NTNU
Norwegian University of
Science and Technology

Skippy
Agents learning how to play curling

Frode Aannevik
Jan Erik Robertsen

Master of Science in Computer Science

Submission date:	June 2009
Supervisor:	Helge Langseth, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The purpose of this project is to work on the subject of AI agents in the domain of curling, and includes the following tasks:

1) Design a curling simulator (using an adequate physical model) that curling-playing agents can interface with.
2) Explore knowledge representation in the curling domain.
3) Build a curling-playing agent and test it in the simulator.

Assignment given: 15. January 2009
Supervisor: Helge Langseth, IDI

Abstract

In this project we seek to explore whether it is possible for an artificial agent to learn how to play curling. To achieve this goal we developed a simulator that works as an environment where different agents can be tested against each other. Our most successful agent use a Linear Target Function as a basis for selecting good moves in the game. This agent has become very adept at placing stones, but we discovered that it lacks the ability to employ advanced strategies that reach over more than just one stone. In an effort to give the agent this ability we expanded it using Q-learning with UCT, however this was not successful. For the agent to work we need a good representation of the information in curling, and our representation was quite broad. This caused the training of the agent to take an unreasonably large amount of time.

Preface

This project is the result of our work during the spring of 2009, and is done as part of our fulfillment of the master program in computer science at The Norwegian University of Science and Technology (NTNU). We would like to thank the following for their help and support during the project; our supervisor at IDI, associate professor Helge Langseth, as well as our friends and family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives an introduction to the report and the project as a whole. We start by describing our motivations for starting this project and continue with specifying our main goals. We have also included a short introduction to the game of curling with the most important rules and some strategies that are useful. For readers that are unfamiliar with curling we hope this will be popular to get a better understanding of this project. Finally this chapter concludes with outlining the structure of each chapter in this report.

1.1 Motivation

As students of Artificial Intelligence we wanted to do a project within this field. Still, there remained to find a suitable problem that we both found interesting and that was not to complex for a project of this size. There are many problems that are suitable for trying to solve using AI techniques, and some that we found interesting were those that involve computers playing games. One game that emerged as a good alternative was that of curling.

First of all there has not been much work on AI in the context of Curling before. Further more, curling poses some new and difficult challenges compared to games that are typically used when creating intelligent agents, such as checkers and other board games. In a game like checkers there is a board and some pieces that can be in a finite number of configurations and a fast computer can calculate all of them. In fact Jonathan Schaeffer et al. [20] have completely solved the game of checkers, showing that with optimal play the worst one can do is a draw.

Curling has been described as chess on ice, which is a good analogy considering curling is a thinking persons game, requiring sound strategy and good planning abilities to succeed in. However there are some aspects that make curling very different form traditional board games. In the case of curling, the board, or actually the ice, is a continuous surface where the pieces, or stones, can be positioned anywhere. This means we can not use the same techniques as in the case of checkers, at least not without some modifications.

1.2 Aims of the project

We want to explore whether or not it is at all possible to create an agent capable of playing curling, given the continuous nature of the game. Or more specifically; will it be possible to do so with the computing power we have at our disposal. Further more, we want to explore how sophisticated such an agent can become. Finding a measure for this might not be so easy. One way is to see whether or not it is able to do better than someone just placing stones at random. If this becomes the case, it will have shown that the agent has acquired at least some of the techniques required to play a good game of curling. Another measure for the agent's success would be to see how it fares against a human player. Although, creating an agent that can consistently beat a human is probably very difficult, but we hope that the agent will be sophisticated enough that it poses a challenge.

We want to implement this project is such a way that the code is reusable. We envision creating an at least rudimentary simulator, with which two players may compete against each other in a game of curling. A player might be either a human or a computer agent, and the simulator allows for any combination of play; computer vs. computer, computer vs. human, or human vs. human. We want the simulator to have a clearly defined API, so that new versions of computer agents can be easily tested. This also opens up for future projects to take advantage of some of our code.

1.3 About Curling

Curling is a team sport played on ice, where the object is to slide stones along the playing field towards a target area called the "house". Two teams of four players have two stones each for a total of sixteen stones. They take turns sliding the stones and the team with stones closest to the center of the house scores points. Each stone in the house which is closer to the center than all the opposing team's stones, earns the team one point. If no stones lie within the house at the end of a round, no team scores points. One round, that is when all players have set two rocks each, is referred to as an end. A game of curling consists of ten such ends, however recreational games often have fewer ends. If the game is tied after ten ends, additional ends are played to break the tie.

In curling, two players may use brooms to sweep the ice in front of the stone as it slides down the ice. This will of course remove any debris that may hinder the stone, but the primary purpose is to control the speed of the stone. The friction from sweeping creates heat, which melts the ice in front of the stone making it go further. Professional players can increase the distance of the shot with several meters.

1.3.1 History of Curling

The exact origin of curling is unknown, there are however paintings from mid 16th century depicting an activity similar to curling. There are also written references from 1540 from a monastery in Scotland involving throwing stones across a frozen pond. It is clear that what started as an enjoyable pastime in northern Europe has become a modern sport with its own world championship and which is included in the winter Olympics. Figure 1.1 shows people playing curling.

Figure 1.1: Left: Curling as played in the olden days on a frozen pond. Right: a modern curling hall. (Images courtesy of worldcurling.org)

1.3.2 Rules

The rules of curling are quite extensive and intricate. In this section we will only discuss the ones that are needed for a basic understanding of the sport, as well as any rules that are relevant for our project. All rules are from World Curling Federation [6].

Dimensions of the Curling Ice - Figure 1.2 shows the dimensions of a curling ice according to official regulations.

Setting a Rock - A players sets a rock by pushing off from the hack, sliding the rock in front of him. The rock must be released before it reaches the near hog line. The player can use in- or out-curl ("give the rock a twist" clock- or counter-clockwise) before releasing the rock. Using curling will make the rock screw as the speed decreases and can be used to "hide" the rock behind other rocks. A stone is out of play if it fails to completely cross the far hog line, if it at any point touches the lines on either side of the ice, or if it completely crosses the back line.

Free Guard Zone - One important rule in curling is that of the Free Guard Zone (FGZ). The zone is the area between the hog line and the tee line excluding the house. See figure 1.2. The rule says that a stone placed within the FGZ cannot be taken out until after the fourth stone has been set. Should this happen, the affected stone(s) will be placed as close as possible to where they lay before, and the offending stone will be removed from play. This rule ensures that there will be stones in play right from the start and the teams are forced to play offensively, which also make the game more exciting for the spectators.

The Last Stone - Having the last stone in an end is an important strategic advantage in curling. Before a match each team throws one rock at the house and the team which comes closest to the button (the center of the house) gets the last stone in the first end. Whenever a team scores points, the opposing team will get the last stone in the next end. If an end is tied, the team that currently has the last stone will keep it in the next end. Having the last stone is often referred to as having the hammer.

1.3.3 Strategies

Last Stone Strategies - As mentioned, having the last stone gives an important advantage. Often teams play defensively, alternatively taking out each others

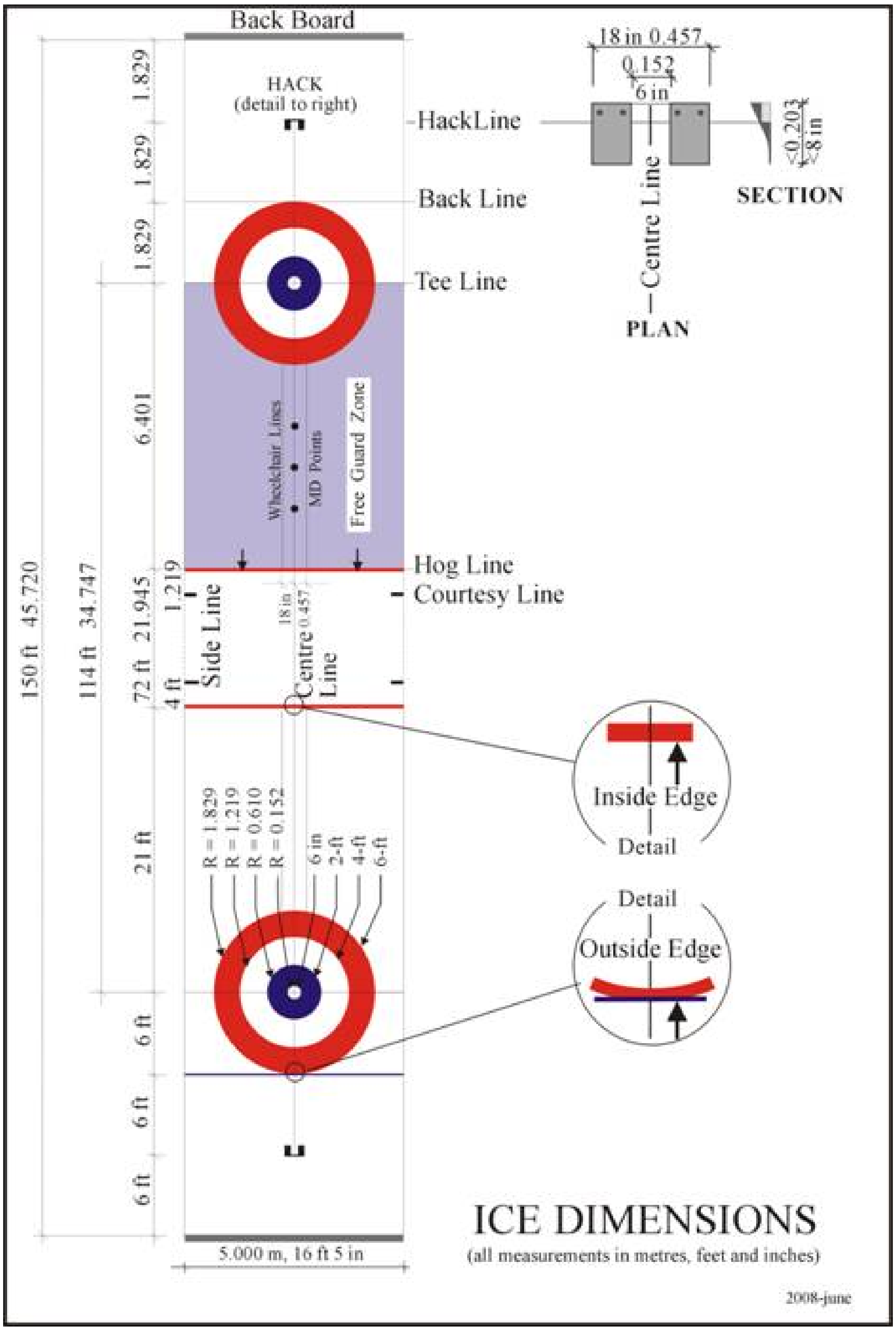


Figure 1.2: Dimensions of curling ice

stones. It is obvious this will result in the team with the last stone scoring a point. However, this also leads to the other team getting the last stone in the next round. In general, most teams consider having the last stone as more valuable than scoring one point. This leads to the following strategy. When having the last stone: Try to take two or more points, and force a tie rather than take just one point. However, take the one point if the alternative is the opponent scoring. Similarly when not having the last stone: Try to take as many points as possible, but if the best you can do is a tie, try to force the opponent to take one point.

The following list describes some common terms often used in curling. They are the basic tools in a curling players arsenal that he can use to get the desired result. Examples of the terms are shown in Figure 1.3.

- Hit and Stay - A takeout where the played stone stays in the spot where it made contact with the stationary stone. Figure 1.3(a).
- Hit and Roll - When a played stone removes an opponent stone and then slides (rolls) to a new position some distance away. Figure 1.3(b).
- Clearing (Peel) - A takeout that removes a stationary stone from play and also rolls from play. Figure 1.3(c).
- Raised takeout - A takeout played to strike a stationary stone, usually a guard, onto the stone behind it to remove it from play. Figure 1.3(d).
- Double takeout - A takeout that removes two of the opponent's stones with the same shot. Figure 1.3(e).
- Raised draw - The played stone promotes another stone into the house. Figure 1.3(f)
- Come Around - A draw that curls narrowly past a guard and comes to rest hidden behind the guard. Figure 1.3(g).
- Split - A stone played at near draw weight to hit a stationary stone in such a way that the stones split in opposite directions, but remain in play. Figure 1.3(h).
- Wick - When a played stone touches a stationary stone just enough so the played stone changes direction. Figure 1.3(i).
- Drawing a port - A stone is played between two stationary stones close to each other. Figure 1.3(j).

- Freeze - A precise draw-weight shot in which the delivered stone comes to rest tight against a stationary stone. Figure 1.3(k).

Finally, we give some terms commonly used to describe properties about the stones in play:

- Shot - At any time during an end, the stone which is closest to the button. Figure 1.4
- Biter - A stone that comes to rest, so that only a portion of its circumference bites the outer edge of the house. Figure 1.4
- Guard - A stone played to a position where it protects, or could later protect, a stone behind it. Figure 1.4

1.4 Report Structure

Chapter 2 presents the problem and challenges for this project.

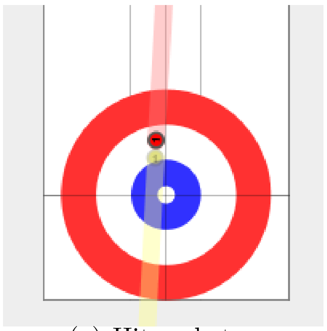
Chapter 4 describes techniques used in this project. Linear Target Function and Reinforcement Learning are some of the techniques presented.

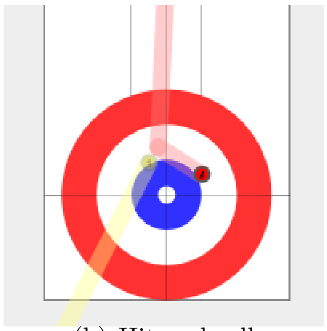
Chapter 5 explains the SkippySimulator architecture, its main components and highlights some of its implementations including the implementation of the agents.

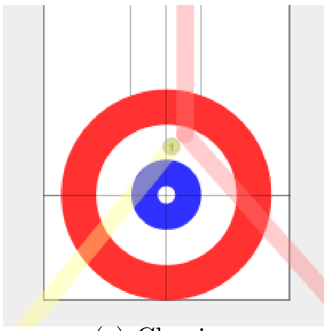
Chapter 6 explains the experiments performed and presents the results from the experiments.

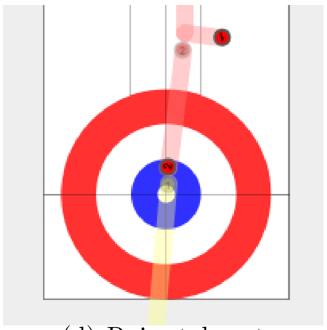
Chapter 7 presents the evaluation of the project.

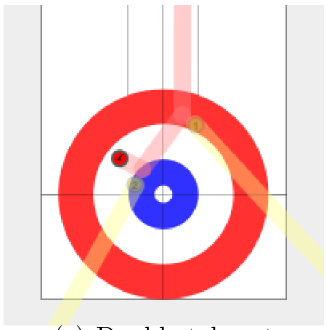
Chapter 8 presents the conclusion from our work and suggest further work.

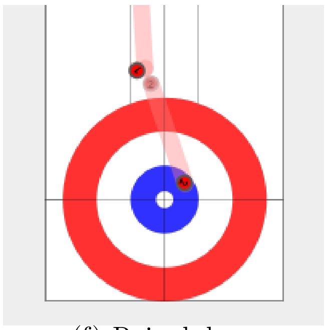

(a) Hit and stay

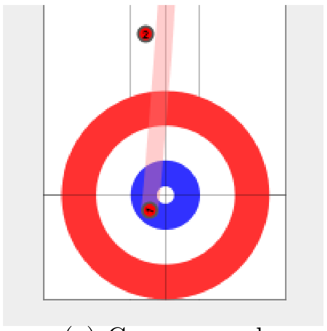

(b) Hit and roll

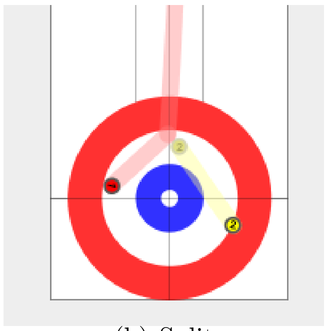

(c) Clearing

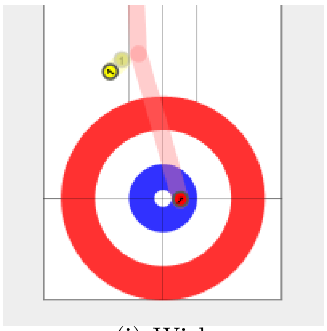

(d) Raise takeout

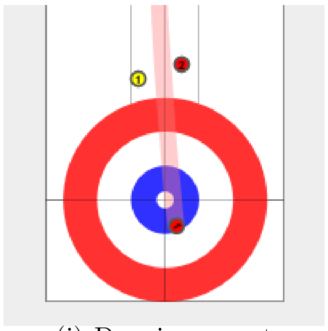

(e) Double takeout


(f) Raised draw


(g) Come around


(h) Split


(i) Wick


(j) Drawing a port

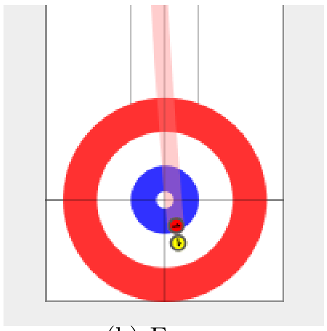

(k) Freeze

Figure 1.3: Curling Terms Examples.

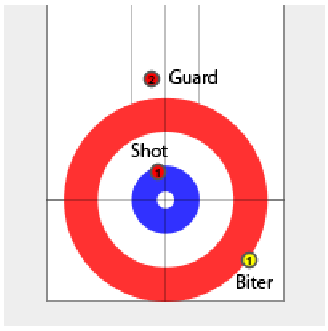


Figure 1.4: Three types of stones.

Chapter 2

The Problem and Challenges

We want to create a simulator that can adequately simulate a game of curling. The main challenge here will be on the physics engine. One of the most important features of curling is how the rocks move on the ice and what happens when they collide. It will be necessary to create a simulator that is as realistic as possible. Further more we need functionality for running full games of curling, where things like who has the hammer and how many points each player has, are handled properly. Finally we want the simulator to have an intuitive interface that let us test any combination of players against each other, be they human or artificial.

When it comes to the artificial agents, we want them to be able to play a good game of curling. That means that they utilize some of the techniques and strategies that increase ones chance of winning, rather than just placing stones at random. The main challenge here will be to find what techniques within the field of Artificial Intelligence are suitable for implementing these agents. Another challenge will be how to represent information about the curling domain, which these agents will rely upon to make their decisions. Unlike curling, the games traditionally used in AI research are discrete. For example chess, where there are only so many states the game can be in, and at any point there is only a relatively small set of actions one can take. In curling everything is continuous, so we need a representation that can adequately describe the world of curling, while at the same time be limited enough that it will be possible for the computers we have at our disposal to work with it.

Chapter 3

Goals and Constraints

This chapter presents a more detailed view of our goals and constraints. The chapter is divided into two sections. Firstly, we deal with the goals and constraints relevant to the task of creating an agent that can learn to play curling, secondly we give technical overview of the requirements for the curling simulator.

3.1 Curling learning problem

Playing Curling as a human is fun and challenging. The game, nicknamed "Chess-on-ice", requires both physical skill and a sound strategy to defeat the opponent. Combining all information available, both from current and previous games, is challenging for a human. For an agent, it is even harder. Even when having a complete picture of a game it is not clear how one agent should use past experience to make sounds strategies/decisions. What strategy should we adopt now? Where should we try to set the next rock? How will our actions now (current rock/round) determine the end results? It is also important to balance the risk and gain: what if we miss our targets?

Besides being non-deterministic and complex, the decisions also need to be taken in a fashionable time. There are no real-time requirements, curling is like chess or other turn-based games, but learning a system with high resource demands is more difficult to control and test.

A possible bottleneck for our system could come from the simulator and how it simulates the curling physics. It will therefore be important to prioritize simulation performance over other features when designing the system. Beside from bottlenecks found in the simulator it will be important to moderate the resource demands from our agent. Example of such is to moderate the demands for task such as *strategy selection* and *set a rock*. The latter task will have a high use frequency and should therefore be both efficient and computationally feasible. This creates constraints for our choice of mechanism to use when enabling agents to learn to play curling, and will requiring the use of both abstractions and finding approaches to simplify the complexity of the game world.

For learning problem we have the following goals:

- Explore the effect of using *Linear Target Function* for finding favorable positions to set a rock.
- Determine possible system for handling the nondeterministic nature of curling.
- Examine the effect of boosting the AI level with Reinforcement Learning (RL). Strategy planning is one example area where RL could be of interest.

For all three goals it will be important to find a balance between accuracy and performance.

3.2 Curling Simulator

The main purpose of the simulator is to provide a basic, underlying system for testing curling playing agents. Compared to other simulator frameworks such as Robocode [17] the main focus will be on the rock physics (collision, curl) and agent integration rather than real-time performance. The goals for the simulator are divided into two main groups, their relationship to curling or to the agent.

3.2.1 Realism

Realism is an important part of any simulator. We have identified three main realism goals that we believe is important to incorporate into the simulator.

- The simulator must use an adequate physic model.
- Simulate the game accordingly to official curling rules.
- Support a mechanism for simulating the nondeterministic nature of curling.

3.2.2 Agent

For the agent subsystem we have specified a set of goals and requirements to lessen the effort needed when using the simulator framework to build curling agents.

- Interface to both human and computer agents.
- Provide the user of the simulator with a clearly defined agent interface containing a basic set of commands required to play curling.
- Give adequate and informative feedback to the agents. This includes the positions rocks, current and score.

Provide interface for both human and computer agents is an important option that allows all agents to play against each other. It opens also up the possibility of having cooperation between human and computer agents. A well defined agent interface is a key feature for the simulator. Apart from providing the agent with a set of basic commands it should also give access to all relevant information about the curling environment created by the simulator. It will in the case of interface design be important to use well known and easy naming convention that enables both novice and expert curlers to understand their purpose.

3.2.3 Constraints

The constraints for the simulator relates to its architecture and implementation.

- Develop the simulator in Java 5.0
- Use third party libraries/framework where possible.
- Simulator must be built as a framework.

Java 5.0

It can be argued that there are other languages suited more for a curling simulator. One example could be *C#* and Microsoft game framework XNA [24]. However, our main programming skills lays in Java, and using it does not require us to use time on learning a new language.

Third party libraries/frameworks

Third party solutions can provide us with great shortcuts in the development of the simulator. There is also no reason to reinvent the wheel, and investigating other solutions could provide us with valuable inspiration.

Chapter 4

Theories behind Skippy agents

The field of Artificial Intelligence or AI is extensive and not easily summarized in one single chapter. Hence, this chapter will be limited to only give an overview over areas within AI that can have possible applications when building a curling playing agent. This includes describing the properties of an Intelligent Agent, and describing the algorithms used in the implementation.

4.1 Intelligent Agent?

The beginning is always a good place to start, so what exactly is an intelligent agent? The answer depends on whom you ask and can vary quite widely. For the purpose of this project however, we will focus exclusively on the definition presented in Russel & Norvig [18, p.31].

> An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.

This definition of agent covers a broad spectrum of machines, from thermostats, to animals, even humans. It is important to recognize that this definition does not require that the agent has the capability to learn something new.

Figure 4.1 presents the basic elements of an agent. In most cases an agent will have one or more sensor(s) that are used to perceive the environment, one or more effector(s) that can change the environment, and a control system. The control system has the task of assigning a mapping from sensors to effectors, and hence provides more rational behavior. We illustrate in Figure 4.2 how one of the simplest types of agent, the Thermostat-Heating Agent, fits this model. This agent has one sensor in the form of a thermometer that senses the temperature

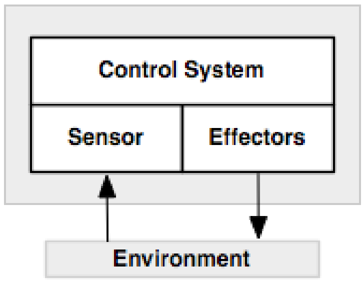


Figure 4.1: Basic anatomy of an agent.

of the environment, and one effector, the heating element that can supply heat to the environment. Finally it has a control system in the form of a very simple logic that says "turn the heat on if the perceived temperature is less than 20 °C , and turn the heat off if it is over 30 °C."

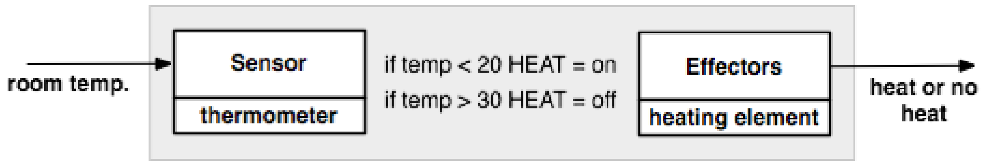


Figure 4.2: Thermostat-Heating Agent.

Even in a simple example as the Thermostat-Heating Agent we have an environment that determines the action of an agent, and in turn, the agent's action modifies the environment.

4.1.1 Environment

There exist many variations of possible environments an agent could operate in; we can use the following classification scheme taken from Russel & Norvig [18, p.45].

Accessible vs. inaccessible: In an accessible environment the agent can detect accurately all relevant information about the environment. This simplifies the agent since it is not required to maintain any internal state to keep track of the world.

Deterministic vs. nondeterministic: A deterministic environment is any environment in which a single action has a single guaranteed effect. It is no uncertainty about what effect an action has when applied in a given state.

Episodic vs. non-episodic: An episode consists of the agent perceiving and then acting. The environment is episodic if the quality of one episode depends just on the episode itself. A non-episodic sequential environment requires the agent to plan ahead.

Static vs. dynamic: If the environment can change while the agent is passive (not executing any action), we say the environment is dynamic for that agent; otherwise it is static. Dynamic environment are more difficult to deal with because the agent needs to worry about the passage of time when planning and deciding on an action.

Discrete vs. continuous: An environment is discrete if there are a fixed, finite number of percepts and actions in it.

The Curling Environment

The curling environment is evaluated by using the scheme suggested by Russel and Norvig. Notice, this evaluation is based on the environment provided by the simulator and not real-life curling. However, areas with key differences will be highlighted together with the reason for the differences.

Accessible or inaccessible: The simulator provides the agent with all information about the state of the environment, which entails accurate information about rock position, progress in match (rock/round number), points, etc. From this stand point it is clearly an accessible environment.

However, there are some aspects with the simulator environment that differ from real-life curling accessibility. One example is the information related to the opponent. In real life, players may observe the other team, both on and off the ice. The simulator however, provides no information about the other agent's actions; an agent can only observe the result of it. Without this type of information you could state that the environment provided by the simulator is inaccessible when compared to real-life. However, we argue that most agents would deem this type of information as irrelevant or of less importance, which makes the environment for our case accessible.

Deterministic or nondeterministic: Curling requires both skill and luck. In order to be a good curler one needs to lay good plans. But there is no guarantee that anyone will be able to execute those plans perfectly. There will always be an element of luck involved, for good or for bad. The simulator adds noise to the agents' effectors to mimic this nondeterministic behavior, making the simulator environment nondeterministic.

Episodic or nonepisodic: The environment is nonepisodic because the result of a round (e.g. the result of the last rock) is dependent on past performance (both of the agent and its opponent).

Having the last rock (hammer) in a round gives a great advantage to a player. And teams will often take turns having the hammer every other round. This fact also makes the environment nonepisodic.

Static or dynamic: Since curling is turn-based, we expect that the environment will change while one player is passive; making the environment dynamic.

Discrete or continuous: In curling the positions of the rocks, as well as the speed and angle at which rocks are sent, are all continuous values. This results in an infinite number of possible states for a curling game and the environment is clearly continuous.

To summarize, the curling environment is accessible, non-deterministic, nonepisodic, dynamic and continuous. Apart from the property of accessibility, these are all factors which complicate the task of creating capable agents for the environment.

4.2 Linear Target Function

The book "Machine Learning" by Tom Mitchell [13, p.7] presents an example of how an agent using a $linear function$ as its basis for decision making can be used when learning to play checkers. We want to explore the possibility of using this technique as a tool to find good moves in a game of curling. A *move* means sending a rock at a specific angle and with a specific speed. To use this agent there are two things we need. Firstly; we need a vector that describes the state of the ice at any point in the game. Secondly; we need an *evaluation function* $V(s) \rightarrow \mathbb{R}$ that maps the desirability of any state s to a real value. We can then find the best action to take at any point by simulating all legal actions, evaluating the resulting states using the V function, and choosing the action that resulted in the most desirable next state. Simulating all legal action will not be possible in the case of curling, given that there are essentially an unbounded number of legal actions. We can solve this problem by limiting the number of actions we test. An action consists of two values, the speed and the angle at which the rock is sent. By defining limits for these values and testing actions with discrete increases in speed and angle we end up with a finite number of actions to test.

4.2.1 Defining the State

As previously mentioned we need a description of the state of a curling game. This description needs to be in the form of a vector of real numbers. We propose the following list of features that we feel adequately describes the state of a curling game. Each feature will be a numerical value and together they make up our state vector.

- x_1: the points in a state, calculated as if the state was a final state at the end of a game. The feature is positive if the player scores and negative if the opponent does).
- x_2: a score for the number of rocks the player has in house.
- x_3: a score for the number of rocks the opponent has in house.
- x_4: the number of guarded rocks belonging to the player.
- x_5: the number of guarded rocks belonging to the opponent.
- x_6: the number of corner guards.
- x_7: the number of center guards.

The score in features x_2 and x_3 is calculated as follows: For each rock in the house, add to the feature: $1 - \frac{d}{12 \text{ ft}}$ where d is the distance of the rock from the button measured in feet. This ensures that a rock closer to the button contributes with more to the feature than a rock at the edge of the house. For a rock to be guarded it needs to be in the house, and have another rock in front of it that lies at most one half rock-width to either side, there by making it harder to take the guarded rock out. In other words, if a player stands at the other end of the ice looking at a guarded rock, it is at least partially concealed by another rock. A center guard is a rock that lies in front of the house, at *most* two feet from the center line, while a corner guard is a rock that lays in front of the house, at *least* 2 feet from the center line.

4.2.2 Defining the V Function

The V function needs to be defined so that it gives a good measure of the desirability of the different states in the game. For a final state (i.e. at the end of a round) the function will be defined explicitly as positive if you win and negative if you loose, with some modifications if the round is tied. For all other states, an approximation of the V function, $\hat{V}$ is introduced. It is calculated as a linear combination of the state vector with a set of weights as follows

$$\hat{V}(s) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6 + w_7x_7 \tag{4.1}$$

where the weights w_i are numerical coefficients, that we will discuss how to learn in Section 4.2.3. The weight w_0 is included to provide an additive constant to the state value. The problem now becomes how to adjust there weights so that a good state is given a high value and a bad state a lower or negative value. In general we will use that a state that leads to a good state is itself a good state and vice versa for a bad state.

In detail we define the following five rules for the behavior of the V function.

Rule *1*. If s is a final winning state, then $V(s) = 100 \cdot \text{points}$.

Rule *2*. If s is a final losing state, then $V(s) = -100 \cdot \text{points}$.

Rule *3*. If s is a final tie state, and the player has the last stone, then $V(s) = 150$.

Rule *4*. If s is a final tie state, and the opponent has the last stone then $V(s) = -150$

Rule *5*. Otherwise, $V(s) = V(\hat{s})$, where $\hat{s}$ is the highest scoring state that is achieved starting from state s and playing optimal until the end of the game.

Rules *1* and *2* are straight forward, it is positive to win and negative to loose. The *points* in these rules denotes the number of points awarded in the round regardless of who won. Rules *3* and *4* deal with the situation where a round ends in a tie. There rules ensure behavior in accordance with the strategies discussed in section 1.3.3. In short, it is more valuable to force a tie and keep the last stone, rather than take just one point. Rule *5* deals with all states that are not final, i.e. not at the end of a round. This rule is the most difficult to implement. It has a look-ahead approach for finding optimal play sequences, and this quickly becomes virtually impossible to compute. Even more so in the non-deterministic environment found in curling. To handle the difficulty of this rule we will use the $\hat{V}$ function as an estimate for finding states with good score.

4.2.3 Training

We want to train the weights used by the $\hat{V}$ function, so that it correctly classifies states as good or bad. To do this we will use the gradient decent rule as described by Tom Mitchell [13, Ch.4.4]. This technique works by adjusting the weights in the direction that minimizes the error between the output of the $\hat{V}$ function and a training value. We start by defining the training value for the estimate of the $\hat{V}$ function as follows

$$V_{\text{train}}(s) = \hat{V}(\text{successor}(s)) \tag{4.2}$$

where successor(s) is the next state where the player can set a rock. We want to adjust the weights of the $\hat{V}$ functions so that the difference between V_{train} and $\hat{V}$ is as small as possible. Using this setup we capture the idea that a state that leads to a good state is itself a good state. In other words, if a round of curling ends with a win, the weights for all the states visited during the course of the game are adjusted such that the estimate of those states become higher, and vice versa if the game resulted in a loss.

To arrive at a rule for updating the weights we start by defining a measure for the error of a weight vector. There are many ways to define this error but we shall see that the following will be convenient.

$$E(\boldsymbol{w}) = \frac{1}{2}(V_{\text{train}}(s) - \hat{V}(s))^2 \tag{4.3}$$

E is defined as a function of $\boldsymbol{w}$ since the $\hat{V}$ function depends on this weight vector. What we want is to find the weight vector with minimum error (the one with lowest E value). If a weight vector only consists of two vectors we can visualize a space with the two vectors along the x and y axis and the associated E values along the z axis. The error then becomes a surface in this space and the task is to find its minimum. This is the same for vectors of any length but it is easier to visualize in 3 dimensions. Having found this minimum means that the corresponding weight vector is the one that causes the $\hat{V}$ to give an estimate that is as close as possible to the desired value. In other words, when testing all actions in a particular state, the action that gives the highest estimate from the $\hat{V}$ function is the one that results in the most desirable next state. To find the minimum of the error surface we can use the gradient descent rule. The gradient descent rule works by first starting with an arbitrary weight vector, then repeatedly modifying it in small steps. At each step the weight vector is altered in the direction that produces the steepest decent along the error surface.

The direction of the steepest descent along the error surface can be found by computing the derivative of E with respect to each component of the vector $\vec{w}$. The training rule for updating the weights thus becomes

$$w_i \leftarrow w_i + \Delta w_i \tag{4.4}$$

where

$$\Delta w_i = -\eta \frac{\delta E}{\delta w_i} \tag{4.5}$$

The negative sign is there because we want the steepest descent not ascent and η is a positive constant called the learning rate, which determines the step size in the gradient descent search. To obtain $\frac{\delta E}{\delta w_i}$ we differentiate E from Equation (4.3) as follows

$$\begin{aligned}
\frac{\delta E}{\delta w_i} &= \frac{\delta}{\delta w_i}\frac{1}{2}(V_{\text{train}}(s) - \hat{V}(s))^2 \\
&= \frac{1}{2}\frac{\delta}{\delta w_i}(V_{\text{train}}(s) - \hat{V}(s))^2 \\
&= \frac{1}{2}2(V_{\text{train}}(s) - \hat{V}(s))\frac{\delta}{\delta w_i}(V_{\text{train}}(s) - \hat{V}(s)) \\
&= (V_{\text{train}}(s) - \hat{V}(s))\frac{\delta}{\delta w_i}(V_{\text{train}}(s) - \boldsymbol{w}^T\boldsymbol{x}) \\
&= (V_{\text{train}}(s) - \hat{V}(s))(-x_i)
\end{aligned} \tag{4.6}$$

Substituting the result of Equation (4.6) into Equations (4.4) and (4.5) yields the weight update rule for gradient descent.

$$\begin{gathered}
\Delta w_i = \eta(V_{\text{train}}(s) - \hat{V}(s))x_i \\
w_i \leftarrow w_i + \eta(V_{\text{train}}(s) - \hat{V}(s))x_i
\end{gathered} \tag{4.7}$$

where η is the learning rate. By using this algorithm the weights are adjusted in such a way as to minimize the error between the training value V_{train}, and the predicted estimate value, $\hat{V}$. Care must be taken when choosing the learning rate η, a too small value might make the algorithm terribly slow, and a too large value might make it inaccurate such that it ‘overshoots’ the minimum it is searching for. A good idea is to have the learning rate decay over time so that it becomes more accurate as it approaches the target.

We can see that a weight will not be adjusted if either $V_{\text{train}}(s) = \hat{V}(s)$ or $x_i = 0$, which is just as we want it. In the first case the weights can be considered properly trained, and in the second case the feature represented by x_i is not present in the given state and the weights should not be adjusted based on *that* training example.

4.2.4 Trade-offs

There is a trade-off between the level of detail (expressiveness) of a representation for a state (size of features) and the ease of learning. The more detailed a representation, the better it will be at approximation the value of a state; however, more details requires more training example in order to learn an accurate estimation. More complex and training demanding representation will also create problem of overfitting, where features may be adjusted to very specific random features of the training data, which have no causal relation to the target function. Overfitting the features will in generally create lower performance, especially for unseen situations.

The technique requires also some level of expert knowledge when designing the representation in order to recognize key features that is needed to effectively reduce the problem of expressiveness versus ease of learning.

4.2.5 Expectation for the agent

Our expectations for the agent using Linear Target Functions are based on how we defined the state vector and the V function. Some of the features in the state vector are mirrored for both the agent and the opponent. We hope that the weights for there features will be adjusted positively for those concerning the agent and negatively for those of the opponent. Specifically that it is a good thing for the agent to have rocks in the house while it is a bad thing if the opponent has. Similarly that it is good if the agent has guarded rocks but not if the opponent has that. If this becomes the case, we expect to see behavior where the agent will try to take out rocks belonging to the opponent and probably also try to let its own rock stay behind in the house to potentially score more points.

One interesting aspect is that of guarding. It might become the case that the agent will be reluctant to set guards because a guard in itself does not score points. However, having guards significantly increases the chance for other rocks to score. We can also see that there is a difference between placing a guard in front of a stone that is already in the house, and placing a guard in front of an empty house in order to play a stone behind that guard later. In the second case the guard does not get its real value until later and it might be difficult for the agent to be able to learn this strategy.

The Last Stone Strategies described in Section 1.3.3 is one example of a strategy that the agent will have problem to capture. We included the rules *3* and *4* in the V function in an effort to capture some of the essence of the strategy, but the effect will be limited by the agents "one-step-lookahead". The strategy can only be used effectively in cases where the agent can plan multiple steps ahead and knows where in the game it is (what step and what round). The knowledge about the round is necessary to avoid using the strategy in the last round. Our agent has no such capabilities and will therefore not be able effectively use the strategy.

To address the problem with "one-step-lookahead" for strategies such as Last Stone we will in Section 4.3 describe a technique that can resolve some of the problems.

4.3 Reinforcement Learning

Reinforcement learning addresses the problem with how an agent that can sense and act in an environment can learn to choose optimal actions to achieve their goals. This technique differs from supervised learning in many ways. The most important difference is that the learning is achieved without using input/output pairs. Instead, after an action is selected the agent is given a reward and the new state, but it is *not* told which action would be in its best long-term interest.
The technique is will be used to address the problem described in Section 4.2.5 related to "one-step-lookahead". In this section we will focus on the learning algorithm called Q learning that can achieve optimal control strategies from delayed rewards.

Markov decision process (MDP) provides a framework for how we can formulate the problem of learning sequential control strategies. An MDP consist of a set of discrete states, $\mathcal{S}$ and has a set of actions, $\mathcal{A}$, that it can perform. At each time step t, the agent knows its current state s_t and selects and action a_t and executes it. The environment responds to the action by giving the agent a reward, $r_t = r(s_t, a_t)$, and producing a new environment, $s_{t+1} = \delta(s_t, a_t)$. Both δ and r function are determined by the environment which can be unknown to the agent. For sake of simplicity we will only illustrate the reinforcement learning on a case in which $\mathcal{S}$ and $\mathcal{A}$ are both finite. The general rule is that both δ and r can be nondeterministic, but we start by consider the deterministic case first.

The goal of the agent is to learn a decision function $\pi : \mathcal{S} \rightarrow \mathcal{A}$, mapping states to actions that result in the greatest future reward. One solution is to use a policy that produces the greatest cumulative reward for the agent, named $V^{\pi}(s_t)$.

$$\begin{aligned} V^{\pi}(s_t) &\equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... \\ &\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \end{aligned} \tag{4.8}$$

The future reward, r_{t+i} in Equation (4.8), is generated by beginning in state s_t and repeatedly using the π decision function to select optimal action ($a_{t+1} = \pi(s_{t+1})$). In the equation ($0 \leqslant \gamma < 1$), γ is a constant that regulates the relative value of delayed versus immediate rewards.

Given the $V^{\pi}(s_t)$ defined we can specify the optimal decision policy, the agent's learning task as stated in Equation (4.9).

$$\pi^{\star}(s) \equiv \arg\max_{\pi} V^{\pi}(s_t), (\forall s) \tag{4.9}$$

4.3.1 Q-Learning

A simple example extracted from Mitchell [13, p.372] illustrates the basic concepts of Q-learning (Figure 4.3). We define the notation $Q(s, a)$ as the expected discounted reinforcement of taking action a in state s, where the value of Q is the reward immediately received upon executing the action a on the environment s plus the value of following the optimal action strategy (discounted by γ).

$$Q(s, a) = r(s, a) + \gamma V^{\star}(\delta(s, a)) \tag{4.10}$$

With the close relationship between Q and $V^{\star}$,

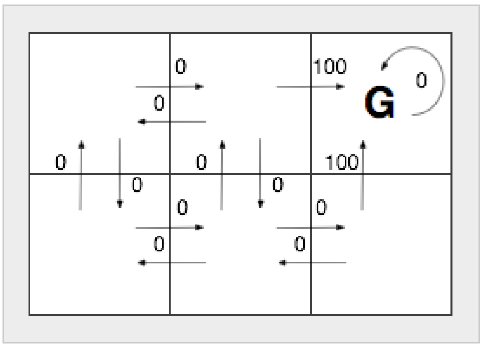
$$V^{\star}(s) = \max_a Q(s, a),$$

makes the task of learning Q correspond to learning the optimal policy and we can rewrite the Equation (4.10) as

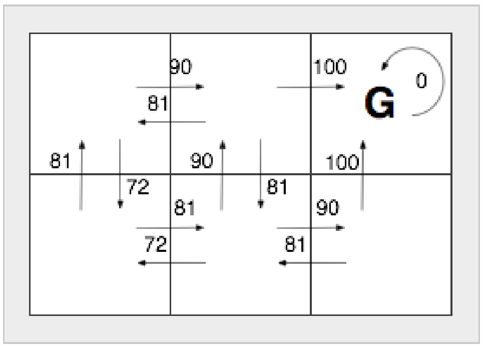
$$Q(s, a) = r(s, a) + \gamma \max_a Q(s, a) \tag{4.11}$$

where the optimal policy is:

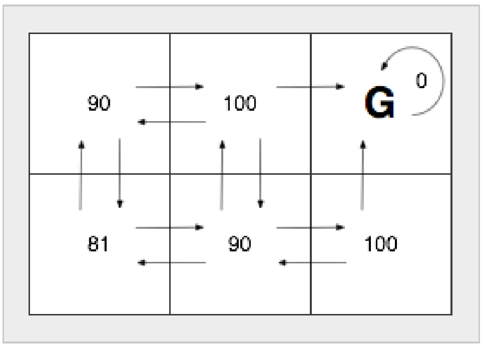
$$\pi^{\star}(s) = \arg\max_a Q(s, a)$$



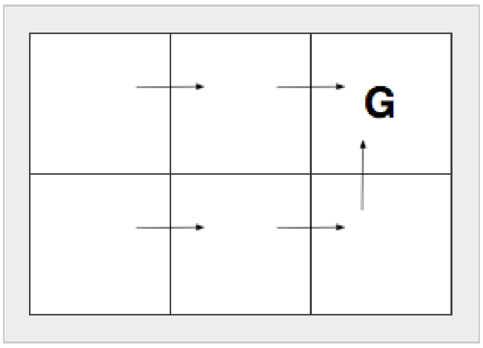
$r(s, a)$ (immediate reward) values.



$Q(s, a)$ values.



$V^\star$(s) values.



One optimal policy.

Figure 4.3: A simple deterministic grid world is used to demonstrate the basic principles of Q-learning. Each cell represents a distinct state, each arrow a distinct action. The reward function $r(s, a)$ gives reward 100 for actions entering goal state **G** and zero otherwise. Values of $V^\star$ and $G(s, a)$ follow from $r(s, a)$ and the discount factor $\gamma = 0.9$. The optimal policy figure corresponds to actions with maximal Q values.

Q-Learning Algorithm

When describing the algorithm we will use the notation $\hat{Q}(s,a)$ for the agent's estimate of the actual Q function. In the learning algorithm the agent store the $\hat{Q}(s,a)$ values in a large table containing the entries for each possible state-action pair. The initial table can be filled with random values or set all to zero. After creating the initial table the agent will iterate over choosing a state, selecting and executing some action and observing the new state and the received reward. The updating rule of the $\hat{Q}$ table can be expressed accordingly:

$$\hat{Q}_t(s,a) \leftarrow r(s,a) + \gamma \max_a \hat{Q}_{t-1}(\delta(s,a),a) \tag{4.12}$$

The training rule expressed in Equation (4.12) applies only on deterministic environment where executing action a in a state s will always result in the same state $\delta(s,a)$ and the same reward $r(s,a)$. Since we are planning to create a nondeterministic environment in the Curling simulator we need a learning rule that is generalized to work on nondeterministic environments.

$$\hat{Q}_t(s,a) \leftarrow (1-\alpha_t)\hat{Q}_{t-1}(s,a) + \alpha_t(r(s,a) + \gamma \max_a \hat{Q}_{t-1}(\delta(s,a),a)) \tag{4.13}$$

where

$$\alpha_t = \frac{100 \cdot \rho}{100 + T_t(s,a)}$$

where

- $T_t(s,a)$ is the number of times action a has been used in state s at time t.
- ρ is the maximum learning rate.

The updated rule updates the $\hat{Q}$ values with a decaying weighted average of the current $\hat{Q}$.

Both Equation (4.12) and Equation (4.13) can be proven to converge to the correct Q function under certain assumptions. One of them is that the agent must select its actions in such way that it will visit every possible state-action pair infinitive often. However, this can be very problematic in a large state and/or action space, and could slow the convergence to a good policy.

Action Selection

When the Q function has converged to the correct, action selection is simply:

$$\pi^\star(s) = \arg\max_a Q(s,a) \tag{4.14}$$

where $\pi^\star(s)$ is the optimal action given a state s. However, while the Q function has not converged, a less greedy approach is necessary to ensure that optimal actions can be learned. One one of the simplest way to "balance" exploration and exploitation is to use the greedy action selection ϵ-Greedy.

$$A_s = \begin{cases} \pi^\star(s) & \text{with probability } 1-\epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases} \tag{4.15}$$

However, ϵ-Greedy has the problem of neglecting the action values, which can delay convergence unnecessary.

In Section 4.6 a more detailed description of the problem of exploration versus exploitation is given, including the description of the algorithm used by our agent.

4.4 State and Action space

The continuous environment found in curling (see Section 4.1.1) presents a problem for the Q-learning described in Section 4.3.1. As described earlier, the main idea with the Q-learning is to use experience to gradually learn the optimal value function, which is the function that predicts the outcome that an agent could receive given a state and when an action is applied. The continuous environment creates a problem for the learning because the value function must operate in a space where states and actions are represented by real-valued values, which means that the value function must be able to evaluate an infinite number of state and actions. Learning in such conditions becomes very difficult because it is very unlikely that an agent could be able to experience exactly the same situation it has before.

A common approach to the problem is to discretize the state and action space into a finite number of discrete state and actions. It is one of the simplest forms for generalization, but has the drawback of compromising between accuracy and efficiency. In order to achieve accuracy, the discrete states must be defined fine enough to prevent aliasing, where functionally different situations map to the same state and are thus indistinguishable. Efficiency is achieved by using a rough quantization of the state and action space in order to reduce the computational load, and thereby fastening the learning time. However, rough discretization is often the cause for bad performance, or the divergence of the learning policy. The process of reducing the state space by experience and expertise (designed) is also very strenuous because of the difficulty of foreseeing the impacts the design could have on the learning. There are several studies on how the drawbacks of quantization can be reduced. Tuyls et al. [12] have used Bayesian networks for quantization of large state space and Uchibe et al. [23] proposes modular reinforcement learning as an effort to account for the compromise between learning and performance.

Other approximation techniques

There are other approaches to the problem of using Q-learning in a continuous environment. One important approach is to avoid the problem associated with quantifying state and action space by using other types of approximations when generalizing the value function needed by the Q-learning.

Several attempts in extending the Q-learning framework to include continuous state and action space suggest other options that can possibly avoid the drawbacks of quantization [22, 8, 19]. The next four paragraphs describes a selection of techniques that allow real-valued state and action in the Q-learning.

Q-Kohonen Touzet [22] describes a Q-learning method that uses artificial neural network to improve the learning. The method uses Kohonen's [10] self organizing map where the state, action and expected value are the elements of the feature vector. Action is selected by choosing the node that matches the state and the possible max value.

Neural Field Q-learning Gross et al. [8] describes the use of dynamic neural fields to distribute reinforcement learning in continuous state and action space. A neural field can be viewed as a recurrent neural network that receives topographically organized information. Neural field were used to encode the expected action values and similar states were clustered by using neural vector quantizer (Neural gas).

CMAC Based Q-learning Ram et al. [19] have used Cerebellar Model Articulation Controller or CMAC [1] in continuous Q-learning. The input or state-action pair activates a specific set of memory location and the arithmetic sum of whose contents is the value of the stored Q-value. The technique is a compromise between a weight-based and a look-up table based approximation.

Curling State Model

We have previous in this section described options available for Q-learning in continuous environment. Because of the uncertainties about possible gains versus the risk of adding more complexity to the learning by using the other approximation techniques, we have chosen to design our model by quantizing the space into discrete states using experience and expertise. To model the game we need to define a series of states, that is both detailed enough to clearly illustrate curling but at the same time be brief enough that the problem does not become infeasible.

There exists little literature about analytical models for the curling sport, compared to other sports such as Baseball (D'Esopo and Lefkowitz [5, p.55-62]), Cricket (Clarke and Norman [4]), and Snooker (Percy[14]). Kostuk et al. [11] suggested modeling curling as a Markov process, MP, using a round by round representation. The paper suggested that there are two types of natural representations in curling; a shot by shot model of the game's progression, or a round by round representation. The model suggested by Kostuk was round by round.

Their model is describing two pieces of information that all competitive strategies in curling should depend upon, namely, the score and whether or not the team has the hammer. The MP model uses a state space described by the vector $\{i, j\}$ where i is the difference in score and j indicates the possession of the hammer. Picture that we have teams A and B, and B is selected as our reference. Whenever B is leading, i is positive, and in reversed, if A is selected and B is leading, i is negative. For the second part of the tuple, if the j value is positive, $j = 1$, B has the hammer. Figure 4.4 show the state transitions with k defined as the number of points scored in a round.

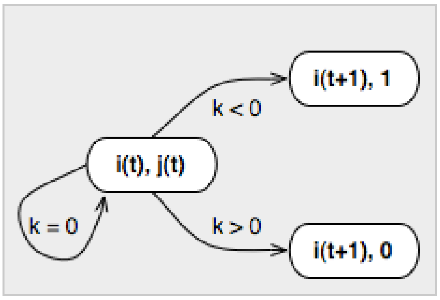


Figure 4.4: Markov chain representation of scoring transition from state $\{i, j\}$ with k defined as the number of points scored in a state.

The model is capable of analyzing end-by-end results as a curling progress.

However, we want to build a model that can be examined during a round, where we can use RL in selecting with the current state to select an appropriate strategy (action group). Kostuk's representation may contain enough information for planning strategies over rounds, but not for a rock-by-rock strategy.

Our State Space To keep the complexity of our model as low as possible we decided to use the same features described in Section 4.2.2, with some minor modifications. The state used is defined by the vector showed in Equation (4.16) and in Table 4.1 for ease of reference.

$$S = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9\} \tag{4.16}$$

State vector description.

x_1	the points in a state (equal to attribute i).
x_2	the number of rocks the player has in house.
x_3	the number of rocks the opponent has in house.
x_4	the number of rocks belonging to the player that is guarded.
x_5	the number of rocks belonging to the opponent that is guarded.
x_6	the number of corner guards.
x_7	the number of center guards.
x_8	the rock the player is setting (0-7, 0 is first rock).
x_9	the agent has the hammer.

Table 4.1: The description of the vectors in the state model.

Not that we have two real numbered features, namely x_2 and x_3 that would if used directly created a infinite number of possible states to evaluate. Note also that x_8 is not part of the features found in Section 4.2.2, but was included to enable the RL to distinguish the parts of a round (e.g. beginning, middle and end) and hence select action propitiate to the progress of a round. This attribute is needed for allowing the agent to possibly recognize the importance of last rock in a round, and possibly strategic advantages early in a round (e.g. place guards early in a round).

State Complexity Table 4.2 shows the number of distinct state descriptions we can have. The number of distinct states for our model has an upper boundary of $5.2 \cdot 10^8$. The precise number is expected to be lower because of the dependency between the features excludes many states. For example, it is not possible to have a state where all rocks are guarded and take points simultaneously, so the state $x_1 > 0$, $x_4 = 16$, $x_5 = 16$ does not exist.
The state is more complex than the model suggested by Kostuk [11], but we believe the added complexity is needed when we want the agent to learn rock-by-rock strategies.

Attrib.	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
Values	17	9	9	9	9	17	17	8	2

Number of combinations: $17^3 \cdot 9^4 \cdot 8 \cdot 2 = 5,2 \cdot 10^8$

Table 4.2: Description of the attributes in the state model.

Curling Action Model

The effectors used in curling can be described by a real-valued vector $\mathbf{u}$ with the following elements:

- u_1: the x value of the aiming point.
- u_2: the y value of the aiming point.
- u_3: the split time (the time the rock use on traveling between the two hog lines).
- u_4: the curling direction, either out- or in-curling (see Section 1.3.2 setting a rock).

The dimensionality of the space is much smaller compared to the dimensionality of the state space, but its real-valued elements presents the same problem as continuous state space for both the linear target function in Section 4.2 and the Q-learning in Section 4.3.1 with infinite number of variations.

To solve the problem of infinite number of action variations we decided to reduce the space by limiting both the boundary and granularity of the elements u_1, u_2, u_3 and u_4.

When discretizing the effector vector we must apart from accuracy and performance account for the nondeterministic environment (see Section 4.1.1) when designing the action space. The nondeterministic behavior of the environment can be observed as noise being randomly added to the effectors. Having this noise or "random behavior" lowers the boundary for where accuracy gains is outgrown by the performance cost. Operating with noise creates a point where finer granularity has lesser impact on accuracy but still the same negative impact on performance.

The following paragraphs describes the finite action space created for both the linear target function learning and the Q-learning.

Linear Target Action The granularity of the action and the size of the actions space will be found by adjusting the granularity of both the x-value and the split-time independently in an effort to provide a satisfactory ration between accuracy and performance (subjectively through testing in the curling simulator).

The y-value has because of both its properties and the nondeterministic environment little impact on the result of an action, and will therefore be set to a constant value.

The size of the action space is:

$$2 \cdot \frac{x_{\text{upper}} - x_{\text{lower}}}{\Delta x} \cdot \frac{s_{\text{upper}} - s_{\text{lower}}}{\Delta s} \tag{4.17}$$

where Δx and Δs is the granularity selected for the x value and split time value, respectively.

Q-Learning Action The action space for the Q-learning will be created by further discretization. However, the discretization will not be done by adjusting Δx or Δs in Equation (4.17) but by first sup-group the environment into action zones (Figure 4.5)

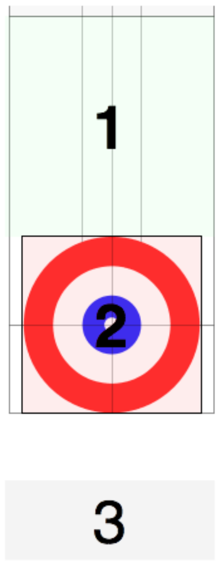


Figure 4.5: The Area the different action groups operates in.

Action Groups

1. Guards
2. In House
3. Take out

It can be arguments for creating more action groups, one example and maybe a natural choice could be to split the *Guard* group into *Center-* and *Corner-guard*, and similarly splitting the *In House* into *Front-* and *Back-House*. However, the branching factor has a significant negative impact on the time required to train the Q-values and can not be ignored. High branch factor, i.e. more action groups and finer granularity creates more options at each state that needs to be tested, and thereby will prolongs the training period.

The three groups selected play an important role in curling as they capture some of the key aspects of both offensive and defensive play. For example, if an agent is of a defensive mind, it will prefer to keep the opponents rocks out of play and hence leads to selecting *Take out* over Guard actions. Offensive minded will in similar way select *Guard* action over *Take out* throughout a round to protects its rocks and hence create opportunity for high reward. The foundation for a curling playing agent is to recognize the situations where offensive, and vise versa, defensive strategies are appropriate. We believe that the agent will be able to learn and recognize the difference, and apply them efficiently against its opponents with our model.

4.5 MDP and our Q-Learning agent

Markov decision process or MDP have three important presumptions that must be present in the domain being modeled. In this section we will shortly describe how the presumptions are represented in the models used by our Q-learning agent. A reinforcement learning task (including Q-learning) that satisfies the Markov property is called a MDP.

MDP comprises of the following three presumptions:

- that the policy π is a direct function of the states S.
- the Markov property
- that the probability $P(s_{t+1}|s_t, a_t)$ is independent of t

Is π a direct function of S?

The first presumption states that for every state s there is a single action a that maximizes the expected cumulated discounted future reward. In the curling domain this means that there is a single action for every state that maximizes the chance of winning.

It is difficult to determine if this is the case for all possible states. However, if we assume that for all opposing strategies there exist an optimal series of actions to perform that maximizes the winning chance, the presumption will hold. This assumption is supported by the fact that most strategies in curling are based on such series of actions.

The Markov Property

The second presumption or the Markov property states that the result state s_{t+1} depends only on the current state s and action a ($s_{t+1} = \delta(s_t, a_t)$).
The presumption depends on how we defined our state description. Our state description (see Section 4.4) is a rock-by-rock model that will in the boundary of a round fulfill this presumption. However, in the first state in all rounds after first round we will have the exception where the result state of the first action (setting first rock) could be dependent on the final state in previous round and the current state. This is a result from the relationship found between rounds related to playing a hammer round.

$\mathbf{P(S_{t+1}|s_t, a_t)}$ independent of t?

The presumption states that the probability of ending up in a state s when taking action a will be the same for independently for what point in time the the action is used.

For our rock-by-rock model the answer will depend on the opposition. Playing against an agent that has knowledge about the total score and rounds played could use such knowledge to its advantage and hence show different behavior in equal round states resulting in a $P(S_{t+1}|s_t, a_t)$ that is dependent on time.

However, for our experiments we will use opponents that does not change theirs behavior between rounds. But against opposition showing such behavior it will be necessary to reassess the defined states so they can include the results from all previous actions.

Consequences

When creating the state model that will be used by the Q-learning agent we had to accommodate the three presumptions stated. From our description of how our model manages to represent the presumption we can see that the model does not completely pass as MDP. The difficulties arise when trying to both limit the state space size while simultaneously making the models expressive enough to fulfill the MDP presumptions.

However, we believe that not fully capturing the last presumption ($P(S_{t+1}|s_t, a_t)$ independent of t) will have little consequence for our results related to the Q-learning.

4.6 Exploration vs Exploitation

In a non-deterministic, partially observable, dynamic and sequential environment, how should decisions be made? Two paths can be followed to obtain good strategies:

Exploit: act optimally accordingly to our current beliefs.

Explore: learn more about the environment.

The environment must for both options be taken under consideration. For example can nondeterministic environments create cases where a state-action pair is wrongly valued. That is, cases where "noise" skews the result of an action, and hence skewing the value of the action. We can also find similar problems when operating in a partially observable environment, where wrong assumptions or an outdated state model could lead to similar skewing of the estimated value of an action. However, in any environment it has been considered that neither exploitation nor exploration can be pursued exclusively without failing at the task [21, p.4].

4.6.1 The K-armed bandit problem

The bandit problem is maybe one of the most generic ways to model an exploitation-exploration. The k-armed bandit is a slot machine with k-arms, each arm with an unknown expected return. The objective of the gambler is to maximize the sum of rewards through iterative plays. The problem in bandit is to find a balance between reward maximization based on the knowledge already acquired and attempting to further increase knowledge.

A K-armed bandit (first described in Robbins [16], 1956) problem is defined by random variables $X_{i,n}$ for $1 \leqslant i \leqslant K$ and $n \geqslant 1$, where each i is the index of a gambling machine (i.e. the "arm" of a bandit). Playing a machine i in succession yield the rewards $X_{i,1}, X_{i,2}...$, which are independent and identically distributed but with an unknown distribution u_i. The independency holds also across machines; i.e. $X_{i,s}$ and $X_{j,t}$ are independent for each $1 \leqslant i,\ j \leqslant K$ and each $s, t \geqslant 1$.

Proposed solutions to the bandit problem The bandit problem has many proposed solutions. Four examples are:

- **ϵ-greedy exploration:** choose apparent best action with probability $1 - \epsilon$, or random action with probability ϵ.
- **Boltzmann exploration:** is a more sophisticated exploration strategy that does not force two choices like ϵ-greedy, its selection is based on the Boltzmann distribution. At each time step t each action a_i has the following probability to be selected.

  $$P(\pi(s) = a) = \frac{\exp(\frac{1}{\tau} \cdot Q_t(s, a))}{\sum_{j=1}^{n} \exp(\frac{1}{\tau} \cdot Q_t(s, a))}$$

  where:

  - n is the number of actions.
  - $Q_t(s, a)$ is the Q-value of the state-action pair in current time step t.
  - τ ($\tau \in \mathbb{R}$ and $\tau \geqslant 0$) is the temperature.
- **Optimistic exploration:** choose an arm that has a possibility of being the best.
- **Bayesian exploration:** assign prior to the arm distributions and based on the rewards, choose the arm with best posterior mean, or with highest probability of being the best.

4.6.2 The UCB algorithm

Upper Confidence Bounds (UCB) algorithm (Auer et al. [2]) is another proposed solution to the bandit problem. The algorithm must first be initialized by playing each arm once. After the initialization we have at each time n, select an arm using Equation (4.18).

$$a = \arg\max_k B_{k,n_k,n}, \tag{4.18}$$

with

$$B_{k,n_k,n} = \underbrace{\frac{1}{n_k}\sum_{s=1}^{n_k} x_{k,s}}_{\hat{X}_{k,n_k}} + \underbrace{\sqrt{\frac{2\log n}{n_k}}}_{c_{n_k,n}} \tag{4.19}$$

where

- n_k is the number of times arm k has been pulled up to time n
- $x_{k,s}$ is the s-th reward obtained when pulling arm k.

The idea behind the algorithm is to select an arm that has a high probability of being the best, given what has been observed so far.
The size of the confidence interval $c_{n_k,n}$ decreases as the number of times an arm k, (n_k) is pull and is increased if the arm is pulled significantly less than the other. The confidence interval guarantees that all arms, independently of its expected reward will be tested again and again, but at exponentially longer intervals.
The term regret is often used as a measure when comparing action-selection algorithm. Regret for an algorithm is defined as the amount lost by using the algorithm rather than selecting the optimal solution (arm) each time. An algorithm that has a regret that grows no more than logarithmically is proven to be optimal [3].

Using the regret measurement, UCB has been proven to achieve logarithmic regret without requiring any preliminary knowledge about the reward distribution (apart from the fact that they are all inside a bounded interval) making it an optimal solution.

Q-learning and UCB Using Q-learning in the reinforcement learning allows us to simplify the equation and notation stated in Equation (4.18) and (4.19) to:

$$a = \arg\max_a \{\ Q(s,a) + \hat{Q}_s \cdot C_{T_{t-1}(a),T_{t-1}(s)}\ \} \tag{4.20}$$

where

- $T_{t-1}(a)$ and $T_{t-1}(s)$, is the number of times the action and state has been used, respectively.
- $\hat{Q}_s$ is the average Q-value for state s. We choose to use the value to scale the confidence interval to match $Q(s,a)$ values.

4.6.3 UCT (UCB applied to Trees)

UCT is a bandit based tree search method based on UCB suggested in Kocsis [9]. The method has shown its strength in problems with huge trees, e.g. in the game of Go [7], with its effective exploration of the trees. UCT treats each node (state) in the tree as an independent bandit, with its child-nodes as independent arms. The algorithm is rollout-based, which means that instead of handling each node iteratively, it simulates playing a sequence of bandits within limited time each starting in the root and ending at one leaf.

The pseudocode of a generic Monte-Carlo planning algorithm is given in Figure 4.6, which shows how the algorithm builds a lookahead tree by repeatedly sampling episodes from the initial state. An episode is a sequence of state-action-reward triplets. We can see that the algorithm iteratively generates sequences (line 3) and returns the highest average long term reward (line 5). The method *selectAction* (line 9), would be a method that uses UCB to select an action when using UCT.

```
 1: function MonteCarloPlanning(state)
 2: repeat
 3:   search(state, 0)
 4: until Timeout
 5: return bestAction(state, 0)

 6: function search(state, depth)
 7: if Terminal(state) then return 0
 8: if Leaf(state, d) then return Evaluate(state)
 9: action:= selectAction(state, depth)
10: (nextState, reward) := simulateAction(state, action)
11: value := reward + $\gamma$ search(nextstate, depth + 1)
12: UpdateValue(state, action, q, depth)
13: return q
```

Figure 4.6: The pseudocode of a generic planning algorithm.

Simulating Curling The environment description of curling (see Section 4.1.1) is a complex environment where lookahead planning is very difficult. There are several factors that make the planning (e.g. running *search*(state, depth) with depth > 1, line 3) toilsome. One key factor is the combination of being dynamic and nondeterministic. Planning several rocks ahead in one round introduces an unknown level of uncertainty.

When planning in environment such as curling it is important to address the following problems:

Opponent strategy Knowing the opponents strategy is an important component when planning. Without such knowledge it becomes more difficult to find and commit to an effective strategy since they are often dependent on the other opponents goals and behavior. Exploration such information will require the agent to adapt the observed behavior of the opponent into a behavior model that can predict the opposition.

Continuous Environment Operation in such environment removes the option of directly searching through the state space. The infinite size makes such task unsuitable. Discretizing and hierarchical decomposition (top-down planning) is example of techniques that can help the planning.

Round and Match The curling relation between a *round* and a *match* further complicates the planning task. Curling strategies are no necessarily restricted to the scope of a round. We expect that in order to predict an agent strategy (assuming that the opposing agent is rational) it would be necessary to both determine round based strategies and long term match strategies.
Evaluating the difficulty of planning in the scope of our simulator and curling environment it was decided not feasible in this project. The agent implemented and described in Section 5.3.2 (Q-learning + UCT) will therefore execute no planning when selecting *bestAction*. This is equal to remove the recursive search in line 11 in Figure 4.6.

Chapter 5

SkippySimulator

In this chapter a detailed description of SkippySimulator and the curling-playing agents is given. This entails describing the architecture, main component processes, and the main part of their implementation.

5.1 Architecture

The purpose of this section is to present an abstract overview of the architecture using UML models to visualize the main elements of SkippySimulator's architecture and reason about how the elements and their relationships support the goals and constraints stated in Chapter 3.

> The main purpose of the simulator is to provide a basic, underlying system for testing curling playing agents.

Section 3.2 describes both the requirements and constraints for the simulator. They will be addressed by creating a curling simulator framework named SkippySimulator. Creating this framework presents both advantages and disadvantages

Advantages

- The process of creating a framework forces us to develop a clear picture of what features and requirements are wanted and required.
- It facilitates code reuse. There is no sense in "reinventing the wheel". By providing the simulator as a framework it can be used in future projects.

Disadvantages

- Framework design is difficult. It is hard to structure the system in such a way that it can fit into someone else's program and still perform the task it is designed to do.

- A proper framework should have all the functionality that a user needs, something that is very difficult to achieve.

5.1.1 Design

The architecture for SkippySimulator is characterized by its use of the Model-View-Control pattern (Reenskaug [15]). Figure 5.1 illustrates the components and their dependency relationships. The task of creating new agents is simplified by separating the agent from the simulator and providing a unified agent interface. With this architecture new agents can be created without having acquired knowledge about the inner workings of the system.

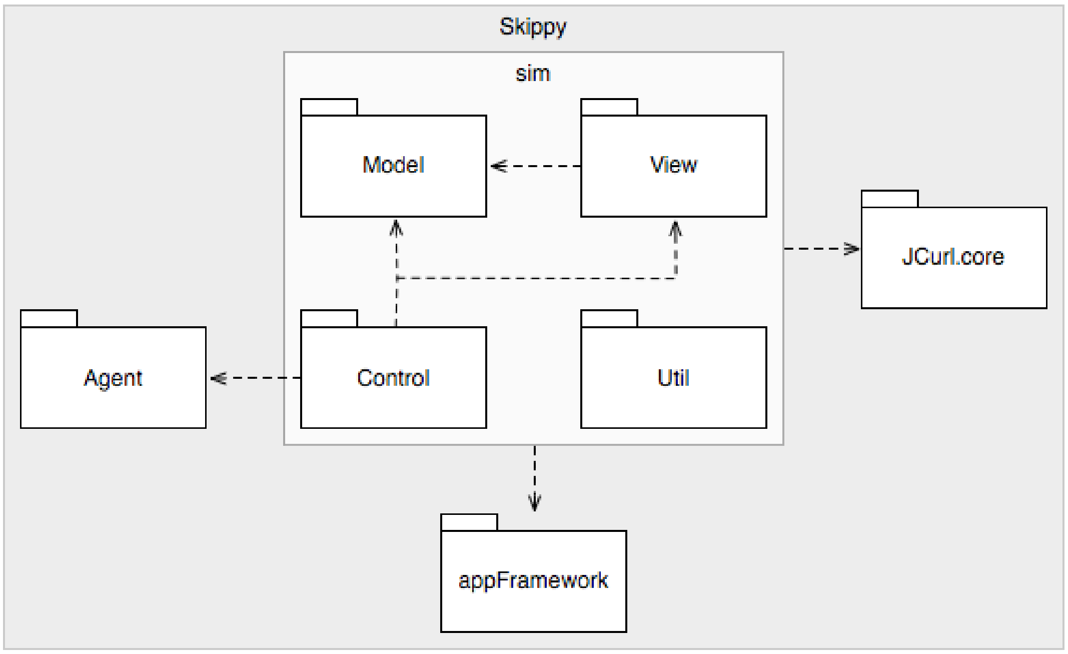


Figure 5.1: Software Architecture of SkippySimulator

Agent: This is the only package a user of the simulator framework needs to think about. It contains the class BasicAgent, that new agents needs to inherit from. As well as a set of hook methods that must be implemented when creating agents.

Control: This package contains control classes. The two most important classes here are *SimulatorApp* which creates and controls the GUI, and the *SimulatorTask* which manages curling simulations. It is responsible for setting up, manipulating, and driving a simulation.

View: Has the class for viewing a curling game.

Model: This package contains the different types of data-structures used by the framework. The class *RockCollection* is the structure used to manage the curling rocks.

Util: The utilities package contains a variety of utility classes used both internally by the controllers and by the views. For example *Zoomable* that contains a number of static methods that allows views to zoom and pan.

External Dependencies: Figure 5.1 shows the two main dependencies SkippySimulator has. AppFramework provides application infrastructure and JCurl handles rock collision and graphics.

5.1.2 Architectural Rationale

In this section we explain our architectural design decisions.

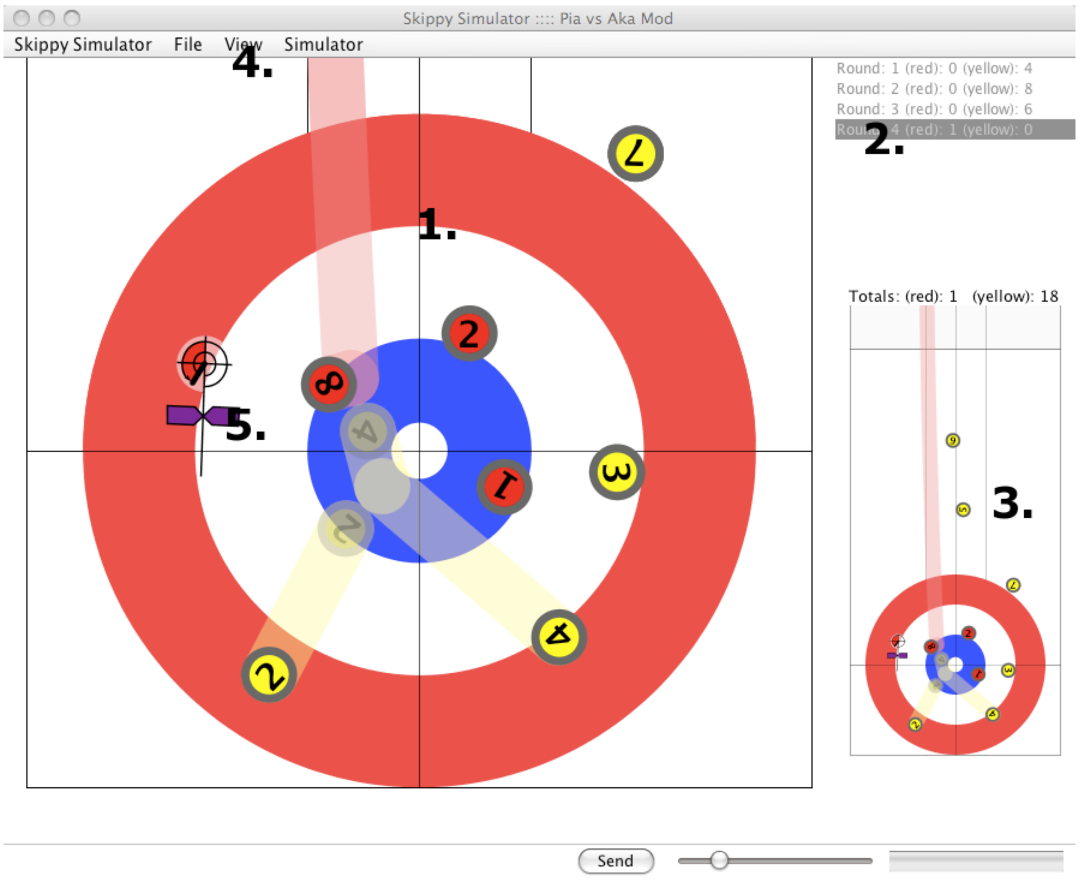
JCurl Framework: The framework comes with many typical curling concepts out of the box. Its comprehensive support for important features such as curl, collision, storage and display allows us to focus more on the agent interaction in Skippy. It will allow us to use more time on agent development rather than implementing the SkippySimulator framework.

AppFramework - Swing Application Framework API: The framework simplifies the building of SkippySimulator by having a defined infrastructure common to most applications that we can base our framework on. Strong points are managing application lifecycle (startup, shutdown), loading resource and action binding (also asynchronously). The use of this framework will shorten the time needed for implementation.

Patterns Rationale: By using the MVC pattern throughout the framework we will get a system with low coherence. This will greatly reduce the difficulty of changing or adding functionality at a later time, but it also makes it easier to cooperate when there are several people working on the same project.

5.2 Main Components

In this section we will describe the main components of the SkippySimulator. A screenshot can be seen in Figure 5.2. The five main components of the application are; the main playing area, the scoreboard, a birds eye view of the house, the menu bar and the interactive controls.



Round: 1 (red): 0 (yellow): 4
Round: 2 (red): 0 (yellow): 8
Round: 3 (red): 0 (yellow): 6
Round: 4 (red): 1 (yellow): 0

Totals: (red): 1 (yellow): 18

Figure 5.2: Screenshot of SkippySimulator's user interface.

The Main Playing Area(1): The main playing area is a zoomable view of the ice. The player is able to pan and zoom as he sees fit to get the best view for the shot. At any given time the location of all rocks in play are shown. After a shot have been made, paths are drawn to better visualize how the rocks moved across the ice. The icon marked 5 in the screenshot is the broom. The player uses the broom to aim when making shots.

The Scoreboard(2): Situated in the upper right corner of the screen is the scoreboard. This is a listing showing how many points were awarded in each round of the game. The totals for both players at any given time are shown at the bottom of the board.

The Bird's-eye View(3): Below the scoreboard is the bird's-eye view. This is just to aid the player during the game. While the main playing field can be zoomed and panned at will, the bird's-eye view always shows just the area where stones are in play. This includes the area from the hog line to the back line at the playing end of the ice.

The Menu bar(4): The menu bar, situated at the very top of the screen includes options for starting the game with human players or computer agents or any combination thereof. There are options for zooming that are designed to help the player. You can for example automatically zoom in on the house.

The Interactive controls(5) The interactive controls for human players are all encompassed in the "broom" seen in the lower right area of the playing field on the screenshot. The broom is shaded in the color of the player whose turn it currently is. By moving the broom around the player indicates where he wishes to aim and the direction the stone will curl is indicated by the thick black line on the broom. The direction the stone will curl can be switched by double clicking on the broom. The purple slide just below the broom is used to adjust the strength of the throw. Finally, clicking the send button located at the bottom of the window will send the rock when the player is satisfied with his/her aim.

5.3 Implementation

In this section we discuss how SkippySimulator and its curling agents are implemented.

5.3.1 SkippySimulator

This section describes the most important aspects of the implementations of the simulator.

BasicAgent

The class is abstract and consists of a set of hooks methods implemented from the *Agent* and *InterActive* interfaces (see Figure 5.3). The class implements resource management (*getResource*, *addResource*) This enables agents to read data from disk. Agents should use the *loadResouces* method for queuing what resources the simulator should load before calling the agent's *setup* method.

Agents that subclass BasicAgent will have the option of overriding one or more of the remaining methods provided by its interface. So, for example, if you want to create an agent that use the mouse to set rocks you would simply override the *isInterActive* method (see example in Listing 5.1).

The methods *setup* and *tearDown* are pre/post game methods that are only called before and after a curling match. These methods are meant for one time tasks such as reading data from disk, initializing agent properties and writing data to disk.

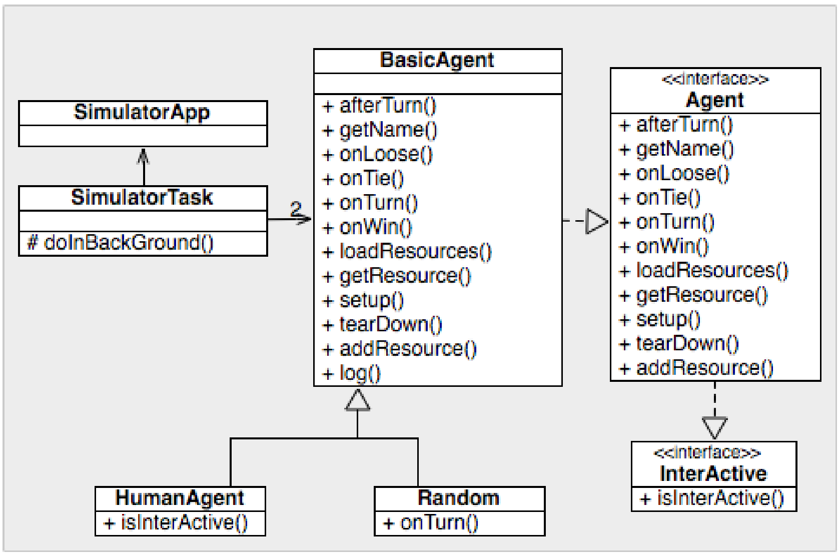


Figure 5.3: BasicAgent class

The methods *onTurn* and *afterTurn* are the pre and post methods for setting a rock. In *onTurn* the agent get access to the rock positions, and are able to set its rock. The *afterTurn* allows the agent to see the results of its actions before the other agent sets a rock.

```
class HumanAgent extends BasicAgent {
    public boolean isInterActive() {
        return true;
    }
}
```

Listing 5.1: HumanAgent example code

Application control

The control of the simulator is handled by several classes; the most important ones are *SimulatorApp*, *SimulatorTask* and *RockBroomControl*. *SimulatorApp* is the main entry point of the program. This class holds all the important components of the game such as the broom and the rocks model, also it ensures that the GUI is drawn properly.

When a game of curling is ready to begin, *SimulatorApp* creates an instance of *SimulatorTask* with the appropriate players. A player is an instance of a subclass of *BasicAgent* as described in the previous section. The *SimulatorTask* class handles the progress of a curling match, ensuring that the players get to take their turns and that the score is calculated correctly at the end of each round. This class also handles sending notifications to the players about events in the game, but this is only relevant for the automated agents as the human players will simply see what happens on their screen.

SimulatorTask utilizes the class *RockBroomControl* to ensure that the correct rock is placed at the hack when a player is about to take his/her turn and that the rocks behave correctly when collisions occur on the ice.

5.3.2 Agent implementation

With the usage of the new framework two main agents where created, *SkippyAgent* and *RLSkippyAgent*, where' the latter one builds upon the foundation of the former.

SkippyAgent

SkippyAgent (see Figure 5.4) is the implementation of the Linear Agent described in Section 4.7. The most important function is *doBestMove*(), which is called when it is the agents turn to set a rock. It uses a Linear Target function to determine where to set the next rock. The best move is found by testing a multitude of actions, then analysing the top three to see which one is the best alternative after the probability of missing/failing is accounted for. The best action with least chance of missing is then selected and comunicated to the simulator. Then it is the oponents turn. When a round is ended (when either *onWin*, *onLoose* or *onTie* is called), the feature weights are adjusted in accordance with the result.

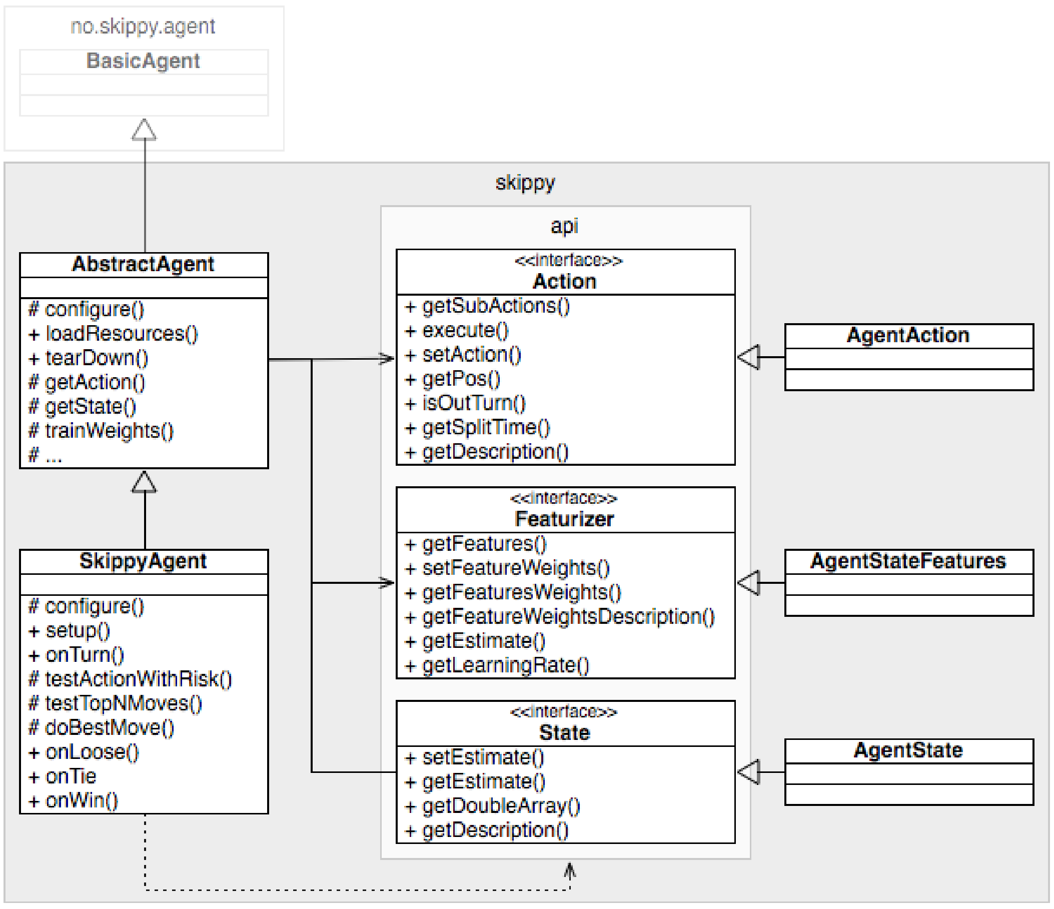


Figure 5.4: UML diagram for SkipppyAgent.

The superclass *AbstractAgent* was created to handle pre and post functionality. It mainly support easy access to loading and saving an agents knowledge. The class extends from *BasicAgent*.

API This is the package that contains the interfaceS used to facilitate and simplify the implementation of an agent. The following paragraphs describes their purpose and main properties.

Action The interface that handles the agents output. The implementation of *getSubAction* makes it possible to create a hierarchy of actions, where sub-action can for example be lower level actions. The *execute* method sets the output as specified in the action.

AgentAction is the implementation created for *SkippyAgent*.

State The interface for implementing the input from the environment that an agent perceives. The *getDoubleArray* should return an array with the information captured in the state. Any two states are identical if and only if they return identical arrays.

AgentState is the implementation used by *SkippyAgent*, it implements the features described in Section 4.2.2.

Featurizer The interface used for describing a state as a series of features with weights. SkippyAgent uses an implementation of this interface, *AgentStateFeatures*, for estimating the value of a state by using Equation (4.7) as described Section 4.2.

RLSkippyAgent

RLSkippyAgent (see Figure 5.5) is the name of the agent that was implemented to use Reinforcement Learning as described Section 4.3.

The agent uses RL to learn the value of a set of action groups[1] given its current curling state. UCT as described in Section 4.6 is used by the agent to control its explorative versus exploitive behavior.

API RLSkippyAgent is dependent on the following classes and packages.

[1]Action group is a collection of actions that share similar properties.

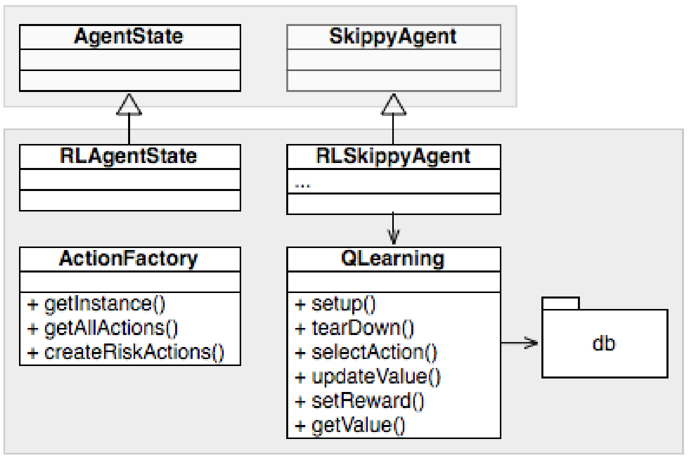


Figure 5.5: UML diagram for RLSkippyAgent.

RLAgentState This is the implementation of the state model described in Section 4.4. The class share the properties found in *AgentState* except it is modified to use only discrete properties and it has an additional feature that describes what rock is next.

QLearning Implements the Q-learning algorithm described in Section 4.3.1. The two most important methods in this class are *selectAction* and *updateValue*. The method *selectAction* implements UCT as described in Section 4.6.3. However, it does not use a lookahead approach, thereby making the selection equal to UCB using Equation (4.20). Method *updateValue* uses the nondeterministic Q-learning rule, Equation (4.13) for updating the value of $\{state, action\}$ pairs in the Q-table. *setReward* is only used on final states (e.g. after the last rock in a round), where the reward is set based on the outcome from the round.

DB package This is a support package used by the *Qlearning* class to manage the Q-table. The size of the state space made it necessary to provide a solid and self managed data model for the large Q-table. This package provides the Q-Learning with access to a database for creating, retrieving and updating entries in the Q-table. The database also enables us to easily run distributed training of the agent, since the database can be accessed from anyywhere.

ActionFactory The *ActionFactory* provides the set of action groups that *QLearning* selects from.

The method *getAllActions* returns a set containing the following groups:

- *Guard* Actions that tries to set guards.
- *Take Out* Actions that can be used to take out rocks.
- *In House* Actions that can place the rock in house.

Chapter 6

Results

This chapter lists the experiments done using the curling simulator. Each experiment is described by its setup and potential weaknesses/expectations. For all experiments we will give a discussion of the results and a conclusion.

6.1 Experiment 1 - Linear Target Function – Single Weight Set

The first experiment concerns SkippyAgent as described in Section 5.3.2 and involves training the feature weights (Section 4.2), as well as running simulations against opponents to see how the agent performs.

6.1.1 Setup

The training of the agent will hold two parts:

- Part 1 - Train SkippyAgent against a random agent, named AKA.
- Part 2 - Train SkippyAgent against a version of itself where the weights have been frozen.

For both training phases we will use a decaying learning rate that is initialized with $\eta = 0.005$ (see Section 4.2.3). The learning rate will have decayed to half its value after 200 rounds of training. The feature weights will be initialized with random values in the interval $-10 < w_i < 10$.

The opposing agent in Part 1, *AKA*, is an agent that has one goal; set all its rock in the house. However, that agent has random noise added to its effectors to create diverse situations that SkippyAgent can learn from. *AKA* is stateless and does not evaluate its input before commencing to set a rock. The opposing agent in Part 2 will use the trained weights from Part 1, but they will be frozen (i.e. the learning rate is set to 0).

Upon completing the two phases of training we will test SkippyAgent in a 5-match tournament. The opponents in the tournament will be *AKA* and an agent named *Scott*. *Scott* is a simple agent that has a more defensive strategy (when compared to AKA) that uses *take-out* actions to keep the ice empty of rocks.

6.1.2 Weaknesses/Expectations

Before we initialized the first training we tried to identify possible weaknesses with our setup, including the assumptions that the agent is built upon and the environment it operates in.

The most important weakness is the feature vector used by the agent. The set of features selected makes the agent incapable of separating all functionally equal states. This means that many different situations will be regarded as being in the same state, which could make it more difficult for the target function to converge.

Another issue is the discrete features *Corner-* and *Center-Guard*. The problem is that it is difficult to differentiate guard actions when the ice is empty, because all actions have the same value. The value of a guard action will only differ if the action changes the feature set by either guarding a new rock or moving/removing an already set rock. The result of selecting a guard action when the ice is empty will be to try to set the rock to a fixed guard position (for example first or last guard action checked). This behavior could disable the agent from learning the advantages of setting guards early in a round in an effort to control access to the house.

The third weakness we identified is related to *Last Stone Strategies* described in Section 1.3.3. The feature set does not capture the difference between rounds where the agent has the last rock, and rounds where it does not. This could make it difficult to train weights that accommodates both situations.

We also discussed an issue related to how the features are trained. Equation (4.2) and Equation (4.3) states that we will update the features after a round by successively adjusting the weights using the successor state.

The possible problem that we discovered can be described as follow:

- We could have rounds where both parties play “perfect” rounds where the opponent always creates a negative valued state for the agent, and the agent creates a positive valued state afterward. This will for the agent result in a state trajectory that alternates from positive to negative valued states.
- Updating such trajectory using the successor state would mean that the agents “perfect” states will be adjusted negatively based on the value of the successor, independently of whether or not the outcome of the round was positive or not.
- The same problem could also occur if both parties play “poor” rounds, which could create the result that negative valued states (and bad states) are updated positively.

The most basic thing to learn is the advantage of setting rocks in house and taking points. We expect that the training of the agent’s features will result in weights that favor setting a rock in house and take points. But we hope that the agent will also realize the value of guarding, even if it does not win consistently.

6.1.3 Results

This section presents the result from Experiment 1. For training, the agent played a total of 40 matches, 20 against AKA and 20 against Scott. In total this amounted to 402 rounds (including tie-breakers) to adjust the weights. Figure 6.1 show how the different weights were adjusted throughout the rounds.

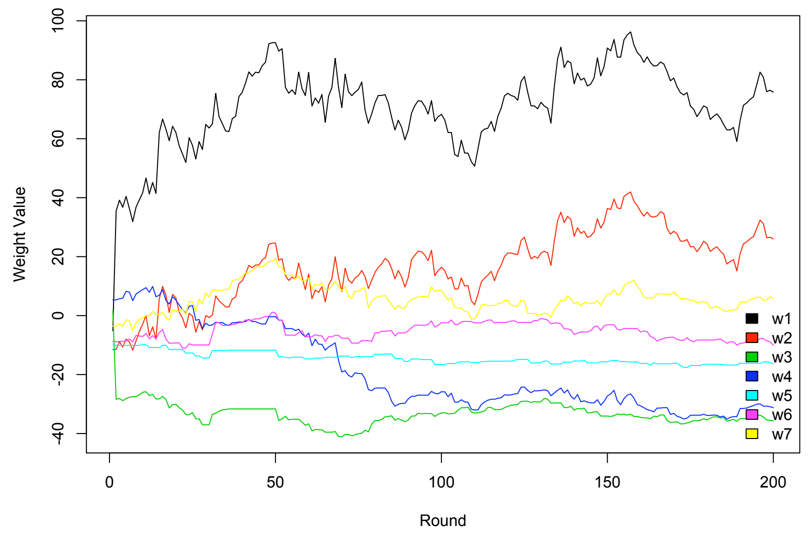
Finally the weights of SkippyAgent was frozen and a 5-match tournament was played against both AKA and Scott, the results of which can be seen in Table 6.1. More detailed descriptions of the results are shown in Table A.1 and Table A.2 in the Appendix.

Opponent	**Victories for Skippy**	**Total Points**	**Avg. pr. Round**
Aka	5	141-8	2.82
Scott	5	23-15	0.44

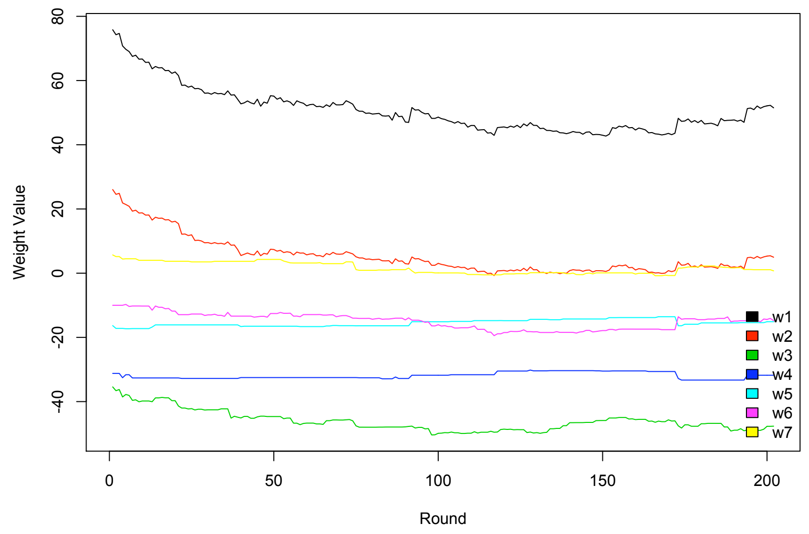
Table 6.1: Results from 5-Matches tournament.

6.1.4 Discussion

Discussing the learned weights independently may be difficult, but it is possible to extrapolate possible explanations and causes for the learned order and value of



(a) Training against AKA, training 20 match a 10 rounds.



(b) Training against itself, 20 match a 10 rounds

Figure 6.1: Weights for SkippyAgent, Figure (a) shows the weights when training against AKA, and Figure 6.1(b) shows the weights when training against itself (opponent uses constant weights) initialized with the previous trained weights.

some of the weights.

The first important issue is how the weights are ordered and whether or not if some weights change order and sign multiple times in the course of the training. The later case would tell that the agent had significantly problem with learning from the training situations.

From Figure 6.1(a) we can see that SkippyAgent was quick to learn both the order and sign of the weights. After completing 75 rounds we can see that both the order and sign remained unchanged.

The learned value of the two weights w_1 and w_2 (Points and Rocks in house) tells us that the agent was able to learn the advantage of setting rocks in house and take points. That both weights are positive can indicate that the agent has learned a very offensive strategy, where it wants to take as many points as possible.

Learning to guard was not as easily achieved. The training result in a positive w_7 (center guard) feature, but we argue that it failed to exploit the full potential of the guards related features w_4, w_5, w_6 and w_7. We believe that to fully exploit guarding actions the agent must recognize the value of guarding other rocks (w_4). The trained values of the guard related features tell us that the agent was not able to see the value of using guard actions and having guarded rocks. The ordering of the guard weights confirms our suspicion that our agent does not fully comprehend the usage of guarding action. For example, with the weight order $w_5 > w_4$ the agent will have a behavior that chooses to guard opposition's rocks over its own rock, a behavior that is not wanted.

However, evaluating the features separately is colored by our design ideas for the features, and it could be the case that we have yet to see the full advantage of the currently trained features.

The result from the tournament (Table 6.1) shows that the training resulted in an agent that had no problem competing against our selected opponents. The lower score against the defensive *Scott* agent confirms that the agent has problem to effectively use guards against defensive agents.

6.1.5 Conclusion

The experiment setting shows good potential, at least against the opposition used. We observed that the agent was able to learn a sensible order of the weights. However, it was not able to exploit the full potential of the guard related features

(w_4, w_5, w_6 and w_7). To address the problem of learning the guard features, we will replace the single feature set with two sets, one set for rounds when the agent has the last rock (hammer rounds) and one set for when the opposition has the last rock (normal rounds) in Section 6.3.

6.2 Experiment 1b

To address the a possible weakness described in the first experiment (Section 6.1.2) with how the feature weights are updated when training, we will in this experiment test an alternative updating policy: Instead of updating the weights with the estimated value of the successor state as described in Equation (4.2) and Equation (4.3) we will replace successor(s) with endState(s):

$$V_{\text{train}}(s) = \hat{V}(\text{endState}(s)) \tag{6.1}$$

The goal with the alternative updating policy is to train the weights more directly to the estimate of a rounds end state, instead of the indirect way by using successor states.

6.2.1 Setup

This experiment only differs from the first experiment with its alternative policy for updating its feature weights. All parameters, training, opponents and tournament match will be done exactly as described in the previous experiment in Section 6.1.1.

6.2.2 Weaknesses/Expectations

The similarity to the first experiment means that this experiment shares all the weaknesses we discussed in Section 6.1.2, apart from the one describing the problem of using the successor state to update the weights.

If the experiment is successful, it should prevent fluctuation in the features weights when training and enable a more steady convergence of the feature weights. However, our expectation for this experiment is divided. We hope that the alternative updating policy will enhance the learning, but it could also create the opposite effect, delaying the convergence of the weights or even prolong the training indefinitely.

6.2.3 Results

After training 250 rounds, the agent had won 179 and lost 76 times with an average of scoring 1.3 points pr. round. Figure 6.2 shows how the weights were adjusted throughout the training.

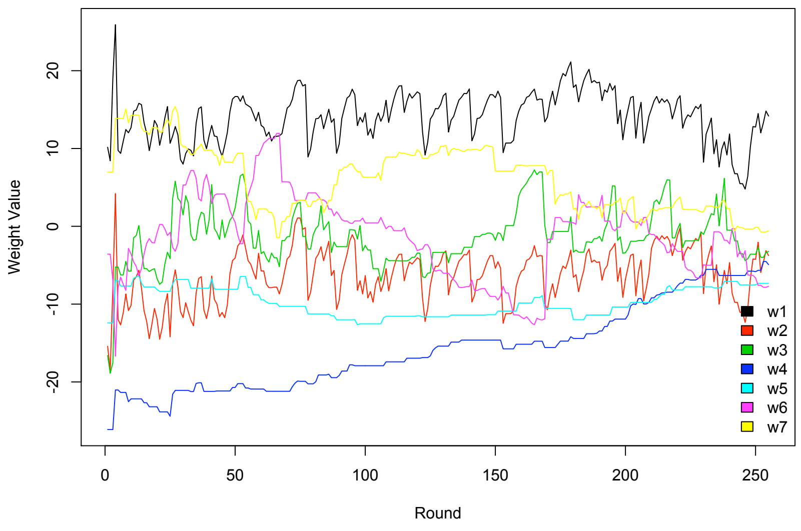


Figure 6.2: SkippyAgent weights from training while using alternative update policy.

Table 6.2 shows the result from the 5-match tournament played after completing the training. In the tournament the agent also played against the SkippyAgent from Experiment 1a. More results from the tournament are given in Tables A.3, A.4 and A.5 in the Appendix.

Opponent	**Victories**	**Total Points**	**Avg. pr. Round**
Aka	5	124-9	2.58
Scott	5	22-14	0.44
SkippyAgent	1	31-43	0.62

Table 6.2: Results from 5-Matches tournament.

6.2.4 Discussion

If we compare the result of the training of the feature weights showed in Figure 6.2 with the results from the previous experiment (Figure 6.1) we can conclude that the alternative updating policy did not have the wanted effect of easier convergence.

Figure 6.2 shows weights that display little signs of convergence after 200 rounds. We can for example observe that the weights $w3$, $w6$ and $w7$ have changed their value sign multiple times throughout the training. This is a clear indication of the problem the agent has when adjusting the feature weights using the alternative update policy. Using the end states introduces too much instability, which results in much more weight fluctuation compared to the result from the previous experiment.

Since we cannot observe convergence in this experiment all discussion around the trained feature values and their internal order becomes much more uncertain. However, there is one important difference to the weights in Figure 6.1 that we will mention. Through the whole training we can observe that $w_3 > w_2$. This means that the agent prefers having the opposition's rocks in the house over its own. Still w_1, which is the number of points taken, is the most positive of all states. We interpret this to mean that that agent likes to have stones in the house (including the oppositions) so that takeouts become difficult, and the stone closest to the button belonging to the agent so that it scores points.

The results from the tournament (Table 6.2) highlights the issue with the order of the weights. Learning a set of features values where $w_3 > w_2$ created a curling agent that was not as competent to take point against AKA and Scott when compared to the agent from the previous experiment (Table 6.1). The result from the match with the previous agent further confirms that the alternative policy is less effective on the curling problem.

6.2.5 Conclusion

The experiment was a failure; the alternative update policy had a severe effect on the convergence of the weights. For the remaining experiments we will go back to the updating policy used in the first experiment.

6.3 Experiment 2 - Linear Target Function – Double Weight Set

Experiment 2 involves modifying SkippyAgent to use an extra set of weights. The two sets will be used independently of each-other depending of whether the agent has the hammer or not. An agent that uses only one set of weights will have the same behavior with and without the hammer. However, this behavior will not always give the same result depending on who has the hammer. This will lead to problems for the agent with regards to learning, as a good move at one point might be really bad another time. We will refer to this agent as SkippyAgentHammer.

6.3.1 Setup

SkippyAgentHammer will be trained against a clone of itself. We will modify the simulator a bit for this training only so that one clone always gets the first shot, and then the other clone always has the hammer. Using this setup we ensure that both sets of weights get the same amount of training, and it also becomes easier to keep track of how the weights are adjusted as the training progresses. Once the training is finished, one agent will use both sets of weights in accordance with whether it has the hammer or not.

The training of both sets of weights will be done with a learning rate initialized at $\eta = 0.005$, which will decay to half its value every 200 rounds. All weights will be randomly initialized with values in the range $-10 < w_i < 10$.

6.3.2 Weaknesses/Expectations

With SkippyAgentHammer we hope to address some of the weaknesses for SkippyAgent. By using two separate sets of weights we hope that the agent will adopt separate strategies for when it has the hammer and not, and that it will be better in both cases. Still there are some strategies that are beyond the scope of this agent, specifically those that span an entire match. A match of curling usually consists of ten rounds, and the best strategy in a round often depends on the result of previous rounds. This is especially true at the end of a game. For example in a situation where an agent can win the whole game by taking one point, that would be the smart thing to do, although its strategy might say that it should force a tie in order to keep the last stone for the next round.

6.3.3 Results

Figure 6.3 shows how both sets of weights were adjusted during 200 rounds of training. Table 6.1 shows the results of five matches against SkippyAgent from experiment 1. See Table A.6 in the Appendix for more details of these matches.

Opponent	**Victories**	**Total Points**	**Avg. pr. Round**
SkippyAgent	4	49-31	0.98

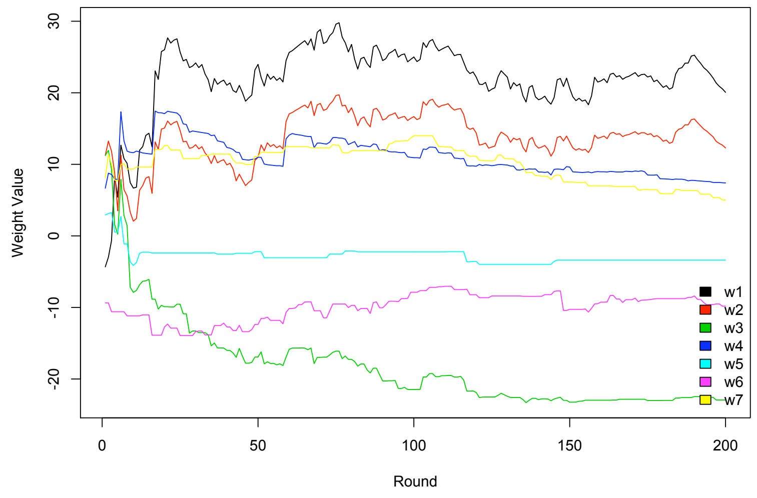
Table 6.3: Results from 5-Matches tournament.

6.3.4 Discussion

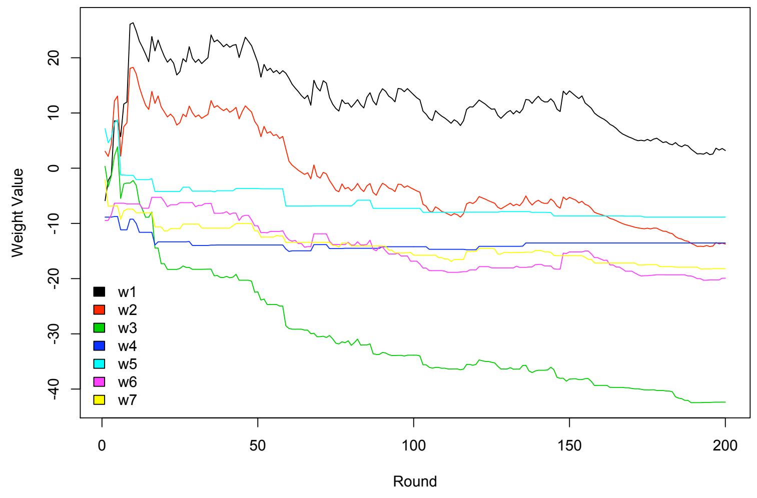
For both sets of weight we can see that there is quite a bit of movement at the beginning of the training. However it only takes about 25 rounds before the weights seem to have converged. There are some movement after this but that can be attributed to noise from the simulator which adds a bit of uncertainty to each throw, or to the fact that the agent is unable to estimate all possible actions.

For the weights without the hammer, the convergence seems quite good and the weights stay about the same for the rest of the training, however in the case of the hammer weights, some of the weights seem to decrease continually after the initial convergence. This led us to suspect that the training after round 25 might be over fitting the weights. We tested this by running several matches against SkippyAgent with both sets of weights trained for 200 rounds. There did not seem to be much improvement for SkippyAgentHammer, and both agents won an equal amount of times. We then tried to roll back the hammer weights to what they were after 25 rounds of training and the results improved quite a bit. We conclude that the hammer weights were in fact over fitted and the results seen in Table 6.1 are obtained using hammer weights that are trained for 25 rounds.

We can se that both sets are similar when it comes to the weights: *score* (w_1), *the agent's rocks in house* (w_2), and *the opponents rocks in house* (w_3). For both sets it is the case that the opponents rocks in house is very negative, while the agents rocks in house is very positive, and score is even more positive. The key difference between the two sets is that while the non-hammer weights value *the agents guarded rocks* (w_4) and *center guards* (w_7) as positive, the hammer weights value these as negative together with *the opponents guarded rocks* (w_5) and *corner guards* (w_6).



(a) Weights without hammer



(b) Weights with hammer

Figure 6.3: Weights for SkippyAgent, Figure 6.3(a) shows the weights that the agent uses when it does not have the hammer. Figure 6.3(b) shows the weights that are used when the agent is playing with the hammer.

When it comes to the 5-match tournament against SkippyAgent we se that SkippyAgentHammer won 4 out of 5 matches. It scored 37% more points than its opponents for an average of 0.98 points per round. By studying Table A.6 some interesting patterns emerge. We see that both agents seem to take points every other round. This is of course because the take turns having the last stone, and we get a clear example of the advantage this gives. However there is one important difference: SkippyAgentHammer is able to take several points when having the last stone while SkippyAgent often only manages to take one point. The strategies we discussed in Section 1.3.3 say that it is better to force a tie than take one point when having the last stone. SkippyAgent does this a total of 14 times while SkippyAgentHammer only does it 3 times. Another important measure of success in curling is the ability to steel points, which is to take points when *not* having the last stone. We see that both agents manage to do this three times each, so no difference there.

6.3.5 Conclusion

The experiment is considered a success. SkippyAgentHammer shows improvement over SkippyAgent. The weights have converged after only a small amount of training, and the values of the weights seem sensible. The observed behavior of the agent largely coincides with what is considered good strategy in curling. Still there are some aspects that this agent is unable to capture. It is very good at placing individual rocks but it lacks the ability to see broader strategies over a whole round or even from one round to the next. We hope that the Q-learning agent described in the next experiment in Section 6.4 will be better at this.

6.4 Experiment 3 - Q-learning

The previous experiment (Section 6.3) was a good improvement to the first experiment described in Section 6.1. In this experiment we will expand on the previous agent by using Q-Learning in an effort to enable the agent to treat a curling round as a sequence of episodes (Section 4.1.1). This should make it possible for the agent to learn better strategies.

The implementation of the agent (*RLSkippyAgent*) used is described in Section 5.3.2. The most significant difference from the previous experiment is the use of Q-learning and the new state space (Section 4.4) that should enable the agent to plan curling strategies within whole rounds, as opposed to just individual stones.

6.4.1 Setup

Since *RLSkippyAgent* is built upon the SkippyAgentHammer, it will use the feature weights trained in the previous experiment (Section 6.3). However, when training the *RLSkippyAgent* we will disable further training of the feature weights by setting the learning rate (η) to zero.

The only training in this experiment will be within the scope of the Q-learning algorithm, that is the training and learning of $Q(s, a)$ values. The Q-learning used by the agent was initialized with the following setup shown in Table 6.4. The experiment will be kick started with a non-empty Q(s,a) table that was created when implementing the *RLSkippyAgent*.

Discount Factor, γ	0.90
Maximum Learning Rate, ρ	0.80
$Q(s, a)$ table size, $\lvert Q(s, a)\rvert$	567084
Number of non-zero $Q(s, a)$-values	56338

Table 6.4: RLSkippyAgent setup.

Training *RLSkippyAgent* will be done by alternating between playing against the following agents.

- *AKA*
- *Scott*
- *SkippyAgent*
- *SkippyAgentHammer*

AKA and *Scott* will be used early in the training where speed is favored, and where *RLSkippyAgnet* is largely just exploring the state space. As the training progresses *SkippyAgent* will be used to provide a more challenging opponent. *SkippyAgentHammer* will be used to continuously test the progress of the experiment. *SkippyAgentHammer* is so far the best agent, and hence provides the best reference point to measure the curling capabilities of *RLSkippyAgent* as the learning progress.

6.4.2 Weaknesses/Expectations

The large state space used in the Q-Learning present a high risk for the efficiency of the learning. A worst case scenario for this experiment will result in an incomplete $Q(s, a)$ table that contains a high ratio of zero entries. This would make it impossible for the agent to select appropriate actions for the states it encounters during play.

If the training results in a $Q(s,a)$ table with a sufficient number of nonzero entries, we should get a result where the agent is able to select "optimal" actions to any of the occurring states. However, there might remain some points of weakness.
One issue relates to the action groups used by the Q-learning (see Figure 4.5). Using only three action groups may not provide the Q-learning with enough granularities to effectively select appropriate behavior to the states played.
Another potential weakness is related to how the *RLSkippyAgent* uses feature weights after selecting one of the action groups. The *guard* action group does not share a similar evaluation function used for the *in house* weight (Section 4.2.1). This makes it difficult to use the guard group since the current feature representation fails to capture functionally different guard situation.

We believe that a successful training will result in a curling agent requiring fewer resources (faster response) and an agent capable of applying curling strategies throughout curling rounds. Knowing where (start, middle, end) in a curling round the *RLSkippyAgent* is should provide the agent with a key advantage that enables the agent to win with all the opponents from the previous experiments.

6.4.3 Results

Figure 6.4 show the distribution between the total number of $Q(s,a)$ values and the number of $Q(s,a)$ values equaling zero throughout the training of *RLSkippyAgent.* Figure 6.5 show the ratio between matches won and the total number of match played for various opponents. (3500 match played against AKA was omitted from the figured since the ratio remained more or less constant after 1500 matches played.) Figure B.1 includes the omitted results.

6.4.4 Discussion

Training *RLSkippyAgent* proved to be very difficult. Figure 6.4 shows the distribution between state-action pairs created as the agent visits states, and how many of these state-action pairs that had zero values after playing over 7500 matches. Comparing the two lines in the graph we can see that the growth of zero valued state-action pair is slower compared to the growth of new state-action pairs. This tells us that the training is nowhere near completing. At this point in the training a desired behavior would be a dissipating growth of new state-action pairs and a

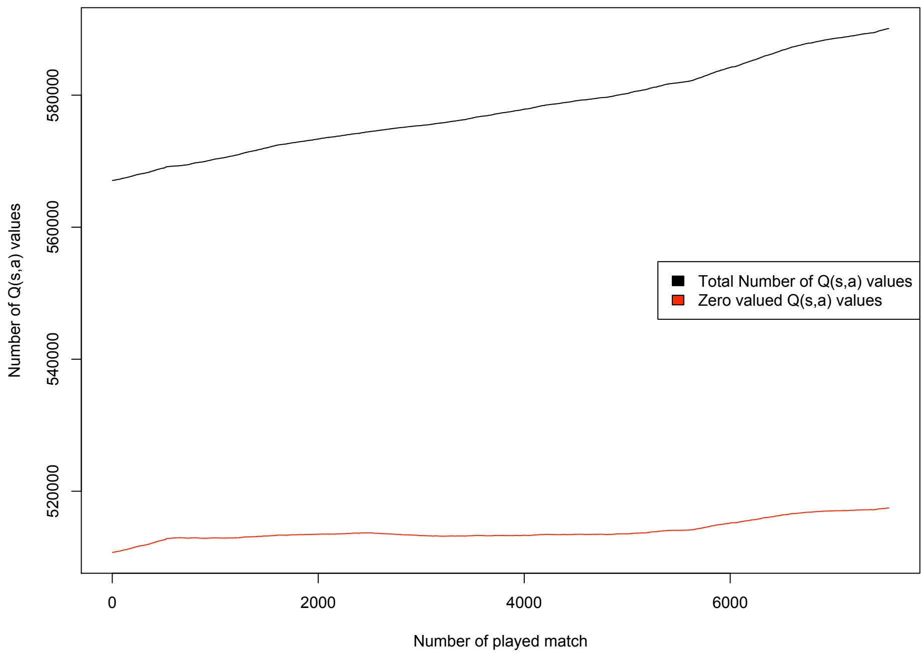


Figure 6.4: The $Q(s, a)$ size as training progressed

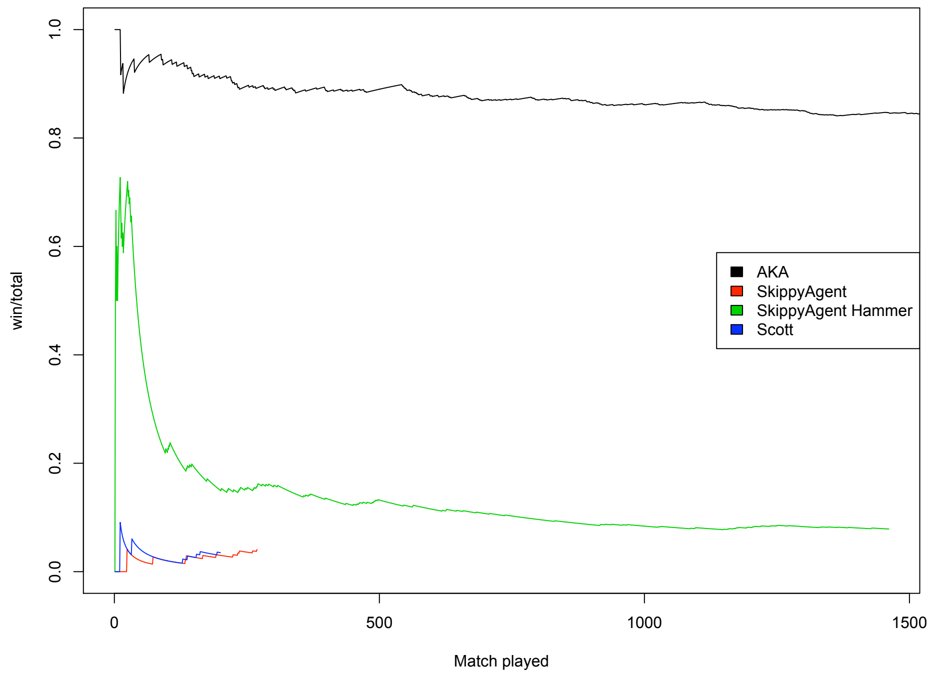


Figure 6.5: The ratio of wins against various agents.

decrease in zero valued state-action pairs.

The match ratio showed in Figure 6.5 further confirms the problem we had with training *RLSkippyAgent* sufficiently. We can see that *RLSkippyAgent* shows poor results when playing against opponents other than *AKA*. Against *SkippyAgentHammer* the winning rate is 20%, this confirms our problem with learning a sufficient number of state-action pairs which is needed to outperform the opponents.

The large state-space used created a much larger problem than expected. Failing to train *RLSkippyAgent* sufficiently within the time allocated to the experiment makes it very difficult to evaluate our idea of using Q-learning. However, we also suspect that the nondeterministic environment contributed to the learning problem.

6.4.5 Conclusion

The experiment was a failure. The resources needed to train *RLSkippyAgent* far exceeded our expectations. For future work on this problem there are steps that can be taken in order to get better results. One approach is to simplify the description of states in a game of curling. This would greatly reduce the size of the state space, and in turn the time needed for training. The state description cannot be too small as this would reduce the expressiveness of the model to a point where it fails to capture the strategies we want the agent to learn. Another approach is to spend more resources, by optimizing the code and utilizing more computing power. The key is probably to find the right balance between a simpler model and more resources.

There might be other techniques within the field of Artificial Intelligence that can be interesting to try on this problem. But after working on this project we still feel that Q-learning looks like a promising approach. Unfortunately we failed to prove that point this time around.

Chapter 7

Discussion

In this chapter we will discuss the results of this project as a whole. The chapter is divided into two sections, one concerning the curling simulator, and one about the agents.

7.1 Curling Simulator

At the start of this project we sat ourselves the goal of creating a computer program capable of adequately simulating a game of curling. We feel that we have largely achieved this goal. With this curling simulator one can play a game, either against another human player or against an automated computer agent. It is also possible for to computer agents to play against each other. There are however some potential for improvement which we will discuss next.

As mentioned in Section 5.1.2 the simulator is using the JCurl framework. Having this was a lot of help during implementation, but there are some issues. As it is now, the core application logic is to tightly bound to the graphical user interface. This makes testing more difficult and running large simulations takes more time since the graphics cannot be turned off.

When it comes to the interfaces against the curling playing agents, we are quite satisfied. The interfaces are rich and well defined, and they allow for a wide range of agents to be implemented.

The core application of the simulator could have been structured a bit better. This also applies to the separation of core logic and user interface as mentioned earlier. For our needs in this project the simulator has worked satisfactorily, but it might require some effort to get acquainted with for future potential users.

7.2 Curling Playing Agents

We implemented several agents using the interfaces provided by the curling simulator. Some were very simple consisting of only a few lines of code, while others grew to be quite sophisticated utilizing advanced techniques from the field of Artificial Intelligence. At any rate, all these agents turned out to be great tools for weeding out bugs in the simulator.

The performance of the agents was both very satisfactory and disappointing. We found that the agents that utilize a Linear Target Function as their basis for selecting actions, performed quite well. See Sections 6.1 and 6.3 for details about the experiments concerning these agents. However, even though these agents are adept at placing individual rocks, we soon realized that they were unable to lay good strategies over the course of a whole round, or even just a few moves ahead.

Our effort to remedy this problem was to expand on these agents using Q-learning. The idea being that this new agent would use Q-learning to capture the wider strategies of curling, such as when to set guards, when to take out rocks, and so on. Then it would use the Linear Target Function to find the best way to execute these moves. Unfortunately this did not work as planed. The time required to train the Q-learning part of the agent far exceeded our expectations. The training of the agent should converge if it is allowed to run indefinitely, but that is impractical with the current implementation. For future work on this problem one could try a more narrow representation of the state in a game of curling. The key will be to find a balance where the representation is expressive enough to capture the strategies of curling, while at the same time simple enough that the learning will converge within a reasonable amount of time.

Chapter 8

Conclusion

At the start of this project we sat ourselves the following goals

- Design a curling simulator (using an adequate physical model) that curling-playing agents can interface with.
- Explore knowledge representation in the curling domain.
- Build a curling-playing agent and test it in the simulator.

The first goal we feel that we have accomplished. The simulator works very well for the purposes of this project, but there is always room for improvement. The second goal is a bit tricky. We chose to use a linear state vector to describe states in a game of curling. Given the continuous nature of curling there is virtually an endless amount of information needed to describe all situations. Using a linear state vector we were able to abstract this information into something a lot more manageable. Our third goal concerns creating curling playing agents. Here we had mixed success. Our first attempt was to use a Linear Target Function as a basis for selecting good moves. This agent worked quite nice for placing individual rocks but lacked the ability for advanced strategy. In an effort to improve this agent, we expanded it using Q-learning that we hoped would capture some of these advanced strategies. Unfortunately this agent did not work as intended. Nonetheless we have created agents that are capable of playing curling and they do provide a fair challenge when played against by humans.

8.1 Further Work

For potential future work on the problems discussed in this project we see several things that can be improved upon. The simulator is functional as it is now but it can be made more efficient. This especially concerns the separation of the core application logic and the graphical user interface. The simulator can also be expanded to use animation thereby making it easier to understand what is happening on the screen as the game progresses.

For the agents there is also room for improvement. The agent using Q-learning is not viable as it is now but we believe it could be made to work using a different state representation that requires less time to learn. It could also be interesting to try other techniques besides Q-learning, for example Case Based Reasoning.

Appendix A

Tournament Results

SkippyAgent - AKA

Match 1	0	1	4	1	5	4	2	2	0	5	24
	1	0	0	0	0	0	0	0	1	0	2
Match 2	4	3	3	4	4	6	0	3	0	4	31
	0	0	0	0	0	0	1	0	1	0	2
Match 3	1	4	2	6	0	2	6	5	0	2	28
	0	0	0	0	1	0	0	0	1	0	2
Match 4	4	1	2	1	4	6	0	4	5	5	32
	0	0	0	0	0	0	2	0	0	0	2
Match 5	2	6	3	1	1	2	7	2	1	1	26
	0	0	0	0	0	0	0	0	0	0	0

Table A.1: Result from five matches between *SkippyAgent* and *AKA*.

SkippyAgent - Scott

Match 1	1	0	0	1	0	0	0	1	0	1	4
	0	0	1	0	0	0	1	0	0	0	2
Match 2	1	0	1	0	0	0	0	0	0	1	4
	0	1	0	1	0	0	0	0	0	0	2
Match 3	1	0	0	0	0	0	0	0	0	2 (1)	4
	0	0	0	0	0	0	0	0	3	0 (0)	3
Match 4	1	0	0	1	0	1	0	1	0	1	5
	0	0	1	0	1	0	1	0	0	0	3
Match 5	1	0	1	0	0	1	0	0	0	2 (1)	6
	0	2	0	0	1	0	0	0	1	0 (0)	5

Table A.2: Result from five matches between *SkippyAgent* and *Scott*.

SkippyAgent Mod - AKA

Match 1	2	0	2	7	1	3	1	0	4	0	20
	0	1	0	0	0	0	0	1	0	1	3
Match 2	2	1	2	2	0	5	3	3	3	5	26
	0	0	0	0	1	0	0	0	0	0	1
Match 3	0	3	1	4	5	5	2	0	3	5	28
	1	0	0	0	0	0	0	2	0	0	3
Match 4	8	1	3	1	0	1	2	5	3	1	25
	0	0	0	0	1	0	0	0	0	0	0
Match 5	6	2	1	0	4	1	2	5	2	2	25
	0	0	0	1	0	0	0	0	0	0	1

Table A.3: Result from five matches between *SkippyAgent Mod* and *AKA*.

SkippyAgent Mod - Scott

Match 1	0	1	0	0	0	0	0	1	0	1	3
	0	0	1	0	0	0	0	0	1	0	2
Match 2	0	1	0	0	1	0	0	1	0	1 (1)	6
	1	0	0	1	0	0	1	0	1	0 (0)	5
Match 3	0	0	0	0	0	1	0	1	0	1 (1)	4
	0	0	1	0	0	0	1	0	1	0 (0)	3
Match 4	1	0	1	0	0	1	0	0	1	1	5
	0	1	0	0	1	0	0	1	0	0	3
Match 5	1	0	0	0	1	0	1	0	0	1	4
	0	1	0	0	0	0	0	0	0	0	1

Table A.4: Result from five matches between *SkippyAgent Mod* and *Scott*.

SkippyAgent Mod - SkippyAgent

Match 1	0	1	0	1	0	1	0	2	0	1	6
	1	0	3	0	1	0	2	0	3	0	10
Match 2	2	0	1	0	0	0	2	0	2	1	8
	0	1	0	1	1	1	0	1	0	0	5
Match 3	0	0	1	0	2	0	1	0	0	2	6
	2	1	0	1	0	3	0	0	2	0	9
Match 4	1	0	1	0	1	0	1	0	2	0	6
	0	1	0	2	0	2	0	1	0	1	7
Match 5	0	1	0	1	0	2	0	0	1	0	5
	2	0	1	0	3	0	4	1	0	1	12

Table A.5: Result from five matches between *SkippyAgent* Mod and *SkippyAgent*.

SkippyAgentHammer - SkippyAgent

Match 1	4	0	2	0	2	0	3	0	1	0	12
	0	1	0	1	0	2	0	1	0	1	6
Match 2	0	2	0	3	1	0	4	0	1	0	11
	1	0	1	0	0	1	0	1	0	1	5
Match 3	0	2	0	0	2	0	0	2	0	1	7
	3	0	1	0	0	1	1	0	2	0	8
Match 4	3	0	2	1	0	0	0	3	1	0	10
	0	1	0	0	2	0	1	0	0	1	5
Match 5	0	3	0	0	3	0	0	2	0	1	9
	1	0	2	1	0	0	1	0	2	0	7

Table A.6: Result from five matches between *SkippyAgentHammer* and *SkippyA-gent*.

Appendix B

Q-Learning Results

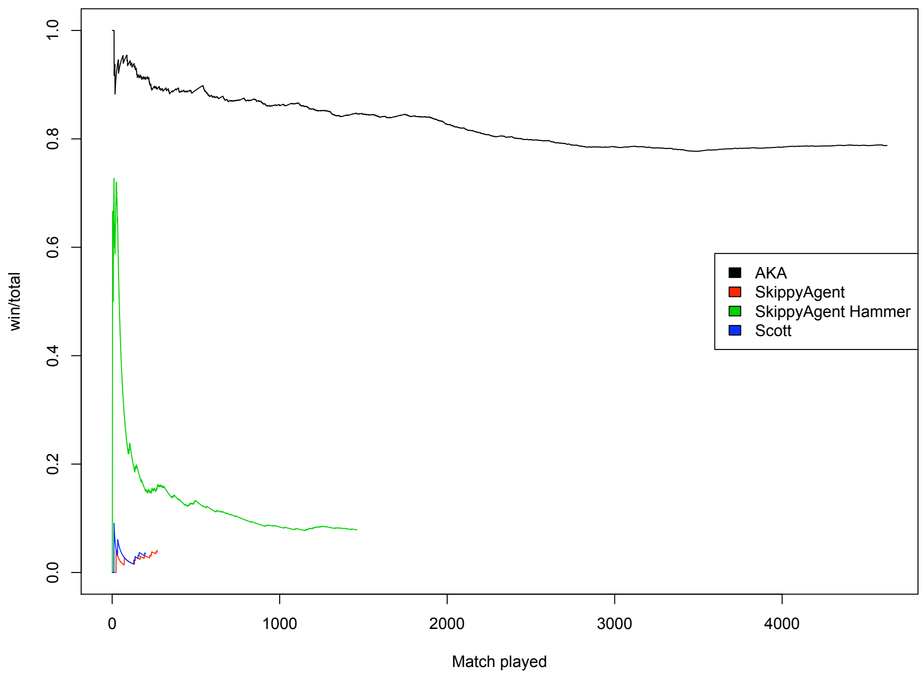


Figure B.1: The ration of wins against various agents including the omitted data not showed in Figure 6.5

Bibliography

[1] Albus, J. S. A new approach to manipulator control: the cerebellar model articulation controller (CMAC). *Journal of Dynamic Systems 97* (Sep 1975).

[2] Auer, P., Cesa-Bianchi, N., and Fischer, P. Finite-time analysis of the multi-armed bandit problem. *Machine Learning 47* (Jan 2002), 235 – 256.

[3] Auer, P., Jaksch, T., and Ortner, R. Near-optimal regret bounds for reinforcement learning. *Advances in Neural Information Processing Systems 21* (Jan 2009), 89–96.

[4] Clarke, S., and Norman, J. To run or not?: Some dynamic programming models in cricket. *Journal of the Operational Research Society 50* (Jan 1999), 536–545.

[5] D'Esopo, D., and Lefkowitz, B. The distribution of runs in the game of baseball. in: Optimal strategies in sports. *North-Holland* (Jan 1977).

[6] Federation, W. C. http://www.worldcurling.org/, 2009.

[7] Gelly, S., Wang, Y., Munos, R., and Teytaud, O. Modification of UCT with patterns in Monte-Carlo Go. *hal.inria.fr* (November 2006).

[8] Gross, H., Stephan, V., and Krabbes, M. A neural field approach to topological reinforcement learning in continuous action spaces. *Proc. 1998 IEEE World Congress on Computational Intelligence, WCCI'98 and International Joint Conference on Neural Networks, IJCNN'98 3* (May 1998), 1992–1997.

[9] Kocsis, L., and Szepesvári, C. Bandit based monte-carlo planning. *Lecture Notes in Computer Science* (Jan 2006), 282–293.

[10] Kohonen, T. *Self-organization and associative memory*, 3rd ed. Springer, Jan 1989.

[11] Kostuk, K., Willoughby, K., and Saedt, A. Modelling curling as a markov process. *European Journal of Operational Research 133*, 3 (September 2001), 557–565.

[12] Maes, S., and Tuyls, K. Reinforcement learning in large state spaces: Simulated robotic soccer as a testbed. *Lecture Notes in Computer Science 2752* (Feb 2002), 319–326.

[13] Mitchell, T. M. *Machine Learning*. McGraw-Hill Science/Engineering/-Mat, March 1997.

[14] Percy, D. Stochastic snooker. *The Statistician* (Jan 1994).

[15] Reenskaug, T. Models - views - controllers. http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html, 2009.

[16] Robbins, H. A sequential decision problem with a finite memory. *Proceedings of the National Academy of Sciences 42*, 12 (December 1956), 920–923.

[17] Robocode. http://robocode.sourceforge.net/, 2009.

[18] Russell, S. J., and Norvig, P. *Artificial Intelligence: A Modern Approach*, 1st ed. Prentice Hall, January 1995.

[19] Santamaria, J. C., Sutton, R. S., and Ram, A. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior 6*, 2 (Jan 1997), 163–217.

[20] Schaeffer, J., Burch, N., Bjornsson, Y., Kishimoto, A., Muller, M., Lake, R., Lu, P., and Sutphen, S. Checkers Is Solved. *Science 317*, 5844 (2007), 1518–1522.

[21] Sutton, R. S., and Barto, A. G. *Reinforcement Learning : An Introduction*, 1st ed. Adaptive computation and machine learning. MIT Press, 1998.

[22] Touzet, C. Neural reinforcement learning for behaviour synthesis. *Robotics and Autonomous Systems 22*, 3-4 (Jan 1997), 251–281.

[23] Uchibe, E., Asada, M., and Hosoda, K. Behavior coordination for a mobile robot using modularreinforcement learning. *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems* (Jan 1996), 1329–1336.

[24] XNA. http://msdn.microsoft.com/en-us/xna/, 2009.