [1],[2]. **Beliefs** are representations of what the agent believes true in its world. **Desires** represent the state of the world that the agent would like to achieve. **Intentions** represent what the agent intends to do.

BDI implementations have a library of plans that are triggered depending on the belief and desires of the agents. The plans are designed by human experts on the domain to which BDI is being applied.

One of the weaknesses of BDI is its lack of learning capabilities. Researchers have addressed this weakness by augmenting the BDI framework with various learning frameworks including decision trees [3], self-organizing neural networks [4],[5], hybrid-architectures using low level learners, and metaplans for plan hypothesis abduction and plan modifications [4].

4.2 AgentSpeak

AgentSpeak is a programming language for BDI agents, based on logic programming, proposed by Rao [6]. It was inspired by the Procedural Reasoning System (**PRS**), an early BDI architecture with a plan library and explicit symbolic representations of beliefs, desires, and intentions, [7], the distributed Multi-Agent Reasoning System (**dMARS**) [8], and BDI Logics [2].

The specific version of AgentSpeak used in this reseach is the one augmented by Jomi F. Hübner and Rafael H. Bordini for use in Jason [9], [10], [11], [12].

4.3 AgentSpeak Components

The architecture of an AgentSpeak agent has four main components as shown in table 2.

AgentSpeak Components
Belief Base
Plan Library
Set of Events
Set of Intentions

Table 2. AgentSpeak Components

A known weaknesses of BDI is its lack of learning capabilities resulting from its dependence on an a-priori **plan library**. BDI plans are designed by human experts on the domain to which BDI is being applied and are fixed. Any situation the BDI agent encounters which does not have a matching plan can result in erroneous agent operation and even agent failure.

This study is motivated at precisely this weakness of a **fixed a-priori plan library**. The idea is to find ways of generating plans without relying on a-priori knowledge.

4.4 AgentSpeak Constructs

The main language constructs of AgentSpeak are shown in table 3.

AgentSpeak Constructs
Beliefs
Goals
Plans

Table 3. AgentSpeak Constructs

4.5 AgentSpeak Syntax

AgentSpeak is based on logical programming. Its syntax for belief, desires (goals), and intentions (plan) reflect the underlying logical programming paradigm.

Beliefs represent the information available to an agent about the environment or other agents. Beliefs are represented in symbolic form by predicates.

The representation of the belief that **wiley** is the **publisher**, for example, is represented as:

```
publisher(wiley)
```

Goals represent states of of fairs the agent wants to bring about (come to believe, when goals are used declaratively). There are two types of goals: **achievement goals** and **test goals** as show in table 4.

AgentSpeak Types of Goals
Achievement
Test

Table 4. AgentSpeak Types of Goals

Achievement goals are used when attempting to change the belief base. An example of an achievement goal is shown below.

```
!write(book)
```

The `!` makes the goal an **achievement goal**. In this case, the agent's goal is to write a book.

Test goals are used when attempting to retrieve information from the belief base. An example of a test goal is show below.

```
?publisher(P)
```

The `?` makes the goal an **test goal**. In this case, the agent wants to find the publisher and bound it to variable `P`.

A BDI agent reacts to events in the environment by executing plans. Events happen as a consequence to changes in the agents beliefs or goals. Plans are "recipes for action, representing the agents know-how".

An AgentSpeak plan has the following general structure:

```
triggering_event : context <- body.
```

The **triggering event** denotes the events that the plan is meant to handle. The **context** represent the circumstances in which the plan can be used. The **body** is the course of action to be used to handle the event if the context is believed true at the time a plan is being chosen to handle the event.

Consider the AgentSpeak's plan fragment shown below.

```
+state(0,0,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.
```

For this plan, if the agent finds itself in `state(0,0,0,0)` and the context `calculation(done)` is true, the the agent will drop the belief `calculation(done)`, it will perform `action(n)` and add the achievement goal `!calculateReturn`.

4.6 Jason

Jason is an Java-based interpreter for an extended version of AgentSpeak. It implements the operational semantics of AgentSpeak, and provides a platform for the development of multi-agent systems. Jason was developed and is currently maintained by Jomi F. Hübner and Rafael H. Bordini [9], [10], [11], [12].

List of References

[1] A. S. Rao and M. P. Georgeff, “Modeling rational agents within a bdi-architecture,” in *KR*, 1991, pp. 473–484.

[2] A. S. Rao and M. P. Georgeff, “Bdi agents: From theory to practice,” in *ICMAS*, 1995, pp. 312–319.

[3] D. Singh, S. Sardiña, L. Padgham, and S. Airiau, “Learning context conditions for bdi plan selection,” in *AAMAS*, 2010, pp. 325–332.

[4] S. Karim, B. Subagdja, and L. Sonenberg, “Plans as products of learning,” in *IAT*, 2006, pp. 139–145.

[5] A.-H. Tan, “Falcon: A fusion architecture for learning, cognition, and navigation,” in *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, vol. 4, july 2004, pp. 3297 –3302 vol.4.

[6] A. S. Rao, “Agentspeak(l): Bdi agents speak out in a logical computable language,” in *MAAMAW*, 1996, pp. 42–55.

[7] M. P. Georgeff and A. L. Lansky, “Reactive reasoning and planning,” in *AAAI*, 1987, pp. 677–682.

[8] M. d’Inverno, D. Kinny, M. Luck, and M. Wooldridge, “A formal specification of dmars,” in *Proceedings of the 4th International Workshop on Intelligent Agents IV, Agent Theories, Architectures, and Languages*, ser. ATAL ’97. London, UK, UK: Springer-Verlag, 1998, pp. 155–176. [Online]. Available: http://dl.acm.org/citation.cfm?id=648204.749438

[9] R. H. Bordini and J. F. Hübner, “Bdi agent programming in agentspeak using jason,” in *IN: PROCEEDINGS OF 6TH INTERNATIONAL WORKSHOP ON COMPUTATIONAL LOGIC IN MULTI-AGENT SYSTEMS (CLIMA VI). VOLUME 3900 OF LNCS.* Springer, 2005, pp. 143–164.

[10] R. H. Bordini and J. F. Hübner, “Bdi agent programming in agentspeak using *jason* (tutorial paper),” in *CLIMA*, 2005, pp. 143–164.

[11] R. H. Bordini, J. F. Hübner, and R. Vieira, “Jason and the golden fleece of agent-oriented programming,” in *Multi-Agent Programming*, 2005, pp. 3–37.

[12] R. H. Bordini, J. F. Hübner, and M. J. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason.* J. Wiley, 2007. [Online]. Available: http://www.worldcat.org/isbn/9780470029008

CHAPTER 5

Previous Research

5.1 Introduction

The research described in Chapters 2, 3, and 4 explores several of the elements proposed as part of this thesis: **use of RL**, **learning plans on BDI systems**, and **extraction of BDI plans from MDPs and POMDPs**.

A more detailed overview of previous research whose elements come closest to the work done as part of this thesis will be discussed on Section 5.2. Section 5.3 will summarize the research and highlight the key differences.

5.2 Research on BDI and Learning

An extensive review of the existing literature in RL and BDI did not uncover any research that made use of RL to learn BDI plans without relying on a-priori knowledge.

As discussed in the **Introduction** on **Chapter 1**, the lack of learning capabilities for BDI systems was recognized as far back as 2004 [1]. Researchers tackled this by augmenting the BDI framework with various learning frameworks including **decision trees**, **self-organizing neural networks**, **hybrid-architectures using low level learners**, and **metaplans for plan hypothesis abduction and plan modifications**. Other relevant research tackled the use of a-priori knowledge, previously learned knowledge and the learning of plans without a-priori knowledge on planning systems, and the integration of learning, planning and execution. **These studies were, however, not investigated in relation to BDI systems**.

Recent research relied on Markov Decision Processes (MDPs) to generate BDI plans from optimal policies for completely specified MDPs [2]. Pereira's work was augmented to work with Partially Observable Markov Decision Processes (POMDPs) using the Witness algorithm. [3]. These two studies come closest to the work done in this thesis. The difference is that neither fully specified MDPs nor POMDPs were considered in this thesis.

Besides the Pereira work, the research done by Karim, **Plans as Products of Learning** [4] is also very similar to the idea studied in this thesis. The difference is that Karim used a learning approach that did not rely on RL.

The next sections describe the details of the work done by Karim, Singh, Dixon, Tan, and Pereira.

Plans as Products of Learning [4]

- **Target**: plan learning and plan improvement
- **Model:** hybrid, inductive
- **Learning Element:** self-organizing neural network (FALCON), hypothesis abduction
- **Goal:** plan learning via plan extraction using PGS and plan improvement using hypothesis abduction

The study investigated a hybrid approach combining a low-level learner with high-level BDI based knowledge extractor and executor called plan generation subsystem (PGS). A PGS algorithm is presented that relies on a-priori clues provided by domain expert. The low-level learned used was FALCON which is a self-organizing neural network. Also investigated was a second approach that used a hypothesis generator to amend existing BDI plans by way of suggesting and executing plans and updating intentions accordingly.

Two examples were investigated. In the hybrid approach **a predator-prey (or pursuit)** task was used. Four predator agents and one prey agent in nontoroidal grid. The task to be learned was for the four predator agents to learn to surround the prey agent on all four sides without any communication. In the hypothesis generator a **rat world (operant conditioning)** task was used. A special BDI interpreter that supported the generation of metaplans for abductions and plan modifications at runtime was used.

Learning Context Conditions for BDI Plan [5]

- **Target:** conditions for plan selection
- **Model** : probabilistic
- **Learning Element** : decision trees
- **Goal**: learn the probability of success for plans

This study highlights the lack of learning from experience particular to the BDI framework as well as the limitations of plan selection that relies on boolean formulas specified at design/implementation time and part of a **plan library**. It explores intelligent plan selection using feedback from plan success or failure to build decision trees that provide the probability of success of plans.

Other relevant research, not directly related to BDI systems, tackled the use of a-priori knowledge, previously learned knowledge and the learning of plans without a-priori knowledge on planning systems, and the integration of learning, planning and execution.

Incorporating Prior Knowledge and Previously Learned Information into Reinforcement Learning Agents [6]

- **Target:** off-policy controller
- **Model:** hybrid
- **Learning Element:** hybrid

- **Goal**: incorporate prior knowledge and previously learned information into RL agents

This study is concerned with the limits and appropriateness of tabula rasa learning and suggests a framework to incorporate a-priori knowledge and previously learned knowledge. The author points that learning tabula rasa might not be "appropriate" for two reasons:

1. system designers may have already embedded some domain-specific knowledge
2. the agent may have learned the task

To solve this problem the author proposes an **off-policy controller** that uses modularized "**prior knowledge sources**" (PKSs) as inputs to an "**exploration control module**". An additional contribution of this study is the discussion of two terms that are of interest when designing RL systems:

1. **state-space deficiency** - refers to features of the state-space that the agent is not able to observe and thus unable learn when instructed by a PKS
2. **representational deficiency** - refers to PKS that use a history of events to make a decision. Reactive RL agents will not be able to learn this task.

The study proposed two approaches, **incremental learning** in which a "large, complex task is decomposed into smaller sub-tasks" with the idea that "solving the sub-tasks may be easier than solving the entire task" and **composable skill synthesis** in which "a problem is broken down into a set of basic skills that the agent must possess in order to complete a task".

Two examples were investigated. For the incremental learning process a **simulated robot domain** task was used which consisted of mobile robot tag. In this task, the agents learn a controller that is then used as a PKS to the next step in the learning process. The task was incrementally learned by first learning how to score with no defense then learning how to score with a single runner and a single defender, and finally by learning how to score with two runners and a single defender. No communication was allowed between the two runners.

In the composable skill synthesis approach a **grid world** task was used. The task's goal was for the agent to move from its start state to a goal state without bumping into any walls. The set of source skills used as PKSs was **object avoidance** (a priori) and **goal homing** (learned in wall-less gridworld).

FALCON: A Fusion Architecture for Learning, COgnition, and Navigation [7]

- **Target:** fusion architecture
- **Model:** connectionist
- **Learning Element:** self-organizing neural network with fuzzy logic
- **Goal**: learn cognitive codes across multi-modal patterns involving sensory input, actions, and rewards

This study describes a cognitive model based on **Adaptive Resonance Associative Map** (ARAM) which is an extension of **Adaptive Resonance Theory** (ART) that fuses sensory input, actions and rewards. ARAM processes a state vector and produces an action vector. When the environment provides a reward vector, ARAM then uses it to associate the state vector, the action vector and the reward vector. ARAM uses fuzzy ART operations.

A simulated minefield navigation task was used to test FALCON. In the task, an autonomous vehicle (AV) started in a random position. The objective was to navigate to randomly selected target position in a specified time frame without hitting a mine. The AV was equipped with five sonar sensors that provided coarse sonar sensor data. The target and mines were stationary.

Um Algoritmo para Extração de um Plano BDI que Obedece uma Política Ótima (An Algorithm for Extracting BDI Plans from an Optimal Policy) [2]

Pereira presents an analysis of hybrid approach BDI-MDP and introduces algorithm **policyToIplan** that extracts BDI plans from optimal policies. The analysis is based on completely specified MDPs.

- **Target:** BDI plans
- **Model:** hybrid BDI-MDP
- **Learning Element:** completely specified MDP
- **Goal**: extract BDI plans from optimal policies

Constructing BDI Plans from Optimal POMDP Policies, with an Application to AgentSpeak Programming [3]

Pereira presents an analysis of hybrid approach BDI-MDP and introduces algorithm **policyToBDIplan** that extracts BDI plans from optimal POMDP policies.

- **Target:** BDI plans
- **Model:** hybrid BDI-MDP (POMDPs)
- **Learning Element:** POMDP using the Witness algorithm
- **Goal**: extract BDI plans from optimal POMDP policies

5.3 Differences Between Proposed Study and Previous Research

Section 5.2 discussed the research whose elements come closest to the work done under this thesis. There are, however, several key differences.

The goal of this thesis is to use reinforcement learning to generate plans without a-priori knowledge on BDI agent systems. The key idea is that the result of reinforcement learning is a policy, or policies in the general case. Since policies map states to actions, the policies can then be used as input to generate plans in BDI agents systems. The approach can then be summarized as a two step process:

1. Use reinforcement learning as the learning module.

2. Use policies learned as input to generate BDI plans.

None of the previous work combines the elements of RL for plan generation on BDI agent systems. The problem selected for study in this thesis is justified by this lack of research exploring the generation of plans in BDI systems using reinforcement learning that does not rely on a-priori knowledge.

Table 5 summarizes previous research and shows the differences to the approach taken in this thesis.

Study	Summary	Differences to Proposed Study
Karim2006 [4]	**Target:** plan learning and plan improvement **Model:** hybrid, inductive **Learning Element:** self-organizing neural network (FALCON), hypothesis abduction **Goal:** plan learning via plan extraction using PGS and plan improvement using hypothesis abduction	Uses self-organizing neural network instead of reinforcement learning. Uses hypothesis abduction (inductive method) instead of value functions
Singh2010 [5]	**Target:** conditions for plan selection **Model:** probabilistic **Learning Element:** decision trees **Goal:** learn the probability of success for plans	Uses decision trees instead of reinforcement learning. Learns probability of of success for plans instead of new plans.
Dixon2000 [6]	**Target:** off-policy controller **Model:** hybrid **Learning Element:** hybrid **Goal:** incorporate prior knowledge and previously learned information into RL agents	Research was not tied to BDI plan learning
Tan2004 [7]	**Target:** fusion architecture **Model:** connectionist **Learning Element:** self-organizing neural network with fuzzy logic **Goal:** learn cognitive codes across multi-modal patterns involving sensory input, actions, and rewards	Uses self-organizing neural network with fuzzy logic instead of reinforcement learning.

Study	**Summary**	**Differences to Proposed Study**
Pereira2007 [2]	**Target:** BDI plans **Model:** hybrid BDI-MDP **Learning Element:** completely specified MDP **Goal:** extract BDI plans from optimal policies	Uses completely specified MDPs instead of MDPs that are not completely specified
Pereira2008 [3]	**Target:** BDI plans **Model:** hybrid BDI-MDP (POMDPs) **Learning Element:** POMDP using the Witness algorithm **Goal:** extract BDI plans from optimal POMDP policies	Uses POMDPs instead MDP that are not completely specified. Uses the Witness algorithm instead of reinforcement learning.

Table 5: Differences Between Proposed Study and Previous Research

List of References

[1] A. Guerra-Hernández, A. E. Fallah-Seghrouchni, and H. Soldano, "Learning in bdi multi-agent systems," in *CLIMA*, 2004, pp. 218–233.

[2] D. R. Pereira and G. P. Dimuro, "Um algoritmo para extração de um plano bdi que obedece uma política Ótima," in *Workshop-Escola de Sistemas de Agentes para Ambientes Colaborativos, Pelotas, Anais do WESAAC 2007*, 2007.

[3] D. R. Pereira, L. Vargas, G. P. Dimuro, and A. C. R. Costa, "Constructing bdi plans from optimal pomdp policies, with an application to agentspeak programming," in *XXXIV - Conferencia Latinoamericana de Informtica, 2008, Santa Fe. Proceedings of CLEI 2008 (Lecture Notes in Computer Science)*, ser. CLEI 2008, vol. 1. Springer, 2008, pp. 1–10.

[4] S. Karim, B. Subagdja, and L. Sonenberg, "Plans as products of learning," in *IAT*, 2006, pp. 139–145.

[5] D. Singh, S. Sardiña, L. Padgham, and S. Airiau, "Learning context conditions for bdi plan selection," in *AAMAS*, 2010, pp. 325–332.

[6] K. R. Dixon, R. J. Malak, and P. K. Khosla, "Incorporating prior knowledge and previously learned information into reinforcement learning agents," Carnegie Mellon University, Tech. Rep., 2000.

[7] A.-H. Tan, "Falcon: A fusion architecture for learning, cognition, and navigation," in *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, vol. 4, july 2004, pp. 3297 –3302 vol.4.

CHAPTER 6

Experimental Implementation

6.1 RL Problem Selection

The RL problem selected was based on the problem by **Poole and Mackworth**[1][1] described at:

`http://artint.info/html/ArtInt_262.html#davids-simple-game-ex`

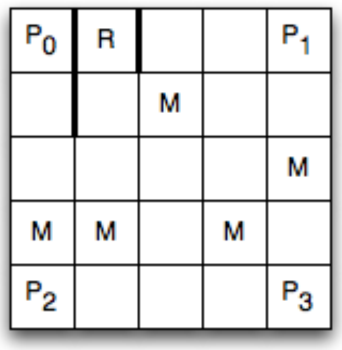

Figure 5. RL problem

6.2 RL Problem Description

The original version of the new RL problem description is included verbatim from **Poole** and **Mackworth**:

> "There are 25 grid locations the agent could be in. A prize could be on one of the corners, or there could be no prize. When the agent lands on a prize, it receives a reward of 10 and the prize disappears. When there is no prize, for each time step there is a probability that a prize appears on one of the corners. Monsters can appear at any time on one of the locations marked M. The agent gets damaged if a monster appears on the square the agent is on. If the agent is already damaged, it receives a reward of -10. The agent can get repaired (i.e., so it is no longer damaged) by visiting the repair station marked R.

[1]In the spirit of academic integrity and honesty, I disclose that **Poole** and **Mackworth** provide an applet implementation for this problem, their code was not studied, referenced or consulted at all. All RL code implementation is my own.

In this example, the state consists of four components: $< X, Y, P, D >$, where X is the X-coordinate of the agent, Y is the Y-coordinate of the agent, P is the position of the prize (P=0 if there is a prize on P0, P=1 if there is a prize on P1, similarly for 2 and 3, and P=4 if there is no prize), and D is Boolean and is true when the agent is damaged. Because the monsters are transient, it is not necessary to include them as part of the state. There are thus 5∗5∗5∗2 = 250 states. The environment is fully observable, so the agent knows what state it is in. But the agent does not know the meaning of the states; it has no idea initially about being damaged or what a prize is.

The agent has four actions: up, down, left, and right. These move the agent one step - usually one step in the direction indicated by the name, but sometimes in one of the other directions. If the agent crashes into an outside wall or one of the interior walls (the thick lines near the location R), it remains where is was and receives a reward of -1.

The agent does not know any of the story given here. It just knows there are 250 states and 4 actions, which state it is in at every time, and what reward was received each time.

This game is simple, but it is surprisingly difficult to write a good controller for it."

6.3 RL Problem Terminology Changes

For this research I changed some of the terminology. The following term are used:

Prizes are referred to as **rewards**. While this term is also used as part of the RL terminology, it should be clear from the context when it refers to RL rewards in general (which can be negative) and when it refers to rewards as the main goal for the agent. Monsters were replaced by **damage positions**.

Action set $A = \{WEST, NORTH, EAST, SOUTH\}$ replaces action set $A = \{up, down, left, right\}$.

State component D is replaced by $AgentStatus = \{U, D\}$ where U stands for **undamaged** and D stands for **damaged**.

Rewards set $R = \{reward = 10, wall\,bump = -1, damage = -1000\}$.

6.4 RL Problem Implementation

The RL problem was implemented in Java following the RL learning model seen in chapter 1 and in figure 6.

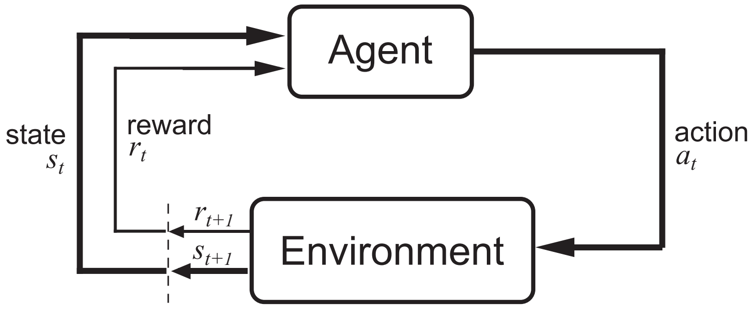

Figure 6. Agent-Environment Model [2]

The code contains 4 main classes:

1. Agent
2. Environment
3. Simulation
4. GUI

The **agent** is initialized and then used to initialize the **environment**. The **simulation** thread runs the loop that implements the learning functionality by having the agent interact with the environment under diverse control parameters. The **GUI** is the entry point because it provides visualization of the agent actions, rewards, resulting states, and learning process.

Figure 7 shows the the environment grid with the agent at position $(2, 2)$. The reward is shown at position $(4, 4)$. The damage positions are shown at positions $(2, 1)$, $(4, 2)$ $(0, 3)$, $(1, 3)$ and $(3, 3)$.

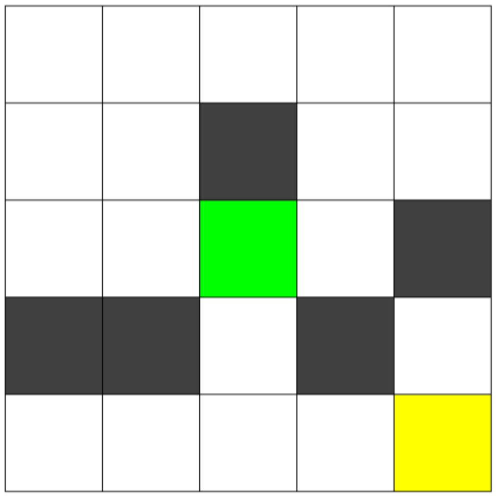

Figure 7. Grid of new RL problem (Agent in green, reward in yellow, damage positions in gray)

The learning functionality for the RL agent was implemented using the *Q* **learning algorithm**. Refer to Section 3.9, Table 1 for the pseudocode for *Q* learning.

Once the simulation starts, the following events take place:

1. An **equiprobable** initial policy is generated.

2. The agent **explores** the environment for **1000 episodes**. Each **episode** lasts **1000 steps**.

3. The **learned policy** is written in text format to a file. The policy is also serialized and written to file for use in the **BDI environment**.

4. The agent does $\epsilon - greedy$ **selection of actions**. For a 1000 episodes. The agent select the greedy action 90% of the time. The agent still explores the environment the remaining 10% of the time.

5. The simulation **stops** at **2000 episodes**.

6.5 RL Initial Policies

The initial **equiprobable policy** is the same for all configurations. The effect of this policy is that when the simulation runs the agent starts without any bias for any of the possible actions. All actions are equally possible and have the same utility.

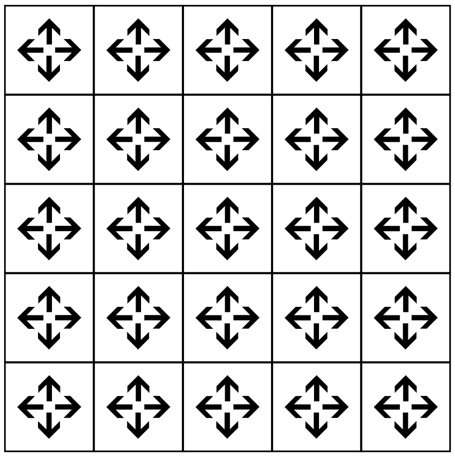

Figure 8. Initial Equi-probable Policy for All Learning Configurations

6.6 BDI Implementation

The BDI functionality was implemented in **Jason**. The code contains 4 main classes:

1. BdiEnvironment
2. EnvironmentMechanics
3. AgentGenerator
4. GUI

The **BdiEnvironment** class extends Jason's **Environment** class and is the entry point into Jason's infrastruture. The **EnvironmentMechanics** class is reused from the RL implementation to provide actual feedback from the environment resulting from the BDI agent's actions. The **AgentGenerator** class implements the main functionality of this research, its purpose is to convert learned RL policies into BDI agents. The GUI class is also reused from the RL implementation to provide visualization of the BDI agent actions, rewards, and resulting states.

6.7 BDI Agent Generation

The **AgentGenerator** class takes the serialized policy learned by the RL agent and converts it into a a BDI agent.

List of References

[1] D. Poole and A. Mackworth, "Artificial intelligence: Foundations of computational agents," 2010. [Online]. Available: http://artint.info/html/ArtInt_262.html#davids-simple-game-ex

[2] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

CHAPTER 7

Results

This chapter presents the results of RL agents learning using the Q learning algorithm and the use of such learned policies to generate plans for BDI agents without relying on a-priori knowledge.

The Q learning algorithm resulted in RL learned policies that are presented and discussed in **Section 7.1 RL Learned Policies**. These policies map the greedy action(s) agents need to select in each state and encapsulate the knowledge learned through **exploration-exploitation** of the environment. Learned policies can show interesting, even counter-intuitive behavior that might not make sense at first. Interpreting learned policies requires keeping track of the actual state of the agent to be able to reference the applicable policy. **Section 7.2 Interpretation of RL Learned Policies** discusses how to interpret policies and what could happen when it is done incorrectly. Learned policies are highly dependent on the structure of the reward set and this affects the agent's behavior. The interrelationship of these factors is explored in **Section 7.3 Reward Set, Learned Policies, and Agent Behavior**. Three environment-agent configurations were tested. These are shown in Table 6. The performance of RL agents during the 1000 episode exploration phase (**random action selection**) and 1000 (**ϵ-greedy) phase (90% greedy and 10% random action selection**), for all three configurations tested, is shown in **Section 7.4 RL Results**. The successful use of RL learned policies to generate BDI agents without relying on a-priori knowledge is the topic of **Section 7.5 BDI Agent Generation**. It includes the generation process and provides an example fragment of the AgentSpeak code generated. The performance of the BDI agents is presented and discussed in **Section 7.6 BDI**

Agent Results. The results of a randomly generated BDI agent, to examine what happens when a BDI agent blindly follows plans without being able to learn, is presented and discussed in **Section 7.7 Random BDI Agent**. Finally, a summary of the results for this thesis are presented in **Section 7.8 Thesis Research Results**.

Configuration	Damage Probability	Description
000	$p(damage) = 0$	Agent status does **NOT** change from **undamaged** to **damaged** when stepping on damage position. Agent receives no damage when stepping on damage position.
001	$p(damage) = 0.5$	Agent status changes from **undamaged** to **damaged** 50% of the time when stepping on damage position. Agent receives damage 50% of the time when stepping on damage position if it is in **damaged** status.
002	$p(damage) = 1$	Agent status changes from **undamaged** to **damaged** 100% of the time when stepping on damage position. Agent receives damage 100% of the time when stepping on damage position if it is in **damaged** status.
003	$p(damage) = 1$	Agent status changes from **undamaged** to **damaged** 100% of the time when stepping on damage position. Agent receives damage 100% of the time when stepping on damage position if it is in **damaged** status. Used for random BDI agent.

Table 6: Environment-Agent Configurations

7.1 RL Learned Policies

Running the Q **learning algorithm** resulted in learned policies through repeated **exploration-exploitation** of the environment. Starting from an equiprobable policy the agent-environment interactions resulted in changes to the **ac-**

tion values for most of the actions in the action set. The selection of the greedy action for every state then determined the policies approximating the optimal policies.

The resulting learned policies are different for all RL agents. This is no surprise given that the RL agents tested were working in different environments. Visual examination of the polices shows that the actions for many states have collapsed to a **single optimal action**. States with more that one action result from those actions having **action values that are tied** or **the agent being unable to explore the environment enough to select an optimal action**.

The learned policies for all three configurations tested are shown in figures 9 to 20. The figures show how the learned policies change as the reward position and the status of the agent change.

7.1.1 Configuration 000 Learned Policies

Figures 9 to 12 show the learned policies for configuration **000**. Examination of the policies as the reward position changes from reward position 0 to reward position 3 shows that the policies guide the agent to select the action that moves the agent in the direction of the reward position.

In general, when the reward position is at position 0, the agent tends to move **WEST** and **NORTH**. When the reward position is at position 1, the agent tends to move **EAST** and **NORTH**. When the reward position is at position 2, the agent tends to move **WEST** and **SOUTH**. Finally, when the reward position is at position 3, the agent tends to move **EAST** and **SOUTH**.

7.1.2 Configuration 001 Learned Policies

Figures 13 to 16 show the learned policies for configuration **001**. Just as in configuration **000**, examination of the policies as the reward position changes from reward position 0 to reward position 3 shows that the policies guide the agent to select the action that moves the agent in the direction of the reward position.

The difference for this configuration is that the agent can receive damage from the damage position 50% of the time. How does this affect the agent's behavior? The agent learns how to deal with the damage positions in the environment. For example, an agent with **damaged** status starting at position $(0, 4)$ with reward position at 0 would first move **EAST** two steps, then **NORTH** two steps to avoid the damage positions. Only then it would move two steps **WEST** and **NORTH** until it gets to the reward position.

7.1.3 Configuration 002 Learned Policies

Figures 17 to 20 show the learned policies for configuration **002**. Just as in configuration **000** and **001**, examination of the policies as the reward position changes from reward position 0 to reward position 3 shows that the policies guide the agent to select the action that moves the agent in the direction of the reward position.

In this configuration the agent, similarly to configuration **001**, learns how to deal with the damage positions in the environment. The difference between configurations **001** and **002** is that in configuration **001**, when the agents steps on a damage position, there is 50% probability that the agent will **not** change from **undamaged** to **damaged** status. This allows the agent to follow the shortest routes between rewards positions as long as it continues to have **undamaged** status. In configuration **002**, stepping on a damage position changes the agent from **undamaged** to **damaged** status with 100% probability. As a result, the agent learns to avoid the damage positions at the cost of having to follow longer routes between the reward positions.

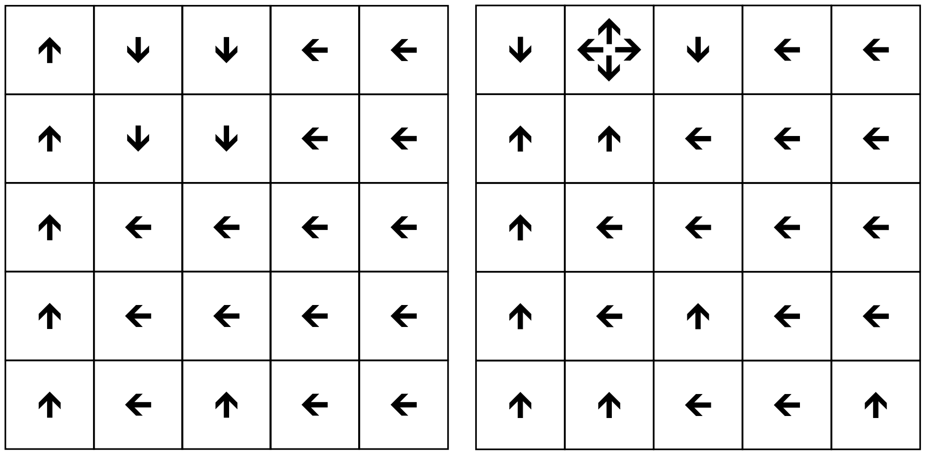

Figure 9. Learned Policies for Configuration 000, Reward at 0, Undamaged and Damaged

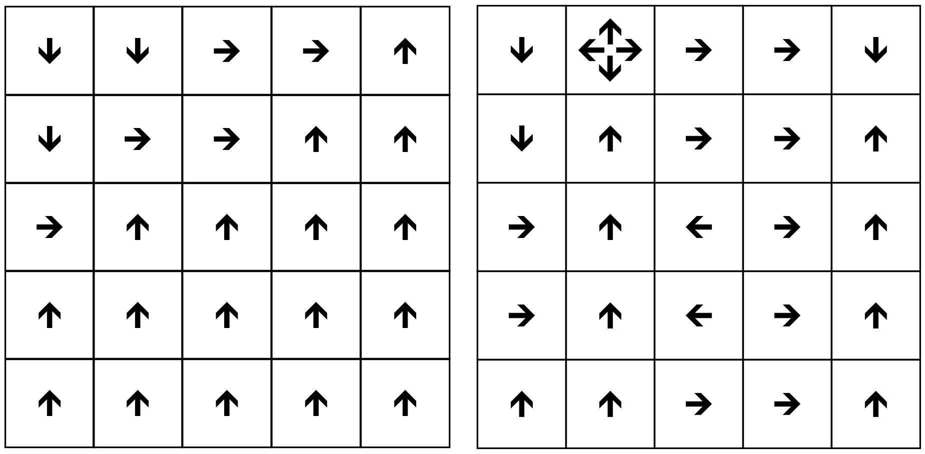

Figure 10. Learned Policies for Configuration 000, Reward at 1, Undamaged and Damaged

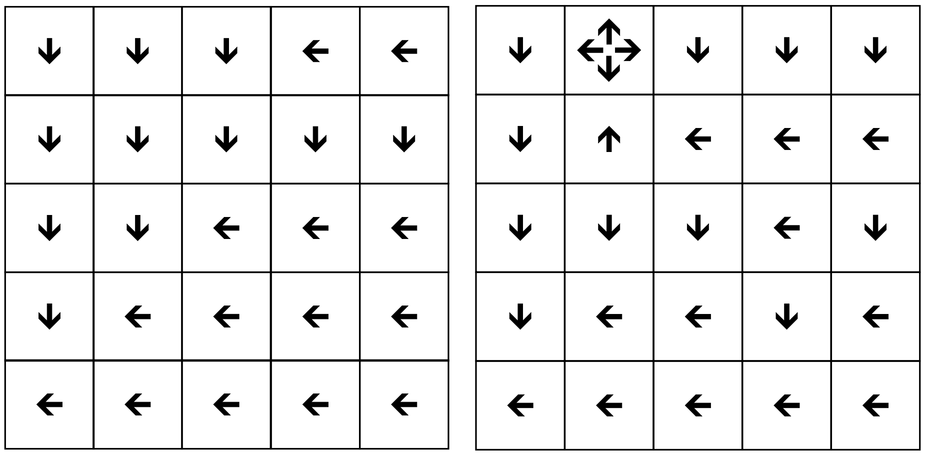

Figure 11. Learned Policies for Configuration 000, Reward at 2, Undamaged and Damaged

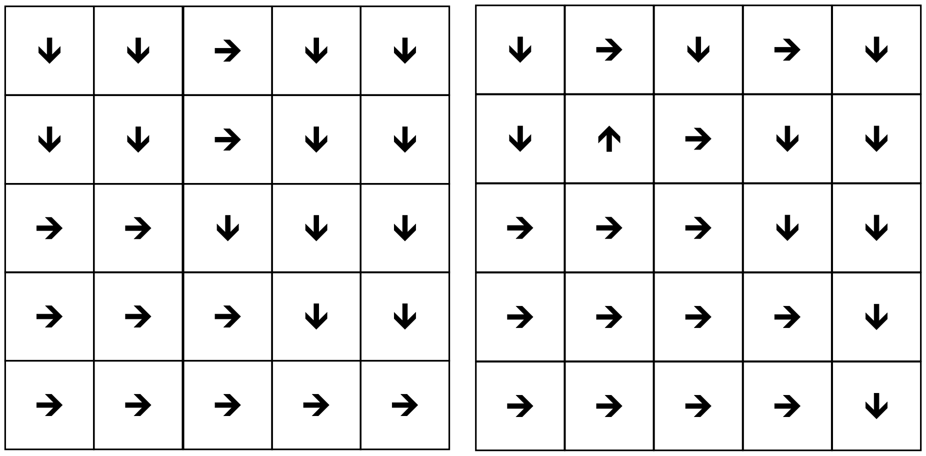

Figure 12. Learned Policies for Configuration 000, Reward at 3, Undamaged and Damaged

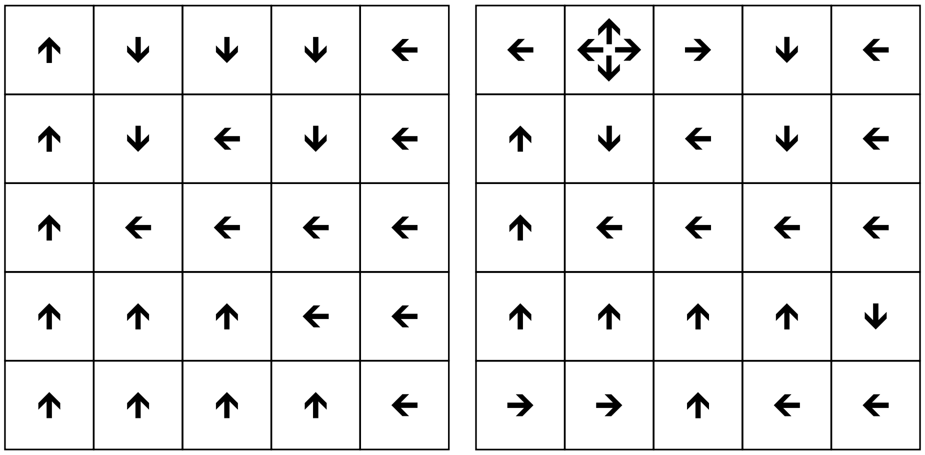

Figure 13. Learned Policies for Configuration 001, Reward at 0, Undamaged and Damaged

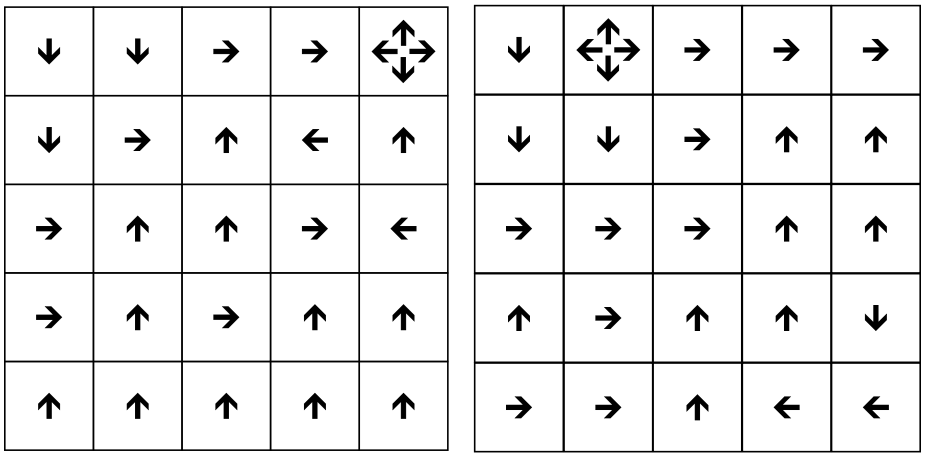

Figure 14. Learned Policies for Configuration 001, Reward at 1, Undamaged and Damaged

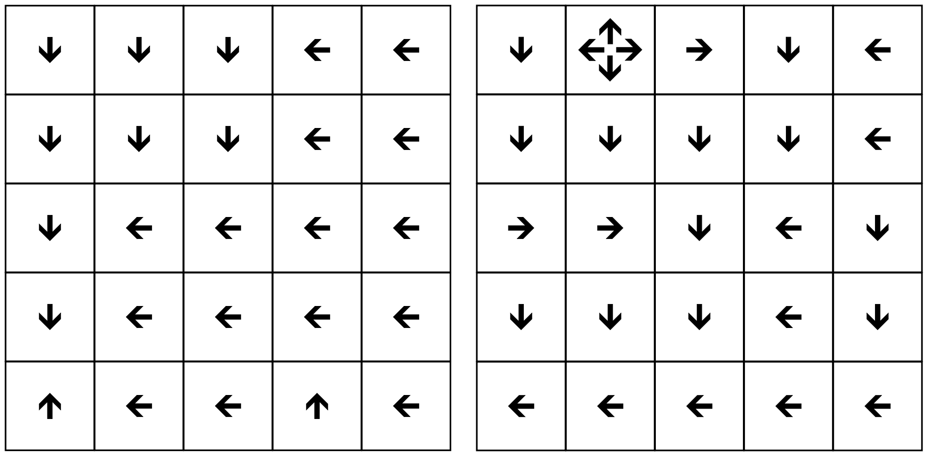

Figure 15. Learned Policies for Configuration 001, Reward at 2, Undamaged and Damaged

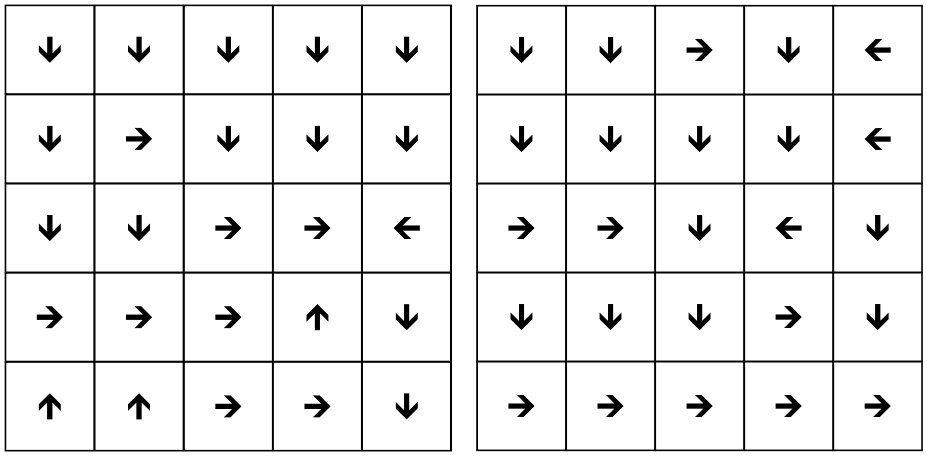

Figure 16. Learned Policies for Configuration 001, Reward at 3, Undamaged and Damaged

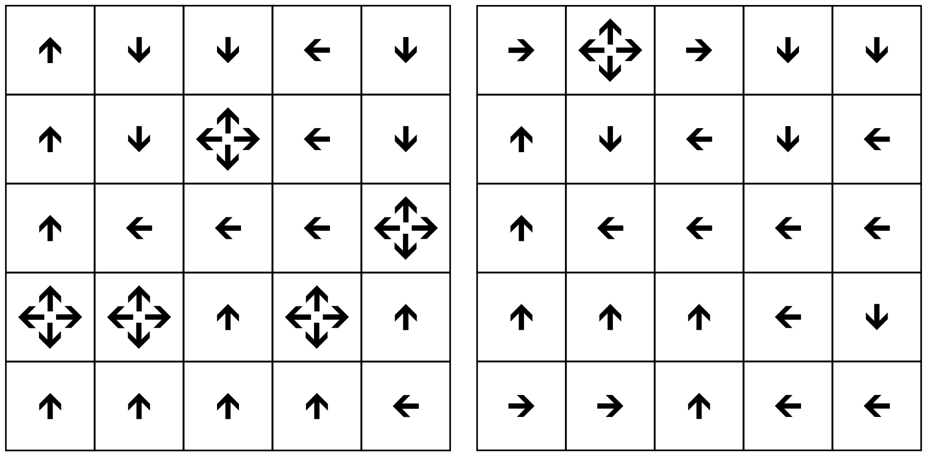

Figure 17. Learned Policies for Configuration 002, Reward at 0, Undamaged and Damaged

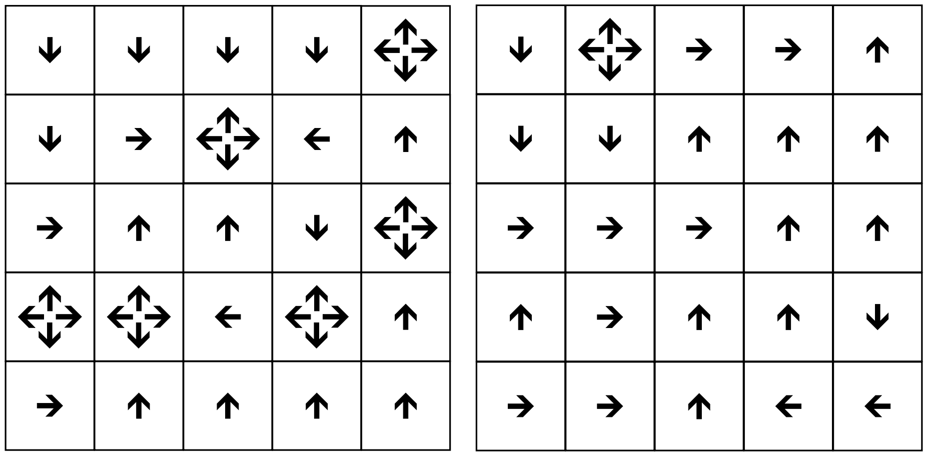

Figure 18. Learned Policies for Configuration 002, Reward at 1, Undamaged and Damaged

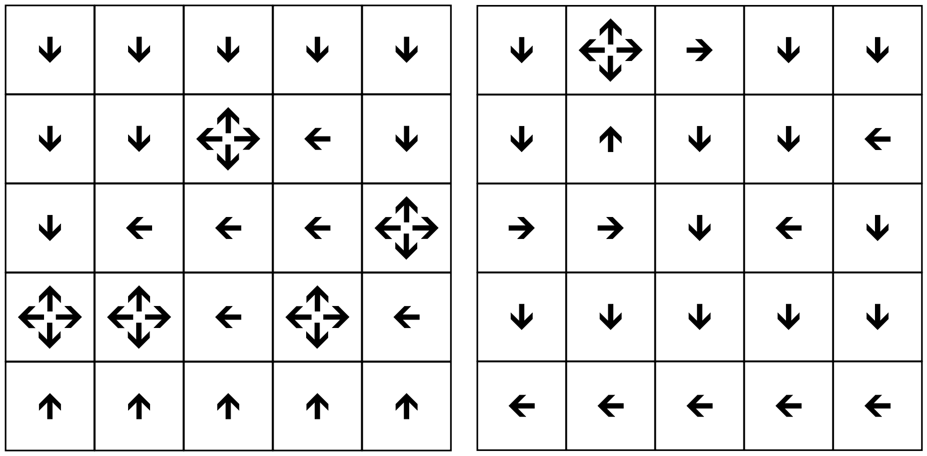
Figure 19. Learned Policies for Configuration 002, Reward at 2, Undamaged and Damaged

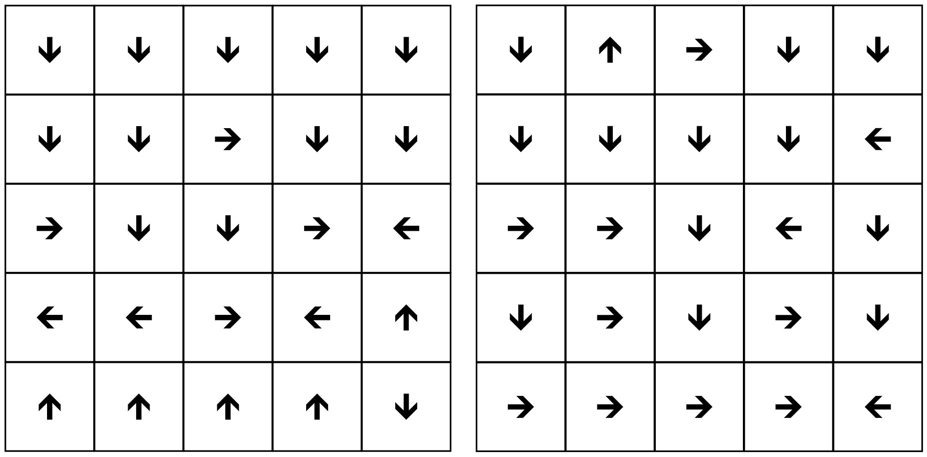
Figure 20. Learned Policies for Configuration 002, Reward at 3, Undamaged and Damaged

7.2 Interpretation of RL Learned Policies

One must be careful when interpreting the RL learned policies. Let us examine an example to show how the policies work and how easy it can be to miss a state change that changes the policy in use.

Consider the case of the RL agent in configuration 0002 [$p(damage) = 1$] with initial $agent\_state = D$ and position $(1, 0)$ [the repair position] and reward position 3 at $(4, 4)$. If we follow the policies in figure 20 we end up the the following sequence $< X, Y, P, S >$, where X is the agent's X position, Y is the agent's Y position, P is the reward position, and S is the agent's status:

$$< 1, 0, 3, U >, < 1, 1, 3, U >, < 2, 1, 3, U >, < 3, 1, 3, U >, < 3, 2, 3, U >, < 4, 2, 3, U >$$

Suddenly the agent finds itself in a loop between the following two states:

$$< 3, 2, 3, U > \longleftrightarrow < 4, 2, 3, U >$$

What happened? **We overlooked that when the agent gets to position $(2, 1)$ the agent changes status to damaged because position $(2, 1)$ is a damage position with damage happening with $p = 1$.** Instead, the correct sequence is:

$$< 1, 0, 3, U >, < 1, 1, 3, U >, < 2, 1, 3, D >, < 2, 2, 3, D >, < 2, 3, 3, D >, < 2, 4, 3, D >$$

$$\cdots, < 3, 4, 3, D >, < 4, 4, 3, D >$$

which gets the reward at reward position 3, position $(4, 4)$.

Learned policies can show interesting, even counter-intuitive behavior that is highly dependent on the structure of the reward set. This is the topic discussed more thoroughly in the next section, section 7.3.

7.3 Reward Set, Learned Policies, and Agent Behavior

Applying RL to learning tasks can result in interesting and sometimes even counter-intuitive policies that are highly dependent on the structure of the reward set. RL researchers and practitioners need to be careful in selecting the reward structure such that the right behavior is learned.

The following example comes directly from the research done for this thesis. The RL task description has a repair station where the agent can repair itself if it is damaged. During one of the learning experiments, a reward was set to reward the agent's action consisting of reaching the repair station when **damaged**. The problem with this approach was that it assigned a high reward to the repair action in relation to the reward for the agent getting damage. The agent learned to exploit this reward differential to get higher rewards. In other words:

$$reward(D \to U) + reward(damage\_position) > reward(reward\_position)$$

This resulted in the following clever behavior by the agent. The agent would get damage which would change its status from undamaged (U) to damaged (D). It would follow by going to the repair position to get repaired and receive the reward for changing its status from damaged (D) to undamaged (U). The reward positions were being ignored. The agent learned a behavior that maximized it average returns even though it was not learning what I was intending it to learn.

> "The agent always learns to maximize its reward. If we want it to do something for us, we must provide rewards to it in such a way that in maximizing them the agent will also achieve our goals. It is thus critical that the rewards we set up truly indicate what we want accomplished. In particular, the reward signal is not the place to impart to the agent prior knowledge about how to achieve what we want it to do. For example, a chess-playing agent should be rewarded only for actually winning, not for achieving subgoals such taking its opponent's pieces or gaining control of the center of the board. ***If achieving these sorts of subgoals were rewarded, then the agent might find a way to achieve them without achieving the real goal.*** For example,

> it might find a way to take the opponent's pieces even at the cost of losing the game. The reward signal is your way of communicating to the robot what you want it to achieve, not how you want it achieved." [emphasis added] [1]

7.4 RL Results

All three RL agents learned policies policies approximating the optimal policies after $1,000$ steps of simulation experience (1000 episodes lasting 1000 steps each).

The results for the RL agent in **configuration 000** is shown in figure 21. In this configuration all damage probabilities were set to 0. The effect of damage probabilities being 0 is that the agent is free to move around and **the only negative rewards come from the agent bumping into the walls in the environment**. As a result, the average returns for this agent are higher than for those in configuration 001 and 002. The agent shows an average return of approximately -100 in the exploration phase and 2750 in the ϵ-greedy phase.

The results for the RL agent in **configuration 001** is shown in figure 22. In this configuration all damage probabilities were set to 0.5. In this configuration with damage probabilities > 0, **the agent is not free to move around without the possibility of having negative rewards due to damage and bumping into the walls**. The average returns clearly show it with approximately -9500 in the exploration phase and -1000 in the ϵ-greedy phase.

The results for the RL agent in **configuration 002** is shown in figure 23. In this configuration all damage probabilities are set to 1. **The agent is certain to have negative rewards due to damage and bumping into the walls**. This results in even lower average returns of -19000 in the exploration phase and -2500 in the ϵ-greedy phase.

Why the negative returns when operating in the ϵ-greedy phase for all three configurations? The negative returns result from the implementation of the ϵ-greedy phase with 90% greedy and 10% random action selection. Even though the agents are acting greedily they do so only 90% of the time. The remaining 10% they are choosing random actions.

The results for all 3 RL agents combined in a single graph are shown in figure 24.

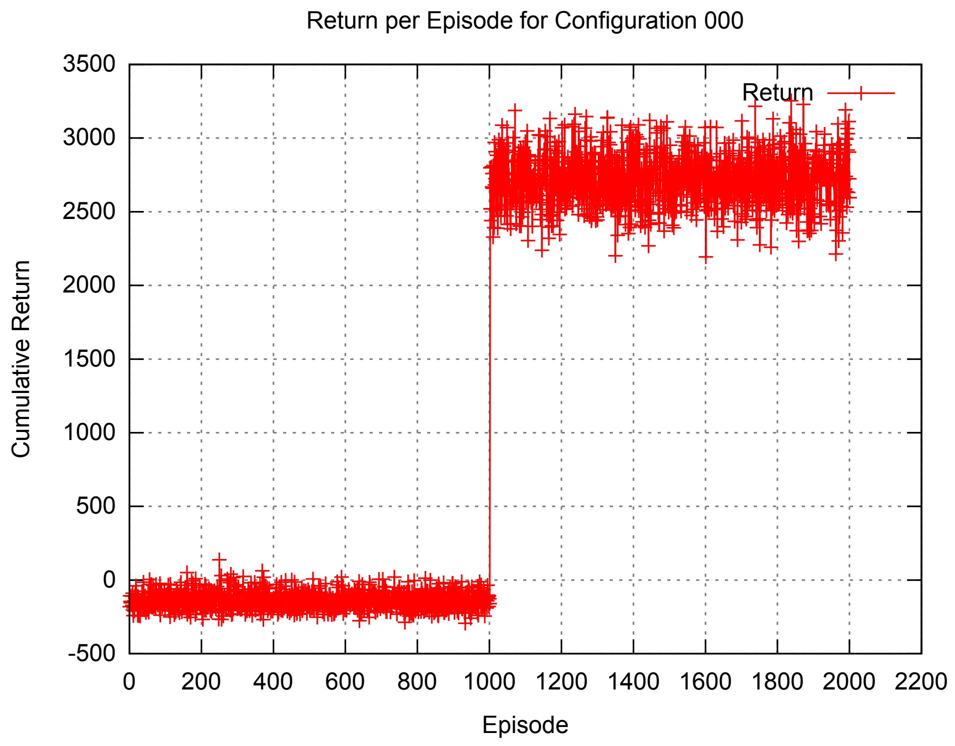

Figure 21. Configuration 000 Results: All damage probabilities are set to 0

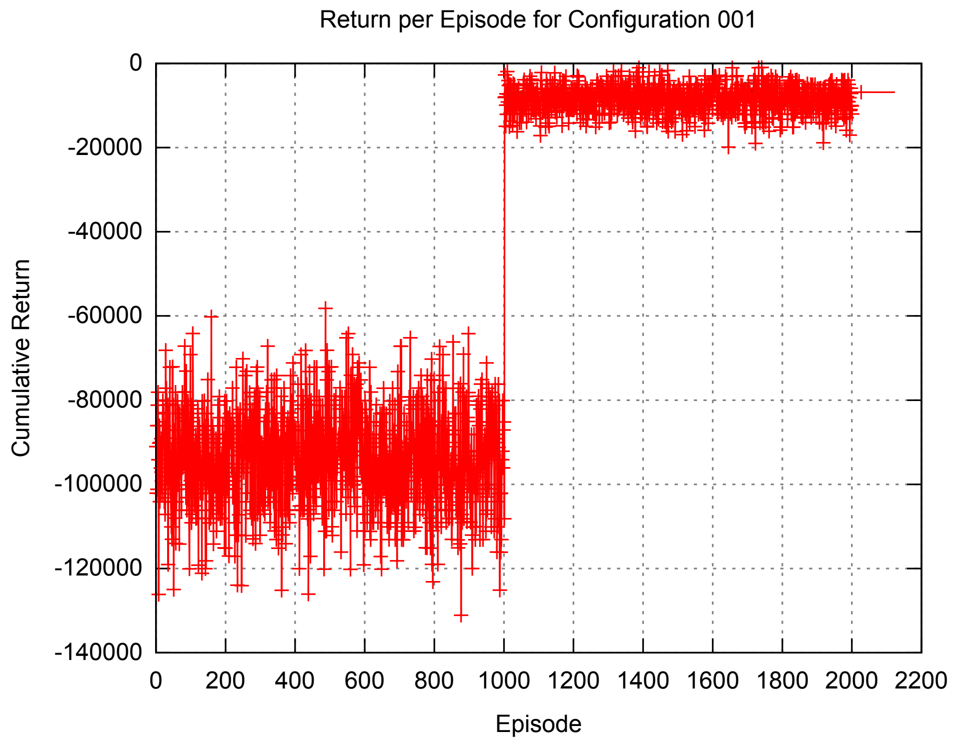

Figure 22. Configuration 001 Results: All damage probabilities are set to 0.5

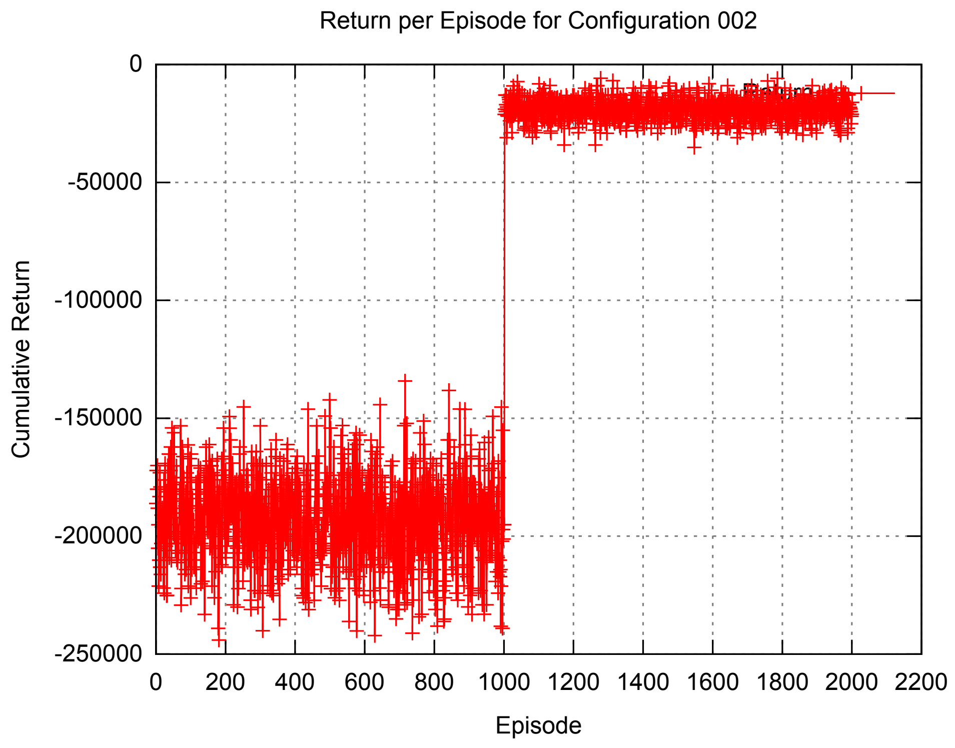

Figure 23. Configuration 002 Results: All damage probabilities are set to 1

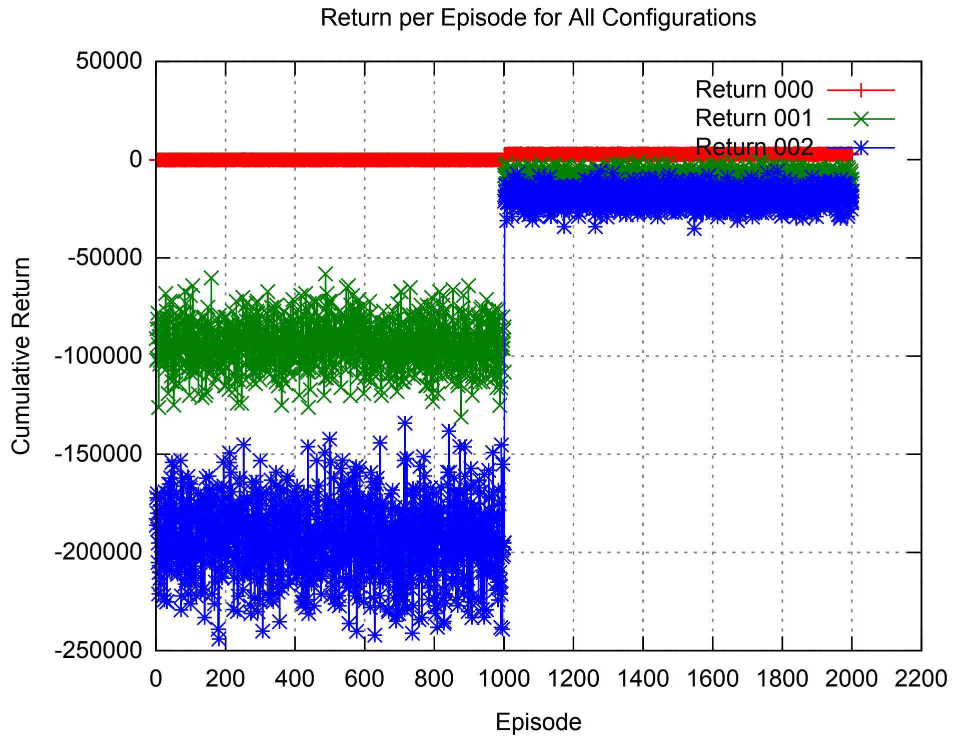

Figure 24. All Configurations Results

7.5 BDI Agent Generation

An **AgentGenerator** class takes the serialized policy learned by the RL agent and converts it into a a BDI agent. A fragment of the the generated code for bdi_agent000 is shown.

```
// Agent bdi_agent000 generated from Q values

/* Initial beliefs and rules */

return(0).
calculation(done).

/* Initial goals */

/* Plans */

+state(0,0,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,0,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

...

+!calculateReturn : not calculation(done)
<- ?reward(R);
   .print("Reward is: ", R);
   ?return(Rt);
   Return = Rt+R;
   -reward(R);
   -+return(Return);
   .print("Total return is: ", Return);
   +calculation(done);
   ?state(X,Y,P,S);
   -+state(X,Y,P,S).
```

Initial beliefs **return(0)** and **calculation(done)** are used to indicate that the agent believes its reward return is 0 and that it is done calculating reward returns. The plans consists of observing the state, performing the optimal action (ties are randomly broken for tied actions during agent generation), observation of the reward and calculation of the return.

A total of 3 agents were generated: **bdi_agent000**, **bdi_agent001**, and **bdi_agent002**. The full listings are included as an appendix.

7.6 BDI Agent Results

After generating the BDI agents, they were evaluated in their environments for 200 episodes lasting 1000 steps each.

The results for **bdi_agent000**, generated from the learned policy of RL agent 000, are shown in figure 25. In this configuration all damage probabilities are set to 0. The average return was approximately 3100.

The results for **bdi_agent001**, generated from the learned policy of RL agent 001 are shown in figure 26. In this configuration all damage probabilities are set to 0.5. The average return was approximately 2250.

The results for **bdi_agent002**, generated from the learned policy of RL agent 002, are shown in figure 27. In this configuration all damage probabilities are set to 1. The average return was approximately 2250.

The results for all 3 BDI agents combined in a single graph are shown in figure 28.

Why are the average returns positive for the BDI agents and negative for the RL agents? This results from the implementation of the BDI **AgentGenerator** class. The RL agents use ϵ-greedy phase with 90% greedy and 10% random action selection, meaning that they still perform random actions that can result in negative rewards. The BDI agents were implemented by selecting the greedy action for all possible states; they do not perform any random action selection. In other words, the RL agents used an exploration-exploitation strategy while the BDI agents only used an exploitation strategy.

The difference in average returns for configuration 000 versus 001 and 002 can be explained by fact that the agent is not concerned with damage positions in configuration 000, it will never get damage. In contrast, the agents have to learn how to maneuver between the damage positions for configurations 001 and 002. This learned behavior ends up increasing the travel distance between reward positions and results in less overall average returns because the number of steps per episode is fixed.

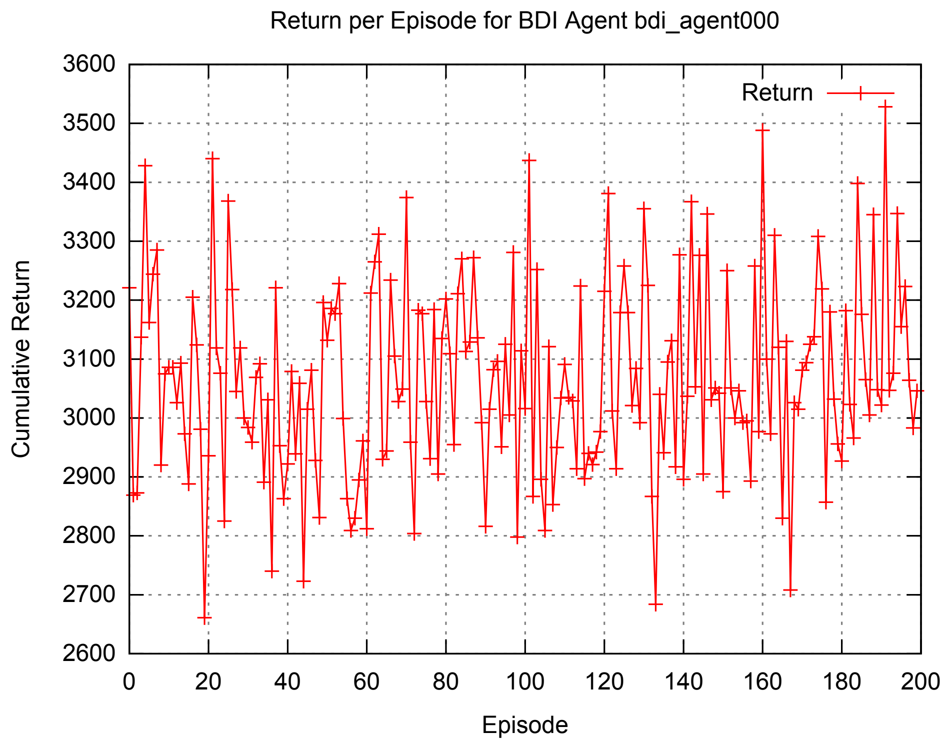

Figure 25. Returns for BDI Agent bdi_agent000

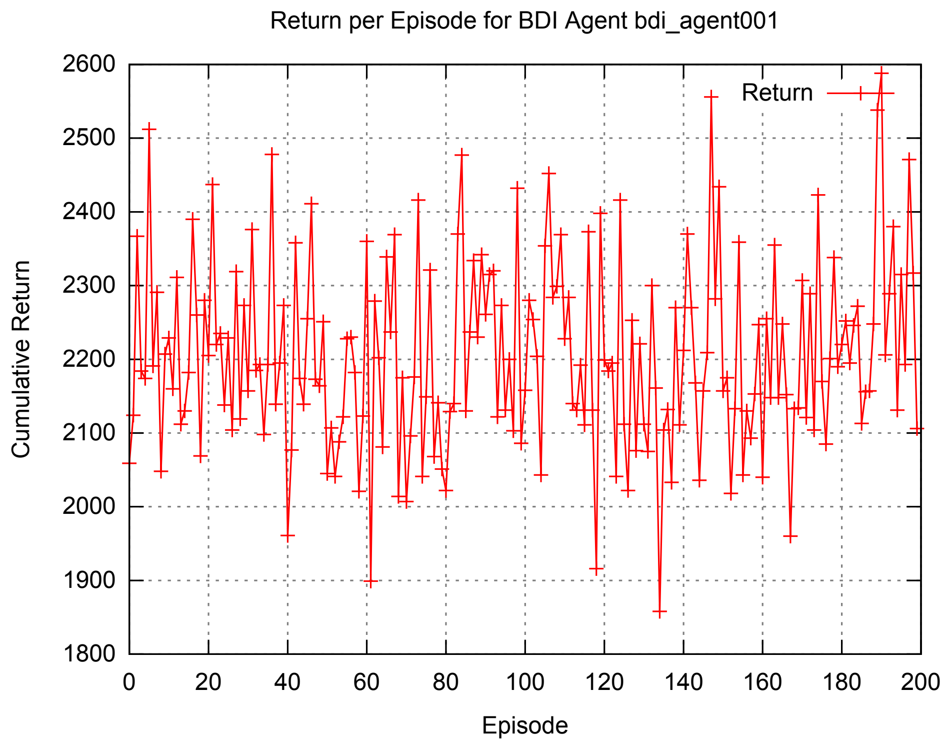

Figure 26. Returns for BDI Agent bdi_agent001

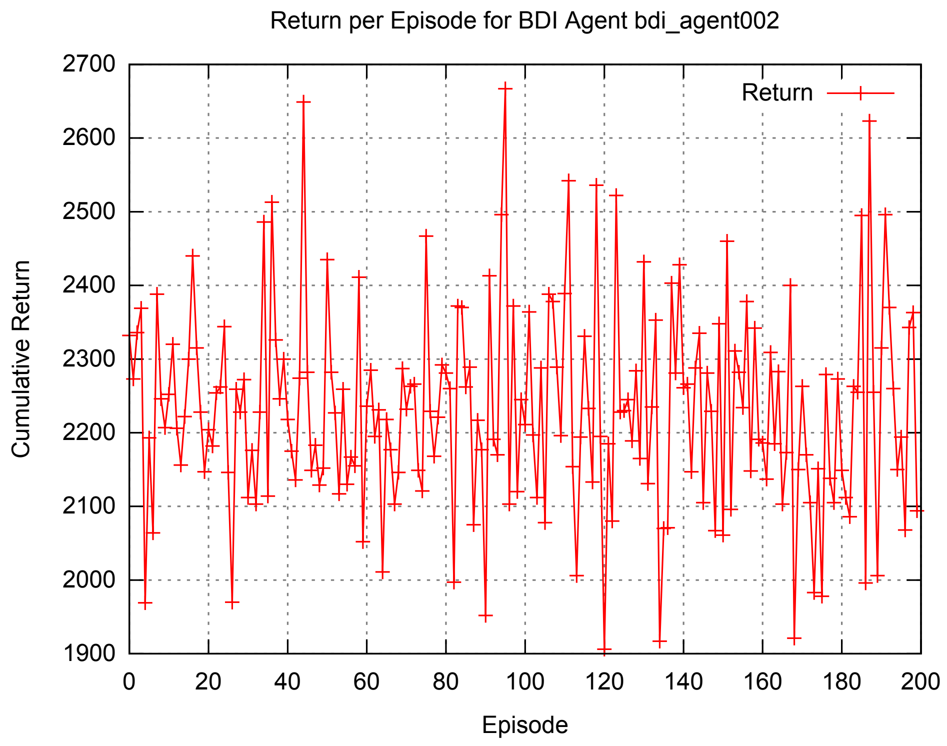

Figure 27. Returns for BDI Agent bdi_agent002

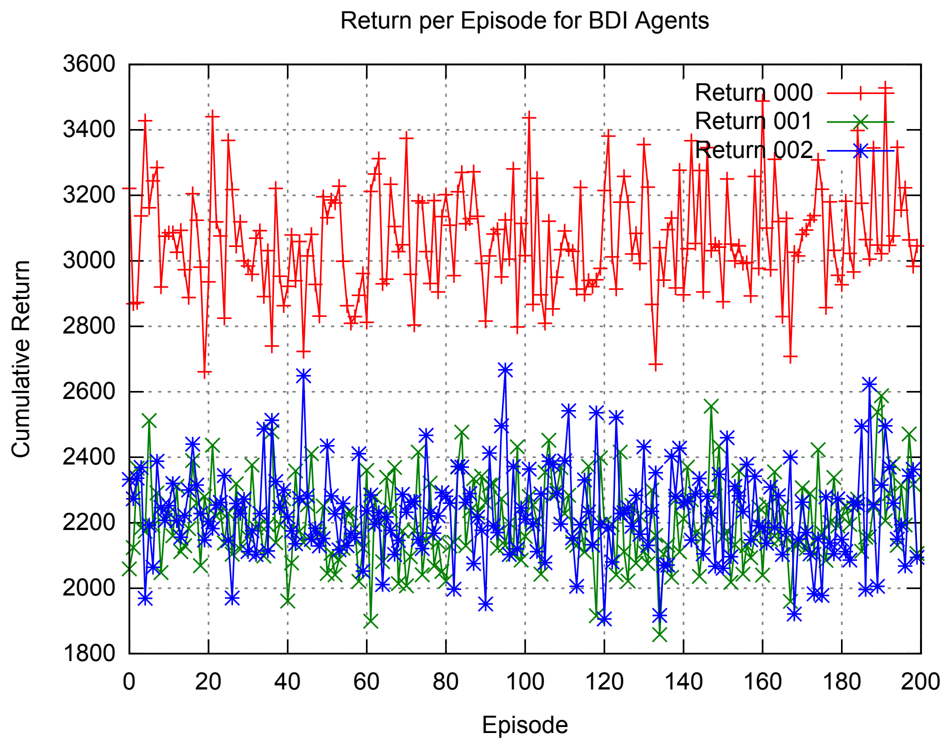

Figure 28. Return for All 3 BDI Agents

7.7 Random BDI Agent

A random BDI agent was created by randomly selecting the best action from the actions available. After a few iterations it became clear how easy it was for the resulting agent to have loops that were not just sub-optimal but completely bad. The agent gets in a 2 step cycle stepping over a damage position over and over. Because the agent's behavior is hard-coded into its plan, the agent is unable to learn. The end result is a pathological loop with returns of $-500,000$.

Figure 29 shows the initial sequence that ends in a fatal loop for the random BDI agent.

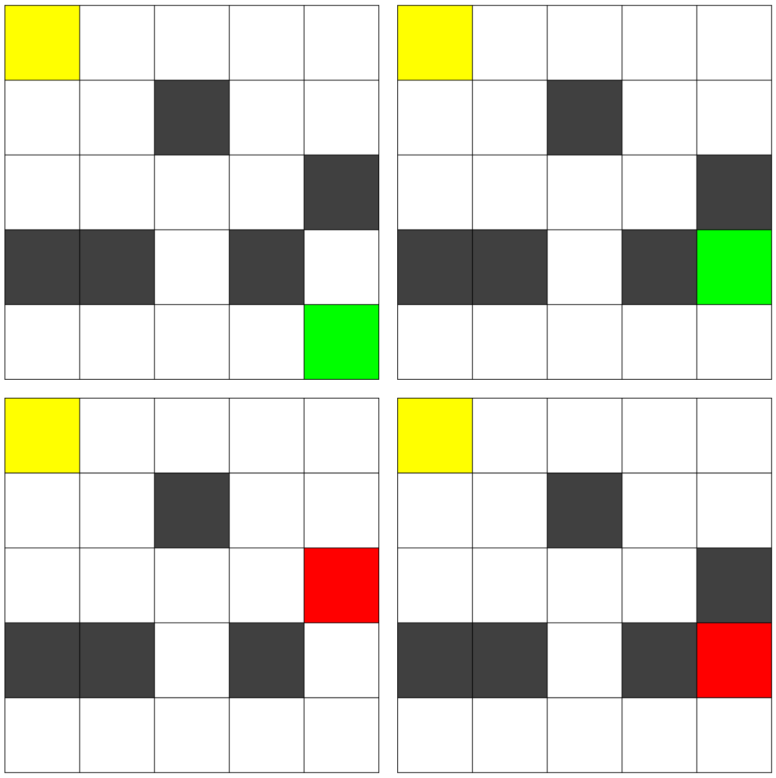

Figure 29. Random BDI Agent Sequence Steps 0,1

Figures 30 and 31 show the returns for the random BDI agent and its comparison with the returns for the other BDI agents.

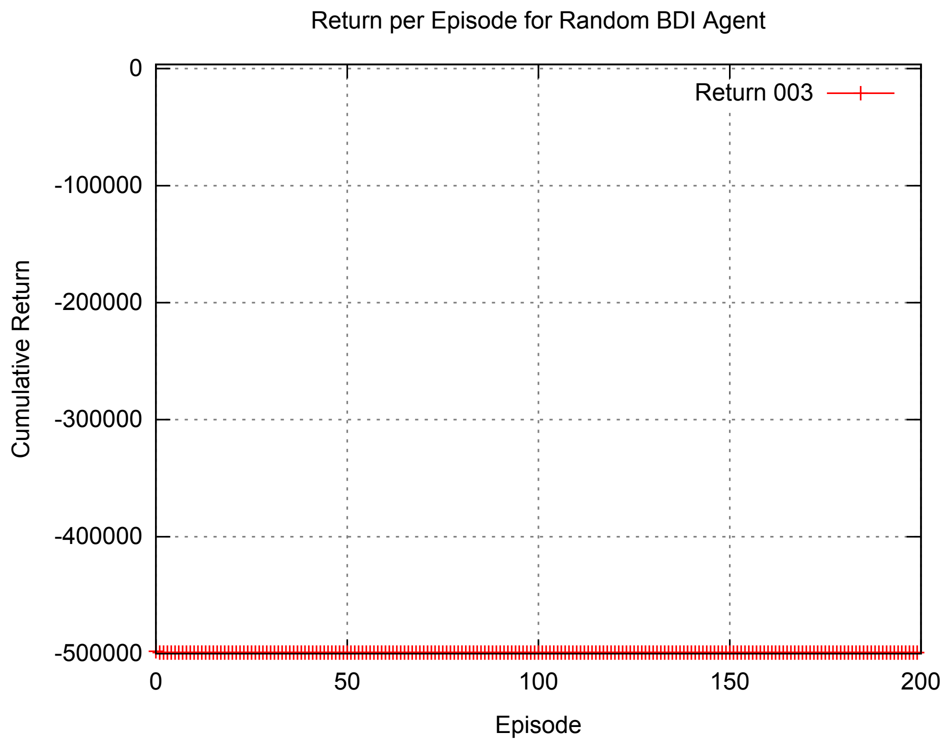

Figure 30. Returns for Random BDI Agent

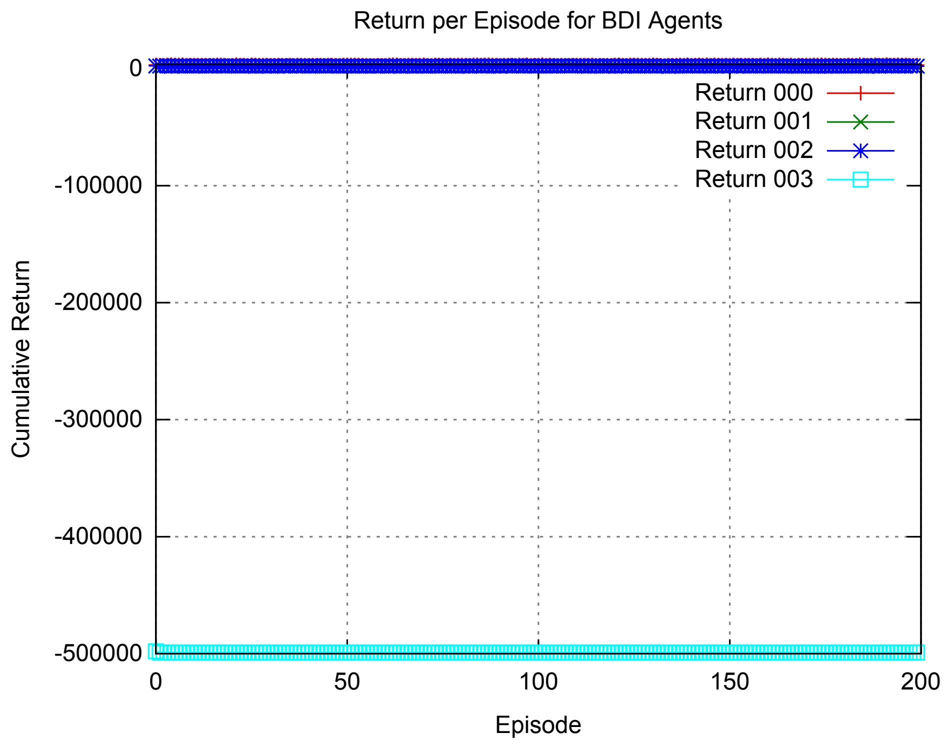

Figure 31. Returns for Random BDI Agent

7.8 Thesis Research Results

The use of RL as a way to to generate plans for BDI agents without relying on a-priori knowledge was **sucessfully** demonstrated.

The RL agents **explored and exploited** their environment resulting in policies that approximated optimal policies and performed much better than random walks. The BDI agents did not exist until they were generated by the **AgentGenerator** using the polices learned by the RL agents. Once the BDI agent's plans could be tested against the environment they were shown to perform successfully.

List of References

[1] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

CHAPTER 8

Discussion

This chapter discusses the contributions of this research, its limitations and future work that can expand and augment the work done as part of this thesis.

8.1 Contributions

The major contribution of this research is the demonstration of the **practical and successful use of RL**, a computational learning framework based on the idea of learning from repeated interactions with the environment, **to generate plans in BDI systems without relying on a-priori knowledge**.

The ability to learn and not follow the same plans over and over is arguably a key characteristic of intelligent systems. The results of this thesis provide another tool to augment BDI agents with learning capabilities. One of the benefits of using RL is that plans need not be fixed. RL can learn online and plans generated can change in response to changes in the environment.

8.2 Limitations

Even though the generation of BDI plans without relying on a-priori knowledge was successfully demonstrated, there remains a big limitation: how to express the BDI plans from learned RL policies. Human input is still necessary to decide the plans' expression or representation. There are arguably infinite ways to express an equivalent plan.

The situation is analogous to the case of writing a compiler for a higher level language, somebody needs to decide what is the target language that the source is being compiled into.

8.3 Future Work

The research conducted in this thesis can be improved in a number of ways.

1. Co-located RL learning and BDI plan generation (Online learning)

2. Apply to problems which require advanced RL techniques: Kanerva prototypes, function approximation, neural networks and others.

3. Focus on task decomposition and independent task learning to learn in more challenging state-spaces.

4. Automatic BDI plan generation by focusing on the grammar/architecture/organization of the plans. There are many ways that plans can be expressed. This can include Domain Specific Languages (DSLs) for combining RL and BDI planning.

APPENDIX

Generated BDI Agents

The generated AgentSpeak BDI agent code for each of the 3 configurations evaluated is show in the next 3 sections.

A.1 Configuration 000

```
// Agent bdi_agent000 generated from Q values

/* Initial beliefs and rules */

return(0).
calculation(done).

/* Initial goals */

/* Plans */

+state(0,0,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,0,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,0,0,0) : calculation(done)
<- -calculation(done);
```

```
   action(w);
   !calculateReturn.

+state(0,1,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,1,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,1,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,1,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,1,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,2,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,2,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,2,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,2,0,0) : calculation(done)
```

```
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,3,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,3,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,3,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,3,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,3,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,4,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,4,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.
```

```
+state(2,4,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,4,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,4,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,0,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,0,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,0,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,1,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.
```

```
+state(1,1,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,1,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,1,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,1,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,2,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,2,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,2,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,2,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,2,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,3,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,3,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,3,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,3,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,3,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,4,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,4,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,4,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,4,1,0) : calculation(done)
<- -calculation(done);
```

```
   action(n);
   !calculateReturn.

+state(4,4,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,0,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,0,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,1,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,1,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,1,2,0) : calculation(done)
```

```
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,1,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,1,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,2,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,2,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,2,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,2,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,3,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.
```

```
+state(1,3,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,3,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,3,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,3,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,4,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(1,4,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,4,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,4,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,4,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.
```

```
+state(0,0,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,0,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,0,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,1,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,1,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,1,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,1,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,1,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,2,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,2,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,2,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,2,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,2,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,3,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,3,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,3,3,0) : calculation(done)
<- -calculation(done);
```

```
   action(e);
   !calculateReturn.

+state(3,3,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,3,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,4,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,4,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,4,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,4,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,4,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(0,0,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,0,4,0) : calculation(done)
```

```
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,0,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,0,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,1,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,1,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,1,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,1,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,1,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.
```

```
+state(0,2,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,2,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,2,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,2,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,2,4,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,3,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,3,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,3,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,3,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.
```

```
+state(4,3,4,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,4,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,4,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,4,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,4,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,4,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(0,0,0,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,0,0,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,0,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,1,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,1,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,1,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,1,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,1,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,2,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,2,0,1) : calculation(done)
<- -calculation(done);
```

```
   action(w);
   !calculateReturn.

+state(2,2,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,2,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,3,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,3,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,3,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,3,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,3,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,4,0,1) : calculation(done)
```

```
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,4,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,4,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,4,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,4,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,0,1,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,1,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,0,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.
```

```
+state(4,0,1,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,1,1,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,1,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,1,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,1,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,1,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,2,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,2,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,2,1,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.
```

```
+state(3,2,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,2,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,3,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,3,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,3,1,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,3,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,3,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,4,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,4,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,4,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,4,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,4,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,0,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,2,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,0,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,0,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,1,2,1) : calculation(done)
<- -calculation(done);
```

```
   action(s);
   !calculateReturn.

+state(1,1,2,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,1,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,1,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,1,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,2,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,2,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,2,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,2,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,2,1) : calculation(done)
```

```
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,3,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,3,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,3,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,3,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,3,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,4,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(1,4,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,4,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.
```

```
+state(3,4,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,4,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,0,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,0,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,0,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,1,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,1,3,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.
```

```
+state(2,1,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,1,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,1,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,2,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,2,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,2,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,2,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,2,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,3,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,3,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,3,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,3,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,3,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,4,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,4,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,4,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,4,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,4,3,1) : calculation(done)
<- -calculation(done);
```

```
   action(s);
   !calculateReturn.

+state(0,0,4,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,0,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,0,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,0,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,1,4,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,1,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,1,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,1,4,1) : calculation(done)
```

```
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,1,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,2,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,2,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,2,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,2,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,3,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(1,3,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.
```

```
+state(2,3,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,3,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,3,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,4,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,4,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,4,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,4,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,4,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+!calculateReturn : not calculation(done)
<- ?reward(R);
   .print("Reward is: ", R);
   ?return(Rt);
   Return = Rt+R;
   -reward(R);
   -+return(Return);
   .print("Total return is: ", Return);
   +calculation(done);
   ?state(X,Y,P,S);
   -+state(X,Y,P,S).
```

A.2 Configuration 001

```
// Agent bdi_agent001 generated from Q values

/* Initial beliefs and rules */

return(0).
calculation(done).

/* Initial goals */

/* Plans */

+state(0,0,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,0,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,0,0,0) : calculation(done)
<- -calculation(done);
```

```
   action(w);
   !calculateReturn.

+state(0,1,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,1,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,1,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,1,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,1,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,2,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,2,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,2,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,2,0,0) : calculation(done)
```

```
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,3,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,3,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,3,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,3,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,3,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,4,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,4,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.
```

```
+state(2,4,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,4,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,4,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,0,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,0,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,0,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,1,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.
```

```
+state(1,1,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,1,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,1,1,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,1,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,2,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,2,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,2,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,2,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,2,1,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,3,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,3,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,3,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,3,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,3,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,4,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,4,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,4,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,4,1,0) : calculation(done)
<- -calculation(done);
```

```
   action(n);
   !calculateReturn.

+state(4,4,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,0,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,0,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,1,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,1,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,1,2,0) : calculation(done)
```

```
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,1,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,1,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,2,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,2,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,2,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,2,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,3,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.
```

```
+state(1,3,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,3,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,3,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,3,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,4,2,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,4,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,4,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,4,2,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,4,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.
```

```
+state(0,0,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,0,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,1,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,1,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,1,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,1,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,1,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,2,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,2,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,2,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,2,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,2,3,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,3,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,3,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,3,3,0) : calculation(done)
<- -calculation(done);
```

```
   action(e);
   !calculateReturn.

+state(3,3,3,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,3,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,4,3,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,4,3,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,4,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,4,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,4,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,0,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,0,4,0) : calculation(done)
```

```
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,0,4,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,0,4,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,0,4,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,1,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,1,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,1,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,1,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,1,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.
```

```
+state(0,2,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,2,4,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,2,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,2,4,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,3,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,3,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,3,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,3,4,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.
```

```
+state(4,3,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,4,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,4,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,4,4,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,4,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,4,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(0,0,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(1,0,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,0,0,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,0,0,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,0,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,1,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,1,0,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,1,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,1,0,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,1,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,2,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,2,0,1) : calculation(done)
<- -calculation(done);
```

```
   action(w);
   !calculateReturn.

+state(2,2,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,2,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,3,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,3,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,3,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,3,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,3,0,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,4,0,1) : calculation(done)
```

```
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,4,0,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,4,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,4,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,4,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,0,1,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,1,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,0,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,0,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.
```

```
+state(4,0,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(0,1,1,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,1,1,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,1,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,1,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,1,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,2,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,2,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,2,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.
```

```
+state(3,2,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,2,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,3,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,3,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,3,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,3,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,3,1,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,4,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,4,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,4,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,4,1,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,4,1,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,0,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,2,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,0,2,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,0,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,0,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,1,2,1) : calculation(done)
<- -calculation(done);
```

```
   action(s);
   !calculateReturn.

+state(1,1,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,1,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,1,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,1,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,2,2,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,2,2,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,2,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,2,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,2,1) : calculation(done)
```

```
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,3,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,3,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,3,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,3,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,3,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,4,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(1,4,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,4,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.
```

```
+state(3,4,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,4,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,0,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,0,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,0,3,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,1,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,1,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.
```

```
+state(2,1,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,1,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,1,3,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,2,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,2,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,2,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,2,3,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,3,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,3,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,3,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,3,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,3,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,4,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,4,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,4,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,4,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,4,3,1) : calculation(done)
<- -calculation(done);
```

```
   action(e);
   !calculateReturn.

+state(0,0,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(1,0,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,0,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,0,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,0,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,1,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(1,1,4,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,1,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,1,4,1) : calculation(done)
```

```
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,1,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,2,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,2,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,2,4,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,2,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,2,4,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(0,3,4,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,3,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.
```

```
+state(2,3,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,3,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,3,4,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(0,4,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,4,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,4,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,4,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,4,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+!calculateReturn : not calculation(done)
<- ?reward(R);
   .print("Reward is: ", R);
   ?return(Rt);
   Return = Rt+R;
   -reward(R);
   -+return(Return);
   .print("Total return is: ", Return);
   +calculation(done);
   ?state(X,Y,P,S);
   -+state(X,Y,P,S).
```

A.3 Configuration 002

```
// Agent bdi_agent002 generated from Q values

/* Initial beliefs and rules */

return(0).
calculation(done).

/* Initial goals */

/* Plans */

+state(0,0,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,0,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,0,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,1,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,1,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,1,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,1,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,1,0,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,2,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,2,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,2,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,2,0,0) : calculation(done)
<- -calculation(done);
```

```
   action(w);
   !calculateReturn.

+state(4,2,0,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(0,3,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(1,3,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,3,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,3,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,3,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,4,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,4,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,4,0,0) : calculation(done)
```

```
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,4,0,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,4,0,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,0,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,0,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(0,1,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.
```

```
+state(1,1,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,1,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,1,1,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,1,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,2,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,2,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,2,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,2,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,2,1,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.
```

```
+state(0,3,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,3,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,3,1,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,3,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,3,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,4,1,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,4,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,4,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,4,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,4,1,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,0,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,0,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,1,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,1,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,1,2,0) : calculation(done)
<- -calculation(done);
```

```
   action(w);
   !calculateReturn.

+state(3,1,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,1,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,2,2,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,2,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,2,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,2,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,3,2,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,3,2,0) : calculation(done)
```

```
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,3,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,3,2,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,3,2,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,4,2,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,4,2,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,4,2,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,4,2,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,4,2,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.
```

```
+state(0,0,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,0,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,1,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,1,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,1,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,1,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.
```

```
+state(4,1,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,2,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,2,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,2,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,2,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,2,3,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,3,3,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(1,3,3,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,3,3,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,3,3,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,3,3,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,4,3,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,4,3,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,4,3,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,4,3,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,4,3,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,0,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,4,0) : calculation(done)
<- -calculation(done);
```

```
   action(n);
   !calculateReturn.

+state(2,0,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,0,4,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,0,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,1,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,1,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,1,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,1,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,1,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(0,2,4,0) : calculation(done)
```

```
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(1,2,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,2,4,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,2,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,2,4,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,3,4,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(1,3,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,3,4,0) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,3,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.
```

```
+state(4,3,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,4,4,0) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,4,4,0) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,4,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,4,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,4,4,0) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,0,0,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,0,0,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,0,0,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.
```

```
+state(3,0,0,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,0,0,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,1,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,1,0,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,1,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,1,0,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,1,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,2,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,2,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,2,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,2,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,3,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,3,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,3,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,3,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,3,0,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,4,0,1) : calculation(done)
<- -calculation(done);
```

```
   action(e);
   !calculateReturn.

+state(1,4,0,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,4,0,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,4,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,4,0,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,0,1,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,1,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,0,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,0,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,0,1,1) : calculation(done)
```

```
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,1,1,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,1,1,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,1,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,1,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,1,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,2,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,2,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,2,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.
```

```
+state(3,2,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,2,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,3,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,3,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,3,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,3,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,3,1,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,4,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,4,1,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.
```

```
+state(2,4,1,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(3,4,1,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,4,1,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,0,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,0,2,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,0,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,0,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,1,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,1,2,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,1,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,1,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,1,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,2,2,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,2,2,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,2,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,2,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,2,1) : calculation(done)
<- -calculation(done);
```

```
   action(s);
   !calculateReturn.

+state(0,3,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,3,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,3,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,3,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,3,2,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,4,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(1,4,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,4,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,4,2,1) : calculation(done)
```

```
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,4,2,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,0,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,0,3,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,0,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,0,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,0,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,1,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,1,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.
```

```
+state(2,1,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,1,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,1,3,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,2,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,2,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,2,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,2,3,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,2,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,3,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.
```

```
+state(1,3,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,3,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,3,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,3,3,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,4,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,4,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,4,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,4,3,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(4,4,3,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,0,4,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,0,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(2,0,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(3,0,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,0,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(0,1,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(1,1,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,1,4,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,1,4,1) : calculation(done)
<- -calculation(done);
```

```
   action(n);
   !calculateReturn.

+state(4,1,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(0,2,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,2,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(2,2,4,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(3,2,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(4,2,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,3,4,1) : calculation(done)
<- -calculation(done);
   action(n);
   !calculateReturn.

+state(1,3,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(2,3,4,1) : calculation(done)
```

```
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,3,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(4,3,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(0,4,4,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(1,4,4,1) : calculation(done)
<- -calculation(done);
   action(e);
   !calculateReturn.

+state(2,4,4,1) : calculation(done)
<- -calculation(done);
   action(s);
   !calculateReturn.

+state(3,4,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+state(4,4,4,1) : calculation(done)
<- -calculation(done);
   action(w);
   !calculateReturn.

+!calculateReturn : not calculation(done)
<- ?reward(R);
   .print("Reward is: ", R);
   ?return(Rt);
   Return = Rt+R;
```

```
-reward(R);
-+return(Return);
.print("Total return is: ", Return);
+calculation(done);
?state(X,Y,P,S);
-+state(X,Y,P,S).
```

BIBLIOGRAPHY

Andre, D., Friedman, N., and Parr, R., "Generalized prioritized sweeping," in *Advances in Neural Information Processing Systems.* MIT Press, 1998.

Baldassarre, G., "Forward and bidirectional planning based on reinforcement learning and neural networks in a simulated robot," in *ABiALS*, 2003, pp. 179–200.

Bauer, T., Erwig, M., Fern, A., and Pinto, J., "Adaptation-based programming in haskell," in *DSL*, 2011, pp. 1–23.

Bertoli, P. and Cimatti, A., "Improving heuristics for planning as search in belief space," in *AIPS*, 2002, pp. 143–152.

Bordini, R. H. and Hübner, J. F., "Bdi agent programming in agentspeak using jason," in *IN: PROCEEDINGS OF 6TH INTERNATIONAL WORKSHOP ON COMPUTATIONAL LOGIC IN MULTI-AGENT SYSTEMS (CLIMA VI). VOLUME 3900 OF LNCS.* Springer, 2005, pp. 143–164.

Bordini, R. H., Hübner, J. F., and Wooldridge, M. J., *Programming Multi-Agent Systems in AgentSpeak Using Jason.* J. Wiley, 2007. [Online]. Available: http://www.worldcat.org/isbn/9780470029008

Bordini, R. H. and Hübner, J. F., "Bdi agent programming in agentspeak using *jason* (tutorial paper)," in *CLIMA*, 2005, pp. 143–164.

Bordini, R. H., Hübner, J. F., and Vieira, R., "Jason and the golden fleece of agent-oriented programming," in *Multi-Agent Programming*, 2005, pp. 3–37.

Buford, J., Jakobson, G., and Lewis, L., "Extending bdi multi-agent systems with situation management," *2006 9th International Conference on Information Fusion*, pp. 1–7, 2006. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4086067

Cao, H., Xi, H., and Smith, S. F., "Supply chain planning: A reinforcement learning approach to production planning in the fabrication/fulfillment manufacturing process," in *Winter Simulation Conference*, 2003, pp. 1417–1423.

de Silva, L., Sardina, S., and Padgham, L., "First principles planning in bdi systems," in *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, ser. AAMAS '09. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 1105–1112. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558109.1558167

Dearden, R., “Structured prioritised sweeping,” in *ICML*, 2001, pp. 82–89.

Dietterich, T. G. and Flann, N. S., “Explanation-based learning and reinforcement learning: A unified view,” in *Machine Learning*, 1995, pp. 176–184.

d’Inverno, M., Kinny, D., Luck, M., and Wooldridge, M., “A formal specification of dmars,” in *Proceedings of the 4th International Workshop on Intelligent Agents IV, Agent Theories, Architectures, and Languages*, ser. ATAL ’97. London, UK, UK: Springer-Verlag, 1998, pp. 155–176. [Online]. Available: http://dl.acm.org/citation.cfm?id=648204.749438

Dixon, K. R., Malak, R. J., and Khosla, P. K., “Incorporating prior knowledge and previously learned information into reinforcement learning agents,” Carnegie Mellon University, Tech. Rep., 2000.

Gal, Y., Reddy, S., Shieber, S. M., Rubin, A., and Grosz, B. J., “Plan recognition in exploratory domains,” *Artif. Intell.*, vol. 176, no. 1, pp. 2270–2290, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2011.09.002

Garcia-Martinez, R. and Borrajo, D., “An integrated approach of learning, planning, and execution,” *Journal of Intelligent and Robotic Systems*, vol. 29, pp. 47–78, 2000.

Georgeff, M. P. and Lansky, A. L., “Reactive reasoning and planning,” in *AAAI*, 1987, pp. 677–682.

Gosavi, A., “Reinforcement learning: A tutorial survey and recent advances,” *INFORMS Journal on Computing*, vol. 21, no. 2, pp. 178–192, 2009.

Grounds, M. and Kudenko, D., “Combining reinforcement learning with symbolic planning,” in *Proceedings of the 5th , 6th and 7th European conference on Adaptive and learning agents and multi-agent systems: adaptation and multi-agent learning*, ser. ALAMAS’05/ALAMAS’06/ALAMAS’07. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 75–86. [Online]. Available: http://dl.acm.org/citation.cfm?id=1898681.1898687

Guerra-Hernandez, A., El Fallah-Seghrouchni, A., and Soldano, H., “Distributed learning in intentional bdi multi-agent systems,” in *Proceedings of the Fifth Mexican International Conference in Computer Science*, ser. ENC ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 225–232. [Online]. Available: http://dl.acm.org/citation.cfm?id=1025121.1025749

Guerra-Hernández, A., Fallah-Seghrouchni, A. E., and Soldano, H., “Learning in bdi multi-agent systems,” in *CLIMA*, 2004, pp. 218–233.

Hübner, J. F., Bordini, R. H., and Wooldridge, M., “Programming declarative goals using plan patterns,” in *DALT*, 2006, pp. 123–140.

ji Lin, L., "Self-improving reactive agents based on reinforcement learning, planning and teaching," in *Machine Learning*, 1992, pp. 293–321.

Kaelbling, L. P., Littman, M. L., and Moore, A. W., "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

Karim, S., Sonenberg, L., and Tan, A.-H., "A hybrid architecture combining reactive plan execution and reactive learning," in *Proceedings of the 9th Pacific Rim international conference on Artificial intelligence*, ser. PRICAI'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 200–211. [Online]. Available: http://dl.acm.org/citation.cfm?id=1757898.1757924

Karim, S., Subagdja, B., and Sonenberg, L., "Plans as products of learning," in *IAT*, 2006, pp. 139–145.

Kemke, C. and Walker, E., "Planning with action abstraction and plan decomposition hierarchies," in *Proceedings of the IEEE/WIC/ACM international conference on Intelligent Agent Technology*, ser. IAT '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 447–451. [Online]. Available: http://dx.doi.org/10.1109/IAT.2006.99

Kreuz, I. and Roller, D., "Relevant knowledge first - reinforcement learning and forgetting in knowledge based configuration," *CoRR*, vol. cs.AI/0109034, 2001.

Kumar, D. and Meeden, L., "A hybrid connectionist and bdi architecture for modeling embedded rational agents," in *In Cognitive Robotics: Papers from the 1998 AAAI Fall Symposium, Technical Report FS98 -02. Menlo Park.* AAAI Press, 1998, pp. 84–90.

Lokuge, P. and Alahakoon, D., "An extended bdi agent architecture with multiple intention reconsideration ability in a vessel berthing application," in *FLAIRS Conference*, 2005, pp. 333–338.

Matsui, T., Inuzuka, N., and Seki, H., "Learning preconditions for control policies in reinforcement learning," 2001.

Meneguzzi, F. R., Zorzo, A. F., da Costa Móra, M., and Luck, M., "Incorporating planning into bdi systems," *Scalable Computing: Practice and Experience*, vol. 8, no. 1, 2007.

Moore, A. W. and Atkeson, C. G., "Prioritized sweeping: Reinforcement learning with less data and less time," in *Machine Learning*, 1993, pp. 103–130.

Nair, R. and Tambe, M., "Hybrid bdi-pomdp framework for multiagent teaming," *J. Artif. Int. Res.*, vol. 23, no. 1, pp. 367–420, Apr. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1622503.1622512

Nason, S. and Laird, J. E., “Soar-rl: Integrating reinforcement learning with soar,” in *Cognitive Systems Research*, 2004, pp. 51–59.

Nilsson, N. J., “Teleo-reactive programs for agent control,” *J. Artif. Intell. Res. (JAIR)*, vol. 1, pp. 139–158, 1994.

Nunes, L. and Oliveira, E., “On learning by exchanging advice,” *CoRR*, vol. cs.LG/0203010, 2002.

Partalas, I., Vrakas, D., and Vlahavas, I., “Reinforcement learning and automated planning: A survey,” in *Advanced Problem Solving Techniques*, Vrakas, D. and Vlahavas, I., Eds. IGI Global, 2008.

Peng, J. and Williams, R. J., “Efficient learning and planning within the dyna framework,” in *Adaptive Behavior*, 1993, pp. 437–454.

Pereira, D. R. and Dimuro, G. P., “Um algoritmo para extração de um plano bdi que obedece uma política Ótima,” in *Workshop-Escola de Sistemas de Agentes para Ambientes Colaborativos, Pelotas, Anais do WESAAC 2007*, 2007.

Pereira, D. R., Vargas, L., Dimuro, G. P., and Costa, A. C. R., “Constructing bdi plans from optimal pomdp policies, with an application to agentspeak programming,” in *XXXIV - Conferencia Latinoamericana de Informtica, 2008, Santa Fe. Proceedings of CLEI 2008 (Lecture Notes in Computer Science)*, ser. CLEI 2008, vol. 1. Springer, 2008, pp. 1–10.

Phung, T., Winikoff, M., and Padgham, L., “Learning within the bdi framework: An empirical analysis,” in *Knowledge-Based Intelligent Information and Engineering Systems, Part III.* Springer-Verlag, 2005, pp. 282–288.

Poole, D. and Mackworth, A., “Artificial intelligence: Foundations of computational agents,” 2010. [Online]. Available: http://artint.info/html/ArtInt_262.html#davids-simple-game-ex

Rao, A. S., “Agentspeak(l): Bdi agents speak out in a logical computable language,” in *MAAMAW*, 1996, pp. 42–55.

Rao, A. S. and Georgeff, M. P., “Modeling rational agents within a bdi-architecture,” in *KR*, 1991, pp. 473–484.

Rao, A. S. and Georgeff, M. P., “Bdi agents: From theory to practice,” in *ICMAS*, 1995, pp. 312–319.

Russell, S. and Norvig, P., *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall, 2003.

Ryan, M. R. K., “Using abstract models of behaviours to automatically generate reinforcement learning hierarchies,” in *In Proceedings of The 19th International Conference on Machine Learning.* Morgan Kaufmann, 2002, pp. 522–529.

Ryan, M. R. and Pendrith, M. D., “Rl-tops: An architecture for modularity and reuse in reinforcement learning,” in *In Proceedings of the Fifteenth International Conference on Machine Learning.* Morgan Kaufmann, 1998, pp. 481–487.

Saad, E., “Bridging the gap between reinforcement learning and knowledge representation: A logical off- and on-policy framework,” in *ECSQARU*, 2011, pp. 472–484.

Singh, D., Sardina, S., and Padgham, L., “Extending bdi plan selection to incorporate learning from experience,” *Robot. Auton. Syst.*, vol. 58, no. 9, pp. 1067–1075, Sept. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.robot.2010.05.008

Singh, D., Sardiña, S., Padgham, L., and Airiau, S., “Learning context conditions for bdi plan selection,” in *AAMAS*, 2010, pp. 325–332.

Subagdja, B. and Tan, A.-H., “A self-organizing neural network architecture for intentional planning agents,” in *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, ser. AAMAS ’09. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 1081–1088. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558109.1558164

Sun, R., Peterson, T., and Sessions, C., “Beyond simple rule extraction: Acquiring planning knowledge from neural networks.” Berlin Heidelberg: Springer-Verlag, 2001.

Sun, R. and Sessions, C., “Learning plans without a priori knowledge,” in *in Proceedings of WCCI-IJCNN’98.* IEEE Press, 2000, pp. 1–6.

Sutton, R. S., “Integrated architectures for learning, planning, and reacting based on approximating dynamic programming,” in *In Proceedings of the Seventh International Conference on Machine Learning.* Morgan Kaufmann, 1990, pp. 216–224.

Sutton, R. S., “Dyna, an integrated architecture for learning, planning, and reacting,” *SIGART Bulletin*, vol. 2, no. 4, pp. 160–163, 1991.

Sutton, R. S., “Planning by incremental dynamic programming,” in *In Proceedings of the Eighth International Workshop on Machine Learning.* Morgan Kaufmann, 1991, pp. 353–357.

Sutton, R. S., “Reinforcement learning architectures,” in *Proceedings ISKIT’92 International Symposium on Neural Information Processing*, 1992.

Sutton, R. S. and Barto, A. G., *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

Tan, A.-H., “Falcon: A fusion architecture for learning, cognition, and navigation,” in *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, vol. 4, july 2004, pp. 3297 –3302 vol.4.

Tan, A.-H., “Self-organizing neural architecture for reinforcement learning,” in *Proceedings of the Third international conference on Advances in Neural Networks - Volume Part I*, ser. ISNN’06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 470–475. [Online]. Available: http://dx.doi.org/10.1007/11759966_70

Thangarajah, J., Harland, J., Morley, D., and Yorke-Smith, N., “Aborting tasks in bdi agents,” in *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, ser. AAMAS ’07. New York, NY, USA: ACM, 2007, pp. 6:1–6:8. [Online]. Available: http://doi.acm.org/10.1145/1329125.1329133

Vieira, R., Moreira, Á. F., Bordini, R. H., and Hübner, J. F., “An agent-oriented programming language for computing in context,” in *IFIP PPAI*, 2006, pp. 61–70.

Walczak, A., Braubach, L., Pokahr, A., and Lamersdorf, W., “Augmenting bdi agents with deliberative planning techniques,” in *Proceedings of the 4th international conference on Programming multi-agent systems*, ser. ProMAS’06. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 113–127. [Online]. Available: http://dl.acm.org/citation.cfm?id=1759324.1759334

Walsh, T. J., Goschin, S., and Littman, M. L., “Integrating sample-based planning and model-based reinforcement learning,” in *AAAI*, 2010.

Watkins, C. J. C. H. and Dayan, P., “Technical note q-learning,” *Machine Learning*, vol. 8, pp. 279–292, 1992.

Watkins, C. J. C. H., “Learning from delayed rewards,” Ph.D. dissertation, King’s College, Cambridge, UK, May 1989. [Online]. Available: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf

Weiß, G., “An architectural framework for integrated multiagent planning, reacting, and learning,” in *Proceedings of the 7th International Workshop on Intelligent Agents VII. Agent Theories Architectures and Languages*, ser. ATAL ’00. London, UK, UK: Springer-Verlag, 2001, pp. 320–330. [Online]. Available: http://dl.acm.org/citation.cfm?id=648207.749621

Wiedermann, J., “A high level model of a conscious embodied agent,” *Cognitive Informatics, IEEE International Conference on*, vol. 0, pp. 448–456, 2009.

Zhao, G., Tatsumi, S., and Sun, R., “Rtp-q: A reinforcement learning system with time constraints exploration planning for accelerating the learning rate,” *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 82, no. 10, pp. 2266–2273, 1999. [Online]. Available: http://ci.nii.ac.jp/naid/110003208162/en/

Zimmerman, T. and Kambhampati, S., "Learning-assisted automated planning: Looking back, taking stock, going forward," *AI Mag.*, vol. 24, no. 2, pp. 73–96, June 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=960150.960160