

Reinforcement Learning and the Parallel Actions Common Goal problem

Radboud University Nijmegen
Computer Science Bachelor Thesis

Author: Ivan Koster
Student number: 0814903
Course number: IBI009 (9ec)

Supervisors:
Dr. ir. M. van Otterlo
Dr. I.G. Sprinkhuizen-Kuyper
Dr. P.C. Groot

June 14, 2012

Abstract

There exists an interesting category of problems, in particular in real-time strategy and colonization games. The player can perform several actions, usually in parallel, like gathering resources and building structures, with a single goal in mind. He has to consider which actions to take and how to schedule them, in order to reach the goal in the fastest way possible. We call these problems *parallel actions common goal* (PACG) problems.

In this thesis we will try to solve these problems with reinforcement learning. To do this, we show that it is possible to capture PACG problems in a Markov Decision Processes (MDP). An MDP can be translated into a programming language, resulting in a simulator. Next, the simulator can be used as the environment for a reinforcement learning agent. We will focus on one complex PACG problem in particular, the browser game Ogame.

After that, we will show that, if implemented in a tabular form, Q-learning can consistently learn the optimal policy in simple PACG problems. This will be done using a simplified version of the Ogame problem. Following this, we will introduce a Q-learning algorithm for more complex PACG problems, which includes Artificial Neural Networks and Experience Replay. We will then test this algorithm on the simplified Ogame problem to show that it also learns optimal policies. At last, we will try to use this algorithm to learn good policies for the complex Ogame problem.

Contents

1. Introduction	1
1.1. Research goal	1
1.1.1. Research products	2
1.1.2. Motivations for this research	2
1.2. Short introduction to Ogame	2
1.2.1. Limiting the problem: explicit assumptions	4
1.3. Structure of this thesis	4
1.4. Theoretical framework	4
1.4.1. Short introduction to Markov Decision Processes	5
1.4.2. Short introduction to reinforcement learning	5
2. Modelling the problem	7
2.1. Toy MDP	7
2.1.1. States	8
2.1.2. Actions	8
2.1.3. Transition function	9
2.1.4. Reward function	10
2.1.5. Transition table	10
2.2. Improving the toy MDP	11
2.2.1. Afterstates	11
2.2.2. Starvation and spamming no_op to gain resources	12
2.2.3. Toy MDP version 2	14
2.3. Ogame MDP	15
2.3.1. From toy MDP to Ogame MDP	15
2.3.2. States	15
2.3.3. Actions	16
2.3.4. Transition function	23
2.3.5. Reward function	25
2.4. Research product one	25
3. Building the environment	26
3.1. State class	27
3.2. Action class	27
3.3. OgameFormulas class	28
3.4. Simulator class	29
3.5. UserTerminal class	29
3.6. Computation time tweaks	29
3.7. A note on Semi Markov Decision Processes	31
3.8. Simulator robustness	32
3.9. Research product two	32

4. Building the learning agent	33
4.1. Q-learning	33
4.2. Exploration policy	34
4.3. The value function and generalization	35
4.3.1. Artificial neural networks	35
4.4. Experience replay	38
4.5. Research product three	38
5. Experiments and results	39
5.1. Defining a good policy	39
5.2. Toy MDP experiments	39
5.2.1. Value iteration experiment	39
5.2.2. TD(0)-learning versus Q-learning experiment	40
5.2.3. Learning the optimal policy with tabular Q-learning	41
5.2.4. Full learning algorithm on toy simulator	45
5.3. Ogame MDP experiments	52
6. Thesis conclusion	54
Bibliography	55
Appendices	56
A. Actions in Ogame	56
A.1. List of actions included in the MDP	60
B. Toy model additions	61
B.1. Functions for toy model actions	61
B.2. Transition tables	62
B.3. Toy MDP version 2	65
B.3.1. States	65
B.3.2. Actions	66
B.3.3. Transition function	68
B.3.4. Reward function	69
C. Ogame MDP functions	70
D. Source code	76

1. Introduction

Imagine a game in which we can build several buildings that have certain resource costs. When choosing to build one, the resources are spent and a timer starts. When the timer runs out, the building is finished. Continuously choosing to build the same building upgrades its level by one at a time, increasing the resource costs and build timer, but also the effectiveness of the building. We can build only one building at the same time. Let us call this our building assembly line. Now imagine we have another assembly line to research certain technologies. The research assembly line behaves similar to the building one. These assembly lines can perform actions in parallel, but they share the same resource pool. Some actions for these assembly lines might have preconditions. For example, some buildings can only be built if we have researched certain technologies to an adequate level. Finally, we have a certain goal to reach. For example, we want to upgrade a certain building or technology to level ten. The faster we reach this goal state, the better. The assembly lines have to work together to reach this goal by sharing resources and meeting preconditions. In the game we have described here, the player tries to solve a problem which we call the *parallel actions common goal* (PACG) problem. This problem is core to exciting games, an example is Ogame.

This thesis is about trying to shed light on how to solve these problems, using Ogame and reinforcement learning (RL). Games like Ogame are usually too complex to bruteforce the answer. This is where reinforcement learning comes in: we are going to use learning algorithms to learn to play Ogame. These algorithms learn a policy: a function that maps the current game state to the best action for that particular state. For Ogame, a policy becomes better the faster it reaches the goal state.

For humans, Ogame is relatively easy to learn to a certain degree, by using simple but effective policies. Finding the optimal policy is hard though: the huge amount of actions and therefore states quickly overwhelm the human mind. We wonder if reinforcement learning can find a good or even the optimal policy.

1.1. Research goal

The main question this research seeks to answer is:

Can reinforcement learning find a good policy for playing Ogame?

Ogame is a text based browser game and the PACG problem that will be investigated for this thesis. An introduction to Ogame will follow shortly in Section 1.2. See Section 5.1 for the definition of a good policy. To find a good policy, we use a reinforcement learning algorithm. There are all kinds of these algorithms, but we will use Q-learning. It is probably the most known reinforcement learning algorithm. It has several advantages and disadvantages. See Section 4.1 for more information about Q-learning.

1. Introduction

To answer the main research question, the research is split into three smaller sub questions:

1. Can we create a model of Ogame, such that we can use it in reinforcement learning?
2. Can we build a piece of software that represents that model, so that we can run reinforcement learning algorithms on it?
3. Can we implement Q-learning in such a way that it can learn a policy for playing Ogame?

1.1.1. Research products

The three sub questions will lead to three products:

1. A model that represents Ogame and can be used in reinforcement learning.
2. A simulator based on the model.
3. An implementation of Q-learning, possibly including techniques to enhance learning performance.

In the end we will have a piece of software with a simulator of Ogame and an implementation of Q-learning. Together they will be used to answer the main research question. The source code for the software can be found in Appendix D.

1.1.2. Motivations for this research

This thesis exists, because there has not been a lot of research on this particular PACG problem. It will try to provide insight in how to tackle it. Another reason is that reinforcement learning is a very exciting topic. One of the best examples is the TD-Gammon algorithm of Tesauro (1994). It is an algorithm that learns to play backgammon. It performed so well, that it is at least equal to the best human backgammon player. Backgammon players worldwide also use a new unconventional opening strategy that TD-Gammon premiered.

For a third reason: this thesis also has a personal point to it. In my youth I have played the game that will be used for this research, Ogame. Since then I always asked myself if a computer could learn to play this game better.

1.2. Short introduction to Ogame

Ogame is a text-based strategy game with a space theme. It was launched in 2000 and currently has around two million players. The game is made and maintained by Gameforge AG.¹

In Ogame a player starts with an undeveloped planet, the Homeworld. The player can develop this planet and eventually colonize other planets. He/she can use three types of resources for this: metal, crystal and deuterium. Resources are produced by production buildings: mines for metal and crystal and a synthesizer for the deuterium. These production buildings need to be powered with energy, which can be produced by a

¹For more information, visit: <http://www.ogame.org> and <http://www.gameforge.com>

1. Introduction



Figure 1.1.: The building assembly line of Ogame

solar plant. If there is not enough energy, the production rate of resources suffers. There are also other utility buildings, like a research lab, where research can be conducted, or the shipyard, where spaceships can be built. For an impression of the building assembly line of Ogame, see Figure 1.1.

The universe of Ogame is organized into galaxies, systems and planets. A universe is divided into ten galaxies, each galaxy has 499 systems and each system has fifteen planet slots. A planet slot can contain a planet, debris field and a moon. Travelling within a system is fast, between systems is slow and between galaxies even slower. A new player gets appointed a new Homeworld in a random galaxy and system. He can build mines and other buildings until the planet is “full”.

Fleets of ships can be sent to planet slots. Space battles occur when two fleets meet each other. The players themselves have no influence over the battle once it starts. If the attacker wins, he steals the resources of the defender. Debris of the ships that got destroyed will go to the debris field of the planet slot, which can be harvested with special ships, by any player.

1. Introduction

There are many other aspects to Ogame, like alliances and expeditions, but these will be left aside in this thesis.

1.2.1. Limiting the problem: explicit assumptions

The main research question says we are looking for a good policy to play Ogame. The game has a large number of actions and essentially no end. This leads to a fast growing and infinite state space.¹ Because of this we will use some restrictions in this thesis. We will define the goal state as the moment when the player owns one Colony Ship. Upon reaching the goal, the player gets a large reward. Building a Colony Ship is a key moment for a player's career, from which they can continue with different strategies. This restriction limits the problem, but keeps it from becoming trivial: this goal state is still significantly deep² in the state space.

To keep the model from becoming incredibly complex, we will also omit the players other than the learning agent. In our model the learning agent will be the only player in the universe. For the proposed goal state this is not unthinkable: it is quite possible a player does not interact with other players before building his/her colony ship.

We will also omit all the actions that do not contribute to getting to the goal state or complicates the problem considerably. Appendix A explains why certain actions are omitted.

Future research could improve upon this thesis by including the omitted actions or changing the goal state. It could also remove the goal state all together and use a different reward model that suits the never ending nature of Ogame better.

Note that we are using Ogame version 3.0.1, which was the live version when this thesis was written.

1.3. Structure of this thesis

We will continue this chapter with the introduction of the theoretical framework. Next, we will model the problem in Chapter 2 and create several models. Then the models will be transferred to a software implementation in Chapter 3. Chapter 4 will introduce Q-learning and several techniques to enhance its performance and then propose a learning algorithm. At last, Chapter 5 will experiment with the learning algorithm and show the results.

1.4. Theoretical framework

To solve our Ogame problem, we will model it as a Markov Decision Process (MDP) and then apply reinforcement learning (RL). RL is the method that will eventually produce the policy we are looking for. RL algorithms are specifically designed to solve MDPs. MDPs are “an intuitive and fundamental formalism for decision-theoretic planning, reinforcement learning and other learning problems in stochastic domains” Wiering & Otterlo (2012)[Section 1.1]. Section 1.4.1 will introduce MDPs and Section 1.4.2 will explain reinforcement learning.

¹A state is a particular setting of the game “board”. The state space of a game is the set of all these states.

²A state deep in the state space is interpreted as: it takes “many” transitions to reach this state from the starting state.

1.4.1. Short introduction to Markov Decision Processes

Some of the earliest research on MDPs was done by Bellman (1957). A more recent approach can be found in Puterman (1994) and Wiering & Otterlo (2012). An MDP models an environment (Ogame) into a set of states and actions. A state contains information about what the game world looks like, for example how much resources a player has. The set of actions contain the actions the player can do in the environment. Performing these actions makes changes to the environment, causing it to transition between states. Performing actions also gives a reward. The goal of an MDP is to perform actions in such a way (following a policy) that it maximizes these rewards. If the states contain enough information to decide which actions to take in order to maximize the rewards, and we do not need information from previous states, then the process has the Markov property.

Formally, a standard MDP consists of the following:

- A finite set of states S .
- A finite set of actions A .
- The transition function $P_a(s, s') = Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}$, which gives the probability for when action a is chosen in state s at time t , it leads to state s' at time $t + 1$.
- The reward function $R_a(s, s') = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\}$, which gives the *expected immediate reward* if choosing action a in state s leads to state s' .

In order to make an MDP model for our Ogame problem, we only have to define these four elements, which we will do in the coming chapter. We will however use a slight deviation of the standard MDP:

- The set of states S will be infinite, since Ogame has no end.
- Besides the set of actions A , we will introduce the set of actions A_s , which includes all actions available in state s . This is necessary, because actions in Ogame can have preconditions.

For more information on MDPs, we refer to Wiering & Otterlo (2012)[Section 1.3.1].

1.4.2. Short introduction to reinforcement learning

The book of Sutton & Barto (1998) is the leading resource on reinforcement learning. This is their introduction: “Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning.”

An agent that uses reinforcement learning has three basic aspects: sense, actions and goal(s). The sense aspect is the way the agent perceives the state of the environment (Ogame). This is given to the agent via a state signal. The agent can perform actions to affect the state and to eventually reach the goal(s).

Reinforcement learning is not supervised learning, which utilizes an all or partially knowing supervisor. Reinforcement learning can be thought of as a journey to uncharted

1. Introduction

areas: the agent knows nothing and has to explore the environment to learn what is there. While learning, the agent has to balance exploring and exploiting. To maximize the reward, it has to exploit actions with good rewards, but it also has to occasionally explore to find these good actions. It is not possible to focus on only one of these two aspects and maximize the reward signal at the same time.

For more information about reinforcement learning, we refer to Sutton & Barto (1998).

Elements of reinforcement learning

The following section is a paraphrase of Sutton & Barto (1998).

A reinforcement learning system has two main elements: the learning agent and the environment, see Figure 1.2 for a graphical impression. In our case, the environment

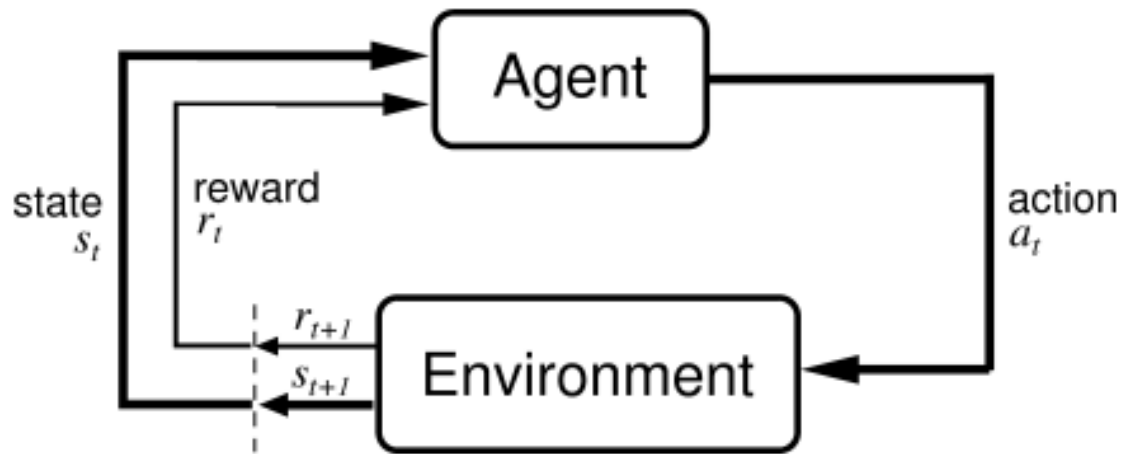


Figure 1.2.: A graphical impression of reinforcement learning

will be a piece of simulation software. This simulator will be based on the MDP model that we will present in the next chapter. The agent is the algorithm, or combination of algorithms, that will try to learn from the environment. It has three sub elements for this: an exploration policy, a reward function and a value function. A policy function is often denoted as π . It is a function which maps every state $s \in S$ to an action $a \in A$ or $a \in A_s$. The agent uses it to pick actions in the learning process.

The reward function is the same as in the MDP. It is a mapping of states to a numerical reward. The agent will try to maximize this reward, thus the reward function inherently defines the goal of the reinforcement learning problem. While the reward function gives the agent an immediate sense of how good a certain state is, the value function gives the agent a sense of what is good in the long run. The value of a state is the combination of the immediate reward plus the rewards the agent can accumulate in the future, starting from this state. The purpose of a reinforcement learning algorithm is to learn this value function. It is usually initialized with an arbitrary value. The learning process consists of taking actions and observing the rewards. The resulting experience is then used to update the estimated value function to a better approximation of the true value function, which it should eventually converge to. The resulting value function can be used to extract the resulting policy. This policy will be used to measure the performance of the agent and to answer our research question.

2. Modelling the problem

Creating an MDP for our Ogame problem from scratch turns out to be though. Ogame is a complex game, with actions that have several preconditions and influence other actions and assembly lines. It is also not immediately clear how the reward function should work when actions are running in parallel. Modelling an MDP for Ogame can be easier if we create a toy MDP first: a model of a simplified Ogame.

The simplified version of Ogame will feature only one kind of resource and two assembly lines, with each one action. When the toy model is finished, it can be scaled up to create the model for the Ogame problem. Another advantage is that we have a simple and small model which we can experiment upon with reinforcement learning. This can give us valuable insight for the later stages of this thesis.

The toy MDP is presented in Section 2.1, while an improved version, toy MDP version 2, is presented in Section 2.2. Finally, Section 2.3 shows the Ogame MDP.

2.1. Toy MDP

To define an MDP of the toy problem, recall that we only have to define the following elements:

- A set of states S .
- A finite set of actions A and A_s for each state $s \in S$.
- The transition function.
- The reward function.

We will define these in the next four sections. In these definitions we will use the notations and functions as summarized in Table 2.1. The complete definition for the

Notation / function	
$s_{variablename}$	Represents the value of the variable <i>variablename</i> in state s .
$(s, a) \rightarrow s'[var \leftarrow value]$	A state transition, where action a is executed in state s . This leads to state s' where the value of var is replaced with $value$.
<code>pint <i>variablename</i></code>	A type indicator, this variable is a positive integer: 0, 1, 2, 3, ...
<code>cost(<i>actionname</i>, <i>level</i>)</code>	Returns the cost of the action <i>actionname</i> for the level <i>level</i> .
<code>time(<i>actionname</i>, <i>level</i>)</code>	Returns the duration (building or research) time of the action <i>actionname</i> for the level <i>level</i> .
<code>prod(<i>actionname</i>, <i>level</i>)</code>	Returns the resource production of the building <i>actionname</i> for the level <i>level</i> .

Table 2.1.: Notations and functions for the toy MDP

2. Modelling the problem

cost, time and prod functions can be found in Section B.1 in the appendix.

2.1.1. States

A state is determined by the values of the available metal, levels of the MetalMine and LaserTechnology and timers for the assembly lines. Table 2.2 contains a legend for these variables. If an assembly line is inactive, the corresponding timer value should be zero. If two actions are running in parallel, the state should reflect that by having the two timer values above zero. Any time a building or research action finishes, the corresponding level value is raised with one.

Summarizing this, the set of states S is defined as:

$$s \in S \text{ if } s = \begin{matrix} < \text{pint res_metal,} \\ \text{pint lvl_MetalMine,} & \text{pint lvl_LaserTechnology,} \\ \text{pint timer_build,} & \text{pint timer_research} > \end{matrix}$$

res_metal	The amount of metal resource owned.
lvl_MetalMine	The current level of the Metal Mine building.
lvl_LaserTechnology	The current level of the Laser Technology research.
timer_build	Building time remaining in seconds.
timer_research	Research time remaining in seconds.

Table 2.2.: Legend for Toy MDP state variables

The starting state s_0 , representing a new Ogame account, is given by:

$$s_0 = \langle 0, 0, 0, 0, 0 \rangle$$

2.1.2. Actions

The set of actions A is defined as:

$$A = \{\text{MetalMine}, \text{LaserTechnology}, \text{no_op}\}$$

no_op is an action an agent can take if he wants to perform a time step in the game.

Preconditions

The set of actions A_s is defined as:

$$\begin{aligned} A_s &\subseteq A \\ \text{MetalMine} \in A_s &\iff s_{\text{timer_build}} = 0 \wedge \text{cost}(\text{MetalMine}, s_{\text{lvl_MetalMine}} + 1) \leq s_{\text{res_metal}} \\ \text{LaserTechnology} \in A_s &\iff s_{\text{timer_research}} = 0 \wedge \\ &\quad \text{cost}(\text{LaserTechnology}, s_{\text{lvl_LaserTechnology}} + 1) \leq s_{\text{res_metal}} \\ \text{no_op} &\in A_s \end{aligned}$$

Note that we write $\text{cost}(\text{MetalMine}, s_{\text{lvl_MetalMine}} + 1)$ instead of $\text{cost}(\text{MetalMine}, s_{\text{lvl_MetalMine}})$, because we want to know if we have enough resources to upgrade the current MetalMine. The cost function returns values for the current level, not for upgrades.

2. Modelling the problem

These predonditions can be explained as: you can only build or upgrade the Metal Mine if you own the necessary amount of resources and are not building anything else. The same goes for Laser Technology. The no_op action is always available, without it, the agent cannot progress in time.

Effects

If the action MetalMine is applied in state s , it results in a new state s' , in which:

- The amount of metal we have is reduced by the metal cost of the action.
- The timer_build is set to the time it takes to build the MetalMine.

Formally:

$$(s, \text{MetalMine}) \rightarrow s' [\text{res_metal} \leftarrow s_{\text{res_metal}} - \text{cost}(\text{MetalMine}, s_{\text{lvl_MetalMine}} + 1)] \\ [\text{timer_build} \leftarrow \text{time}(\text{MetalMine}, s_{\text{lvl_MetalMine}} + 1)]$$

Analogously the action LaserTechnology is defined as:

$$(s, \text{LaserTechnology}) \rightarrow s' [\text{res_metal} \leftarrow s_{\text{res_metal}} - \text{cost}(\text{LaserTechnology}, s_{\text{lvl_LaserTechnology}} + 1)] \\ [\text{timer_research} \leftarrow \text{time}(\text{LaserTechnology}, s_{\text{lvl_LaserTechnology}} + 1)]$$

If the action no_op is applied in state s , it results in a new state s' , in which only the increase of time is taken into account. Essentially, the no_op action does a time step of one second and handles all the administrative tasks:

- The amount of metal we have is raised by the metal production we have.
- The lvl_MetalMine is raised by 1 if we finished building it.
- The lvl_LaserTechnology is raised by 1 if we finished researching it.
- The timer_build is lowered by 1.
- The timer_research is lowered by 1.

Formally:

$$(s, \text{no_op}) \rightarrow s' \begin{aligned} &[\text{res_metal} \leftarrow s_{\text{res_metal}} + \text{prod}(\text{MetalMine}, s_{\text{lvl_MetalMine}})] && \iff s_{\text{timer_build}} = 1 \\ &[\text{lvl_MetalMine} \leftarrow s_{\text{lvl_MetalMine}} + 1] && \iff s_{\text{timer_research}} = 1 \\ &[\text{lvl_LaserTechnology} \leftarrow s_{\text{lvl_LaserTechnology}} + 1] && \iff s_{\text{timer_build}} > 0 \\ &[\text{timer_build} \leftarrow s_{\text{timer_build}} - 1] && \iff s_{\text{timer_build}} > 0 \\ &[\text{timer_research} \leftarrow s_{\text{timer_research}} - 1] && \iff s_{\text{timer_build}} > 0 \end{aligned}$$

2.1.3. Transition function

The actions in our model are all deterministic, so the transition function is:

$$P_a(s, s') = \begin{cases} 1 & \text{if } a \in A_s \wedge (s, a) \rightarrow s' \\ 0 & \text{otherwise} \end{cases}$$

2. Modelling the problem

2.1.4. Reward function

We have not defined a goal state yet for the toy model, but we want to reward the agent for reaching the goal state. Thus we have to define one, for example if our MetalMine and LaserTechnology are both level 3. Formally:

$$s_g = \langle -, 3, 3, -, - \rangle \text{ (where } - \text{ stands for an arbitrary value)}$$

When designing the reward function, we have to carefully consider what we want the agent to learn. We want learn how to get to the goal state, but also in the fastest way possible. So we will reward the agent for getting to the goal state, but punish him for taking longer to do so. This results in the following reward function:

$$R(s) = \begin{cases} 50 - x & \text{if } s \text{ is a goal state} \\ -x & \text{otherwise} \end{cases}$$

Where x is the number of seconds passed since the previous (parent) state.

In other words: the MetalMine and LaserTechnology actions will always reward 0, since they do not advance time. The no_op action will generally reward -1, except when a goal state is reached, then it will reward +49. Note that the no_op action always receives the big +50 reward for reaching a goal state, even though MetalMine or LaserTechnology might be the “deciding” action. This occurs because the no_op increases the level variables. Upon increasing the level variable that triggers a goal state, it receives the +50 reward. This is not desirable behaviour, because an agent would now learn that no_op is good, instead of that MetalMine or LaserTechnology is good. The incorrectly crediting will be addressed in the following sections.

2.1.5. Transition table

A state transition table from starting state to goal state can give us a more graphical impression of how this MDP works. It is also a quick and dirty check if our model does not have anomalies. A hand made transition table can be found in Appendix B.2 (Table B.4).

2.2. Improving the toy MDP

The toy MDP we currently have turns out to have a few problems:

- When scaled up to a full model for Ogame, the state space becomes enormous. Every no_op action generates a new state, which is just slightly different from the parent state.
- A phenomenon similar to starvation exists between the assembly lines.
- If an agent wants to build something, but does not own the resources yet, it has to “spam” no_op actions to advance time and gather resources. Ogame players do not continuously click buttons to gather resources for a desired action. They simply decide that they want to do action a in the future, when the resources are available.
- The “wrong” actions are being credited with the high reach-the-goal reward.

2.2.1. Afterstates

The scaling problem can be solved by introducing afterstates, see Sutton & Barto (1998)[Chapter 6.8]. Afterstates are states that contain the same information, but are reached via different transition paths. For example, in Tic-Tac-Toe, player one choosing action a and then player two choosing action b , leads to a state with the same game board as player two choosing b and then player one choosing a . Combining these two states, generated by different transition paths, in the same afterstate can greatly reduce the state space.

In our case, afterstates are a bit different: they are states that actually hold the result of an action. For example, the afterstate of the MetalMine action would be the state in which the Metal Mine has finished building. In the learning process we will only consider these afterstates. This greatly reduces the state space, by throwing away the unimportant states between action selection and action completion.

To reduce the state space even more, we can also only apply afterstates if the agent can pick a “serious” action, in other words: an action other than no_op.¹ This is a good thing, because we do not really want to bother the agent with having to advance time by using no_op. When a human player plays Ogame, he also does not have to click a button every time he wants to go one second forward in time. Luckily our definition of the actions in the MDP includes preconditions that precisely fit these afterstates: a state s is an afterstate if and only if state s has at least one action available besides no_op. So, if we want to only consider afterstates in the MDP, we want it to automatically apply the no_op action if it is the only action available. To do this, we changed the transitions in the MDP. So far this was our notation for transitions:

$$(s, a) \rightarrow s'[var \leftarrow value]$$

¹This definition of afterstates makes it possible that you cannot “end the game” until you reach an afterstate, even though you already accomplished the goal. This problem will be tackled in the next section, where “serious” actions are guaranteed to become available at a goal state.

2. Modelling the problem

Automatically applying the no_op action can be done by changing it to a recursive notation:

$$(s, a) \rightarrow \begin{cases} s' & \text{if } |A_{s'}| > 1 \\ s'' & \text{if } A_{s'} = \{\text{no_op}\} \end{cases} \quad \begin{cases} \text{where } (s, a) \rightarrow s'[var \leftarrow value] \\ \text{where } (s, a) \rightarrow s'[var \leftarrow value] \wedge (s', \text{no_op}) \rightarrow s'' \end{cases}$$

The first case applies when we have a transition which results in an afterstate, then we apply the transition in the same way as we did in the earlier MDP. The second case applies when we have a transition from which we can only continue by doing no_op. We first apply the transition and then continuously apply the no_op action. This will continue until we reach an afterstate.

For a graphical impression, compare Figure 2.1 to Figure 2.2. Both figures show a transition path in which the agent starts in afterstate s_a , chooses some action b and continues using no_op actions until he reaches the next afterstate $s_{a'}$. In the first figure, which shows the old toy MDP, the agent has to perform all the no_op actions to reach $s_{a'}$ himself. The toy MDP improved with afterstate transitions is shown in the second figure. Here the agent only has to perform action b .

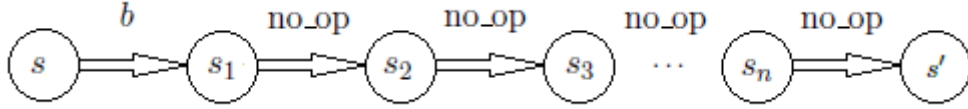


Figure 2.1.: Old transition system without looping no_op

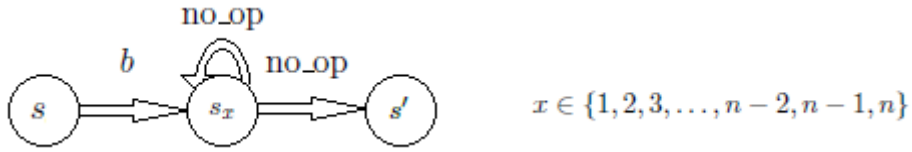


Figure 2.2.: Afterstate transition system with looping no_op

Note that we will not rewrite every single action effect to this notation, because of clarity and brevity. We showed how it works and assume it is implicitly used in the model. The introduction of these afterstates reduces the state space incredibly. If we rewrite the transition table B.4 for this new model to table B.5, the number of transitions lowers from 47 to 18. This is a decrease of about 62%. This reduction becomes even larger, the longer the game goes on, due to the nature of Ogame: timers for actions become longer every time you upgrade that action.

2.2.2. Starvation and spamming no_op to gain resources

Because the assembly lines share the same resource pool, the agent will probably learn with some kind of bias towards the cheapest actions, simply because they are available more. This behaviour can be compared to starvation. The only way to prevent starvation in the current model is by using no_op actions and hoping that different, more attractive actions become available. Unfortunately this is not a good method, because performing no_op actions gives a negative reward. This might make the learning agent even less inclined to prevent starvation.

2. Modelling the problem

Besides starvation, the model in general facilitates no_op “spamming” behaviour. When playing Ogame, a player usually finds some action a very attractive, but cannot start it yet because of a lack of resources. They will then usually wait until the resources are gathered and then start the action. Essentially, at time t , one decides to perform action a at time $t + x$, for some x . This decision can be simulated by performing no_op actions, but for the same reasons as with the starvation, it is a bad method. On top of this, learning when to use the no_op action would become really complex: when using it for the “wait until action a can be performed”, it puts the experiences for every action a into one: no_op. It also lacks information: we cannot say for which action we are waiting. The no_op action would not be an action with just one use, but many uses. A reinforcement learning agent will probably have difficulties learning when to use no_op this way, or he might not learn it at all.

Wait and no-wait actions

We can solve both problems of the previous mentioned section by introducing *wait* and *no-wait* actions. The wait actions are the “perform action a at time $t + x$ ” actions. These actions put the corresponding assembly line in a waiting mode and when the resources are gathered, the action is automatically performed. No other actions can be performed while in waiting mode or when one is still being performed.

We will not add wait actions for every single action in the model, this would be redundant. Instead, we will replace all the current actions by corresponding wait actions. For example, when choosing the action MetalMine, if enough resources are available, the action will start, if not, waiting mode will be activated for this action. This way, the agent can choose to perform an action, even though it does not own the necessary resources yet. As a small example, action preconditions might change from:

$$\text{MetalMine} \in A_s \iff s_{\text{timer_build}} = 0 \wedge \text{cost}(\text{MetalMine}, s_{\text{lvl_MetalMine}} + 1) \leq s_{\text{res_metal}}$$

To:

$$\text{MetalMine} \in A_s \iff s_{\text{timer_build}} = 0 \wedge \neg s_{\text{waiting_build}}$$

Where $s_{\text{waiting_build}}$ is *true* if the building assembly line is in waiting mode and *false* if not. An action effect might change from:

$$(s, \text{MetalMine}) \rightarrow s' [\text{res_metal} \leftarrow s_{\text{res_metal}} - \text{cost}(\text{MetalMine}, s_{\text{lvl_MetalMine}} + 1)] \\ [\text{timer_build} \leftarrow \text{time}(\text{MetalMine}, s_{\text{lvl_MetalMine}} + 1)]$$

To:

$$(s, \text{MetalMine}) \rightarrow \begin{cases} s' & [\text{res_metal} \leftarrow s_{\text{res_metal}} - \text{cost}(\text{MetalMine}, s_{\text{lvl_MetalMine}} + 1)] \\ & [\text{timer_build} \leftarrow \text{time}(\text{MetalMine}, s_{\text{lvl_MetalMine}} + 1)] \\ & \text{if } \text{cost}(\text{MetalMine}, s_{\text{lvl_MetalMine}} + 1) \leq s_{\text{res_metal}} \\ s' & [\text{action_build} \leftarrow \text{MetalMine}] \\ & [\text{waiting_build} \leftarrow \text{true}] \\ & \text{otherwise} \end{cases}$$

2. Modelling the problem

Where s_{action_build} can be set to the action which the building assembly line is waiting for.

If wait actions are applied this way, starvation can occur. If one assembly line has expensive actions and is waiting for resources, another assembly line can constantly consumes all resources. To prevent this, we introduce one no-wait action for each assembly line. These actions put the assembly line in a waiting-for-nothing mode or “do nothing for some time” mode.

The precondition and effect of a no-wait action could look like this:

$$\begin{aligned} no_wait_build \in A_s &\iff s_{timer_build} = 0 \wedge \neg s_{waiting_build} \wedge \\ &\neg(s_{waiting_research} \wedge s_{action_research} = null) \\ (s, no_wait_build) &\rightarrow s'[action_build \leftarrow null] \\ &[waiting_build \leftarrow true] \end{aligned}$$

The `no_wait_build` action can only put the building assembly line in the waiting-for-nothing mode if:

- it is not currently building something.
- it is not already in waiting mode.
- the other (research) assembly line is not in waiting-for-nothing mode, to prevent deadlocks.

These actions give the learning agent means to prevent starvation, without getting negative rewards.

Added benefit of the wait and no-wait actions is that the reward problem is also solved: the reward function does not wrongly credit actions anymore. It now gives the large *reach-the-goal* reward to the “deciding” action or the no-wait action that precedes it.

2.2.3. Toy MDP version 2

To include all the above mentioned improvements, our toy model needs to be rewritten quite a bit. For example:

- New variables need to be added to the state, to represent whether an assembly line is in waiting mode, and for which action.
- Preconditions for actions need to be changed, and added for the no-wait actions.
- The effects of the actions need to be changed.

The complete rewritten toy MDP version 2 can be found in Appendix B.3.

Note that the `no_op` action still does all the administrative tasks, but the learning agent itself does not have to choose it anymore. `No_op` was used for two things: advancing time and hoping that more attractive actions would come available. Both these things have become trivial since the addition of afterstates, wait and no-wait actions. When using this model in a learning algorithm, the `no_op` action can be hidden from the agent with no problems.

Also note that we can easily find the optimal transition path from start to goal state by hand. This is convenient later in this thesis, where we want to see how reinforcement learning performs on the toy model. See table B.6 for the optimal transition path. It turns out that the optimal path from start to goal state takes 34 simulated seconds. The best policy an agent could learn is a policy that follows this path.

2.3. Ogame MDP

2.3.1. From toy MDP to Ogame MDP

Given the toy MDP as a base, the task of modelling the Ogame environment is a lot more manageable. We can achieve this by adding and expanding the toy MDP:

- Besides cost, time and production functions, we add functions for calculating the production factor and storage capacities.
- Adding all the Ogame actions. See Appendix A.
- Add the third assembly line (shipyard).
- Rewriting the no_op action to handle all the new administrative tasks.

We will, again, define the four elements needed for an MDP in the coming sections. In these definitions we will use the same notations for transitions as in the toy MDP. We need some new types and functions for the Ogame MDP. A simple overview is given in Table 2.3. The full definitions of the functions can be found in Appendix C.

Type / function	
pint <i>variablename</i>	A type indicator, this variable is a positive integer: 0, 1, 2, 3, ...
double <i>variablename</i>	A type indicator, this variable is a double-precision floating-point.
bool <i>variablename</i>	A type indicator, this variable is a boolean: <i>true</i> or <i>false</i> .
action <i>variablename</i>	A type indicator, this variable is an action $a \in A$
cost_resource(<i>actionname</i> , <i>level</i>)	Returns the <i>resource</i> cost of the action <i>actionname</i> for the level <i>level</i> .
time_assemblyline(<i>cost</i> , <i>level_roboticsfactory</i>)	Returns the duration (building or research) time of an action for <i>assemblyline</i> . The calculation uses only the resource <i>cost</i> of the action and the <i>level_roboticsfactory</i> .
prod_resource(<i>levels</i>)	Returns the <i>resource</i> production, calculating it by using the <i>levels</i> of several buildings.
prec_costs(<i>resources</i> , <i>actionname</i> , <i>level</i>)	Returns <i>true</i> if we own the resources needed to perform the action <i>actionname</i> for the level <i>level</i> .
prod_factor(<i>levels</i>)	Returns the production factor (0.0 - 1.0), calculating it by using the <i>levels</i> of several buildings.

Table 2.3.: Notations and functions for Ogame MDP

2.3.2. States

The state structure stays the same as in the toy model. However, we now store the resources in a double type instead of pint, because of introducing the production factor in the model. Under influence of the production factor, production can become a real number instead of a natural one.

2. Modelling the problem

The set of states S is defined as:

$s \in S$ if $s =$ < double res_metal, double res_crystal, double res_deuterium,
pint lvl_MetalMine, pint lvl_CrystalMine, pint lvl_DeuteriumSynthesizer,
pint lvl_SolarPlant,
pint lvl_RoboticsFactory, pint lvl_Shipyard, pint lvl_ResearchLab,
pint lvl_CrystalStorage, pint lvl_MetalStorage, pint lvl_DeuteriumTank,
pint lvl_EnergyTechnology, pint lvl_ImpulseDrive, pint lvl_ColonyShip,
pint timer_building, pint timer_research, pint timer_shipyard,
bool waiting_building, bool waiting_research, bool waiting_shipyard,
action action_building, action action_research, action action_shipyard >

Table 2.4 explains what these variables represent.

Variable name	Explanation
res_resource	The amount of <i>resource</i> owned.
lvl_actionname	The current level of the building, research or ship <i>actionname</i> .
timer_building	Building time remaining in seconds.
timer_research	Research time remaining in seconds.
timer_shipyard	Ship building time remaining in seconds.
waiting_building	True if build assembly line is in waiting mode.
waiting_research	Idem for the research assembly line.
waiting_shipyard	Idem for the shipyard assembly line.
action_building	The action for which the build assembly line is waiting or performing.
action_research	Idem for the research assembly line.
action_shipyard	Idem for the shipyard assembly line.

Table 2.4.: Legend for Ogame MDP state variables

The starting state s_0 , which represents a new Ogame account, is defined as:

$s_0 =$ < 500, 500, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, *false*, *false*, *false*, *null*, *null*, *null* >

Note that a new account starts with 500 metal and 500 crystal.

2.3.3. Actions

The set of actions A is defined as:

$A = \{$ MetalMine, CrystalMine, DeuteriumSynthesizer, SolarPlant,
RoboticsFactory, CrystalStorage, MetalStorage, DeuteriumTank,
ResearchLab, Shipyard,
EnergyTechnology, ImpulseDrive, ColonyShip,
no_building, no_research, no_shipyard, no_op $\}$

It contains all the actions we determined (see Appendix A), including the no_wait actions for the three assembly lines and the administrative no_op action.

Preconditions

$A_s \subseteq A$ is the set of actions which includes all actions available in state s . For clarity, we will first define three sets of available actions, one for each assembly line. We will then combine these three sets to create A_s .

2. Modelling the problem

Part 1: building assembly line The set $A_{building_s}$ will contain the available building actions in A_s .

$$\begin{aligned} & \{ \text{MetalMine, CrystalMine, DeuteriumSynthesizer, SolarPlant, RoboticsFactory,} \\ & \text{CrystalStorage, MetalStorage, DeuteriumTank, ResearchLab} \} \subseteq A_{building_s} \\ & \iff s_{timer_building} = 0 \wedge \neg s_{waiting_building} \end{aligned}$$

$$\begin{aligned} & \text{Shipyard} \in A_{building_s} \\ & \iff s_{timer_building} = 0 \wedge \neg s_{waiting_building} \wedge s_{lvl_RoboticsFactory} \geq 2 \end{aligned}$$

Part 2: research assembly line The set $A_{research_s}$ will contain the available research actions in A_s . This set will be created from the set $A_{req_research_s}$, which contains all the action requirements in Ogame itself. This split up is needed, because $A_{req_research_s}$ is used later in the preconditions for the no-wait actions.

$$\begin{aligned} & \text{EnergyTechnology} \in A_{req_research_s} \iff s_{lvl_ResearchLab} \geq 1 \\ & \text{ImpulseDrive} \in A_{req_research_s} \iff s_{lvl_ResearchLab} \geq 2 \wedge s_{lvl_EnergyTechnology} \geq 1 \end{aligned}$$

$$A_{research_s} = \begin{cases} A_{req_research_s} & \text{if } s_{timer_research} = 0 \wedge \neg s_{waiting_research} \\ \emptyset & \text{otherwise} \end{cases}$$

Part 3: shipyard assembly line The shipyard assembly line is handled in the same manner.

$$\begin{aligned} & \text{ColonyShip} \in A_{req_shipyard_s} \iff s_{lvl_Shipyard} \geq 4 \wedge s_{lvl_ImpulseDrive} \geq 3 \\ & A_{shipyard_s} = \begin{cases} A_{req_shipyard_s} & \text{if } s_{timer_shipyard} = 0 \wedge \neg s_{waiting_shipyard} \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

Note that we pretend the shipyard assembly line works the same as the other two. In Ogame it actually works like a sort of waiting queue, where you can queue up actions, as long as you have the resources for them. In Section 1.2.1 we determined that we only have to build one Colony Ship, so this behaviour is not needed in our MDP.

Combining parts 1, 2, 3 and adding no-wait actions We can now combine the sets into set A_s :

$$A_{building_s} \cup A_{research_s} \cup A_{shipyard_s} \subseteq A_s$$

Note that we used \subseteq , instead of $=$, for a reason: we still want to add the no-wait actions to A_s .

2. Modelling the problem

The no-wait actions tend to get a bit complex, in order to avoid deadlocks. Deadlocks occur if all the assembly lines are in waiting mode. To avoid these, the no_building action is only available if:

- The building assembly line is not building something or in waiting mode.
- Not all other assembly lines are in a no-waiting mode.
- Not all other assembly lines have no “serious” action to choose. Where a serious action is any action besides a no-wait action or no_op.
- The research assembly line is not in waiting-for-nothing mode while the shipyard assembly line has no serious action available.
- The shipyard assembly line is not in waiting-for-nothing mode while the research assembly line has no serious action available.

$$\begin{aligned}
 \text{no_building} \in A_s \iff & s_{\text{timer_building}} = 0 \wedge \neg s_{\text{waiting_building}} \wedge \\
 & \neg(s_{\text{waiting_research}} \wedge s_{\text{action_research}} = \text{null} \\
 & \quad \wedge s_{\text{waiting_shipyard}} \wedge s_{\text{action_shipyard}} = \text{null}) \wedge \\
 & \neg(A_{\text{req_research}_s} = \emptyset \wedge A_{\text{req_shipyard}_s} = \emptyset) \wedge \\
 & \neg(s_{\text{waiting_research}} \wedge s_{\text{action_research}} = \text{null} \wedge A_{\text{req_shipyard}_s} = \emptyset) \wedge \\
 & \neg(s_{\text{waiting_shipyard}} \wedge s_{\text{action_shipyard}} = \text{null} \wedge A_{\text{req_research}_s} = \emptyset)
 \end{aligned}$$

The remaining no-wait actions have similar preconditions:

$$\begin{aligned}
 \text{no_research} \in A_s \iff & s_{\text{timer_research}} = 0 \wedge \neg s_{\text{waiting_research}} \wedge \\
 & \neg(s_{\text{waiting_building}} \wedge s_{\text{action_building}} = \text{null} \\
 & \quad \wedge s_{\text{waiting_shipyard}} \wedge s_{\text{action_shipyard}} = \text{null}) \wedge \\
 & \neg(s_{\text{waiting_building}} \wedge s_{\text{action_building}} = \text{null} \wedge A_{\text{req_shipyard}_s} = \emptyset)
 \end{aligned}$$

$$\begin{aligned}
 \text{no_shipyard} \in A_s \iff & s_{\text{timer_shipyard}} = 0 \wedge \neg s_{\text{waiting_shipyard}} \wedge \\
 & \neg(s_{\text{waiting_building}} \wedge s_{\text{action_building}} = \text{null} \\
 & \quad \wedge s_{\text{waiting_research}} \wedge s_{\text{action_research}} = \text{null}) \wedge \\
 & \neg(s_{\text{waiting_building}} \wedge s_{\text{action_building}} = \text{null} \wedge A_{\text{req_research}_s} = \emptyset)
 \end{aligned}$$

$$\text{no_op} \in A_s$$

We have now fully defined A_s .

Note that there are two extra preconditions for some actions:

- On a new Ogame account, the deuterium production is zero, while metal and crystal get produced in small amounts. Actions that require deuterium are not available if there is no deuterium production. They can become available if a Deuterium Synthesizer is built and the production factor is above zero. Note that if keeping these actions available, without deuterium production, the agent might learn to avoid these actions. This, however, adds extra complexity to an already complex problem. It makes learning harder and more preconditions have to be added to avoid deadlocks.
- Actions that require more resources than we can store in our storage tanks, should not be available. It is also possible for an agent to learn this behaviour himself, but we decide against it for the same reasons.

2. Modelling the problem

These preconditions will not be included in the MDP, for the sake of clarity. They will, however, be included in the simulator in the next chapter.

Effects

Because most actions have a similar effect, we will introduce two sets, to be used as shorthands:

$$\begin{aligned} \text{BuildingActions} &= \{ \text{MetalMine, CrystalMine, DeuteriumSynthesizer, SolarPlant,} \\ &\quad \text{RoboticsFactory, CrystalStorage, MetalStorage, DeuteriumTank} \} \\ \text{ResearchActions} &= \{ \text{EnergyTechnology, ImpulseDrive} \} \end{aligned}$$

Simple building actions

$$\forall ba \in \text{BuildingActions}, \quad (s, ba) \rightarrow \left\{ \begin{array}{l} s' \quad [\text{res_metal} \leftarrow s_{\text{res_metal}} - \text{cost_metal}(ba, s_{\text{lvl_ba}} + 1)] \\ \quad [\text{res_crystal} \leftarrow s_{\text{res_crystal}} - \text{cost_crystal}(ba, s_{\text{lvl_ba}} + 1)] \\ \quad [\text{res_deuterium} \leftarrow s_{\text{res_deuterium}} - \text{cost_deuterium}(ba, s_{\text{lvl_ba}} + 1)] \\ \quad [\text{timer_building} \leftarrow \text{time_building}(\text{cost_metal}(ba, s_{\text{lvl_ba}} + 1), \\ \quad \quad \text{cost_crystal}(ba, s_{\text{lvl_ba}} + 1), s_{\text{lvl_RoboticsFactory}})] \\ \quad [\text{action_building} \leftarrow ba] \\ \quad \text{if } \text{prec_costs}(s_{\text{res_metal}}, s_{\text{res_crystal}}, s_{\text{res_deuterium}}, ba, s_{\text{lvl_ba}} + 1) \\ \\ s' \quad [\text{action_building} \leftarrow ba] \\ \quad [\text{waiting_building} \leftarrow \text{true}] \\ \quad \text{otherwise} \end{array} \right.$$

In other words, actions start building if there are enough resources for that action. If not, it puts the building assembly line in waiting mode for that action. Note that $s_{\text{lvl_ba}}$ is shorthand for $s_{\text{lvl_<action>}}$, where $\langle \text{action} \rangle$ is substituted with ba . This was added to improve readability.

Complex building actions The actions ResearchLab and Shipyard are a bit more complex than the simple building actions, because of two reasons:

- In Ogame, the ResearchLab cannot be upgraded if research is conducted at the same time.
- The Shipyard cannot be upgraded if ships are being constructed at the same time.

We will define the action ResearchLab to put itself in the waiting mode until the research is finished, idem for the Shipyard. For the next two definitions, we have shortened ResearchLab to RL and Shipyard to SY , because of layout reasons.

2. Modelling the problem

$$\begin{aligned}
 (s, RL) &\rightarrow \left\{ \begin{array}{l}
 s' \quad \begin{array}{l}
 [\text{res_metal} \leftarrow s_{\text{res_metal}} - \text{cost_metal}(RL, s_{\text{lvl_RL}} + 1)] \\
 [\text{res_crystal} \leftarrow s_{\text{res_crystal}} - \text{cost_crystal}(RL, s_{\text{lvl_RL}} + 1)] \\
 [\text{res_deuterium} \leftarrow s_{\text{res_deuterium}} - \text{cost_deuterium}(RL, s_{\text{lvl_RL}} + 1)] \\
 [\text{timer_building} \leftarrow \text{time_building}(\text{cost_metal}(RL, s_{\text{lvl_RL}} + 1), \\
 \quad \text{cost_crystal}(RL, s_{\text{lvl_RL}} + 1), s_{\text{lvl_RoboticsFactory}})] \\
 [\text{action_building} \leftarrow RL] \\
 \text{if } \text{prec_costs}(s_{\text{res_metal}}, s_{\text{res_crystal}}, s_{\text{res_deuterium}}, RL, s_{\text{lvl_RL}} + 1) \wedge \\
 \quad s_{\text{timer_research}} = 0 \\
 s' \quad \begin{array}{l}
 [\text{action_building} \leftarrow RL] \\
 [\text{waiting_building} \leftarrow \text{true}] \\
 \text{otherwise}
 \end{array}
 \end{array}
 \right. \\
 (s, SY) &\rightarrow \left\{ \begin{array}{l}
 s' \quad \begin{array}{l}
 [\text{res_metal} \leftarrow s_{\text{res_metal}} - \text{cost_metal}(SY, s_{\text{lvl_SY}} + 1)] \\
 [\text{res_crystal} \leftarrow s_{\text{res_crystal}} - \text{cost_crystal}(SY, s_{\text{lvl_SY}} + 1)] \\
 [\text{res_deuterium} \leftarrow s_{\text{res_deuterium}} - \text{cost_deuterium}(SY, s_{\text{lvl_SY}} + 1)] \\
 [\text{timer_building} \leftarrow \text{time_building}(\text{cost_metal}(SY, s_{\text{lvl_SY}} + 1), \\
 \quad \text{cost_crystal}(SY, s_{\text{lvl_SY}} + 1), s_{\text{lvl_RoboticsFactory}})] \\
 [\text{action_building} \leftarrow SY] \\
 \text{if } \text{prec_costs}(s_{\text{res_metal}}, s_{\text{res_crystal}}, s_{\text{res_deuterium}}, SY, s_{\text{lvl_SY}} + 1) \wedge \\
 \quad s_{\text{timer_shipyard}} = 0 \\
 s' \quad \begin{array}{l}
 [\text{action_building} \leftarrow SY] \\
 [\text{waiting_building} \leftarrow \text{true}] \\
 \text{otherwise}
 \end{array}
 \end{array}
 \right.
 \end{aligned}$$

2. Modelling the problem

Research actions According to Ogame, these actions can only be performed if the ResearchLab building is not being upgraded. If it is, the action is put in waiting mode.

$$\forall ra \in \text{ResearchActions}, \quad (s, ra) \rightarrow \left\{ \begin{array}{l} s' \quad \begin{array}{l} [\text{res_metal} \leftarrow s_{\text{res_metal}} - \text{cost_metal}(ra, s_{\text{lvl_ra}} + 1)] \\ [\text{res_crystal} \leftarrow s_{\text{res_crystal}} - \text{cost_crystal}(ra, s_{\text{lvl_ra}} + 1)] \\ [\text{res_deuterium} \leftarrow s_{\text{res_deuterium}} - \text{cost_deuterium}(ra, s_{\text{lvl_ra}} + 1)] \\ [\text{timer_research} \leftarrow \text{time_research}(\text{cost_metal}(ra, s_{\text{lvl_ra}} + 1), \\ \quad \text{cost_crystal}(ra, s_{\text{lvl_ra}} + 1), s_{\text{lvl_ResearchLab}})] \\ [\text{action_research} \leftarrow ra] \end{array} \\ \text{if } \text{prec_costs}(s_{\text{res_metal}}, s_{\text{res_crystal}}, s_{\text{res_deuterium}}, ra, s_{\text{lvl_ra}} + 1) \wedge \\ \quad \neg(s_{\text{timer_building}} > 0 \wedge s_{\text{action_building}} = \text{ResearchLab}) \\ s' \quad \begin{array}{l} [\text{action_research} \leftarrow ra] \\ [\text{waiting_research} \leftarrow \text{true}] \\ \text{otherwise} \end{array} \end{array} \right.$$

Shipyard actions Similar to the research actions, these actions can only be performed if the Shipyard building is not being upgraded. If it is, the action is put in waiting mode.

$$(s, \text{ColonyShip}) \rightarrow \left\{ \begin{array}{l} s' \quad \begin{array}{l} [\text{res_metal} \leftarrow s_{\text{res_metal}} - \text{cost_metal}(\text{ColonyShip})] \\ [\text{res_crystal} \leftarrow s_{\text{res_crystal}} - \text{cost_crystal}(\text{ColonyShip})] \\ [\text{res_deuterium} \leftarrow s_{\text{res_deuterium}} - \text{cost_deuterium}(\text{ColonyShip})] \\ [\text{timer_shipyard} \leftarrow \text{time_shipyard}(\text{cost_metal}(\text{ColonyShip}), \\ \quad \text{cost_crystal}(\text{ColonyShip}), s_{\text{lvl_Shipyard}})] \\ [\text{action_shipyard} \leftarrow \text{ColonyShip}] \end{array} \\ \text{if } \text{prec_costs}(s_{\text{res_metal}}, s_{\text{res_crystal}}, s_{\text{res_deuterium}}, \text{ColonyShip}) \wedge \\ \quad \neg(s_{\text{timer_building}} > 0 \wedge s_{\text{action_building}} = \text{Shipyard}) \\ s' \quad \begin{array}{l} [\text{action_shipyard} \leftarrow \text{ColonyShip}] \\ [\text{waiting_shipyard} \leftarrow \text{true}] \\ \text{otherwise} \end{array} \end{array} \right.$$

No-wait actions The no-wait actions simply put the assembly lines in waiting-for-nothing mode.

$$(s, \text{no_building}) \rightarrow s' \begin{array}{l} [\text{action_building} \leftarrow \text{null}] \\ [\text{waiting_building} \leftarrow \text{true}] \end{array}$$

$$(s, \text{no_research}) \rightarrow s' \begin{array}{l} [\text{action_research} \leftarrow \text{null}] \\ [\text{waiting_research} \leftarrow \text{true}] \end{array}$$

$$(s, \text{no_shipyard}) \rightarrow s' \begin{array}{l} [\text{action_shipyard} \leftarrow \text{null}] \\ [\text{waiting_shipyard} \leftarrow \text{true}] \end{array}$$

2. Modelling the problem

The no_op action Before we define the no_op action, we will create some boolean values. These booleans increase the readability of the no_op definition.

$$\begin{aligned}
 b_1 = & \text{ } s_{\text{waiting_building}} \wedge s_{\text{action_building}} \neq \text{null} \wedge \\
 & \text{prec_costs}(s_{\text{res_metal}} + \text{prod_metal}(s_{\text{lvl_MetalMine}}), \\
 & \quad s_{\text{res_crystal}} + \text{prod_crystal}(s_{\text{lvl_CrystalMine}}), \\
 & \quad s_{\text{res_deuterium}} + \text{prod_deuterium}(s_{\text{lvl_DeuteriumSynthesizer}}), \\
 & \quad s_{\text{action_building}}, s_{\text{lvl_action_building}} + 1) \wedge \\
 & (s_{\text{action_building}} \neq \text{ResearchLab} \vee s_{\text{timer_research}} < 1) \wedge \\
 & (s_{\text{action_building}} \neq \text{Shipyard} \vee s_{\text{timer_shipyard}} < 1)
 \end{aligned}$$

Note that, again, $s_{\text{lvl_action_building}}$ is shorthand for the level of the intended action. This definition in other words: b_1 is *true* if:

- The building production line is in waiting mode (but not for no-wait).
- There are enough resources to perform the action.
- If the production line is waiting for ResearchLab, we cannot be researching something.
- If the production line is waiting for Shipyard, we cannot be building a ship.

To summarize: b_1 is *true* if the building assembly line can start building after gathering resources for some action. b_2 and b_3 do the same for the research and shipyard assembly lines, keeping the Ogame constraints in mind:

$$\begin{aligned}
 b_2 = & \text{ } s_{\text{waiting_research}} \wedge s_{\text{action_research}} \neq \text{null} \wedge \\
 & \text{prec_costs}(s_{\text{res_metal}} + \text{prod_metal}(s_{\text{lvl_MetalMine}}), \\
 & \quad s_{\text{res_crystal}} + \text{prod_crystal}(s_{\text{lvl_CrystalMine}}), \\
 & \quad s_{\text{res_deuterium}} + \text{prod_deuterium}(s_{\text{lvl_DeuteriumSynthesizer}}), \\
 & \quad s_{\text{action_research}}, s_{\text{lvl_action_research}} + 1) \wedge \\
 & \neg(s_{\text{timer_building}} > 1 \wedge s_{\text{action_building}} = \text{ResearchLab}) \wedge \\
 & \neg b_1
 \end{aligned}$$

$$\begin{aligned}
 b_3 = & \text{ } s_{\text{waiting_shipyard}} \wedge s_{\text{action_shipyard}} \neq \text{null} \wedge \\
 & \text{prec_costs}(s_{\text{res_metal}} + \text{prod_metal}(s_{\text{lvl_MetalMine}}), \\
 & \quad s_{\text{res_crystal}} + \text{prod_crystal}(s_{\text{lvl_CrystalMine}}), \\
 & \quad s_{\text{res_deuterium}} + \text{prod_deuterium}(s_{\text{lvl_DeuteriumSynthesizer}}), \\
 & \quad s_{\text{action_shipyard}}) \wedge \\
 & \neg(s_{\text{timer_building}} > 1 \wedge s_{\text{action_building}} = \text{Shipyard}) \wedge \\
 & \neg b_1 \wedge \neg b_2
 \end{aligned}$$

Note that b_2 is *false* if we are upgrading the ResearchLab or if b_1 is true, because b_1 might have used all the resources we needed for this action. b_3 is *false* if we are upgrading the Shipyard or if b_1 or b_2 is true, because either might have used the resources we needed.

2. Modelling the problem

$$b_4 = b_1 \vee s_{\text{waiting_building}} \wedge s_{\text{action_building}} = \text{null} \wedge (b_2 \vee b_3 \vee s_{\text{timer_research}} = 1 \vee s_{\text{timer_shipyard}} = 1)$$

$$b_5 = b_2 \vee s_{\text{waiting_research}} \wedge s_{\text{action_research}} = \text{null} \wedge (b_1 \vee b_3 \vee s_{\text{timer_building}} = 1 \vee s_{\text{timer_shipyard}} = 1)$$

$$b_6 = b_3 \vee s_{\text{waiting_shipyard}} \wedge s_{\text{action_shipyard}} = \text{null} \wedge (b_1 \vee b_2 \vee s_{\text{timer_building}} = 1 \vee s_{\text{timer_research}} = 1)$$

b_4 is *true* if the waiting mode on the building assembly line should be removed. b_5 and b_6 do the same for the research and shipyard assembly lines.

With these boolean values we can define the `no_op` action, which can be found in Figure 2.3. Note that again shorthands are used, for example: $s_{\text{lvl_action_building}}$ is shorthand for $s_{\text{lvl_<action>}}$, where $\langle \text{action} \rangle$ is substituted with $s_{\text{action_building}}$. `prod_levels` is shorthand for: “ $s_{\text{lvl_MetalMine}}, s_{\text{lvl_CrystalMine}}, s_{\text{lvl_DeuteriumSynthesizer}}, s_{\text{lvl_SolarPlant}}$ ”

The definition is split into four parts:

1. Rule 1-9, basic tasks. Increase the resources owned with production, increase the levels of the buildings and research if needed, lower the timers.
2. Rule 10-20, building assembly line. Start building actions if they are in waiting mode and the necessary resources are gathered. Rule 20 removes waiting mode if needed.
3. Rule 21-31, research assembly line. Idem. Rule 31 removes waiting mode if needed.
4. Rule 32-41, shipyard assembly line. Idem. Rule 41 removes waiting mode if needed.

Note that variables `res_metal`, `res_crystal` and `res_deuterium` can be substituted more than once: in rules 1-3 and potentially in 10-15, 21-26 or 32-37. However, state s' contains only the latest substitution.

The `no_op` action has the same function in the Ogame MDP: it advances simulated time in the model by one second. If an assembly line has finished waiting for resources, actions are automatically applied. This is done in rules 10-19, 21-30 and 32-40, using the boolean values b_1 , b_2 and b_3 . In addition, waiting mode is removed from the assembly lines if needed, in rules 20, 31 and 41, with boolean values b_4 , b_5 and b_6 . Normal waiting mode is removed from an assembly line if it starts an action. The waiting-for-nothing mode is removed if another assembly line starts or completes an action.

If both assembly lines have finished waiting for resources at the exact same simulated second, they cannot start at the same time, even if there are enough resources for both. This is possible in toy MDP version 2, but omitted here because of complexity and the fact that this only happens on very rare occasion (or maybe even never). We will, however, implement it in the simulator.

Also note that the model currently does not use the `storage_capacity` function, to determine if our storage is full of resources. If storage warehouses/tanks are full, there can be no production. Adding this to the `no_op` definition means one more rule for every rule which substitutes a `res_resource` variable, to check whether the storage is full. This amounts to twelve rules extra. To avoid making the `no_op` action even more complex, we will omit this here. It will, however, be implemented in the simulator.

2.3.4. Transition function

Identical to the toy MDP: see Section 2.1.3.

2. Modelling the problem

$(s, \text{no_op}) \rightarrow s'$	$[\text{res_metal} \leftarrow s_{\text{res_metal}} + \text{prod_metal}(\text{prod_levels})]$	
2	$[\text{res_crystal} \leftarrow s_{\text{res_crystal}} + \text{prod_crystal}(\text{prod_levels})]$	
3	$[\text{res_deuterium} \leftarrow s_{\text{res_deuterium}} + \text{prod_deuterium}(\text{prod_levels})]$	
4	$[\text{lvl_saction_building} \leftarrow \text{slvl_action_building} + 1]$	$\iff \text{stimer_building} = 1$
5	$[\text{lvl_saction_research} \leftarrow \text{slvl_action_research} + 1]$	$\iff \text{stimer_research} = 1$
6	$[\text{lvl_saction_shipyard} \leftarrow \text{slvl_action_shipyard} + 1]$	$\iff \text{stimer_shipyard} = 1$
7	$[\text{timer_building} \leftarrow \text{stimer_building} - 1]$	$\iff \text{stimer_building} > 0$
8	$[\text{timer_research} \leftarrow \text{stimer_research} - 1]$	$\iff \text{stimer_research} > 0$
9	$[\text{timer_shipyard} \leftarrow \text{stimer_shipyard} - 1]$	$\iff \text{stimer_shipyard} > 0$
10	$[\text{res_metal} \leftarrow s_{\text{res_metal}} + \text{prod_metal}(\text{prod_levels}) -$	
11	$\text{cost_metal}(s_{\text{action_building}}, \text{slvl_action_building} + 1)]$	$\iff b_1$
12	$[\text{res_crystal} \leftarrow s_{\text{res_crystal}} + \text{prod_crystal}(\text{prod_levels}) -$	
13	$\text{cost_crystal}(s_{\text{action_building}}, \text{slvl_action_building} + 1)]$	$\iff b_1$
14	$[\text{res_deuterium} \leftarrow s_{\text{res_deuterium}} + \text{prod_deuterium}(\text{prod_levels}) -$	
15	$\text{cost_deuterium}(s_{\text{action_building}}, \text{slvl_action_building} + 1)]$	$\iff b_1$
16	$[\text{timer_building} \leftarrow \text{time_building}(\text{cost_metal}(s_{\text{action_building}}, \text{slvl_action_building} + 1),$	
17	$\text{cost_crystal}(s_{\text{action_building}}, \text{slvl_action_building} + 1),$	
18	$\text{slvl_RoboticsFactory})]$	$\iff b_1$
19	$[\text{waiting_building} \leftarrow \text{false}]$	$\iff b_4$
20		
21	$[\text{res_metal} \leftarrow s_{\text{res_metal}} + \text{prod_metal}(\text{prod_levels}) -$	
22	$\text{cost_metal}(s_{\text{action_research}}, \text{slvl_action_research} + 1)]$	$\iff b_2$
23	$[\text{res_crystal} \leftarrow s_{\text{res_crystal}} + \text{prod_crystal}(\text{prod_levels}) -$	
24	$\text{cost_crystal}(s_{\text{action_research}}, \text{slvl_action_research} + 1)]$	$\iff b_2$
25	$[\text{res_deuterium} \leftarrow s_{\text{res_deuterium}} + \text{prod_deuterium}(\text{prod_levels}) -$	
26	$\text{cost_deuterium}(s_{\text{action_research}}, \text{slvl_action_research} + 1)]$	$\iff b_2$
27	$[\text{timer_research} \leftarrow \text{time_research}(\text{cost_metal}(s_{\text{action_research}}, \text{slvl_action_research} + 1),$	
28	$\text{cost_crystal}(s_{\text{action_research}}, \text{slvl_action_research} + 1),$	
29	$\text{slvl_ResearchLab})]$	$\iff b_2$
30	$[\text{waiting_research} \leftarrow \text{false}]$	$\iff b_5$
31		
32	$[\text{res_metal} \leftarrow s_{\text{res_metal}} + \text{prod_metal}(\text{prod_levels}) -$	
33	$\text{cost_metal}(s_{\text{action_shipyard}})]$	$\iff b_3$
34	$[\text{res_crystal} \leftarrow s_{\text{res_crystal}} + \text{prod_crystal}(\text{prod_levels}) -$	
35	$\text{cost_crystal}(s_{\text{action_shipyard}})]$	$\iff b_3$
36	$[\text{res_deuterium} \leftarrow s_{\text{res_deuterium}} + \text{prod_deuterium}(\text{prod_levels})$	
37	$\text{cost_deuterium}(s_{\text{action_shipyard}})]$	$\iff b_3$
38	$[\text{timer_shipyard} \leftarrow \text{time_shipyard}(\text{cost_metal}(s_{\text{action_shipyard}}),$	
39	$\text{cost_crystal}(s_{\text{action_shipyard}}),$	
40	$\text{slvl_Shipyard})]$	$\iff b_3$
41	$[\text{waiting_shipyard} \leftarrow \text{false}]$	$\iff b_6$

Figure 2.3.: The effect of the no.op action

2. Modelling the problem

2.3.5. Reward function

Our goal state, as mentioned in Section 1.2.1, is owning one Colony Ship:

$$s_g = \langle -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, - \rangle$$

$$R(s) = \begin{cases} 1,000,000 - x & \text{if } s \text{ is a goal state} \\ -x & \text{otherwise} \end{cases}$$

Where x is the number of seconds passed since the previous (parent) state.

To get an indication on how long it takes to get to a decent goal state, we followed a simple and static policy on the simulator, which will be introduced in the next section. When following this policy to the goal state, it takes 837,629 simulated seconds. Figure 2.4 shows this. We give our agent -1 reward each second. In the end, we want him to get an accumulative small positive reward for finding a good transition path. For this reason, we choose 1,000,000 as the reward for reaching the goal state.

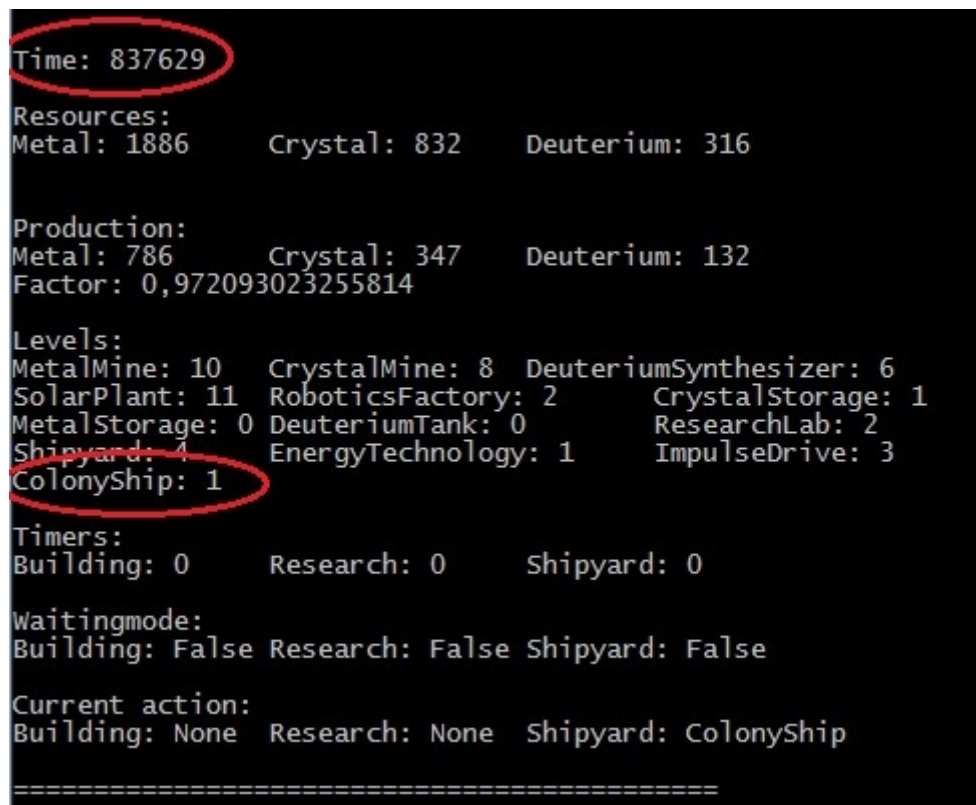


Figure 2.4.: Sample goal state produced by a simple policy

2.4. Research product one

We now have a fully defined MDP for Ogame. Research question one, “Is there a model of Ogame, which we can use in reinforcement learning?”, can now be answered with yes, by providing this model.

3. Building the environment

Now that we have an MDP for both the toy problem (toy MDP v2) and the Ogame problem, we want to represent them in a programming language. By doing this, we created the environment shown in Figure 1.2. Consequently, we can let an agent learn within that environment. We will call this piece of software a simulator.

Let us first determine what the agent needs from the environment, by recalling Section 1.4.2. The agent needs to know:

- What the current game state is. Thus it needs a state signal from the environment.
- To pick actions, it needs to know what the available actions are for the current state. Thus it needs a signal that gives him A_s .
- What the reward for reaching a state is. Thus it needs a reward signal from the environment.

The agent needs to be able to:

- Perform actions in A_s . Thus it needs a method to execute actions and change the environment.

To let the environment meet these requirements, the simulator needs to implement the following methods (for a class-based programming language):

- `GetStateSignal()`
- `GetPossibleActions()`
- `GetReward()`
- `DoAction(Action a)`

Because the learning process for this problem is episodic, we also have to implement the following methods:

- `IsTerminal()`, a method to check whether the current environment state is a goal state.
- `Reset()`, a method to reset the simulator to s_0 .

With these methods the agent can learn episode after episode.

In this thesis we will describe how we implemented the simulator for both the toy MDP version 2 and the Ogame MDP. The implementation is done in the C# programming language. Testing reinforcement learning on the toy MDP can give us invaluable insight for the bigger Ogame problem. In the next sections we will discuss all the classes coded for the Ogame simulator and highlight some interesting code snippets. Afterwards, we will show some tweaks to the simulator's computing time. Because it is redundant, code for the toy simulator will not be discussed. A link to the complete source code for the Ogame simulator can be found in Appendix D. It has significant internal documentation, to make it easier to jump into.

3.1. State class

This class implements the state variables we used in the model, which can be copied one-on-one to the simulator. The constructor of this class sets all these fields to their zero value, according to the starting state we defined in the model. Besides these fields, there are some extra fields for internal use in the simulator. These are for example for calculating the time differences between states for the `GetReward()` method.

The State class also includes the `IsTerminal()` method:

```
public bool IsTerminal()
{
    return LevelActions[(int)Action.ActionNames.ColonyShip] >= 1;
}
```

It returns true if the environment is in a goal state and false otherwise.

Because Ogame is a game with no end, the state space is also infinitely large. Learning with an infinite state space is hard when working with goal states, because it might take a very long time to reach the goal state, depending on the action selection method. Some learning algorithms will not work if the state space is infinite. To combat this we created a new `IsTerminal()` method that implements a ceiling on the state space:

```
public bool IsTerminalWithTimeLimit()
{
    return LevelActions[(int)Action.ActionNames.ColonyShip] >= 1 ||
        TimeCurrent >= TimeLimit;
}
```

The variable `TimeLimit` in the current implementation is set to 50,000,000 seconds. Recall that the example transition path in the Ogame model only takes 837,629 simulated seconds. Setting the `TimeLimit` that high makes sure that there is plenty of time to find a goal state. If the goal state is not reached within this bound, the agent simply does not receive the big positive reward for the goal state.

3.2. Action class

The action class implements some preconditions for actions, which we did not include in the model. We mentioned these earlier:

- An action that requires deuterium, while deuterium production is zero, cannot be available.
- Actions that require more resources than we can store in our storage tanks, cannot be available.

These two preconditions are implemented in the following two code snippets of the `AreRequirementsMet(State s)` method:

```
if (OgameFormulas.ResourceProduction(2, State) == 0.0 &&
    RequiresDeuterium(State))
    return false;
```

```
if (OgameFormulas.ResourceCost(0, ActionName,
    State.LevelActions[(int)ActionName] + 1)
    > OgameFormulas.StorageCapacity(0, State))
    return false; //Return false if we do not have enough metal capacity
```

3. Building the environment

Besides implementing these two features, which are not in the model, it also implements the Ogame technology requirements, for example:

```
case ActionNames.ColonyShip :
    return State.LevelActions[(int) ActionNames.Shipyard] >= 4 &&
        State.LevelActions[(int) ActionNames.ImpulseDrive] >= 3;
```

It returns false if the ColonyShip cannot be built yet, due to lack of research.

The `IsAvailable(State s)` method determines if a certain action is available in state s . It combines the `AreRequirementsMet(State s)` method and the rest of the preconditions in the model. When checking for example if MetalMine is available, it uses the following code:

```
bool NoOpAddition = true;
return NoOpAddition && State.TimerBuilding == 0 &&
    !State.WaitingBuilding && AreRequirementsMet(State);
```

Recall the preconditions in the model:

$$(MetalMine) \in A_{building_s} \iff s_{timer_building} = 0 \wedge \neg s_{waiting_building}$$

The source code looks almost identical to the model. The `NoOpAddition` variable is used to determine the availability of the no-wait actions.

The following code determines if the no_building action is available:

```
if (NoOperation)
    NoOpAddition = !(State.WaitingResearch &&
        State.ActionResearch == null && State.WaitingShipyards &&
        State.ActionShipyards == null) &&
        (AreRequirementsMetAtLeastOneResearch(State) ||
        AreRequirementsMetAtLeastOneShipyards(State)) &&
        !(State.WaitingResearch && State.ActionResearch == null &&
        !AreRequirementsMetAtLeastOneShipyards(State)) &&
        !(State.WaitingShipyards && State.ActionShipyards == null &&
        !AreRequirementsMetAtLeastOneResearch(State));
return NoOpAddition && State.TimerBuilding == 0 &&
    !State.WaitingBuilding && AreRequirementsMet(State);
```

Which acts the same as our model:

$$\begin{aligned} no_building \in A_s \iff & s_{timer_building} = 0 \wedge \neg s_{waiting_building} \wedge \\ & \neg(s_{waiting_research} \wedge s_{action_research} = null) \wedge \\ & \neg(s_{waiting_shipyards} \wedge s_{action_shipyards} = null) \wedge \\ & \neg(A_{req_research_s} = \emptyset \wedge A_{req_shipyards_s} = \emptyset) \wedge \\ & \neg(s_{waiting_research} \wedge s_{action_research} = null \wedge A_{req_shipyards_s} = \emptyset) \wedge \\ & \neg(s_{waiting_shipyards} \wedge s_{action_shipyards} = null \wedge A_{req_research_s} = \emptyset) \end{aligned}$$

Note that we will hide the no_op action from the agent in the simulator, because as mentioned earlier: it serves no purpose for the agent.

3.3. OgameFormulas class

This class implements the formula functions defined in Appendix C. It also implements the peculiar rounding behaviour mentioned in that appendix. This was done by checking the output of the formulas, under different inputs, and comparing them to the numbers provided by a test account on Ogame.

3.4. Simulator class

The simulator class implements the five methods a reinforcement learning algorithm needs: Reset, GetStateSignal, GetPossibleActions, DoAction and GetReward. The GetPossibleActions method basically shifts through the AllActions list and filters out the unavailable actions:

```
public List<Action> GetPossibleActions()
{
    return Action.AllActions.Where(a => a.IsAvailable(CurrentState))
        .ToList();
}
```

The GetReward method implements the reward function:

```
public int GetReward()
{
    if (CurrentState.IsTerminal())
        return 1000000 - (CurrentState.TimeCurrent -
            CurrentState.TimePreviousAfterstate);
    return (CurrentState.TimeCurrent -
        CurrentState.TimePreviousAfterstate) * -1;
}
```

Which is equivalent to the reward function in our model:

$$R(s) = \begin{cases} 1,000,000 - x & \text{if } s \text{ is a goal state} \\ -x & \text{otherwise} \end{cases}$$

Where x is the number of seconds passed since the previous (parent) state.

3.5. UserTerminal class

This class is a simple user terminal, which can be used to play Ogame on the simulator. It can be used as a visual aid in presentations about this thesis. Figure 3.1 gives a graphical impression.

3.6. Computation time tweaks

When experimenting with the simulator, it turned out that first few state transitions are fast, but it quickly became very slow. The no_op action, which calls the NextSecond method in the State class, seemed to be the problem. Actions in Ogame quickly start to take thousands of seconds and running the no_op action thousands of times eats up a lot of CPU cycles.

The solution to this problem is the following: in the model it is possible to calculate how long it takes to reach the next afterstate. Using the formula functions in Appendix C, we can calculate how long actions take. We can also calculate how long waiting actions take, by using the cost and production functions. With this information we can implement a new method that jumps forward in time not one, but x seconds. In effect we remove all the wasted operations of continuously performing the NextSecond method. To achieve this, three new methods were implemented: TimeUntilNextEvent, NextAfterstate and NextX.

3. Building the environment

```
Time: 0

Resources:
Metal: 500      Crystal: 500      Deuterium: 0

Production:
Metal: 30      Crystal: 15      Deuterium: 0
Factor: 0

Levels:
MetalMine: 0    CrystalMine: 0    DeuteriumSynthesizer: 0
SolarPlant: 0    RoboticsFactory: 0    CrystalStorage: 0
MetalStorage: 0    DeuteriumTank: 0    ResearchLab: 0
Shipyard: 0    EnergyTechnology: 0    ImpulseDrive: 0
ColonyShip: 0

Timers:
Building: 0      Research: 0      Shipyard: 0

Waitingmode:
Building: False  Research: False  Shipyard: False

Current action:
Building: None   Research: None   Shipyard: None

=====

Available actions
0. MetalMine
1. CrystalMine
2. DeuteriumSynthesizer
3. SolarPlant
4. CrystalStorage
5. MetalStorage
6. DeuteriumTank

Pick an action, type its number and press enter:
0

Executed action: 0. MetalMine

Reward of the afterstate: -18
```

Figure 3.1.: A demonstration of the user terminal

`TimeUntilNextEvent` calculates how many seconds it takes to get to the next event. An event occurs whenever one of the assembly lines finishes an action or finished waiting for an action. Under this definition, if the agent is in an afterstate, at least one event will trigger. `TimeUntilNextEvent` might return 0, which means several events happened at the same moment in time. This might occur when several assembly lines finish waiting at the same moment. We mentioned in the previous chapter the Ogame MDP does not implement the behaviour to start several actions at the same moment. Using this event system, we implemented it again.

The `NextX` method is an improved `NextSecond` method (and replaces it), which can jump x seconds, instead of just one. The method only works if x is zero or higher.

The `NextAfterstate` method combines `TimeUntilNextEvent` and `NextX`: it is used

3. Building the environment

by the simulator after the DoAction method, to automatically jump to the next after-state. The method works as follows:

```
public void NextAfterstate()
{
    int numPossibleActions =
        Action.AllActions.Count(a => a.IsAvailable(this));
    int Time = TimeUntilNextEvent();

    while (Time == 0 || numPossibleActions == 0)
    {
        NextX(Time);
        numPossibleActions = Action.AllActions.
            Count(a => a.IsAvailable(this));
        Time = TimeUntilNextEvent();
    }
}
```

The method comes down to looping NextX(Time) as long as we have no serious action to choose (corresponds to our afterstates) or there is an event at this moment.

Another computation time tweak that was implemented was reducing the amount of calls to the ProductionFactor method in the OgameFormulas class. It turned out this method got called quite often, even though this value does not change in the time between events. So we opted to save the production factor in a variable in the State class and only update it in a call to NextX.

The current implementation runs at a speed of around 2000 learning episodes per hour on one core of an AMD Phenom II X3 720 CPU, running at 2.8 GHz. This includes the Q-learning algorithm and a neural network with one hidden layer of 102 hidden neurons (see the next chapter).

There probably remain many small or large tweaks to the computation time of the simulator code, but we simply cannot explore and implement them due to time constraints.

3.7. A note on Semi Markov Decision Processes

The TimeUntilNextEvent method, introduced in the previous section, looks very similar to the holding time function in Semi Markov Decision Processes (SMDP). Some of the earliest research on SMDPs was done by Jewell (1963). A more recent approach can be found in Chen & Lu (2010). SMDPs are standard MDPs, but with one alteration. In standard MDPs, when an action is chosen, the transition to a new state is performed instantaneously. In SMDPs, the transition is not instant, but has a certain holding time. The holding time is usually calculated with some probability distribution function. After the holding time passed, the process completes the transition and a new action can be chosen.

The similarity suggest that in future research, it might be interesting to explore SMDPs for the PACG problem, where the holding time function could be replaced by the TimeUntilNextEvent method.

3.8. Simulator robustness

Due to

- transferring our model to a programming language,
- implementing some behaviour our MDP does not have,
- making a few tweaks to lower the computation time,
- and our model not being proved 100% error free yet,

there is a chance our simulator is not error free. To make sure our model and simulator is robust, we let a simple agent pick random actions on the simulator, for a certain amount of time. If no errors, for example deadlocks, show up, then we consider the simulator error free. For this simulator we implemented the simple agent and let it pick random actions. Sometimes we stumbled upon a deadlock or an error. These turned out not to be programming errors, but shortcomings in the Ogame model. For example, we forgot certain constraints on the no-wait actions. After fixing these errors in the simulator and retroactively the Ogame model, the agent played the simulator for 16 hours and 20 minutes error free. We now consider the current implementation of the simulator error free. Note that the random action agent runs a lot faster than 2000 episodes an hour, due to the absence of Q-learning and the neural network.

3.9. Research product two

We now have a simulator for both the toy and Ogame MDP. Research question two, “Can we build a piece of software that represents that model, so that we can run reinforcement learning algorithms on it?”, can now be answered with yes by providing the Ogame simulator.

4. Building the learning agent

Now that we have an Ogame simulator that acts as the environment, we can start building the learning agent. The agent will use a Q-learning algorithm. Because of the complexity of the Ogame problem, the learning agent will need some additional components to learn effectively. This chapter will introduce Q-learning in Section 4.1. Afterwards, additional components/algorithms are introduced. Section 4.2 shows a simple exploration policy, while Section 4.3 explains how a large state space can be handled with generalization functions. Finally, Section 4.4 introduces experience replay, which can help with the data-inefficiency of Q-learning.

4.1. Q-learning

One of the best known and most used reinforcement learning algorithms is Q-learning, which was introduced by Watkins (1989). It is designed to find the optimal policy in an MDP. It does this by learning an action-value function, Q , continuously increasing its estimate by exploring the MDP. It has some nice advantages. For example, it is proven to converge to the optimal policy if the value function is implemented in a tabular format: “We show 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” Watkins & Dayan (1992). Q-learning is off-policy, which means that it is a lot less sensitive to how the environment is explored, as opposed to an on-policy algorithm, as long as every action is explored a significant amount of times.

Q-learning implements a state-action value function:

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

This formula defines the value function: what are the expected rewards when starting in state s , performing action a , and then continue taking actions according to (exploration) policy π . It includes the immediate reward of performing action a , but also the values of possible next state-action pairs. The values of the next state-action pairs are weighted by a discount factor γ . In other words: if $0 < \gamma < 1$, the rewards in the far future are discounted more than rewards in the close future. An agent with $\gamma = 0$ is called myopic: it is only interested in immediate rewards. For episodic learning problems, discounting is not necessary and γ can be set to 1, which means future rewards are fully accounted for in a state-action value.

Note that for our problem it is not exactly clear how to set our discount factor. In episodic tasks it can be set to 1, but our learning problem floats in a grey area between episodic and non-episodic. It can take close to infinity¹ steps to reach a terminal state: the agent can infinitely pick actions that do not “close the gap” to the goal state,

¹Note that the Ogame simulator is capped to 50,000,000 simulated seconds, which is still incredibly long for a learning problem.

4. Building the learning agent

depending on the exploration policy. Our best bet is to experiment with $\gamma = 1$ and values very close to 1.

The idea of Q-learning is to start with arbitrarily initialized estimate Q-values. By experiencing the environment and using our own estimate Q-values, we can continuously update these estimates, until they eventually approach the true Q-values. The method of starting with estimates and continuously improving them is called bootstrapping. The Q-value updates, for a state transition from s to s' with action a , are done via the following update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

Note that Q-values $Q(s, a)$ where s is a terminal state are initialized and kept at 0. Otherwise, the non-zero value would constantly be added to Q-values prior to state s . These prior Q-values then continuously increase in value and this inflation removes our guarantee at convergence. This effect especially breaks Q-learning in combination with a generalization function. See the next section for more about generalization.

The pseudo-code for Q-learning can be found in Algorithm 1. The algorithm uses

Algorithm 1 Q-learning, source: Sutton & Barto (1998)[Chapter 6.5]

```
Initialize  $Q(s, a)$  arbitrarily
repeat(for each episode):
    Choose  $a$  from  $s$  using policy derived from  $Q$ 
    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
```

a parameter, the learning rate α . The learning rate decides how big the Q-value updates should be. Note that the agent learns nothing when $\alpha = 0$. To converge to an optimal policy, the agent also has to decrease the learning rate appropriately, hitting 0 at the “sweet spot”. Knowing when to stop learning is hard though, in this thesis we will use constant α values, for example $\alpha = 0.05$. Future research could improve upon this.

4.2. Exploration policy

The exploration policy determines the agent’s exploration behaviour. The agent’s goal is to gather a lot of rewards, to do this, he needs to exploit known good actions. But to find these good actions, he has to explore. Thus he needs a good balance between these two aspects.

One of the simplest exploration policies is ϵ -greedy exploration. The policy generally follows the value function, which means it will usually pick the action which has the highest Q-value for state s . With a chance of ϵ however, it chooses a completely random action in A_s . The parameter ϵ can be altered, to change the agent’s balance between exploring and exploiting. There are many more exploration methods, for example one that explores each action in the state space a minimum amount of times.

4.3. The value function and generalization

When the state space of a learning problem is large, the value function cannot be implemented in a tabular format. The table grows too big to fit in a computer’s memory. The value function can however be approximated with generalization functions, for example artificial neural networks. Another advantage is that generalization functions can group similar states and actions together in the value function. This effect can be beneficial for the learning process.

The downside is that when using a generalization function, we have lost our guarantee at convergence. With Q-learning we are learning from estimates. If the initial estimate is bad, the following estimate might be bad too. When using generalization this can lead to very bad estimates or even divergence. This makes finding the right parameters, for learning successfully, harder.

When implementing Q-learning with a tabular Q-value representation, it worked for the toy MDP, but not for the Ogame MDP. It turns out that the state space is too large. When capped to 50,000,000 simulated seconds, it does not fit, by large, in the memory of a conventional desktop computer (4GB). Because of this we have to resort to approximating the value function using a generalization function.

4.3.1. Artificial neural networks

Our choice is to use artificial neural networks (ANN) in combination with a backpropagation learning algorithm. They are easy to use and many implementations can be found on the internet. In this thesis we use the NeuronDotNet 3.0 library¹. Neural networks are quite simple at the core and if they work, they are a fast learning tool (depending on the amount of neurons). Unfortunately, if they do not work or not well, it is hard to reason about them, due to their nature.

An ANN consists of units (neurons) connected by directed links, Russel & Norvig (2003)[Chapter 20.5]. If a neuron gets activated, it propagates its activation through its output links. Links also have a weight associated with them, which alters the strength and/or the sign of the connection. The input value of a neuron is calculated by summing the weighted values of the input links. Formally, for links from neurons j with activation a_j to neurons i :

$$in_i = \sum_{j=1}^n W_{j,i} a_j$$

This input value is then given to an activation function (g), which then determines if the neuron is “active” (near +1) or “inactive” (near 0):

$$a_i = g(in_i) = g\left(\sum_{j=1}^n W_{j,i} a_j\right)$$

A feed forward ANN structures its neurons in three types of layers: one input, one output and zero or more hidden layers. Each neuron in a single layer is usually connected via a link to every neuron in the next layer: a one-to-many connection. If we pretend the ANN is a simple function, the input layer usually has one input neuron

¹The library can be found at <http://sourceforge.net/projects/neurondotnet/>

4. Building the learning agent

for every parameter the function takes. An ANN usually has one output neuron, which represents the output of the function. Note that if an ANN uses zero hidden layers and one output neuron, it becomes a linear function. For most purposes one hidden layer performs best, so we will consider only one hidden layer.

An ANN can learn by using a backpropagation algorithm. When given an input and a corresponding output, the algorithm can run the input through the network and then calculate the error between the output of the network and what the output should be. Formally, with input x , network n_W with weights W and true output y :

$$Err = y - n_W(x)$$

Then the weights of the links in the network can be updated, starting with the links to the hidden neurons to the output neuron:

$$W_{j,i} \leftarrow W_{j,i} + \alpha * a_j * Err * g'(in_i)$$

Where g' is the derivative of g . Afterwards, the error is backpropagated through the links of the input neurons to the hidden neurons, because they are also responsible for a fraction of Err . The weight of the link between the hidden neurons and output neuron decides their contribution to the error:

$$W_{k,j} \leftarrow W_{k,j} + \alpha * a_k * g'(in_j) * \sum_i W_{j,i} * Err * g'(in_i)$$

The backpropagation algorithm can also be extended to more than one output neuron and more than one hidden layer.

The weights in the network can be initialized in any way the user wants. For our problem we will initialize the weights with a normalized random function.

An ANN as value function generalization can be implemented in two ways. We can create one network, give it a state action pair (s, a) and then the value that it should learn for that pair. The other way is creating multiple networks: one for each action. We then give the network, corresponding to action a , the state s and the value that it should learn for that state. Since neural networks use neuron inputs, both ways introduce a new parameter: a mapping function from state signal to neuron inputs. The former method also needs a mapping from actions to neuron inputs. For our implementation we will use the latter method, with multiple networks. We reason that with multiple networks, specialized in one action, we will have an easier time finding the right mapping function.

The disadvantage, when choosing the multiple network method, is that we lose generalization over state-action pairs. For example: the network might learn that a certain group of actions is good for a certain group of states. This is impossible in the multiple network method, because they can only learn if a group of states is good for their particular action.

With both methods however, the mapping function largely decides how well the network will generalize the value function and therefore how well our algorithm will learn.

Q-learning with artificial neural networks

Note that the Q-learning algorithm changes slightly when used with an ANN. Recall the Q-update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

4. Building the learning agent

This update rule assumed we used a tabular format to save the Q-values. Since we are now using an ANN with backpropagation, we have a new update target. The update rule for Q-learning with an ANN is:

$$Q(s, a) \leftarrow_{BP(\alpha)} r + \gamma \max_{a'} Q(s', a')$$

In this update rule we are telling the network to update its estimate of $Q(s, a)$ using the reward and previous estimate. The update is done by the backpropagation algorithm under learning rate α .

Mapping states to input neurons

As mentioned earlier, the way in which we present an ANN our state information largely decides how well our algorithm will learn. Unfortunately designing a good mapping requires extensive knowledge of how an ANN reacts to different input structures and also a lot of time to experiment with different mappings, we claim to have neither of them. Nevertheless, we can make a mapping inspired on other scientific literature.

A very successful Q-learning algorithm with an ANN is TD-Gammon of Tesauro (1994), mentioned in the introduction of this thesis. Inspired by this algorithm, we came up a mapping function, which can be found in Figure 4.1.

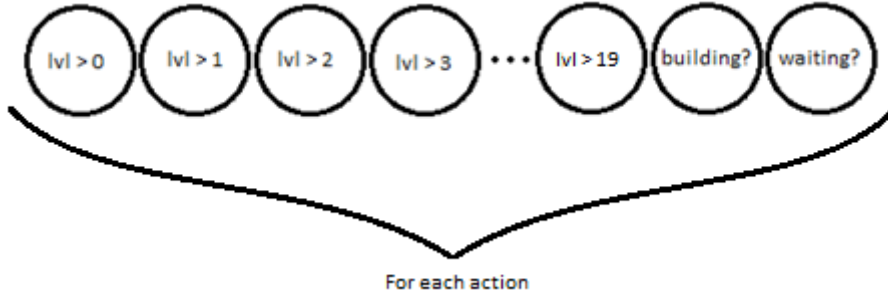


Figure 4.1.: State information to neuron inputs mapping function

For every action in the model, except no.op, we create a few inputs to represent how high the level of the corresponding building or research is in the state. Through experience with Ogame, we know that while reaching for the goal state with a reasonable policy, no action should become above level 19 (which includes a lot of wiggle room). We will add nineteen inputs to represent this. For example if we have an action of level 0, we have nineteen times a 0 as input. If we have an action of level 1, we have a 1 as input, followed by eighteen times a 0. Besides that we will add two inputs, which represent if the current action is being executed (1 or 0) or if the action is in waiting mode(1 or 0).

This neuron input structure captures almost all the state variables, the only variables not encoded are the resources and timers. These variables are left out, because they are not necessary for the agent: the agent has access to wait actions, making the resources variables unnecessary, and timers are trivialized due to the afterstate system.

Note: the toy model simulator uses the same mapping structure, except it uses only three inputs for the levels of the actions.

4.4. Experience replay

When using Q-learning in combination with neural networks, the networks tend to need a lot of updates to learn. Besides that, according to Adam et al. (2012), Q-learning is data inefficient: “classical Q-learning and SARSA algorithms, which are indeed computationally efficient, but are data inefficient: They use every sample once, to incrementally improve the solution, after which they discard the sample”.

For every learning update we want to do, we have to make calls to the simulator. Our simulator is not optimized for speed yet, which makes the data inefficiency even worse. It takes relatively more time to run the simulator, than the learning algorithm. This is where experience replay comes into play. Experience replay (ER) stores the experience samples we gain by choosing actions and then repeatedly presents this experience to the learning algorithm. “This increases data efficiency, while exploiting the computational efficiency of the underlying algorithm.” Adam et al. (2012). The experience replay algorithm has many parameters that can be altered:

- The number of experience samples stored.
- Replaying experience samples can be done randomly, temporal or backwards temporal.
- The number of replays can change, in other words, one experience sample can be replayed ten or maybe a thirty times.

The original experience replay was introduced by Lin (1992), which used backwards temporal ER. One of his observations was: “Experience replay can be more effective in propagating credit/blame if a sequence of experiences is replayed in temporally backward order.” This method seems the most promising for the Ogame problem, because the big reward is given at the end of an episode. The specifics of a good ER method for the Ogame problem would be:

- Backwards temporal.
- For every episode played on the simulator, we will replay the whole episode R amount of times. R is a new parameter for the learning algorithm, which can alter the success of the learning agent.
- After every episode the experience database is cleared.

4.5. Research product three

We have implemented the above mentioned algorithms and additional components (Q-learning, ANN, mapping function) for both simulators. Note that ER has only been implemented for the toy simulator so far. The learning algorithm for the Ogame simulator is included in the source code in Appendix D.

We can now answer research question three, “Can we implement Q-learning in such a way that it can learn a policy for playing Ogame?”, by providing this algorithm stack. Though, learning any policy is a trivial task: if we initialize our implementation with an ANN with random weights, we have a first estimate of the Q-value function and already “learned” a policy (see Section 1.4.2). The next chapter will focus on finding good policies, which is all besides trivial.

5. Experiments and results

The remainder of this thesis will try to find an answer to the main research question: “Can reinforcement learning find a good policy for playing Ogame?” First we will define what a good policy is in Section 5.1. We will continue with several experiments, trying to learn a good or even optimal policy, on the toy and Ogame simulator in Section 5.2 and Section 5.3 respectively.

5.1. Defining a good policy

Recall from Section 2.3.5 that a transition path from start to goal state, using a simple action selection policy, takes 837,629 seconds. We will define that a learned policy is good, if following that policy from a start state leads to a goal state in 837,629 seconds or less.

Note that this is already a pretty strict definition of a good policy: the learning agent should be at least as good as a human that made up a simple policy, even though we have no idea yet how our algorithm performs and we know that Q-learning with an ANN is not guaranteed to converge.

5.2. Toy MDP experiments

To get insight in how reinforcement learning behaves with simple PACG problems, with respect to different parameters, we will start with some experiments on the toy MDP.

5.2.1. Value iteration experiment

Recall that if we follow the optimal policy for the toy model, we have a transition path that takes 34 simulated seconds. Before we start with reinforcement learning, we will first implement an algorithm called value iteration. It is an algorithm that calculates an optimal policy for a finite MDP, given a maximum error value and discount rate, by calculating the values of all the states. For more information, see Russel & Norvig (2003)[Chapter 17.2].

While reinforcement learning learns a value function, value iteration calculates it by brute force. With this algorithm we can check if it is possible to find the 34 second policy under different discount rate values. In effect, it gives us an idea in what range our discount rate parameter should be when learning.

Since the value iteration algorithm requires a finite MDP, we capped the state space to 100 seconds for this experiment. This is more than enough to reach the goal state, but also small enough for the value function to be saved in tabular format. We set the maximum error for the calculations to a significantly low value: 0.001. The graph in Figure 5.1 contains the result for this experiment.

5. Experiments and results



Figure 5.1.: Optimal policy / discount rate, calculated with value iteration and 0.001 maximum error

Note that we can draw a rough trend line over this graph, in the form of a parabola. The discount rate determines the horizon of the agent: how much it values future rewards. The parabola shows the horizon's influence on the agent. With a short horizon the agent comes up with a decent 38 second policy. When we let it consider long term rewards a bit more, the resulting policy becomes worse. Eventually when the agent values long term rewards enough, the result is better and better. The parabola form is interesting: we expected some sort of linear decrease, from bad to better policy.

From this experiment we can conclude that the learning agent has to be very interested in long term rewards. A discount rate of 0.97 or higher is needed to learn the optimal policy. In the Ogame MDP it takes far longer to find the goal state, so the agent might have to be even more “farsighted”.

5.2.2. TD(0)-learning versus Q-learning experiment

As a second experiment we wanted to check if Q-learning is indeed a good choice for our problem. To this end we implemented both TD(0)-learning and Q-learning in a tabular format. TD(0)-learning is an on-policy learning algorithm which learns V-values instead of Q-values. While Q-learning learns the values of a state-action pair, TD(0)-learning learns the values of states, by combining the values for all the actions that are possible in that state. This makes TD(0)-learning on-policy, because the way we explore affects the V-value we learn. A second difference is that for TD(0)-learning we need to know the transition function, while Q-learning also works if we do not know it. We decided to run a learning test, to compare the two algorithms, with the following parameters:

- Learning rate $\alpha = 0.05$
- Discount rate $\gamma = 0.97$
- ϵ -greedy exploration rate = 0.1
- Number of learning episodes: 10,000

Since we now introduce randomness, due to ϵ -greedy exploration, we will run these tests ten times for both algorithms. The graph in Figure 5.2 contains the results.

5. Experiments and results

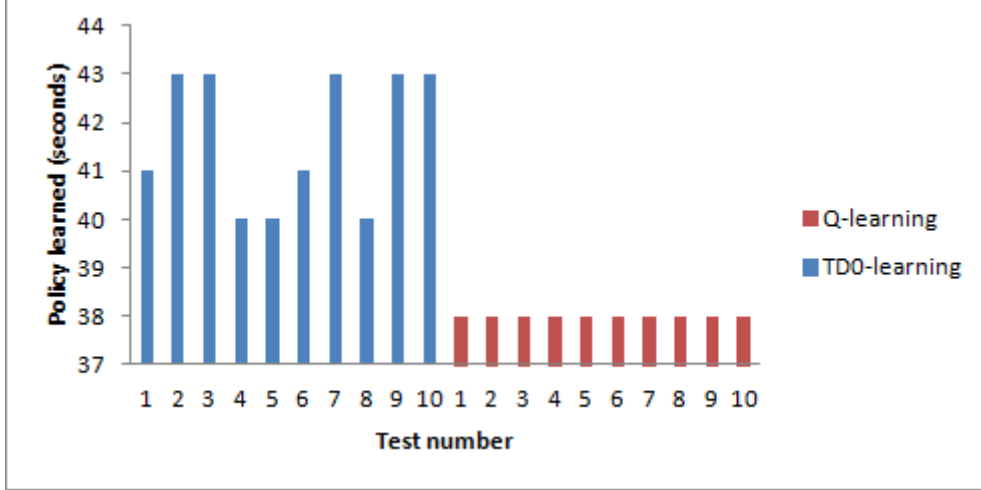


Figure 5.2.: TD(0)-learning versus Q-learning, toy MDP. $\alpha = 0.05$, $\gamma = 0.97$, $\epsilon = 0.1$ and 10,000 learning cycles.

Q-learning performs consistent by producing 38 second policies. TD(0)-learning produces a mean 41.7 second policy, with a standard deviation of around 1.35. The consistent and inconsistent performances are due to on-policy or off-policy learning. TD(0)-learning could possibly perform better with a different exploration function. For our problem, we will stick to Q-learning, since it seems to produce better and more consistent policies with ϵ -greedy exploration.

Note that in these tests neither of the two algorithms produced the 34 second policy. This is most likely caused due to the constant 0.1 ϵ -greedy exploration. If there is a constant chance of performing random actions and the learning rate is not reduced to 0, the Q-values will never converge to their true values. The next section is an attempt to find the optimal policy.

5.2.3. Learning the optimal policy with tabular Q-learning

As mentioned, Q-learning is guaranteed to converge when implemented in a tabular form and α is reduced appropriately. We want to find an optimal policy, to confirm that there is not something wrong with our MDP. It is hard to hit the sweet spot when reducing α , because it has to become 0 the moment when the optimal policy is learned. For the toy MDP we know this policy is the 34 seconds policy. For the Ogame MDP, and PACG problems in general, we do not know the optimal policy: we are trying to learn it.

Because of this, we will try to find the optimal policy while leaving α constant. We might not converge to the true Q-values, but we might come close enough that our learned Q-values produce the optimal policy. Since the constant random exploration is the most likely cause of not finding the optimal policy, we will try a custom exploration policy.

5. Experiments and results

Logarithmic ϵ -greedy exploration policy

First we will try ϵ -greedy exploration, but instead of a constant ϵ , we will use a slowly decreasing one. We want ϵ to decrease as a function of the episode number, as shown in Figure 5.3.

The used logarithmic function is:

$$\epsilon = 1 / (1 + \frac{e^{\frac{x}{0.09N}}}{50})$$

Where x is the current episode number, N is the total number of episodes and e is Euler's number.

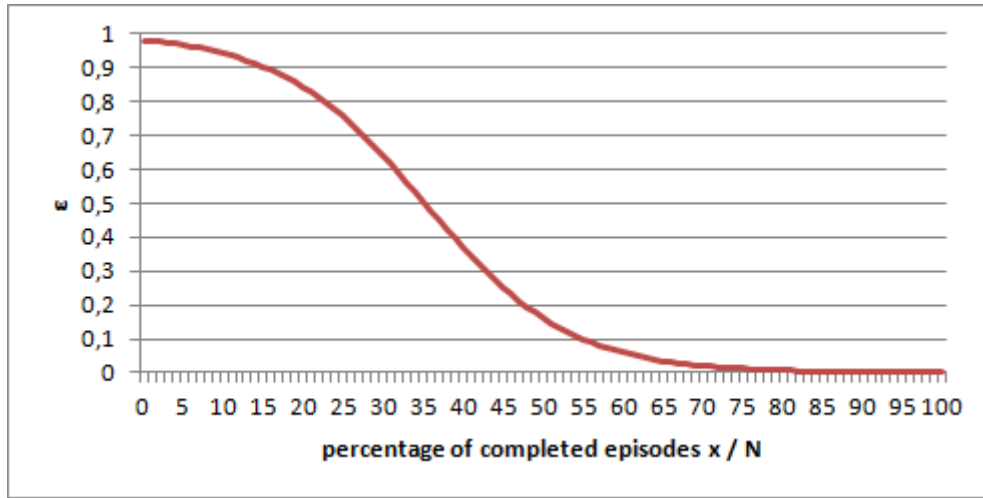


Figure 5.3.: ϵ as a function of x/N in percents

Using this logarithmic function, the agent will start the learning process with a lot of exploring. Gradually the exploration becomes less and exploitation increases. When ϵ approaches 0, the agent almost exclusively exploits. When exploiting, the estimates of the exploited Q-values should become better and better estimates. Eventually, the estimated Q-values should come very close to the true Q-values. We then ran some tests with the following parameters:

- Learning rate $\alpha = 0.05$
- Discount rate $\gamma = 0.97$
- Logarithmic ϵ -greedy exploration as defined above.
- Number of learning episodes: 350,000

It turns out that the Q-learning algorithm almost always learns the optimal policy with these parameters, which can be seen in Figure 5.4. In 100 tests, Q-learning learned the optimal policy 96 times. The 4 times it did not, it came very close by producing a 35 second policy. When increasing the number of learning episodes to 500,000, we give the algorithm even more exploration and exploitation time. Suddenly the algorithm consistently¹ learns the optimal policy, as can be seen in Figure 5.5.

¹Learning the optimal policy consistently is interpreted as learning it 100 times out of 100 tests.

5. Experiments and results

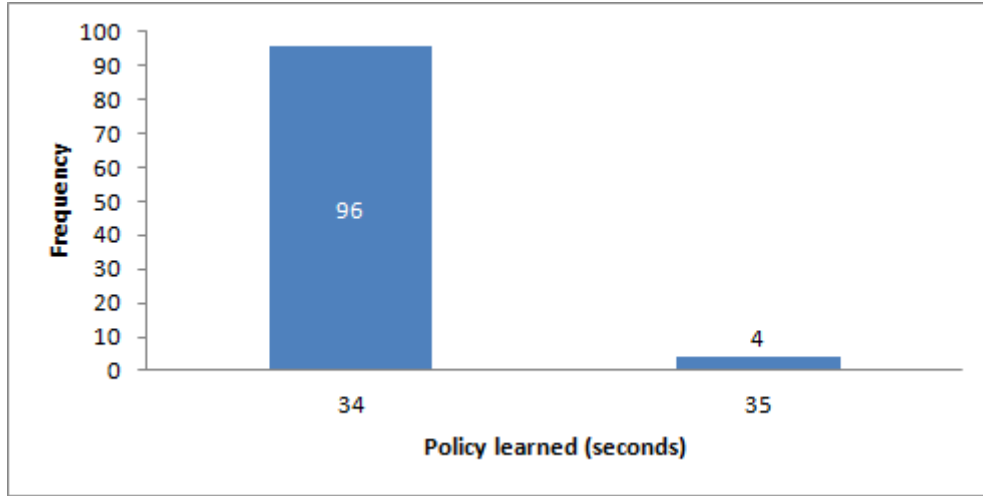


Figure 5.4.: 100 learned policies with logarithmic ϵ -greedy exploration, Q-learning, toy MDP. $\alpha = 0.05, \gamma = 0.97$ and 350,000 learning episodes.

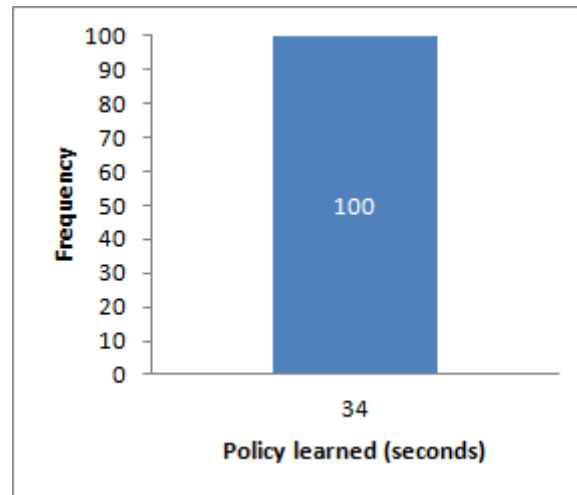


Figure 5.5.: 100 learned policies with logarithmic ϵ -greedy exploration, Q-learning, toy MDP. $\alpha = 0.05, \gamma = 0.97$ and 500,000 learning episodes.

5. Experiments and results

Apparently, the PACG problem needs a lot of early exploration plus a lot of late exploitation, when using ϵ -greedy exploration. This sounds logical when going to a more abstract level: the agent needs to discover how the assembly lines work together and how to exploit them as best as he can. From these experiments we can also observe that a simple PACG problem, like the toy MDP, needs a large amount of learning episodes to learn the optimal policy with logarithmic ϵ -greedy exploration. The observation that 500,000 learning episodes are needed to learn the optimal policy consistently is alarming.

Random exploration policy

Besides logarithmic ϵ -greedy exploration we want to try a completely random exploration policy. This policy simply picks a random action in A_s every time. When exploring every state-action pair infinitely many times, the estimated Q-values start to approach the true Q-values, even if the learning rate is not decreased to 0. Because we have to visit every state-action pair a lot of times, we can expect to need even more learning episodes to find the optimal policy.

Under the same parameters as in the last section it turns out we need about 1,000,000 learning episodes to consistently learn the optimal policy, which the graph in Figure 5.6 shows.

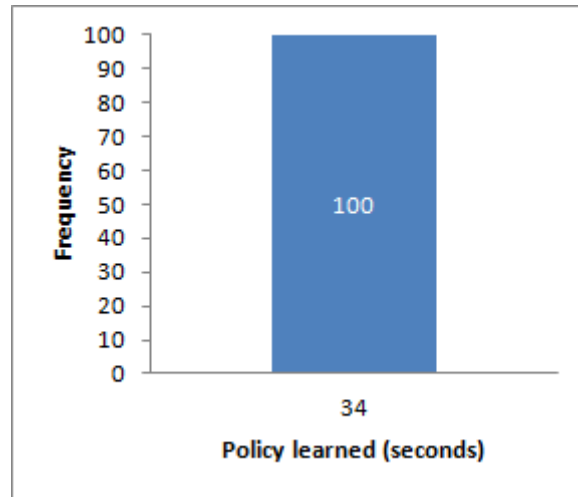


Figure 5.6.: 100 learned policies with random exploration, Q-learning, toy MDP. $\alpha = 0.05, \gamma = 0.97$ and 1,000,000 learning episodes.

Conclusion: Q-learning solves simple PACG problems

The toy problem is a very simple PACG problem. By modelling it as an MDP and subsequently learning the optimal policy with Q-learning, we can conclude the following: Q-learning can learn the optimal policy for some simple PACG problems.

5.2.4. Full learning algorithm on toy simulator

The full learning algorithm stack (Q-learning, ANNs, ER), as shown in Chapter 4, is meant for more complex PACG problems. To get some insight in how it works with a simple PACG, we will test it on the toy MDP.

When experimenting with the full algorithm, the parameters we can consider are:

- Learning rate α
- Discount rate γ
- Exploration policy
- Number of neurons in the hidden layer N_h
- Number of experience replays per learning episode R
- Number of learning episodes E
- Mapping function of state signal to neuron inputs

The range of possible combinations of parameters is huge, asking for a lot of experiments. On top of that, neural networks introduce another level of randomness, besides exploration, due to random initialized weights. Unfortunately, performing tests with the full algorithm stack is time consuming: running 10,000 learning episodes, with a hidden layer of 6 neurons, takes about 15 seconds computing time on the previous mentioned system specifications. Increasing the number of hidden neurons or learning episodes takes even more computing time. Due to time constraints we cannot fully research each of these parameters and their combined interactions. Instead, we will pick a base parameter setting and explore the various parameters from there on. Some of these base parameters were determined in previous sections, some were determined in pilot experiments.

Base parameter settings

The base parameter settings we will use are:

- Learning rate $\alpha = 0.05$
- Discount rate $\gamma = 0.97$
- ϵ -greedy exploration rate, $\epsilon = 0.1$
- Number of hidden neurons $N_h = 6$
- Number of experience replays $R = 1$
- Number of learning episodes $E = 50,000$
- Mapping function as defined in Section 4.3.1

We tested our learning algorithm 100 times with these settings and Figure 5.7 shows the resulting learned policies.

The base parameter settings seem to produce mostly mediocre policies of 40 or 41 seconds. It turns out the algorithm learned the optimal policy once. The chance of learning this policy might even be lower than 1%, if the number of tests is increased.

The following sections will show for every parameter how it influences the learning agent.

5. Experiments and results

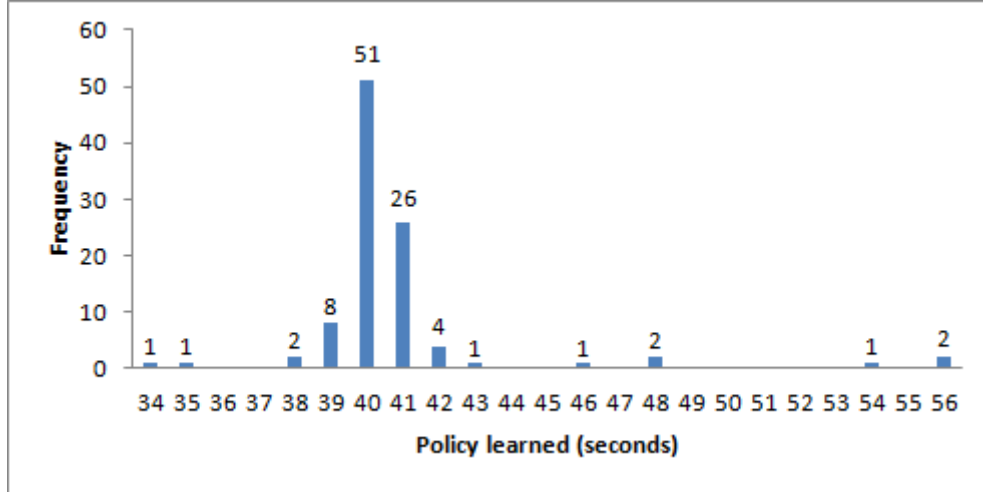


Figure 5.7.: 100 learned policies through full learning algorithm stack, toy MDP. Base parameter settings.

Learning rate α

When working with neural networks, it is best to keep α low. The learning rate determines how large the learning updates are. When it is high and the ANN is given an update value, the network makes a big step towards that update value. If the next update value is a completely different value for a similar state, the error value (calculated by the backpropagation algorithm) is big. The next update is even bigger due to this large error value. If this process goes on, an oscillating behaviour can emerge and the Q-values will diverge, never reaching any meaningful values. To avoid this, the learning rate is best kept low. Another solution is adding momentum to the backpropagation algorithm, which reduces oscillation by considering previous changes in weights during updates, but this is outside the scope of this thesis.

Keeping the learning rate low assumes that the number of learning episodes is high. If it is low, maybe due to limited computation time, it is possible that a higher learning rate is beneficial: a lucky exploration of good actions is learned faster, though it has to avoid oscillating to do so. The next experiment used the base parameters, but with $\alpha = 0.1$ and the results can be found in Figure 5.8.

Our expectations are quite visible: a few lucky explorations let the agent learn the 34 and 35 second policies. A few unlucky explorations also made the agent learn policies in the 45 to 50 seconds range.

Note that the agent also learned policies of 126 and 130 seconds, which are products of oscillation. In the toy simulator the state space is limited to 100 simulated seconds. In effect, any state after 100 seconds is considered terminal. When following a policy that has not reached a goal state in time, it is cut short after the first transition that crosses the 100 second line. The 130 second policy is the result of constantly choosing two actions: MetalMine and no_wait_research. The Q-values for these actions (if not all) have diverged, creating a policy which constantly picks these actions.

Compared to the base test, the doubled learning rate resulted in a few better policies, but more worse ones. In addition, it even failed 6 out of 100 times due to oscillation.

5. Experiments and results

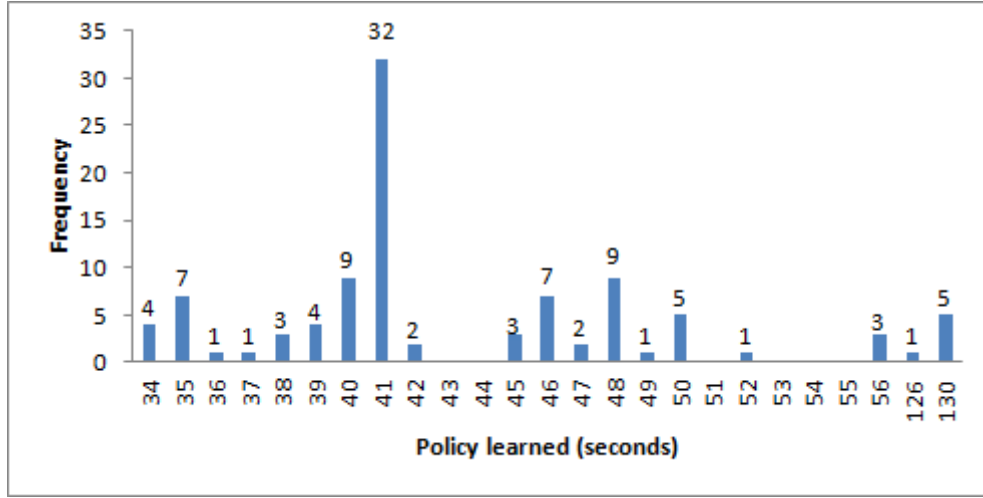


Figure 5.8.: 100 learned policies, full algorithm, toy MDP. Base parameter settings, $\alpha = 0.1$. Note that the policies above 100 seconds are products of oscillation.

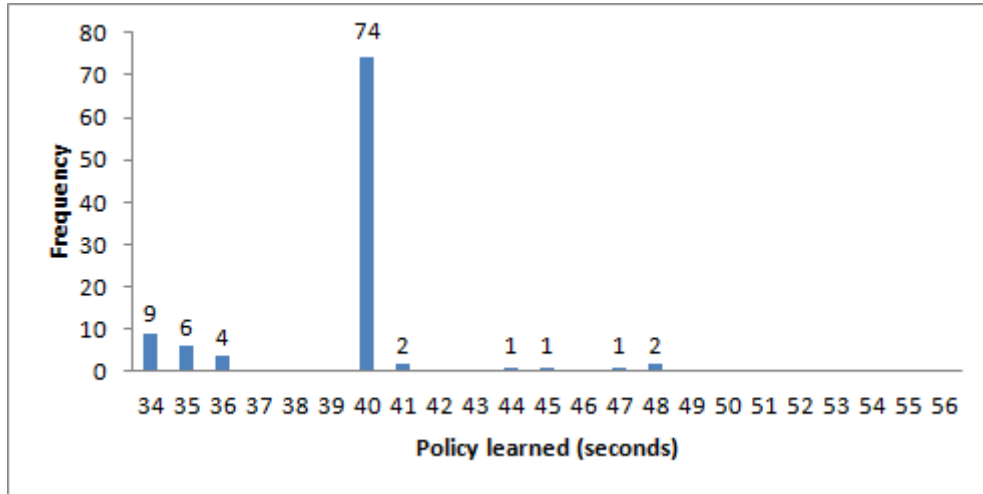


Figure 5.9.: 100 learned policies, full algorithm, toy MDP. Base parameter settings, $\alpha = 0.01$

5. Experiments and results

For the next experiment we lowered the learning rate to $\alpha = 0.01$, which resulted in Figure 5.9.

Compared to the base test, lowering the learning rate resulted in a better performing agent. The agent learns more good policies in the 34 to 36 second range and never learns really bad policies. Oscillation is completely avoided. The smaller but steady Q-value updates seem to work well with neural networks.

On a final note: learning rate and number of learning episodes tend to go hand in hand. A low learning rate is beneficial to avoid oscillating. Recall that when $\alpha = 0$, the agent learns nothing. When lowering the learning rate and thus learning slower, the number of learning episodes should generally be increased, because more learning is needed. This falls in line with common knowledge that neural networks require lots of learning episodes.

Discount rate γ

Recall from Section 4.1 that we want to test $\gamma = 1$. A Q-value should then take into account all the possible future rewards, undiscounted. This could be beneficial for the toy and Ogame problem, because the big reward is given at the end. A decent goal state can be reached in 837,629 simulated seconds. While there are fewer transitions, we can imagine that it is high. When discounting the big reward every transition, even with a discount rate of 0.97 it becomes unnoticeable in the Q-values for the states early in the transition path. For example with a $\gamma = 0.97$ and a big reward is reached after 100 transitions, it is counted for $0.97^{100} \approx 0.048$, which is very low for an estimate based learning method. The following test used the base parameters, but with $\gamma = 1$, and the result can be found in Figure 5.10.

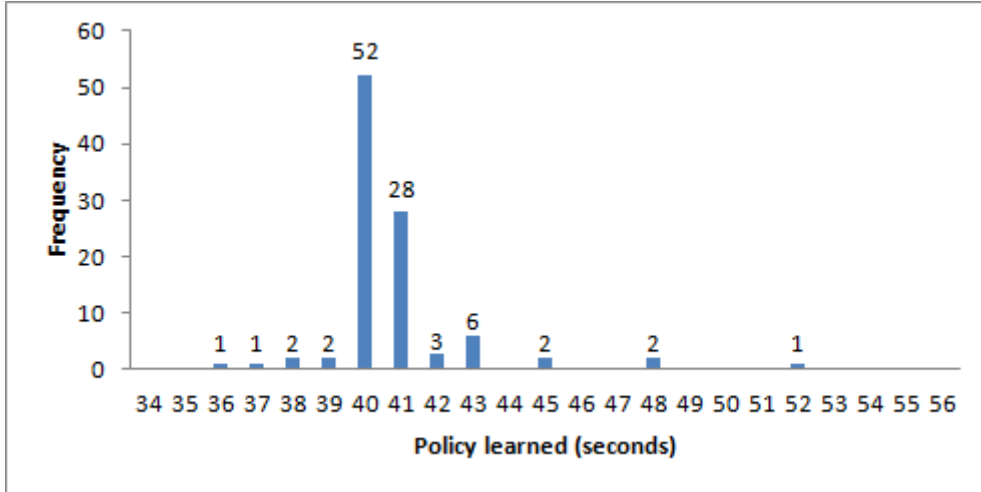


Figure 5.10.: 100 learned policies, full algorithm, toy MDP. Base parameter settings, $\gamma = 1$

Compared to the base test, the frequency distribution of learned policies is almost identical. Using $\gamma = 1$, the agent learned less good policies, but also less bad policies. This could also be caused by the randomness of exploration and the relatively small experiment pool of 100 tests.

5. Experiments and results

In the Ogame MDP, the goal state, on average, is several times farther from the starting state. These results suggest that it is a good idea to use $\gamma = 1$ on the Ogame MDP. In the worst case, results do not change much. In the best case it could help learning greatly, by fully counting the big reward at the end in the early Q-values.

Exploration policy

Section 5.2.3 suggested that, for PACG problems, ϵ -greedy exploration is inferior to logarithmic ϵ -greedy exploration. In the next experiment we used the base parameters, but changed the exploration policy to the logarithmic ϵ -greedy method. The result is shown in the graph in Figure 5.11.

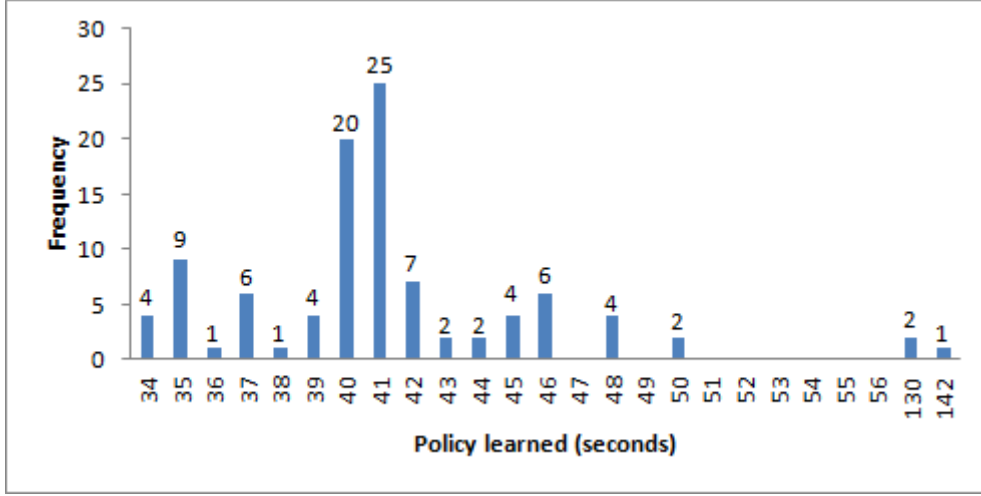


Figure 5.11.: 100 learned policies, full algorithm, toy MDP. Base parameter settings, logarithmic ϵ -greedy exploration. Note that the policies above 100 seconds are products of oscillation.

This exploration method resulted in a much larger frequency spread than the base test. It spread evenly to better and worse policies. Note that three times the networks failed and oscillated. This oscillation and the even spread is possibly caused by the large amount of random exploration at the start, because we are showing the neural networks completely different experience samples in rapid succession.

Since this exploration method, with tabular Q-learning, seemed to work better with an increased number of learning episodes, we ran an additional experiment with 100,000 episodes. This is double the amount of the base experiment. Figure 5.12 is the result.

From the figure we can conclude that doubling the number of learning episodes means worse performance. The algorithm failed seven times by oscillating, while the frequency spread is almost even. In the previous experiment we speculated that the failures are caused by the high exploration at the start of the learning process. When increasing the number of learning episodes, we also increase the number of these completely random experience samples given to the neural networks. This could possibly increase the chance of oscillation and explain the increase in failures.

5. Experiments and results

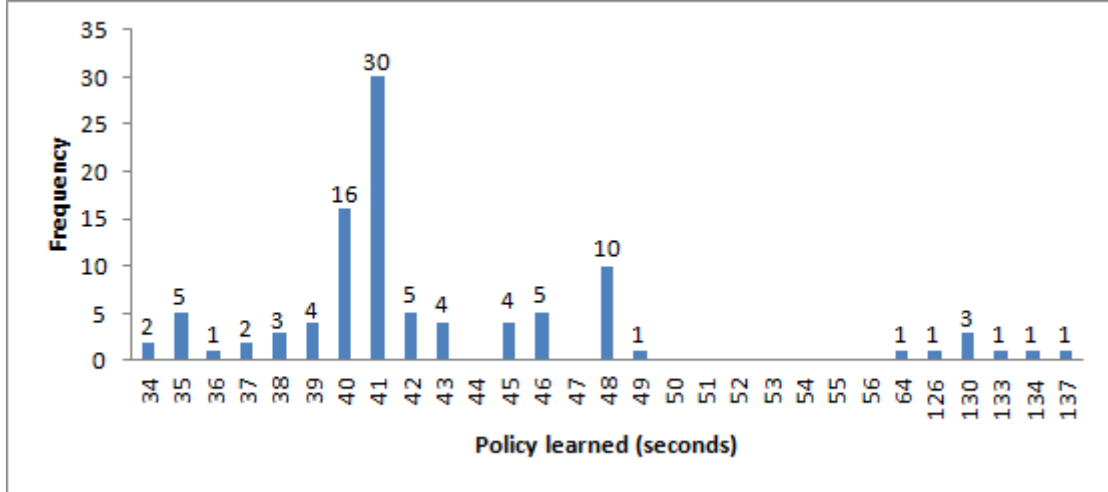


Figure 5.12.: 100 learned policies, full algorithm, toy MDP. Base parameter settings, logarithmic ϵ -greedy exploration, 100,000 learning episodes. Note that the policies above 100 seconds are products of oscillation.

To confirm our suspicion, we ran another test, but this time changing the upper limit of the logarithmic function, that determines ϵ , to 0.1 instead of 1. The greatly reduced initial exploring should prevent oscillating. However, we should still get the logarithmic ϵ -greedy exploration's advantage of better convergence. The result of this experiment is found in Figure 5.13.

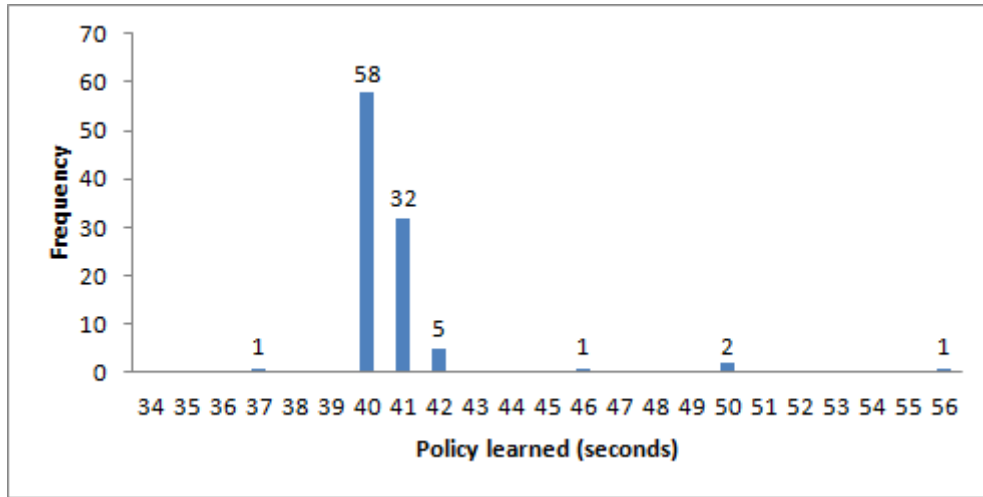


Figure 5.13.: 100 learned policies, full algorithm, toy MDP. Base parameter settings, logarithmic ϵ -greedy exploration with 0.1 upper limit.

Changing the upper limit of ϵ to 0.1 results in a more stable agent. Oscillation failures are absent, confirming the suspicion that they are caused by very high exploration. The frequency distribution is quite similar to the base experiment. Unfortunately, the agent also lost the good policies it found by the high exploration.

In the end, the change to a logarithmic function for ϵ does not seem a great

5. Experiments and results

improvement. However, Section 5.2.3 suggests that this exploration policy performs better when coupled with a higher number of learning episodes. This will be explored further later in this section.

Other parameters

The remaining parameters to be experimented with are:

- Number of neurons in the hidden layer N_h
- Mapping function of state signal to neuron inputs
- Number of experience replays per learning episode R
- Number of learning episodes E

Due to time constraints, we cannot include these experiments in this thesis. This is unfortunate, because changing the amount of hidden neurons or the mapping function can have a large impact on learning.

From experiments in the previous sections we can however speculate that for consistently learning the optimal policy in the toy MDP, with the full algorithm, the number of learning episodes E has to be in the range of 50,000 to 1,000,000. For reference: Q-learning needed 500,000 episodes to do this. ANN Q-learning might need less due to generalization, but it might also need more episodes due to ANNs tendency to need lots of updates. For the Ogame MDP, this range probably has to be multiplied with a number around ten to cope with the increased complexity.

The number of experience replays parameter R does not have to be considered for the agent's performance: increasing it has a similar effect as increasing the number of learning episodes. Both methods generate more experience samples, and they do so in a different way. Increasing E does generate "new" samples, but since the MDP is deterministic and the reward function is constant, these new samples are always from the same set of experience samples. The increasing R method simply takes a subset of these samples and duplicates them. In the end, both methods are fishing in the same experience sample pool. Thus, when enough episodes are played, the effect of increasing either parameter is the same. However, computation time of both methods might differ. This depends on the computation time of the simulator and experience replays. Increasing the number of replays is preferred with a relatively slow simulator.

Consistently learning the optimal policy with tabular Q-learning

In the previous experiments we have seen that the full algorithm stack occasionally produces the optimal policy for the toy MDP. This suggests that there is a set of parameters which lets the algorithm consistently learn the optimal policy. We hope to find this parameter set by combining the insights of the previous sections. Based on the previous experiments, we will try the following parameter set:

- Learning rate $\alpha = 0.01$
- Discount rate $\gamma = 1$
- Logarithmic ϵ -greedy exploration rate with an upper limit to ϵ of 0.3
- Number of learning episodes $E = 500,000$
- The other parameters are equal to the base experiment.

These settings resulted in the graph in Figure 5.14.

5. Experiments and results

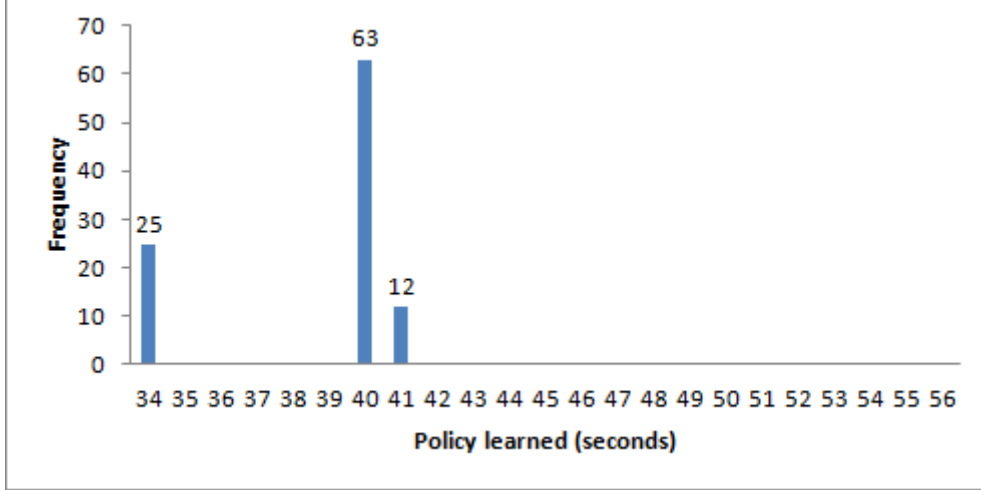


Figure 5.14.: 100 learned policies, full algorithm, toy MDP. Base parameter settings, logarithmic ϵ -greedy exploration with 0.1 upper limit, $\alpha = 0.01$, $\gamma = 1$ and 500,000 learning episodes.

These results show the best agent until now. In 25 out of 100 tests, or 25%, it learned the optimal policy. This is a vast improvement over all the previous experiments. It strongly suggests that a certain parameter set can lead to an agent that learns the optimal policy in all (or nearly all) of the learning tests. However, finding this optimal parameter set is very time consuming, especially when the number of learning episodes are increased. Future research could focus on finding these parameter sets.

5.3. Ogame MDP experiments

We have shown that the ANN, Q-learning and ER algorithm stack, meant for complex PACG problems, works on simple PACG problems. Next, we want to know if it also works with the more complex PACG problems, like our Ogame problem.

In Section 3.6 it was mentioned that the Ogame simulator, in combination with the full algorithm stack, runs at 2,000 learning episodes per hour. This is too slow to conduct extensive experiments, especially when millions of learning episodes are needed. The computation time of the simulator and/or learning algorithm has to be decreased before research is possible. This is not possible in the limited time frame of this thesis, but future research could focus on this.

We speculate that it is possible to find good policies for the Ogame MDP, using the proposed algorithm stack. There are many possible parameter configurations. In the previous sections we have determined possible values for several parameters:

- Learning rate should be low: $\alpha = 0.01$.
- Discount rate should be $\gamma = 1$.
- Logarithmic ϵ -greedy exploration with some low upper limit to ϵ , to avoid oscillation.
- Number of learning episodes E should be in the range of 500,000 to 10,000,000, also depending on the exploration policy.

5. Experiments and results

If alternating these parameters does not provide satisfying results, the other parameters, like the mapping function, could also be altered.

Another interesting suggestion would be to use a custom exploration policy that uses information that we know of the Ogame problem. We have shown a simple transition path in the Ogame MDP, which was used to define a good policy. This transition path was made by following some simple rules, for example:

- If the production factor is lower then 1, build a SolarPlant.
- If the MetalMine level is equal to the CrystalMine level, build MetalMine.
- If the Crystalmine level is equal to the DeuteriumSynthesizer level, build CrystalMine.
- Build DeuteriumSyntesizer.
- Etcetera...

Following these simple rules from top to bottom at every action selection, it becomes a policy. This policy could be used as the initial exploration policy. The agent could explore 1,000 learning episodes with this policy, and then continue with a logarithmic ϵ -greedy exploration policy. Using information that we already know of the problem and giving it to the agent, might drastically improve performance.

6. Thesis conclusion

The main research question this thesis wanted to answer is: “Can reinforcement learning find a good policy for playing Ogame?” We cannot answer with a definite yes or no. We can however say that it is more than likely.

We have shown that reinforcement learning can consistently learn the optimal policy for simple PACG problems, like the toy Markov Decision Process (MDP) introduced in this thesis. We also introduced a Q-learning algorithm with Artificial Neural Networks (ANNs) and Experience Replay (ER), meant for more complex PACG problems. Consecutively, we have shown that this algorithm can learn the optimal policy on the toy MDP. The best performance so far was an experiment that showed that in 25% of the learning cases, the algorithm learned the optimal policy. Experiment results suggest that the right parameter set can increase this performance.

Due to the results of the toy MDP, we speculate that it is highly likely to learn good policies for the Ogame MDP. Unfortunately, due to an inefficient Ogame simulator and limited time frame, this thesis cannot confirm this yet.

Future research should focus on optimizing the Ogame simulator’s computation time and searching for a parameter set which produces good policies.

Bibliography

- Adam, S., Buşoniu, L., Babuška, R., Experience Replay for Real-Time Reinforcement Learning Control. *IEEE Transactions on Systems, Man, and Cybernetics - part C: Applications and Reviews* 42(2), 201–212 (2012)
- Bellman, R.: A Markovian Decision Process. *Journal of Mathematics and Mechanics* 6(4), 679–684 (1957)
- Chen, T., Lu, J.: Towards Analysis of Semi-Markov Decision Processes. *Artificial Intelligence and Computational Intelligence* 6319, 41–48. Springer, Berlin Heidelberg (2010)
- Jewell, W.S.: Markov-renewal programming I: Formulation, finite return models; markov-renewal programming II: infinite return models, example. *Operations Research* 11, 938–971 (1963)
- Lin, L.: Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *Machine Learning* 8(3), 293–321 (1992)
- Puterman, M.L.: *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York (1994)
- Russel, S., Norvig, P., *Artificial Intelligence: A Modern Approach*. Pearson Education, New Jersey (2003)
- Sutton, R.S., Barto, A.G., *Reinforcement Learning: An Introduction*. MIT Press, Cambridge (1998)
- Tesauro, G. J.: TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation* 6(2), 215–219 (1994)
- Watkins, C.J.C.H.: *Learning from delayed rewards*. PhD thesis, Kings College, Cambridge (1989)
- Watkins, C.J.C.H., Dayan, P.: Q-learning. *Machine Learning* 8(3/4), 279–292 (1992)
- Wiering, M., Otterlo, M. van: *Reinforcement Learning, State of the Art*. Springer-Verlag, Berlin Heidelberg (2012)

A. Actions in Ogame

The following summarisation of actions in Ogame was compiled by going through all the actions a player can perform in Ogame. Some actions, that are completely trivial for reaching the goal state, were left out of this list. An example of a trivial action: changing a planet name.

The following sections will explain, for each group of actions, why they are in the Ogame MDP or left out.

Production setting actions

In Ogame, the player can set the energy production and energy consumption levels of certain buildings. The setting can be changed from 0% to 100%, with steps of 10%. The buildings affected are:

- For energy consumption:
 - Metal Mine
 - Crystal Mine
 - Deuterium Synthesizer
- For energy production:
 - Solar Plant
 - Fusion Reactor
 - Solar Satellite

So in total there are six buildings, which can be set to eleven different energy levels. Six times eleven is thirty-three actions available (if the corresponding building was constructed). Besides increasing the number of available actions with large degree, and therefore exploding the state space, these actions are not really effective. Changing the energy production from the default 100% to something lower gives a negative effect in resource production and thus serves no purpose at all. If a human player has no reason to take these actions at all, why should our agent be bothered with these actions at all? Changing the energy consumption settings almost always gives a negative effect on resource production.

The only situation in which this can give a positive effect, is when total energy production is lower than total energy consumption. This means there is a production factor lower than 100%, which influences the resource production. By changing the energy levels of certain resource production buildings, the player can put more energy into producing one resource, then the others. This effect is really small, but nevertheless there. If the player builds more energy production facilities, these actions become trivial again.

We omit these actions in the model, because these actions are almost always trivial and increase the state space by a significant large amount. Bothering the agent with these actions will make it very hard to learn.

A. Actions in Ogame

Building construction actions

Some of these actions are mandatory to reach the goal state, some will let the agent reach the goal faster and some are trivial. Table A.1 lists all non-trivial building construction actions, if they are included or excluded in the Ogame MDP and the reason why.

Building name	Included?	Reason
Metal Mine	Yes	Producing more metal can increase speed.
Crystal Mine	Yes	Producing more crystal can increase speed.
Deuterium Synthesizer	Mandatory	Deuterium production is needed for research and Colony Ship.
Solar Plant	Mandatory	Energy is needed for production, see Deuterium Synthesizer.
Fusion Reactor	No	Alternative means to produce energy, but regarded by player base as an useless action. Could be added in future model/research.
Robotics Factory	Mandatory	Needed to build Shipyard, but also increases building construction speed.
Nanite Factory	No	While drastically increasing building construction speed, it lies far deeper in the state space than the goal state, thus making this action trivial.
Shipyard	Mandatory	Needed to build Colony Ship.
Metal Storage	Yes	Increases the amount of metal that can be stored. Could potentially increase speed by opening up new scenarios, while it slightly increases the state space.
Crystal Storage	Mandatory	Needed to build Colony Ship, also: see Metal Storage.
Deuterium Tank	Yes	See Metal Storage.
Research Lab	Mandatory	Needed to conduct research, see next section.
Terraformer	No	Even deeper in the state space than Nanite Factory, while having almost trivial effect.

Table A.1.: Building actions in the Ogame MDP

Besides these actions there are some trivial ones that have no effect on reaching the goal state:

- Alliance Depot
- Lunar Base
- Sensor Phalanx
- Jump Gate
- Missile Silo

Because these are trivial they will be excluded from the model.

A. Actions in Ogame

Research actions

The player can conduct research in the Research Lab. Of these actions, only two are needed for reaching the goal state:

- Energy Technology
- Impulse Drive
 - Needs Energy Technology

These two actions will be added to the MDP. There are several other technologies a player can research, but they are trivial, and thus not included.

Shipyard actions

The player can build space ships in the Shipyard. Of these actions, only one is needed for reaching the goal state:

- Colony Ship
 - Needs Impulse Drive

This action will be added to the model. There are several other space ships a player can build, but they are trivial for reaching the goal state.

Other actions

We now have all the non-trivial actions we want for the model. There are several other categories of actions, but these will not be included. Table A.2 lists all these categories and why they are excluded of the model.

A. Actions in Ogame

Category of actions	Reason why they are excluded from the model
Expeditions	The player can gain the ability to embark on expeditions. This usually is usually deeper into the state space then our goal state. It would also make the model incredibly complex: the player can send expeditions of ships to every system, exploding the state space.
Merchant	The merchant can be used to trade resources. These actions require dark matter, which can only be acquired by two methods. One: spending real life money, unfortunately our learning agent has no money. Two: performing in game expeditions, see Expeditions.
Defence Construction	Since we omitted other players and our agent is alone, building defences is useless.
Fleet	Since we omitted other players, we cannot engage in fleet actions.
Galaxy	The galaxy page gives a player a view of the systems in the universe, and who occupies the planets in them. Since there are no other players, we have no use for these actions.
Alliance	Since we omitted other players, there will be no alliances. Creating an alliance for oneself is trivial for reaching the goal state.
Recruiting Officers	A player can recruit officers, which can give several positive effects, like increased energy or resource production. These actions cost dark matter, and are therefore not included. See Merchant.

Table A.2.: Other, excluded, actions in Ogame

A.1. List of actions included in the MDP

Summarizing the above sections, the list of all Ogame actions that will be included in the model is:

Metal Mine, Crystal Mine, Deuterium Synthesizer, Solar Plant,
Robotics Factory, Research Lab, Shipyard,
Crystal Storage, Metal Storage, Deuterium Tank,
Energy Technology, Impulse Drive,
Colony Ship

B. Toy model additions

B.1. Functions for toy model actions

level	metal production / sec	cost in metal	build time in seconds
0	1	-	-
1	2	4	4
2	3	8	8
3	4	16	16
4	5	32	32
\vdots	\vdots	\vdots	\vdots

Table B.1.: Functions for the MetalMine action

level	cost in metal	research time in seconds
1	2	2
2	4	4
3	8	8
4	16	16
\vdots	\vdots	\vdots

Table B.2.: Functions for the LaserTechnology action

level	cost in metal
1	0
2	0
3	0
\vdots	\vdots

Table B.3.: Functions for the no_op action

B.2. Transition tables

The following tables contain state transition paths from starting state s_0 to goal state s_g , for different versions of the toy MDP. Table B.4 contains the transition path for toy MDP version 1. Table B.5 adds afterstates to the toy MDP. Finally, table B.6 contains the optimal transition path for toy MDP version 2. For the sake of layout, the actions MetalMine and LaserTechnology are shortened to MM and LT. The no_wait_build and no_wait_research actions are shortened to no_b and no_r. Coloured cells and bold text is used to show when an action leads to its intended result.

For the transition table B.4 the action selection is: pick the left most action from A_s . The action selection in the transition table B.5 is a bit different: at a certain point we chose no_op actions instead of the left most, to make the transitions in this table equivalent to the transition in table B.5.

Transition table B.4 has 47 transitions. Introducing afterstates for transition table B.5 reduces this to 18 transitions, following an equivalent transition path.

Table B.6 contains the optimal transition path in toy model version 2, which is quiet easy to find by hand. The action MetalMine is the only action which can speed up the process, because it makes resource production higher. To reach the goal state we only need MetalMine and LaserTechnology both on level 3. Building MetalMine 3 first (while doing no_wait_research) and then LaserTechnology 3 gives the highest resource production and thus the least amount of time wasted. Getting from start to goal state via the optimal path takes 34 simulated seconds. One can verify this by recalling the reward function:

- -1 per second
- +50 for reaching the goal

Summing up the rewards (16) and then subtracting this from 50 gives 34 seconds.

B. Toy model additions

Table B.4.: State transition table for toy MDP version 1

State transition	Available actions in s	Reward in s
$\langle 0, 0, 0, 0, 0 \rangle$	no_op	
$(\langle 0, 0, 0, 0, 0 \rangle, \text{no_op}) \rightarrow \langle 1, 0, 0, 0, 0 \rangle$	no_op	-1
$(\langle 1, 0, 0, 0, 0 \rangle, \text{no_op}) \rightarrow \langle 2, 0, 0, 0, 0 \rangle$	LT, no_op	-1
$(\langle 2, 0, 0, 0, 0 \rangle, \text{LT}) \rightarrow \langle 0, 0, 0, 0, 2 \rangle$	no_op	0
$(\langle 0, 0, 0, 0, 2 \rangle, \text{no_op}) \rightarrow \langle 1, 0, 0, 0, 1 \rangle$	no_op	-1
$(\langle 1, 0, 0, 0, 1 \rangle, \text{no_op}) \rightarrow \langle 2, 0, 1, 0, 0 \rangle$	no_op	-1
$(\langle 2, 0, 1, 0, 0 \rangle, \text{no_op}) \rightarrow \langle 3, 0, 1, 0, 0 \rangle$	no_op	-1
$(\langle 3, 0, 1, 0, 0 \rangle, \text{no_op}) \rightarrow \langle 4, 0, 1, 0, 0 \rangle$	MM, LT, no_op	-1
$(\langle 4, 0, 1, 0, 0 \rangle, \text{MM}) \rightarrow \langle 0, 0, 1, 4, 0 \rangle$	no_op	0
$(\langle 0, 0, 1, 4, 0 \rangle, \text{no_op}) \rightarrow \langle 1, 0, 1, 3, 0 \rangle$	no_op	-1
$(\langle 1, 0, 1, 3, 0 \rangle, \text{no_op}) \rightarrow \langle 2, 0, 1, 2, 0 \rangle$	no_op	-1
$(\langle 2, 0, 1, 2, 0 \rangle, \text{no_op}) \rightarrow \langle 3, 0, 1, 1, 0 \rangle$	no_op	-1
$(\langle 3, 0, 1, 1, 0 \rangle, \text{no_op}) \rightarrow \langle 4, 1, 1, 0, 0 \rangle$	LT, no_op	-1
$(\langle 4, 1, 1, 0, 0 \rangle, \text{LT}) \rightarrow \langle 0, 1, 1, 0, 4 \rangle$	no_op	0
$(\langle 0, 1, 1, 0, 4 \rangle, \text{no_op}) \rightarrow \langle 2, 1, 1, 0, 3 \rangle$	no_op	-1
$(\langle 2, 1, 1, 0, 3 \rangle, \text{no_op}) \rightarrow \langle 4, 1, 1, 0, 2 \rangle$	no_op	-1
$(\langle 4, 1, 1, 0, 2 \rangle, \text{no_op}) \rightarrow \langle 6, 1, 1, 0, 1 \rangle$	no_op	-1
$(\langle 6, 1, 1, 0, 1 \rangle, \text{no_op}) \rightarrow \langle 8, 1, 2, 0, 0 \rangle$	MM, LT, no_op	-1
$(\langle 8, 1, 2, 0, 0 \rangle, \text{MM}) \rightarrow \langle 0, 1, 2, 8, 0 \rangle$	no_op	0
$(\langle 0, 1, 2, 8, 0 \rangle, \text{no_op}) \rightarrow \langle 2, 1, 2, 7, 0 \rangle$	no_op	-1
$(\langle 2, 1, 2, 7, 0 \rangle, \text{no_op}) \rightarrow \langle 4, 1, 2, 6, 0 \rangle$	no_op	-1
$(\langle 4, 1, 2, 6, 0 \rangle, \text{no_op}) \rightarrow \langle 6, 1, 2, 5, 0 \rangle$	no_op	-1
$(\langle 6, 1, 2, 5, 0 \rangle, \text{no_op}) \rightarrow \langle 8, 1, 2, 4, 0 \rangle$	LT, no_op	-1
$(\langle 8, 1, 2, 4, 0 \rangle, \text{LT}) \rightarrow \langle 0, 1, 2, 4, 8 \rangle$	no_op	0
$(\langle 0, 1, 2, 4, 8 \rangle, \text{no_op}) \rightarrow \langle 2, 1, 2, 3, 7 \rangle$	no_op	-1
$(\langle 2, 1, 2, 3, 7 \rangle, \text{no_op}) \rightarrow \langle 4, 1, 2, 2, 6 \rangle$	no_op	-1
$(\langle 4, 1, 2, 2, 6 \rangle, \text{no_op}) \rightarrow \langle 6, 1, 2, 1, 5 \rangle$	no_op	-1
$(\langle 6, 1, 2, 1, 5 \rangle, \text{no_op}) \rightarrow \langle 8, 2, 2, 0, 4 \rangle$	no_op	-1
$(\langle 8, 2, 2, 0, 4 \rangle, \text{no_op}) \rightarrow \langle 11, 2, 2, 0, 3 \rangle$	no_op	-1
$(\langle 11, 2, 2, 0, 3 \rangle, \text{no_op}) \rightarrow \langle 14, 2, 2, 0, 2 \rangle$	no_op	-1
$(\langle 14, 2, 2, 0, 2 \rangle, \text{no_op}) \rightarrow \langle 17, 2, 2, 0, 1 \rangle$	MM, no_op	-1
$(\langle 17, 2, 2, 0, 1 \rangle, \text{MM}) \rightarrow \langle 1, 2, 2, 16, 1 \rangle$	no_op	0
$(\langle 1, 2, 2, 16, 1 \rangle, \text{no_op}) \rightarrow \langle 4, 2, 3, 15, 0 \rangle$	no_op	-1
$(\langle 4, 2, 3, 15, 0 \rangle, \text{no_op}) \rightarrow \langle 7, 2, 3, 14, 0 \rangle$	no_op	-1
\vdots	\vdots	\vdots
$(\langle 13, 2, 3, 12, 0 \rangle, \text{no_op}) \rightarrow \langle 16, 2, 3, 11, 0 \rangle$	LT, no_op	-1
$(\langle 16, 2, 3, 11, 0 \rangle, \text{no_op}) \rightarrow \langle 19, 2, 3, 10, 0 \rangle$	LT, no_op	-1
\vdots	\vdots	\vdots
$(\langle 43, 2, 3, 2, 0 \rangle, \text{no_op}) \rightarrow \langle 46, 2, 3, 1, 0 \rangle$	LT, no_op	-1
$(\langle 46, 2, 3, 1, 0 \rangle, \text{no_op}) \rightarrow \langle 49, 3, 3, 0, 0 \rangle$	MM, LT, no_op	49

B. Toy model additions

Table B.5.: State transition table for toy MDP version 1 with afterstates

State transition	Available actions in s	Reward in s
$\langle 0, 0, 0, 0, 0 \rangle$	no_op	
$(\langle 0, 0, 0, 0, 0 \rangle, \text{no_op}) \rightarrow \langle 2, 0, 0, 0, 0 \rangle$	LT, no_op	-2
$(\langle 2, 0, 0, 0, 0 \rangle, \text{LT}) \rightarrow \langle 4, 0, 1, 0, 0 \rangle$	MM, LT, no_op	-4
$(\langle 4, 0, 1, 0, 0 \rangle, \text{MM}) \rightarrow \langle 4, 1, 1, 0, 0 \rangle$	LT, no_op	-4
$(\langle 4, 1, 1, 0, 0 \rangle, \text{LT}) \rightarrow \langle 8, 1, 2, 0, 0 \rangle$	MM, LT, no_op	-4
$(\langle 8, 1, 2, 0, 0 \rangle, \text{MM}) \rightarrow \langle 8, 1, 2, 4, 0 \rangle$	LT, no_op	-4
$(\langle 8, 1, 2, 4, 0 \rangle, \text{LT}) \rightarrow \langle 17, 2, 2, 0, 1 \rangle$	MM, no_op	-7
$(\langle 17, 2, 2, 0, 1 \rangle, \text{MM}) \rightarrow \langle 16, 2, 3, 11, 0 \rangle$	LT, no_op	-5
$(\langle 16, 2, 3, 11, 0 \rangle, \text{no_op}) \rightarrow \langle 19, 2, 3, 10, 0 \rangle$	LT, no_op	-1
\vdots	\vdots	\vdots
$(\langle 43, 2, 3, 2, 0 \rangle, \text{no_op}) \rightarrow \langle 46, 2, 3, 1, 0 \rangle$	LT, no_op	-1
$(\langle 46, 2, 3, 1, 0 \rangle, \text{no_op}) \rightarrow \langle 49, 3, 3, 0, 0 \rangle$	MM, LT, no_op	49

Table B.6.: State transition table for toy MDP version 2

State transition	Available actions in s	Reward in s
$\langle 0, 0, 0, 0, 0, \text{false}, \text{false}, \text{null}, \text{null} \rangle$	MM, LT, no_b, no_r, no_op	
$(\langle 0, 0, 0, 0, 0, \text{false}, \text{false}, \text{null}, \text{null} \rangle, \text{MM}) \rightarrow$	LT, no_r, no_op	0
$\langle 0, 0, 0, 0, 0, \text{true}, \text{false}, \text{MM}, \text{null} \rangle$		
$(\langle 0, 0, 0, 0, 0, \text{true}, \text{false}, \text{MM}, \text{null} \rangle, \text{no_r}) \rightarrow$	LT, no_r, no_op	-4
$\langle 0, 0, 0, 4, 0, \text{false}, \text{false}, \text{MM}, \text{null} \rangle$		
$(\langle 0, 0, 0, 4, 0, \text{false}, \text{false}, \text{MM}, \text{null} \rangle, \text{no_r}) \rightarrow$	MM, LT, no_b, no_r, no_op	-4
$\langle 4, 1, 0, 0, 0, \text{false}, \text{false}, \text{MM}, \text{null} \rangle$		
$(\langle 4, 1, 0, 0, 0, \text{false}, \text{false}, \text{MM}, \text{null} \rangle, \text{MM}) \rightarrow$	LT, no_r, no_op	0
$\langle 4, 1, 0, 0, 0, \text{true}, \text{false}, \text{MM}, \text{null} \rangle$		
$(\langle 4, 1, 0, 0, 0, \text{true}, \text{false}, \text{MM}, \text{null} \rangle, \text{no_r}) \rightarrow$	LT, no_r, no_op	-2
$\langle 0, 1, 0, 8, 0, \text{false}, \text{false}, \text{MM}, \text{null} \rangle$		
$(\langle 0, 1, 0, 8, 0, \text{false}, \text{false}, \text{MM}, \text{null} \rangle, \text{no_r}) \rightarrow$	MM, LT, no_b, no_r, no_op	-8
$\langle 16, 2, 0, 0, 0, \text{false}, \text{false}, \text{MM}, \text{null} \rangle$		
$(\langle 16, 2, 0, 0, 0, \text{false}, \text{false}, \text{MM}, \text{null} \rangle, \text{MM}) \rightarrow$	LT, no_r, no_op	0
$\langle 0, 2, 0, 16, 0, \text{false}, \text{false}, \text{MM}, \text{null} \rangle$		
$(\langle 0, 2, 0, 16, 0, \text{false}, \text{false}, \text{MM}, \text{null} \rangle, \text{LT}) \rightarrow$	LT, no_r, no_op	-3
$\langle 7, 2, 1, 13, 0, \text{false}, \text{false}, \text{MM}, \text{LT} \rangle$		
$(\langle 7, 2, 1, 13, 0, \text{false}, \text{false}, \text{MM}, \text{LT} \rangle, \text{LT}) \rightarrow$	LT, no_r, no_op	-4
$\langle 15, 2, 2, 9, 0, \text{false}, \text{false}, \text{MM}, \text{LT} \rangle$		
$(\langle 15, 2, 2, 9, 0, \text{false}, \text{false}, \text{MM}, \text{LT} \rangle, \text{LT}) \rightarrow$	LT, no_r, no_op	-8
$\langle 31, 2, 3, 1, 0, \text{false}, \text{false}, \text{MM}, \text{LT} \rangle$		
$(\langle 31, 2, 3, 1, 0, \text{false}, \text{false}, \text{MM}, \text{LT} \rangle, \text{no_r}) \rightarrow$	MM, LT, no_b, no_r, no_op	49
$\langle 34, 3, 3, 0, 0, \text{false}, \text{false}, \text{MM}, \text{null} \rangle$		

B.3. Toy MDP version 2

Notation / function	
$s_{variablename}$	Represents the value of the variable <i>variablename</i> in state <i>s</i> .
$(s, a) \rightarrow s'[var \leftarrow value]$	A state transition, where action <i>a</i> is executed in state <i>s</i> . This leads to state <i>s'</i> where the value of <i>var</i> is replaced with <i>value</i> . Note: this MDP uses the afterstate transition system introduced in Section 2.2.1.
<code>pint variablename</code>	A type indicator, this variable is a positive integer: 0, 1, 2, 3, ...
<code>bool variablename</code>	A type indicator, this variable is a boolean: <i>true</i> or <i>false</i> .
<code>action variablename</code>	A type indicator, this variable is an action $a \in A$
<code>cost(actionname, level)</code>	Returns the cost of the action <i>actionname</i> for the level <i>level</i> .
<code>time(actionname, level)</code>	Returns the duration (building or research) time of the action <i>actionname</i> for the level <i>level</i> .
<code>prod(actionname, level)</code>	Returns the resource production of the building <i>actionname</i> for the level <i>level</i> .

Table B.7.: Notations and functions for toy MDP version 2

The definition for the cost, time and prod functions can be found this appendix.

B.3.1. States

The set of states *S* is defined as:

$$s \in S \text{ if } s = \begin{array}{ll} < \text{pint res_metal,} \\ & \text{pint lvl_MetalMine,} \quad \text{pint lvl_LaserTechnology,} \\ & \text{pint timer_build,} \quad \text{pint timer_research,} \\ & \text{bool waiting_build,} \quad \text{bool waiting_research,} \\ & \text{action action_build,} \quad \text{action action_research} > \end{array}$$

Table B.8 explains what these variables represent.

Variable name	Explanation
<code>res_metal</code>	The amount of metal resource owned.
<code>lvl_MetalMine</code>	The current level of the Metal Mine building.
<code>lvl_LaserTechnology</code>	The current level of the Laser Technology research.
<code>timer_build</code>	Building time remaining in seconds.
<code>timer_research</code>	Research time remaining in seconds.
<code>waiting_build</code>	True if build assembly line is in waiting mode.
<code>waiting_research</code>	Idem for the research assembly line.
<code>action_build</code>	The action for which the build assembly line is waiting or performing.
<code>action_research</code>	Idem for the research assembly line.

Table B.8.: Legend for Toy MDP state variables

B. Toy model additions

Using this we can also define the starting state s_0 , which represents a new Ogame account:

$$s_0 = \langle 0, 0, 0, 0, 0, false, false, null, null \rangle$$

Note: *null* is the null-value for the action variables.

B.3.2. Actions

The set of actions A is defined as:

$$A = \{\text{MetalMine}, \text{LaserTechnology}, \text{no_wait_build}, \text{no_wait_research}, \text{no_op}\}$$

Preconditions

The set of actions A_s is defined as:

$$A_s \subseteq A$$

$$\begin{aligned} \text{MetalMine} \in A_s & \iff s_{\text{timer_build}} = 0 \wedge \neg s_{\text{waiting_build}} \\ \text{LaserTechnology} \in A_s & \iff s_{\text{timer_research}} = 0 \wedge \neg s_{\text{waiting_research}} \\ \text{no_wait_build} \in A_s & \iff s_{\text{timer_build}} = 0 \wedge \neg s_{\text{waiting_build}} \wedge \\ & \neg(s_{\text{waiting_research}} \wedge s_{\text{action_research}} = \text{null}) \\ \text{no_wait_research} \in A_s & \iff s_{\text{timer_research}} = 0 \wedge \neg s_{\text{waiting_research}} \wedge \\ & \neg(s_{\text{waiting_build}} \wedge s_{\text{action_build}} = \text{null}) \\ \text{no_op} \in A_s & \end{aligned}$$

Effects

$$\begin{aligned} (s, \text{MetalMine}) & \rightarrow \begin{cases} s' & [\text{res_metal} \leftarrow s_{\text{res_metal}} - \text{cost}(\text{MetalMine}, s_{\text{lvl_MetalMine}} + 1)] \\ & [\text{timer_build} \leftarrow \text{time}(\text{MetalMine}, s_{\text{lvl_MetalMine}} + 1)] \\ & \text{if } \text{cost}(\text{MetalMine}, s_{\text{lvl_MetalMine}} + 1) \leq s_{\text{res_metal}} \\ s' & [\text{action_build} \leftarrow \text{MetalMine}] \\ & [\text{waiting_build} \leftarrow \text{true}] \\ & \text{otherwise} \end{cases} \\ (s, \text{LaserTechnology}) & \rightarrow \begin{cases} s' & [\text{res_metal} \leftarrow s_{\text{res_metal}} - \text{cost}(\text{LaserTechnology}, s_{\text{lvl_LaserTechnology}} + 1)] \\ & [\text{timer_build} \leftarrow \text{time}(\text{LaserTechnology}, s_{\text{lvl_LaserTechnology}} + 1)] \\ & \text{if } \text{cost}(\text{LaserTechnology}, s_{\text{lvl_LaserTechnology}} + 1) \leq s_{\text{res_metal}} \\ s' & [\text{action_research} \leftarrow \text{LaserTechnology}] \\ & [\text{waiting_research} \leftarrow \text{true}] \\ & \text{otherwise} \end{cases} \end{aligned}$$

In other words: if there are enough resources, the actions behave the same way as in toy MDP version 1. However, if that is not the case, the corresponding assembly line is put in waiting mode for this action.

B. Toy model additions

$$(s, \text{no_wait_build}) \rightarrow s'[\text{action_build} \leftarrow \text{null}] \\ [\text{waiting_build} \leftarrow \text{true}]$$

$$(s, \text{no_wait_research}) \rightarrow s'[\text{action_research} \leftarrow \text{null}] \\ [\text{waiting_research} \leftarrow \text{true}]$$

The no-wait actions simply put the assembly lines in waiting-for-nothing mode.

Before we define the no_op action, we will create some boolean values. These booleans increase the readability of the no_op definition.

$$b_1 = s_{\text{waiting_build}} \wedge s_{\text{action_build}} \neq \text{null} \wedge \\ \text{cost}(s_{\text{action_build}}, s_{\text{lvl_action_build}} + 1) \leq \\ s_{\text{res_metal}} + \text{prod}(\text{MetalMine}, s_{\text{lvl_MetalMine}})$$

b_1 is *true* if the building assembly line can start building after gathering resources for some action.

$$b_2 = s_{\text{waiting_research}} \wedge s_{\text{action_research}} \neq \text{null} \wedge \\ \left(\neg b_1 \wedge \text{cost}(s_{\text{action_research}}, s_{\text{lvl_action_research}} + 1) \leq \right. \\ \left. s_{\text{res_metal}} + \text{prod}(\text{MetalMine}, s_{\text{lvl_MetalMine}}) \right) \\ \vee \\ b_1 \wedge \text{cost}(s_{\text{action_research}}, s_{\text{lvl_action_research}} + 1) + \\ \text{cost}(s_{\text{action_build}}, s_{\text{lvl_action_build}} + 1) \leq \\ s_{\text{res_metal}} + \text{prod}(\text{MetalMine}, s_{\text{lvl_MetalMine}}))$$

b_2 is *true* if the research assembly line can start researching after gathering resources for some action. Note that b_2 can be *true* even if b_1 is *true*, but it will be *false* if the building assembly line used up too much resources, so that the research action cannot start.

$$b'_2 = s_{\text{waiting_research}} \wedge s_{\text{action_research}} \neq \text{null} \wedge \neg b_1 \\ \wedge \text{cost}(s_{\text{action_research}}, s_{\text{lvl_action_research}} + 1) \leq \\ s_{\text{res_metal}} + \text{prod}(\text{MetalMine}, s_{\text{lvl_MetalMine}}))$$

b'_2 is *true* if the research assembly line can start researching and b_1 is *false*.

$$b_3 = b_1 \vee (s_{\text{waiting_build}} \wedge s_{\text{action_build}} = \text{null} \wedge (b_2 \vee s_{\text{timer_research}} = 1))$$

$$b_4 = b_2 \vee (s_{\text{waiting_research}} \wedge s_{\text{action_research}} = \text{null} \wedge (b_1 \vee s_{\text{timer_build}} = 1))$$

b_3 is *true* if the waiting mode on the building assembly line should be removed. b_4 does the same for the research assembly line.

B. Toy model additions

(s, no_op) → s'	[res_metal ← s _{res_metal} + prod(MetalMine, s _{lvl_MetalMine})]	
2	[lvl_MetalMine ← s _{lvl_MetalMine} + 1]	⇔ s _{timer_build} = 1
3	[lvl_LaserTechnology ← s _{lvl_LaserTechnology} + 1]	⇔ s _{timer_research} = 1
4	[timer_build ← s _{timer_build} - 1]	⇔ s _{timer_build} > 0
5	[timer_research ← s _{timer_research} - 1]	⇔ s _{timer_research} > 0
6	[res_metal ← s _{res_metal} + prod(MetalMine, s _{lvl_MetalMine}) -	
7	cost(s _{action_build} , s _{lvl_action_build} + 1)]	⇔ b ₁
8	[timer_build ← time(s _{action_build} , s _{lvl_action_build} + 1)]	⇔ b ₁
9	[waiting_build ← false]	⇔ b ₃
10	[res_metal ← s _{res_metal} + prod(MetalMine, s _{lvl_MetalMine}) -	
11	cost(s _{action_build} , s _{lvl_action_build} + 1) -	
12	cost(s _{action_research} , s _{lvl_action_research} + 1)]	⇔ b ₂
13	[res_metal ← s _{res_metal} + prod(MetalMine, s _{lvl_MetalMine}) -	
14	cost(s _{action_research} , s _{lvl_action_research} + 1)]	⇔ b' ₂
15	[timer_build ← time(s _{action_build} , s _{lvl_action_build} + 1)]	⇔ b ₂
16	[waiting_research ← false]	⇔ b ₄

Note that $s_{lvl_action_build}$ is shorthand for $s_{lvl_<action>}$, where $< action >$ is substituted with s_{action_build} . This was added to improve readability. The definition is split into three parts: the first one for basic administrative tasks and the second for the building assembly line. The last part is for the research assembly line. Also note that the res_metal variable can be substituted more than once: in rules 1, 6, 10 and 13. However, state s' contains only the latest substitution.

The no_op action still has the same function in this improved MDP: it advances simulated time in the model by one second. If an assembly line has finished waiting for resources, actions are automatically applied. This is done in rules 6-8 and 10-15. In addition, waiting mode is removed from the assembly lines if needed, in rules 9, 16 and through boolean b_3 and b_4 . Normal waiting mode is removed if an assembly line starts an action. The waiting-for-nothing mode is removed if the other assembly line starts or completes an action.

If both assembly lines have finished waiting for resources at the exact same simulated second, the build assembly line can start his action first. If there are enough resources left, the research assembly line may start his action afterwards. This behaviour is handled by rules 10-12, 15 and boolean b_2 .

B.3.3. Transition function

Identical to the previous MDP:

$$P_a(s, s') = \{1 \mid a \in A_s, (s, a) \rightarrow s'\}$$

B. Toy model additions

B.3.4. Reward function

Identical to the previous model:

$s_g = \langle -, 3, 3, -, - \rangle$ (where $-$ stands for an arbitrary value)

$$R(s) = \begin{cases} 50 - x & \text{if } s \text{ is a goal state} \\ -x & \text{otherwise} \end{cases}$$

Where x is the number of seconds passed since the previous (parent) state.

C. Ogame MDP functions

Note that the following functions use formulas from Ogame.¹ They might also use the values from table C.1, which gives the base cost values and the cost increase factor for actions in Ogame.

Action name	base_metal	base_crystal	base_deuterium	cost_increase_factor (CIF)
MetalMine	60	15	0	1,5
CrystalMine	48	24	0	1,6
DeuteriumSynthesizer	225	75	0	1,5
SolarPlant	75	30	0	1,5
RoboticsFactory	400	120	200	2
CrystalStorage	1000	500	0	2
MetalStorage	1000	0	0	2
DeuteriumTank	1000	1000	0	2
ResearchLab	200	400	200	2
Shipyards	400	200	100	2
EnergyTechnology	0	800	400	2
ImpulseDrive	2000	4000	600	2
ColonyShip	10000	20000	10000	1

Table C.1.: Base cost values and CIF for actions in Ogame 3.0.1

Function

pint cost_metal(action *a*, pint [level]=1)

Description

Returns the metal cost for a certain action.

Parameters

- action *a*: The action for which we want to know the metal cost.
- pint [level]=1: Optional parameter, represents the level of the action. Default value is 1.

Output

pint base_metal * CIF^{level - 1}

Additional notes

This function looks up the base_metal value in table C.1

¹Which can be found on: <http://board.ogame.org/board703-miscellaneous/board156-archive-version-2-0/board705-help-questions-archive/board631-faq-s-guides/576831-formula-thread-v3/>

C. Ogame MDP functions

Function

`pint cost_crystal(action a , pint [level]=1)`

Description

Returns the crystal cost for a certain action.

Parameters

- action a : The action for which we want to know the crystal cost.
- pint [level]=1: Optional parameter, represents the level of the action. Default value is 1.

Output

`pint base_crystal * CIFlevel - 1`

Additional notes

This function looks up the `base_crystal` value in table C.1

Function

`pint cost_deuterium(action a , pint [level]=1)`

Description

Returns the deuterium cost for a certain action.

Parameters

- action a : The action for which we want to know the deuterium cost.
- pint [level]=1: Optional parameter, represents the level of the action. Default value is 1.

Output

`pint base_deuterium * CIFlevel - 1`

Additional notes

This function looks up the `base_deuterium` value in table C.1

Function

`bool prec_costs(pint metal, pint crystal, pint deuterium, action a , pint [level]=1)`

Description

Returns true if the agent owns the resources to perform action a .

Parameters

- pint metal: The amount of metal owned.
- pint crystal: The amount of crystal owned.
- pint deuterium: The amount of deuterium owned.
- action a : The action for which we want to know if we own the resources.
- pint [level]=1: Optional parameter, represents the level of the action. Default value is 1.

Output

true if the agent owns enough resources.

false: if the agent does not own enough resources.

Additional notes

This function uses the `cost_resource` functions to determine the output.

C. Ogame MDP functions

Function

pint time_building(pint metal, pint crystal, pint lvl_roboticsfactory)

Description

Returns the duration of a building action in seconds.

Parameters

- pint metal: The metal cost of the action.
- pint crystal: The crystal cost of the action.
- pint lvl_roboticsfactory: The current level of the RoboticsFactory building.

Output

```
pint  $x = \frac{\text{metal} + \text{crystal}}{2500} * \frac{1}{\text{lvl\_roboticsfactory} + 1}$ 
pint  $r = x * 3600$ 
if  $r > 100$  then  $r = r - 90$ 
return  $r$ 
```

Function

pint time_research(pint metal, pint crystal, pint lvl_researchlab)

Description

Returns the duration of a research action in seconds.

Parameters

- pint metal: The metal cost of the action.
- pint crystal: The crystal cost of the action.
- pint lvl_researchlab: The current level of the ResearchLab building.

Output

```
pint  $x = \frac{\text{metal} + \text{crystal}}{1000} * (1 + \text{lvl\_roboticsfactory})$ 
pint  $r = x * 3600$ 
return  $r$ 
```

Function

pint time_shipyard(pint metal, pint crystal, pint lvl_shipyard)

Description

Returns the duration of a shipyard action in seconds.

Parameters

- pint metal: The metal cost of the action.
- pint crystal: The crystal cost of the action.
- pint lvl_shipyard: The current level of the Shipyard building.

Output

```
pint  $x = \frac{\text{metal} + \text{crystal}}{5000} * \frac{2}{\text{lvl\_shipyard} + 1}$ 
pint  $r = x * 3600$ 
return  $r$ 
```

C. Ogame MDP functions

Function

double prod_factor(pint lvl_metalmine, pint lvl_crystalmine, pint lvl_deuteriumsynthesizer, pint lvl_solarplant)

Description

Returns the production factor, given the parameters. The value is calculated by comparing energy production and consumption.

Parameters

- pint lvl_metalmine: The level of the MetalMine.
- pint lvl_crystalmine: The level of the CrystalMine.
- pint lvl_deuteriumsynthesizer: The level of the DeuteriumSynthesizer.
- pint lvl_solarplant: The level of the SolarPlant.

Output

```
pint energyproduction = 20 * lvl_solarplant * 1.1lvl_solarplant
pint energymetal = 10 * lvl_metalmine * 1.1lvl_metalmine
pint energycrystal = 10 * lvl_crystalmine * 1.1lvl_crystalmine
pint energydeuterium = 20 * lvl_deuteriumsynthesizer * 1.1lvl_deuteriumsynthesizer

double r =  $\frac{\textit{energyproduction}}{\textit{energymetal} + \textit{energycrystal} + \textit{energydeuterium}}$ 
if r > 1.0 then r = 1.0
return r
```

Function

double prod_metal(pint lvl_metalmine, pint lvl_crystalmine, pint lvl_deuteriumsynthesizer, pint lvl_solarplant)

Description

Returns the production of the MetalMine in metal per second.

Parameters

- pint lvl_metalmine: The level of the MetalMine.
- pint lvl_crystalmine: The level of the CrystalMine.
- pint lvl_deuteriumsynthesizer: The level of the DeuteriumSynthesizer.
- pint lvl_solarplant: The level of the SolarPlant.

Output

```
pint metalproduction = 30 * lvl_metalmine * 1.1lvl_metalmine + 30
double r = metalproduction * prod_factor(lvl_metalmine, lvl_crystalmine,
    lvl_deuteriumsynthesizer, lvl_solarplant)/3600
return r
```

Function

double prod_crystal(pint lvl_metalmine, pint lvl_crystalmine, pint lvl_deuteriumsynthesizer, pint lvl_solarplant)

Description

Returns the production of the CrystalMine in crystal per second.

Parameters

- pint lvl_metalmine: The level of the MetalMine.
- pint lvl_crystalmine: The level of the CrystalMine.
- pint lvl_deuteriumsynthesizer: The level of the DeuteriumSynthesizer.
- pint lvl_solarplant: The level of the SolarPlant.

Output

```
pint crystalproduction = 20 * lvl_crystalmine * 1.1lvl_crystalmine + 15
double r = crystalproduction * prod_factor(lvl_metalmine, lvl_crystalmine,
    lvl_deuteriumsynthesizer, lvl_solarplant)/3600
return r
```

Function

double prod_deuterium(pint lvl_metalmine, pint lvl_crystalmine, pint lvl_deuteriumsynthesizer, pint lvl_solarplant)

Description

Returns the production of the DeuteriumSynthesizer in deuterium per second.

Parameters

- pint lvl_metalmine: The level of the MetalMine.
- pint lvl_crystalmine: The level of the CrystalMine.
- pint lvl_deuteriumsynthesizer: The level of the DeuteriumSynthesizer.
- pint lvl_solarplant: The level of the SolarPlant.

Output

```
pint deuteriumproduction = 12.8 * lvl_deuteriumsynthesizer * 1.1lvl_deuteriumsynthesizer + 15
double r = deuteriumproduction * prod_factor(lvl_metalmine, lvl_crystalmine,
    lvl_deuteriumsynthesizer, lvl_solarplant)/3600
return r
```

Additional notes

In Ogame, the deuterium production calculation includes the planet's temperature. The temperature is a somewhat random value variable that has bounds, depending on the planet slot. Planet temperature is close to trivial for this research. Because of this, we calculated the mean temperature of all the possible Homeworld planets and in effect used 12.8 as base, as opposed to using 10.0 as base and increasing or decreasing it depending on the temperature.

C. Ogame MDP functions

Function

storage_capacity(pint level)

Description

Returns the storage capacity for a certain level of a storage warehouse/tank.

Parameters

- pint level: Represents the level of the storage building.

Output

```
pint  $r = 12500 * e^{20 * \frac{\text{level}}{33}}$ 
return  $r$ 
```

Additional notes

e is Euler's number

Note that in these definitions we have not spoken about the rounding of numbers. Ogame has peculiar behaviour considering rounding of numbers, and sometimes behaves in a way that seems illogical. To avoid making these definitions more complex, we have omitted this peculiar behaviour. The simulator will implement the correct rounding behaviour.

D. Source code

The source code can be downloaded from Google Drive (without logging in) by going to the following URL:

<https://docs.google.com/open?id=0B8xFIlzMhHcpTkhRbFVsWTZ4dFE>

Click on File ->Download to download a .zip package, which includes the project folder of a Microsoft Visual Studio 2010 project.

If for some reason Google Drive is not available, the package can also be requested by sending an email to: *ivankoster at gmail dot com*

To run the source code in Visual Studio 2010, the NeuronDotNet 3.0 library has to be referenced, which can be found on <http://sourceforge.net/projects/neurondotnet/>. Referencing in Visual Studio 2010 can be done by right-clicking References ->Add Reference in the Solution Explorer panel. Open the Browse tab, look for the NeuronDotNet.Core.dll file and click OK.

If Visual Studio 2010 is not owned or not wanted, the plain source code can be found in the .zip package: the .cs files in the OgameReinforcementLearning folder.