MASTER THESIS

Combinatory Tests of AI Techniques

for Common Tasks in Computer Games

Pelle Coltau Jens Juul Jacobsen Brian Jensen

The Faculty of Engineering and Science

AALBORG UNIVERSITY

Department of Computer Science

TITLE:
Combinatory Tests of AI Techniques for Common Tasks in Computer Games

THEME:
Master Thesis E06 - Machine Intelligence Group

PROJECT PERIOD:
February - June, 2006

TERM:
DAT6

PROJECT GROUP:
d642a

GROUP MEMBERS:
Coltau, Pelle
Jacobsen, Jens Juul
Jensen, Brian

ADVISOR:
Bangsø, Olav

NUMBER OF COPIES: 7

NUMBER OF PAGES: 108

APPENDIX PAGES: 30

TOTAL NUMBER OF PAGES: 138

ABSTRACT:

In this project we empirically test various learning AI techniques for their appliance in computer games. This is done through a self-developed modular framework for solving the board game Risk. The framework covers tasks commonly needed in computer games such as analysis of the AIs environment, movement planning, etc.
The AI techniques tested in this project are scripting, decision trees, neural networks, Bayesian networks, and naive Bayes classifiers. We have implemented a scripted AI for Risk which is used to construct training data for the learning techniques.
We discuss different aspects of training the various techniques and the problems we encountered while doing so.
We present a way to test the importance of each module in the framework and discuss different ways of planning tests with a high number of AI participants. We come up with a testing scheme that will eventually present the best AI for playing Risk without having to play all possible AI compositions against each other.
The result of the tests is an AI composed of different AI techniques that each is the best at its task.
The tests show that learning AI techniques can learn how to solve different tasks involved in computer games. In some cases they even outperform their teachers.
Finally we reflect upon the test results, what most the likely appliance of each AI technique should be in computer games.

Aalborg, June 6, 2006

COLTAU, PELLE

JACOBSEN, JENS JUUL

JENSEN, BRIAN

Foreword

This project was developed by students on the 10th semester in the Machine Intelligence Group at the Department of Computer Science, Aalborg University, Denmark.

This master thesis documents the continued work of "Artificial Intelligence in Computer Games" [CJJ06]. This project empirically tests the ability of learning algorithms to learn common tasks in computer games. The performance is tested against a non-learning implementation of each task.

We would like to thank Frank Jensen from Hugin Expert in support concerning the Hugin API. We would also like to thank Atle C. Pedersen for kindly lending computational power when testing.

A summary of this report is found in Appendix H.

A digital version of the report can be found through the link below.
URL: http://www.cs.aau.dk/library/

This report also contains an enclosed CD. The content of this CD can be seen in Appendix G. If the CD is not included or the report is a digital version, a web link to the content of the CD can be seen in the same Appendix.

Contents

1 Introduction **1**
- 1.1 Project Goal . . . 1
- 1.2 Report Overview . . . 1
- 1.3 The Board Game Risk . . . 2
  - 1.3.1 A Few Definitions . . . 2
- 1.4 JRisk . . . 3

2 Previous Work **5**
- 2.1 The Framework . . . 5
  - 2.1.1 The Information Givers . . . 6
  - 2.1.2 The Master Prioritizer . . . 7
  - 2.1.3 The Round Planner . . . 7
- 2.2 Applied AI Techniques . . . 11
  - 2.2.1 Scripting . . . 11
  - 2.2.2 Decision Trees . . . 13
  - 2.2.3 Neural Networks . . . 15
  - 2.2.4 Bayesian Networks . . . 20
  - 2.2.5 Naive Bayes Classifiers . . . 23
- 2.3 Additions to the Framework . . . 24
  - 2.3.1 New Modules . . . 24
- 2.4 Framework Behavior . . . 26
  - 2.4.1 Class Structure Overview . . . 29
- 2.5 Assumptions in Previous Work . . . 33
- 2.6 Combining AI Techniques with Modules . . . 35
- 2.7 Bayesian Network Redesign . . . 37
  - 2.7.1 IG: Close to Continent . . . 37
  - 2.7.2 IG: Close to Winning . . . 38

3 Training **41**
- 3.1 The Scripted AI . . . 41
  - 3.1.1 What Should Be Stored? . . . 42
- 3.2 The Converter . . . 43
  - 3.2.1 Making Continuous Values Discrete . . . 43
  - 3.2.2 Constructing the Training Data . . . 44
  - 3.2.3 Two Approaches for Learning from Losers . . . 45

3.2.4 Choosing an Approach . . . 46
3.2.5 Training with Winners Only . . . 47
3.2.6 Class Diagram . . . 48
3.3 The Trainer . . . 49
3.3.1 Training Tools . . . 50
3.3.2 Training Data Set Size . . . 51
3.3.3 Training Iterations . . . 52

4 Testing **55**
4.1 Finding the Best Combination of Modules . . . 55
4.1.1 Limiting The Number Of Games To Play . . . 55
4.1.2 Limiting the Number of AIs . . . 57
4.2 Most Important Modules . . . 59
4.3 Time Used In Each Module . . . 60
4.4 Model Size . . . 61
4.5 User Experience . . . 61
4.6 Miscellaneous Recordings . . . 62
4.7 Number of Games Played . . . 63
4.8 Summary of the Tests . . . 63
4.8.1 Building the Best AI From Test Results . . . 63
4.9 Performing the Tests . . . 64
4.9.1 Scheduling Module Performance Test . . . 64

5 Implementation **67**
5.1 Script Implementation . . . 67
5.2 Training Data Converter . . . 67
5.3 Training The AI Models . . . 68
5.3.1 Limitations in Training . . . 68
5.3.2 Techniques Implemented on Modules . . . 68
5.3.3 Dividing Into Ranges . . . 70
5.3.4 Training Bayesian Networks . . . 70
5.3.5 Training Decision Trees and Naive Bayes Classifiers . . . 71
5.4 Training Neural Networks . . . 72
5.5 Implementing the Tests . . . 73

6 Results **75**
6.1 Training Amount . . . 75
6.2 Module Importance . . . 75
6.3 Module Performance . . . 77
6.4 Test Results . . . 79
6.4.1 Initial Army Placement . . . 79
6.4.2 Estimate opponents' mission . . . 80
6.4.3 How Close an Opponent is to Winning . . . 80
6.4.4 The Opponents' Next Moves . . . 81
6.4.5 How Close an Opponent is to Owning a Continent . . . 81
6.4.6 MP Goal Weighting . . . 81
6.4.7 Prioritize Attack Plan . . . 82
6.4.8 Score Attack Plan . . . 82
6.4.9 Calculate Defense Cost . . . 83
6.4.10 Score Merged Plan . . . 83

6.4.11 Prioritize Territory Needing Defense 83
6.5 Building the Best AI 83
6.5.1 The Best AI 83
6.5.2 The Challenger AIs 84

7 Reflection 89
7.1 Generalization of Risk 89
7.1.1 Risk vs. Counter-Strike 90
7.1.2 Risk vs. The Sims 90
7.1.3 Risk vs. Command & Conquer 91
7.1.4 The Generality of Risk 92
7.2 Generalization of the Framework 92
7.2.1 Our Framework in Counter-Strike 92
7.2.2 Our Framework in The Sims 93
7.2.3 Our Framework in Command & Conquer 94
7.2.4 Summary on the Generality of Our Framework 94
7.3 Generality of the Applied Techniques 95
7.3.1 Generality of Neural Networks 95
7.3.2 Generality of Decision Trees 96
7.3.3 Generality of Naive Bayes Classifiers 97
7.3.4 Generality of Bayesian Networks 97
7.3.5 Generality of Scripts 98

8 Conclusion 101
8.1 Future Work 104

Bibliography 107

A Changes in the Scripted AI 109
A.1 IG: How Close The Opponent Is To Owning a Continent 109
A.2 IG: Ownership 109
A.3 IG: How Close The Opponent Is to Winning 109
A.4 Master Prioritizer 110
A.5 RP: Calculate Attack Plan Cost 112
A.6 RP: Make Attack Plan 112
A.7 RP: Place Armies 112
A.8 RP: Prioritize Attack Plan 112
A.9 RP: Prioritize Territory Needing Defense 114
A.10 RP: Transfer Armies 114

B Rules of Risk 115
B.1 Cashing in RISK Cards Phase 117
B.2 Defend Phase 117
B.3 Attack Phase 118
B.4 Fortify Phase 118
B.5 RISK Activity Diagram 118

C The Goals from the Master Prioritizier 121

D Trainer Data **125**
D.1 Neural Network . . . 125
D.2 Decision Trees and Naive Bayes . . . 125
D.3 Bayesian Networks . . . 127

E Testing Algorithm **129**
E.1 Algorithm for Generating a Test Schedule . . . 129

F Test Suite **131**

G Enclosed CD **133**
G.1 JRisk . . . 133
G.2 Training Data Converter . . . 134
G.3 Trainer . . . 135

H Summary **137**

Chapter 1

Introduction

In commercial games released today the majority of AI opponents are designed and implemented using scripts. This leaves a lot of design choices to the designer, but it also means that as the size and complexity of game increases, so does the chance that the AI designer overlooks some detail in the aspects of the game. An obvious solution for this would be using learning AIs to handle different tasks within a game, and thereby relieve the AI designer, since the learning AI would learn the details by itself.

Left is now to choose which learning AI techniques to choose for the various tasks commonly used in computer games. There are a lot of possible techniques to choose from, but which ones suit which tasks the best?

1.1 Project Goal

The goal of this project is to examine how well different AI techniques perform compared to each other at various tasks in computer games. Some of the techniques are trained learning algorithms and others are not.
The AI performance is measured in terms of time spent solving its task, and its success rate. These terms have been chosen since they cover both the AI's system requirements as well as its usability in solving a given task. The test is done through the use of an AI framework for which the full design is covered in [CJJ06]. The design is modular and covers the different tasks involved in both estimating uncertainties and planning.

1.2 Report Overview

A brief description of this framework will be given in the following section along with descriptions of some interesting and necessary changes that have been made during its implementation. Furthermore, we will make a brief repetition of some important points made in the previous report which also applies to this project.

Followed by this, there will be a discussion of how the training of the learning algorithms are performed, and how the non-learning (*scripted*) will generate training data for the learning algorithms.

When the tests have been performed, we will use gathered results to find the best AI based on its performance.

We will argue that the tasks in our framework are general enough to apply to most computer games and therefore the best performing techniques will prove better in other games too.

1.3 The Board Game Risk

Before going into a description of the work done in [CJJ06], an introduction to the board game Risk is needed. This game was in [CJJ06] selected as the environment in which our AI should act. Risk is a turn-based strategy game in with three to six players. The board consists of 42 territories. Each territory belongs to one of 6 continents. Each territory is owned by a player and is occupied by a number of armies belonging to that player. The object of the game is to fulfill your mission. A mission can for instance be "Conquer Asia and South America". A turn in the game consist of four phases:

1. **Cashing in RISK cards:** Risk cards are earned through attacking territories and can be cashed in to gain extra armies (reinforcements).
2. **Defend:** This phase is for placing the reinforcements on the board.
3. **Attack:** A player can attack with a number of armies from a territory occupied by himself to a neighboring territory occupied by an opponent.
4. **Fortify:** The last phase is where a number of armies are transfered between two neighboring territories both occupied by the player.

When a player has fortified his turn is done and it is the next player's turn.

There exist many different rules for the game of RISK. Different rules have both been published by the developers of the original game, but also many "house-rules" have over the years become widely used.

A detailed description of the rules of RISK used in this project can be found in Appendix B.

1.3.1 A Few Definitions

Throughout this report we will use a few terms that needs to be defined here. When we use the word *turn*, it means what happens when a player holds the dice. A players turn is what goes on within the four phases mentioned earlier. A *round* goes from when a player ends his turn until he gets the turn again. This means that the statement *"an opponents wins 0 rounds from now"* means that the opponent will win before the player gets his next turn.

When a player *occupies* a territory it means that the player owns it, and will be the one defending it from attacks. When a player attacks a territory he tries to *conquer* it.

When refering to “an occupied territory”, it generally means that the territory is occupied by the AI. And “an opponent territory” is the opposite.

1.4 JRisk

There are quite a few different implementations of the RISK game available on the Internet, both commercial versions and some free. We found a game called JRisk on the Internet that seemed suitable. It is written in Java [jav] and implements a design that will let us, somewhat easily, replace the AI part. The game is open source and all source code is available. JRisk can be found at [jri]. There are a few differences between the rules in JRisk and the rules in this project. Our rules has been implemented in JRisk without any problems. In JRisk there was implemented two different AIs called the “Easy AI” and the “Hard AI”. Their behavior is undocumented and it has been decided that it is unnecessary to analyze it any deeper.

The JRisk used in this project is version 1.0.7.6.

Chapter 2

Previous Work

This chapter will contain a brief description of the AI framework that was presented in [CJJ06]. Each of the AI techniques used in this and the previous project will also briefly be presented. Some changes had to be made to the framework in order to accommodate improved behavior and create a better interface for the different AI techniques. These changes are described in a bit more detail. Some interesting points made during the previous project will be repeated and elaborated upon. Lastly, each technique will be combined with the framework.

2.1 The Framework

The AI framework is a highly modular framework which is designed for the game RISK. To obtain the specific modules, we have analyzed RISK and have designed a layered AI architecture. Each module has some input and produce an output. The reason for this, is that this makes it easier to implement the behavior of the modules using different AI techniques. The input where found by analyzing what information should be available to the given module, for it to produce its output, but without losing the modularity. Having a limited number of inputs was also a goal, to avoid the learning AI techniques having to deal with too much information.

Compared to the original design, some of the input and output have changed. The reason for the changes is partly to accommodate new behavior, in order to make the scripted AI better, and partly to make the modules more compatible with the different AI techniques. The original design included a full design of the scripted implementation of each module. As the input and output has changes, the script design has also changed. These changes can be seen in Appendix A. This section will only contain a description of the modules in the framework — their behavior, input and output.

The framework is a layered model which consists of three layers: The set of “Information Givers”, the “Master Prioritizer”, and the “Round Planner”. The

layered model can be seen in Figure 2.1.

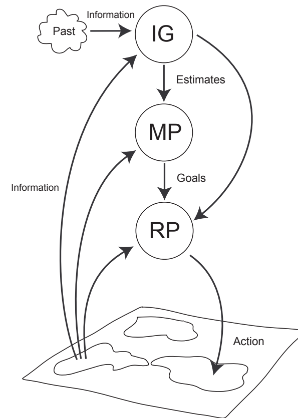

Figure 2.1: *A modular design of the AI divided into IG, MP and RP.*

2.1.1 The Information Givers

The top layer is the set of "Information Givers" (IGs). Their task is to gather information concerning the board and other players. They share this information with the lower levels in the architecture and, if needed, amongst each other. The following will describe the IG modules in the framework, their behvavior, input and output.

IG: Estimating Opponents' Missions (IG Mission)

This IG is used to estimate what mission the opponents have. To calculate this information, the IG has access to a list of which actions each player has performed throughout the game, called *the past*. The estimate has a attached certainty, which reflects how certain the IG is on its estimate. The more past actions that points towards a specific mission, the higher the certainty of the mission is.

IG: How Close an Opponent is to Winning (IG Winning)

This IG is used to estimate how close each player is to fulfilling his mission within a fixed number of rounds. This fixed number was chosen to be 5, since beyond 5 rounds from now, the precision gets too small. This information is calculated through the use of the estimate of each opponents' missions and the certainty of these, the current board state, and knowledge of how many RISK cards the opponents have available.

A change from the original design is, that this IG also uses information on whether it is run in the begining of a turn or not. If it is not the beginning of a turn, the AI has already received his reinforcements and this naturally change the behavior of the module.

The output from this IG is how close each opponent is to winning within a number of rounds.

IG: How Close an Opponent is to Conquering a Continent (IG Continent)

This IG is used to estimate how close each player is to conquering an entire continent within 5 rounds. To make this estimate, it takes the current board state and the number of RISK cards owned by the players as input.

A change from the original design is that the IG, for the same reason as above, also gets information on whether or not it is the beginning of a round.

IG: The Opponents' Next Moves (IG NextMove)

This IG takes the estimates from all the other IGs, the current board state, and the opponents' RISK cards as input and outputs an estimated list of territories each opponent player will attack in his next turn. This information is obtained by having the AI look at the game from an opponent's point of view and determine what it would do itself given that it had the mission it has estimated the opponent to have.

IG: Ownership

This IG is not a part of the original design. It states whether or not a continent is fully occupied by a player. It takes the board as input and the output states which continent is owned by which player. The output may also state that a given continent is not owned by any player.

2.1.2 The Master Prioritizer

The second layer in the AI framework is the Master Prioritizer (MP). The task of the MP is to determine what the AI's tasks in the current round are. It uses all information gathered by the IGs to prioritize these tasks. This results in a list of prioritized goals. The goals vary from conquering a specific continent to defending another, or attacking players. Basically, the job of the MP is to determine when in the game it is useful to work towards the AI's own mission and when it is nessesary to prevent other players from fulfilling their missions. This is done by prioritizing the player's own mission goals higher whenever none of the other players are close to winning, and prioritizing other players' estimated mission goals higher when they are getting close to winning. The output from the MP is a list of goals with an attached priority. The list of goals can be seen in Appendix C.

2.1.3 The Round Planner

The Round Planner(RP) is the third layer in the AI architecture. It is responsible of carrying out the goals prioritized by the MP. The idea behind the RP is that it should consider the goal distribution given by the MP and see what goals are possible to carry out in the current round.

The RP is responsible for the four phases that makes up a turn in the RISK game. These are decribed in the Section 1.3 and more detailed in Appendix B. These are called *The Planner*, which is responsible for making attack plans and defense lists that the *Reinforce AI*, the *Attack AI*, and the *Fortrify AI* will use to place armies, attack territories, and fortify territories accordingly.

Each of these AI parts together makes up the RP. The parts themselves are divided into a set of smaller modules that each handle specific tasks needed by that part.

RP Planner: Make Attack Plan

Attack plans are very important structures in the framework. They are lists going from an occupied territory to an opponent's territory and are paths the AI will use in conquering territories. This RP module handles the creation of these plans. To do so, it gets an input stating what the starting occupied territory is, what the target territory is, the MP goal distribution, and the current board state. The output is an attack plan going from one occupied territory through a series of opponent-occupied territories and ending in a opponent-occupied territory.

Apart from this, we redesigned this RP module to also state a number of armies to leave in a specific territory or continent. This calculation was meant to be elsewhere, but put here as it seems more fitting.

RP Planner: Calculate Attack Plan Cost

This RP Planner module is responsible for estimating how high the cost of performing an attack plan is. This is measured in number of armies. The cost is split into two seperate estimates. One which is an optimistic minimal cost, and one which is a bit more realistic: the estimated cost. To calculate these costs, the module gets an attack plan as input.

This input is slightly different than in the original design where the module apart from the attack plan also got the current board state and the MP goal distribution. These were removed through a redesign of the module. The information needed from the board was how many opponent armies that were placed in the opponent territories in the attack plan. However, this information was already possible to obtain through the attack plan itself, being a list of enemy territories. The MP goals were used to determine how many armies that were needed to leave behind at each territory. If a player had the mission "18 territories with two armies on each" and may need to leave two armies behind, this would be reflected through the MP goals. We decided to let this information be reflected directly on the attack plan instead.

RP Planner: Prioritize Attack Plan

This module is used to give an attack plan a priority. This priority is a measure for how much closer an attack plan is to satisfy the different MP prioritized goals. Generally, the more MP goals an attack plan satisfies, the higher priority it should get. If the goals are also given high priority by the MP, then the attack plan that fulfills them should score even higher. To give a priority, the module takes an attack plan, the current board state, and the MP goal distribution as input. The output is a priority which is attached to the attack plan.

RP Planner: Discard Attack Plan

This module's task is to remove attack plans that are far too expensive to fulfill, by outputting "yes" or "no" to discarding of the plan. To decide this, it uses an attack plan priority, the minimum army cost, estimated army cost, reinforcements received, and the number of RISK cards each player owns.

To this RP Planner module, information whether or not it is run in the beginning of the turn has been added to the original design. The reason for this is, that the number of reinforcements from RISK cards should only be used if the module is actually run in the beginning of a turn.

RP Planner: Calculate Defense Cost

This module is used to calculate how much it costs to defend a specific territory from the opponents' armies. To calculate this, the module takes the territory in question and the current board state as input. It also takes an influence map [Rab02] as input. This influence map states the influence of enemy armies on each territory on the board. Only influence on the territory in question is needed. The module outputs a cost as the number of armies required to defend the territory.

RP Planner: Prioritize Territory Needing Defense

This RP Planner module puts a priority on defending occupied territories. This is done by looking at the territory compared to the MP goals, the board, and the opponents' next moves.

In the original design, this module also used information on how many RISK cards each opponent had. This input was removed since this information is already indirectly encapsulated in the opponents' next moves.

RP Planner: Remove Goals

This module has been added. The conquering of a territory might have fulfilled one or more of the goals from the MP, and therefore the goal distribution should be revised. This module has been added, to avoid calling the IGs and the MP again, every time a territory has been conquered. This module takes a goal distribution as input and produce a new goal distribution. This module will be described in detail in Section 2.3.

RP Reinforce

The Planner modules have now been described. These modules have together created a list of attack plans with priority and costs. A priority and cost of defending each occupied territory on the board has also been calculated. This results in two lists: an *attack plan list* and a *defense plan list*. These two lists are combined to a *Merged plan list*, which is used by the modules in Reinforce AI and Fortify AI. A description of the modules in these parts of the Round Planner now follows.

RP Reinforce: Cash Cards

This RP Reinforce module judges whether or not the player should cash his RISK cards in the following round by outputting "yes" or "no" to cash cards.

This is done through looking at the priorities of attack plans in the merged plan, and the minimum and estimated number of armies needed to fulfill the plans. These are evaluated against the number of armies gained from occupying territories and continents, and the number of possible armies gained through cashing RISK cards.

In the original design both the priorities and the minimum and estimated costs are used as seperate inputs. However, this information is now attached directly to an attack plan. Therefore the merged plan is used as input instead of the seperate values.

RP Reinforce: Score Merged Plan

In the original design, defense plans and attack plans were sorted by their priority whenever the most important plan should be selected. But the most important plan is not always the wisest choice. Sometimes, also the cost of a plan is in many cases usable as a factor. So, some priority versus cost ratio is needed. The result is this module. The module takes a priority and a cost as input, and output a score. In Section 2.3 the module is described in detail.

RP Reinforce: Place Armies

This module's task is to select where to place the received reinforcements on the board.

As with *Cash Cards* it no longer uses seperate lists as input but rather a *Merged Plan List* to decide where to place armies. Furthermore the module takes the board as input to recognize which territories are better to place in over others. This may for example be border territories.

RP Attack: Score Attack Plan

In the original design, there were no actual modules in the Attack AI — the Attack AI just performed the attack plan. However, two module has been added, namely the Score Attack Plan module and the Do Attack Plan module. This module is used to put a score on an attack plan. The reasoning behind this module is that it is not always best to simply perform the highest prioritized attack plan. There is some ratio between the plans importance and its cost. This is what this module is used to evaluate. Therefore the input to it is a priority and cost, and the output is a score. This module will be described further in Section 2.3.

RP Attack: Do Attack

This module performs the selected attack plan, and it also aborts the attack plan is it becomes too expensive to continue. This module takes an attack plan as input. It outputs where to attack (from and to) and how many armies should be used in the attack. This module will be described in detail in Section 2.3.

RP Fortify: Transfer Armies

This module moves armies from one occupied territory to a neighboring occupied territory. To find out which territories to move between, the module has the defense list, the goal distribution and the current board state available as input. The output from this module is a starting territory, an ending territory, and a number of armies.

Initial Army Placement

The Initial Army Placement module (IAP) is only run in the beginning of the game. It is used to place armies in the territories given to the player as the game starts. To do so, it uses the output from the *IG: Estimating Opponents' Missions*, the current board state, and an influence map stating how much influence each opponent army has on each territory on the board. The mentioned output from the IG is what corresponds to the "Mission" input in the original design. However, this estimate is better to use since it gives the module access to what the IG believes the other players' missions are. This allows the AI to place armies not only due to its own mission, but also consider the opponents' mission while doing it. The output from this module is a territory to place an army in.

The framework has now been presented, inluding a brief introduction to some new modules. These modules will be described in detail, but first each if the AI techniques used in this project will be presented.

2.2 Applied AI Techniques

There exist many different techniques to implement an AI. In [CJJ06] a set of AI techniques were selected to be used. The reason for selecting exactly those, is that their are either used commonly in computer games, or because they seem useful. The following sections are based upon Section 2.8 in [CJJ06]. In this report, a discussion on the time and space complexity of each technique has been added to each section. This is useful when discussing the training of the techniques later in the report.

2.2.1 Scripting

Scripting is a very intuitive way of creating an AI. By analysing what situations that can emerge in the game, the AI designer designs different behavior into the AI to handle these situations. Even though this may sound very simple, it might be a rather complicated task since in most games the number of possible scenarios to handle can get very numerous. Hard coding scripts for an AI is one type of scripting. We define hard coding as designing the AI to work in one *specific* environment. The downside of hard coding the AI is that you cannot change anything in its working environment.

In Figure 2.2, an AI has a hard coded representation of a path from 1 to 4. If it is told to move from 1 to 4, it has been hard coded to know that it needs to go from 1 to 2, from 2 to 3 and from 3 to 4 to get to the destination. However,

if the environment was changed with the one in Figure 2.3, the hard coded AI would not see the connection from 1 to 4 with the old code. Therefore, if anything is changed in the environment the hard coded AI needs to be changed too. This little example was used to show that it is not a good idea to hard code everything. Rather it is a good idea to examine more dynamic ways of designing the AI so it can be used in different environments.

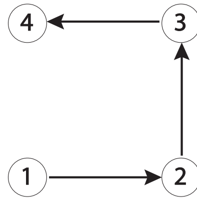

Figure 2.2: *A graph representing how the AI can move from 1 to 4.*

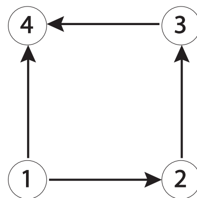

Figure 2.3: *Another graph representing how the AI can move from 1 to 4.*

Simple Scripting

A script is build up by having some *conditions* which needs to be met for the script to be run, and some *actions* which is the actual output of the script. This is illustrated in Figure 2.4. As seen, scripting is merely a set of *if-then* statements where the state of the AI and observations of the world determines what needs to be done.

```
if
  CONDITIONS_HERE_ARE_MET
then
  DO_THESE_ACTIONS
else
  DO_THESE_ACTIONS
```

Figure 2.4: *A basic script it divided into conditions and actions.*

Advantages and Disadvantages of Scripting

The major benefit of scripting is also one of its drawback. This is that the AI designer has full control over how the AI behaves in every situation in the game. This is also a drawback because the designer must make sure that the AI *can* in fact handle every situation. If some situations are overlooked, the AI might behave irrationally or even badly when the situation occurs.

Another drawback with the scripted AI is that it is not adaptive. It will never get better than its designer made it. This means that once a player has learned how to beat it in the game, he will most likely beat it every time since it does not change. Of course this last statement may not always be true since in most games luck plays a certain role, but in general when you figure out how an AI works, beating it becomes relatively easy. There are some measures that can be taken to avoid the AI becoming too predictable such as letting the AI choose randomly from more actions in different situations.

One major advantage concerning scripted AIs is that they are quite easy to understand and design. This is of course dependent of the size of the game, but in general this is true. Also a scripted AI does not need any training data to work and is therefore not dependent of the trustworthiness of such training data nor the effort needed to generate such.

2.2.2 Decision Trees

A decision tree (which is also known as a classification tree) is used for making decisions based on a set of training data. There are two types of decision trees. One which is learned through training data and one which is manually build. We will first cover the learned trees and later we will describe the second type of tree.

The tree structure is learned from training data. The tree can then be used to tell what to do when the world looks in a certain way. A decision tree can be used to *classify instances*. An instance would in this context then be the state of the world at a certain point in time. The world can then be classified as being in a different discrete amount of states. The world is represented through attributes having different values. Basically the final result of building the decision tree is a set of *if-then* statements used to decide on a specific state.

The Training Data

The training data required for building the tree is a range of attributes with associated values that describe certain observations in the world. The number of different values that an attribute can have needs to be discrete and finite, meaning that they cannot just be e.g. integers. One of the attributes is the *target attribute* which is the attribute that the tree outputs, being the decision taken by the tree. In the decision tree, each attribute will be a node with a number of edges leaving it. This number is determined by the number of values the attribute can have.

Building the Decision Tree

An approach to building the decision trees is using the *ID3* algorithm [Mit97]. In the *ID3* algorithm all attributes are tested to see which one gives the highest *information gain* on the target attribute. The information gain is a measure stating how much information there is gained on the target attribute from the attribute being examined. Calculating the information gain requires calculation of the *entropy* for the attributes. The entropy is a measure of how "dirty" the training data is. If all examples of a certain attribute in a specific state results in the target attribute being in only one specific state, the entropy is zero. The further the entropy moves from zero the dirtier the data is. The link between entropy and information gain is that the lower the entropy, the higher the information gain.

Calculating the information gain is the main part of the ID3 algorithm. The attribute with the highest information gain on the target attribute, is used as the root node of the decision tree. This node has a number of children corresponding to the number of possible values the attribute has. Which node is placed on a branch is determined by re-calculating entropy and gain based on the value of the edge. This is done for all edges going from the parent.

This procedure continues until all examples has zero entropy or all attributes have been entered into the tree from the root and down. Attributes are only considered once in any branch of the tree meaning that going from a leaf to the root, any attribute will only be met once. This makes sense since an attribute already entered into the tree cannot supply any more information further down the tree that it has already done once. A more concrete example with training data can be found in [Mit97] from page 59.

The result is a tree based on classification examples where every attribute node branches into the number of values possible for that attribute, until the target attribute is reached as the bottom leaf. The leaf is the decision the tree decides. This corresponds to the action output from for example a script. An example of a tree is seen in Figure 2.5.

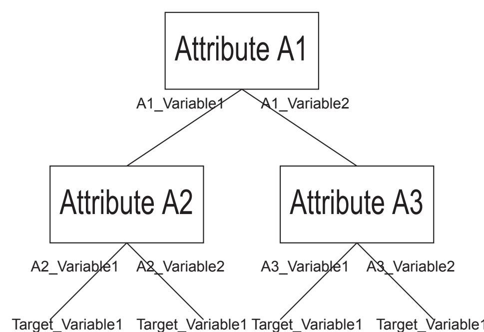

Figure 2.5: *An example of a decision tree with three attributes and a target attribute(the decision that is output). Each have attribute has two different values.*

At the nodes of the decision trees, we keep track of how many observations there are of a specific path. This ensures that the tree can be used for making decisions on events that have not previously been observed. If a new unknown

event is observed, it is tracked through the tree until the unhandled event occurs. The tree is then traversed further through the "most likely" events, meaning the ones that has occurred most frequently.

Converting the Tree to Rules

After the tree has been built, it can be converted to a set of rules. This is done by tracing each possible path from the root node to the leafs.

These rules are simple *if-then* statements and are therefore both easy to grasp and fast to work with.

Time and Space Complexity

As found in [Wu95] the time complexity of training a decision tree is as follows: In the trained decision tree, the maximum number of leaf nodes n is equal to the total number of training examples. The maximum length from the root to each leaf node a is equal to the number of attributes. So the total number of nodes in a decision tree will always be less than na. At the root, ID3 must for each example calculate the information gain for each attribute. If b is the maximum number of possible values for an attribute, the time complexity for this is $O(bna)$. Time complexity at other nodes in the tree is always less than that at the root. Therefore, the worst time complexity for ID3 is $O(bna \times na) = O(bn^2a^2)$.

ID3's total computational requirement per iteration is thus proportional to the product of the size of the training set, the number of attributes and the number of non-leaf nodes in the decision tree. The same relationship appears to extend to the entire process, even when several iterations are performed. No exponential growth in time or space has been observed as the dimensions of the induction task increase, so the technique can be applied to large tasks, claims [Qui86]. But because ID3 potentially will need to keep the whole set of training data in memory while training, the combination of available computer memory and the size of the training data will set the limit.

The time complexity for classifying an instance with the trained decision tree is $O(a)$, since it in worst case will be needed to travel through a nodes starting at the root before reaching a leaf node. Looking at the space complexity in worst case, the decision tree will for each attribute node need b children at every level in the tree. An since the worst number of levels is a, the worst case space complexity is $O(a^2b^2)$.

2.2.3 Neural Networks

An artificial neural network (ANN) is a rough way of simulating the human brain electronically.

Generally speaking ANNs are used to derive a meaning from a complicated or noisy set of data, such as character and face recognition. It is also applicable to problems for which symbolic representations are often used, such as different elements (e.g. the board) in a RISK game. A trained ANN can be seen as an "expert" giving his opinion on a given dataset.

An ANN takes some input (e.g. real numbers) and produces some output (e.g. real numbers again). Whether the network produces the correct output is

determined by the modeler or some other benchmark. If the output is incorrect, the network is adjusted and an output is produced again. If the network produces a correct output the network is not adjusted. This is called *training the network*. Training is repeated until the modeler thinks the network performs satisfactory. However, repeated training on an ANN may result in *overfitting*. Overfitting is where the network is adjusted to respond to the training data only, and not to the general case as it was intended to do. Overfitting ANNs is also discussed in Section 4.6.5 in [Mit97].

Generally speaking, ANNs are networks made of artificial neurons connected to each other. Such networks can be connected in many different ways to form an ANN, but this section will only cover the *feedforward* type of network.

But before discussing artificial neural network architecture, it is useful to discuss these artificial neurons.

The Sigmoid Unit

An artificial neuron (also called a *neurode*) can be modeled in different ways, depending on its usage. The most used neurode is called the *sigmoid unit*. The unit receives an arbitrary number of inputs and produces a real numbered output. Each input has an associated *weight* which determines the contribution of that input.

The input to a sigmoid unit is multiplied by each weight and summed resulting in the net input to the unit:

$$net = \sum_{i=0}^{n} w_i x_i > 0 \tag{2.1}$$

where n is the number of inputs and w_i is the weight for input x_i.

The output is then calculated as follows:

$$o(net) = \frac{1}{1 + e^{-net}} \tag{2.2}$$

The output ranges from 0 to 1, and it maps a very large input domain to a small range of outputs and is therefore often called a *squashing function* of the unit. Figure 2.6 shows the sigmoid function.

Neural Network Architecture

The architecture of neural networks deals with how neurodes are combined to form a neural network. This section will only cover the *multilayer feedforward* type of network, as it is the only type used in this project. Other types are explored in [ED90].

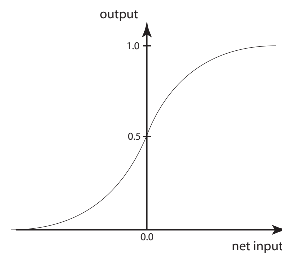

Figure 2.6: *The S-shaped sigmoid function takes a large input domain and outputs a real number between 0 and 1.*

The multilayer feedforward[1] network has three types of layers (see also Figure 2.7):

1. **Input layer**: Each input node is connected to each node in the hidden layer. The number of input nodes entirely depends on the application for which the network is intended. The input layer has no weights associated.

2. **Hidden layer**: There can be an arbitrary number of hidden layers. Each neurode in a hidden layer is connected to each neurode in the next hidden layer. The number of hidden layers and the number of neurodes in each hidden layer also entirely depends on the applications. Too few hidden neurodes and the network will not be able to learn the training data, but too many neurodes and the network will take forever to train and also overfitting may occur.

3. **Output layer**: The output neurodes take the output from neurodes in the last hidden layer and calculates an output. This is the output from the network. Again, the number of output neurodes depends on the application.

Training Neural Networks

Training neural networks is simply a matter of tweaking the weights in the network in order to get a satisfactory output. The most common method is called the *back-propagation algorithm*.

The Back-propagation Algorithm

Essentially, the back-propagation algorithm (BPA) learns the weights for a multilayer network with a fixed set of neurodes and interconnections. BPA attempts to minimize the difference between the actual output of the ANN and the expected output given by a set of training examples. With many different training examples, the ANN should hopefully conform to the general case that the training examples in combination describe. It should not conform to each

[1] In some literature called a *back-propagation model*, since the *back-propagation algorithm* is often used on this type of network. But to avoid confusion, we will only refer to this type of network as a feedforward network.

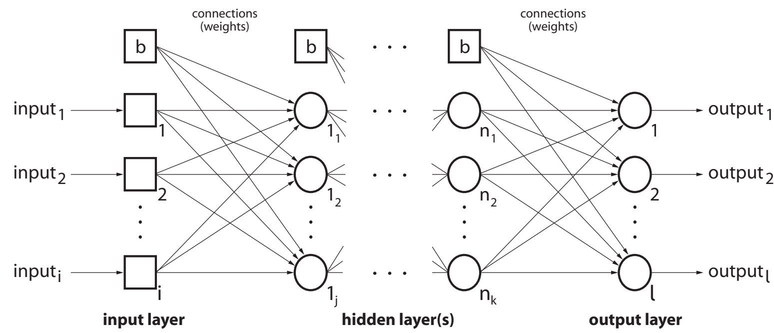

Figure 2.7: *The multilayer feedforward model. Each circle is a neurode and each square is an input node. The network has i input nodes, n hidden layers and l output nodes. Hidden layer 1 has j neurodes and hidden layer n has k neurodes.*

training example separately, since there in some cases could exist two training examples with the same input, but with different output. The ANN should in such a case produce some average between the two.

In this relation, it is useful to have some measurement of how well an ANN has been trained, this is called the *error measure*. The error measure (E) can be viewed as the difference between the output of the network and the training examples.

In Figure 2.8 one can see a visualization of the *hypothesis space* H of possible weights and their associated error E. In this context the hypothesis space is the set of all possible weight values in a ANN, and the resulting error E for each combination of weight values. This produces a hypothesis space with n number of dimensions, where n is the number of weights in the network plus 1. The extra dimension is the resulting error E. In Figure 2.8 there are only two weights, which results in a 3-dimensional hypothesis space H.

Minimizing E is a matter of finding the point on H with minimum error. It has been compared with letting a small ball roll down H and where it comes to rest, is the point of minimum error. But H might have a local minimum where the ball also could come the rest - a small dent in H. The BPA is unfortunately only guarantied to converge to such a local minimum, but in practice, it has shown to produce excellent results. Testing whether a local minimum has been reached, one could perform the BPA again from a new starting point and see if any lower error is reached.

The arrows in Figure 2.8 shows the direction in which the weights should be tweaked to produce a smaller E. Finding such a direction is a matter of calculating the derivation of $E(\vec{w})$. This is also called the *gradient* of E with respect to $\vec{w}$ (later in this report also called the error vector). An elaboration on this can be seen in [CJJ06], Section 2.8.4.

Before applying the BPA, a network should of course be constructed with n_{in} input nodes, n_{out} output neurodes and n_{hidden} hidden neurodes. All weights in the network should be initialized to a small random number[2] (e.g. between -.05 and .05), but any value is permitted, which correspond to starting a random

[2]Finding the initial values of weights should be done intuitively. In [ED90], it is recommended to use random values between -.3 and .3 for the initial weights.

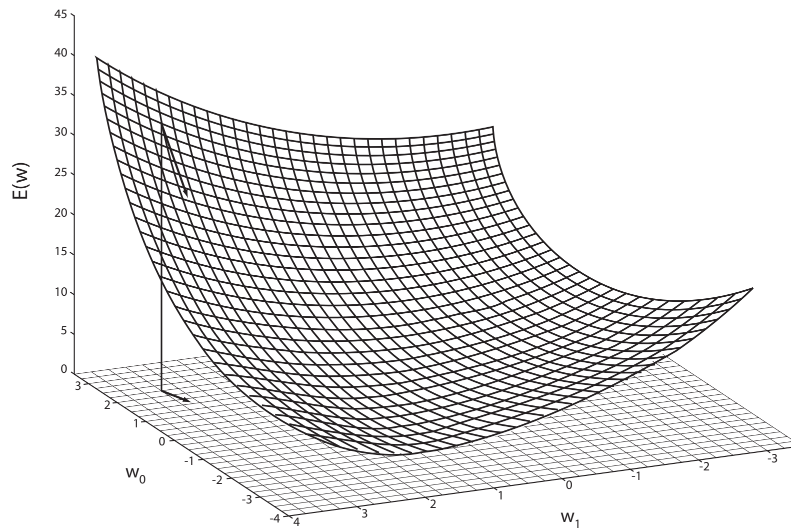

Figure 2.8: *For a linear neurode with two weights, the hypothesis space H is the w_0, w_1 plane. The vertical axis indicate the error E for a given weight vector hypothesis. The arrows shows the negated gradient in a particular point, indicating which direction produce the steepest descent along the error surface.*

place in the weight space.

The BPA then starts out by calculating the output of the network given the input from the first training example. It then calculated the error in all outputs of each neurode for this example. The weights are then updated using the error in each neurode. The updating of weights is slowed by a *learning rate*, which is multiplied on the error, before it is used in the updating of the weights.

The algorithm then starts over again with a new training example. The set of training examples are repeated again, unless some termination condition is met. This could for instance be when the error measurement E has become sufficiently close to zero. But one should also be aware of any overfitting with repeated training. Overfitting could be continuously monitored by calculating a *generalization accuracy* which states how well the network conforms to a given generalization example.

Time and Space Complexity

The time complexity of the BPA was in [SL94] compared to another neural network training algorithm. The time complexity of BPA was empirically found to $O(N^4)$ for the N2N encoder problem. The N2N encoder problem deals with successfully training a neural network with N input units, 2 hidden units and N output units connected in a standard feedforward network. The neural networks used in this project does not necessarily compare to the N2N encoder problem, but it can set a worst-case standard for the complexity of the BPA.

The space complexity of the training of neural networks is linear. The only information needed to store between training iterations is the weights in

the network, which are calculated as $\#weights = \#hidden(\#input + \#output)$, where $\#input$, $\#hidden$, and $\#output$ are the number of input, hidden and output units respectively. It is obvious to see that the space complexity is linear with respect to the number of units in the network. With respect to the amount of training data, the space complexity is constant. Except the weights in the network, no data needs to be stored between each training example.

Querying the neural network is just a matter of calculating the output for each neurode, and can obviously be done in linear time with respect to the number of weights.

2.2.4 Bayesian Networks

A Bayesian network (BN) is a model of the relations between different events in a real world situation. It is useful for supplying information on a specific event based on observations on other events. This information is expressed in certainties. As an example, take a car that will not start. You see two possible causes for this: you are out off gas or the spark plugs are dirty. Both causes influences whether or not the car will start. If you *know* that the car is out of gas, you believe this is the cause of the car not starting. This will of course lower your belief that the cause it the dirty spark plugs (even though they may be dirty too).

The BN Structure

The following is the definition of a BN [Jen02]:

- A BN is a set of nodes interconnected by directed edges.
- The nodes are variables with discrete set of mutually exclusive states.
- The nodes and edges together form a directed acyclic graph.
- Each variable A with parents $B_1, ..., B_n$ has a *potential* table attached which specifies $P(A|B_1, ..., B_n)$.

The *potential* is a probability table for a variable stating how likely a variable is to be in a certain state. This is also referred to as the conditional probability table (CPT) for the variable.

An example of a BN is seen in Figure 2.9. If a variable has been observed to be in a certain state, the variable is said to be instantiated. Another term used in this situation is that evidence has been entered on the variable. When evidence has been entered into a BN, it is propagated. This means that the evidence is distributed to all relevant nodes in the BN.

Propagation in BN

Propagation in BNs are performed in a *junction tree*. A junction tree is build in the following way.

First the BN is moralized. If two parents have a common child, there is put a link between them. The moral graph is undirected.

With the moral graph, the following procedure is used to build a junction tree:

- Choose a simplicial node X to be the first in the elimination order. That the node is simplicial means that F_X is a clique.
- Eliminate nodes in F_X that only have neighbors in F_X.
- Denote F_X with an index V_i with i being the number of nodes eliminated from F_X.
- Denote the nodes remaining from F_X with S_i. This set is called the *separator set*.
- Remove eliminated nodes from the graph. Choose the next clique and repeat the procedure without resetting i.
- Continue until all cliques have been removed.
- Connect all cliques V_i to a separator S_i.
- Connect all separators S_i to a clique V_j where $j > i$.
- Attach two mailboxes to the separators for sending and receiving information in the tree.

The steps involved in propagation are covered in [Jen02]. Roughly propagation is done through two steps: a *collection* step and a *distribution* step. In the collection step, information is sent towards a chosen root node from the leafs. In the distribution step information is sent the opposite direction, from the root to the leafs. When both these steps have finished, the tree is fully propagated. The point of passing information up and down the tree is to keep it consistent for instance when evidence is entered at some node.

A message between two cliques consists of the potential on the sending clique where the variables not in the separator are marginalized. This is possible because between any two clique-nodes in a join tree, their intersection of variables is present in all nodes between them. The intersections is what is placed in the separators. This means that if any variable is needed further down in the tree, it will be present in the separators on the path between the nodes, meaning that it will not be marginalized.

Bayesian Learning

Instead of manually specifying the CPTs of a BN, which is often a very difficult task, learning can be used. This learning is done based on training data. The training data, in the case of a BN solving a task in the RISK game, is information gathered on previously played games. From these past games, statistical information is gathered concerning values of each node in the BN. For example for a node A with the states a_1 and a_2, the occurrences of each state will be respectively $\frac{a_1}{ALL}$ and $\frac{a_2}{ALL}$, where ALL denotes the number of occurrences of A.

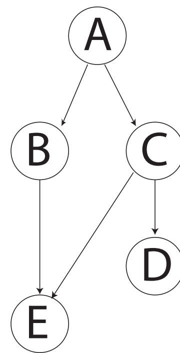

Figure 2.9: *An example of the BN structure.*

The previous method works if training data is available for the states of all nodes in the BN. However, this may not always be the case. Often there may be nodes which are not part of the training data. Therefore their probability distributions cannot be specified through statistics. To specify these, Bayesian learning algorithms can be used. These algorithms takes already known information (found as above) and uses it to tune the unspecified nodes' probabilities towards usable distributions.

The EM Algorithm

The Expectation-Maximization (EM) algorithm is an example of such a Bayesian learning algorithm. As described in [Tho], if the training data is missing information on some variables, the algorithm gives an estimate on the most likely state of the variable. Each iteration consists of two steps, first the expectation step where the missing values in the data set are filled with calculated expected values, then the maximization step uses the filled-in data set to calculate new maximum likelihood estimate for the variables. This way missing values are filled in and can be used in the next iteration. The algorithm will continue until the difference between the maximum likelihood estimates calculated in iteration n and the maximum likelihood estimate in iteration $n - 1$ is smaller than some predefined threshold value, or has run for some predefined number of iterations.

This can be useful in the game of Risk because although most variables are always observable there might be some which can never be observed.

Modeling Techniques

In this section some modeling techniques that might be useful when the BNs will be constructed in the Design phase.

Divorcing

The modeling of a BN can be quite tedious to do by hand since the size of the conditional probability table (CPTs) of each node grows exponentially with its number of parents. Thus the amount of probabilities to specify grows at the same speed. Specifying them by hand will therefore take a very long time.

Also when computing the probabilities in the BN the sheer size of it will make the calculation time increase rapidly and the BN itself may be too big to handle. To reduce the CPT size *divorcing* can be used when modeling. An example of this can be seen on Figure 2.10.

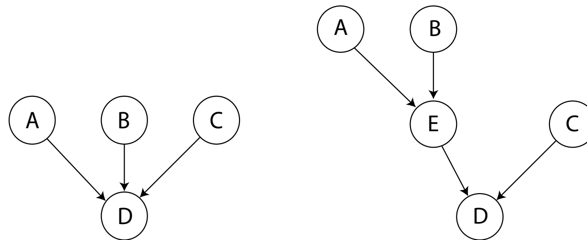

Figure 2.10: *The node E divorces A and B and thereby reduces the CPT of D.*

E is used to lower the number of states the variable D has to consider. The divorcing results in D having a smaller CPT if the number of states in E is lower than the combined number of states in A and B.

Time Complexity

The EM algorithm handles each training example independently. This means that its computational complexity with respect to the size of the training data i linear. However, the EM algorithm does a propagation of evidence in the M-step. The computational complexity of propagating a BN is known to be NP-hard.

2.2.5 Naive Bayes Classifiers

A naive Bayes network is a special BN. It is one where the observed node is child of all other nodes in the network. This structure is seen in Figure 2.11, where a target attribute is parent to all other attributes in the BN. Naive Bayes uses an assumption that all attributes in the underlying model are independent of each other [Mit97]. This assumption is to simplify the calculations for the classification since the probability of seeing a specific conjunction of the variables is found by multiplying the probabilities for the attributes. Notice even though this assumption may not always hold, naive Bayes may in fact still perform quite well. Using a naive Bayes classifier, it is possible to classify new training examples based on experience from training data.

The Training Data

The training data for the naive Bayes classifier (NBC) is much like the one used in decision trees. Meaning that there is one target variable that is decided which is dependent on the other variables.

Learning

Classifying a new instance, which is a new observation of the attributes' states, is done in the following way:

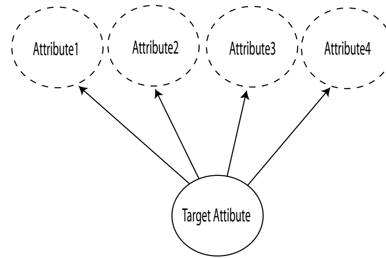

Figure 2.11: *A naive Bayes network.*

$$v_{NB} = \underset{v_{j \in V}}{argmax}\, P(v_j) \prod_i P(a_i|v_j)$$

where v_{NB} is the target value that is output. V is the set of states in the target attribute. a is an attribute from the new observation and v is the target attribute. When the NBC is presented a new configuration of attributes to decide on it compares it to the training data. For each state v in the target attribute, $P(a_i|v)$ is calculated. Finally the product of all $P(a_i|v)$ and $P(v)$ is found. The result of the classifier is the one where v results in the highest probability.

Time and Space Complexity

Since the time required to do the calculations for each training example is constant, this does not affect the time complexity. However, these calculations are needed to be done for each training example, so the time complexity is then $O(n)$ where n is the number of training examples.

The space required for the naive Bayes classifier is constant since the size of the network never changes.

2.3 Additions to the Framework

Some additions have been made to the design of our framework. This has been done through implementation of the original design and then reassessing the decisions made, resulting in additional modules.

2.3.1 New Modules

Five new modules have been designed. These do not incorporate very large changes, but are merely added because some functionality were needed in the framework. The new modules are: the IG “Continent Ownership”, “Remove Goals”, “Score Attack Plan”, “Score Merged Plan” and “Do Attack”. Each new module will be described in the following section, including an analysis of which AI techniques could be used to implement their behavior.

Continent Ownership

This module has been made to give the MP access to some simplified board information. It is in fact the list of continents and the owner of that given continent. The owner might be null, if no player owns the entire continent. No AI technique (neural network, decision tree etc.) could give better information than the direct information from the board. So it does not make sense to have anything other than scripting for this module.

Remove Goals

The conquering of a territory might have fulfilled one or more of the goals from the MP, and therefore the goal distribution should be revised. This module has been added, to avoid calling the IGs and the MP again, every time a territory has been conquered. For instance, the goal distribution given by the MP might have some positive value in "obstruct Africa", but the last attack just conquered a territory in Africa. Then it would in fact be necessary to run each of the IGs and the MP again, to revise the goal distribution. But this takes some time, and it seems more efficient and just as effective to remove fulfilled goals and then continue with the same goal distribution. One could also argue, that if a new goal distribution were calculated, it most likely would resemble the last goal distribution, since no territories have been lost and the enemy has not gained any reinforcements. Only if a whole continent has been conquered by the AI, then the IGs and MP is called again, since that is a major change in the power balance.

Analyzing whether or not a goal has been fulfilled is simply a matter of looking at the board and compare it to the goals one by one. So only scripting will be used for this module. No AI technique would do it any better.

Score Attack Plan

Instead of just selecting the attack plan with the highest priority, there should be some evaluation of the attack plan where the cost also is used. For instance, when two attack plans are almost equally important (the priorities are almost equal), but one of the plans is much cheaper - then the cheapest should be selected. This would naturally lead to a scoring function where $score = priority/cost$. A few examples of such a function can be seen in column 3 in Table 2.1. As one can see, such a function favors very low cost much more than priority, which is not the intention. In this example, the plan with priority 0.3 is in fact selected. The function is therefore changed to favor high priority instead. The function $score = priority^{10}/cost$ is found suitable. This function is depicted in column 4 in the same table.

It is not certain whether or not this function is the best when selecting attack plans, so it should be analyzed which AI techniques could handle this module better than our script:

Neural Network: It is straight forward to have a neural network that has two real numbered inputs and a single output producing a score.

Decision Tree: It is also straight-forward to have a decision tree with two attributes for the cost and priority and then a target attribute for the score.

Priority	Cost	*Priority/Cost*	*Priority*^{10}*/Cost*
0.90	4	0.2250	0.08717
0.85	3	0.2833	0.06562
0.70	8	0.0875	0.00353
0.60	7	0.0857	0.00086
0.50	3	0.1667	0.00033
0.30	1	0.3000	0.00001

Table 2.1: *The Priority vs. Cost function. Third column depicts the most common function. Fourth column depicts the function used in the "Score Attack Plan" module.*

The values of each of the attributes should be divided into states: "0.0-0.1", "0.1-0.2" etc.

Naive Bayes Classifier: Implementing this technique for the score module can be done in the same manner as for the decision tree.

Bayesian Network: A BN for this module would resemble the Naive Bayes Classifier, since the priority and cost both influence the score and they are both independent of each other. So there is no need for a BN for this module.

Score Merged Plan

A *Merged Plan List* is a list consisting of both attack plans and defense plans. This module scores the elements of this list like the module "Score Attack Plan" did for attack plans. The reasoning behind this module has already been argued in Section 2.1.3. The AI techniques that seem usable to this module are the same as for "Score Attack Plan".

Do Attack

Instead of just performing the attack plan as the original design dictated, it was decided that an attack plan might not necessarily go as planned. Therefore, the ability to abort an attack plan was added and this new module was hereby created. There are two criteria that can stop an attack plan:

- If the attack plan has become more expensive than the estimated cost.
- If the probability of winning the next battle in the attack plan is too small.

If any of these criteria are true, then the attack plan is aborted and a new set of attack plans is constructed.

2.4 Framework Behavior

This section will cover the behavior of the framework. This was not described in detail in the original design, but is very important because it could potentially put some limitations on the performance of the AI techniques. The behavior of the framework between the modules (e.g. merging plans etc.) stays the

same, even when the modules implemented as different AI techniques evolve their behavior. This means that the performance of an AI technique is limited by the behavior of the framework. One can say that the behavior of learning algorithms are limited to the environment in which they are implemented.

One could even say that the design of the framework could be a limitation on the AI - perhaps there exist some better design and combination of modules, that would perform better than our framework. But even with a non-perfect design, one can still compare AI techniques, since they all are implemented in the same system. The performance of each AI technique is therefore only measured using our framework.

But as stated above, analyzing where the behavior of the framework could potentially limit the AI is important. This analysis will be made alongside the following description of the behavior of the framework.

For a better overview of the framework, Figure 2.12 shows the dataflow in the framework.

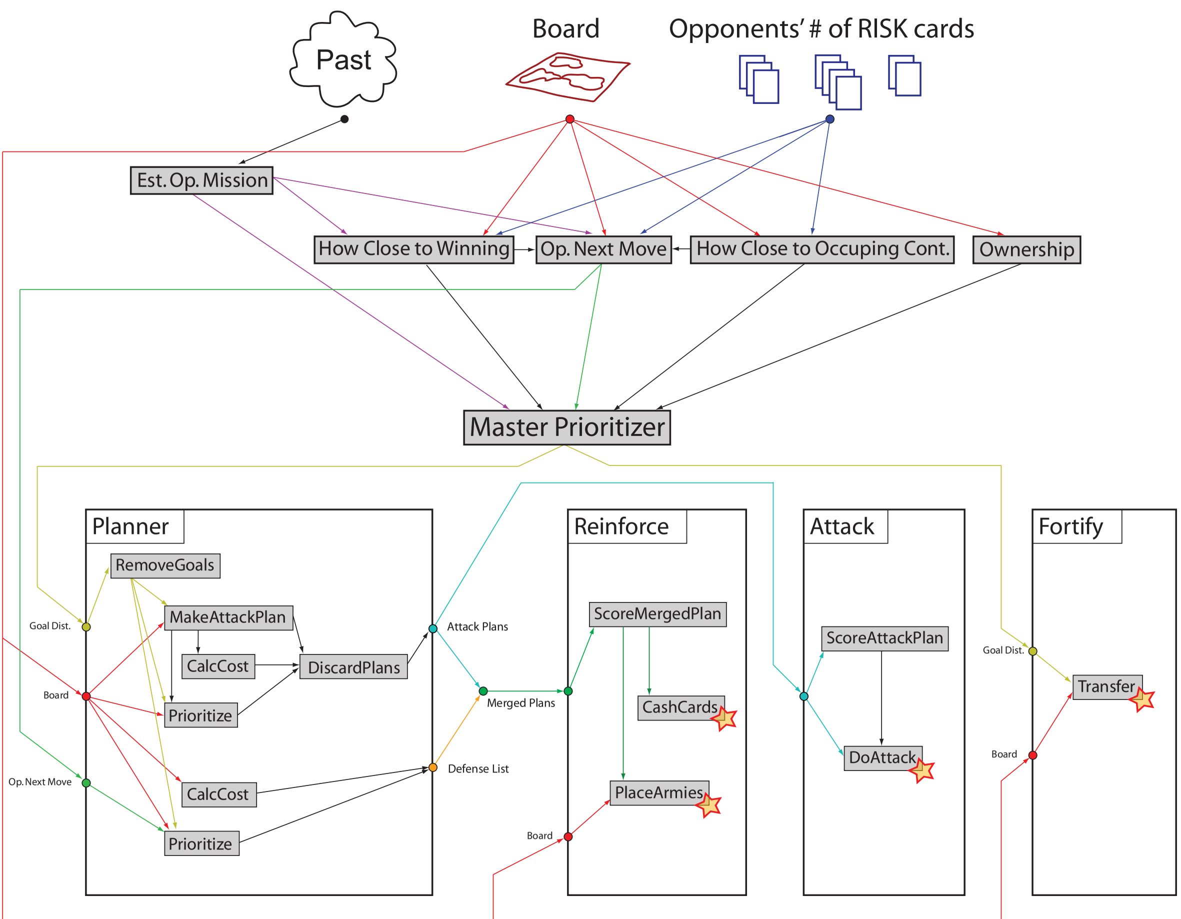

Figure 2.12: *The information flow in our framework. At the top, the different information directly available (the past, board and opponents' number of RISK cards) is located. These feed information to many different modules in the framework. Below the direct available information, the Information Givers are located. They mainly feed information to the Master Prioritizer below. The MP delivers a goal distribution to the different parts of the Round Planner. These parts, called the Planner, Reinforce, and Fortify are located at the bottom of the figure. The colors on the arrows are merely for making the figure more readable. All gray boxes are modules. The stars indicates that the module performs a Risk action, such as placing armies on the board.*

2.4.1 Class Structure Overview

The class diagram [Lar00] for the framework can be seen in Figure 2.13. In the figure, public methods on classes are not included. The important methods are discussed later. Dashed boxes illustrate classes already implemented in JRisk[3]. Thin lined boxes are "module" classes, which are used in implementing the behavior of each module in our framework. Thick lined boxes illustrate "framework" classes, which are used to control when the different modules are called and do data conversion between modules. All "module" classes also have a "TrainingExampleWriter" class associated, which deals with producing training examples for the learning algorithms.

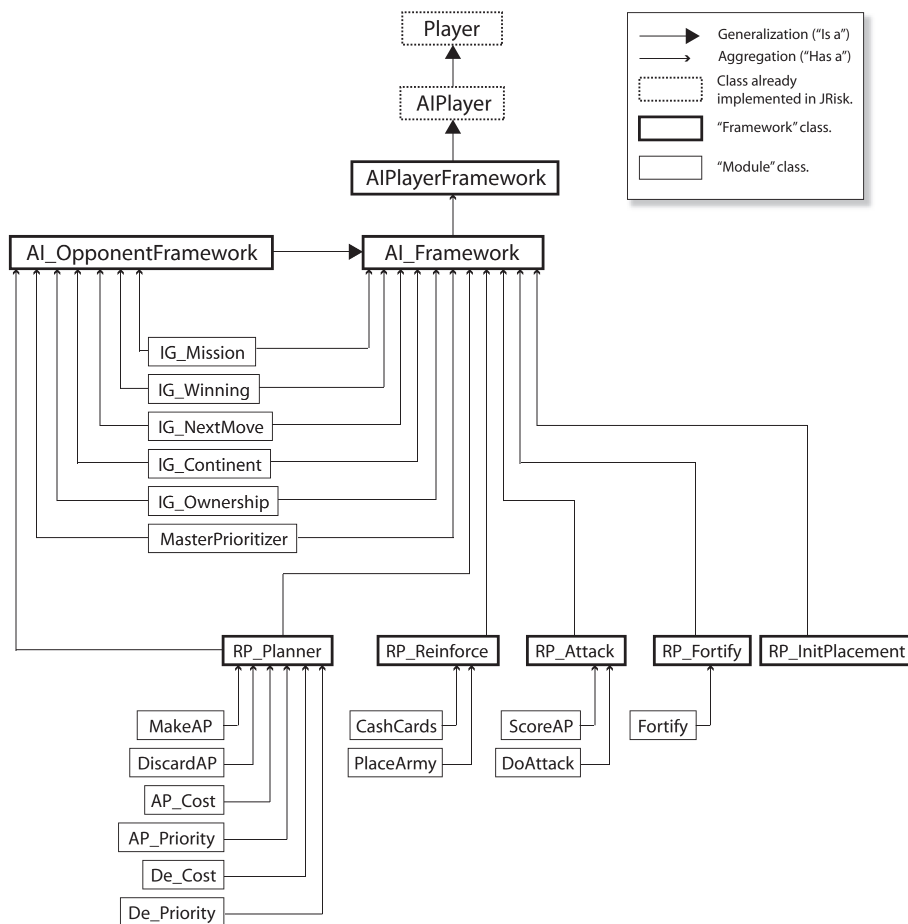

Figure 2.13: *A simple class diagram for the framework.*

[3]In the version of JRisk, which were used in the original design, the AIs in the game were all implemented in the player class, not as stand-alone classes. We corrected this and implemented all AIs as classes inheriting the AIPlayer class. A later version of JRisk adapted this design (though not due to us).

AI_Framework Class

The AI_Framework class is the main class in our framework. This is responsible for running all the IG modules, the MP and the different parts of the RP according to which phase the game is in. Recall that the game of Risk can be in four different phases:

1. Initial army placement.
2. Reinforce.
3. Attack.
4. Fortify.

The game is only in phase 1 once per game, but phases 2-4 are repeated each time a player gets a new turn. In JRisk, these phases have been expanded, which is also reflected in AI_Framework class:

- **Trade cards:** This phase is where the player decides whether to cash cards or not.
- **Reinforce:** This is both the normal reinforce phase, but also used in the initial army placement. A boolean flag is set to indicate which phase it is.
- **Attack:** This is the normal attack phase, but has been split into the three next phases also. This phase is for selecting where to attack.
- **Roll:** This phase is for deciding how many dice to use in the attack.
- **Defend:** This phase is for deciding how many dice the defender will roll.
- **Move:** This phase is when the attack has been won and armies must be moved to the conquered territory.
- **Fortify:** This is the normal fortify phase.
- **End turn:** This phase is for doing clean up when a player is done fortifying and can do nothing else than end his turn[4].
- **Game ended:** This phase is for doing clean up and reading the winner's mission when the game has ended.
- **Select capital:** This is used in a special kind of Risk - Capital Risk. It is unused in our framework.

In the AI_Framework class, each of these phases have a corresponding method. Every time the AI has to make a decision, the phase is checked and the corresponding method is called. These methods then call their respective modules and convert the output to actions in Risk. Some of these methods are more interesting than the others concerning imposing limitations on the AI.

[4] In our version of JRisk, one can not be certain that the game ever comes in this phase at all, so it is unused in the framework. Any clean up can be done just before returning the fortify command.

The RunRP_Planner Method

The first method is called *runRP_Planner* and is responsible for calling the modules in the Planner class, but also for converting the output of those modules into a "merged plan list". This list is used in the Reinforce class to determine where to place new reinforcements. Recall that the Planner class produce "attack plans" and "defense plans". A single attack plan contains a list of territories, but a single defense plan contains a only a single territory. Merging these plans are described in the original design and has not been changed - the start territory in each attack plan is kept and the rest is removed. This list of start territories is then merged with the territories from the defense plans. This results in a merged list of territories with a priority and a cost. The *place_army* module then place armies in the territories with the highest score calculated by the "Score Merged Plan" in this merged list. There are a number of issues with this procedure:

- **Attack plan information is thrown away:** Perhaps the reinforcement would perform better, if it had access to each entire attack plan. But the removal of most of the attack plan was decided because it was concluded through analysis, that none of the AI techniques seemed to perform very well with massive amount of input, which entire attack plans would be. The only relevant information seemed to be the starting territory for each attack plan, including cost and priority.
- **Defense priority vs. attack plan priority:** As stated above, the merged plan list is sorted by score, for the *place_army* module to place armies in the most important territories only. This could either be a territory from an attack plan or from a defense plan. But for this to be sound, the score of a defense plan and the priority of an score plan must be comparable. But calculating the score is based on its priority and cost, and calculation of the priorities for attack plans and defense plans are based on entirely different information. The attack plan priority is based on how well the attack plan fulfills the goal distribution given by the MP, and the defense plan priority is based on the "next move" estimate of the opponents. This is a potential problem, that one must be aware of.

All of these issues might potentially impose limitations on the AI, but as argued, they can not be without.

The Attack Method

The second method is the *attack* method, which now also will be described a bit more in detail.

The *attack* method is mainly responsible for calling the *scoreattackplan* and the *doAttack* modules, which are located in the Attack class. First of all, the *attack* method takes the list of attack plans produced by the Planner and feeds it to the Attack class. The Attack class use the *score* module to find a score for each attack plan. The *doAttack* module is then called with the attack plan with the highest score as input. The *doAttack* module just attacks from the first territory in the attack plan to the second territory in the plan. But instead of just performing the attacks, it also keeps track of how well the attack plan

progresses, meaning that it compares the actual cost of the attack plan so far with the estimated cost. If the attack plan becomes too expensive, it should be stopped and revised. This is done by outputting a boolean flag as true. This flag is then read by the *attack* method in the AI_Framework class, which then runs the planner again to generate new attack plans. This flag can also be set to true if the probability of winning the next attack is lower than some threshold.

If, at some point the planner no longer produce any valid attack plan (because the *discard* module has discarded them all), the command "endattack" is given, and the game goes into the fortify phase.

There are some limitations in the *attack* method:

- One of the limitations of the *attack* method, is the fact that the *doAttack* module is only implemented as a script, and because the term "revise the attack plan, if the cost gets too expensive" impose some limitations, since "too expensive" is defined by the designers, and not found through learning.
- The threshold for when the probability of winning a battle is too low is also decided by the designers. One way of dealing with this problem would clearly be to have learning involved. This was actually considered in the early stages of the original design, but removed because it seemed unnecessary. At this point it would require large changes in the design, and compared to the possible benefits that would be gained from it, it has been decided not to include learning in these two cases.

AI_OpponentFramework Class

This class is used when estimating the "next move" of an opponent. When the "next move" module is called, an instance of this class is created for each opponent in the game. The class uses the same IG instances as the real framework, so they are not called again - the same estimates are just used. But a new instance of the MP and the Planner class is created. The MP and Planner are run, and a list of attack plans are produced. The most important (the highest score provided by the score module) attack plan is then used as the "next move" of that opponent.

There does not seem to be any other limitations in this class, that the ones imposed by the framework. Also, there was not found any learning algorithms usable to implemented this module, so whether or not this class sets any limitations is in fact irrelevant.

RP_Planner Class

The Planner class is the class that handles everything concerning attack and defense plans. It makes all possible plans from the current board, gives them a cost and priority, and discards bad plans. For each occupied territory on the board, a "defense cost" and "priority" are calculated by calling the appropriate modules.

The Planner class has a $run()$ method which does the above. In a bit greater detail, this is:

1. Run *RemoveGoals* module.
2. From all occupied territories to all unoccupied territories, run *MakeAttackPlan* module and add the resulting path to a *list_of_attack_plans*.
3. Run *AP_Priority* module on each element in that list.
4. Run *AP_Cost* module on each element in that list.
5. Add each list element that only attacks a single territory to a *simple_attack_plan_list*.
6. Run *DiscardAttackPlan* on each element in *list_of_attack_plans*.
7. If the *list_of_attack_plans* is empty and the player has not received a RISK card in this round, add *simple_attack_plan_list* to the *output*. Otherwise add *attack_plan_list* to the *output*.
8. For all occupied territories, run *De_Priority* and *De_Cost*, and add territories to a *defense_list* that have priority and cost larger than zero.
9. Return the *output* from this *run()* method.

When the *discard* module discards all attack plans, it would result in the AI ending his attack an go into the fortify phase. But this is not always a good idea. If the AI has not attacked any territories in his turn, then he would not receive a RISK card, which can later be used to gain more reinforcements. One should always, if possible, try to attack just a single territory in order to receive a RISK card. The *simple_attack_plan_list* is used for this purpose. These attack plans only attack a single territory and are only used if all other plans have been discarded and the AI has not received a RISK card. It is on the other hand not certain that these attack plans will ever be used. The *doAttack* might deem the attack plan impossible to use because of its probability of winning the attack.

There only seem to be very little limitations here. All other behavior is modularized as much as possible. The fact that attack plans are made from each occupied territory to every opponent territory expand the number of attack plans to the most possible. The only real limit here, is the whole idea with attack plans with priorities and cost. This might not be the best procedure for playing Risk at all. But this is the environment that has been designed and in which we will test the AI. Also, the prioritizing of goals and calculations of costs seem to be common in many games, which makes this design valid.

2.5 Assumptions in Previous Work

The framework and possible limits it imposed on the AI has now been discussed.

In [CJJ06] a lot of interesting discussions and conclusions were made on the different choices made throughout the project. These discussions will now be elaborated upon.

Training From Scripted vs. Training From Human Players

The most desirable choice on how to train an AI to play against human players, is in fact to generate training data from actual games against them. This would allow the AI to capture a lot of the nuances of a game as it is played by a human. This is however not possible in this project. A lot of training data is needed to train the learning AIs and we do not have the resources to gather this data with human players. Therefore we will use another approach. We have designed and implemented a scripted AI to take the place of human players and will use this AI to generate the training data used to train the learning AIs. The design was made based on an analysis of how we believe RISK should be played. Therefore the AI to some extent mimics the way we play RISK. Therefore we assume it is adequate for the basis of training the learning AIs. The AI was designed to at least defeat the AIs distributed with JRISK, in which it has succeeded. In 2 out of 3 games it defeated the "Hard AI".

Finding a Board Score In RISK

Another important conclusion made in the previous project was that it is not possible to find a usable way to determine how good a board state is in RISK. Good in the sense of how beneficial a player's armies are placed. This is not possible due to the contradictory tasks involved in RISK. You need to fulfill your own mission, but you have to prevent the others from fulfilling theirs as well. The first is important since it is required to win. The other is required in order not to lose. Therefore a board score should score high if it seems to fulfill both these terms. If one of them is less fulfilled, should they count equally? Is it more important to fulfill your own mission than preventing others from fulfilling theirs? These questions are hard to answer. Also a board score in RISK is very dependent on what you estimate the opponents' missions to be, since this will be needed to say whether or not you are preventing him from winning. This means that the better the IG for this task is, the better and more precise the board score will be. This will give the AIs implementing a good IG a huge advantage compared to the ones that does not whenever a board score is needed. A more in-depth explanation of why it is not possible to find a board score is found in [CJJ06] on page 61.

Training Methods

Not having a way to score the board rules out the usage of reinforcement learning in this project. Reinforcement learning is a technique used to train learning AIs [Mit97]. Roughly, learning is done by scoring the output action compared to all other possible actions. With a usable board score, it would be possible to compare actions by comparing the board score for all possible actions. But this is not possible without any measure for how good one action is compared to another. Instead we rely solely on supervised learning. In this technique an action is either right or wrong. There is no "degree" of how wrong it may be. A supervisor looks at the action and says whether it is wrong and gives the right result.

In this project the supervisor is, in most cases, the scripted AI. Since we have argued that it is a substitute for a human player, we believe it can be

trusted to tell the learning AIs what is right to do. The right decision is in most cases what the script itself would do. In fact, there is only one case where the script output is not used to train the AI. This is in the module "IG: Estimate Opponents' Missions". In this module, we actually have the precise mission available after the game has been played, and therefore we will use this when training instead of the script's estimate of the mission. This should improve the performance of the trained versions of this module since they will always be guided towards the correct mission, no matter how poorly the scripted version may perform.

However, for every other module we define the following: Every game won by the scripted AI, is an example positive behavior by the script. Every game the script lost is an example of negative behavior. In most cases the learning AI should only learn the positive behavior of the script. This is however not entirely true and is discussed further in Chapter 3.

2.6 Combining AI Techniques with Modules

As written in the previous section, we have chosen a broad selection of AI techniques to implement the different modules in the framework. It is not possible to implement all modules with all techniques, in most cases because the input or output could not be represented in the AI technique, or because the task of the module was too simple. A more detailed argumentation for this is given throughout Section 2.10 in [CJJ06]. A list of all modules and which AI techniques that will implement them is given in Table 2.2. This table also includes the modules not present in [CJJ06]. In the following, we will do a brief presentation of each AI technique used in this project.

	Script	Random	Decision Tree	Neural Net.	Bayesian Net.	Naive B. C.
Initial army placement	x	x	x	x		x
Estimate opponents' missions	x	x	x	x	x	x
How close an opponent is to winning	x	x	x	x		x
The opponents' next moves	x	x				
How close an opponent is to owning a continent	x	x	x	x	x	x
Continent ownership	x					
MP Goal Weighting	x	x		x		
Make attack plan	x	x				
Calculate attack plan cost	x	x				
Prioritize attack plan	x	x	x	x	x	x
Score attack plan	x	x	x	x		x
Discard attack plan	x	x	x	x		x
Remove goals	x					
Calculate defense cost	x	x	x	x	x	x
Score merged plan	x	x	x	x		x
Prioritize territory needing defense	x	x	x	x		x
Cash cards	x	x	x	x		x
Place armies	x	x	x	x		x
Transfer armies	x	x		x		

Table 2.2: *Each of the modules in the framework has been combined with the AI techniques. First the Initial Army Placement module, then each of the Information Givers, then the Master Prioritizer and lastly the modules in the Round Planner.*

2.7 Bayesian Network Redesign

In [CJJ06] we presented some Bayesian networks for use in some of the modules. These relied on the concept of time slicing to let them model changes over time. We will now show another way of modeling the modules without the use of time slicing, since this will give a large reduction of the number of nodes.

2.7.1 IG: Close to Continent

The Bayesian network for this IG is no longer modeled as a time sliced BN. The time sliced version was chosen to specify what changes that may occur over time. In the original BN which is seen in Figure 2.14, the thing changing over time was the the number of armies placed within a specific continent and the territories surrounding it.

This information could however be entered into a BN without the use of time slicing. The new BN is seen in Figure 2.15.

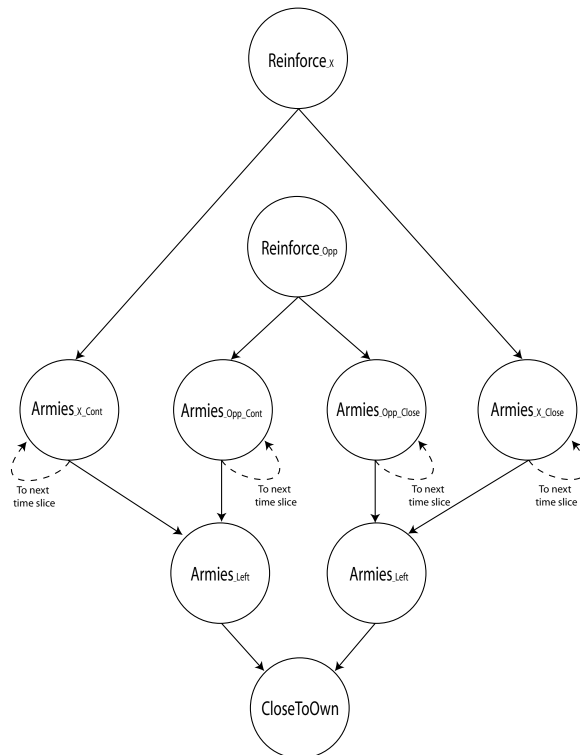

Figure 2.14: *The previous design of the BN "IG:Close To Continent"*

Instead of using the nodes "Reinforce_x" and "Reinfoce_opp" to specify how many reinforcements a player and his opponents will receive, this information will be entered directly into the nodes stating the number of armies a player has within and outside a continent. This information can be found using a function that estimates the number of reinforcements available in a given round from how the board looks right now. It is possible to use the same func-

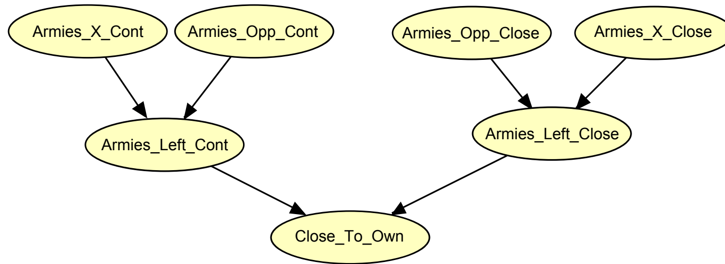

Figure 2.15: *The new design of the BN "IG:Close To Continent"*

tion in a gaming situation, and thereby instantiating the network with respect to the round it is estimating for.

2.7.2 IG: Close to Winning

In [CJJ06] we argued that this IG should be modeled as a time sliced BN since all inputs to the "Close to win" node, such as the "Close to continent" IG, were themselves time sliced BNs. However, since this is not the case any longer, this IG will not be time sliced either. The previous version of the BN is seen in Figure 2.16 and the new one in Figure 2.17.

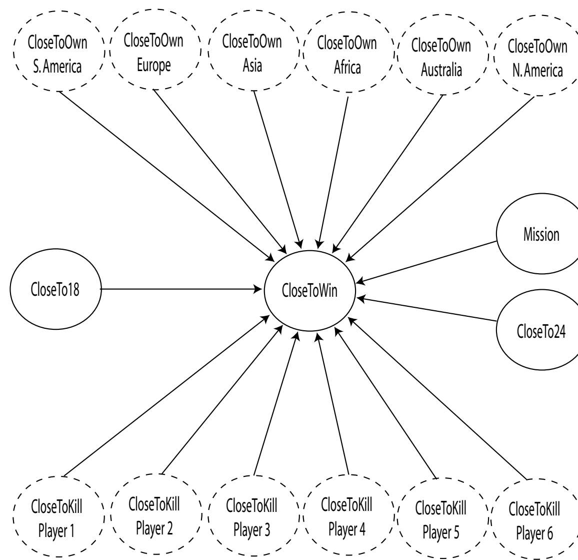

Figure 2.16: *The previous design of the BN "IG:Close To Winning"*

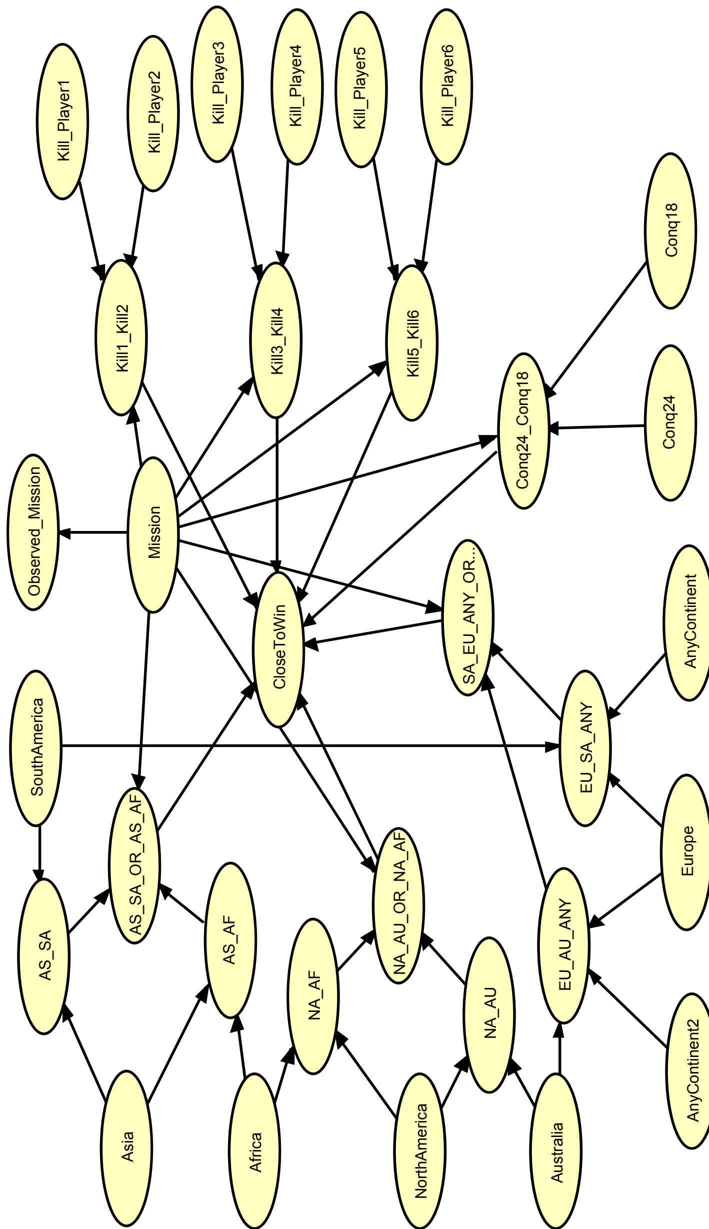

Figure 2.17: *The new design of the BN "IG:Close To Winning"*

As seen in the new network, a lot of separator nodes have been introduced to lower the number of states to the target node by divorcing its parents. The first separators are the ones determining how close an opponent is to fulfill a given mission. This is for example the node "AS_AF". This node is child of "Asia" and "Africa". When the game is running and we want to use the network to find out how close an player is to winning, these nodes are instantiated to how close he is to conquer each of the continents. "AS_AF" then gives information on whether or not a player is close to fulfilling the mission given evidence of each parent node. This scheme is repeated for each of the "Conquer Continent X and Y" missions. The "AnyContinent" nodes are used to give input for how close a player is to fulfilling a mission that includes conquering any third continent but the two required by the mission.

Each pair of nodes stating whether or not a player is close to fulfilling a mission are parents to another node. The task of this node is simply to lower the number of inputs to the "Close To Win" node. They have two states specifying whether or not one of the missions is close to being fulfilled. The "Kill_Player X", the "Conquer24" and "Conquer18" are "simple" missions, meaning they are not made up from composite goals like the "Conquer Continent X and Y". Pairwise they are parents of nodes choosing whether or not a player is close to fulfilling one of them.

Common for all the nodes stating whether or not a player is close to fulfilling one of two missions is that they also have the "Mission" node as a parent. This is again to limit the number of states on the "Close To Win" node. By this construction, the network should learn that the player is only close to fulfilling one of the two missions if the player has the mission. The "Mission" node is also parent to the "Observed Mission" node. Since, when training, we have both the estimate of a mission given by the script and the real mission available, it is possible to have this construction. Through this, the BN will learn how trustworthy the script's estimates on missions are. The "Observed Mission" node is the one that will be instantiated while playing the game. Both it and the "Mission" node has one state for each possible mission in the game.

Chapter 3

Training

In order to make it easier to train the models for each AI technique, it will be convenient to construct a framework that can handle the task of reading information from the game history and train the AI techniques on the basis of this. Such a learning framework can be seen in Figure 3.1.

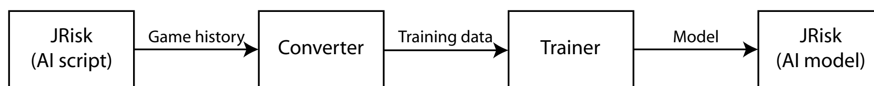

Figure 3.1: *An overview of the learning framework.*

The framework consists of four parts. The first part is the Risk game with our scripted AI. The second part is the training data converter which is responsible for converting the game history from the games played by the scripted AI into a series of training examples. The next part is the Trainer which uses these training examples to train a model based on a learning AI technique which can then be used in the Risk game (the final part in the figure).

This chapter will follow the sequence depicted in the figure and describe the details involved in each of the framework's tasks. It will start with a short introduction of each task, which then will be described in details.

3.1 The Scripted AI

As written in [CJJ06] Section 2.9, the scripted AI should generate training data for the learning AI techniques. To get more varied game play, the "hard AI" and "Easy AI" (which was already present in JRisk) should also participate in the games played. However, they only participate through their play against the scripted AI, whose actions in return are stored. The games should also consist of a random amount of players (between 3 and 6 players) — again for the sake of varied game play.

The composition of games will then result in at least one scripted AI, and then between 2 and 5 more random players ("hard AI", "easy AI" or the scripted

AI). This means that a game in theory can consist of 6 scripted AIs. Only the data from the scripted AI will be a part of the game history.

The scripted AI's history of played games needs to be stored in a format that allows it to be easily read by the Converter. The file format for the saved game history can be seen in [CJJ06], Section 3.3. It will now be discussed what and when the data from each module is saved and why it is done that way.

3.1.1 What Should Be Stored?

In order to train a given module, the input and output of the module needs to be stored, so that a learning algorithm can learn what to produce given a certain input. This is common to all modules. It might now seem straight-forward how to save training data for a module: every time the module is run, the input and output is saved as a training example. This is by far the easiest procedure and will produce many training examples from just a single game. But the modules in our framework depend highly on each other, the output from one module is the input to another, which result in some problems. An example is the "attack plan priority" module. This module takes an attack plan as input and produces a priority. But later in the sequence of modules, this attack plan with cost and priority might be discarded because of a high cost. It can therefore not be concluded whether or not the output from that particular run of the "priority" module helped in winning the game, since its associated attack plan is never used. So it is in fact useless as a training example.

Therefore, only the priorities of used attack plans are stored as training samples. This is also a problem in other modules, which will be explained in detail below.

Information Givers

All the output produced by the Information Givers are used in the Master Prioritizer, which is what the Round Planner base all its decisions upon. This means that every time an IG is run, it is used in the whole decision making process of the AI. Generating training data for an IG is therefore straight-forward: when an IG is run and produces an output, the input and output for that particular run is stored as a training example for later use.

Master Prioritizer

The training data for the MP is stored in the same manner as for the IGs. Arguments for this are the same as for the IGs.

Priority, Cost and Discard for Attack Plans

Training data for the modules for calculating priority and cost for a attack plan is stored as written in the introduction to this section. It can not be determined whether a priority and cost for an unused attack plan helped win the game or not, therefore training data for priority and cost is only stored for attack plans that are actually used. The same goes for the "Discard attack plan" module: Determining whether or not the discarding of a given attack plan helped to AI in winning or not can not be done.

Priority and Cost for Defense Plans

Defense plans are mostly used when placing reinforcement armies in the beginning of a round. But only the plans with the highest priority are used, which obviously means that some plans are not used. This also means that training data for the priority and cost for defense plan modules is only generated from defense plans that are actually used - in fact the same procedure as above.

Cash Cards, Place Armies, Fortify and Initial Placement

These modules directly produce an action in the game, meaning that their output is always used. The procedure for storing training data for these modules is therefore the same as for the IGs: every time a given module is run, the input and output is stored as a training example for that particular module.

3.2 The Converter

How the convertion of game history into training examples should be done, depends on both the module and the AI technique. Each module has defined what input it takes and what output it gives, and it differs from module to module what kind of data this is. Also, each AI technique requires to have the input represented in each their way.

The Converter will go through the game history and for each set of input and output data, it will create a training example suitable for a specific AI technique. These training examples can then be written to a file and then be used by the Trainer to train the AI model.

3.2.1 Making Continuous Values Discrete

The game history could for example consist of integer or float values. The number of possible values for such data is then potentially infinite, but not all AI techniques can handle this. Their input has to be discrete and only belong to a limited number of states. The problem is: How do we transform an input with a potentially infinite number of states into an input with a few states without losing information?

Dividing the Infinite Input Into a Few States

One way to do it, is to make an estimate on the distribution of the input values. If for example the input is the number of armies in a territory, the minimum value according to the rules of Risk is 1. The number of armies could potentially be infinite but a quick estimate would be that a useful maximum value could be 100 armies. If the number of armies is greater than 100, then the input would be mapped to 100. This gives us 100 states, one for each number between 1 and 100, which is still a large number of states for e.g. a BN to handle. This state space can then be further divided, for example with a state for each ten armies, resulting in ten states total, which is acceptable.

The aforementioned method assumes the input values are evenly distributed between 1 and 100, but this will almost never be the case in a Risk game. Most

territories will have only a handful armies in them, the number seldomly exceeds 10 and a territory with 100 armies will almost never be observed. Instead it would be wise to make the division of states reflect how frequently the values are observed. The intervals should be smaller around often observed values where more precision is needed. Some information will evidently be lost when mapping an infinite domain into a limited domain, since this corresponds to rounding off a float value to an integer value. But doing it this way the states will fit the observed values and minimize the information loss.

To determine the distribution of an input variable one could simply make the computer play a large number of games and count how many times each value of the input variable is observed. This can be done directly by going through the game history generated by the scripted AI, so no extra playing is needed.

From a Few States to Infinite Output

Another problem occurs when the output from an AI model with few states has to be used in the game where the state space is potentially unlimited. One state in the AI model may cover a range of values in the game. So here the problem is the opposite: How do we reconstruct the real value on basis of a value belonging to a simplified state space?

Since each state in the AI model covers a range of real values, a solution to this problem would be to take the real value that lies in the middle of this range. This way the information loss is minimized, since this gives the value which on average is closest to the correct one.

3.2.2 Constructing the Training Data

Each player in a game generates training data, but the learning AI techniques should be guided towards the correct behavior, meaining that it should learn to win the game. This would indicate that the only information that should be trained from is data stored for players that have won a game. This approach has two major downsides. The first is that a lot of data would be omitted. If a game has six players, there will only be one winner. This means that data for five players would be thrown away, and this is hardly efficient. The second and maybe more serious downside of this approach is that if the AI only learns from the winning scripts, they will actually never improve beyond the scripts. They will at best mimic the scripts, but will never become better than them[1].

This is in fact the whole downside to this kind of supervised learning implemented in this project. The AI can at best become as good as the scripts. At no point does it evaluate its own performance and try to become better[2]. But it has already been argued, that the only way the AI can learn anything, is to say that if a game is won everything it did was perfect, if not then everything was wrong.

[1]This does however not include the Mission Information Giver, since in learning, it directly has access to which mission a player actually had and it therefore does not try to mimic the script.

[2]This could have been solved by having human players create training data, but still, the AI would at best become as good as the best of the players.

If on the other hand, reinforcement learning was possible, where each action is given a score of how well that action performed, it could more easily be spotted which actions were better than others. One could then infer that performing these good actions would have a better chance of leading to victory. But as argued in Section 2.5 on page 33, it is impossible to calculate whether or not a given action will result in a better situation or not. Therefore, the only way to evolve beyond the training data is by doing something other than what the training data dictates. But the actions of the winners have already been deemed perfect, so only the actions of the losers can be used in trying to evolve beyond the scripts.

Therefore, a way to learn from the players who lost the game might be useful.

3.2.3 Two Approaches for Learning from Losers

There are two different ways an AI technique can learn from losers. Which of the two approaches is used depends on the technique.

First of all, the logic explanation to why a player is losing is that the behavior of the losing player is wrong. So a given AI should learn not to do as the loser. And here lies the problem of learning from losers: Finding the correct behavior (or at least better behavior), when only incorrect behavior is present.

The two approaches in finding this correct behavior are the following:

Approach 1: Everything, but incorrect behavior. The correct behavior lies in the domain of "everything, but the incorrect behavior", so doing everything else will over time yield the correct behavior. But "everything else" is quite a large search space, and in some AI techniques this is impossible.

Approach 2: The opposite behavior. Doing the opposite of the incorrect behavior might be a better solution. This narrows the search space, but determining the opposite of a given behavior is in some cases impossible. Also, doing the opposite of some incorrect behavior might be even worse.

The up- and downsides of the two approaches will now be analyzed and after that it will be decided which approach each AI technique should use.

Everything, but the Incorrect Behavior

A way to turn a losing AI's output into training data is by assuming that every thing it has done in the game is wrong. If for example we have three possible actions an AI can perform (A, B, C), and the losing AI performed action A, our assumption says that both B and C are correct actions that would have led the AI to victory. This may not always be true, but we believe that if enough games are played, the training data will be varied enough for the learning AIs to discover, what the best action is.

If, in our small example, losers often have chosen action A, we believe that the good action is either B or C. Later in the training data we discover that there are also a lot of examples where the winner has chosen action B. Since losers has chosen A and winners has chosen B, our AI will learn that B is in fact the

best choice in a given situation. This small example shows that appart from the assumption that every thing the loser does is wrong, we also assume that every thing the winner does is correct. Without this second assumption, we would not have any way of pointing the trained AI in the right direction when given, in the example, two right choices whenever loser's data is used.

A thing that need to be handled is how much loser data to use. In a game of RISK, there are possibly six players. Since there is only one winner, it means there are five losers. So there will be five times more losers than winners. The problem is that the amount of winner data is then five times smaller than loser data. The result of this is that winner data has five times less influence on the training than the loser data. This is a problem since the winner data is from an AI that has succeeded whereas the loser data is not. Therefore, winner data should at least has an equal amount of influence on the training of the AI as loser data.

There are basically two ways of fixing this problem. The first way is by multiplying the amount of winner data by the number of losers. This will ensure that the winner data has as much influence on the training as loser data. The downside of this is that the amount of examples in the training data will be multiplied with the number of players. In practice, this is not feasible if the number of examples is very high.

The other way is to only use data from one loser. A random of the five losers is then selected. Even though the number of examples will be fewer, this is a more applicable solution. From our assumption that everything the loser did was wrong, we get that using one or five losers to train from does not matter. We only care about whether they succeeded or failed in their task. This approach will keep the number of training examples down compared to the other approach, which could be a plus if the number of training examples is too high for a learning technique to handle.

The Opposite Behavior

Doing the opposite of some action or behavior may result in better behavior than the behavior of a losing player. But it might also result in even worse behavior. Which of the two statements holds in practice is definitely worth looking into. But in some cases it is not possible to find the opposite of some action, but if it *is* possible, then it is useful for limiting the search space of possible actions.

3.2.4 Choosing an Approach

With the two approaches' pros and cons in mind, it can now be decided for each AI technique which approach to use.

Neural Networks

The behavior of a neural network can be viewed in the output vector of the network, and the behavior of network can be corrected by changing the resulting error vector. It would not make sense here to follow the first approach here, since "doing everything else" would in fact mean that the network should be

trained with every other error vector possible, which is impossible. The set of error vectors could be limited into ranges, but instead it was chosen to follow the second approach, since the number of error vectors to train from would be very large - even if it is limited into ranges. Following the second approach is in fact possible to do for neural networks. Learning to do the opposite of a given behavior of a neural network is simply a matter of negating the error vector. Whether or not this will yield to better behavior over time, will only show through the results of the testing.

Bayesian Networks

In Bayesian networks the behavior of a player in a game would in all cases correspond to one or more nodes being in certain states. This means that doing "everything else" corresponds to the nodes being in all other states but the given state.

A way of implementing this in Bayesian Networks is to distribute the loser example for a given node over the number of possible states of that node, except the actual state of the loser example. This can be done directly in the training data. When encountering a loser example, make $s - 1$ examples of the node being in every other state than the actual example, where s is the number of states the given node has. To ensure fairness for the winner examples, each time a winner example is encountered, it is copied $s - 1$ times.

For example, if the Bayesian network for an Information Giver has a node determining the output estimate of the IG with, for instance, 20 states. A loser example for this node would then consist of 19 example of the node being in every other state, but the actual state. A winner example would consist of 19 identical examples of the node being in the actual state.

In Bayesian networks it does not make sense to find the "opposite behavior", since it is unknown what the opposite state of a given state is.

Decision Trees

This is in fact done in the same manner as for Bayesian networks. The training examples for losers consist of examples of the target attribute being in every other state but the actual state.

Naive Bayes Classifiers

Naive Bayes Classifiers corresponds roughly to Bayesian networks and the same approach can be used.

3.2.5 Training with Winners Only

As already stated, the theory that losers can be used in evolving the learning AI beyond the training data, is only a theory. Actual tests will show whether or not it holds. It would therefore be a good idea to let the learning AIs also train from winners only. There is in our opinion a small chance that the AIs would become better than the scripts. The reason for this, is that the behavior

of winners must be a small subset of the full behavior of the scripts. If the AIs adapt only the behavior of the winner, they might in most cases be better than the more general script. This is illustrated in Figure 3.2.

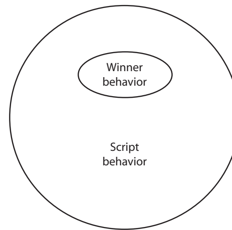

Figure 3.2: *The behavior of the winning scripts is a subset of the behavior of all scripts. If an AI only learns the behavior of the winning scripts, it will most likely be better than the general script.*

The pitfalls of this hypothesis, is whether or not one can say that the winning script's behavior is a subset of the general behavior of the script, since the scripts generally are deterministic. In some cases though, the scripts take random actions, when a number of actions seem equally valuable. However, we argue that if a script is fully deterministic and an AI model is trained with only the actions the script did when it won, then a fully trained AI model will not be any different from the script. If the trained model and the script are given the same situation they will react the same. Therefore an AI that learns from a deterministic script will not be any better than the script.

If on the other hand, the scripts are non-deterministic (even if it is only in isolated situations), then the winners can have reacted differently than the losers in the same situation. We then say that the winning behavior is a subset of the script's behavior, which thereby gives a learning AI the chance to learn the winning behavior and become better than the losers.

3.2.6 Class Diagram

The class diagram for the converter can be seen in Figure 3.3. Each box represents a class while the arrows shows their relationship, either an aggregation or a generalization as defined in [Lar00].

The responsibility of the different classes are:

TrainingData: This class is the main class in the converter. It loads and parses all game history files, which are in XML format. It fills the read data into the corresponding data structures.

XMLInput: Holds the information on the current game history file in process.

GameData: This is a container class for all types of data structures, such as all types of module outputs (cost estimates, attack plans etc.) and other data structures read from the game history.

CostAttribute, GoalDistributionAttribute, etc.: All of the different data structures. These classes holds the information read from the game history and has methods for converting the data to neural network training data,

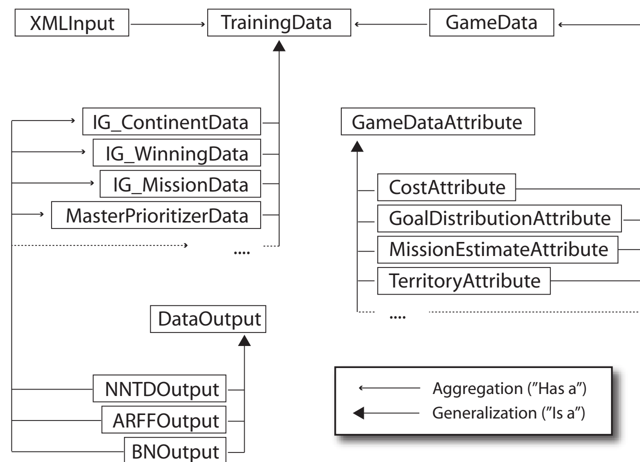

Figure 3.3: *The class diagram for the training data converter.*

Weka training data (ARFF file format), and training data for Bayesian networks.

GameDataAttribute: The super class for the data structures. It holds common methods for those classes.

IG_ContinentData, IG_WinningData, etc.: These classes are responsible for converting the read game history to training data, using the above mentioned data structures. There is a single class for each module, because the game history used varies from module to module — the game history used by a module corresponds to the input and output of that module.

NNTDOutput, ARFFOutput, and BNOutput: Each of these classes are responsible for outputting training data in a suitable data format.

DataOutput: The super class for the above classes. It holds common methods for those classes.

The training data is output in different file formats depending on the AI technique. These file formats are described in Appendix D.

3.3 The Trainer

Using the training examples generated by the converter, this part of the framework is responsible for training the models for each AI technique. The trainer will then output a trained model that can be used from within JRisk as a replacement for the scripted AI modules. In the JRisk module, an input suited for the given AI model can be constructed from the input given to module from the game itself. Using this input, the AI model can be queried and an output value can be retrieved.

The idea is to construct a "training example" from the module input which the AI model then classifies. Therefore the input need to be converted into

a format readable by the AI model. Forunately this input convertion corresponds to the convertion that was done by the Converter when transforming game history into training examples, so this can simply be reused. Also, the resulting output from the AI model may need to be converted into a format understandable by JRisk. This convertion maps the values from the possibly limited state space of the AI model to the possibly unlimited state space of the JRisk module output.

3.3.1 Training Tools

Instead of implementing all the different AI techniques ourselves, we looked at a number of existing implementations to see if they could suit our needs.

The first implementation we looked at was the Weka Machine Learning Project (Weka) from [WF05]. Weka is a collection of machine learning algorithms for solving datamining problems. It provides a single interface across all machine learning techniques, which makes it easy to switch one technique with another without affecting the rest of the system. It is open source and implemented in Java so it will be easy to use from within JRisk, plus we can modify it to suit our requirements if needed. The techniques in Weka include the four we have chosen for this project: Neural networks, Bayesian networks, naive Bayes classifier, and decision trees.

The neural network implementation is however restricted by Weka's flexibility. In order to have the same interface to all techniques, the output from a model is restricted to have only one output variable. This makes it unusable to us, since most of our neural networks have more than one output node. Therefore we need to make our own implementation of neural networks.

For Bayesian networks there already exists the commercial Hugin [AOJJ89] application, which is specialized in dealing with Bayesian networks. One could therefore assume that Hugin possess the most efficient techniques for Bayesian networks. And since Hugin also has graphical modeling tools for manipulating with the networks, and also because the Hugin decision engine can be used directly from our Java program, it will be our choice of a Bayesian network implementation.

Weka's implementation of the two remaining techniques, naive Bayes classifier and decision trees, suit our needs and therefore Weka was chosen for these.

Naive Bayes Classifier

In Weka, the naive Bayes classifier is implemented as described in [DHS73] which follows the procedure previously mentioned in Section 2.2.5. It suited our needs and needed no modifications to be used in this project.

Decision Tree

The decision tree classifier in Weka is an implementation of the ID3 algorithm as described in [Qui86]. It does not support training data with missing values but since all our training data is complete this will not be an issue. A thing that *is* an issue has to do with classifying unseen instances:

When the decision tree has been trained it can be used to classify instances. Starting at the root node it checks the value of the corresponding instance attribute to see which of its children it should visit next. At the chosen child node this check is repeated with its corresponding attribute; this will be repeated until a leaf node is reached. The value of the leaf node will then be the classification of the instance. But what happens if we try to classify an instance where, following the procedure just described, there does not exist a path from the root node to a leaf? Then the decision tree is unable to classify the instance!

It is suggested in [Qui86] that in a case with an instance where the path ends in a NULL leaf, it would be better if the decision tree assigned the instance the most frequent classification instead of failing. However, in Weka this behavior is not implemented: the decision tree classifier will simply fail when reaching a NULL leaf. This needs to be fixed before Weka can be used for building decision trees with the ID3 algorithm. Because Weka is open source it is possible for us to extend the existing classifier and implement this behavior.

The definition of "the most frequent classification" which should be assigned to the instance is not clear. But it can be done by assigning the classification that was most often given to training examples when the decision tree was trained. This is under the assumption that the training examples reflect the real world domain. This could be the leaf value that appear most often in the decision tree as a whole, but this would not be optimal since the distribution of leaf values for the tree as a whole not necessarily matches the distribution of leaf values for some smaller branch of the tree. It would be better to use the distribution of leaf values for that particular branch and then assign the classification that appears most often in this. This will give the most probable classification according to the instance's other attributes that have defined the path down to the NULL leaf.

Neural Networks

As written in the introduction to this section, the implementation of neural networks in Weka was unsatisfactory. So we implemented neural networks ourselves. The implementation includes data structures for the neural network architecture, and an implementation of the back propagation algorithm (BPA) based on [Mit97] which is used in the training of neural networks. It was chosen because it is well documented and widely used.

Bayesian Networks

For Bayesian networks we used the Hugin Researcher application for modeling and training the networks. Inside the game we used the Hugin Decision Engine library for inference. It uses a propagation algorithm called Hugin Propagation proposed by [JLO90]. This algorithm is based on the algorithm previously described in Section 2.2.4.

3.3.2 Training Data Set Size

When it is time to train the different learning AI techniques, it must be determined how much training data is needed.

The two contradictory factors in this is the following:

- There should be a sufficient amount of data, so the learning algorithms have an opportunity to become adequately trained.
- But the amount of data should not become so large, that it will take forever for the algorithms to learn anything. Also, the storage capacity available sets a natural limit.

The amount of data might also be highly dependent on the different AI techniques. But through the analysis of the time complexity of each of the AI techniques used, no exponential time complexity was found with respect to the amount of training data. So it must be concluded that the amount of training data used does not affect the AI techniques differently. The amount of data will therefore be the same for each AI technique and make them more easily comparable.

Finding the right amount of data is not an easy task, as it also is highly dependent on the available computing power. A goal that we feel would be adequate is a couple of thousand played games. This will hopefully result in many thousand module examples. But the amount of available storage is 50 gigabytes, which might pose a limit.

3.3.3 Training Iterations

When learning algorithms are trained from training examples, it is not always enough to iterate through the examples once. Depending on the learning algorithm and the size of the training data, it is necessary to have different criteria for ending the training. These criteria are:

- Iterate until a fixed number of iterations have been reached.
- Iterate until the network converges to some acceptable state.
- Iterate a fixed number of times, test the learning algorithm's behavior and performance in its application, and iterate again until the performance is satisfactory.

Which criteria is selected depends on the AI technique.

Neural Networks

In many cases, neural networks are trained until the error in the network is sufficiently small, meaning that the network has converged. However, one must be aware that if the training data is iterated too many times, overfitting can occur, meaning that the network has trained to respond correctly to the training data only. The network should be trained to generalize over the training data, so that the network also responds correctly to data that is from outside the training data set. This can be avoided by checking the generality of the network using a sample of the training data as described in [Mit97]. This sample is not used in training, but only used to check if the network responds correctly

to data it has not been trained with.

However, the plan to have the network converge to some sufficiently small error does not work when the network is trained using both winners and losers. The training data will therefore consist of data from both winners and losers. When learning from a winner example, the network will use the error vector to update the weights in the network. But when learning from a loser example, the error vector will be negated. So the general error of the network, when the training data has been iterated through, will most likely not decrease, as it would when normal training is used. Depending on the winner-loser ratio, the error can decrease and increase, but will most likely remain the same. The network is not trained to respond to the generality of the training data, it is actually trained to act outside the training data. And therefore having the network converge to sufficiently small error does not make sense.

Determining whether or not the behavior of the network is acceptable can only be done by implementing the network in a game and test the behavior emperical. If the network behaves poorly, another iteration through the training data might be necessary. This approach however, can hardly be done automatically and might, depending on hardware availablity, take a great amount of time to get acceptable results. So it seems that the only practical solution here to is have a fixed number of iterations when training neural networks. When learning from winners only we will also use the fixed iterations approach, although it would have be possible to use convergence as a criteria. We do this to make the results comparable with the network trained with winners and losers, because they will then be trained for the same number of iterations.

Bayesian Networks

When training the Bayisian Networks using the EM algorithm in Hugin, it is possible to specify a number of iterations for the algorithm to go through the training data or specify the convergence limit. The convergence limit specifies how small the changes in the probabilities when propagating should be before the network is evaluated to be "fully trained".

There are a few things to consider when deciding whether to use one over the other. The first thing concerns the training data. If the set of training examples is large, and there a many entries where data is unavailable, it could be unwise to let the network train until it converges. This would simply take too long depending on the number of nodes with unknown values. On the other hand, if there is no missing or unknown data in the training data set, the EM algorithm simply counts observations of each state in each node, making it no problem to let it converge.

The second thing to consider is the size of the network. If the network is large, it will take much longer for the EM algorithm to update all nodes than in a smaller network.

Specifically for the networks used in this project, we will let the "IG: Close to continent" be trained until it converges. The reason for this is that we have training data available for each node in the network. Therefore this should

pose no problem. The network should be able to perform at least as well as the scripted AI when fully trained.

The “IG: Close to winning” network has a lot more nodes than the “IG: Close to continent”. There is not data available for a lot of the nodes in the training data. This means that the EM algorithm will take much longer when going through the training data. Through testing we have found that it takes roughly 24 hours for the EM algorithm to go through 1.5 million training examples one time. Therefore the number of iterations this network will run will be two. Whether or not this is sufficient for the network to perform decently will be shown through testing. However, the training data is made from nearly 800.000 board states, so the network may still be able to adapt some rational behavior from these states.

Chapter 4

Testing

There are numerous things to test when deciding whether or not a specific AI technique applies to solve a given task from our framework. We divide the testing into hard and soft values. The hard values are those that can be measured directly by analyzing software behavior and thus are objective values, whereas the soft ones are harder to measure since they are subjective to each player. We will first cover the hard values and then continue to the soft values. What we are interested in is finding the AI composition that yields the most successful AI in the terms of won games which is also fun to play against.

4.1 Finding the Best Combination of Modules

The first step in finding an AI that gives good opposition is to find the combined AI that performs best. The modular design of our framework makes it possible to implement different AI techniques on each module. This means that it is possible to make a large number of AIs by combining these modules in various ways. The different techniques that can be used to implement specific modules is seen in Table 2.2 on Page 36.

We assume that our scripted AI is not the best performing AI, and by exchanging individual modules with trained AI techniques, the performance, opposition-wise, will be better.

There are a few different ways of finding the best performing AI from the set of possible AI compositions. These will be considered throughout this section.

4.1.1 Limiting The Number Of Games To Play

Without doubt, the best way of finding the best AI is letting all possible AI compositions play against each other until an overall winner has been established.This means having them all play against each other in games consisting of both tree, four, five, and six players. However, from Table 2.2, it is found that there are over one hundred million different AI combinations. This is found

by multiplying the number of AI techniques implementing each of the modules. The "Random" technique is not included in this calculation since this technique is not considered valid in this project. The high number of AI compositions highly limits the possibility of building all of them and playing them against each other to obtain the best AI. To resolve this problem, one method would be limiting the number of games each AI must play against each other to a low number. This could for example be ten games. This however, is not a very good idea since there is a certain amount of luck involved in the game and ten games will therefore not be enough to counter for this fact, and the results will therefore be useless since the lucky AIs may be deemed winners because they were lucky and not because they actually performed better. Therefore, the AI composition needs to be found in another way.

Tournament

One approach to limit the number of games to play is by having a tournament scheme as seen in Figure 4.1 where the winners of each group progress to the next step in the tournament. The tournament approach has the benefit that it is fairly easy to implement and that it will find the best overall AI in relatively short time since not all combinations of possible games are played. However, since not all possible combinations of games are played, some good AIs may be thrown away if placed in an unfortunate group. If an AI loses most games in one group, it is of course removed from the tournament. But this AI may in fact have been the best if it had been put in another group, from which it would have been passed on to the next round of the tournament. One may argue that this means nothing since it will then lose whenever it is meets the one that could defeat it in group four, thus in the end, the result would be the same. But this is not true. If the overall best AI is placed in a group with the ones that can defeat precisely it, but loses to everyone else, and the specific AI can defeat everyone else, this AI is removed and will never reach round two where it would defeat everyone. This is a flaw in the tournament approach. The benefit of the tournament approach is the not all AI compositions needs to play each other to find a winner. But still it is not useful in this project since it still requires a very high number of games to be played. Especially in the first round of the tournament which includes all possible AI compositions. This means that the tournament method will not be considered a viable way of finding the best AI in this project.

Fixed Size Game

Another approach to limiting the number of games played is by playing only games of a certain number of players instead of playing games with both three, four, five, and six players. Playing all AIs against each other in fixed sized games requires some careful planning. A thing to consider and keep track of is how many games the different AIs have played against each other. This is very important since no AI should have the benefit of playing more games than others. The reason for this requirement is that this would give the AI a possible advantage of winning more games than an other AI that might be equally good, but has not been allowed to play as many games. It may not be

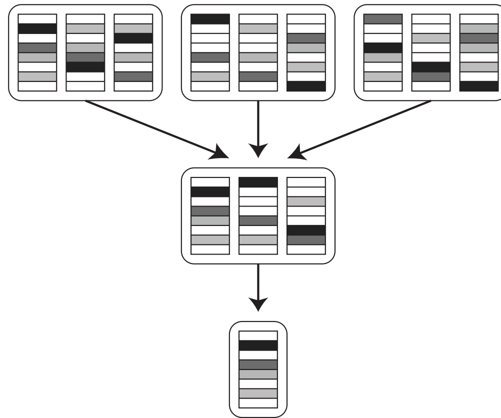

Figure 4.1: *In the tournament the winning AI from each group is allowed to play in the next step of the tournament until a winner is found. The different colors within the AIs represent the different modules which is implemented by different AI techniques.*

possible to let all AIs play a precisely equal number of games. This is depends on the number of different AIs.

The downside of using fixed size games is that the AIs have been trained using a random number of players (from three to six players). Therefore they should also be evaluated playing under this circumstance since there may be an AI that performs better against three players than it does against six players. This is overcome by first letting all AIs play against each other an equal number of times in a game with three players. Next they get to play each other an equal number times in a game with four players and so forth up to six players. This will ensure that AIs get to play in every possible environment, which is what is needed to analyze whether or not it is versatile enough to qualify as the best performing AI.

Random Sized Game

Instead of having all AIs play each other in one fixed sized games, it is possible to let them play in random sized games. The main thing to consider in this approach is still how to make sure that all players have played each other an equal amount of times. This method will reveal the best performing AI over a number of games with a random number of players and will, over time, reveal the best performing versatile AI. However, there is still need of quite a few examples of the AIs playing in both three, four, five, and six player games to determine whether or not it is better than the other AIs. Therefore, even a testing scheme based on using random sized game will be too imposible for this project.

4.1.2 Limiting the Number of AIs

At this time it should be clear that the previous atempts to limit how many games that needs to be played has failed because they all involved playing all possible AI compositions against each other. Therefore we will now take some

other approaches for isolating the best performing AI composition, without having to go through all possible AI compositions.

Stepwise Evolution

One approach to limit the number of AI compositions to test is to do a "stepwise evolution" of the AI. This means gradually evolving an AI from an initial composition towards the best possible AI composition one module at the time. This procedure works are follows:

All modules in the AI are initially using one AI technique e.g. scripting. This is the initial AI. Then one by one, the initial technique on the modules are replaced by the different techniques available for that module. These new AIs are played against each other to find a winning AI. This winning AI is the one that has won the most games from a series of games. The winning AI is the new initial AI, and the same steps are performed once again to find the best technique on the next module. The final result is an AI utilizing the best AI technique on each module. This should also be the best overall performing AI.

Figure 4.2 shows how this method evolves from the initial AI to the best performing AI. The leftmost AI configuration in each round is the initial AI. The example given in the figure is for a game with four players. This example needs to be generalized to work for games with three to six players. There needs to be some structure on how to choose which players to play in a game. Also, there should not be any players of the same type in a game since this may pose a problem. They are both equal, meaning that the probability of either of them losing or winning should also be equal. If any of them wins, they would naturally be chosen as the winner of the game and the "Winner count" for both would go one up no matter which of them was the actual winner. Also, their chance of winning are doubled since there are two of them in the game. The solution to this is naturally to take the average of all their wins, but this actually punishes the AI for having a "twin" in the game. So therefore twins are not allowed in the game. Also having a twin in the game would ensure that at least one of them will lose each game since there is only one winner.

Best Module Approach

This approach is similar to "Stepwise Evolution"(SE) in the sense that initially the AI is entirely scripted. The difference from SE is that even though the best technique for a module is been found in the first step, the technique is not used in the next step. Instead the initial AI remains the same and each module is exchanged with each usable AI technique. This is seen in Figure 4.3. After the best technique for each module has been found, they can be combined to one full AI composition. This procedure should produce the same result as SE, and may be used to check if the best AI is in fact found through SE. If the result of this approach differs from that found from SE, it may indicate that certain modules may not be very important to the complete AI since more techniques may be used to implement them. The major benefit with this approach over SE is that it is possible to run the best module approach in parallel. This is possible because each module is tested within a static initial AI. In SE each module is tested within an AI build from the best performing techniques already tested, meaning that testing needs to be done in some predetermined order.

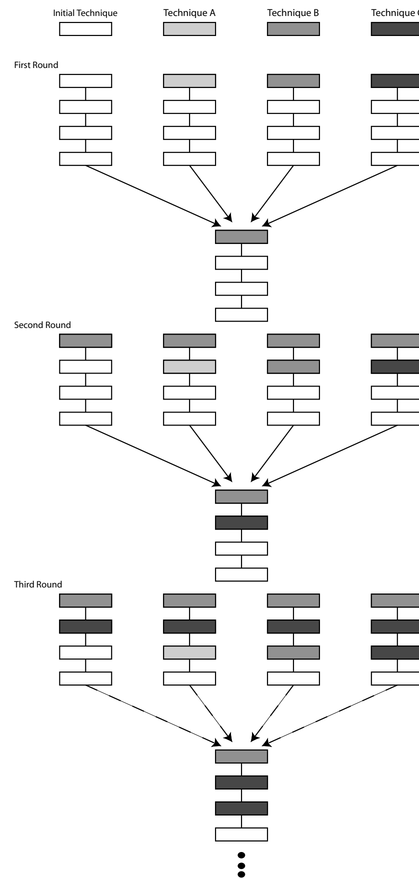

Figure 4.2: *Using Stepwise Evolution to find the best AI. Each step(round) shows the AIs participating in the game. The arrow points towards the winner. At each step the winning technique of the previously tested module is incorporated in the AI. Continually building the best performing AI as the modules are tested.*

4.2 Most Important Modules

A beneficial test to make is a test of which modules that are actually most important in our framework. This test is necessary when analyzing which AI techniques are best on a particular module. If the module is not really important to the AI composition, the success or failure of that AI may in fact not be subscribed to a specific technique.

A fairly simple approach to test which modules of the winning AI are most important is by exchanging each module one at a time with a "random" module. Meaning one that gives a random output. By testing the module importance this way we assume that "random" should not make many good decisions, and therefore an AI implementing the random at a module should not win many games. This test is rather important since it gives some guidelines to whether or not the success of the AI relates to a specific AI technique. If an AI is still winning if a given module has been replaced with the "random"

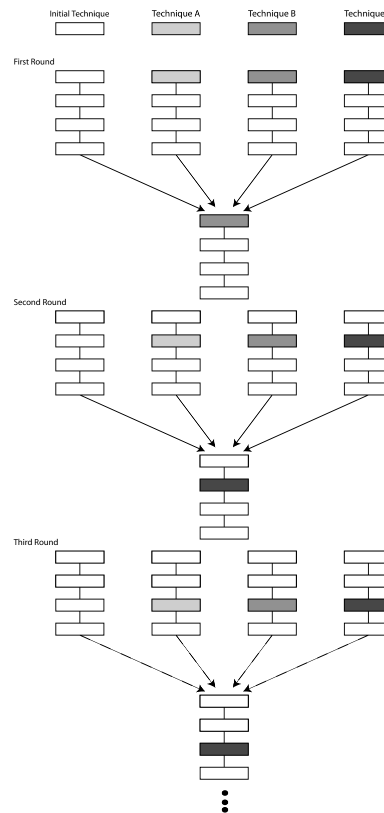

Figure 4.3: *Using Best Module Approach to find the best AI. Each step(round) shows the AIs participating in the game. The arrow points towards the winner. At each step the same initial AI composition is used. Only the module being tested has been exchanged with the various AI techniques. The winning techniques are memorized so the best performing AI may be build after all tests have been completed.*

module, it may mean that this module has none or little influence on the AI's performance.

4.3 Time Used In Each Module

This test is used to discover the relationship between the success of a technique and the average time it uses to deliver an output. This may turn out to be closely related to the task of discovering whether or not the AI is fun to play against since a player do not want to wait a very long time for the AIs' turns to pass.

Ideally this test should be done counting CPU cycles instead of system time. This would be to avoid the possibility of the operating system interrupting the game while we are in fact measuring the time within a module. Unfortunately it is not possible to count CPU cycles within a Java function and we have not been able to find a code profiler that could do so.

Since it is not possible to count CPU cycles, the time spent in each mod-

ule is measured within each module as “system time usage”. This is done by implementing a check for what the system time is as the AI starts running the module and another check on the system time as the module returns its output. With this method we of course have the problem with the OS interrupts, and we need to keep this in mind while analyzing the performance times of each technique. To somewhat counter for the “current work load” and different CPU speeds of the test systems on a system, we run a benchmark before a new game is started. In the benchmark we test how long it takes to take the square root of a random number 10.000 times. Using this benchmark is not perfect since the workload of a system may change while the game is being played, but since the game is run many times, it should even out the differences in workload.

Another minor problem is when timing some of our scripts that uses “out of script” methods. For instance we have “RP Planner: Make Attack Plan” which uses a path finding algorithm placed outside the Planner class. We also have the “IG: Opponents’ Next Moves” which runs the MP and RP from the opponent’s side of the game. The time these “outside” tasks will still be counted as time used in the script being times since the calling of these “outside” scripts are part of the times script’s design. If a learning AI technique should catch the behavior of the script it would also need to catch the behavior of the called script. This means that the trained technique on the module would be timed for doing the same amount of work as the script, including “outside” calls. So to keep things fair, work done outside the scripted module is counted as if it was done within the module itself.

4.4 Model Size

We will keep track of some information regarding the size (in MB) of the models we train. The size of the models affects the model load times and the amount of memory required to run the game. Higher memory requirements increases the risk of performing worse due to swapping between RAM and hard drive. Loading models means both loading from the physical disk and creating data structures to hold the model within the game. In this project we do not consider the actual reading from the user’s hard disk to RAM since this is not really dependent of which AI technique that is being used, but rather the speed of the user’s hard disk. The construction of the data structures however is dependent of which AI technique is being used. Therefore we are only interested in the time it takes to handle this.

4.5 User Experience

The user experience of the AI is the soft value we would like to test. The thing we are interested in is whether or not our best performing AI is fun to play against.

To determine if the best AI is fun to play against will require a broad array of users. These users must range from players with low experience in playing RISK to very experienced users. The reason for this range is that the AI may be fun to play against if you have low experience and dull if you have much.

One thing that influences how fun the AI is to play against is how hard it is to beat. It must neither be to hard, nor to easy to beat. This is of course dependent on the user's experience with RISK and therefore hard to handle without having a difficulty setting for the AI. So what we could get from this test would be a mark for how hard our AI is to beat by users of different experience levels. If the AI's mechanics are too simple to see through, it will become to simple to beat.

A thing that may influence the experience of the AI, apart from how hard it is to beat is how much time it uses to take its turn since no player wants to wait too long for the AI to finish its turn, since this basically is time where he is doing nothing but waiting. This time is influenced directly by the time each module uses, hence this is why the time used in each module is important.

There are basically two ways of testing the AI. The first is having a closed group of handpicked testers of different skill level that play the game and report their experiences back to us after each game. This method has the advantage that we have full control over the testers and are, if needed, available to assist them with any problems. The other method is a public test where the game is put on the Internet for players to download. The game should be distributed along with a form to fill out for feedback. This method has the advantage that the possible test group will be much broader and have greater variety. However, there is a risk that there may not be any feedback at all. If users find the game boring, they are most likely not to give feedback, since they already may feel that they have wasted enough of their time already. If they do in fact send feedback, it is most likely biased in a negative direction so they may not supply the objective feedback that we in fact need. Likewise are users who find the game fun biased in a positive direction. They may have problems giving feedback on the things that are not so good in the game. So in fact, the best method for having users tests is by doing it in a closed environment where we, as developers, are more in control.

4.6 Miscellaneous Recordings

There are a few other things we will record while the AIs play against each other. These are some useful results that may benefit us while analyzing the results. One of the things we record is what missions the different players have. This may be useful to examine whether or not an AI technique performs better if it has specific missions in the game. Also the number of rounds a game lasts is recorded. We have observed that often in games where the first player has the "Conquer 24 Territories" mission, the player wins within his first turn. Whether this observation is true or not can be seen through the test results.

Another way of obtaining this result would be by having an AI play against the other AIs with each mission a fixed number of games. This can be done by making sure an AI is given the required mission in a predetermined number of games. We will not be doing this in this project since it will require a lot of work for too little gain. We believe that over time, this tendency will still shine through by just recording what mission the winner has.

We will also be recording which missions the techniques have when they lose a game. This should indicate which missions the techniques may not be good a fulfilling.

4.7 Number of Games Played

To obtain a clear indication of which AIs that are superior to others requires a large number of games to be played. It is hard to estimate the number of required games, but while tweaking the scripted version of the AI towards defeating the AIs incorporated in JRISK, we found that we need around 500 games to have a clear indication of which AI is superior to the other.

4.8 Summary of the Tests

Before proceeding to describing in detail how the test will be performed, a small summary of what exactly we will be testing and what requirements there is to the tests, may be in its place. We will find the best AI composition through the mentioned "Best Module" approach. The choice was between this and "Stepwise Evolution" approach due to the fact that the number of possible AI compositions are to high for us to do a complete test utilizing all possible AIs. Since we assume that both "Best Module" and "Stepwise Evolution" produce the same AI, either one of them would be useful for this project. However, since "Best Module" is able to run in parallel, this is the chosen method for this project. The data used to find the best performing AI is:

- Number of games won by each technique within games of three, four, five, and six players.
- The number of games that each technique must play against the others is 500.
- The time used by each technique in the modules they implement.
- The time it takes to initialize a module.
- How important each module is in the framework.
- How many rounds a game lasts.
- What the AI's missions are in each game.
- What the size of each model is.

4.8.1 Building the Best AI From Test Results

After the tests have been completed, it should be possible to find the best AI.

We will use the module importance tests to justify which modules that should be replaced by the best performing AI techniques. If a module is not important, there is no real point in letting it use a well performing AI technique implement it. Instead the less time consuming technique should be used to implement it since this means less time spent in the AI's turn. However, if there is a clear tendency for a better performing AI technique on an unimportant module, we will still use this technique in the best AI. The argument for doing this is that the importance test is based on 500 games. This may not be quite enough to make a final statement on the importance of a module, and therefore we will let the performance tests have the benefit of the doubt. Also, unless the

time used by this technique is much greater than the fastest module, this is no real loss.

After the most successful AI has been constructed, it will make sense to let it play some other combinations of AIs to ensure its success. These could theoretically be random AI compositions, but to ensure it, it may make more sense to have a more systematic approach. This would be playing against combinations of the best AI with other techniques on the important modules. It is not useful to exchange unimportant modules since they will not impact the success rate of the AI. Also an AI constructed using "Stepwise Evolution" could be compared to the AI constructed through "Best Module". If these are not identical, they could play against each other to determine the best AI.

If any of the new AI compositions are remarkably better than the one we have found, it will be considered the best. The best AI should then be passed to the end users for them to supply feedback on the user experience of the AI. However, we do not in this project have the resources to have a user experience test, and will therefore not perform a such.

4.9 Performing the Tests

In this section we will go over the details involved in performing the more practical side of performing the tests. This includes the considerations of how to uphold the different requirements specified of the test, and the design of how to implement them.

4.9.1 Scheduling Module Performance Test

As written above, the performance of each AI technique for each module is tested separately. The best performing AI technique for each module is then combined resulting in the best AI.

But there are some issues when finding the best AI technique for a given module:

1. Each AI technique should play an equal amount of games, for comparison reasons.

2. Each AI technique should play an equal amount of games against each of the other AI techniques. Otherwise a given technique could play an unequal amount of games against an easy opponent compared to a harder opponent.

3. Each AI technique should be tested in games with all possible combinations of the other AI techniques, for the same reasons as above. For example, a neural network should play in a game with a decision tree only, but also in a game with both a decision tree and a Bayesian network.

4. As already stated, each technique should be tested in 3 player games, 4 player games, 5 player games and 6 player games, since the performance way vary when playing different amounts of players.

	Script	Neural Net.	Decision Tree	Bayesian Net.
In game	x	x		x
Not in game			x	

Table 4.1: *A single test game. The game consist of three players: one player with the given module implemented as a script, another player with the module implemented as a neural network and lastly a player with the module implemented as a Bayesian network.*

5. In a single game, no AI technique must be present more than once. In a game with two of the same technique, that technique will have at least one loser, and not necessarily a winner, which is unfair.

All these issues must be fulfilled when performing the tests. This is in fact a scheduling problem — determining which techniques should play who and in what order. Also, a different amount of AI techniques have been implemented for each module. So the schedules for each module is different.

This can luckily be solved in a very systematic way. This is best illustrated in an example.

A given module has been implemented with the following AI techniques: Scripted, Neural Network, Decision Tree and Bayesian Network (script, nn, dt and bn for short). Each of these techniques can either be in a given test game or not. One can not only play test games with all techniques included, since this would violate issue 3. Some way of making a list of games with all combinations of AI techniques for a given module is needed.

Table 4.1 illustrates a single test game and which AI techniques are included in that game. The game consist of three players: one player with the given module implemented as a script, another player with the module implemented as a neural network and lastly a player with the module implemented as a Bayesian network.

In this manner, the entire list of games to be played (schedule) of all combinations of AI techniques for a given module would consist of $2^4 - 1$ games. The case where none of the AI techniques are in the test game is useless in the test, which explains the minus 1. Such a list would in fact be seen in Figure 4.9.1

The first game in the schedule consist of only a scripted version of the module. The second game consist of a player with a scripted module and another player with a neural network implementation of the module.

To ensure issue 4, there should be 4 different lists. One for games with 3 players, one for games with 4 players, 5 players and 6 players.

If the schedule for example is a 4 player game schedule, then games with less than 4 players (e.g. game 1, 2, 3, 5 etc) are padded with non-framework players, such as the “easy AI” and “hard AI” already implemented in JRisk. If it is a 3 player game list, then games with more than 3 players are removed from the list.

Implementing The Schedule

Table 4.1 is an example of a single test game and can actually be viewed as bit string “1101”, instead of the table. This results in a very easy way of im-

1. script
2. script, nn
3. script, nn, dt
4. script, nn, dt, bn
5. script, dt
6. script, dt, bn
7. script, bn
8. nn,
9. nn, dt
10. etc.

Figure 4.4: *An example of a game schedule.*

plementing a method for creating a schedule. Making all combinations of AI techniques is in fact just a matter of listing the bit strings from "0001" to "1111".

This will result in a game list consisting of the same games as the list depicted in Figure 4.9.1. This list will however start with "bn", "dt", "dt, bn" etc.

When having games where the number of techniques to play in a game exceeds the number of player slots, we simply cut the exceeding players away. This can be done because the used method ensures that games between the removed techniques and those kept will eventually be played too. If there, on the contrary, are too few techniques to fill all available player slots in a game, the remaining slots are filled with random "easy" or "hard" AIs. An algorithm for generating such a test schedule can been found in Appendix E.1.

In this manner, each technique will play each other technique exactly the same number of times.

Ensuring The 500 Games

As stated in the previous section, it is necessary that each AI technique play at least 500 games, to ensure that the results are not based on luck and randomness.

In a game with three to six players, a full schedule for a module implemented with 4 different AI techniques will consist of 59[1] games. Any of the techniques will be in precisely 31 of those games[2]. For a technique to play at least 500 games, it must play the whole list of games $500/31 \approx 17$ times, resulting in $59 * 17 = 1003$ games played for a full test of that given module.

[1]When playing with four to six players, all possible combinations of AIs are possible (from binary 0001 to 1111). 1111 corresponds to 15 and having three of these gives us 45 games to play. For three players, the 1111 string is not possible, hence we have only 14 games. $45 + 14 = 59$

[2]Found by counting.

Chapter 5

Implementation

This chapter will briefly discuss the implementation of this project, and some of the problems that occurred.

5.1 Script Implementation

We have implemented the framework designed in [CJJ06] including the changes given in Chapter 2. Apart from the framework, we have implemented the scripted AIs for each module within the framework. Furthermore many of the scripts have also been redesigned before they were implemented. Details on the redesigned scripts are found in Appendix A. The changes made to the scripted AIs were made to make it perform better, in the sense of winning more games compared to the “Hard” and “Easy” AIs implemented in JRISK.

While the scripted AIs were implemented, the author of JRISK released a new version of the game with an “Extra Hard” AI. We do not have any knowledge of what changes were made to the new AI type, only that it should be harder to beat. When comparing our scripted AI to this new AI we noticed no difference in performance. Our AI was successful in beating both “Easy”, “Hard”, and “Extra Hard”. The goal of the scripted AI was to beat “Hard” and “Easy” in more than half of the played games. Having fulfilled this, the scripted AI was considered ready to produce training data.

5.2 Training Data Converter

The Training Data Converter has been implemented as described in Section 3.2. The Converter converts the game history produced by the scripted AIs into training data for each of the AI techniques. As described, the game history amounted to 30 gigabytes of data, which more than doubled when converted. If one is not accustomed to handling 70-80 gigabytes in around two million files, it can cause some problems. Such a large amount of data is for instance not easily moved. This also sets some limits on the applications handling the amount of data. Some of these limits are described in the next section. The applications developed solely for this project that handled the data, were all

developed with this in mind, such that they for instance not loaded the entire data into memory.

5.3 Training The AI Models

This section will cover some of the choices we have made during implementation of the various AI techniques. Also we will list some practical problems we have encountered during this phase. Some of them were solved but others were not.

5.3.1 Limitations in Training

Because the training data conversions and training of the AI techniques are very time and memory consuming, it imposed some limits on which techniques were trained with "winners and losers", "winners only" and which were trained with both. How each AI technique will be trained can be seen in Table 5.1.

	Decision Tree	Neural Net.	Bayesian Net.	Naive B. C.
Winners and losers	x	x		x
Winners only		x	x	

Table 5.1: .

The training data conversion and training of decision trees was quite problematic, therefore there was no time to train with winners only. The same goes for naive Bayes classifiers. Neural networks however, was implemented by us and there was therefore no problems with exaggerated memory usage, so neural networks were trained with both "winners and loser" and "winners only". The training of Bayesian networks in Hugin was very time consuming on the very large "winners and losers" training data set, and was aborted after almost a day of training on a tenth of the data. Bayesian networks was therefore only trained with winners.

5.3.2 Techniques Implemented on Modules

In Table 2.2 on page 36 the different AI techniques we wanted to implement on each module were presented. Due to various problems described in this section and time constraints, some techniques had to be been omitted. Table 5.2 shows which techniques have been implemented. Techniques marked with "x" are the ones implemented and trained in this project. The ones marked with "-" are the ones that were planned but were omitted. The "Random" AI has been removed since the random-technique is not used for testing. The new coloum "NNWO" are neural networks trained with winners only.

	Script	Decision Tree	Neural Net.	Bayesian Net.	Naive B. C.	NNWO
Initial army placement	x	-	x		-	x
Estimate opponents' missions	x	-	x		-	x
How close an opponent is to winning	x	-	x	x	x	x
The opponents' next moves	x					
How close an opponent is to owning a continent	x	-	x	x	x	x
Continent ownership	x					
MP Goal Weighting	x		x			x
Make attack plan	x					
Calculate attack plan cost	x					
Prioritize attack plan	x	-	x	-	-	x
Score attack plan	x	x	x		x	x
Discard attack plan	x	-	-		-	
Remove goals	x					
Calculate defense cost	x	x	x	-	x	x
Score merged plan	x	x	x		x	x
Prioritize territory needing defense	x	x	x		x	x
Cash cards	x	-	-		-	
Place armies	x	-	-		-	-
Transfer armies	x		-			-

Table 5.2: *Table showing which modules have been implemented with which AI techniques. "x" means that the technique is implemented for the given module, "-" means it is not.*

5.3.3 Dividing Into Ranges

Some of the AI techniques, such as Bayesian networks and decision trees are only able to handle a discrete set of states of a node/attribute. In theory there is no limit on the number of states the different techniques can handle, but in practice there are. We have chosen a maximum of 20 states on each node in both decision trees, naive Bayes classifiers, and Bayesian networks. However, in some cases this number was too high. More details will be given on this in the following sections. As discussed in Section 3.2.1, it is not always a good idea to just have a linear distribution over the values in the states. To find an appropriate distribution, we made an "Interval Maker" tool.

The "Interval Makes" simply does the following: For a given module, sort all the output from that module. Specify the number X of intervals you want it divided into (e.g. 20 for 20 states). The data will then be divided into X "chunks" of data, where each interval is from the lowest to the highest number in the chunk. This is illustrated in Figure 5.1 which is a sorted list of values, divided into 4 intervals.

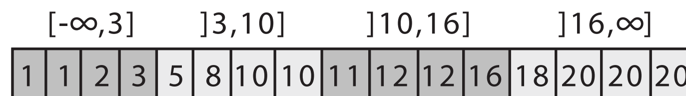

Figure 5.1: *A sorted list of values are divided into four intervals.*

5.3.4 Training Bayesian Networks

When modeling the Bayesian networks for this project, our intention was to use the "Interval Maker" just described. However, for the "IG: Close to Winning" a linear distribution are used for the nodes "Kill_Player X", the "Conquer24" and "Conquer18". Because of time constraints we have not found intervals for these nodes.

All nodes in both networks have a random probability distribution before the network is trained. This should ensure that we do not end up having an even distribution at some nodes in the network after it has been trained. Having even distributions on nodes in a network may result in a network giving a "static" distribution on other nodes regardless of instantiations.

As earlier mentioned, we use Hugin for training the Bayesian networks. Hugin uses an EM learning algorithm for training the BN.

Problems

A problem we experienced while training our BNs emerged from Hugin's way of training them. In Hugin, training is done by specifying a training data file which is loaded into the computer's memory. This approach may be feasible for some projects, but in our case this approach is far from optimal. The amount of training examples used in this project is very large. We had a 6 gigabytes training data file which is obviously too large to handle for most systems, and also for Hugin which, being a Java program, has a maximum of 4GB available. When using EM learning it is not necessary to load everything into

memory since EM only uses one training example at a time. Using this knowledge we came up with a simple fix for the problem. We simple divided the large file into several smaller files. By doing this, it was possible to train with training sets of any size with the only problem that we had to manually load new files whenever Hugin had processed one file. This leads directly to the next problem: There is no way of telling how far into a training file Hugin is. This means that we had no way of estimating how long it would take before a network was trained. Of course it is rather difficult to give a timed estimate on this since EM learning is a NP Hard problem, but at least an output of what number training example it was processing should be possible. This made it hard for us to estimate whether or not the BN was actually trainable with 6 gigabytes of training data, without spending weeks.

5.3.5 Training Decision Trees and Naive Bayes Classifiers

The AI techniques decision tree and naive Bayes classifiers were both implemented using Weka, so the cause of a problem with one technique will most likely also apply to the other technique. This section will cover the problems we had training the techniques which caused some of them not to be implemented in the game. We will then explain what actions we took to try and solve these problems and finally describe the general training procedure for these techniques.

Problems

The main problem that caused the training of some of the decision trees and naive Bayes models to fail, was that the Trainer ran out of memory. This happened even though the Trainer had allocated the system maximum of 3 GB memory. This is because Weka loads the entire set of training example into memory. And although the space required to train does not grow exponentially with the number of training examples, a hefty amount of training examples could still bring the system to the knees.

We also had problems with the Java Virtual Machine which would suddenly crash without a reason. It did so after the Trainer had run for some hours and after restarting the Trainer, it crashed after the same period of time. A search on the Internet did not give any result as to what the cause of the Java error was. We suspect this *could* be related to the problem with memory usage but cannot be sure.

Reducing Memory Usage

The memory usage depends on three factors:

- The number of attributes.
- The number of states in each attribute.
- The number of training examples.

The most effective way to lower the memory usage is to reduce the number of states in each attribute. This will both affect the number of branches in the

tree plus reduce the number of training examples. The latter is because of the way the Converter generates training examples from winners and losers: If there are t states in the target attribute, $t-1$ (the one action performed by the loser) training examples will be generated for a loser. Also, $t-1$ training examples will be generated for a winner in order to match the amount of loser examples.

A more brutal way of lowering the memory usage would be to just cut away a big part of the training examples and only train from a selection of the available training examples. This can lead to a poorly trained AI, an AI that potentially could have been better.

Training Procedure

At first it was tried to train the AI model as it has been designed. Then, if the training of the model ran out of memory we tried the following three steps:

1. Halve the number of states in the attributes.
2. Quarter the number of states in the attributes.
3. Cut away some of the training examples containing quartered attribute states.

When these steps had all been tried we had to consider the model untrainable and omitted it from the tests. When halving and quartering states or removing states in general, the model will become more inaccurate. On the other hand, when removing training examples, the model will not only become inaccurate, but it might also output wrong. This is why the removal of training examples is the last resort and only used when removing more states would make the model unusable.

5.4 Training Neural Networks

Implementing the training of neural networks did not produce any significant memory and time consumption problems. However, the matter also discussed in Section 3.3.3, were it is discussed that training neural networks (that also learns from losers) until it converges is impossible. Training the neural network for a fixed number of iterations is the only practical solution. The nerual networks that trained from winner only could however have been trained until they converged, but to compare the two approaches, it seemed fair only to training these neural networks a fixed number of iterations also. The number of iterations were selected to be 50 for all neural networks. This might seem a bit low, but the amount of available training examples is so great, so we believe that 50 iterations will be sufficient. The number of iterations were also selected based on the time it took to train each network. The most simple networks took around half a day to go through all iterations, but the more complex networks took many days. This also sets a natural limit on the number of iterations. Whether or not the neural networks have trained sufficiently will be shown in the results of the tests.

5.5 Implementing the Tests

Scheduling of the tests to find the best performing technique in each module was implemented as described in Section 4.8 and in Section 4.9.

These tests produced a file for each game played, consisting data such as "run times" for each module, the number of players in the game, the number of rounds the game lasted etc., which resulted in 3 gigabytes of data. A "Game Statistics Analyzer" then had to be written. This analyzer gathered all the information and produced summaries, such as the maximum, minimum, average and median of all the run times of each technique for each module, which mission each player won with, the amount of games a given technique for a given module has won etc.

These results will be presented in the following chapter.

Chapter 6

Results

This chapter will describe the results from testing module importance and module performance. First the amount of training data generated will be presented. This is the data used in the training of the learning algorithms. Afterward, the testing of the importance of each module is presented and analyzed upon. This is followed be a description and analysis of the performance test. The outcome of this result analysis is a proposal for the best combination of AI techniques in each module.

6.1 Training Amount

This section will cover the results of the training data generations — the amount of games played and number of module runs recorded.

We set a goal to generate around a couple of thousand played games, a maximum of 50 gigabytes data or use a maximum of one week to generate the data. The result was 6 days of computing time, 3275 generated games and 30 gigabytes of data. Table 6.1 shows the amount of data generated for each module.

Many of the modules use the board as input, and in some cases it is the same board used in a series of modules. So only unique board states are saved. The modules that use the same board state just link to the same board state file.

The data types “attack_plan” and “defense_plan” holds data for the modules in the Planner. Only attack and defense plans that are actually carried out are saved.

It might seem a bit odd, that there are almost twice as many IG_Mission files than there are for the other IGs. This is because a mission estimate is needed every time a initial placement of armies is made.

6.2 Module Importance

We ran the module importance test where the scripted AI for each module was replaced by the random AI as described in Section 4.2. They played against a

Module/Data Type	# Files	Data (MB)
IG_Mission	253104	13637
IG_Continent	131879	1371
IG_Winning	131879	2179
Master Prioritizer	131879	2713
RP_CashCards	11941	55
RP_Fortify	54047	219
RP_PlaceArmies	336881	1497
InitialPlacement	121225	721
Attack_plan	214822	852
Defense_plan	7473	29
Board states	784111	6166

Table 6.1:

random amount of JRisk's Easy and Hard AI's. 500 games was played for each module and the number of wins for the module with the scripted AI and with the random AI was recorded.

Table 6.2 shows the result of this test. The first two columns show for each module how many times it won with a scripted AI and with a random AI. The two last columns show how much lower the number of victories was when the scripted AI for the module is replaced with the random AI. Looking at the table, *Diff.* is the difference in wins and *Diff.*% is the difference in wins in percent for the random AI compared to the scripted AI. The larger the percentage value is, the more impact the module has on the whole AI's performance. Figure 6.1 shows this graphically.

Looking at the graph it is clear that not all modules impacts the AI's performance significantly. For example, the "Calculate defense cost" module only affects the AI's performance with 6%. This means that if even if the AI technique for that module was replaced with an AI technique that was as bad as the random AI, we would only be able to measure a 6% difference in the number of won games. Since a lot of factors influences who wins the game, a difference of only 6% will get lost among them and will not influence the number of won games in such a degree that we can measure a clear tendency. This also means that even if the module's AI technique was replaced by a super duper AI technique, it would not show in the test results.

In general, we will only consider a module's test result for valid if its impact is still at least 15%. Eight of our 17 modules does not fulfill this requirement. The modules are: "Estimate opponents' missions", "How close an opponent is to winning", "Calculate attack plan cost", "Calculate defense cost", "Score merged plan", "Prioritize territory needing defense", "Cash cards", and "Transfer armies". These modules are considered unimportant in the test result. Modules with an importance of at least 15% are considered fairly important, while modules with an importance of at least 25% are considered very important.

	Script	Random	Diff.	Diff.%
Initial army placement	193	119	74	38
Estimate opponents' missions	160	133	27	17
How close an opponent is to winning	185	166	19	10
The opponents' next moves	175	144	31	18
How close to owning a continent	172	139	33	19
MP Goal Weighting	216	112	104	48
Make attack plan	169	133	36	21
Calculate attack plan cost	179	149	30	17
Prioritize attack plan	238	62	176	74
Score attack plan	188	132	56	30
Discard attack plan	197	145	52	26
Calculate defense cost	173	162	11	6
Score merged plan	163	150	13	8
Prioritize territory needing defense	137	171	-34	-25
Cash cards	163	151	12	7
Place armies	171	108	63	37
Transfer armies	146	171	-25	-17

Table 6.2: *Results of the module importance test. For each module is it shown how many times it won with a scripted AI and with a random AI. Next it is shown how much worse the module performs when it is replaced by random AI: Diff. is the difference in wins and Diff.% is the difference in wins in percent.*

6.3 Module Performance

This section will discuss the results of the performance tests of each technique in each module. The tests were performed as described in Section 4.8 and in Section 4.9.

Each module will be discussed and a technique will be selected for the "Best AI", which should be the best compinations of AI techniques in modules possible. There will also be selected some "challenger" techniques in each module. These challengers will make up an AI that the "Best AI" could be evaluated against.

Some modules in the framework have only been implemented as scripts and is not included in the performance test.

The different test result can be seen in the following graphs:

Performance graph: Figure 6.2 in page 79 shows the number of wins per 100 games for each technique in each module.

Tun time graph: Figure 6.3 on page 85 shows the median run time for each technique in each module. The times have been divided by a benchmark which is calculated in the beginning of each game. This makes the times comparable even when the tests have been run on different systems. The times are in microseconds per benchmark.

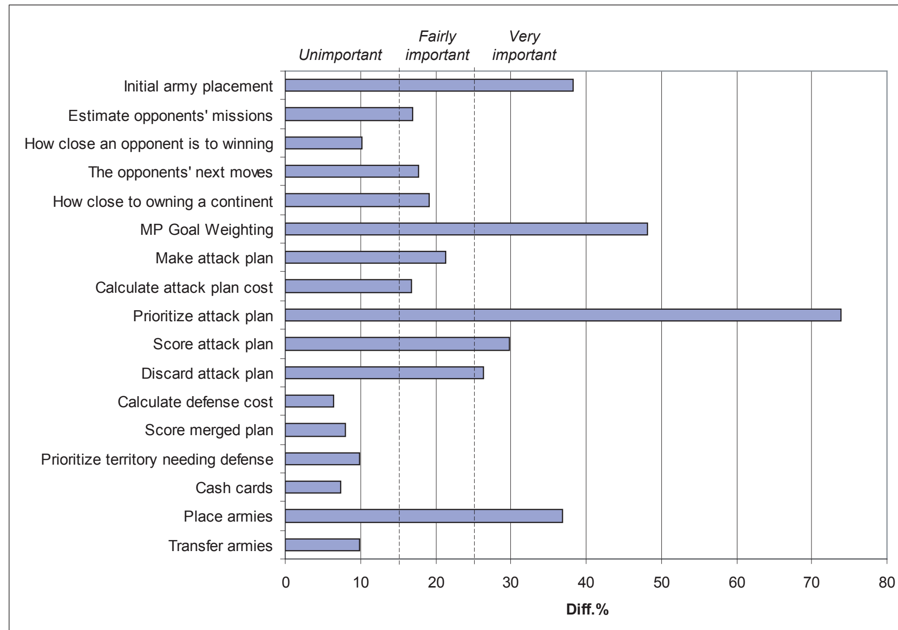

Figure 6.1: *This graph is made from the values of Table 6.2's last column, Diff.%. It shows how much lower the number of game victories is when the scripted AI for a module is replaced by the random AI.*

Turn time graph: Figure 6.4 on page 86 shows the median turn time for each technique in each module. The turn times are calculated by starting the clock when the AI begins his turn and stopped when he has finished fortifying and his turn ends. The times have been also been divided by a benchmark. The times are in microseconds per benchmark.

These graphs will all be used in the evaluation of the tests, but also the module importance test described in Section 6.2 will be used in the evaluation. The module importance is a direct measure of how influential each module is in the framework and thereby also how precise the test results are. If a module is not important in the framework, it can not be determined with certainty which AI technique is the best, unless there are some considerable difference in the performances. On the other hand, if a module is important in the framework, then the performance of each AI technique is directly comparable.

It was previously argued that the load time of models also is interesting to analyze. But load times only impact the beginning of the game, not the overall course of the game, nor the actual performance of the given technique. It has therefore been decided that load times are not analyzed further.

The actual performance tests where performed in 18 days. Before this, the scripted AI spent 6 days generating data followed by a couple of weeks of data conversion, where many of the conversions had to be restarted because of various problems such as excessive memory usage, server reboot etc. The range of systems used to generate training data, data conversions, and performing the tests can be seen in Appendix F.

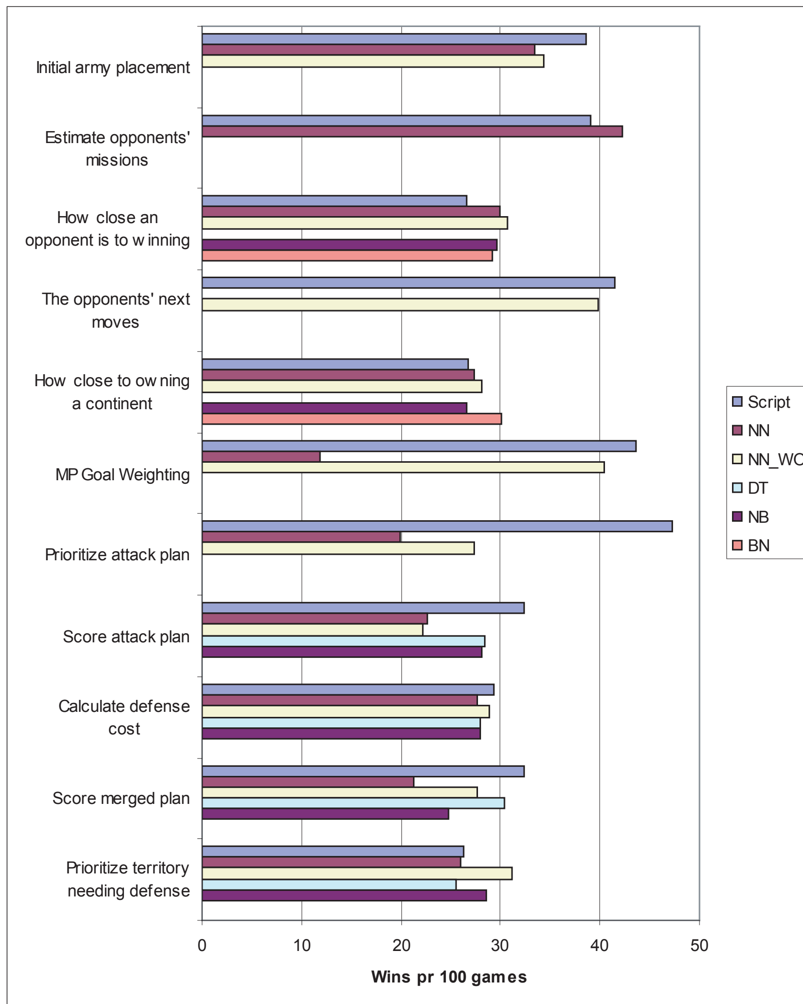

Figure 6.2: *This graph shows the number of wins per 100 games for each technique in each module. This will be called the performance graph.*

6.4 Test Results

This section contains an analysis of the results gathered through the module performance tests.

6.4.1 Initial Army Placement

Looking at the importance graph, we see that this module is fairly important. This means that this module has some effect on the output of the game since a change in performance of this module changes the general performance of the AI. Therefore we directly compare the values for the AI techniques in the performance graph since if one of them performs worse than another, this too will affect the general AI.

As seen in the graph, the script performs slightly better that both NN and

NN_WO. As argued in 5.4 the reason for this may be that both NNs have not trained enough.

The difference between the two NNs is very small, meaning that they perform equally good. The slight difference between them we subscribe to general uncertainties, but the equal performance is rather interesting. We would have assumed that one of them would perform better than the other because of the difference in training. NN_WO was trained by only winners whereas NN was trained by both winners and losers. This result indicates that the difference in the two training methods has no saying when training NNs.

Looking at the run time graph, we see that the run time for the NNs is roughly eight times longer than that for the script.

Summing up these result we conclude that the NNs performed slightly worse than the script for this module and their run times were also worse. This means that the scripted version of this module will be used for the "Best AI". However, the NNs on this module should be considered for the AI to challenge the "Best AI". The purpose of the "Challenger AI" is discussed in 4.8.1.

6.4.2 Estimate opponents' mission

Besides being rated fairly important in the importance graph, this module is interesting to us. This is because the NN on this module is the only one that has been trained with certain information instead of information estimated by the script[1]. This means that this IG is not relying on the output of the scripted AI and is therefore trained directly towards the correct value. Looking at the performance graph we see that the NN performs slightly better than script, which means that it has been trained to perform at least as well as the scripts.

Looking at the run time graph, we see that the NN performs roughly factor 120 worse than the script. However as seen in the turn time graph, the factor 120 means only a turn time increase of one percent. The radical time consumption of the NN is tributed the conversion of the past to NN input, which is a lengthy procedure.

Even though it is unclear, due to uncertainties based on the low importance, to say whether or not the NN plays better than the script, we will still use the NN in the "Best AI". We believe that the NN does perform better than our script since it has been trained towards giving the correct output given its input.

6.4.3 How Close an Opponent is to Winning

As seen in the importance graph, this module is rated unimportant. This means that the results are too uncertain to trust. Therefore we will look at the query and round times to find a technique for the "Best AI". We notice that the query time of the neural networks are 300 times slower than our script whereas the NB is 860 times slower, and the BN is roughly 7000 times slower. When looking at the turn times there is not much difference between the techniques except that the BN is a little slower than the rest.

There is not point in using anything else than our script in the "Best AI" for this module.

[1]Recall the when training this module we have certain information on the AIs mission

6.4.4 The Opponents' Next Moves

Looking at the importance graph we see that this module is just on the border between being fairly important and unimportant.

On the performance graph we see that the NN_WO and the script performs equally well. This measure of course has some uncertainties since the importance measure borders to "unimportant".

However, looking at the time differences we see that the neural network is roughly 5000 times faster than the script. Also in the round time the use of NN_WO gives a speed increase on 630%.

Therefore, we will use NN_WO in the "Best AI".

6.4.5 How Close an Opponent is to Owning a Continent

This module is fairly important to the AI.

Looking at the performance graph we see that all techniques on this module performs almost equally. BN seems to have a slight advantage since it scores a little higher that the remaining techniques. This may either be because uncertainties in the data, but it may also be because the BN is trained with winners only. It should thereby train towards the scripted behavior. What is really interesting is that it in fact outperforms the script. This indicates that BN may in fact be capable of catching "winning behavior" as discussed in 3.2.5. NN and NN_WO performs equally well, which is surprising since we would believe that NN_WO would actually perform better since it is always trained in the right direction whereas NN may not. The difference between the neural networks and the script is so small that it is subscribed to being uncertainties. The NB performs as well as the script in this module.

Looking at the run times for this module, we see that both neural networks are four time faster than the scripts whereas the BN is 27 times slower to query and the NB tops with a query time which is 130 times slower than the script. The explanation of the extremely high query times of NB may be that the module consists of 30 models which all in turn needs to be queried as described in **??**. Also when looking at the turn times of the module we see that the time spend in the module is not much dependent of the AI technique used. Only NB is somewhat more time consuming than the remaining techniques.

The conclusion to all this is that we will be using BN in the "Best AI" and use NN_WO in the challenging AI.

6.4.6 MP Goal Weighting

In the importance graph we see that this module is very important to the AI. This comes as no surprise since this module gathers all information from the IGs and outputs the overall goals for the AI in the current turn. Without this module, the AI would simply not have any goal to work towards.

Looking at the performance table we see that NN_WO performs almost as good as the script whereas NN performs very poorly. This indicates that training with both winners and losers does not work for neural networks, but training with winners only does. This may be due to the fact that when training with both winners and losers, we negate the error, and instead of adjusting the weights in the correct direction (as with winners only) we adjust the weights

in another direction. This direction is not necessarily the correct one meaning that the network will be trained to perform worse than before. This means that training is most likely to be slower for NN than for NN_WO, and in worst case that NN will behave randomly.

Looking at the run time table we see that the script is roughly 73 times faster than both neural networks.

Based on these results we will use the script in the "Best AI". NN_WO will be used in a "Challenger AI".

6.4.7 Prioritize Attack Plan

This is the most important module according to the importance graph, which means that we can rely on the results in the performance graph.

For this module, the script is almost twice as good as the NN_WO which is the best of the two NN's. Given enough training time the NNs might be able to at least mimic the behavior of the script, so the explanation for the NN's performance could lie here.

The NN_WO model performs quite a lot worse than the script, while the NN at the same time is nearing NN_WO. The poor results for NN and NN_WO may again be due to lack of sufficient training data.

The run time for the script is twice as long as the two NN's which causes an increase in turn time by 20% compared to the two NN's. If the NN's could be trained to match the scripted AI, they would be a better choice than the scripted AI because of their low run time consumption. But as the AI models look now, the script's performance is by far the best, it should be used for the "Best AI". None of the NN's are close enough to the script's performance to be picked to challenge the "Best AI".

6.4.8 Score Attack Plan

Looking at the importance graph reveals this module to be fairly important to the AI.

All techniques perform slightly worse than the script for this module. DT and NB performs equally and are slightly better than both NN and NN_WO. That DT and NB performs better than the neural networks is probably because of the low number of input for this module. The DT is fairly simple to build with only two regular variables and a target variable. The NB is also fairly simple to train, which explains why both of these perform well in this module. The reason that they do not match the scripts may either be due to uncertainties in the data, but also it may be because of the way they are trained. It may not always be wise to do "everything else" since this may slow the training down since "everything else" may sometimes contain more wrongs than rights. That the two neural networks perform equally well is again rather interesting since we would assume that NN_WO would train faster. It may be coincidental or due to uncertainties that their performance is equal.

Looking in the run times for the module, we see that the script is roughly 500 times faster than both neural networks, and 1000 times faster than both DT and BN. This is because this is a simple mathematical function. It is faster feeding the input directly to a function than queering any of the models.

Since the script outperformed all other techniques in this module, it is chosen for the "Best AI". Since DT and NB are close to matching the performance of the script, both are selected to be part of the challenger AIs.

6.4.9 Calculate Defense Cost

The importance graph shows that this module's effect on the AI performance is unimportant. Therefore the performance results for this module are also very close and difficult to conclude on. However, it can be seen that the scripted AI is remarkably faster than the others. It is therefore chosen as the "Best AI" for this module.

6.4.10 Score Merged Plan

The importance for this module is very low according to the importance graph. Therefore the result for it should be taken with a huge grain of salt. It seems that the technique used for the "Best AI" should therefore be picked with execution speed as the primary argument.

The scripted AI's run time is more than 200 times smaller than its closest rival, but the turn time graph reveals that the run time has no effect on the turn time which for all techniques in about the same. The difference at only 5% is so small that it most likely is caused by small errors in the measurement.

Since the time measures are so close to each other, and the performance measure cannot be trusted completely, the scripted AI is chosen as "Best AI".

6.4.11 Prioritize Territory Needing Defense

This module is not important but still we need to select an AI technique for the "Best AI".

Looking at the module performance graph reveals that all techniques perform fairly equal. NN_WO performs slightly better closely followed by NB. The module not being important explains the even distribution since it does not matter which technique is used.

Looking at the run time graph reveals that DT and NB use roughly 9000 times more time in the module, but when looking in the turn time graph we see that the time each turn takes is roughly indifferent.

We will therefore use NN_WO in the "Best AI".

6.5 Building the Best AI

After having analyzed the gathered training results, it is now possible to compose both the "Best AI" and the "Challenger AIs".

6.5.1 The Best AI

The "Best AI" is the one implementing the most successful technique on each module. The techniques have been analyzed through looking at the importance of the module they implement, their performance, and the time they use

Module	Technique
Initial army placement	Script
Estimate opponents' missions	NN
How close an opponent is to winning	Script
The opponents' next moves	NN_WO
How close an opponent is to owning a continent	BN
Continent ownership	*Script*
MP Goal Weighting	Script
Make attack plan	*Script*
Calculate attack plan cost	*Script*
Prioritize attack plan	Script
Score attack plan	Script
Discard attack plan	*Script*
Remove goals	*Script*
Calculate defense cost	Script
Score merged plan	Script
Prioritize territory needing defense	NN_WO
Cash cards	*Script*
Place armies	*Script*
Transfer armies	*Script*

Table 6.3: *The Best AI composition.*

to output a result. The "Best AI" is seen in Table 6.3. The techniques written in *italic* are the ones used on modules that has not been tested.

6.5.2 The Challenger AIs

The challenger AIs are AIs that will challenge the "Best AI". The challenger AIs are initially clones of the "Best AI". The difference between the challengers and the best AI is that the challengers will use different AI techniques on some of the fairly and very important modules. A list of the challenger AIs is seen in Table 6.4.

The next step would be playing a series of games where the "Best AI" and all challengers play against each other. This should, as with the performance tests, be done systematically meaning that each AI should play against all other AIs an equal amount of times. This is to ensure that all AIs are equally treated in the sense of the number of games they play, and number of oppponents in the games.

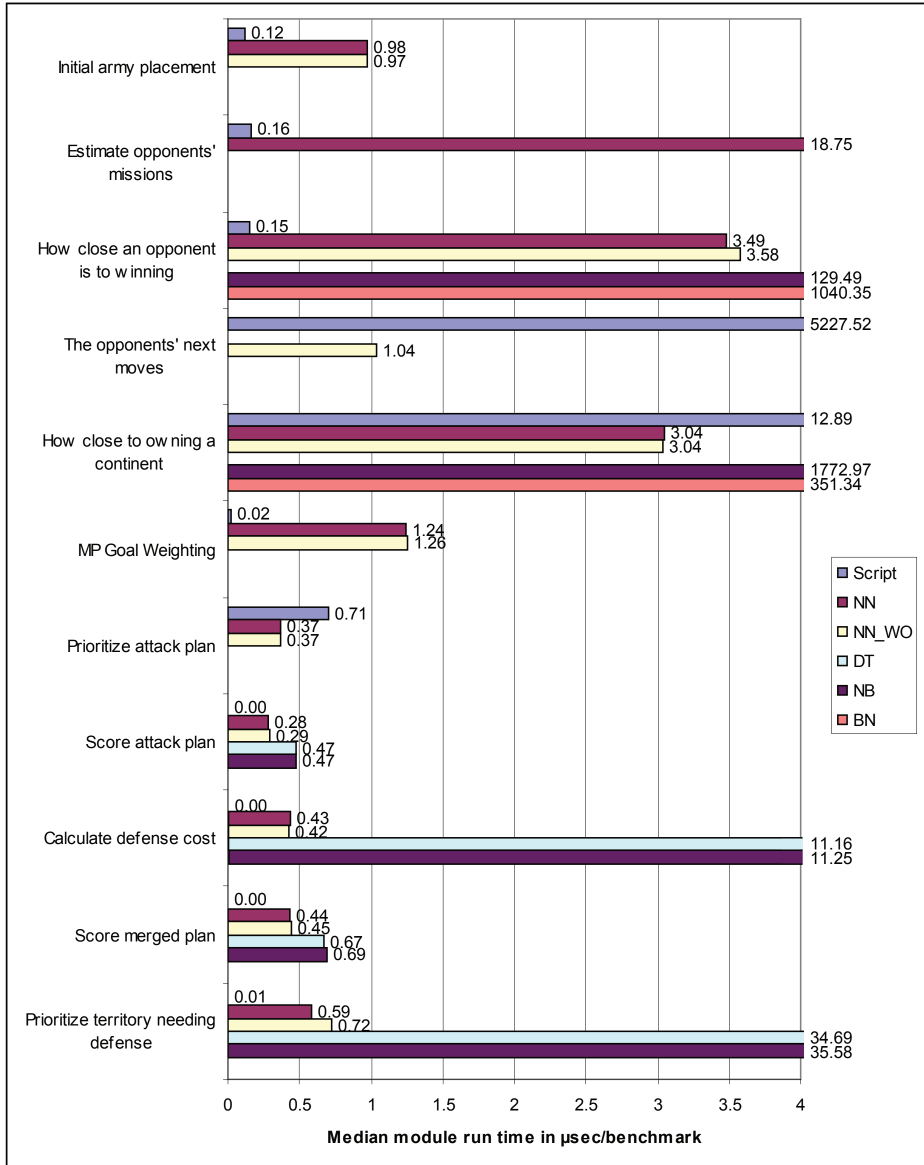

Figure 6.3: *This graph shows the median run time for each technique in each module. The times have been divided by a benchmark which is calculated in the beginning of each game. This makes the times comparable even when the tests have been run on different systems. The times are in microseconds per benchmark. This will be called the run time graph.*

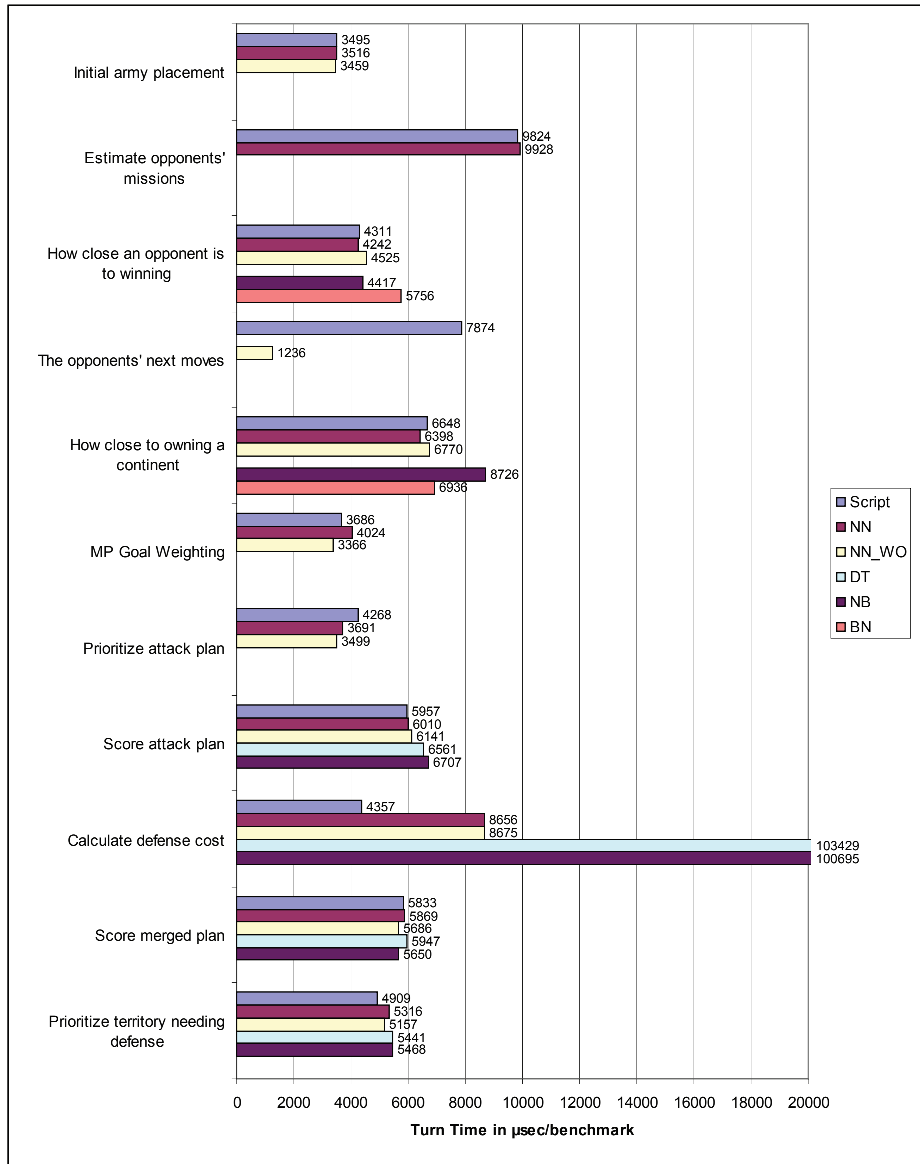

Figure 6.4: *This graph shows the median turn time for each technique in each module. The turn times are calculated by starting the clock when the AI begins his turn ("receives the dice") and stops when he ends his turn ("gives the dice to the next player"). The times have been also been divided by a benchmark. The times are in microseconds per benchmark. This will be called the turn time graph.*

Module	Challenger 1	Challenger 2	Challenger 3	Challenger 4	Challenger 5	Challenger 6
Init. army placement	NN	NN_WO	Script	Script	Script	Script
Est. opponents' missions	NN	NN	NN	NN	NN	NN
Close to winning	Script	Script	Script	Script	Script	Script
Opp. next moves	NN_WO	NN_WO	NN_WO	NN_WO	NN_WO	NN_WO
Close to continent	BN	BN	NN_WO	BN	BN	BN
Continent ownership	*Script*	*Script*	*Script*	*Script*	*Script*	*Script*
MP Goal Weighting	Script	Script	Script	NN_WO	Script	Script
Make attack plan	*Script*	*Script*	*Script*	*Script*	*Script*	*Script*
Calc. attack plan cost	*Script*	*Script*	*Script*	*Script*	*Script*	*Script*
Prioritize attack plan	Script	Script	Script	Script	Script	Script
Score attack plan	Script	Script	Script	Script	DT	NB
Discard attack plan	*Script*	*Script*	*Script*	*Script*	*Script*	*Script*
Remove goals	*Script*	*Script*	*Script*	*Script*	*Script*	*Script*
Calculate defense cost	Script	Script	Script	Script	Script	Script
Score merged plan	Script	Script	Script	Script	Script	Script
Prioritize defense	NN_WO	NN_WO	NN_WO	NN_WO	NN_WO	NN_WO
Cash cards	*Script*	*Script*	*Script*	*Script*	*Script*	*Script*
Place armies	*Script*	*Script*	*Script*	*Script*	*Script*	*Script*
Transfer armies	*Script*	*Script*	*Script*	*Script*	*Script*	*Script*

Table 6.4: *Challenger AI compositions.*

Chapter 7

Reflection

Until now we have argued that we would like to find the best performing AI for playing Risk. In this section we will argue that the framework made for playing Risk is general enough to suit other games. This is done in three parts. In the first part we will argue that Risk is general compared to other games. Secondly, we will argue that our framework is also usable for other games than Risk. In the third part we will go through each module and argue whether or not these too can be generalized. By doing this, we are able to argue that the AI techniques which performs best at each module will be the best at solving the general case in most computer games. Furthermore we will analyze which applications we believe the different AI techniques can be used to solve in computer games.

7.1 Generalization of Risk

When looking analytically at Risk, it becomes clear that there are two different goals of the game:

1. Conquer territories.
2. Prevent the opponent from conquering your territories

These goals can be divided into two groups: offensive goals and defensive goals, with the first goal being offensive. For completing these goals some resources are available. In Risk these resources are the number of armies placed on the board. As the number of resources decrease, so does the chance of winning the game. Generally, offensive goals can be said to improve the income of resources, where defensive goals prevents it from decreasing. In Risk the income of resources is highly based on how many, and also which, territories the player occupies. Conquering more territories means that more armies are available for reinforcing occupied territories on the board. Losing territories means a loss in income.

We believe that this is common in most computer games. Therefore we will now go through a couple of games to argue that the these goals are also present

in them. We will try to do a broad selection genre-wise to make sure that it is not limited to a specific range of games.

A weighting between when to meet offensive and defensive goals is what an AI playing Risk should handle since you cannot always win by constantly aiming for either of them. Our framework handles this weighting in the MP meaning that in games where this weighting is needed, our framework will be applicable.

By showing that AIs in other games can also be expressed within these two goals, we argue that our framework, is also able to handle an AI for these.

7.1.1 Risk vs. Counter-Strike

Counter-Strike is a real-time 3D shooter with two teams. One team has to place a bomb at a predefined spot in a closed environment, and the other team has to prevent them from placing the bomb[1]. To aid in their tasks, both teams has a broad selection of weapons available which are bought in the beginning of each round. Money are earned by shooting enemies and through winning the round. The winning condition of both teams is to kill all of the opposing team's members or fulfill their missions before the round ends at some specified time. More information on Counter-Strike is found on [cs].

Analyzing the tasks in the game reveals the following categorization:

Offensive goals: Kill opposing team members. Fulfill mission.

Defensive goals: Do not get killed. Prevent team members from getting killed. Prevent opposing team from fulfilling their mission.

The resources available for fulfilling the goals are the money earned throughout a round. These allows the player to buy better weapons the following round. If the player do not have any money, he must settle with cheaper weapons which will make his goals harder to fulfill.

So at these points Counter-Strike has something in common with Risk.

7.1.2 Risk vs. The Sims

The Sims is a micro management game. The task of the player is to control the life of a "virtual human". The player is responsible of making sure that his avatar has a job to earn money. Money is used to improve the life of the avatar by buying things for his house. The main objective in the game is keeping the avatar happy. There are some variables in the game which states the mental and physical health of the avatar. These are managed by letting the "Sim" watch TV, read, have a party, etc. Negative things like fires, not eating, not cleaning, etc, makes the Sim unhappy. More information on The Sims is found on [sim].

There is no real winning condition for the game since there is always something the Sim needs. So the goal of the game is just to keep the Sim alive and happy.

Not having a winning condition does not mean that the goals of the game cannot be divided into offensive and defensive goals. The offensive goals are

[1] There are a few more missions that will not be used here.

those that benefits the avatar in a major way, whereas the defensive are the basic tasks that needs to be done to prevent the Sim from dying or getting too depressed. The lists of possible things to do in the game is too long to list here, so we present a selection which should give an impression of how to categorize them.

Offensive goals: Getting a career. Getting married. Etc.

Defensive goals: Cleaning. Cooking. Eating. Making sure that the Sim is not late for his job. invite people over for parties. Etc.

Here the offensive goals improves the economy of the avatar, whereas the defensive goals keeps him alive and keeps him from getting depressed and so forth. A feature in The Sims here, is that it is actually through the defensive goals the offensive goals are met. By managing the defensive goals, the offensive goals are more or less automatically fulfilled. By making sure that the Sim is happy and gets to work every day will most likely get him promoted on his job. This means that he will make even more money and will be able to buy more things to make him even happier. This is of course a simplification since it is not that simple in the game. Neglecting the defensive goals will get the Sim depressed or sick, which will eventually make him lose his job, and thereby his income. This balancing of the goals does not mean anything special when comparing it to for example Risk.

7.1.3 Risk vs. Command & Conquer

Command & Conquer is a real-time strategy game. The point of the game is to destroy the opponent's units and/or buildings. To do this the player has an arsenal of different units available (soldiers, tanks, flying machines, etc.). He may build more from his base, given that he has the resources available for it. Resources are gathered by sending a special "Harvester" unit to harvest crystals at specific places on the map. The resources are shared, meaning that if one player harvest them, they are not available to other players. There is a fixed number of crystals available on the map. As the game progress, the player needs to build new buildings to improve his technology and thereby be able to build better vehicles. More information on C&C is available through [cc].

The offensive and defensive goals for Command & Conquer therefore look at follows:

Offensive goals: Destroy enemy units. Destroy enemy buildings. Destroy enemy harvesters. Harvest crystals. Advance in technology.

Defensive goals: Defend own units. Defend own buildings. Defend own harvesters.

Again the offensive goals are those that contributes to higher income, whereas the defensive ones prevents unnecessary spending on funds. If the defensive goals are not fulfilled it means that funding must be spend on rebuilding destroyed material. This will prevent the fulfillment of the offensive goals since the player's army is not increasing in size when it is getting destroyed.

7.1.4 The Generality of Risk

Through the analysis of these games, we have discovered that an AI solving them should handle a set of both offensive and defensive goals. This does not only apply to the specific games analyzed, but to the whole genre-class. The reason for this is that within a genre-class, the objective of the games seldomle changes. Therefore we argue that an AI framework that solves Risk, will also be able to solve a whole range of other games.

7.2 Generalization of the Framework

In the previous section we argued that Risk catches a feature present in most games, namely the presence of offensive goals that add to an increment in needed resources, and defensive goals that prevents the decrement of resources.

In this section we will analyze whether our framework, being built for Risk, is also general enough to be useful for other games.

We will go through all three layers in our framework model (IG, MP, and RP) and discover if each have a place in the games we have examined in the previous section.

7.2.1 Our Framework in Counter-Strike

The following sections shows why we believe our framework is usable for building an AI that plays Counter-Strike. We will not give complete lists of what the different layers are useful for since this would require a deeper analysis of the game. Instead we will merely give a few examples to show some of the usages of each layer.

IGs in Counter-Strike

In Counter-Strike (CS), the players can only see what their avatars can see. Therefore a player has to guess where an opponent is if he for instance disappears around a corner. If an AI should take the player's place in the game, an IG would definitely be used for this estimation task. Another use of an IG would be finding the best hiding spot based on where the AI believe the enemies are. This would be useful to find both the best cover if the AI is under attack and also if the AI wants to find a long term hiding place.

So there would be some use in having IGs in a CS AI.

MP in Counter-Strike

It is possible to make a list of goals to use in CS. Some of these goals are "Plant bomb", "Shoot at enemy", "Move to", "Defuse bomb", "Get better weapon", etc. What goal to take is highly dependent on the situation, meaning the situation as presented by the IGs.

RP in Counter-Strike

An RP in CS would have the same task as in Risk: To convert the MP goals into usable plans based on the available actions (move, shoot, reload, etc.).

CS Conclusion

The previous examples shows that our AI framework would indeed be useful for building an AI for CS. There are some uncertainties which can be handled in the IG layer, goals to be handled by MP, and actions that needs to be planned within the RP. Since the game is real-time, neither of the layers can use to much time to output, but this is not depending on the framework, but on the implementation of the layers.

7.2.2 Our Framework in The Sims

The following sections shows why we also believe our framework is also usable for building an AI that plays The Sims. Note that this is not an AI that take the role of an NPC in the game, but one that takes the role of a human player.

IGs in The Sims

In The Sims we do not find any need for IGs. Everything is already presented through the graphical interface. Of course the AI should not analyze the GUI, but this also means that all information needed is already presented through some "internal" information givers which are already available. There is nothing unknown that needs to be estimated, hence there is no need for information givers in the game. This however does not mean that our framework is useless. This only means that the game mechanics already handles what we would require from the IGs.

MP in The Sims

It is possible to make a list of goals for the avatar in The Sims to do. These are for example "Go to work", "Go to sleep", "Go to the bathroom", "Cook meal", etc. Since all information is available to the MP from the game itself (for example that the avatar needs to go to the bathroom), it should be possible for the MP to find a correct goal to output ("Go to the bathroom").

RP in The Sims

The RP in The Sims would decide how to fulfill the goals emerging from the MP by changing the goals into actions (Walk, Use, Dance, etc.).

The Sims Conclusion

As shown, it is possible to divide the problem of playing The Sims into smaller tasks that fit within our framework. The only thing to notice is that the IGs are already given by the game. This should not pose a problem since this only means that the MPs input is already defined. In 7.1.2 we noticed that the defensive goals are the most important ones in The Sims. This is not a problem in our framework since this just needs to be handled by the MP.

7.2.3 Our Framework in Command & Conquer

The following shows how our framework can be used to make an AI for Command & Conquer (C&C).

IGs in Command & Conquer

There are some uncertainties in C&C which can be estimated by some IGs. For instance, you do not have full information of the map at all times. It only is possible to see what is going on in an area around each owned unit and building. So if a player unit passes an enemy unit, an IG should start estimating not only where the enemy units are going but also where they come from (the enemy base). An IG estimating the size of the enemy's army, and also how improved his technology is, would also be extremely useful. So IGs would have their right in a game like C&C.

MP in Command & Conquer

The MP will in C&C work in the same manner as in the other games described. It outputs goals based on how the world looks and the estimates given by the IGs. Some of the goals it could issues could be "Attack target", "Defend target", "Go to position", "Build vehicle X", "Build building X".

RP in Command & Conquer

The RP in C&C would be responsible for carrying out orders issued from the MP. It would handle all planning and resource management involved in both attacking and building a base/vehicles.

C&C Conclusion

In the previous we have seen that it is possible to build an AI for C&C within our framework. There are some uncertainties which are covered by the IG, some goals to handle by the MP, and some actions that must be performed through the RP.

7.2.4 Summary on the Generality of Our Framework

As we have argued, our framework applies for building AIs for other games than Risk. As mentioned earlier: Not only do we believe that it goes for the specific games named, but also for the class of games they represent. The reason for this assumption is that the only things that separate the named games from those within the same genre is added functionality and new graphics. Added functionality is handled within the RP which will have new ways of fulfilling its goals.

That the framework is general is an important point to make, since this allow us to make conclusions concerning the generality of the AI technique used within the different layers in our AI framework.

7.3 Generality of the Applied Techniques

Now that the generality of Risk as a game and our framework as a general AI framework for implementing common AIs has been discussed, it is time to discuss the generality of each the applied AI techniques for implementing AIs in other computer games.

7.3.1 Generality of Neural Networks

When looking at the applications of neural network, one could conclude that they are born estimators, which also comes apparent through their deployment in real life. Most applications of neural networks tend to pertain to pattern recognition, or deriving some abstract meaning from complex data. This can also be seen in the way neural networks are constructed and trained — the idea of having weights incrementally adjust toward some generality of the training data. However, this suggests quite heavily that neural networks are not prone to producing some exact value from some input, such as summation of real values.

Another aspect to consider, especially in computer games, is the computation time used on querying the network for an output. This is one of the advantages of neural networks. If a computer game, or some other application for that matter, has some computational hard and time-consuming procedure in which the output is some abstraction over some large and complex data set, then it would be wise to have a neural network try to learn to mimic the procedure and perhaps produce a result much quicker. This can also be seen in the results from the tests in this project, such as the *IG NextMove*, which is very time consuming. The corresponding neural network produce an output much faster (1000 times faster actually). With a bit more training of the particular neural network, it should mimic the script much better and thereby be a valid substitute.

Problems in Training

Looking a bit closer on the result of the tests made in this project, one can conclude that the hypothesis of learning from losers by doing the opposite, did not seem to be effective. The reason might be that the networks were not trained sufficiently. But as argued, there is no way of knowing when to stop the training. The network will never converge to the training data, since the desired behavior of the network lies outside the training data. One could solve this by training the networks further, run the test again, train some more if necessary and so on. But this is very time-consuming and could not be implemented in the time scope of this project.

Generally, the networks trained with winners only did not perform as well as the scripts. The reason for this could also be lack of training. It would in fact have been possible to train these networks until they converged to some small error. The reason why the two were trained in the exact same way with an

equal amount of iterations was to compare the two procedures of neural network training. The performance of the "winners only" networks could probably have been improved with more training. But compared to neural networks trained with "winners and losers", the "winners only" networks performed generally much better. So the conclusion would be that "doing the opposite of losers" does not seem to produce improved behavior. Whether or not the "winners only" networks become better than the general behavior of the scripts, as argued in Section 3.2.5, could not be concluded. The networks did not outperform the scripts, but with more training they might.

Applications of NN in Computer Games

The usability of neural networks in computer games is quite good. As already mentioned, the run time (query time) of a neural network, makes it very usable in also real-time computer games. The ability to mimic the behavior of script or procedure (but not copy the behavior) makes it very usable for heavy data analysis, such as the decision making process of an AI agent, where the corresponding script might take too long.

7.3.2 Generality of Decision Trees

Looking at our test results for the four modules for which this technique was implemented, decision trees worked quite well. The technique performed best with modules with a simple input in terms of number of variables, while it performed worse when the input was more complex.

A weakness with decision trees is that if the number of states in the variables is large then it will not be able to build a model that covers all cases, because this would require training examples where all situations are present. Then, when the trained model is used to classify a previously unseen instance, it ends up at a NULL leaf and does not know which classification to give. We then made it return the leaf value that had been seen most often given the state of the instance's other variables. This inevitably will lead to bad decisions.

In general, decision trees would be best suited for tasks where the total amount of states in variables is minimal, to avoid meeting a situation it has not seen in its training examples.

With the simplicity of decision tree in mind, the run time seems a bit in the high end. This is likely caused by Weka whose ability to be flexible and to be used by many AI techniques inevitable will make it slower than a specialized implementation made solely for the purpose of decision trees. The run time could probably be lowered this way. However, the run time for a decision tree is low enough to find usage in e.g. computer games where response time is usually of high importance. As long as the input is quite simple, its run time is low, but as the input gets more complex, the run time rises accordingly.

Problems in Training

In our case, the decision trees failed to train on half of the modules. The training example amount was large and each variable had 20 states. These two factors made training a problem because of the way the ID3 algorithm works. The more states, the lower chance ID3's candidate model will correctly classify

the rest of the training examples. ID3 continuously constructs a new candidate model from a bigger and bigger amount of the training examples until it correctly classifies the rest of the examples. With many states and variables, it potentially will continue to fail until the candidate model covers all training examples. However, with a training example amount of 3 GB, this will not work.

We trained all decision trees with "winners and loser" examples, and since the decision trees performed well, this suggests that this idea of doing "everything else that what the loser did" is a viable way of training a decision tree. It would be interesting to see how this method compares to training with "winners only".

7.3.3 Generality of Naive Bayes Classifiers

The naive Bayes classifier's performance is rather fluctuating but generally it performs well in our tests. In "Score attack plan" which has a simple input it shares the second place with decision tree, but is not far away from the script which performs best. In "How close to owning a continent" which has a complex input it shares the last place with the script, with which it has a comparable performance. In fact, with complex input it never performs worse than a decision tree. This gives a hint on that naive Bayes handles complex input quite well, and since its performance with simple input is decent, it seems like a good choice for an all-round technique. It is interesting this is the case despite the fact that the naive Bayes' assumption on variable independence is very unlikely to always hold.

Compared to the other AI techniques it has the advantage that it is fast to train while pertaining a decent performance. The run time is the same as the decision trees, which makes sense, since they both rely on simple table look-ups when calculating the output. Besides, they are both a part of the Weka implementation so they will suffer from the same possible slow-downs Weka may introduce. As with decision trees, the run time increases with the complexity of the input. Overall it seems to handle different situations quite well, and since it is quite simple it can easily be incorporated in a computer game where it will probably have a decent performance.

Problems in Training

We had some problems with getting the naive Bayes classifier trained through Weka, since it ran out of memory. This is because Weka loads all training examples at once, even when this is not needed in the case of naive Bayes. A custom implementation only loading one at a time would be able to train naive Bayes no matter the amount of training examples. This would make it possible to test the naive Bayes technique with the rest of the modules. We also only tried testing naive Bayes with "Winners and losers" but not "Winners only" with which it might perform even better.

7.3.4 Generality of Bayesian Networks

Having worked with Bayesian networks through this entire project, we have come aware of their pros and cons. The major benefit of working with them

is the simplicity of the model. It is quite easy to model causalities between different nodes representing real-life observations. An expert to the domain can incorporate his knowledge directly into the model structure and thereby make the model fit the domain. The downside of this is that one tends to forget what happens when one node causally influences another. Namely that the CPT of the influenced node grows exponentially with the size of the number of states in the parent node.

In a perfect world it would be possible to specify the CPTs of all nodes by hand, but when you have roughly 20 states in each node in a network, this is no longer an option. Instead we trained each node using the EM algorithm.

Problems in Training

We started out by training the BNs using both "winners and losers" data, but it turned out that it would take far too long to train any of our networks this way which is why the BNs are trained with winners only.

The training of "IG: Close to winning" used roughly 24 hours going through its winners-only training data one time. With training data from both winners and losers, the amount of training data would be multiplied with the number of states within the node we wish to observe minus one. So the time it takes to train a larger network with a large amount of training data, is something that needs to be taken into consideration when wanting to train BNs.

Applications of BN in Computer Games

As already mentioned, the pros of using BNs is its modeling. The cons are their query times. Going through the test results, we find that the performance of the BNs are matching that of NNs, DTs, and NBs. However, the query time for BNs is extremely long compared to especially NNs. The query time is of course dependent of the BN model since propagating it is NP-hard. This is a problem if the technique should be used in for example a real-time game, where decisions may need to be made in split seconds. But, as indicated by the test results, BNs are very useful when doing estimates based on input that causally influences one another. So its main applicability in computer games lies in information gathering. So if we should state where the main use of BNs is in our framework, it would be would be in the set of IGs.

7.3.5 Generality of Scripts

Scripting is the most general of all AI techniques. This is also why this is the most commonly used technique. It is a very intuitive approach to constructing an AI since it is based on how the developer believes the AI should behave. As such, the developer has 100% control over the AI behavior, given there is no randomness in the scripts. However, the success of the AI is highly dependent how well it has been designed. If it is too transparent what the AI does in different situations, no one will have problems beating it which is fatal for an AI since without AI opposition, a game will lose its value.

Problems in Implementation

The major downside of scripting is, that as the game complexity rises, so does the things to handle by the AI. Thereby the design and implementation tasks grow very complex too. It is by far easier to train e.g. a neural network to handle a complex task, given that sufficient training data is available, than coding a script for it. In this project this is best seen through the script of the MP. This script handles a lot of different input, and output a list of goals which is very important to the entire AI. Because of this importance, the script needed to be carefully designed, implemented, and tweaked in order to behave optimally. A neural network trained from the training data performs almost as well a the script without any tweaking. But of course the training of the NN requires the presence of training data, which in our case was in fact generated by the scripts.

Applications of Scripts in Games

As mentioned, scripting is by far the most general of all AI techniques. This is also visible in this project since we at first build our entire AI as scripts to generate training data. This is also a major benefit for scripts, that they are highly versatile. Generally, scripting is also the most commonly used method for AI design in the game industry.

Chapter 8

Conclusion

This chapter will conclude the combinatory tests of AI techniques in computer games performed in this project.

The work in this report is based on the work in [CJJ06], which documents the design of a framework for an AI playing the board game Risk. The purpose of this framework is to test the usability of common AI techniques in common tasks in computer games in general. The framework is therefore designed to be modular, incorporating many common tasks in computer such as information gathering, estimation and planning. The framework is also designed to be general enough to cover other types of games, meaning that the results of these tests can be generalized to include most computer games.

The designed model is a three layered framework consisting of the first layer called Information Givers, the second layer called the Master Prioritizer and the third layer called the Round Planner. Each layer consist of numerous modules implementing the different tasks in each layer. Each module is defined to take some input and produce an output. This results in a complex interconnection of modules passing data to each others. Each of the three layers define different types of modules:

The Information Givers inspect the environment in which the AI act and produce estimates and abstractions on various useful information. This information is available to both the Master Prioritizer and the Round Planner.

The Master Prioritizer utilizes the information provided by the Information Givers and prioritizes a list of goals the AI should act upon. The goals are dependent on the game.

The Round Planner use the prioritized list of goals and possibly some information from the Information Givers to plan a series of actions to do. How the Round Planner produce this plan is highly dependent on which game it is implemented in, which also makes the composition of modules in the Round Planner dependent on the game. However some of the modules in this project are very general for most games, such has prioritizing

plans and estimating their cost.

The work in [CJJ06] also includes an analysis of the most common AI techniques used for implementing agents in computer games. This analysis covers scripting, which is the most used method of implementing AI behavior. The behavior is implemented directly in the code by the programmer or designer and does not include any learning. Different learning techniques were also explored. These are: neural networks, decision trees, Bayesian networks, and naive Bayes classifiers. Which technique can fully implement the behavior of each module is then analyzed. The input and output of each module set some constraints on which techniques can be used for that given module.

Common to all learning techniques is that they learn from training examples. It was decided that the scripted implementations of all the modules should generate training examples for the learning algorithms by playing thousands of games of Risk. The fact that the learning techniques only learns from scripted modules, raises the question of whether or not this will result in the learning modules only mimicking the behavior of the scripts, instead of improving the module behavior. It is therefore proposed that instead of learning to do the same as every training example, the learning algorithms will do the same as the winning players, but not do the same as the losing players. It is proposed that the learning algorithms could either do the opposite of the losing players or do everything else but the losing players.

In this project the framework and scripted versions of each module were implemented. Alongside this a training data converter, learning framework, data analyzer, neural networks and an interval maker was also implemented. The interface between JRisk (the Java implementation of the game Risk) and each of the AI techniques was also implemented.

The framework and the scripted modules were, aside from a few changes, implemented as designed in [CJJ06]. This included some new modules, and it was found which AI techniques could implement these. When the implemented framework and scripts were done, the generation of training data for the learning techniques was initiated. This resulted in data from 3275 played games and more than a million training examples for all the modules.

The training data converter was constructed to convert this generated game history into the file formats needed by each of the techniques. The decision trees and naive Bayes classifiers both used the Weka Machine Learning Project and the game history was therefore converted into the Weka ARFF file format. For Bayesian networks, the game history was converted to the Hugin learning file format. Since neural networks were implemented solely for this project, the game history was converted into a training data file format that was designed for this purpose.

In this project various ways to test the different AI techniques was proposed and the chosen method was thereafter implemented. The chosen method tests each module separately, since a complete combinatory test of all techniques in all modules would result in millions of combinations of AIs. The applied test systematically combined the different techniques in each module to find the most winning, and the fastest technique for each module.

The results showed that the proposal of learning from losers by doing the opposite in neural networks did not result in improved behavior compared to the scripts. Learning from winners only in neural networks however shows acceptable performance with respect to the number of won games. It was argued that with some more training it might perform at least as good as the scripts.

Another significant result is the fact that in some cases the neural network performed factors faster than the script with an equal amount of won games, meaning that the neural network could easily replace the script. This is very useful in real-time games were the AI has limited time to plan, but most definitely also useful in other games and applications.

For decision trees and naive Bayes classifiers, doing everything else but the losers, did result in some improved behavior compared to the scripts. This is actually quite interesting, since the techniques were pulled in one direction by the winners and in all other directions by the losers. However the downside of decision trees and naive Bayes classifiers is the long query time compared to scripts.

Bayesian networks performed very well when training from winners only, even with only a few iterations through the training data. Like the decision trees and naive Bayes classifiers, the Bayesian networks are very time consuming when querying the network, so in modeling the network one should keep the model as simple as possible.

From the test results we were able to compose an AI that implements the best performing techniques on each module, the "Best AI". This AI should be the best performing AI, but to make sure, we also composed some AIs which implement the techniques performing nearly as good as the best techniques on important modules. This have resulted in six "Challenger AIs" which can challenge the "Best AI" and reveal if some of the modules have been falsely chosen due to uncertainties in the test data. This test was not run simply because it takes too long, so it would not be finished before the deadline of this project.

Specific for the AI techniques we found that neural networks are useful for making fast decisions when given a large amount of input. This makes it very useful in planning, where even scripts may take too long to process the data.

Decision trees are also useful in situations where time matters. However, the input to these needs to simple in terms of few input variables for the trees to be effective. When there are too many input variables, training the trees becomes problematic and therefore they become imprecise.

Bayesian networks are not useful for time-critical tasks in computer games. The propagation of the networks are simply to slow for this. Therefore BNs are mainly for use in long term information gathering tasks or planning in turn-based games.

Naive Bayes perform decently within all task in both performance and query time. Therefore this model is qualified for most tasks within the framework.

Scripting is useful for every task. This is shown through the implementation of the scripted AI. However, scripting takes far more time to implement than using learning techniques. Also the scripts are not capable of learning, meaning that they are "trapped" in the script designers way of thinking.

We have argued that our framework designed for Risk is general enough to handle other games. This is done through showing that it is possible to divide Risk into two sets of goals that it must prioritize in order two win: offensive and defensive goals. We show that exactly these goals are also present in games of other genres. With this said, we are able to argue that an AI technique which is good at solving a task within the framework in Risk is also good at solving this task within other games.

8.1 Future Work

There is still a few thing that would be interesting to investigate regarding this project. In this section we will go a few of the more obvious choices.

Challenger Test

The test between our "Best AI" and the "Challenger" AIs was not run in this project due to time restrictions. Therefore an obvious extension to this project would be running this test to see if the "Best AI" is in fact the best AI.

Stepwise Evolution

Another interesting discovery to make would be by implementing and running the "Stepwise Evolution" approach to see if it, as argued, produce the same AI as the "Best Module" approach.

Implement DT and NB

We have argued that Weka may slow down the query times of the DTs and NBs. Therefore it would be interesting to implement DTs and NBs ourselves, thereby rule out the possibility of Weka slowing down the run times of the techniques.

Training Issues

There are numerous extensions to the way that we train that could be applied:

More training

We have argued that the lack of training have some responsibility in the techniques not matching the performance of the scripts. Therefore it would make sense to let the techniques train more. NN_WO should for example be allowed to train until they converge to see, if they perform as well, or even outperform, scripts.

Winners only in DT and NB

It would be interesting to train both DTs and NBs with winners only to see if these new trained versions would outperform those trained with both winners and losers. This would supply us with final evidence on whether or not the "do

as winners and everything else but losers”-method is better than just doing the same as winners always.

Bibliography

[AOJJ89] S. K. Andersen, K. G. Olesen, F. V. Jensen, and F. Jensen. Hugin - a shell for building bayesian belief universes for expert systems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, 1989.

[cc] Command & Conquer Official Homepage. http://www.ea.com/official/cc/.

[CJJ06] Pelle Coltau, Jens Juul Jacobsen, and Brian Jensen. Artificial intelligence in computer games. Aalborg University, Department of Computer Science, Fredrik Bajers Vej 7, building E, DK-9220 Aalborg, Denmark, January 2006. Available through the Electronic Document Library http://www.cs.aau.dk/library.

[cs] Counter-Strike Official Homepage. http://www.counter-strike.net.

[DHS73] Richard Duda, Peter Hart, and David G. Stork. *Pattern Classification and Scene Analysis*. Wiley, New York, 2nd edition, 1973.

[ED90] Russel C. Eberhart and Roy W. Dobbins, editors. *Neural Network PC Tools - A Practical Guide*. Academic Press, Inc, 1990.

[jav] Java by Sun Microsystems. http://www.java.com.

[Jen02] Finn V. Jensen. *Bayesions Networks and Decision Graphs*. Springer, 2002.

[JLO90] F. V. Jensen, S. L. Lauritzen, and K. G. Olesen. Bayesian updating in causal probalistic networks by local computations. *Computational Statistics Quarterly*, (4):269–282, 1990.

[jri] JRisk by Yura Mamyrin. http://jrisk.sourceforge.net/.

[Lar00] Lars Mathiassen and Andreas Munk-Madsen and Peter Axel Nielsen and Jan Stage. *Object Oriented Analysis and Design*. Marko Publishing ApS, 2000.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Companies, Inc., 1997.

[Qui86] R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[Rab02] Steve Rabin, editor. *AI Game Programming Wisdom*. Charles River Media, inc., first edition, 2002.

[sim] The Sims Official Homepage. http://thesims.ea.com.

[SL94] James V. Stone and Raymond Lister. On the relative time complexity of standard and conjugate gradient back propagation. *1994 IEEE International Conference on Neural Networks*, 1:84–87, 1994.

[Tho] Thomas D. Nielsen. Notes on EM Algorithm.

[wek] Weka ARFF file format specification. http://weka.sourceforge.net/wekadoc/index.php/en:ARFF_(3.5.2).

[WF05] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005.

[Wu95] Xindong Wu. *Knowledge Acquisition from Databases*. Ablex Publishing Corporation, 1995.

Appendix A

Changes in the Scripted AI

This appendix contains all adjustments made to the scripted AI. All of these adjustments were made either while implementing the AI or while tweaking it towards beating the AIs distributed with JRISK.

A.1 IG: How Close The Opponent Is To Owning a Continent

This IG has one more input than the one in the original design. It takes a Boolean "BeginningOfTurn", stating whether or not the module is run at the beginning of the AI's turn. The reason for this is that this IG may be run in the middle of the AI's turn if the AI have conquered a continent. However, it should not in this case consider reinforcements that it should receive itself in the current turn, when estimating how close it itself is to occupying a continent (since these reinforcements have already been used).

A.2 IG: Ownership

This new IG is used to get knowledge of if a player fully occupies an entire continent. It outputs a two-dimensional array (continents and players) with a Boolean value stating whether or not a given player occupies the continent. The input to this IG is the board. To fill this array, it runs through the territories within a continent and checks if a certain player occupies them all. If so, the place in the array belonging to that player and that continent is set to true, if not it is set to false.

A.3 IG: How Close The Opponent Is to Winning

Appart from the input mentioned in the original design, this IG is also given a Boolean telling whether or not it is run in the beginning of a round. This is needed since the IG may be run in the middle of a round. When this happens, it

must not consider reinforcements that the AI itself should receive in the current round. Another input given to the IG is an estimate on how close an opponent is to occupy a continent. This is needed to calculate the winning estimate if an opponent is estimated to have a continent mission.

A.4 Master Prioritizer

Instead of doing an intitial distribution of point followed by a redestribution of the same points, another approach is simply to give the goals the points they claim and nothing else. This approach should give a more ″clear″ description of precisely how important a goal is. This may also benefit the part of the RP evaluating plans. The difference from the original design: Only goals that have some importance are given points. In the original MP, goals that had no ″real″ importance could be passed to the RP in certain situations due to the initial distribution. Goals no longer sum to 100. There was no real reason for them to do so. In the old version, if the highest score of any of the goals was 20, we would have no real knowledge of how important that specific goal is without looking at all other goals. In the new version, the importance is in fact the importance of the goal. The same problem persisted regarding a 50/50 distribution over two goals. There was no way to know how important these goals really were. We know that they would probably be high, but not how high. They could for example be 70/70, 80/80 or even 100/100 in reality. This would in fact leave it up to the RP to decide when a goal is important since what is received is not necessarily a list of goals with an importance attached on each, but may possibly only be a prioritized list of goals. But this prioritization also occurs directly in the new version.

Since the redistribution step is removed the MP no longer needs to have the two modes "Preventive" and "Mission Specific". These were used to decide where to take points from and where to put them. These are not needed in the new version, since there is no redistribution. The goals' importance are distributed directly and passed to the RP.

The MP has the new IG "continentOwner" as input to gain information on whether or not a continent is fully occupied by a player since this information is now used to determine whether or not to defend or obstruct that continent.

For all goals there is a rule that if the AI is very close to winning (estimate above 0.98 within this round), goals involved in their mission are given 1.0 importance points. Else the points given is as in the following.

Conquer and Defend Goals

Mission specific goal are given a basic importance of 0.5 in addition to the number of points given if close to occupying the continent.

Conquer Continent Goals

These are distributed in the same way as "Conquer and Defend" if the AI's probability of winning within the current round is above 0.9.

Defend Continent Goals

Continents are only given a defend goal if they are fully occupied and not, as stated in the original design, if they are estimated to occupy it within the next five rounds. The reason for this is that AI turned out to play too defensively when considering what happens over five rounds. This also makes sense, since over five rounds, all players may be able to occupy most continents since they will receive reinforcements five times (which is at least three armies each time). A continent which is occupied by the AI is given 0.6 points. This number may seem high, but it is reasonable since occupying and defending continents is an important task in RISK. If the continent is mission specific to the AI, it will be giving an additional 0.4 points meaning that the total will be 1.0. This makes defending mission specific continents a very important task.

Obstruct Continent Goals

The obstruct goal is given if an opponent is fully occupying a continent. If the continent is estimated to be in the opponent's mission, the continent is given 0.3 points in "obstruct and defend". If the player occupying the continent is a player the AI must destroy, 0.2 points are given to the obstruct goal on that continent, since it is important that the AI's target opponent is not able to get more reinforcements if it can be avoided. If the continent is mission specific to the AI, there will be added an extra 0.2 on "obstruct and defend" to allow the AI to get a foothold in that continent. If the opponent is not close to winning, there will only be assigned "Obstruct and defend" points to a continent if it is mission specific to the AI (0.4 points). Else there will be assigned 0.3 points in "Obstruct" and another 0.2 if the opponent is the target of the AI's mission.

Attack Player

This goal will get 0.3 points assigned if a player is close to winning within the next five rounds. If the AI has a player as mission target, attack on that player is set to 0.4. If that player is estimated to win within the following round, the attack goal is given 1.0. If any player is close to winning, they will be given 0.2 points in the goal.

18 Territories

If the AI has this mission the goal is given 0.2 points. If he also is close to winning within the next five rounds, this goal is given another 0.3 points.

24 Territories

If the AI has this mission this goal is given 0.2. If he also is close to winning within the next five rounds, this goal is given 0.2 points.

A.5 RP: Calculate Attack Plan Cost

The board and MP goals are no longer given as input to this module. The board was never used for anything in the design of this module. MP goals were used to decide how many armies to leave on each territory in cases where the AI had the "18 with two on each". This information has been put on the attack plan type, since this is more efficient than taking the complete board into this module for this information only.

Formula 3.14 in [CJJ06] has changed from
$Cost_{estimated} = \sum_{t \in T_{plan}} \Gamma(A_t) + Cost_{min} \times A_{min}$
to
$Cost_{estimated} = \sum_{t \in T_{plan}} \Gamma(A_t) + Cost_{min}$

The multiplication with A_{min} was already the Formula for $Cost_{min}$ (3.13): $Cost_{min} = |T_{plan}| \times A_{min}$, which means that it should not be done again.

A.6 RP: Make Attack Plan

Instead of just making two attack plans (one with and without HamPath in the continent of the target territory), all possible combinations of attack plans are made. For each continent in the A* path, both a direct (A*) path and a HamPath is found through the continent. The reason for this is that the AI on its path to the target territory might move through another continent, or move out from a continent, that could be taken at a low cost. To avoid the AI not occupying these continents these plans are considered too.

A.7 RP: Place Armies

It may seem a little unclear in the original design how armies are placed in the beginning of a round. If a merged plan list (a list containing both attack plans and territories needing defense) exists, this is sorted by priority. The one with the highest priority will have its cost paid (if possible) and is removed from the list, if any more armies are left to place then the next in the list will have its cost payed and so forth until no more reinforcements are available. If the highest prioritized territory has the same priority as other elements in the list, the one with the lowest cost is payed first, then the next less costly and so forth. If the merged plan list is empty, either from the beginning or because every item in it have had their cost fulfilled, the armies are placed in the border territories of an occupied continent. If no continents are occupied, the armies are placed in random territories occupied by the AI.

A.8 RP: Prioritize Attack Plan

Only one of the original formulas for calculating a priority for an attack plan gave a satisfactory result, and the rest was therefore redesigned.

As written in the original design, a list of priorities are calculated. One for each goal given by the MP, stating how well the attack plan satisfies that goal. This list is then summed to get the resulting priority.

Conquer Continent

Formula 3.15 on page 112 in the original report holds and is unaltered.

Conquer and Defend Continent

This type of goal is covered by the above formula in the original report and not discussed further. But this needs to be extended a bit.

If the attack plan result in a conquered continent that can not be defended, then it does not satisfy this goal. This can be examined by simulating the attack plan, and then calculate whether or not the continent can be defended.

Simulating an attack plan is done by calculating the average army loss for each territory in the attack plan subtracted from the amount of armies in the starting territory. The average army loss is found by using formula 3.1 on page 90 in the original report. Afterward, when simulating the attack plan is done (and the continent has been conquered), the “Calculate Defense Cost” module is used on each of the border territories in the continent. This will calculate how many armies is needed for defending each of the border territories, which is easily compared with the actual number of armies located in each of those territories. If there is to few armies in just one of the border territories, then it can be inferred that the continent can not be defended.

Obstruct Continent

The original design argued that formula 3.15 also could be used for determining how well an attack plan satisfy the obstruct continent goal. This is not true. If an attack plan contains a territory in a given continent, then the attack plan satisfy the obstruct goal for this continent, otherwise not. The following formula describes this:

$$P_{plan,obstruct_continent} = min(1, |T_{goal} \cap T_{plan}|) \times P_{goal} \tag{A.1}$$

where T_{goal} is the set of territories in a given continent not owned by the given player. T_{plan} is the set of territories in the attack plan. P_{goal} is priority of the given goal.

Obstruct Continent and Defend Obstruction

Is not discussed in the original report, but done in the same manner as checking whether a conquered continent can be defended.

Kill Player

There is an error in formula 3.16 on page 112 in the original report. If T_{plan} conquers exactly all of a given player’s territories, then the formula will return zero: if $A_{target} == A_{plan}$, then the first term in the formula is zero. So the formula has been redesigned:

$$P_{plan,kill} = \frac{A_{plan}}{A_{target}} \times \frac{|T_{target} \cap T_{plan}|}{|T_{target}|} \times P_{goal} \tag{A.2}$$

Conquer 24 Territories

The original formula (3.17) did not state how well a given attack plan made the player come any close to conquering 24 territories as intended. It only gave higher priority the more territories the attack plan conquered, which resulted in longer attack plans getting better priority. The following formula returns a better result:

$$P_{plan,24territories} = min(1, \frac{|T_{plan}| + |T|}{24}) \times P_{goal} \tag{A.3}$$

Conquer 18 Territories

The original design for how well an attack plan satisfy this goal lacked the same expressive power as the above. The new formula is the following:

$$P_{plan,18territories} = min(1, \frac{|T_{plan}| + |T|}{18} \times \frac{|T_{p>1}|}{|T_{plan}|}) \times P_{goal} \tag{A.4}$$

where $T_{p>1}$ is the number of territories in the attack plan, where more than 1 army is left behind.

A.9 RP: Prioritize Territory Needing Defense

This RP module no longer takes the opponents number of RISK cards as input. The reason for this is that this information is already contained in "Opponent's next move".

A.10 RP: Transfer Armies

The priority of a path, P_{Path}, found from one occupied territory with extra armies to another occupied territory is:

$$P_{Path} = \frac{priority \times \frac{sourceArmies}{totalArmies}}{pathLength^2} \times multiplier$$

where *priority* is the priority the target territory's priority, sourceArmies is the number of armies in the source territory and totalArmies is the total number of the AI's armies on the board. The pathLength is the number of territories concerned in the path. The multiplier is a float used for tweaking the formula. The result of this formula is that it will make the AI's larger armies move towards territories needing defense if they are somewhat close to them. The higher the priority a territory has, and the larger the army is, the further away they may be to get a high score. However if the priority is high enough, the formula will be more willing to select territories closer to the target territory with fewer armies to quickly fulfill its need for reinforcement, even though this may be a short-term solution. However planning defense over more than 2-3 turns is not usually an option available in RISK, therefore we believe this formula holds.

Appendix B

Rules of Risk

RISK is a board game with three to six players. The board consists of 42 territories. Each territory belongs to one of 6 continents. In Figure B.1 the board of a RISK game is depicted.

Figure B.1: *The board in a RISK game.*

At the beginning of the game, territories are distributed equally and randomly between each player. Also, each player is assigned a secret mission that he must solve in order to win the game.

The possible missions are:

- Conquer Asia and South America

- Conquer Asia and Africa
- Conquer North America and Africa
- Conquer North America and Australia
- Conquer Europe and South America and a 3rd continent of your choice
- Conquer Europe and Australasia and a 3rd continent of your choice
- Occupy 18 territories with at least 2 armies in each territory
- Occupy 24 territories
- Kill black player
- Kill purple player
- Kill red player
- Kill yellow player
- Kill blue player
- Kill green player

If your own color is the color you need to kill, or if someone else kills that color, your mission changes to ”Occupy 24 territories”.

Initially each territory is occupied by a single army. Each player then gets a number of armies to place in the territories they occupy. This number is dependent on how many players there are in the game:

- 3 players: 35 armies each
- 4 players: 30 armies each
- 5 players: 25 armies each
- 6 players: 20 armies each

Then in turn the players place an army in one of their occupied territories until no more armies are available. This is called the *Initial army placement*.

When this is done, the game is ready to start.

A player’s turn consists of four phases:

1. Cashing in RISK cards
2. Defend
3. Attack
4. Fortify

For a better overview, Figure B.2 on page 119 shows the activities in the four phases.

B.1 Cashing in RISK Cards Phase

In the first phase, the player may trade in RISK cards for extra armies. RISK cards are earned from successfully conquering an opponent's territory. It is only possible to earn one RISK card in a turn. There are three types of RISK cards: a cannon cards, an infantry card and a calvary card. There exist one RISK card for each territory in the game. Each RISK card has the corresponding territory depicted on it.

It is possible to cash RISK cards when three or more cards are on hand. The traded sets have the following values:

- 3 Cannon: results in 4 extra armies.
- 3 Infantry: results in 6 extra armies.
- 3 Calvary: results in 8 extra armies.
- 3 Different: results in 10 extra armies.

A player must have a maximum of five RISK cards at a time, meaning he must trade cards before receiving the sixth.

If you own one of the territories depicted on the traded RISK cards, you may place two armies in that territory. This can only be done for one of the three RISK cards traded.

B.2 Defend Phase

In the defend phase, players also receive reinforcement armies for occupied territories. The number of armies are given the following way:

- The total amount of your territories divided by 3, with a minimum of 3 and rounded down (e.g. 14 territories gives 4 armies).
- If you own South America an extra 2 armies are earned.
- If you own North America an extra 5 armies are earned.
- If you own Europe an extra 5 armies are earned.
- If you own Africa an extra 3 armies are earned.
- If you own Asia an extra 7 armies are earned.
- If you own Australia an extra 2 armies are earned.

All armies are then placed in the territories of your choice before entering the next phase. This also include the armies gained from cashing RISK cards.

B.3 Attack Phase

The next phase is the attack phase. In this phase, which is optional, the player may try to conquer opponent players' territories. This is done by attacking from an occupied territory into a neighboring opponent territory. The attacker decides whether to attack with one, two, and three armies. The attacker use the amount of dice according to the number of armies he attack with (1 army = 1 die, 2 armies = 2 dice and 3 armies = 3 dice). The defender has the same choice, but can only defend with 2 armies.

An attack is done as follows:

1. The attacker decide where to attack (source and target territory).
2. The attacker decide whether to attack with 1, 2 or 3 armies.
3. The defender decide whether to defend with 1 or 2 armies.
4. The attacker and defender both roll their dice.
5. The dice of both attacker and defender are ranked according to value and are compared for each rank. If, in one rank comparison, the dice are the same, the defender wins in this rank.
6. The defender and attacker lose the amount of armies equal to the rank comparisons they lost.
7. The attacker then decide whether to attack again or to stop (retreat). If the attacker decides to attack, step 2-7 are repeated. If the attacker has no more armies to attack with, he has lost and must continue to attack somewhere else or move on to next phase.
8. If the attacker has won he has to move a minimum of X armies to the captured territory, where X is the amount dice he used in the winning roll.

When conquering a territory, the attacker must leave at least one army in the territory he attacks from.

B.4 Fortify Phase

In the final phase, which is also optional, the player may fortify a territory by moving armies from exactly one territory to a neighboring territory. He must leave at least one army at the territory he moves armies from.

When a player kills another player, he gets all the killed player's RISK cards.

B.5 RISK Activity Diagram

These four phases results in Figure B.2, which is an activity diagram for a player in a game of RISK.

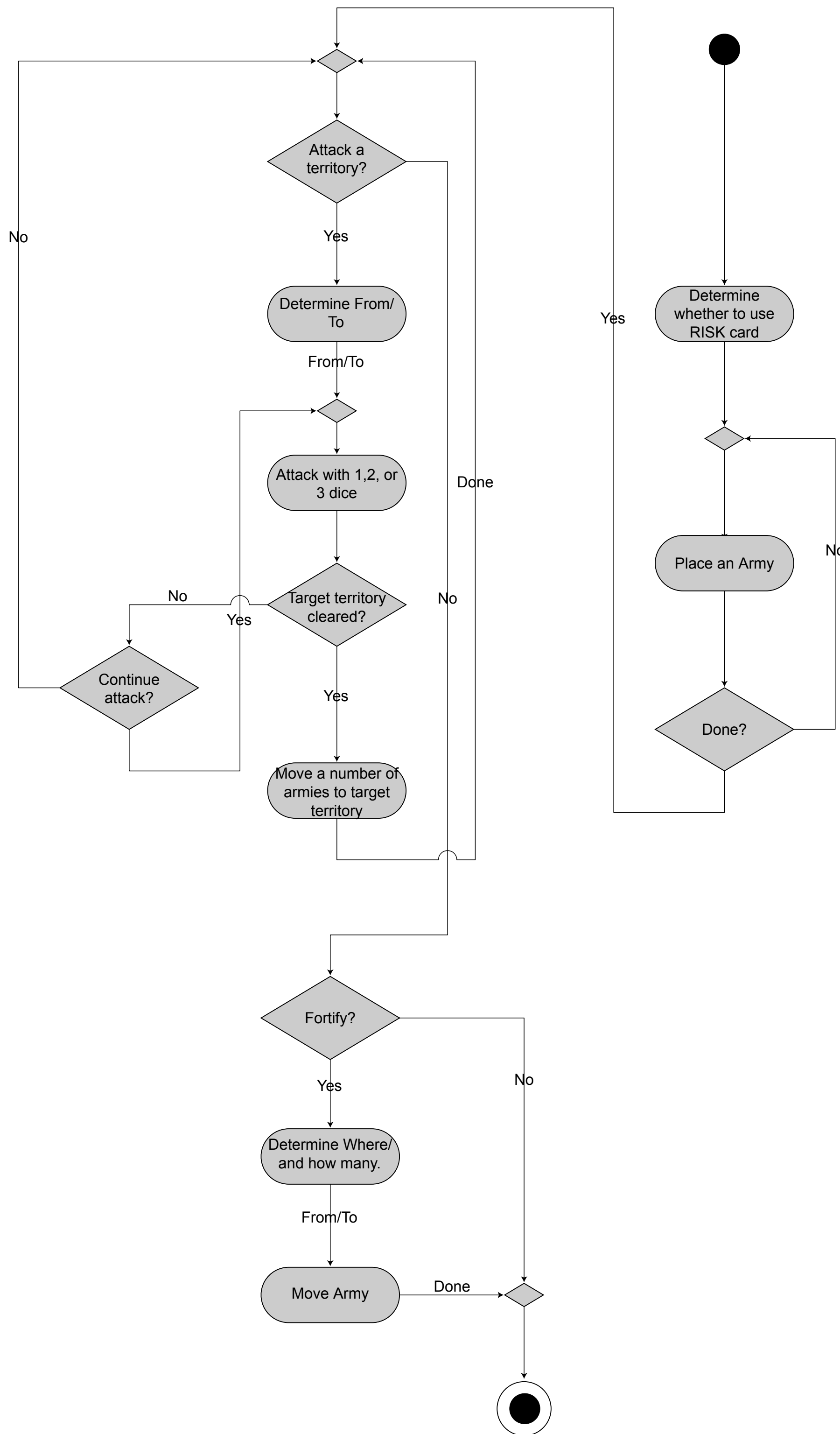

Figure B.2: *The activities of a player's turn in RISK. Diamond shapes are conditions and rounded boxes are actions. The starting point of the figure is the black dot. Phase 1 is the first action "Cash RISK cards" following the starting point. Phase 2 is the next action and following condition on whether the placement is done. Phase 3 is the next six actions and conditions which represent the "Attack" phase. The last condition and two actions are phase 4, "Fortify". The turn ends in the circle at the bottom.*

Appendix C

The Goals from the Master Prioritizier

The following will describe in detail the complete set af goals that the Master Prioritizer should decide upon.

Conquer and defend continent: This goal is meant as conquering a given continent and holding that continent a complete round, such that the AIs reinforcements increase, and perhaps complete a part of his mission. This is done by leaving sufficient armies on the border-territories towards neighboring continents. The difficulty of such a goal varies a great deal throughout a game. A player might only be a single territory away from completing this goal, or the given continent might belong completely to another player. There is one goal of this type for each continent in the game, a total of six goals.

Conquer continent: This goal is a lighter version of the above. It has no requirement of defending the conquered continent in the following round. This goal is most likely to be used in the completion of a mission, where the capturing of a continent is important, not the following defence. There is also one goal of this type for each continent.

Defend Continent: This goal can only be given if the continent has already been captured. The importance is weighted according to the importance of the continent: how many reinforcements does it pay, is it in my mission, etc. In most cases defending a captured continent is always important. There is also one goal of this type for each continent.

Obstruct continent: The meaning of this goal is: obstruct an enemy-controlled continent by capturing a territory in that continent. If an enemy player has captured a complete continent, it is in most cases important that he is not allowed to keep that continent. So capturing a territory in that continent should be prioritized as important. If some information giver has calculated that the given player is close to completing his mission, then it is even more important to obstruct one of his continents. There is also one goal of this type for each continent.

Obstruct continent and defend obstruction: If it is of importance that a player does not recapture a lost continent, or if another player is simply close to capturing a continent, it should be important that the AI reinforce armies in an occupied territory in that continent, so that the continent is not easily captured - thereby defending his obstruction. There is also one goal of this type for each continent.

Attack player: This goal is mainly used, when the AIs mission is to kill a given player. If he has captured some rewarding continent, and is well fortified, it might be time to begin hunting down his enemy and winning the game. This goal could also be used as a vengeance action. If a given player has annoyed him throughout the game, it might result in an interesting game play, if the AI suddenly starts to retaliate. The goal can also be used if the AI discovers that a weaker player has many RISK cards. Prioritizing this goal high would encourage the AI to attack and destroy this player to get his RISK cards. There is one goal of this type for each player in the game, and since there can be up to six players this gives six goals.

18 territories with at least 2 armies: This goal is only used, when the AI has the mission *Occupy 18 territories with at least 2 armies on each territory*, and the MP has calculated that the AI is close to fulfilling this mission.

24 territories: This goal is only used, when the AI has the mission *Occupy 24 territories*, and the MP has calculated that the AI is close to fulfilling this mission. It is implied, that there is no restriction of 2 or more armies on each territory - at least 1 army on each territory is needed.

In Figure C.1 one can see a depiction of the information flow from and to the Master Prioritizer. The *MP* corresponds to the Master Prioritizer, and IG_{1-5} is the Information Givers, that provide indirect information to the MP. The outward edges from *MP* are the different weights given to each goal. The *Co&De* is the *Conquer and defend Continent* goal, the *Co* is the *Conquer Continent* goal and so on.

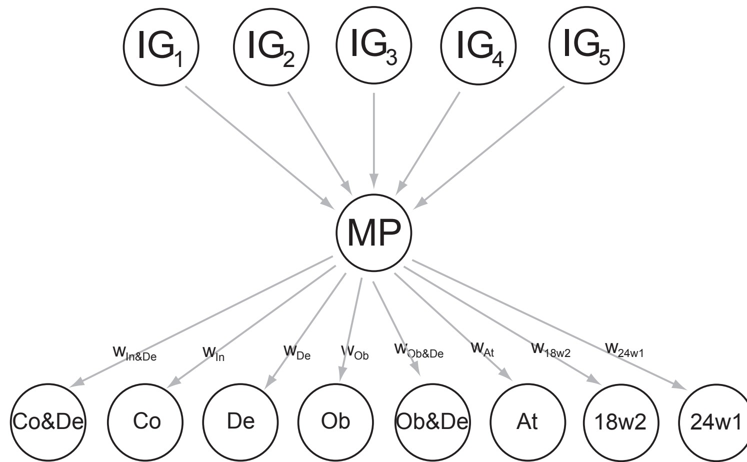

Figure C.1: *The Master Prioritizer MP, receives some information from the Information Givers IG_{1-5} and outputs a weight to each of the goals.*

Appendix D

Trainer Data

This appendix will give an explanation on how the various training data files look.

D.1 Neural Network

The training data file format for neural networks has been designed for this project. Essentially the format must contain the following:

- How many input, hidden and output nodes the network consist of.
- A series of training example with input values and corresponding target values.
  - Input values for a single training example is a list of real values. Each value correspond to each input node.
  - The target values for the same training example also consist of real values. There is a target value for each output node in the network.
- Also, for each training example, there is a little remark on whether or not this example is produced by a winner or a loser.

An example of such a file can be seen in Figure D.1. The extension of such a file is ".nntd".

D.2 Decision Trees and Naive Bayes

Since the decision tree and naive Bayes classifiers are both a part of Weka, the training data file format for these is the same. Weka uses the ARFF file format [wek] which is a text file that describes the attributes of the training example. A simple example of an ARFF file can be seen in Figure D.2. The example is a simplified version of our Round Planner module "Score Attack Plan" which based on the attack plan's estimated cost and priority determines a score for it.

```
@input_nodes 5
@hidden_nodes 3
@output_nodes 2

@data
W{0.1,0.4,0.56,0.7,-0.5},{1.0,0.4}
W{0.4,-0.6,0.3,-0.4,-0.1},{-0.3,0.6}
L{0.51,0.34,0.6,-0.75,0.55},{0.9,-0.3}
L{0.63,0.34,-0.56,0.73,0.1},{0.12,0.45}
```

Figure D.1: *A simple example of a training data file for neural networks (nntd).*

```
@relation RP_ScoreAttackPlan

@attribute CostEstimated {1, 2, 3, 4, 5}
@attribute Priority {0.0, 0.25, 0.5, 0.75, 1.0}
@attribute Score {0.0, 0.25, 0.5, 0.75, 1.0}

@data
1,0.5,0.75
5,0.25,0.5
2,0.5,1.0
```

Figure D.2: *A simple example of a training data file in ARFF format.*

The ARFF file consists of two parts: First a header part which describes the attributes and their states then followed by the training data itself. The name to be associated with the training data is specified after the keyword *@relation*, in this case “RP_ScoreAttackPlan”. Each attribute is then specified with *@attribute* followed by its possible states separated by commas. In this example the “Priority” attribute has five states between 0.0 and 1.0. The last attribute that is specified will automatically become the target attribute, in this case “Score”.

The keyword *@data* marks the start of the data section. Each training example consist of the observed values of each attribute separated by commas. In this case, the following attribute states have been observed in the first training example: “CostEstimated” is 1, “Priority” is 0.5, and “Score” is 0.75. The rest of the file will be line by line of training examples.

D.3 Bayesian Networks

Figure D.3 shows the contents of a training data file as required by the EM algorithm in Hugin. The first entry is a list of all the nodes we wish to learn on in the network. These are separated by commas. The following lines specifies what state is observed on that specific node in each training example. The entries here are also comma separated. If there is no observation on a node, it is entered as "N/A".

```
Node1,Node2,Node3
S1,S3,S6
S2,S3,S5
S1,N/A,S5
S3,S2,S4
```

Figure D.3: *The structure of the training data files used by Hugin. The first line is the name of each node in the BN. The following lines are the observed states for each node. The "N/A" state is used to specify that no observations on that given node is seen in the current training example.*

A thing to make sure of when creating this file is that the nodes mentioned in line one are in fact present in the network. Hugin will not load the file if some nodes are not present. A more serious thing to be certain about is that the states observed in the file are in fact present in the node to which they should belong. Hugin gives no warning if it encounters a state not present in the node. This means that if the states in the file differs from the ones in the net it will not be discovered before the EM algorithm has terminated, and even then it can only be discovered through faulty network behavior. This is highly problematic when working with very large training files.

Appendix E

Testing Algorithm

This appendix contains an explanation of an algorithm for generating a test schedule.

E.1 Algorithm for Generating a Test Schedule

A schedule from a set of different AI techniques, for instance {script, nn, dt, bn}, can be done as follows:

1. Iterate over the value $i = [1; 2^s[$, where s is the size of the technique-set.
2. Convert the value i to a bit string and add a technique where the bit is on. For example: bit string = "1010" would result in a game with two players, where the given module is implemented as a scripted and as a decision tree respectively. If the value i = 7 ("0111") the result would be a game with three players. First a player with the given module implemented as a neural network. Then a player with the module implemented as a decision tree. And lastly a player with the module implemented as a Bayesian network.

Appendix F

Test Suite

The performance test of all the techniques in each module in the framework were performed in 18 days. The range of computers used in the tests can be seen in Table F.1. Some of the system were either *dedicated* (D) to performing the test, *dedicated except normal usage* (d) or *shared* (s) with other users, which is most typical for public servers.

Name	OS	CPU	RAM	Status
fire1	Solaris 9	8x900 MHz (SPARC)	31768 MB	s
fire2	Solaris 9	2x900 MHz (SPARC)	4069 MB	d
homer	RedHat Linux 3	2x2.8 GHz (Intel x86)	4096 MB	s
pelle	Win XP	1.8 GHz (AMD Athlon 64)	1024 MB	d
edblab1	Win XP	2.8 GHz (Intel Pentium)	512 MB	D
edblab2	Win XP	2.8 GHz (Intel Pentium)	512 MB	D
edblab3	Win XP	2.8 GHz (Intel Pentium)	512 MB	D
gr.room1	Win XP	2.8 GHz (Intel Pentium)	512 MB	d
gr.room2	Win XP	1.1 GHz (AMD Athlon)	512 MB	D
jens	Win XP	1.9 GHz (AMD Athlon XP)	768 MB	D
brian	Win XP	3.2 GHz (Intel Pentium)	1024 MB	d
laptop1	Win XP	1.8 GHz (AMD Sempron)	1024 MB	d
atle	Win XP	2.0 GHz (AMD Athlon XP)	768 MB	d

Table F.1: *The suite of systems running the performance tests. The multi-processor systems ran more tests simultaneously. since JRisk only runs in two threads, which would only utilize two of the processors on the given system. The status coulmn show the status of dedication for each system. "D" stands for total "dedication" to running the test. "d" stands for "dedicated except for normal usage". "s" stands for "shared" with other users, which is most typical for pblic servers.*

Appendix G

Enclosed CD

This report includes an enclosed CD. This CD contains the following:

JRisk: The actual game, where one can play against the different implementations of the AI framework in a game of Risk.

ai-data: The trained models for each technique for each module. This also includes a sample of the training examples from 10 games. This can be converted and trained on each of the techniques.

Training Data Converter: This is the converter.

Trainer: And the trainer.

report.pdf: This report in a digital version.

If the CD is not enclosed or the report is a digital version, the content of the CD can be found through the following link:

http://www.pellecoltau.dk/master-thesis/cd.zip

G.1 JRisk

This is the implementation of the board game Risk. It includes different implementations of the AI framework discussed in this report. The following AIs can be selected in the game:

Easy: The easy AI already implemented in JRisk.

Hard: The hard AI already implemented in JRisk.

Extra hard: The new hard AI implemented in a newer version of JRisk.

Scripted Framework: Creates a purely scripted implementation of the framework.

Best framework: Creates the best implementation of the framework, which were found through the research documented in this report.

Custom framework: Creates an implementation of the framework, which is leaded from the file "custom_framework.txt". This makes it possible to create your own custom framework. Please be aware that not all techniques can be used in all modules. Please consult Table 5.2 on page 69. Please also notice that Bayesian networks (BN) only works if Hugin Researcher [AOJJ89] is installed on the computer. Therefore, the best AI does NOT use BN in the module "IG Continent", but instead a neural network trained with winners only, which is the second best technique for that module.

JRisk can be run in different with different user interfaces: Swing (graphical) or command line.

Command Line Interface

The command line interface is pretty self explanatory. But the following commands are important:

newgame: Starts a new game.

newplayer: Creates a new player. The usage can either be "newplayer human [color] [name]" or newplayer ai [ai-type] [color] [name]". The possible entries for "ai-type" are the following:

- **easy:** The easy AI..
- **hard:** The hard AI.
- **extrahard:** The extra hard AI.
- **framework:** The purely scripted framework.
- **framework_best:** The best framework.
- **framework_custom:** The custom framework, loaded from the file "custom_framework.txt".

startgame mission: Begins the game. The AI only works with mission types of Risk games.

Swing Interface

In the graphical user interface (GUI), one can select different AIs. These AIs corresponds to the AIs described above. The GUI is quite unstable and might crash when selecting players. **The authors of this report is not responsible for instability in third party software**. Also, there is "feature" when playing the game: In the AIs turn, the board might turn all red, blue or some other color for a small second - this is only a small insight into the decision making process of the AI, which is caught by the GUI.

G.2 Training Data Converter

The Training Data Converter will convert the game history located in "ai-data/training_examples/" into training data for given technique in a given module.

The usage of the converter is explained by running “explain_usage.bat”. The converted data will be located in “ai-data/training_examples_converted/”. Please consult Table 5.2 on page 69 to see which techniques can be converted in which modules.

G.3 Trainer

The Trainer will train a given technique in a given module. The trainer will train the technique from the data converted by the Training Data Converter. this trainer can only be used to train neural networks, decision trees and naive Bayes classifiers. Bayesian networks are trained used Hugin Researcher [AOJJ89]. Again, consult Table 5.2 on page 69 to see which techniques can be trained in which modules. Please note that the trained modules located in “ai-data/ trained_models/” will be overridden if the trainer is run. The models already located in “ai-data/trained_models/” were trained from 3275 games, and only 10 games are distributed with this CD.

If any of the two BNs are trained, please copy them to the root of the JRisk folder, otherwise the Hugin API can not find them.

Appendix H

Summary

In commercial games released today the majority of AI opponents are designed and implemented using scripts. This leaves a lot of design choices to the designer, but it also means that as the size and complexity of the game increases, so does the chance that the AI designer overlooks some detail in the aspects of the game. An obvious solution for this would be to use learning AIs to handle different tasks within a game, and thereby relieve the AI designer, since the learning AIs would learn the details by itself.

Left is now to choose which learning AI techniques should be chosen for the various tasks commonly found in computer games. There are a lot of possible techniques to choose from, but which ones suit which tasks the best?

In this report we test how well different learning AI techniques compare to each other in solving various tasks in computer games. The learning AIs used are neural networks, decision trees, naive Bayes classifiers, and Bayesian networks.

An AIs performance is measured in terms of time spent solving its task, and its success rate. These terms have been chosen since they cover both the AI's system requirements as well as its usability in solving a given task. The test is done through the use of an AI framework which we have designed in a previous project [CJJ06] and implemented in this project period. A brief description of the previous project is included in this report. The AI framework is designed to play the board game Risk. It is modular and covers different tasks involved in both estimating uncertainties and planning. It has been designed such that at each module it is possible to exchange a technique implementing it with another technique.

We have implemented a scripted AI for playing Risk which will serve as a basis for generating training data for the learning AIs. We cover various aspects of training and find that it possible to use data for both winning and losing scripts to generate training data. Not only do we find it possible, but we also believe that learning from both winners and losers, will make the training of the learning techniques somewhat faster.

We use various tools to aid us in training the different models. These are Hugin which is a tool for building, training, and querying Bayesian networks, and Weka for training and querying naive Bayes classifiers and decision trees. We implement neural networks with the back-propagation algorithm ourselves.

We consider how to compose an AI that implements the best performing technique on each module. It is not possible for us to simply let all AI compositions play against each other since it is possible for us to make over 10 million possible AI compositions. We therefore come up with an approach we call "Best Module" approach which severely lower the games that needs to be played. This approach uses the "pure" scripted AI composition, where all modules are scripts, in all games. The only module not being scripted is the one being tested on. For this module the scripted AI will be replaced by each of the different techniques found to be usable on this module. When all modules have been tested this way, the best technique from each module are used to form the "Best AI".

We argue that our framework, designed for Risk, is general enough to handle other games. This is done through showing that it is possible to divide Risk into two sets of goals that it must prioritize in order two win: offensive and defensive goals. We show that exactly these goals are also present in games of other genres. With this said, we are able to argue that an AI technique which is good at solving a task within the framework in Risk is also good at solving this task within other games.

The main results of the report regards the generality of the specific AI techniques:

- Neural networks are useful for making fast decisions when given a large amount of input. This makes it very useful in planning, where even scripts may take too long to process the data.
- Decision trees are also useful in situations where time matters. However,the input to these needs to simple in terms of few input variables for the trees to be effective. When there are too many input variables, training the trees becomes problematic and therefore they become imprecise.
- Naive Bayes classifier perform decently within all task in both performance and query time. Therefore this model is qualified for most tasks within the framework.
- Bayesian networks are not useful for time-critical tasks in computer games. The propagation of the networks are simply to slow for this. Therefore BNs are mainly for use in long term information gathering tasks or planning in turn based games.
- Scripting is useful for every task. This is shown through the implementation of the scripted AI. However, scripting takes far more time to implement than using learning techniques. Also the scripts are not capable of learning, meaning that they are trapped in the script designers way of thinking.