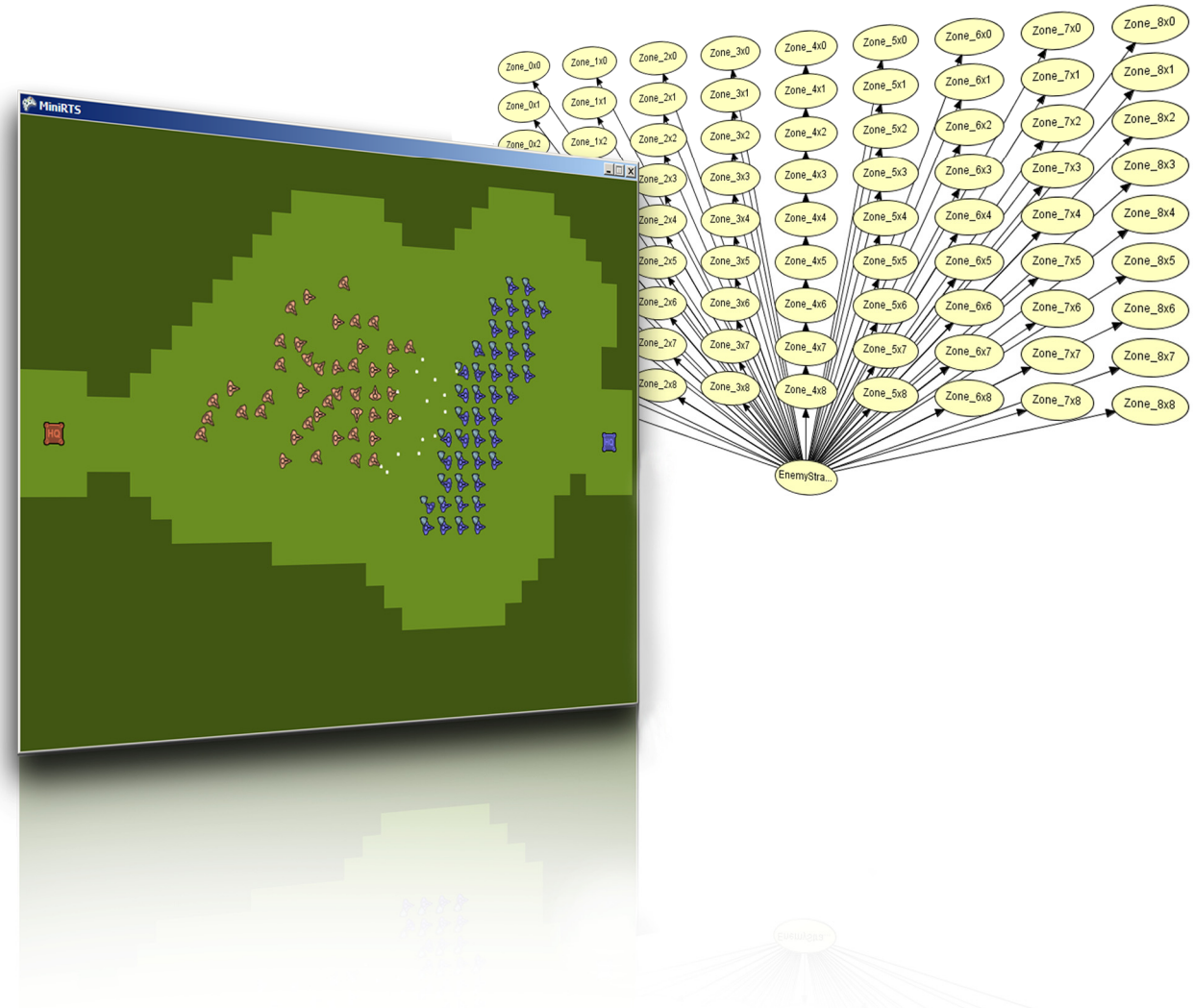
AALBORG UNIVERSITET

MiniRTS



Strategy Identification using Bayesian Grid Models

by Mads Dancker-Jensen, 2008

AALBORG UNIVERSITY

Department of Computer Science
Selma Lagerlöfsvej 300
http://www.cs.aau.dk

Title:
MiniRTS: Strategy Identification using Bayesian Grid Models

Theme:
Strategy Identification

Semester:
SW10, 1. Feb. - 14. Aug. 2008

Group:
d632c

Members:
Mads Dancker-Jensen

Supervisor:
Yifeng Zeng

Copies: 4

Report - pages: 41

Appendices: 2

CD: 1

Total pages: 52

Abstract:

This report describes the investigation of using *Bayesian Grid Models* for reasoning in RTS games.
The first part of this report documents the implementation of the environment used to study the performance of the *Bayesian Grid Model*. The environment is a RTS game called *MiniRTS* which is similar to the commercial games but has been simplified to fit the purposes of this project best.
In the experimentation phase of the project, different scenarios have been created with the purpose of comparing different models to see which factors could improve the performance.
Finally the report concludes and evaluates upon the results, and describes which improvements could be made in the future.

Preface

The following report is written during the fall of 2008 by one Software Engineering student at the Department of Computer Science at Aalborg University.

When the words *we* and *our* are used, it refers to *the author* of this report and *he* refers to *he/she*.

When code is presented, it may differ from the actual source code. It may have been modified and have had some details removed to make it fit into the report. This has been done to heighten the legibility of the code, and to focus on essential functionality.

The first time an abbreviation is used, the entire word or sentence is written, followed by the abbreviation in parentheses. Throughout the rest of the report, the abbreviation is used.

It is expected that the reader has basic knowledge of software engineering, and computer games.

Mads Dancker-Jensen

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Real-Time Strategy (RTS) games is a popular genre of computer games and it is in the best interest of the industry to develop more interesting and entertaining RTS games. One area of developing a RTS games involves developing Artificial Intelligence (AI) and since a successful RTS must have good AI the game developers has put more attention on in this area.

RTS games is a category of computer games where the players have to think out strategies to beat the enemy. Such games are often war games where the player can see the battlefield form top-view and controls troopers, tanks and other kinds of military equipment. Real-Time means that the game is not turn-based, so the players do not have to wait for other players to end their turn.

Most RTS games have Computer Controlled Players (CP) to play against as a substitute for human players. But most of the CP is not satisfying because of various problems. In some games the CP is extremely stupid, due to poor AI, while others are extremely hard to beat. Not because of an intelligent AI, but because they cheat.

It is important to distinguish between the high-level and low-level (also known as unit-level) AI in RTS games. The unit level AI is responsible for navigation such as finding the shortest path from A to B and avoiding obstacles. It is also responsible for identifying enemies and attacking. The high level AI corresponds to the CP, which is responsible for choosing strategies and carries them out by grouping unit and issue orders.

This project will try to look into the subject of creating a high-level AI for RTS games that will be able to identify the opponent's strategy. This project

Figure 1.1: Screenshot from the popular game StarCraft [3]

will focus on how to do that using Bayesian Networks, or more specific a model which in this project is called the Bayesian Grid Model. Since RTS games are often very complex, and have many different variations of possible strategies, we will in this project create a mini game, which only represents a fraction of the commercial games. This mini game is called MiniRTS.

1.1 Related Work

In general it is in the last few years that attention has been directed at AI in RTS games, though most research has focused on the low-level, e.g. by improving unit cooperation [10], making the units learn from experience using and more intelligent in general, while less focus has been at the high level AI.

It is very hard to find research material no non-cheating high-level AI in RTS games, mainly because is a relative new research topic [2] and also because of the commercial interest, where implementation are closed source. A lead producer of upcoming sequel of StarCraft, StarCraft II [4] speaks out they are developing a non-cheating AI [6] for the game.

The low amount of research in this topic only makes our project more

interesting.

1.2 Content

The following list shows the content of this report:

- **Chapter 2 - Mini Game:** A description of the game which is used as test environment in this project.
- **Chapter 3 - Techniques:** A description of the techniques used in this project.
- **Chapter 4 - Approach:** A description of the idea and concepts which is going to be tested.
- **Chapter 5 - Experiments:** Explains the experiments and afterwards shows the results.
- **Chapter 6 - Conclusion:** Evaluates and conclude upon this project.

Chapter 2

Mini Game

This chapter will describe the mini game which is the environment for studying the use of Bayesian Networks to identify strategies.

Since many commercial games are closed source, we either has to use an open source RTS or create our own. There are some open source RTS available for downloading e.g. ORTS [7], but they would need too much modification for the purpose of this project. Instead a new RTS game is created for this project.

The game we want to create should be similar to the commercial games to be able to prove that this project could be used in such games. But we also have to consider simplifying the problem so we can focus on the important part.

The name of the game created in this project is *MiniRTS*. This chapter will first descript the rules of MiniRTS. Then there will be a short introduction of the development environment used to implement MiniRTS, and last the implementation where the most important parts of the MiniRTS are described. This chapter will only explain the game. The AI part of MiniRTS will be explained in the Approach chapter on page 20.

2.1 Rules

This section describes the rules of the game. The basic rules should be similar to the rules of the most popular commercial RTS games, but some parts are cut away since they are irrelevant for this project. In many RTS games the player has to build a base, gather resources and produce units for

an army. These parts are removed form this game. Instead the player starts with an army and cannot build new units. To minimize the complexity of the game there are no obstacle on battlefield. The goal of the game is to destroy the enemy Head Quarter (HQ) or all the enemy units.

Like most other RTS games, MiniRTS also have a unit level AI which is responsible for the behavior of the units. The units can be regarded as small agents with sensors and are able to gather information about the environment and act accordingly. The players of the game can issue orders to the units belonging to their team. If the player orders a unit to move form location A to B, it is the units responsibility to move form A to B while avoiding obstacle. The unit is also responsible for attacking enemy units if they are within range of fire.

In some games units will not always obey orders, e.g. if a unit is under heavy attack, the unit could go in panic mode and run away. In MiniRTS units will always obey orders even though it is a suicide mission. There are four main types of orders which can be issued to the units:

- **Move:** This order tells the unit to move to a new location.
- **Attack:** Given an enemy unit, the attack order tells the unit to move so the enemy unit is within range of fire and attack it.
- **AttackMove:** The AttackMove order tells the unit to move to a new location like the Move order, but while moving the unit should attack any enemy unit it spots and destroy it.
- **Defend:** The defend order tells the unit to stand still and defend the position. The Defend order is the default order which means that if the unit has been issued no other order it will be defending.

Fog of War (FOW) is an important aspect of many RTS games which are a layer of dark clouds covering the battlefield. The player can only see the areas of the battlefield where he has units placed. This means that it is often necessary to send units out to scout to be able to identify the enemy strategy. MiniRTS can also be played with Fog of War enabled.

2.2 Development Environment

MiniRTS is implemented using the XNA Framework developed by Microsoft where the target language is C#. XNA is a collection of tools with the intention of making game development easier and less time consuming. The XNA Framework provides commonly used data structures, classes and methods which are the foundation of most of today's games. [8]

2.3 Implementation

This section describes the implementation of MiniRTS where the most important classes are described.

2.3.1 GameObject

GameObject is the base class of all objects in MiniRTS which is a part of the game, or either has to updated in the game loop or drawn to the screen. The game contains a collection of active game objects *(GameObjectCollection)* and all *GameObjects* added to this collection will automatically be updated and/or drawn. The *GameObjectCollection* also ensures that adding and removal of *GameObjects* does not conflict with the update and draw process.

2.3.2 Actor

In a RTS game it is necessary that object moves in Real-Time e.g. if an object has to move to a new location it has to move towards it with a certain speed and not pop to that location instantly. The Actor class is responsible for such behavior and implements two methods:

- **MoveTo:** This method takes a destination and movement speed as input, and tells the actor to move towards the destination with that speed. The actor does not stop moving until it has reached the destination or a new destination is given.
- **RotateTo:** This method takes a rotation target and rotation speed as input, and tells the actor to rotate towards the rotation target. Since there are two ways to rotate (clockwise and counter clockwise) it will always pick the shortest.

The Actor class inherits form *GameObject* so it can be drawn and updated.

2.3.3 TileMap

Tile maps have many purposes. In MiniRTS the *TileMap* class also is used for different purposes and therefore the class is made generic so it can contain different types of objects. The *TileMap* is very similar to a two dimensional array but with added functionalities and a definition of the tile size. These functionalities allow different ways of getting access to the tiles, e.g. getting all tiles inside a certain rectangle.

2.3.4 Unit

Units are the main element of RTS games and which in war games often troopers and/or tanks. In MiniRTS the *Unit* class is the base class of all the different unit type classes (e.g. troopers and tanks) and since units should be able to move around it inherits from the Actor class. The Unit class has a collection of different variables which can be modified in the inherited classes so we can create a variety of units, e.g. a weak but very fast unit or a slow but powerful unit.

List of variables:

- **MovementSpeed:** Defines how fast the unit can move around.
- **RotationSpeed:** Defines how fast the unit rotates.
- **HitPoints:** Defines the number of hit points. If the unit reaches 0 hit points the unit has been killed/destroyed.
- **HitPointsMax:** Defines the maximum number of hit points. E.g. strong armored unit like a tank will have a large maximum amount of hit points while a weak unit like a trooper will have a low maximum amount of hit points.
- **Armor:** Defines how much armor the unit has. The armor absorbs some of the damage given to the unit. E.g. if a unit with 15 in armor is hit by a projectile which does 50 in damage, the unit will only loose 35 health points.

- **SightRadius:** Defines how far the unit is able to see. This has to do with Fog of War, where the players of the game are only able to see areas of the map where they have units placed.
- **FireRange:** Defines how long the unit is able to shoot.
- **CoolDownTime:** Defines how long it takes before the unit is able to fire again.

Besides all these variables which allow us to create different unit types, the *Unit* class is also responsible Navigation and Issuing Orders.

2.3.5 Pathfinding

In order to make the units able to navigate around MiniRTS have a Path Finding module. The Path Finding module contains tree classes:

- **PathNode**
- **PathNetwork**
- **AStarAlgorithm**

To be able to use the A* algorithm [5] for navigation a network of path nodes is needed. The solution that fits this project best is simply to align the path nods in a grid since the game is grid based. So the *PathNetwork* inherits from *TileMap* and each tile contains a *PathNode*.

Since the units should not be able move trough each other, the Unit class has an IsMovePossible method which is called by the AStarAlgorithm to tell if a move from one cell to another is possible. In this way units can move around each other. But "dead-lock" situations can occur e.g. when two unit moving in opposite directions blocks the path of each others. One solution to this problem could be Cooperative Pathfinding [10] where the group will find the overall best path to a new location. But this seems to be overkill for the purpose of this project so we stick to an ad-hoc local repair A* method where "dead-lock"-situations is solved when they occur, which should be enough for this project.

2.3.6 Orders

In a RTS game the player should be able to issue orders to the units e.g. if a unit should move to a location or if should attack an enemy.

All orders are classes which inherit from the *Order* class. The *Unit* class has a method called *IssueOrder* which take an order as input.

The *IssueOrder* method is meant to be overridden in inherited classes so special types of units can receive special orders to execute their unique task. E.g. consider a unit which is able switch to stealth mode it would have to receive a stealth-order to activate that mode. Other types of units which are not able to execute that order would just ignore it.

```
1  class Unit
2  {
3      public virtual void IssueOrder(Order order)
4      {
5          if (order is Move)
6          {
7              MoveTo((order as Move).Position);
8          }
9      }
10 }
11
12 class StealthUnit : Unit
13 {
14     public override void IssueOrder(Order order)
15     {
16         if (order is StealthMode)
17         {
18             ActivateStealth();
19         }
20
21         base.IssueOrder(order);
22     }
23 }
```

Listing 2.1: Order processing for units

2.3.7 Groups

In the game, it is necessary to be able to giver orders to a group of units, and then be notified when all the units in the group have done executing the order.

The Group class is responsible for this task the units can be added and removed from groups. Units can be in multiple groups at the same time.

2.3.8 Fog of War

As Fog of War is an important aspect of many RTS games, MiniRTS have a *FogOfWar* class which implements this concept. The *FogOfWar* class is a *TileMap* containing integer values and has the same size as the battlefield. Each integer in the *TileMap* tells if the corresponding cell on the battlefield is visible or not. If the integer is equal to zero the cell is visible, and if it is greater than zero it is not.

Each team has a Fog of War layer associated with them, and only the units on that team can influence the layer. When units move around they subtracts one from all cell in the *FogOfWar* tile map which are inside their sight radius, and then adds one to all cells at the new location. Figure 2.1 shows two units (circles) and how the influence the *FogOfWar* tile map.

2.3.9 Scenario

To enable testing it is necessary to be able to setup scenarios. A scenario is a definition of how many teams are in the game, the number of units, the type of units, where the units are placed, initial visible areas on the FoW layer, and also additional rules besides the ordinary rules of the game.

The *Scenario* class has the following methods:

- **Initialize:** The *Initialize* method is used to initialize all needed components of a scenario, e.g. initialize the players in the scenario.
- **Update:** The *Update* method is used to update the components in the scenario, which are not updated elsewhere.
- **Draw:** The *Draw* method can be used to draw additional content on the battlefield, e.g. if extra information about the units has to be shown.

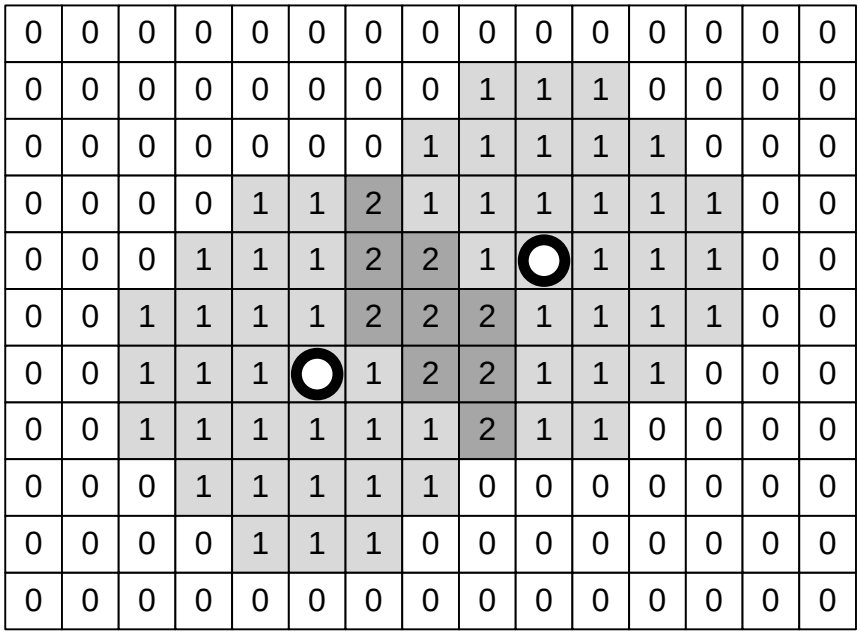


Figure 2.1: Fog of War

- **Load:** The *Load* method can load the battlefield settings from a file. These settings are the teams in the game, the position of each unit and so on.
- **Save:** The *Save* method saves the current battlefield settings in to a file.
- **ResetCondition:** This method can be used to tell when the scenario should reset itself. The implementation of this method should return true if the scenario should be reset, otherwise is should return false.
- **Reset:** This method is call when the reset conditions are fulfilled. It is used to set up the scenario when it is reset.

Most of these methods should be implemented in inherited scenarios, except the *Load* and *Save* methods.

Chapter 3

Techniques

This chapter describes the techniques used in this project which are Bayesian Networks and Naive Bayes Classifier, and is written using primarily [9], [11] and [12]

The reason why Bayesian Networks is chosen for this project is because they can be used to simulate human-like reasoning. The computer player of this project should be able to make choices based on information about the opponent, but it should do it in a non-cheating manner where it only has access to the same information as a human player. Consider a game where full information of the opponent is not available due to e.g. Fog of War. In such game one strategy could be to fool the opponent by attacking with a small group of units, while at the same time secretly building a huge army outside the range of the enemy sight. A human player would direct attention at the attacking units and therefore not notice the secret army, but a cheating computer player would be aware of it.

Bayesian Networks are well suited for this task because the can be used to make decisions even though all information is not available. Identification of the enemy strategy is a classification problem and among the most practical approaches to certain classification problems is Naive Bayes Classifier.

Naive Bayes Classifiers has proven to be one of the most consistently well-performing set of classifiers.

The two following sections describe Bayesian Networks and Naive Bayes Classifiers.

3.1 Bayesian Networks

Bayesian Networks is a graph model for probabilistic reasoning which represents a set of variables and their probabilistic independencies. It is used for reasoning under some uncertainty e.g. in medicine where it is often used for diagnostics of a patient. If the network is given the symptoms of the patient it can compute the probability for various diseases that the patient could have.

Formally, a Bayesian Network is a directed acyclic graph (DAG) where the nodes are variables with a finite set of states. The arcs between the nodes represent dependencies. Figure 3.1 shows an example of a Bayesian Network where the sprinkler and rain can cause the grass to be wet. The rain can affect the sprinkler because if it has not rained for a while, the sprinkler has to be activated.

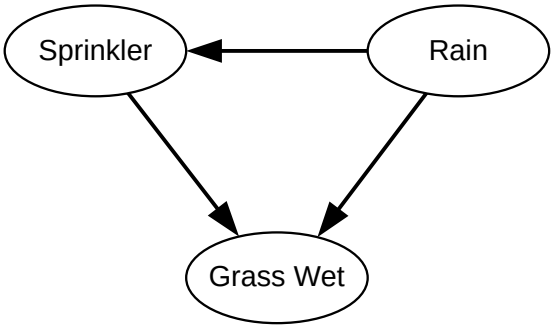


Figure 3.1: Serial Connection

3.1.1 Bayes Theorem

When working with Bayesian Networks we are often interested in determining the best hypothesis, given some observed data, e.g. if we can observe that the sky is cloudy the best hypothesis might be that it is going to rain.

Let the space of hypotheses be called H where h is on specific hypothesis and D represents the data that can be observed. h may have some initial probabilities even though no data is observed, e.g. even though the sky has not been observed, the chance for rain is 35%. These initial probabilities is often called *prior probabilities* and is denote $P(x)$ where x is the hypothesis. If we want to write the probability of something given something else, it is denote $P(x|y)$ which means the probability of x given y, also called the *pos-*

terior probabilities. E.g. the probability for rain given that we can observe clouds, might be 70%.

All this leads us to the Bayes Theorem which is the foundation of Bayesian Networks. Bayes Theorem is important because it provides a way to calculate the posterior probability of a hypothesis given some data, $P(h|D)$, using the prior probability $P(h)$ together with the probabilities $P(D)$ and $P(D|h)$.

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

3.1.2 D-Separation

Representing dependencies is an important property of Bayesian Networks. Dependency means that the probabilities of one variable can influence the probabilities of others. In this way, information can flow through the network. If two variables A and B are connected through a third set of variables and information cannot flow between them, A and B are said to be d-separated. If A and B are not d-separated they are said to be d-connected. Probabilities can be changed due to some knowledge about a variable. When the state of a variable is known it is called evidence. The following shows the three types of connections where d-separation can occur.

Serial Connection

Figure 3.2 shows a Serial connection where C is dependent on B which is dependent on A. Information can flow through a serial connection unless there is evidence on B. If there is evidence on B then A and C are d-separated and they cannot influence each other.

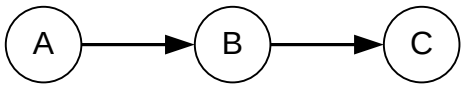


Figure 3.2: Serial Connection

Diverging

Figure 3.3 shows a Diverging connection, where B and C are dependent on A. Information can flow between B and C unless there is evidence on A.

Evidence on A, d-separates B and C.

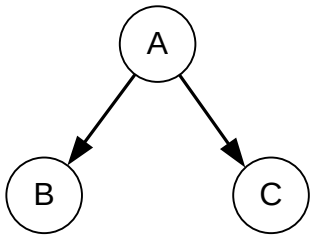


Figure 3.3: Diverging Connection

Converging

Figure 3.4 shows a Converging connection. In a Converging connection no information can flow between A and B. But if there is evidence on C or one of its children A and B are d-connected and information can flow between them.

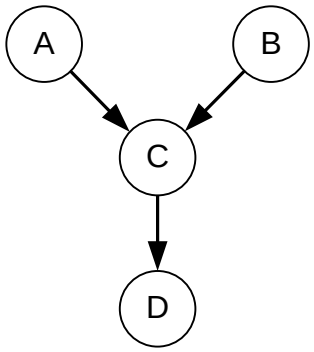


Figure 3.4: Converging Connection

3.2 Naive Bayes Classifier

A classifier is a model used for classification problems e.g. a model which is able to classify if an incoming e-mail is a spam mail or not, see figure 3.5. These classification models contain two types of variables; attributes variables and class variables. The attribute variables can be considered as input for the model while the class variables can be considered as output. Figure 3.5 shows an example of a classifier. The spam-mail example shows the attribute variables (Input) and the class variable (Output).

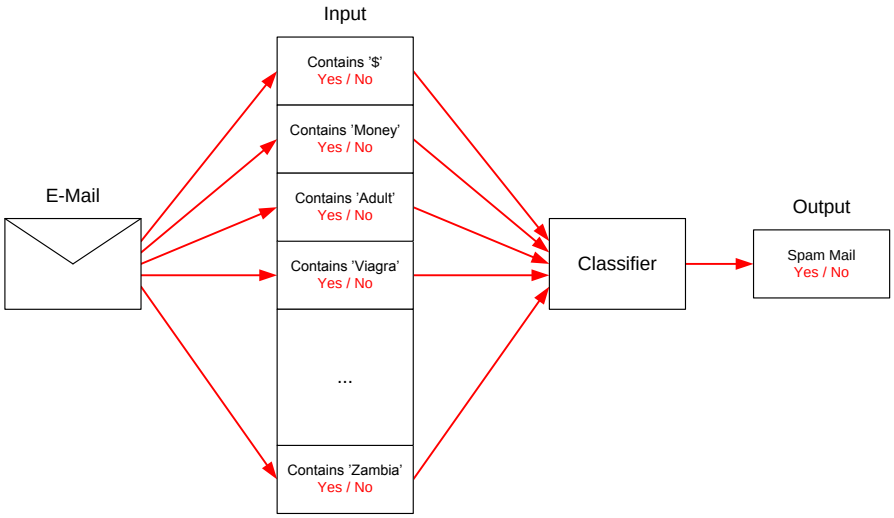


Figure 3.5: Example of a classifier

Bayesian Networks is a model which is very suitable for classification problems. A Naive Bayes Classifier is one of the simplest Bayesian Network based classifiers where there are no dependencies between the attribute variables given the class variable. The only dependencies are between the class variable and the attribute variables where all the attributes are dependent on the class variable. Figure 3.6 shows the Naive Bayes Classifier.

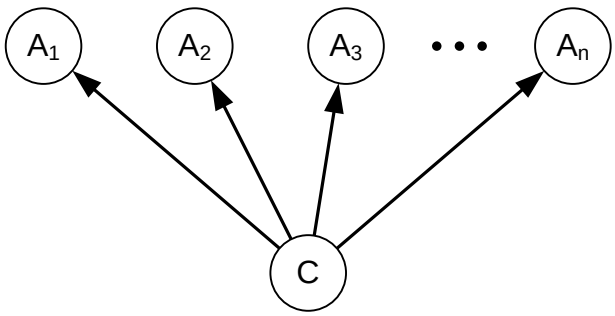


Figure 3.6: Naive Bayes Classifier

Naive Bayes Classifiers has proven to be one of the most consistently well-performing set of classifiers.

Learning Classifier

When we want to find the most likely classification using a Naive Bayes Network and give the attribute values $a_1, a_2, ...a_n$ the following theorem is used:

$$V_{bn} = argmax_{v_j \in V} P(v_j) \prod P(a_i|v_j)$$

It originate from Bayes Theorem and has been rewritten using the assumption that all attribute variables are independent.

Because Naive Bayesian Classifiers uses the assumption that all the attribute variables are independent, learning the network becomes relative easy. Given some training data, learning the classifier is just a matter of counting occurrences target value. Table shows some 3.1 which tells if the weather is good.

Outlook	Temperature	Humidity	Wind	GoodWeather
Sunny	Hot	High	Weak	Yes
Sunny	Hot	High	Strong	No
Rain	Hot	High	Weak	No
Overcast	Mild	Normal	Strong	Yes
Overcast	Cool	Normal	Strong	No
Sunny	Mild	Normal	Weak	Yes
Overcast	Mild	High	Weak	No
Rain	Cool	Normal	Strong	No
Sunny	Hot	Normal	Weak	Yes
Overcast	Mild	High	Weak	Yes
Rain	Cool	High	Strong	No
Overcast	Cool	Normal	Strong	No

Table 3.1: Training Data

Using this data we can estimate the probability for good and bad weather just by counting the occurrences:

$$P(GoodWeather = Yes) = \frac{5}{12} = .42$$

$$P(GoodWeather = No) = \frac{7}{12} = .58$$

Similarly we can estimate the conditional probability of summer given the weather:

$$P(Season = Summer|GoodWeather = Yes) = \frac{3}{5} = .60$$

$$P(Season = Summer|GoodWeather = No) = \frac{1}{7} = .14$$

Using the Naive Bayes Theorem we can also estimate the most likely weather classification given the training data and the following observed attribute variables: *Season = Summer, Temperature = Hot, Humidity = Normal, Wind = Strong.* The following shows how Naive Bayes Theorem is used:

$$P(Yes)P(Summer|Yes)P(Hot|Yes)P(Normal|Yes)P(Strong|Yes) = .012$$

$$P(No)P(Summer|No)P(Hot|No)P(Normal|No)P(Strong|No) = .0073$$

Because $GoodWeather = Yes$ has the highest value, the classifier GoodWeather as Yes. To get the conditional probability of $P(Yes)$ we just need to normalize the probability estimated form the training data $\frac{.60}{.60+.14} = .81$.

All this works well if you have a complete dataset with all excising occurrences. But this method fails if our training data only contains a fraction of the existing occurrences. The probability estimation of $P(Temperature = Cool|GoodWeather = Yes)$ is $\frac{n_c}{n}$ where n is the number of occurrences of $GoodWeather = Yes$ and n_c is the number of occurrences where $Temperature = Cool$ given $GoodWeather = Yes$. In reality the probability for $P(Temperature = Cool|GoodWeather = Yes) = .08$, but since we have no observation of this occurrences in our training data n_c is equal to zero, and the probability of $Temperature = Cool$ given $GoodWeather = Yes$ will also be zero. To overcome this underestimation the m-estimation is introduced:

$$P(a_i|v_j) = \frac{n_c + mp}{n + m}$$

p is a priori estimate for $P(a_i|v_j)$ and a typical method for choosing p if no other information exist is to assume uniform priors. This is done by counting the possible values k of the attribute and let $p = \frac{1}{k}$.

m is a constant weight called *equivalent sample size* and is user specified.

Chapter 4

Approach

This chapter will describe how to approach the subject, and describe the concepts of the methods that are going to be used. But before going into detail with the techniques we will use, we will look into what identifies strategies and they how can be managed.

4.1 Strategy Identification

First we have to consider which factors we could use to identify the enemy strategy. It should be no more than a human would be able to observe since the CP should be non-cheating. This list shows what a human player can observe when playing:

- **Unit position:** Tells something about where the interests of the player are, e.g. if the player has located a large amount of units around a resource field, it might indicate that he is protecting it so he can gather resource. The player could also have a large amount of units around the enemy base, which might indicate that he soon is going to attack.
- **Unit movement:** The unit movement tells something about the next goal of the player or if the player is in a heavy fight the movement tells if he is retreating.
- **Unit type:** The unit types also tell a lot about the enemy strategy. One strategy could be to build some of artillery units and a lot of infantry units. The artillery could the most dangerous of the enemy

units from a long save distance, and when they are destroyed, then attack with the infantry.

- **Unit groupings** The unit grouping also tell something about the strategy like in the previous example where the combination of artillery and infantry. It is likely that in order to execute that strategy these units will move in a group.
- **Timing** The timing of which player attacks or expands his base could indicates if the player is playing aggressive or defensive. If the enemy player attacks with an army very early in the game he is properly playing aggressive, and it is likely the he's base is poorly defended since there would be not enough time to build a sufficient deface.

There could also be many other factors which could indicate the strategy, but it very much depends on the game. These factors listed here are the most general and gave us an idea of what a CP could use to identify the enemy strategy. In this project we will only use the unit position and the next sections will describe how to continue with this approach.

4.2 Strategy Management

What we want is to be able to identify the enemy strategy. Depending on the game there are many different strategies where some strategies are very similar to each other while others are very different. The goal of this project is not to identify which strategies are available in a game, but identify which of some known strategies the enemy is using and further more pick the strategy that counters the enemy strategy best.

The idea is to make at tree of strategies, where the root is the most general strategy and the leaves are the most specific. Actually the root is the base strategy for all strategies. This strategy tree is used to identify the enemy strategy, where the first step is to identify the general strategy and from that point a more specific strategy can be identified until a leaf is reached. Each strategy in the strategy tree contains a counter strategy or a list of counter strategies which will be used against the enemy. In some cases there will be an uncertainty in identifying the enemy strategy. If the uncertainty is too high the player could choose the more general strategy and execute that counter strategy.

4.3 Bayesian Grid Model

In this approach we will only try to identify the enemy strategy by observing the unit positions. A model which can represent the position of the units on the battlefield is needed. One approach could be to use a general Bayesian Network where one node represents the whole battlefield. The problem with this approach is that it will either perform very poorly due to low number of states or it will be very complex because the node should contain many states representing all the variants of the unit positions. Instead a Bayesian Grid Model is used, which is a variant of a Naive Bayes Classifier. The battlefield is split into several grid cells and the Bayesian Grid Model has an attribute variable node corresponding to each grid cell on the battlefield. The class variable of the Bayesian Grid Model is the node that classifies which strategy the enemy is using. Figure 4.1 shows the Bayesian Grid Model where x and y is the number of grid cells on the battlefield.

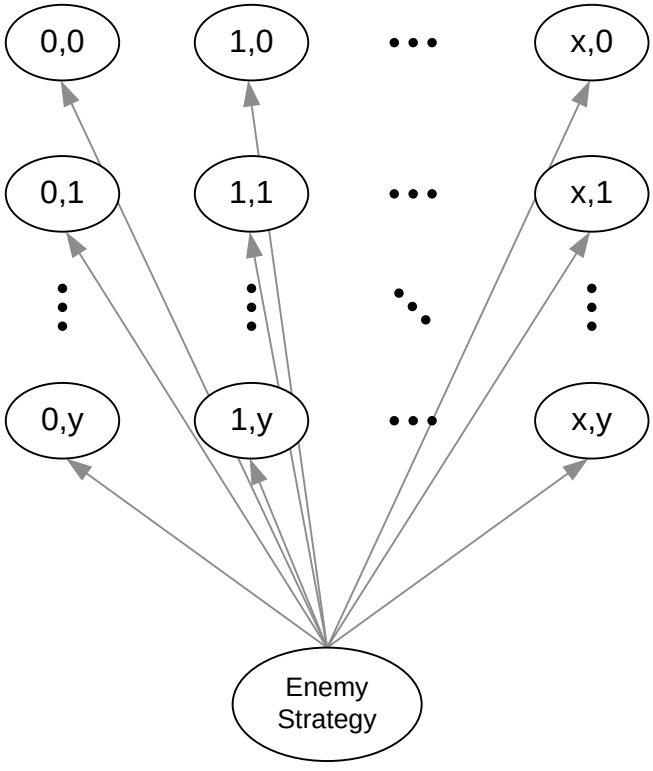


Figure 4.1: Bayesian Grid Model

To measure the position of units on the battlefield the number of units

in each grid cell is counted. The Bayesian Grid Model has to represent this. There are two ways of doing this either measure the actual number of units in each grid cell or measure the percentage of units in each grid cell. Each of these two ways has some pros and cons. To define these variable nodes and measure the number of units in each grid cell we have to define a number of states, and since we cannot define an unlimited number of states we have to figure out the best maximum value (number of units) and the interval. One way to find the maximum value is to count how many units can fit into one grid cell. The problem with that is that some games got flying units which means that units can overlap each other and therefore there can be unlimited units in a grid cell. The problem with using the percentage of units in a cell is that it is necessary to know the total number of units in the game. It is no problem if all information on the battlefield is available, but many RTS games use Fog of War which means that only parts of the battlefield are visible. The good thing about using percentage is the scalability. Even though the Bayesian Grid Model has been trained with a certain number of units, the model will still work if the game is played with a different number of units.

To be able to identify the enemy strategy, each node in the strategy tree contains a Bayesian Grid Model which is a variant of a Naive Bayes Classifier. This model is used to identify a more specific strategy.

4.4 Tools

In order to be able to model Bayesian Networks a tool is needed and in this project a tool called Hugin Expert is used. Hugin Expert is a set of tools which can be used to model Bayesian Networks. It has a graphical editor where it is easy to create networks. The editor can also be used to get a graphical view of a network and be used to test it by manually putting evidence on different variable nodes.

Hugin Expert also provides an Application Programming Interface (API) supporting several programming languages, including C# which is the target programming language of this project.

By using the API enables us to develop programs which use Bayesian Networks. In this way we can create programs which make decisions based on Bayesian Networks. It also makes it easy to generate very large and

complex networks which otherwise would be impossible to create with the graphical editor. [1]

4.5 Implementation

This section describes how the Bayesian Grid Model is implemented in MiniRTS using the Hugin Expert API. There are two classes which are described; the Strategy Learner and Strategy Identifier. Both classes can handle Bayesian Grid Model of any size. Afterwards the implementation of the Strategy Tree will be described.

4.5.1 Strategy Learner

The *StrategyLearner* is a class which is responsible for learning a given Bayesian Grid Model. It has the following four methods:

- **Load:** The load method can load a Bayesian Grid Model of any size. The Bayesian Grid Model can either be a model with no data or a model which has been trained. A trained model is may not sufficient enough so in some cases it can save time to load the trained model instead of starting all over with an empty model.
- **Save:** The save method is used to save a Bayesian Grid Model after it have been trained so it contains all the collected data.
- **SetValues:** This method is used to synchronize the state of the battlefield with the states of the attribute variables of the Bayesian Grid Model. It takes a team as input, and the percentage of units in each grid cell owned by that team is stored in the corresponding attribute variables.
- **Adapt:** The adapt method makes the Bayesian Grid Model adapt to the new attribute variable settings.

4.5.2 Strategy Identifier

The *StrategyIdentifier* class is responsible for identifying the enemy strategy. To identify the enemy strategy it uses a given Bayesian Grid Model of any size. This class has the following three methods:

- **Load:** The load method can load a Bayesian Grid Model of any size.
- **SetValues:** Like in the *StrategyLearner* this method synchronizes the battlefield state with the Bayesian grid model. In addition this method can be given Fog of War which leaves the unseen areas of the battlefield to be uncertain in the Bayesian Grid Model.
- **SelectStrategy:** This method returns the strategy type which is identified.

4.5.3 Defining Strategies

In this section we will describe the strategies used in this project. So for our approach we define the following strategies.

Aggressive Strategies:

- **Attack from north**
- **Attack from the middle**
- **Attack from south**

Defensive Strategies:

- **Defend north lane**
- **Defend middle lane**
- **Defend south lane**

The strategies are selected so they counter each other. Attack from north can be countered by the Defend north lane and so on. Figure 4.2 shows the strategy tree.

To help defining these strategies, strategy points are placed on the battlefield. Strategy points define where to move to when the strategies is executed, e.g. the *AttackNorth* strategy point tells where all the units should move to when attacking from north. Figure 4.3 shows how the strategy points are placed.

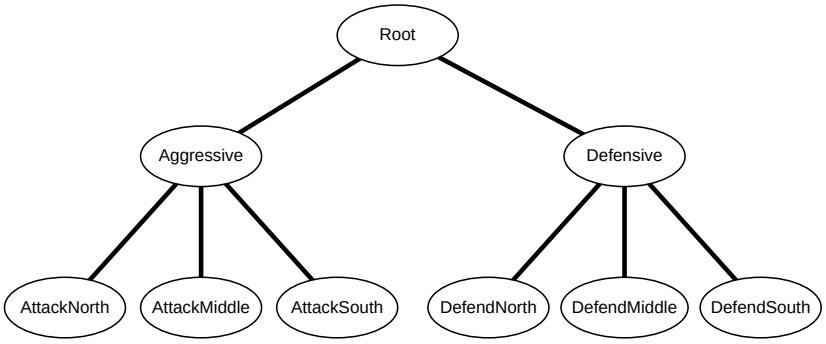


Figure 4.2: Bayesian Grid Model

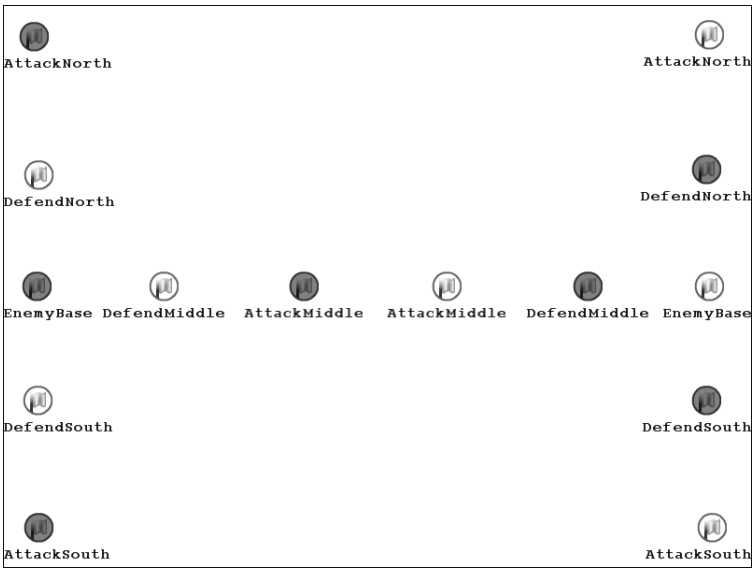


Figure 4.3: Strategy Points

4.5.4 Tree Based Learner

The *TreeBasedLearner* is a strategy learner which implements the strategy tree and used the strategy definitions from the previous section. It uses three *StrategyLearners* described in section 4.5.1 on page 24 where one is used to learn the *Aggressive* and *Defensive* patterns, the other is used to learn the *AttackNorth*, *AttackMiddle* and *AttackSouth* strategies while the last is used to learn the *DefendNorth*, *DefendMiddle* and *DefendSouth* strategies.

4.5.5 Tree Based Identifier

The *TreeBasedIdentifier* and is similar to the *TreeBasedLearner*, but in addition it can use Fog of War to make the input data incomplete.

4.5.6 Naive Identifier

To be able to argue that using Bayesian Grid Models is actually better than a simple approach we need to compare it with such an approach. In this project the simple approach is called the Naive Identifier.

The Naive Identifier splits the battlefield into six zones, one for each type of strategy. The zones are placed in the areas where the different strategies are played, e.g. if the player attacks from north the units will move in the north region of the battlefield. Figure 4.4 shows the six zones where the aggressive zones are rectangular and the defensive are radius zones. The defensive zones are radius zones because when the player is playing a defensive strategy all the units will group together in these areas. The aggressive zones are rectangular zones span from the left to the right side of the battlefield because the units will move from the one end to the other while playing aggressive. The placement of the defensive zones are based upon the defensive strategy points form figure 4.3, while the aggressive zones are created by dividing the battlefield in three equally sized zones.

When identifying the strategy it is just a matter of counting the units in the different zones and the zone with most units is used to select the identified strategy. But as the figure shows the zones overlap each other due to the fact that some strategies are played in the same regions e.g. the *defend middle* is almost fully overlapped by the *attack middle* zone. These overlaps make the counting of units rater complicated, so instead of doing a lot geometry calculations a much simpler approach is used. The total number of units in the aggressive zones is multiplied with 0.75 to lower the number. So if all units are inside the *defend middle* zone and also inside the *attack middle*, the defend middle will be rated higher due to the reduction of count of attack middle.

In this way we have created a very simple but also reliable identifier to testes against the Bayesian Grid Model based identifiers.

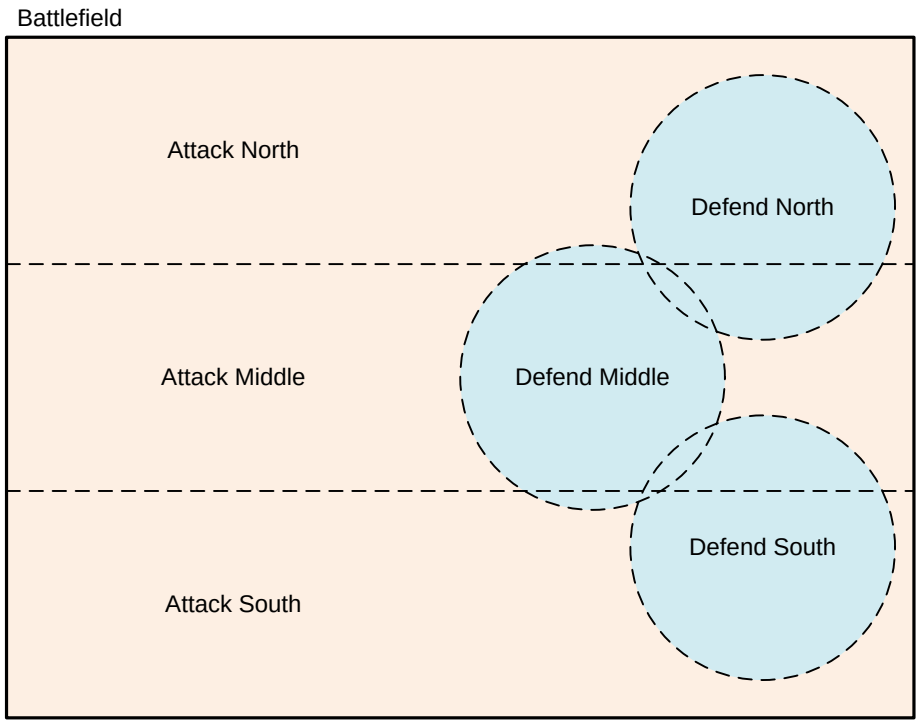


Figure 4.4: Naive Identifier

4.5.7 Generating Models

In this project we want to test different models against each other the to see which one performs best. In order to do that we need at tool which can create a variety of Bayesian Grid Models, since it is very large and it is impossible to create them by hand in the Hugin Expert model editor.

The *ModelGenerator* is a tool with the purpose of creating all the variants of the Bayesian Grid Models. Figure 4.5 shows the Graphical User Interface (GUI) which can be used to generate these models. The following list explains the input boxes:

- **Filename:** The name of the file where the model is saved, when it has been generated.
- **Size X:** The horizontal number of attribute variables in the Bayesian Grid Model.
- **Size Y:** The vertical number of attribute variables in the Bayesian Grid Model.

- **Number of States:** The number of states in each attribute variable.
- **Max Value:** If the model does not use percentage the *Max Values* defines the maximum number of units the attribute variables can count. If *Use Percentage* is checked the maximum value will always be 100%.
- **Output States:** The states which the class variable should contain.

Even though the *ModelGenerator* GUI has mostly been used for test e.g. to test if the model is generated correctly, the *ModelGenerater* can also be accessed in program code.

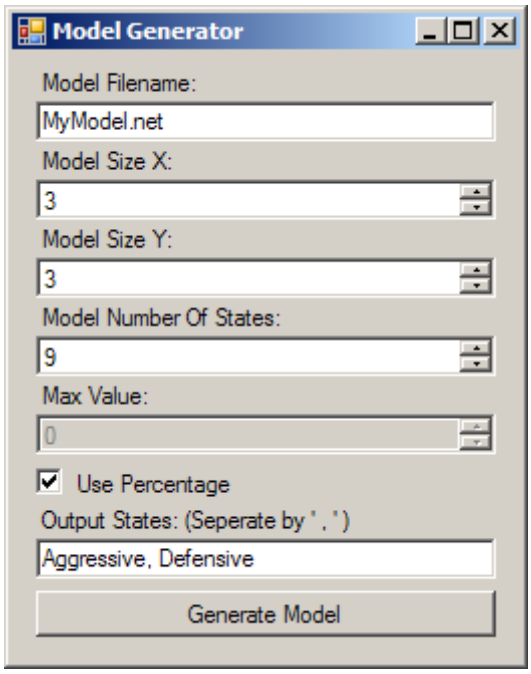


Figure 4.5: Model Generator GUI

Chapter 5

Experiments

This chapter describes experiments performed to study which strategy identifier is the best. But before going to the results we first describe the tools used to monitor the performances of the different identifiers. Afterwards the different scenarios, which is used in the experiments is explained. At last the results will be shown and commented.

5.1 Graph Plotting Tools

In order to be able to monitor the performance for the diffident models we need tools that can generate reprehensive data. The following sections describe the two tools which are used.

5.1.1 Strategy Monitoring Tool

This tool can keep track of which strategies the players are playing, and at the same time keep track the identification of the strategy. Every time the player changes strategy this tool generates a time stamp together with a value representing the strategy. The same applies with the identifier, when the identifier identifies the strategy. This data can then be plotted into graphs to see how long it takes for the identifier to identify the strategy, or if it identifies the right strategy at all.

5.1.2 Strategy Evaluator Tool

This tool is an evaluation tool used to evaluate the different identifiers. The tool constantly compares the strategies selected by identifiers with the

real strategy which is being played. Penalty points are given to the identifier when the identified strategy dos not correspond to the real strategy. The penalties are defined by a penalty matrix and each strategy is given a number, so the identified and real strategy together creates an index in the matrix, e.g. AttackMiddle and AttackSouth = index(2, 3). Table 5.1 shows the penalty matrix, and the following list shows how the strategies are numerated.

1. AN = Attack North
2. AM = Attack Middle
3. AS = Attack South
4. DN = Defend North
5. DM = Defend Middle
6. DS = Defend South
7. NI = Not Identified

Strategies	AN	AM	AS	DN	DM	DS	NI
AN	0	1	2	10	15	15	20
AM	1	0	1	15	10	15	20
AS	2	1	0	15	15	10	20
DN	10	15	15	0	1	2	20
DM	15	10	15	1	0	1	20
DS	15	15	10	2	1	0	20

Table 5.1: Penalty Matrix

The penalties are given based on how different the strategies are from each other, and how critical it is to identify the wrong strategy, e.g. if player A thinks that player B is defending his base but in reality is attacking, player A might attack and leave his base defenseless. This is critical miss-identification and therefore the penalty points are high. The penalty points between similar strategies is low, e.g. if the player miss-identifies the enemy strategy as attack north but it in reality is attack middle it is not that critical. The distance between defend middle and defend north is small, and the units can move to the other location in a short period of time. The same applies with the defensive strategies because when player B is playing

defensive he is no threat to the base, and player A can just pick another attacking direction.

5.2 Scenarios

In order to be able to run experiments, the game has to be setup in a way that is suitable for the experiments. These setups are called scenarios and will be described in this section. All scenarios use the *Scenario* class described in chapter 2 which they inherits form.

5.2.1 Learning Scenario

The learning scenario is designed specific to learn the Bayesian Grid Models so it can be used to identify the enemy strategy. Figure 5.1 shows the battlefield setup of the learning scenario, where there are two players. Both players have a Head Quarter (HQ) and 23 units and are located in each end of the map.

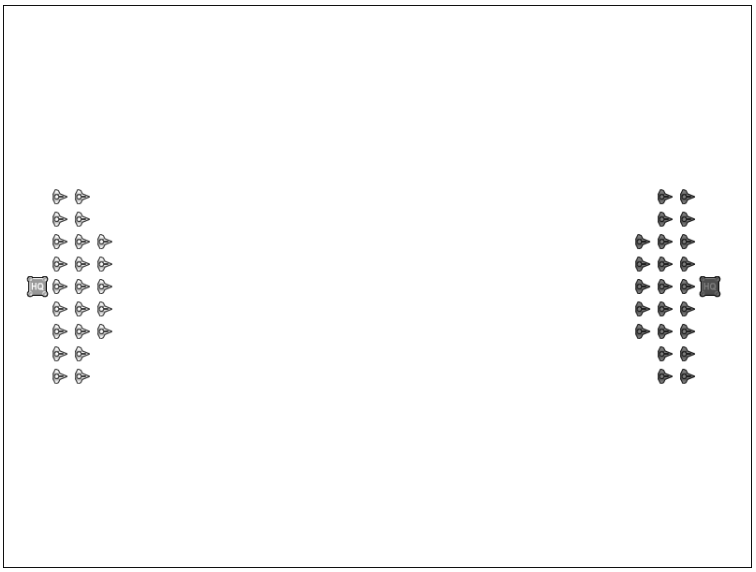


Figure 5.1: Scenario

To learn the Bayesian Grid Model properly it is necessary to plays all the strategies against each other. In the beginning of each game the players pick random strategies but to avoid a never ending game the players can not pick a defensive strategy at the same time.

The learning scenario resets the battlefield when one of the players wins the game, which is either when the enemy HQ is destroyed or all the enemy units are destroyed.

To get a decent amount of data the game is played 500 times where each player chooses a random strategy. Two TreeBasedLearners collects data for each player and after the 500 games the data is saved and ready to be used in the test scenarios.

The learning scenario can be given a list of any Bayesain Grid Model sizes, an the scenario will automatically generate these models using the *ModelGenerator* and afterwards train them.

5.2.2 Test Scenario 1

The first test scenario is a scenario where all information on the map is available. This scenario is created to test how well different models perform and is compared against each other. The battlefield setup is different from the learning scenario. In this scenario we only have one player because the other player is irrelevant for this test.

When the game starts the player selects a strategy. The models then have to identify the strategy. After some time the player changes strategy and then the models have to identify the new strategy. In this scenario the *Strategy Monitoring Tool* is used described in section 5.1

5.2.3 Test Scenario 2

This scenario is used to test how the increasing number of grid cells and number of states in the Bayesian Grid Models will influence the performance of the identifiers. To compare the performance the *Strategy Evaluator Tool* is used. The battlefield settings are the same as in the previous scenario, where there is only one team on the battlefield.

5.2.4 Test Scenario 3

In his scenario we want to test how well the identifiers perform when the battlefield data is incomplete due to Fog of War. The areas which are visible in the layer Fog of War are place semi-randomly. By semi-random we mean that the visible areas are randomly placed, but still not so random that we will never catch activity. In a way which ensures that at least a few units

are revealed. Figure 5.2 shows the Fog of War. The rest of the battlefield setting is the same as before.

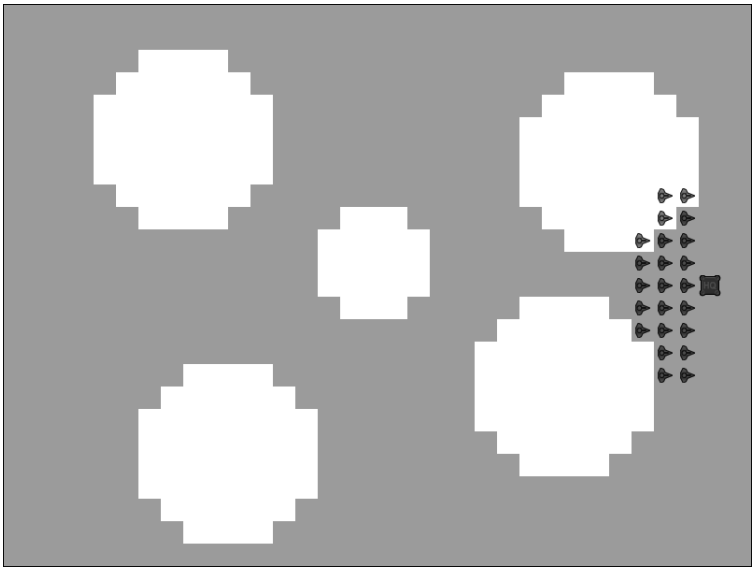


Figure 5.2: Scenario with Fog of War

5.3 Results

This section will present the results from the experiments in the different test scenarios. Before showing the results let's define some notations. The Bayesian Grid Models can vary in size and can have different number of states. The following notation AxBxC notates the complexity of the Bayesian Grid Model where A is the number of attribute variables in horizontal direction, B is the number in vertical direction and C notates the number of states. E.g. 4x3x5 notates a Bayesian Grid Model which is 4 wide and 3 high, and have 5 states. All figures of from the results can be found in a large version in the appendix A on page 44.

5.3.1 Test Scenario 1

This section presents the results from the first test scenario. The following figures shows the strategy changes of the player and the identifier over time, where the Y-Axis contains all the different strategies that can be played and

the X-Axis is the time line. The graphs have been generated by using the *Strategy Monitoring Tool* from section 5.1 on page 30.

In general all the identifiers tested in this scenario were able to identify the strategy but some performed better than others. There are two factors, the time it takes to identify the strategy and what strategy it identifies. As the figures shows, the Naive Identifier and the 3x3x3 Identifier are more likely to pick the wrong strategy, before actually picking the right one.

Both the 9x9x9 and 11x11x9 Identifier performs better than the Naive and 3x3x3 Identifier, but which one performs best is hard to tell. It seems that the 11x11x9 Identifier can identify the strategy a little faster than the 9x9x9 sometimes, but the figure also shows that it at one point identifies the strategy wrong.

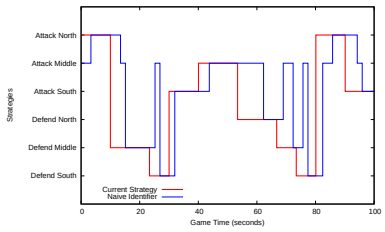


Figure 5.3: Naive Identifier

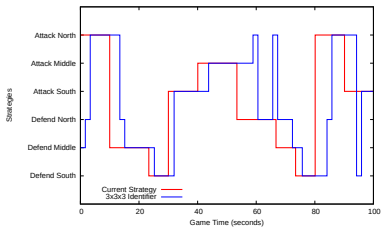


Figure 5.4: 3x3x3 Identifier

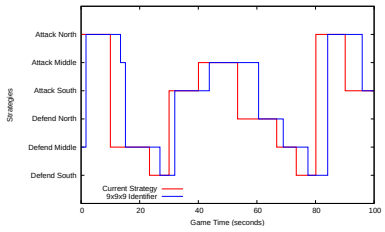


Figure 5.5: 9x9x9 Identifier

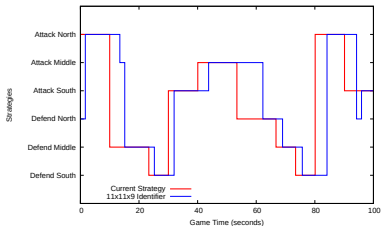


Figure 5.6: 11x11x9 Identifier

It is very hard to tell which one performs best by just looking at these figures. This scenario just gave a brief overview of the performance of some identifiers the next sections will look into the models in more detail.

5.3.2 Test Scenario 2

In this scenario we tested how the increasement of the model complexity will influence the performance of the strategy identifiers. The following figures

has been generated using the *Strategy Evaluator Tool* described in section 5.1 on page 30.

Figure 5.7 shows the result, where the model grid size is increased. The X-Axis represents different grid sizes and the Y-Axis represents the penalties. Each graph on the figure represents a different identifier. By looking at the figure we can see that in general, increasing the number of grid cells improves performance. But as the complexity get higher, the improvement gets smaller. It even looks like complexity can be too high and worsens the performance, but this could also be due to the uncertainties of at model which have not been trained enough. When comparing the model with different number of states, it can be noticed that the three graphs is following each other, it means the increasing the number of states does improve the performance of all models. Though the one with 21 states performs best followed by the one with 9 states (note: comparing the states when the grid size is low seems to be unreliable). All the Bayesian Grid Models performs better than the Naive Identifier.

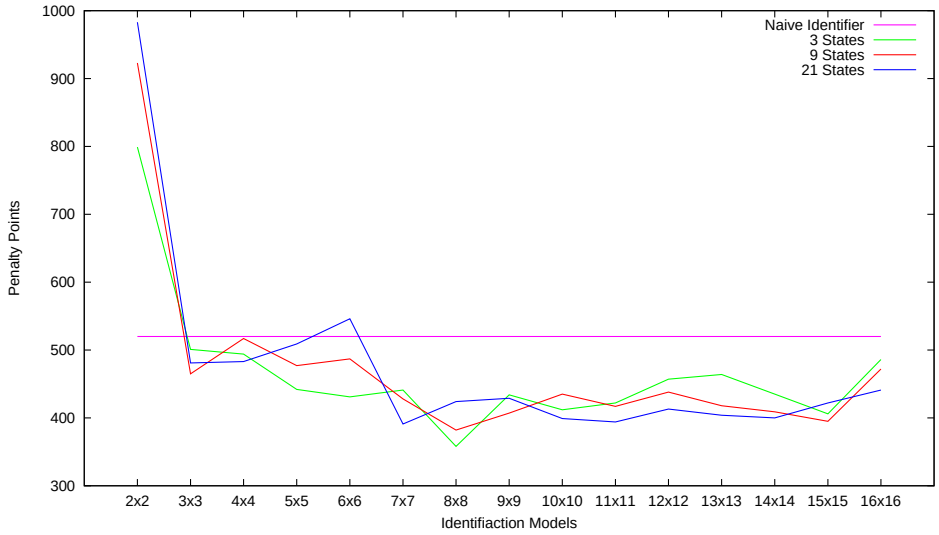


Figure 5.7: Increasing Model Grid Size

Figure 5.8 shows the same as the other, but instead it is the number of states that is shown on the X-Axis, and the different graphs are the grid sizes. This figure tells the same as the other, where comparing a grid models with low complexity is unreliable, and increasing the number of states improves the performance.

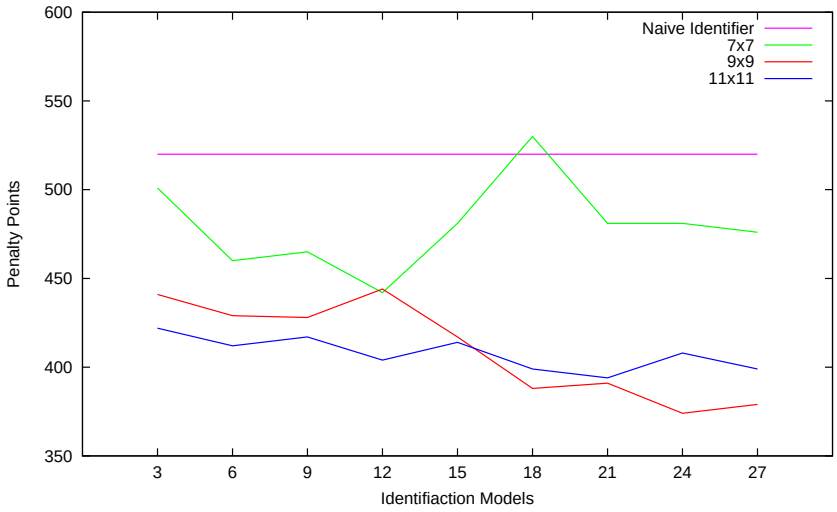


Figure 5.8: Increasing Number of States

The scenario has shown that in an open environment where all information is available, the Bayesian Grid Models performs very well when they have a reasonable grid size. The next scenario will add Fog of War.

5.3.3 Test Scenario 3

The scenario has tested how Fog of War has influenced the performance of the identifiers. Figure 5.9 shows the same as figure 5.7 but in this one the graphs is generated on incomplete data due using Fog of War. The results shows once again that increasing the complexity of the grid model increases the performance, though the given penalty points are in general much higher. Also the Bayesian Grid Models performed better than the Naive Identifier.

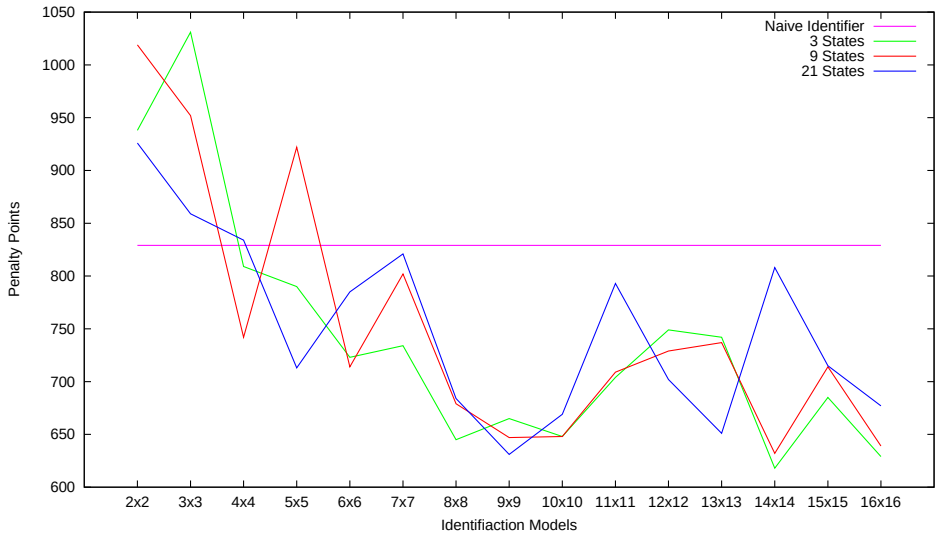


Figure 5.9: Strategy Evaluation with Fog of War

Chapter 6

Conclusion

This chapter will evaluate and conclude upon this project. The focus of this project was on non-cheating AI in RTS games where the goal was to study the usability of Bayesian Grid Models to identify the enemy strategy when give the positions of units. The results of this project will be discussed in the discussion section

6.1 Discussion

In this project we have studied which identifier performed best and how the increasing model complexity (size and number of states) would improve the performance. Increasing the number grid cells improved the model until a certain point and after that; increasing complexity did not improve performance. Changing the number of states also improved performance. The project has also shown that Bayesian Grid Model is usable in games with Fog of War.

Since the goal was to develop an AI for a RTS game to is important that the Bayesian Grid Model does not consume too much CPU resources. The model is very usable since the results have shown that a complex model is not necessary.

Most models performed well and were able to identify the enemy strategy. Even the performance of the Naive Identifier was acceptable. Why not always use the Naive Identifier then? There are different reasons why to use Bayesian Grid Models instead of the Naive Identifier. First, in general the Bayesian Grid Models performed better as long the model had a proper grid

size and number of states. Second, imagine a game with many strategies instead of the six used in this project. The Naive Identifier should contain many zones and different statements to be able to identify one specific strategy. Creating such an identifier would be at very complicated job, and it has to be done manually. By using Bayesian Grid Models

This project has shown that using the position of units can be used to identify strategies. But using positions alone is not enough, e.g. when the player is attacking and suddenly retreads. The Bayesian Grid Models do not capture this before the units are at the defensive positions, but it can be trained to capture it. This requires a Bayesian Grid Model with a large number of grid cells since a moving group of units is spread out. If the model has a low number of cells it would not capture it since too many units would be in the same cell. An alternative could be to capture the unit movement directions in addition.

6.2 Future Work

This project has only investigated subtopic of creating a non-cheating AI for RTS games, where the focus was on using the positions of the units to identify the enemy strategy. As described in section 4.1 on page 20 there is more information than the unit position that helps identifying the enemy strategy. This information could be used to improve the strategy identifiers.

In this project we used a strategy tree to identify the correct strategy, and select the counter strategy. This was just a simple solution that fitted this project. The strategy management is a whole other research area of the high-level AI, and maybe there exists a better way to align the strategies than in a tree. The type of strategies also depends very much on the rules of the game, so if one should use Bayesian Grid Models identify strategies in a game another organization of strategies could be preferable.

The environment in this project used a battlefield with no obstacles. Most RTS games have different maps where there are obstacles like trees, rocks, cliffs and so on. Using Bayesian Grid Models in this project was no problem since the battlefield was always the same. But when battlefields can change one Bayesian Grid Model trained on one map will not work in another, so a new Bayesian Grid Model has to be trained for each map. To avoid this, a more general way of representing the battlefield than grid

cells may be considered. This could be by representing the battlefield using important key areas on the battlefield e.g. resource areas or outposts.

Chapter 7

Bibliography

[1] Hugin Expert A/S. *HUGIN EXPERT*. http://www.hugin.com/, 2008. Online: 14/08-2008.

[2] Alex J. Champandard. *Research Opportunities in Game AI*. http://aigamedev.com/questions/research-opportunities, 2007. Online: 14/08-2008.

[3] Blizzard Entertainment. *Blizzard Entertainment - StarCraft*. http://www.blizzard.com/us/starcraft/, 1998. Online: 14/08-2008.

[4] Blizzard Entertainment. *StarCraft II*. http://www.starcraft2.com/, 2008. Online: 14/08-2008.

[5] Olav Geil. *The Mathematical Foundation of A**. http://www.math.aau.dk/~olav/undervisning/dat06/astar.pdf, 2006. Online: 09/12-2007.

[6] Chris Sigaty James Mielke. *Starcraft 2's AI Does Not Cheat Like Before*. http://aigamedev.com/links/2008-week-19, 2008. Online: 14/08-2008.

[7] Alexander Kovarsky Marc Lanctot terling Orsten Michael Buro, Timothy Furtak. *ORTS - A Free Software RTS Game Engine*. http://www.cs.ualberta.ca/~mburo/orts/, 2007. Online: 14/08-2008.

[8] Microsoft. *xna.com*. http://www.xna.com/, 2008. Online: 14/08-2008.

[9] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[10] David Silver. Cooperative pathfinding. 2005.

[11] Wikipedia. *Bayesian network - Wikipedia, the free encyclopedia*. `http://en.wikipedia.org/wiki/Bayesian_network`. Online: 14/08-2008.

[12] Wikipedia. *Naive Bayes classifier - Wikipedia, the free encyclopedia*. `http://en.wikipedia.org/wiki/Naive_Bayes_classifier`. Online: 14/08-2008.

Appendix A

Large Graphs

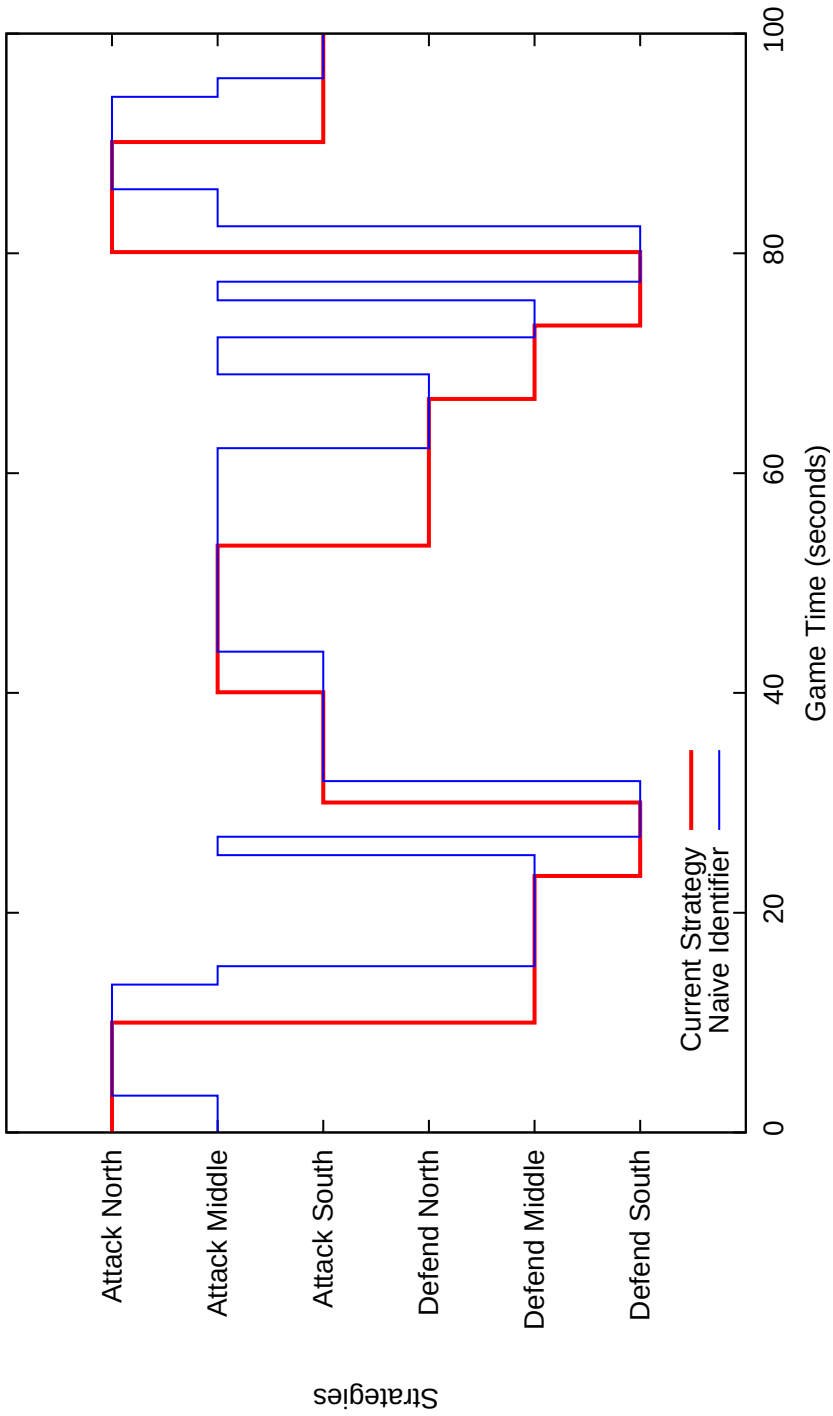


Figure A.1: Naive Identifier

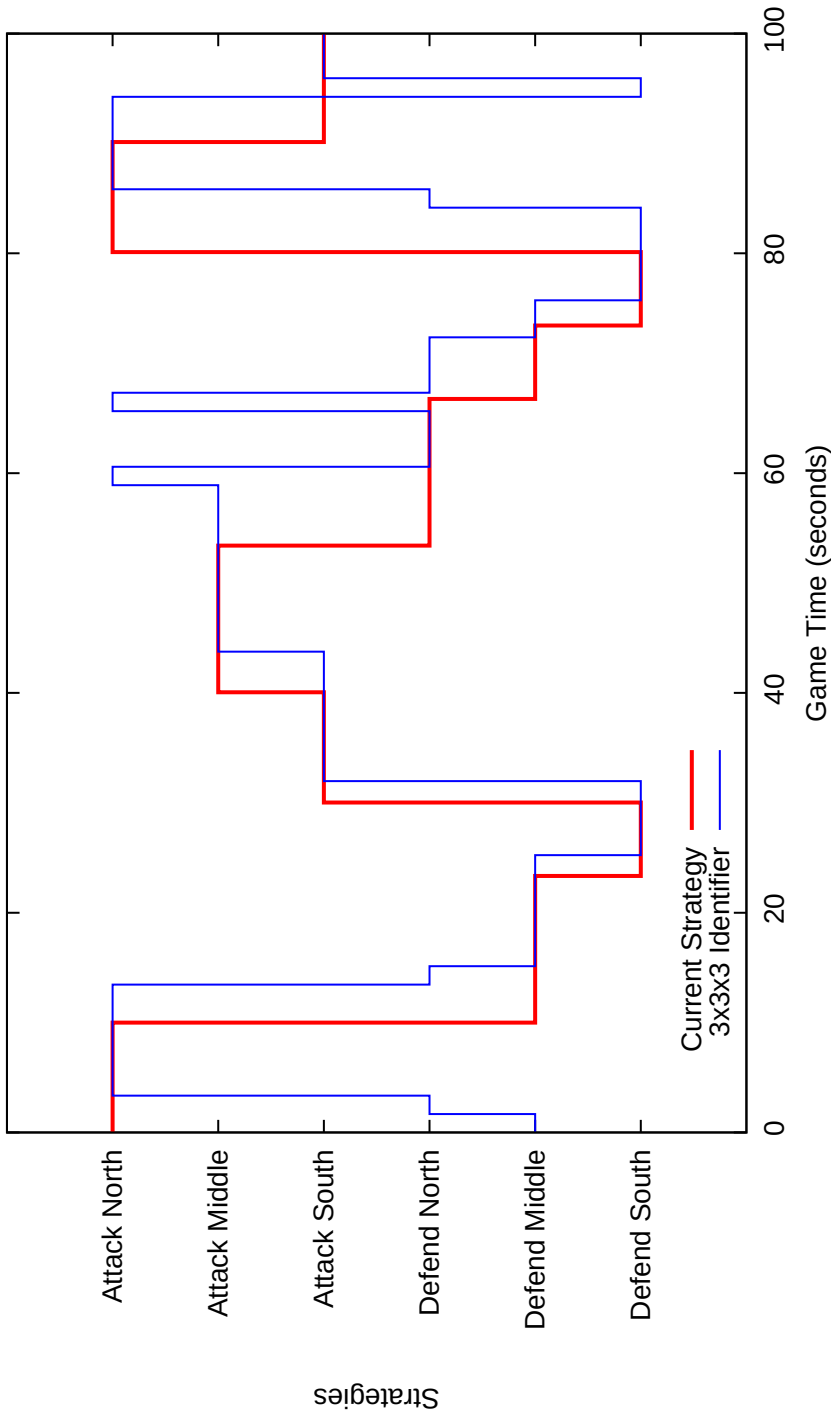


Figure A.2: 3x3x3 Identifier

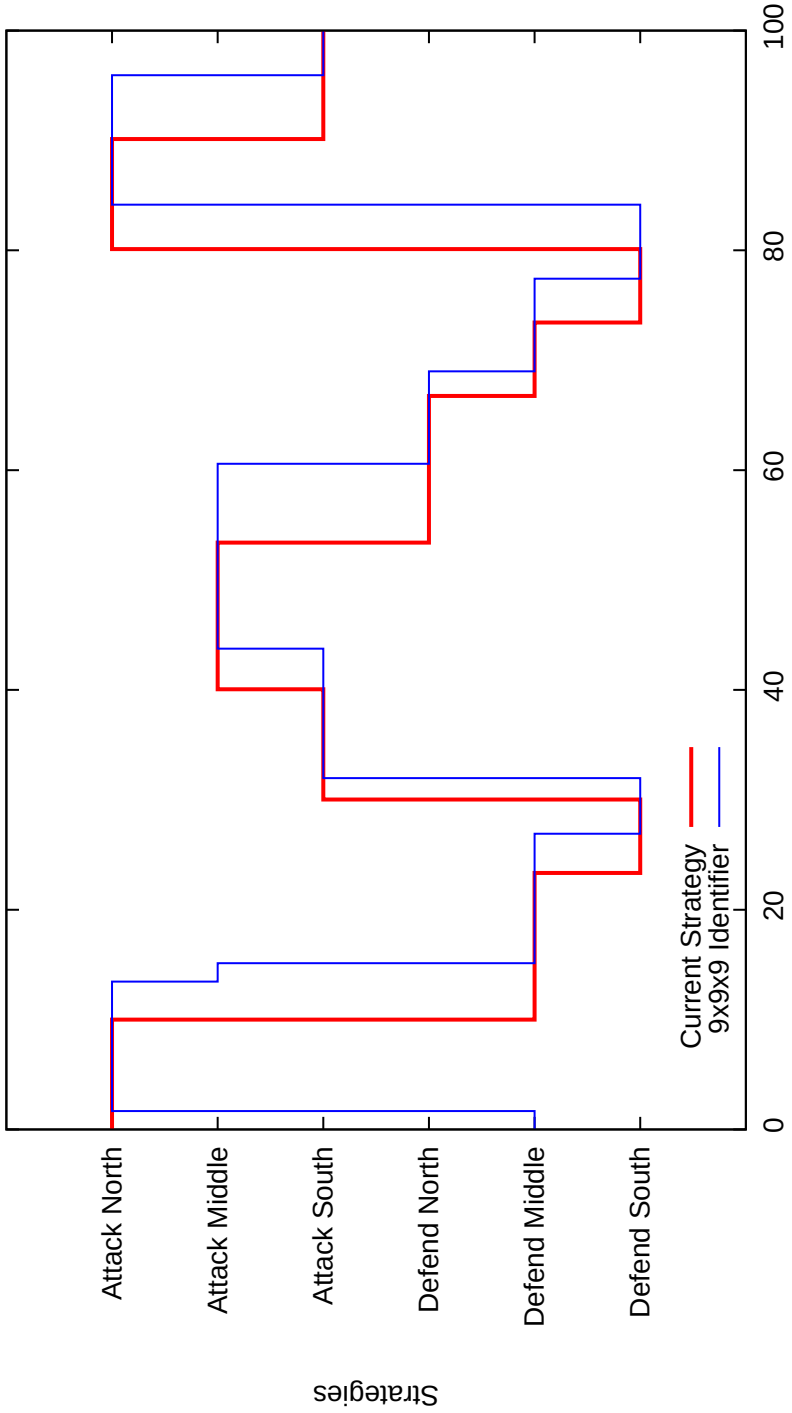


Figure A.3: 9x9x9 Identifier

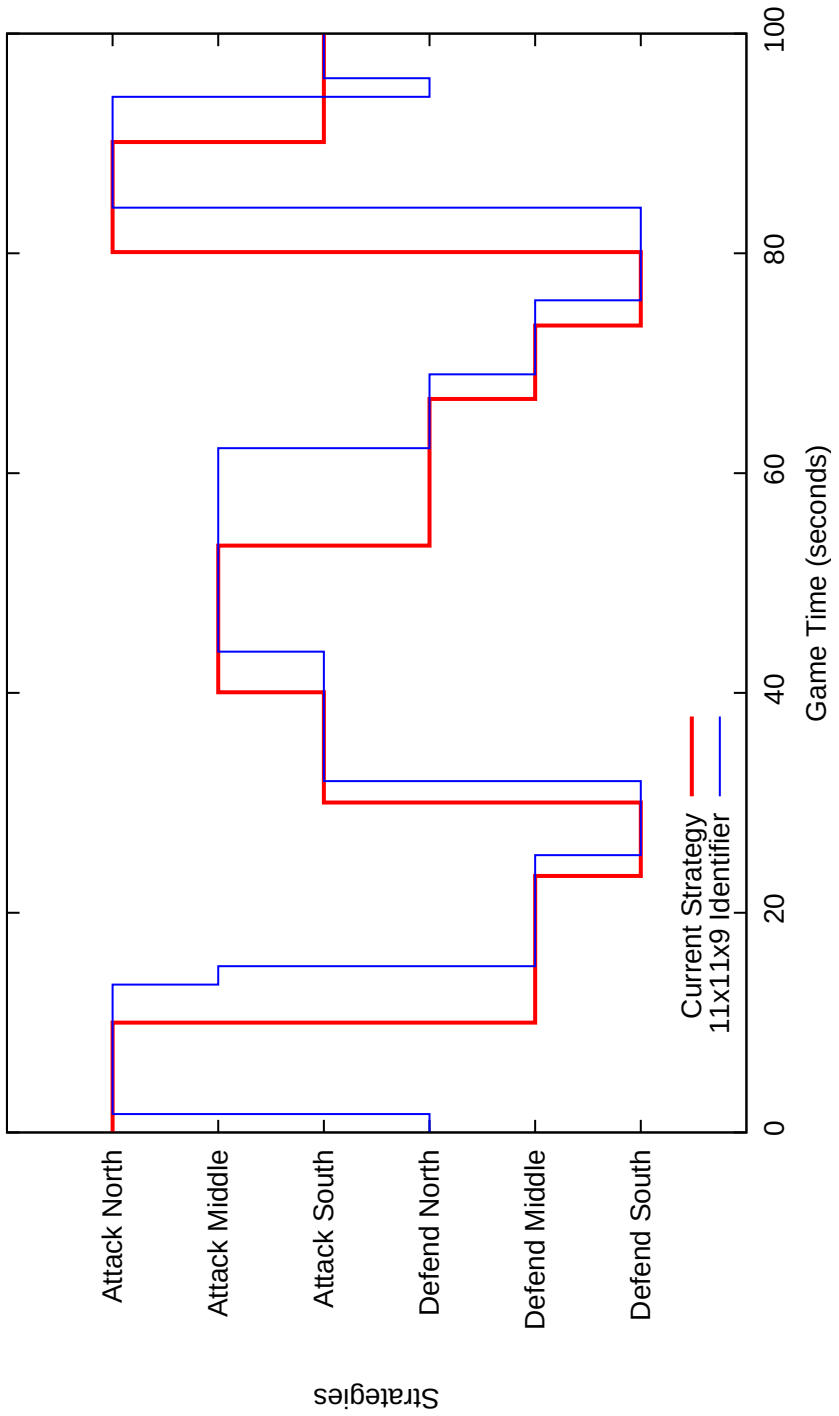


Figure A.4: 11x11x9 Identifier

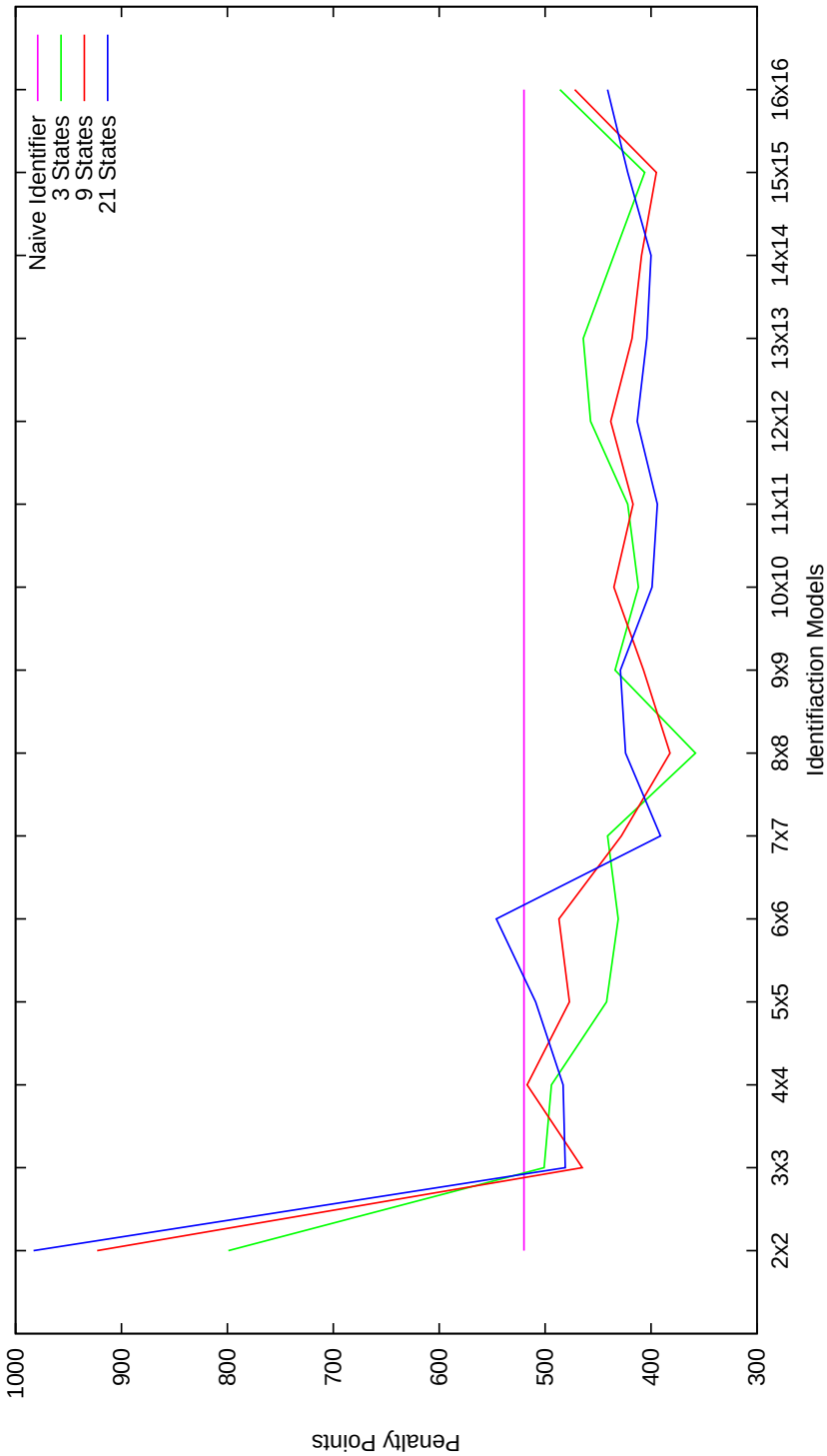


Figure A.5: Increasing Model Grid Size

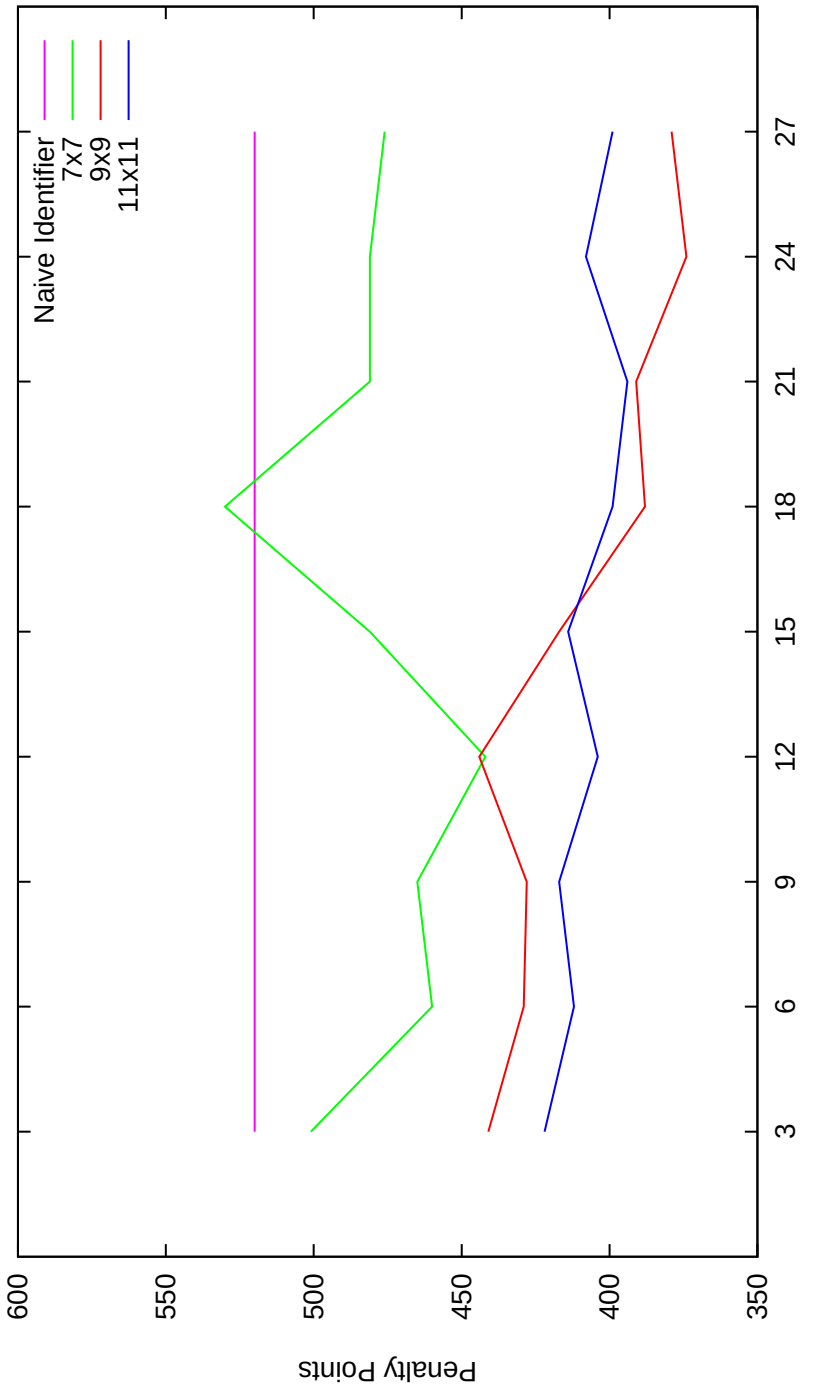


Figure A.6: Increasing Number of States

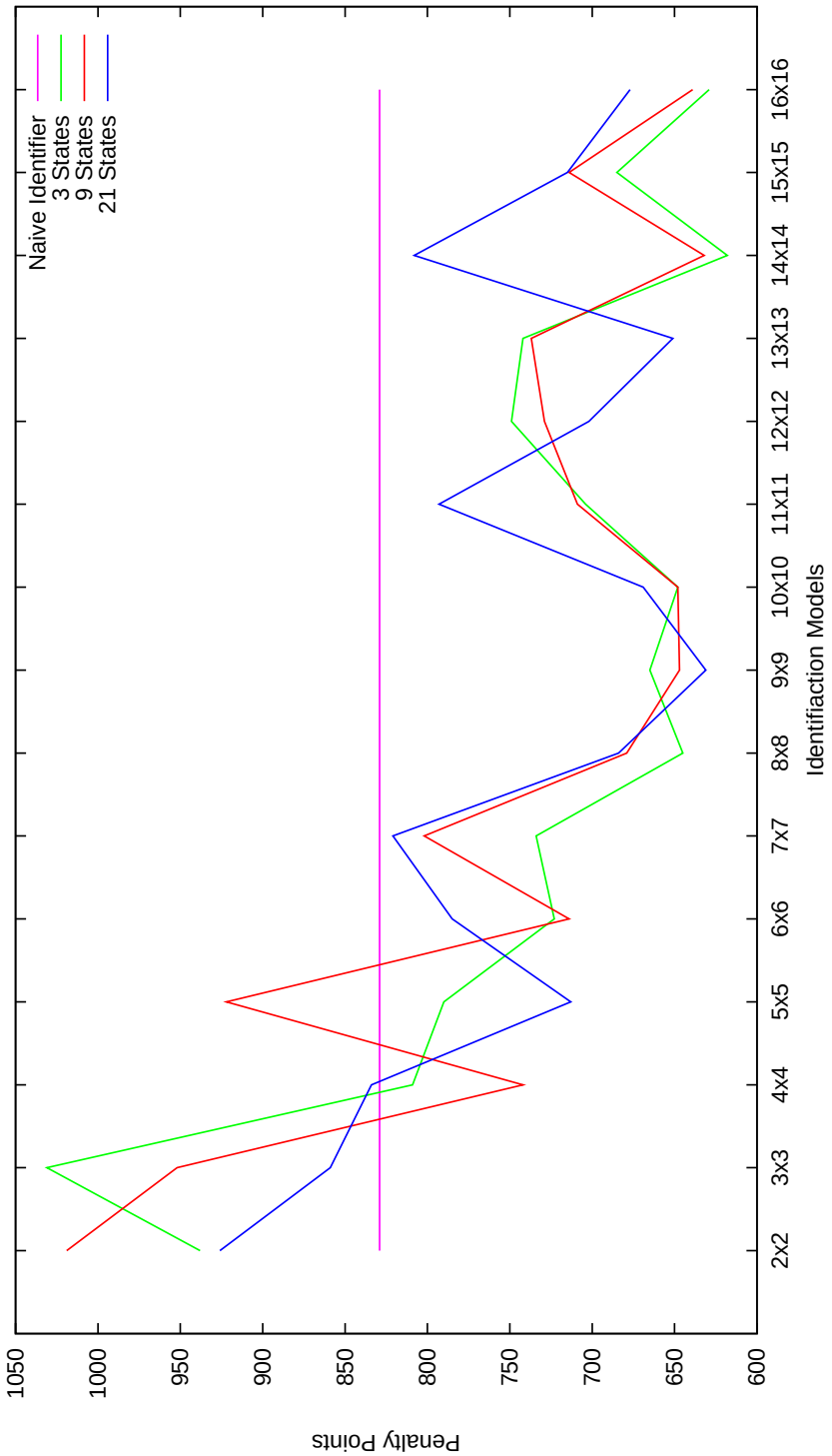


Figure A.7: Strategy Evaluation with Fog of War

Appendix B

CD Content

Included in the report is a CD which contains the following content.

- MiniRTS source code
- MiniRTS complied code
- This report