Incorporating Trust and Trickery Management in First Person Shooters

E1-121

Master Thesis

June 2004

Department of Computer Science
Aalborg University
Fredrik Bajersvej 7E
DK–9220 Aalborg
DENMARK

Faculty of Engineering and Science
Aalborg University

Department of Computer Science

TITLE:

Incorporating Trust and Trickery Management in First Person Shooters

PROJECT PERIOD:
DAT6,
January 2nd 2004 –
June 11th 2004

PROJECT GROUP:
E1-121

GROUP MEMBERS:
Niels Christian Nielsen
Henrik Oddershede
Jacob Larsen

SUPERVISOR:
Olav Bangsø

NUMBER OF COPIES: 8

REPORT PAGES: 80

APPENDIX PAGES: 12

TOTAL PAGES: 92

SYNOPSIS:

This report describes how human character traits can be incorporated into computer controlled characters (bots) in team based First Person Shooter (FPS) type computer games. The well known FPS Counter-Strike was chosen for use as a test environment.
Initially a model of a bot which is exclusively based on the success of the team is devised.
This model is extended into one that allows for two kinds of personalities in a bot: a trickster personality, that lies about information in order to receive personal gain, and a team based personality that, regardless of its own gain, does what it believes is best for the team as a whole.
Detecting tricksters on a team is accomplished through incorporation of trust management. In short this means that each bot maintains trust values for all other bots. The personal trust values and the trust values received from other bots (reputation) are combined to determine whether or not any specific bot can be trusted to provide correct information. Applying trust management to a group of bots which must make group based decisions instead of only individual ones is not a trivial matter. Therefore these issues are also discussed.
Concludingly, issues about how trickster type bots may overcome the trust management implementation in their team mates are discussed. We call this Trickery Management. This extension means a trickster tries to guess threshold values in the trust management implementation of its team mates, calculating when to try to trick and when not to.

Acknowledgments

This report was written by group E1-121 on the DAT6 semester of Computer Science at Aalborg University and is to be evaluated June 2004.

The group would like to thank Hugin Expert for providing us with access to a full version of their tool Hugin free of charge.

The relevant code can be accessed on the following website:

http://www.cs.auc.dk/∼henrik/dat6/

Henrik Oddershede

Niels Christian Nielsen

Jacob Larsen

Contents

Part I

Domain Analysis

Chapter 1

Introduction

In recent years graphics and sound in computer games have improved tremendously. While it is still possible to sell games based mostly on impressive graphics, players do seem to demand more in form of aspects like a story line or advanced Artificial Intelligence (AI) depending on the type of game. The term AI is not meant to describe the general form of artificial intelligence often researched, but is rather used to describe the actions of a computer controlled character, so even a scripted sequence of actions carried out by any computer controlled character can be considered to be AI. The use of the term AI is mostly due to historical reasons. AI in computer games has been greatly improved in the later years. [17] discusses different types of AI used in commercial games, and [7] mentions different kinds of computer games. This project focuses on the development of a cooperative AI in a First Person Shooter (FPS) game. In FPSs, computer controlled characters are called **bots**.

While board-type games have properties that open for the possibility of creating efficient search methods for "solving" a game, this is not the case in FPSs. According to [6], one can not rely on search techniques to find the best possible action at a given time, as is the case in most board-type games researched thus far. FPSs have the following more advanced properties:

1. A player in a FPS does not have access to the complete game state as does a player in e.g. chess or checkers.

2. The choice for action of a player in a FPS unfold continuously as time passes. At any time, the player can move, turn, shoot, jump, or just stay in one place. There is a breadth of possible actions that make search intractable and require more knowledge about which actions might be useful.

The problem with board-type games is that they are not interesting enough that people actually play them against computer controlled opponents - at least not to the extent they play FPSs. This project aims to research AI for popular games, even if they do not have the same properties as board-type games. According to [10], the game Counter-Strike is one of the most popular FPSs ever created, and thus it was chosen for implementation and test purposes.

1.1 Problem Definition

The research described in this report focuses on contributing to the development of a trust management system for FPSs, where the complete state of the game is not known and where actions unfold continuously as time passes. In order to make this contribution and render probable the applicability of it, an implementation of a cooperative bot called **CO-Bot** is carried out. CO-Bot has the following properties:

- CO-Bot is group aware and cooperative.
- CO-Bots have different personalities, some of which are selfish.

A CO-Bot with a selfish personality is called a trickster.

- Tricksters attempt to trick team mates into taking actions to the benefit of the trickster.
- Tricksters as well as non-tricksters try to detect tricksters. They do so by maintaining and utilizing a trust management system.
- Tricksters try to go unnoticed about their trickery, in order to stay trusted and thus maximizing their influence to their own benefit. This is conducted utilizing trickery management.

The above properties are developed in an incremental order. First, a simple decision model is developed, that enables CO-Bot to play the game of Counter-Strike in that it provides basic playing strategies. Furthermore, it provides CO-Bot with group awareness and a flexible cooperative ability, so that it tries to optimize the success rate of the group through cooperation. This cooperative decision model is tested against an (externally developed) existing bot, to show whether CO-Bots and their ability to cooperate are superior to existing bots.

Next an extended decision model provides CO-Bot with human-like characteristics, in that each CO-Bot is given personalities, making some bots tricksters. Again the personalized decision model is tested against an existing bot. Compared to the first test, this test is expected to provide results that are less optimal. This expectation is due to the selfishness of tricksters. Keep in mind that the reason for developing tricksters is not to create an optimal bot, but to incorporate human characteristics into CO-Bot and thereby increase the interest in playing FPSs against bots.

By contributing to the development of a trust management system, the goal of increased interest in FPSs is assumed to have been reached, in that it is expected that human players joining a team of CO-Bots experience more interesting behavior than they would joining optimized bots. The trust management system extends CO-Bot with the ability to detect tricksters and to keep tricksters' influence at a minimum while at the same time being able to cooperate. Trust managing CO-Bots are tested against an existing bot and they are expected to do better than CO-Bots that do not incorporate a trust management system.

Finally a system is developed, that provide tricksters with the ability to optimize their trickery. This is meant as a countermeasure to trust management. It is expected that tests will show a slight decrease in success rate for the group as a whole against existing bots, compared to the case of having tricksters that do not optimize their trickery because the tricksters are harder to detect now. This contribution is expected to increase the interest in playing FPSs even further.

Chapter 2

Related Work

This chapter is concerned with related work on bot AI in FPSs. It provides a background and an incentive for implementing the group based model.

2.1 Intelligent Bots in FPS's

Previously some work has been done on making intelligent agents for deathmatch type First Person Shooters.

SOAR Quakebot

[6] explains a framework for incorporating anticipation in a bot for the game Quake II.

The bot (called Quakebot) uses the SOAR architecture. This architecture uses basic objects called operators. Each operator consists of primitive actions (in other contexts normally referred to as atomic actions), internal actions, and abstract actions. Primitive actions are actions like move, turn, and shoot. Primitive in this case means they cannot be divided into smaller actions. Internal actions are entities like remembering the last known position of the enemy. Abstract actions are high-level actions which must be dynamically decomposed into primitive actions at the bottom level. Examples of abstract actions could be goals like get-item or wander.

Figure 2.1 shows a partial operator hierarchy for the SOAR architecture. Only a small fraction of the architecture is shown in the figure but it clearly shows the general idea.

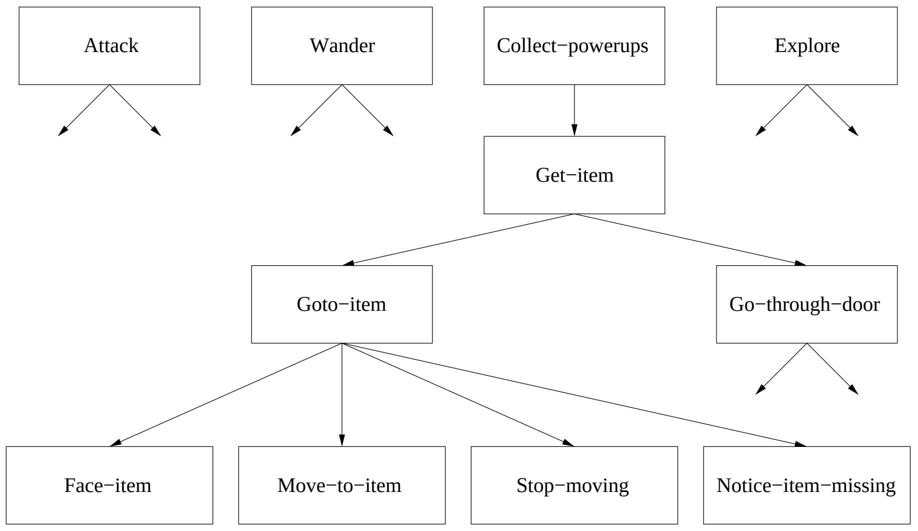


Figure 2.1: Partial operator hierarchy for the SOAR architecture. The most abstract actions are at the top level.

SOAR does not use any predefined ordering to determine which operators to select and apply. Proposition, selection, and application of operators are implemented using if-then rules. The rules are fired in parallel if they match the working memory of the SOAR implementation. The working memory holds all the bots' information about the current situation. Firing a rule alters the working memory by adding or deleting information.

The Quakebot is equipped with artificial intelligence through anticipation/prediction. This is used when the enemy is facing away from the bot and is done through maintaining an internal representation of the enemy bot's state. The Quakebot assumes the actions for the enemy is the same as its own and the prediction is done by contemplating what the bot would do itself if it was in the enemy's situation. The prediction stops if the bot comes to a point where uncertainty appears.

Influence Diagrams in FPS AI Module

[8] shows how to implement a deathmatch style bot using influence diagrams. The focus in the work is to learn to predict the outcome of an encounter with an enemy and use this information to choose the best decision in any given situation. The danger level of each area (part of the map/game world) is learned in order to avoid or seek an encounter in a specific situation. Any time a bot decides not to try to engage the enemy it will try to improve its state by picking up weapons and health packs scattered around the game world. Improving the bots internal state will make it more likely to win an encounter with an enemy and thus make the decisions taken by the bot

more aggressive.

Figure 2.2 shows the final influence diagram for making decisions about which tactic to chose in a given situation.

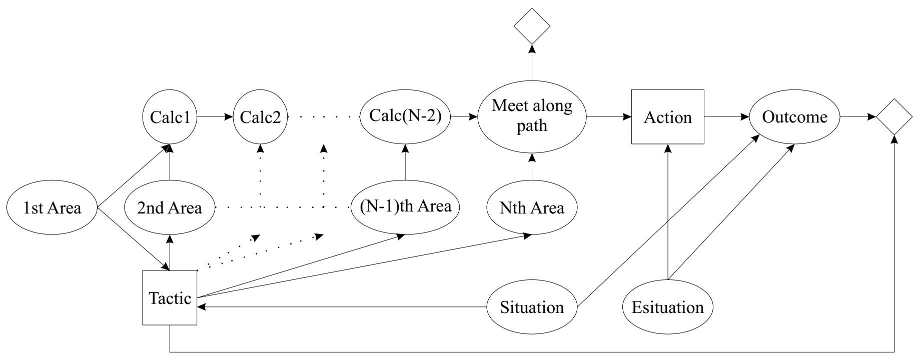


Figure 2.2: Influence diagram for bot decision making. Each area node represents the danger level of that area.

The decisions to consider both contain decisions about seeking an encounter and about trying to improve the bot's situation later on in the game by deciding to pick up items. Decisions, which are combinations of the two (like trying to pick up an item at the end of a dangerous route), are also available. The decision making is as one would expect based on the highest expected utility of the decisions in the influence diagram.

2.2 Learning Models of Other Agents Using Influence Diagrams

Influence diagrams have previously been used in a Multi Agent System (MAS). [12] shows how to model one other agent in an anti-air defense scenario using influence diagrams. Two bases (the agents) have a common goal of defending the ground from incoming missile attacks. Two missiles are launched at the ground at once and the bases must try to minimize the damage by intercepting the missiles before they hit the ground. Each base may launch just one interceptor. The idea is to minimize the damage to the ground when there is no communication between the bases.

The influence diagrams are used by each base to model the other. Utilizing this a base can to some extent predict the behavior of the other base and react accordingly. The influence diagram for deciding the best action of Base 2 (B2) is shown in Figure 2.3.

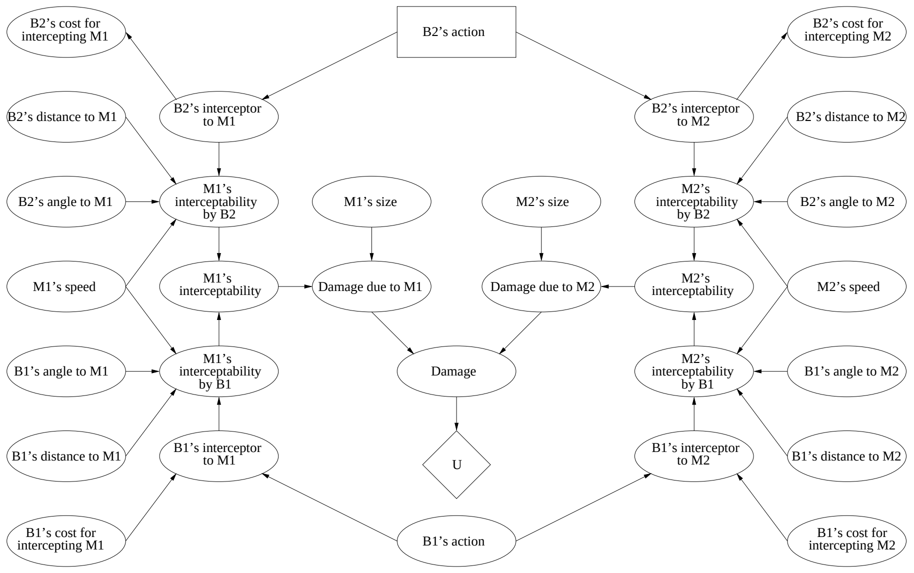


Figure 2.3: Decision model of defense unit B2. It includes a model of B1.

The authors of the article mentions what inaccuracies might occur when updating the influence diagrams because of a changing environment.

- The utility function of an agent might change. The new utility function has to be learned.
- Probability distributions in the model may be inaccurate. The correct probability distribution must be learned.
- The structure of the influence diagram may be wrong. A new structure that fits the observation data must be learned.

2.3 Learning Trust

[1] shows how to make bots choose to perform actions that reveal the trustworthiness of another bot. The article states there are three components to the model building process:

- Adopting an *a priori* or initial model.
- Engaging or observing the other agent in informative interactions.
- Updating the initial model based on such interactions.

Each of these components involve significant time and computational cost commitments by the modeling agent. The general idea is to estimate the nature of other agents in as few iterations as possible. The article presents a way of deciding on which actions to take to maximize the information revealed by the action the other agent chooses to take. This is possible because the other agent is expected to choose an action based on whatever action it sees other bots take. All this means it could learn the states of other agents more accurately and quickly.

2.4 Central and Decentral Approach to Group Decisions

[15] explains advantages and disadvantages of the central and the decentral approach to group based decision making. A decentralized approach is one where the agents exchange requests and intentions among themselves and the artificial intelligence is distributed equally among the group members. In a centralized approach the group has a leader which receives information and issues commands to other group members. Here the leader knows more about the game state than the other group members.

Chapter 3

Bayesian Networks and Influence Diagrams

This chapter is concerned with the theory of Bayesian networks which is used when constructing CO-Bot. The chapter is based on [4].

3.1 Definition of Bayesian Networks

Definition 1 A Bayesian network consists of the following

- A set of variables and a set of directed edges between the variables.
- Each variable has a finite set of mutually exclusive states.
- The variables and the directed edges form a Directed Acyclic Graph (DAG) (A directed graph is said to be acyclic if there is no directed path $A_1 \rightarrow \dots A_n$ such that $A_1 = A_n$.)
- A conditional probability table $P(A|B_1, \dots, B_n)$ is attached to each variable A with parents $B_1, \dots, B_n$.
- A variable A which has no parents has a initial probability table $P(A)$ attached.

A variable is in exactly one state, but there may be uncertainty about which state.

3.2 Example of a Bayesian Network

Figure 3.1 shows an example of a Bayesian network with four nodes. The probability tables to specify are P(B), P(D), P(A|B), and P(C|B,D).

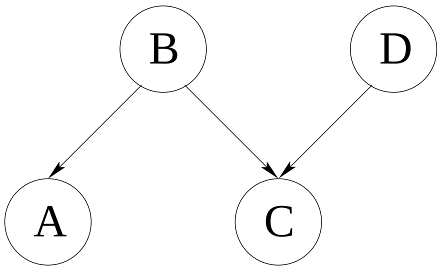


Figure 3.1: An example of a Bayesian network. The probability tables to specify are P(B), P(D), P(A|B), and P(C|B,D).

3.3 d-separation

One of the core elements in Bayesian networks is that the certainty of one variable can influence the certainty of other variables and vice versa. This leads to different types of connections. In the explanations of connections the term evidence will be used. Evidence on a variable is a statement of the certainty of its state. There are two kinds of evidence, hard and soft evidence. Hard evidence means that the exact state of a particular variable is known. In this case the variable is said to be instantiated. Soft evidence is evidence received from instantiated variables or from other variables containing soft evidence.

d-separation is a concept used when examining whether variables can influence each other. Two variables are said to be d-connected if evidence on one variable influences the other variable. If evidence on one variable cannot influence the other, the variables are said to be d-separated.

Serial Connections

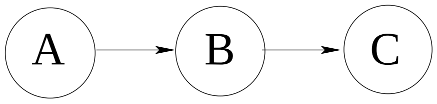


Figure 3.2: Serial connection. A and C are d-separated given B.

Figure 3.2 shows the concept of a serial connection. If there is evidence on the variable A, this has an influence on the certainty of B and then on C through B. Similarly, evidence on C will change the certainty of A through B. However, if B is instantiated then variables A and C cannot influence each other. It is said that A and C are d-separated given B.

Diverging Connections

An example of a diverging connection is shown in Figure 3.3. This construction makes it possible for variables A and C to communicate information. This communication between children is only possible if the parent variable is not instantiated. If the state of B is known, the children are said to be d-separated.

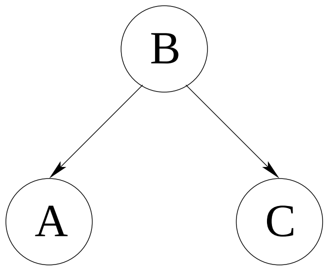


Figure 3.3: Diverging connection. A and C are d-separated given B.

Converging Connections

The last possible construction using three variables is converging connections. This concept is illustrated in Figure 3.4. The variables B and C are d-separated if neither variable A or one of its descendants of A has received hard evidence.

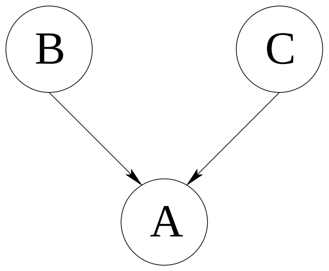


Figure 3.4: Converging connection. B and C are d-separated given A or one of A's decendants.

Example of d-separation

Figure 3.5 shows an example of a Bayesian network. This network will be examined to explain propagation of evidence in Bayesian networks.

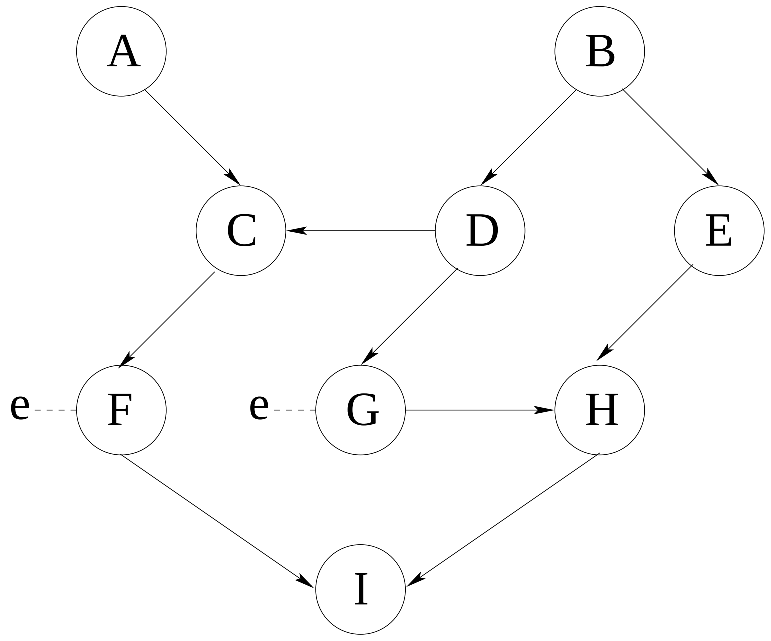


Figure 3.5: An example of d-separation. The variables A and I are d-connected.

Consider the situation where we want to examine whether variables A and I are d-connected or d-separated. The variables F and G are instantiated in this example. Evidence may propagate from A to D because it is a converging connection and a child of C (namely the variable F) is instantiated. Evidence may not flow from C to I through F because it is a serial connection and the middle variable (again F) is instantiated. No propagation from D to H may happen through G for the same reason. This means the last possibility is for evidence to propagate through the variable B to E. This is possible because this is a diverging connection where the parent (B) of the two variables has not received evidence. The connection from E to I through H is a serial connection without instantiated intermediate variable which makes it possible for evidence to propagate to the variable I. This means that the variables A and I are d-connected.

3.4 Divorcing

Consider the situation in Figure 3.6. If the variables A, B, C and D have five states each, the conditional probability table of variable E will have a total of $5^4 \cdot n$ entries where n is the number of states in variable E itself. If $n = 5$, the table in E will have a size of $5^4 \cdot 5 = 5^5 = 3125$ cells. This huge amount of cells may be very troublesome to handle for anyone working with the network.

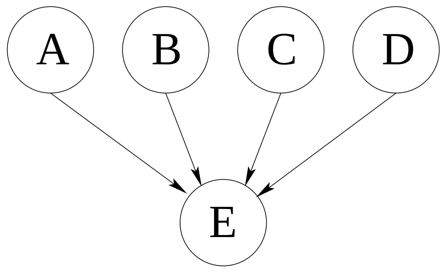


Figure 3.6: Network before divorcing variables A and B.

A way of reducing the effect of this problem is shown in Figure 3.7. The idea is to introduce a mediating variable thereby reducing the number of edges going directly towards the variable E. In this case the variable F is introduced and "collects" information from variables A and B in for instance 5 states. This technique is known as *divorcing*. In this case the new number of cells in E's table will be $5^3 \cdot 5 = 625$. Of course the table of the newly inserted variable F has a number of cells too but this is only $5^2 \cdot 5 = 125$ which still makes for far fewer cells in total. This all means that when a situation occurs where variables (in this case A and B) represent something which can be collected into a single variable, considering divorcing is a good idea.

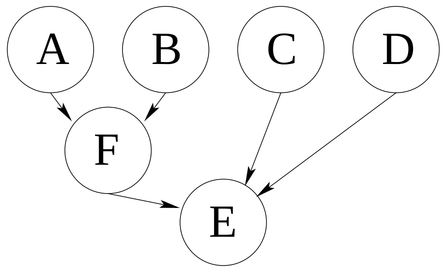


Figure 3.7: Network after divorcing variables A and B into a new node F.

3.5 Adaptation

When constructing a Bayesian network there are several ways of choosing the conditional probabilities for a model. However, there will almost always be a degree of uncertainty about the correctness of the probabilities chosen. This is called second-order uncertainty, and it raises an interesting question: Is it possible to reduce the second-order uncertainty in a systematic way? When a system is running new cases, it would be nice if the model could gain experience from these. In the following, a statistical approach will be described.

Fractional Updating

The task is to modify the parameters gradually with the new cases entered. To make this feasible, two simplifying assumptions must be made:

- Global independence: The second-order uncertainty for variables are independent of each other. This means that the conditional probabilities for the variables can be modified independently.
- Local independence: For each variable the second-order uncertainty of the distributions for different parent configurations is independent. In other words, given three variables A, B and C, where A is the child of both B and C, and given two configurations (b_i, c_j) and (b'_i, c'_j); then the second-order uncertainty of $P(A|b_i, c_j)$ is independent of the second-order uncertainty of $P(A|b'_i, c'_j)$, so the two distributions can be modified independently.

Now consider $P(A|B,C)$ and let all the variables be ternary. Under the assumptions given, $P(A|b_i, c_j) = (x_1, x_2, x_3)$ can be considered as a distribution established from a number of previous cases where (B, C) was in state (b_i, c_j). This makes it possible to express the certainty of the distribution by a fictitious *sample size* s. The larger the sample size the smaller the second-order uncertainty, which leads to working with a set of counts (n_1, n_2, n_3) and a sample size s such that $s = n_1 + n_2 + n_3$ and

$$P(A|b_i, c_j) = (\frac{n_1}{s}, \frac{n_2}{s}, \frac{n_3}{s}).$$

This idea is called *fractional updating*. Updating is best explained through an example. Consider the example where a case is received with evidence $e = \{A = a_1, B = b_i, C = c_j\}$. In this situation the probabilities are updated as follows:

$$x_1 := \frac{(n_1 + 1)}{(s + 1)}; x_2 := \frac{(n_2)}{(s + 1)}; x_3 := \frac{(n_3)}{(s + 1)}$$

Now, we get a case e with $B = b_i$ and $C = c_j$ and for A only a probability distribution $P(A|e) = P(A|b_i, c_j, e) = (y_1, y_2, y_3)$. In this case it is not possible to work with integer counts and updates are done as such: $n_k := n_k + y_k$ and $s := s + 1$. This means x_k is updated as follows:

$$x_k := \frac{(n_k + y_k)}{(s + 1)}$$

In general, a case is given which looks like $P(b_i, c_j|e) = z$. Then $s := s+z$. The distribution $P(A|b_i, c_j, e) = (y_1, y_2, y_3)$ is used to update the counts. The sample size is increased by z and thus $n_k := n_k + zy_k$. This leads to

$$x_k := \frac{(n_k + zy_k)}{(s + z)}$$

Unfortunately, fractional updating has one major drawback as it tends to overestimate the count of s, thereby overestimating the certainty of the distribution. As an example assume that $e = \{B = b_i, C = c_j\}$. This case tells nothing about $P(A|b_i, c_j)$, but fractional updating will still add one to the count of s, and thereby take it as a confirmation of the current distribution.

3.6 Influence Diagrams

Definition 2 An **influence diagram** consists of a DAG over chance nodes, decision nodes and utility nodes with the following structural properties:

- There is a directed path comprising all decision nodes.
- The utility nodes have no children.

For the quantitative specification, we require that:

- The decision nodes and the chance nodes have finite set of mutually exclusive states.
- The utility nodes have no states.
- To each node, A, a conditional probability table $P(A|pa(A))$ is attached. $pa(A)$ is called the parent set of A and consists of all parent nodes to node A.
- To each utility node U a real-valued function over $pa(U)$ is attached.

Any link from a chance node into a decision node is called an *information link* and indicates that the state of the parent is known prior to the decision being made. A network is equipped with an information link from a chance node, C, to a decision node, D, if C is observed prior to D but after any decision made before D.

Links between decisions are called *precedence links.* These are present to express in which order decisions are made.

Links between chance nodes are called *functional links* and are similar to the ones found in Bayesian networks. Links from chance nodes to utility nodes are named the same.

3.7 Example of an Influence Diagram

To explain what an influence diagram is, consider an example. This semester a student is going to have four different courses. At the start of the semester he must decide how to distribute his work effort among them. He may work normally in each course but may also work harder in one or more courses at the expense of one or more of the other courses. Halfway through the semester he must choose for which three of the courses he wishes to take the exam. His grade will depend on how difficult the courses are (something that he will not know at the start of the semester), how much work effort he has put into them and, of course, for which he actually takes the exam.

This situation can be represented as a Bayesian network extended with decision and utility nodes, this can be seen in Figure 3.8.

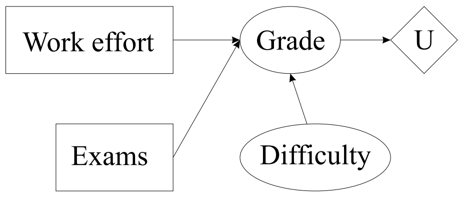


Figure 3.8: Bayesian Network extended with decision and utility nodes for the school problem.

To represent the order of decisions the network is extended with precedence links. In the example this means that a link should be drawn from the decision *Work Effort* to the decision *Exams*. This can be seen in Figure 3.9, which is called an influence diagram, where the chance node *Difficulty* is observed before the decision *Exams* is made.

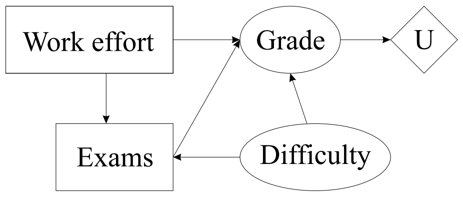


Figure 3.9: Influence diagram for the school problem.

Note that a precedence link between a decision node, D_1, and another decision node, D_2, is redundant and can be removed if there is a another directed path from D_1 to D_2.

Chapter 4

Counter-Strike

This chapter provides a description of the various aspects of Counter-Strike. Weapons, equipment and the different kinds of missions available are explained. The chapter is concluded by a description of how to interface Counter-Strike. The information in this chapter has been collected from [14].

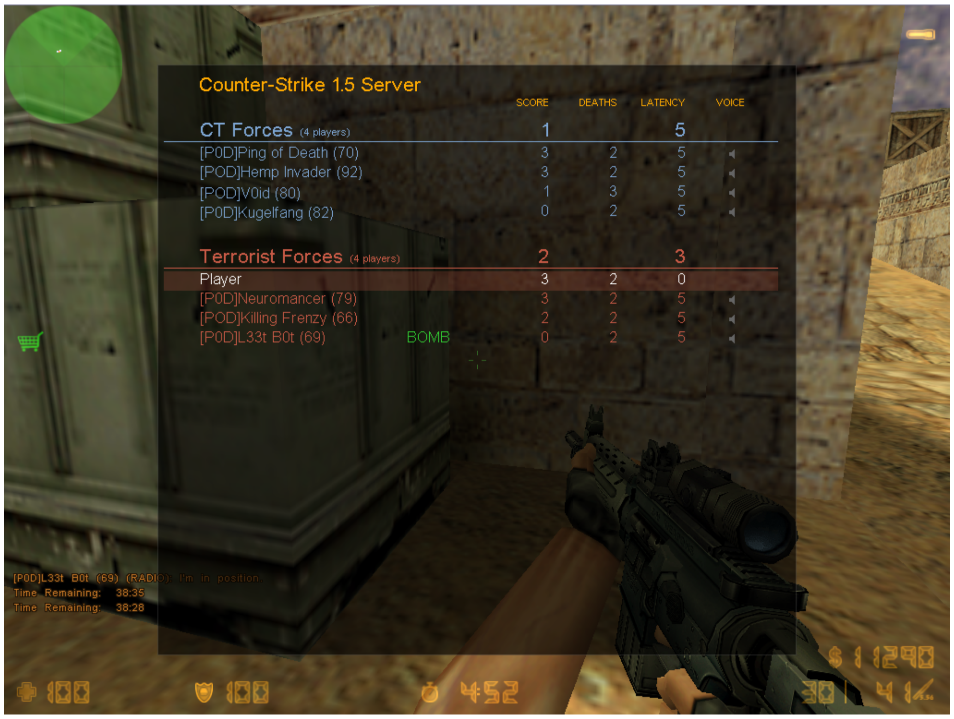
4.1 What is Counter-Strike

Counter-Strike is a team based First Person Shooter modification for the game Half-Life by Valve Software. It is set in a Terrorists vs. Counter-Terrorists environment and is based on a somewhat realistic game world.

A game consists of several rounds on the same game map where the two teams each try to accomplish a certain goal. After an amount of time set by the server administrator the map is changed. The goal depends on the type of game (map) chosen and will be explained later in this chapter. After each round the players are awarded credits which can be used to buy equipment at the beginning of each round. The amount of credits earned in a round is dependent on many factors but basically the winning team are awarded the most credits.

If a player survives a round she keeps any equipment currently in her possession. The player is still awarded credits and may choose to buy better equipment or perhaps buy nothing and save the credits for use in a later round. The equipment is always lost if the player dies or a new game map is loaded.

A big part of Counter-Strike is the in-game stats. This is a kind of score board containing both information about the kills/deaths of individual players as well as wins/losses of rounds for each team. Figure 4.1 shows a screen shot of the in-game stat page.


Counter-Strike 1.5 Server

		SCORE	DEATHS	LATENCY	VOICE
CT Forces (4 players)		1		5	
[P0D]Ping of Death (70)		3	2	5	◂
[POD]Hemp Invader (92)		3	2	5	◂
[POD]V0id (80)		1	3	5	◂
[P0D]Kugelfang (82)		0	2	5	◂
Terrorist Forces (4 players)		2		3	
Player		3	2	0	
[P0D]Neuromancer (79)		3	2	5	◂
[POD]Killing Frenzy (66)		2	2	5	◂
[P0D]L33t B0t (69)	BOMB	0	2	5	◂

```
[P0D]L33t B0t (69) (RADIO): I'm in position.
Time Remaining:   38:35
Time Remaining:   38:28
```

Figure 4.1: A screen shot of the in-game stats page.

The individual stats are the main reason for some players acting selfish and doing what is best for themselves and not what is best for the team as a whole. Some players play for the team and other play for themselves.

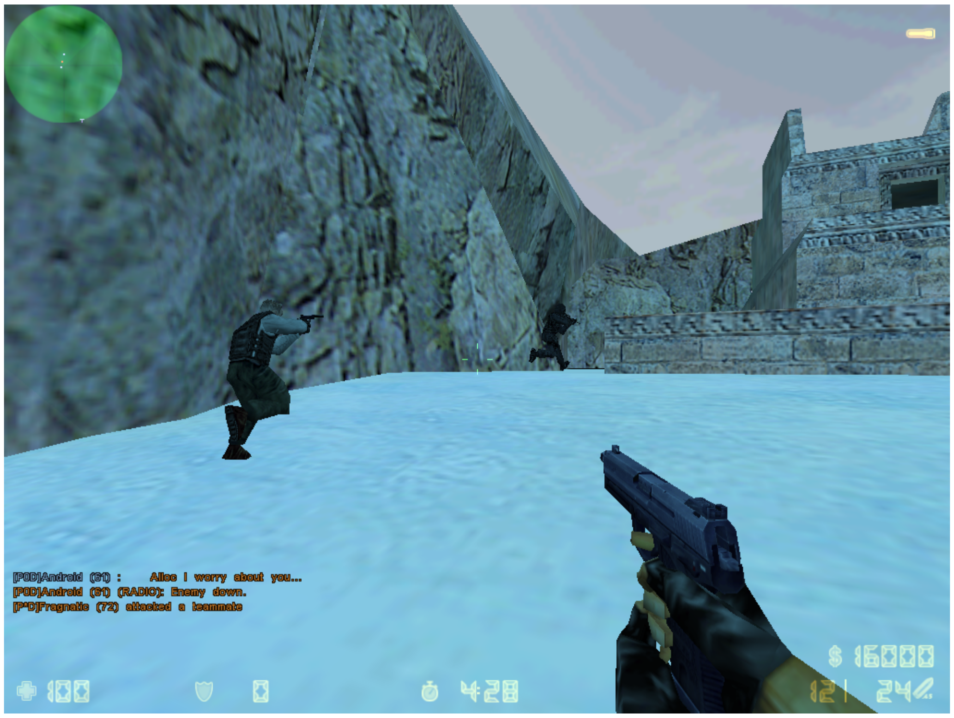
4.2 Mission Types

In the game there are several mission types to choose from. Each of these has a very different gameplay and thus different tactics. All mission types are described in this section.

Assassination

In this game type one of the counter-terrorists is chosen at random to be a VIP (Very Important Person). The VIP is only equipped with a knife, a handgun, and ample body armor at the start of each round and cannot buy any weapons or equipment (this includes ammo). Furthermore it is not possible for the VIP to pick up items. The goal of this mission is for the VIP to reach a certain destination on the map within a predetermined time limit for the counter-terrorists to win the round. The only other option for the counter-terrorists to win is to kill all terrorists on the map. All teammates of the VIP may buy equipment as usual and it is their assignment to protect the VIP from the terrorists who wish to kill him. The terrorists win if the VIP is killed or the destination is not reached within the time limit.

Figure 4.2 shows a screen shot from an assassination mission from the counter-terrorists point of view. The nearest team mate in the picture is the VIP.



```
[P0D]Android (61) :     Alloc I worry about you...
[P0D]Android (61) (RADIO): Enemy down.
[P*D]Fragnatic (72) attacked a teammate
```

Figure 4.2: A screen shot from an assassination type mission. The nearest team mate in the picture is the VIP.

Bomb Defusal

In this mission type the terrorists are the ones under pressure to accomplish a task. They must plant an explosive device in one of several possible locations and see that it is not disarmed by the counter-terrorists. Furthermore they do not wish to be too close to the bomb when it explodes because this will kill them and cause them to lose their current equipment. If the bomb is detonated within time or the entire counter-terrorist team is killed the terrorist team wins. The game only contains one bomb and it is assigned to a random terrorist at the start of a round. If it is dropped (either by purpose by the player herself or because the carrier is killed) it may be picked up by any player on the terrorist team.

The counter-terrorists must either kill all terrorists before the bomb is placed, disarm the bomb after it has been planted or let the time run out to win the game.

Figure 4.3 shows a terrorist planting the bomb at a bomb site.

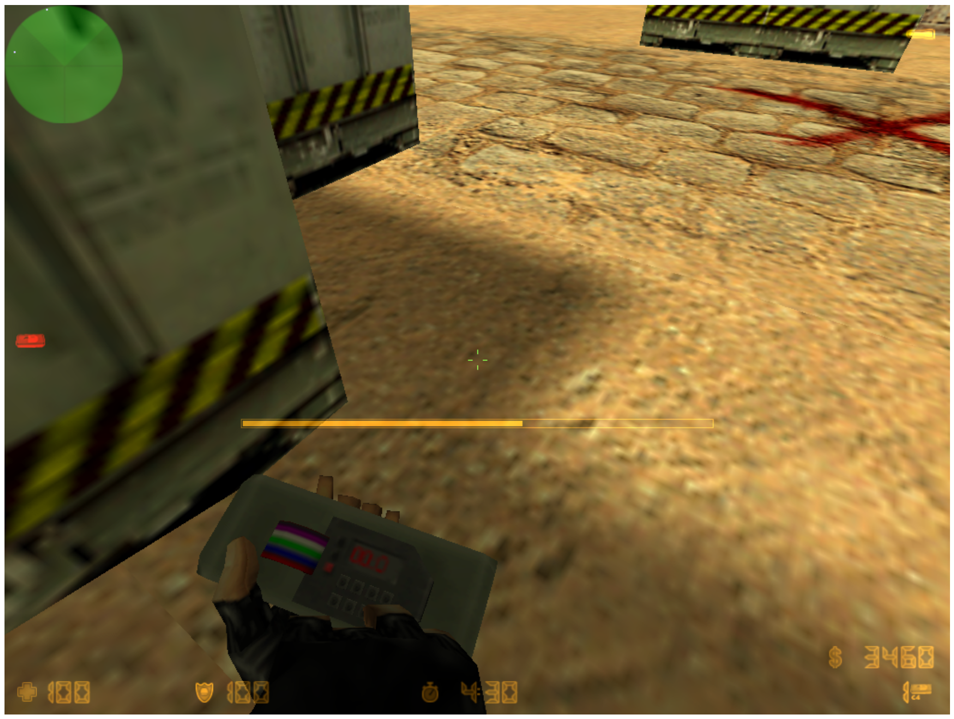


Figure 4.3: A terrorist planting the bomb at one of the bomb sites.

Hostage Rescue

The hostage rescue type missions are set in a scenario where the terrorists have taken a number of people hostage. The counter-terrorists must either kill all terrorists or rescue the hostages by bringing them to the hostage rescue zone within the time limit to win the round. Otherwise the terrorists win.

If a player (regardless of team) kills a hostage, credits are deducted from the players account. Counter-terrorists earn extra credits rescuing hostages because it is not an easy task first reaching the hostages and then bringing them to the rescue zone without being killed.

An example of what a hostage character who must be rescued by the counter-terrorists might look like is shown in Figure 4.4.

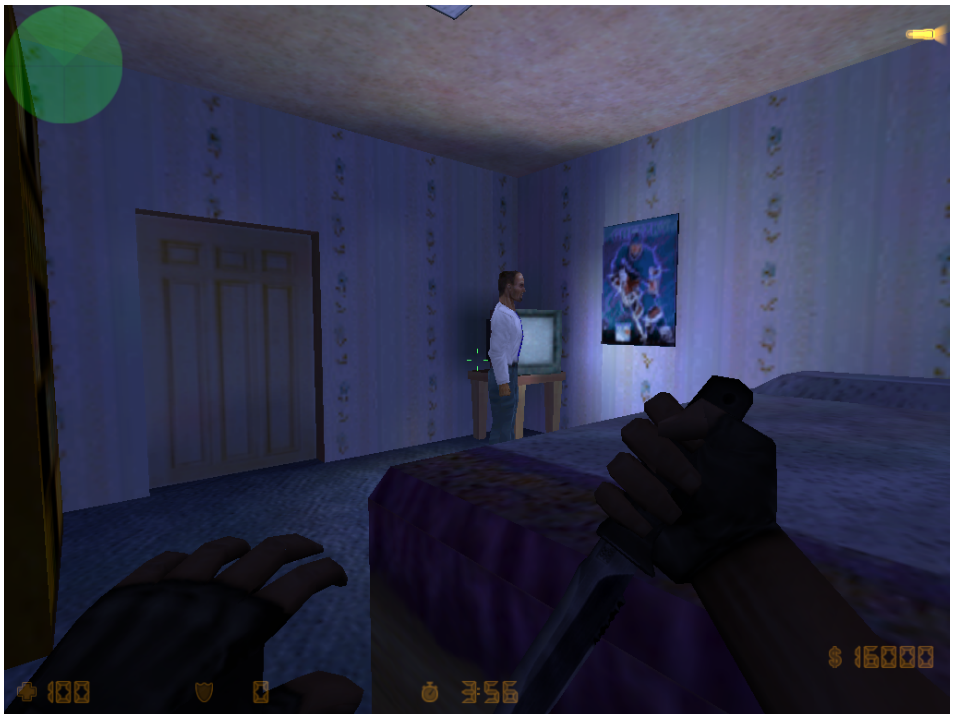


Figure 4.4: A hostage being held captive by the terrorists in a dark room.

4.3 Equipment

In Counter-Strike you are able to carry only a limited amount of weapons. You always start with a knife and a handgun (secondary/back-up weapon). Which handgun you start with depends on the team selection. Furthermore you may carry one large gun (your primary weapon) if you desire and have the credits to buy one or are able to pick a discarded or lost one up from the ground. Any weapon bought can not be sold again and if the player wants to replace the weapon she may throw it on the ground to make room for another. This also means it is possible for a player to buy weapons for a team mate by first buying one for the team mate, throw it on the ground and then buy another for herself.

Besides these items you may have extra equipment like grenades and armor to help you out.

Weapons

All the different kinds of weapons and other in-game items are described in this section.

Knife

All players are equipped with a knife at the beginning of each round. This weapon can be used only at really close range, and is mostly useful when you are out of ammo for any other weapon.

Hand guns

The hand guns are not very accurate, powerful, or fast but they are cheap and can be carried alongside a bigger weapon. A player is given a handgun at the start of each round. Which gun depends on which team the player belongs to.

Shotguns

Shotguns are excellent short range weapons. On longer ranges however their damage is minimal.

SMG's

Sub Machine Guns (SMG's) are good in close quarter combat. They are not expensive and deliver quite a punch in short time. On longer ranges they are not great because their accuracy is inferior to the bigger automatic rifles.

Machine Guns

The machine gun is a powerful high ammo capacity weapon but very inaccurate. Because of this inaccuracy it is mostly useful for laying suppressing fire. Because of the high price of this weapon compared to its utility it is not a frequently used one.

Rifles

There are essentially two kinds of rifles. The automatic rifles which can be regarded somewhat the same as the SMG's only more powerful and accurate, is one class of rifle. The other class is sniper rifles which are slow firing but deliver a powerful punch when they hit. Some automatic rifles and all sniper rifles are equipped with scopes which makes aiming easier. In the case of sniper rifles aiming without the scope is nearly impossible.

Other Items

Bomb Defusal Kit

The bomb defusal kit is needed only in bomb defusal missions. It decreases the time used to disarm a bomb by half and may thus mean the difference between making it in time or die trying.

Kevlar Vest

The kevlar vest protects the person wearing it from bullets to some extent. It covers the torso area which is the largest and also an area where a hit causes a high degree of damage.

Kevlar Vest and Helmet

This combined set of armor consists of a kevlar vest and a helmet. The helmet protects the head area which is the most vulnerable body part on the player. The vest is the same as before.

Smoke grenade

The smoke grenade is used to provide cover. When thrown it expels into the surrounding environment a thick grey smoke which makes for great cover to escape from or advance on the enemy.

HE Grenade

The HE grenade (High Explosive Grenade) is a regular hand grenade. It explodes within a few seconds of being thrown and causes damage to anyone within the surrounding area. Like all other grenades it can be thrown over objects and bounce off walls to hit the target.

Flashbang

The last kind of grenade. When this explode it does not cause damage but rather blinds anyone looking at the explosion. This blindness occurs for a period of time (seconds) gradually getting better. The more directly any person looks at the explosion the longer the blindness will last. These grenades are mainly used for blinding enemies in a room before entering. Sometimes flashbangs backfire because an enemy is able to turn around before it explodes and not get severely affected. This way the enemy is ready for an oncoming assault where the one who threw the flashbang thinks the enemy is blinded.

Night Vision

In dark areas the enemy may hide and be hard to spot. Using night vision goggles makes the enemy visible even in dark environments. This article is not useful on all maps but in certain situations it gives the wearer a distinct advantage.

Flashlight

The flashlight allows players to see in dark places. Unlike the night vision goggles the flashlight is also visible to other players. This for example means an enemy is able to see if a flashlight is used by the opposing team to examine the nearby area. This item is free and part of the standard equipment pack.

4.4 Teamwork

A big part of Counter-Strike is teamwork. The player has access to a radar showing the location of all team mates. Besides this, radio messages may be used to communicate with all living team mates. They may communicate messages like ”Storm the front”, ”Stick together team”, ”Follow me”, ”Negative”, ”affirmative” and much more.

In Counter-Strike any player can see stats when desired. This means it is always known to any player which players are still alive on any of the two teams regardless of which team the player is on. This is important information for selecting the correct tactic.

4.5 Interfacing Counter-Strike

Half Life was designed to be modified by upcoming game programmers and other enthusiasts. It comes with a complete standard developer kit (SDK) to interface the functionality of the Half Life engine. As already stated, counter strike is such a modification. However, counter strike does not support computer controlled characters or bots, and unfortunately, no source code exists that allows modifications of counter strike itself. Therefore, creating a bot normally requires making a new modification for the Half Life engine using the SDK provided. Ultimately, this amounts to creating a completely new game.

Since creating an entirely new game in the course of one semester is impossible, the solution (as proposed by Botman [2]) is to create an intervening dynamically linked library (dll). An intervening dll is a piece of software that captures all function calls from the Half Life engine to Counter Strike and vice versa. This intervention happens without the knowledge of either - that is, to the Half Life engine, the intervening dll is just another modification and to Counter Strike the intervening dll acts like the Half Life engine. Indeed one could filter out or add functionality before communicating the original function call back and forth between the two. In this way bots are spawned in the game as any other player, except they are controlled by an AI that resides in the intervening dll. Thereby bots have exactly the same possibilities as human players.

Chapter 5

Existing Bots

This chapter investigates some of the existing Counter-Strike bots in order to reveal weaknesses that should be remedied in this project. Two of the most prominent bots will be investigated: The HPB Bot [2] and the PODBot [5].

As a common denominator, these bots rely heavily on the existence of waypoints in the game level placed by a human. A waypoint marks an interesting spot in the game level, e.g. a camping spot, a doorway or just somewhere in the game level that the bots should cover now and then. Figure 5.1 shows a screen shot from Counter-Strike with waypoints marked by sticks of different colors. Green sticks mark "normal" waypoints that bots should cover now and then, the red stick marks a waypoint that is especially important to terrorist, and the small light-blue stick marks a camping waypoint where the bot should crouch. There are several other types of waypoints that are not shown in the figure.

Without waypoints, bots have no means of navigating through the world except for the raw image of the rendered world. Since performing thorough image analysis is a very time-consuming and error prone task with no real advantage compared to using waypoints, this method is not used by any well-known bots.

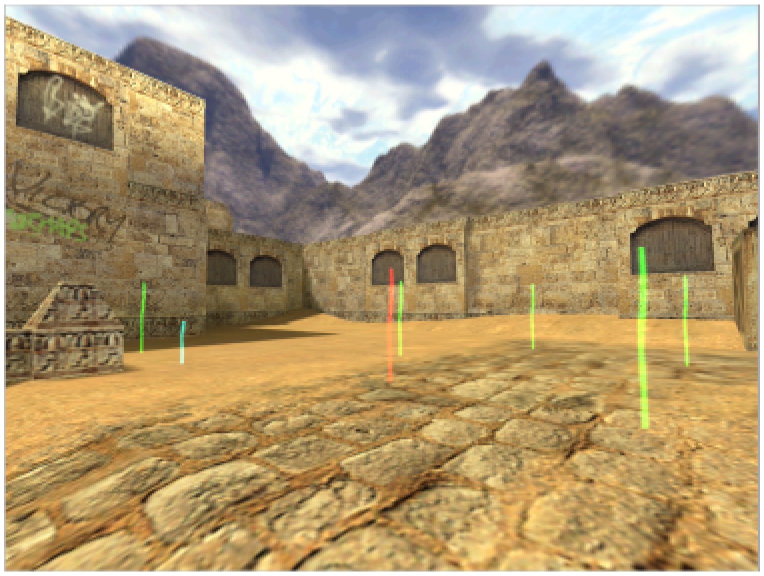


Figure 5.1: Waypoints in Counter-Strike. Each type of waypoint has its own color.

5.1 The HPB Bot

As described in Section 4.5 Jeffrey Broome pioneered a method for creating bots for mods like Counter-Strike. His creation, the HPB Bot, is meant as a basis for creating other bots and the source code is therefore available at no expense. It mainly consists of template code pointing out where future bot developers should focus. As a consequence, the HPB Bot in itself is not an interesting adversary, as the bots basically stick to themselves, almost never engaging in combat. Thus HPB Bot functions as a shortcut to interfacing Counter-Strike.

5.2 The PODBot

PODBot is, as all other well-known Counter-Strike bots, based on the HPB Bot. PODBots are able to perform Counter-Strike specific actions such as purchasing weapons, placing a bomb in the bomb defusal missions, rescuing hostages and so forth. The PODBot is unable to perform sophisticated plans that regard the team as a whole. It is limited to recognize e.g. that in a specific situation it might be sensible to support another team member in his plan. Otherwise, most decisions are only regarding the bot itself.

Furthermore, most of the decisions are scripted. Listing 5.1 shows a typical example of the scripting of the bot. In the example, the bot stumbles upon a camping waypoint and considers whether to camp. The code dictates that if the bot has the primary weapon and it has not camped there for a long time, then it should camp there now. With this type of scripting, the bot does not learn e.g. whether it was a good choice to camp with this type of primary weapon.

```
// Reached Waypoint is a Camp Waypoint
if (paths[pBot->curr_wpt_index]->flags & W_FL_CAMP) { // Check if Bot has got a primary weapon and hasn't camped before
  if ((BotHasPrimaryWeapon(pBot)) &&
  (pBot->fTimeCamping+10.0<gpGlobals->time)) { //[...] //Perform camping
```

Listing 5.1: A piece of code from PODBot that determines whether or not to camp.

Another example of the scripted decision making can be seen in Listing 5.2 which is a part of a function returning the type of waypoint, the bot chooses to pursue. `iRandomPick` has been assigned a random value between 0 and 100. As can be seen from line 1, 3 and 5, the probability for the bot to choose a certain waypoint is set by hand. The bot is not able to learn that e.g. there should be a higher probability of pursuing the camp waypoint.

```
1  if (iRandomPick<30)
2    return(g_rgiGoalWaypoints[RANDOM_LONG(0,g_iNumGoalPoints)]);
3  else if (iRandomPick<60)
4    return(g_rgiTerrorWaypoints[RANDOM_LONG(0,g_iNumTerrorPoints)]);
5  else if (iRandomPick<90)
6    return(g_rgiCampWaypoints[RANDOM_LONG(0,g_iNumCampPoints)]);
7  else
8    return(g_rgiCTWaypoints[RANDOM_LONG(0,g_iNumCTPoints)]);
```

Listing 5.2: Script from PODBot to determine which waypoint to pursue.

5.3 Other Bots

Lots of other bots for Counter-Strike exist, but a common characteristic for all of them is that they focus on refining the scripts that perform the decisions made instead of refining the actual decision making. [2] provides a number of links to bot development projects.

Part II

CO-Bot

Chapter 6

Basis for Modeling CO-Bot

This chapter concerns itself with explaining the basic elements of designing CO-Bot and furthermore explains the test environment which is common to all iterations of the CO-Bot implementation.

6.1 An Incremental Approach to Modeling.

This next four chapters comprise the incremental development of a cooperative multiplayer bot called **CO-Bot** which is a terrorist-only bot for Counter-Strike. Initially a cooperative decision model is developed throughout Chapter 7. This decision model enables a group of CO-Bots to make the best possible decisions for the group as a whole. In order to make CO-Bot more human-like, an urge of tricking team mates for personal gain, using phony information, is introduced in Chapter 8. CO-Bot must now consider not only what is optimal for the group as a whole but also whether a given individual goal is fulfilled optimally. Chapter 9 extends CO-Bot with the ability to detect tricksters to some extent, and finally Chapter 10 extends the capabilities of tricksters to be able to trick other bots in a sophisticated manner and thereby exploit trickery to the limit.

Common to all decision models, is the ability to distribute roles among CO-Bots to apply a strategy. The following defines the terms "role" and "strategy" more precisely.

Definition 3 A specific task that is part of a cooperative plan is called a **role**.

Definition 4 A set of roles used to fulfill a cooperative goal is called a **strategy**.

Note that any CO-Bot can take on any role even if the CO-Bot is unfit to do so. For instance, if the role is to provide for sniper cover, a CO-Bot that does not posses a sniper rifle can still take on this role. In that case the sniper cover is less likely to actually provide adequate cover.

6.2 Attack Strategies

When a group of CO-Bots wish to attack, a set of different strategies for doing so should be considered. Each strategy has distinct tasks which must be carried out. This means different strategy-specific roles must be assigned to the bots. For each possible group size, a number of different strategies are available. Table 6.1 shows the different strategies available for each possible group size. The group size is limited to 3 bots for test purposes to minimize the number of cases which must be collected and the number and complexity of the influence diagrams used in the implementation.

Group size	Possible strategies
1 bot	- Storm - Hold back
2 bots	- 2 bots storm - 1 bot storms and 1 bot lays down suppressing fire - The 2 bots both suppress
3 bots	- 3 bots storm - 2 bots storm and 1 bot holds back providing suppressing fire - 1 bot storms while the other two lays down suppressing fire - The 3 bots all suppress

Table 6.1: Possible strategies for different group sizes.

[9] shows an example of how a Special Weapons And Tactics (SWAT) team operates when entering a room possibly occupied by the enemy. The group is organized in a line by the doorway and the first SWAT officer throws a grenade (a flashbang) inside the room. The group now enters the room as quickly as possible to clear the room of any enemies.

The tactics mentioned in Table 6.1 are not in total compliance with the general idea of [9] when storming an enemy stronghold because the roles of team mates are quite similar and Counter-Strike is not a game based on exclusively realistic situations like the Rainbow six series of games (see [3]). The idea of most strategies having quite different roles to choose from makes the choice of which role to choose more interesting than if the roles were more similar. Even if the strategies are not comparable to real life strategies they are still interesting from a theoretical point of view. The differences in danger level associated with and expected utility of each role should make for an interesting series of choices for each bot.

6.3 Test Environment

The test environment is the defusal map de_dust depicted in Figure 6.1.

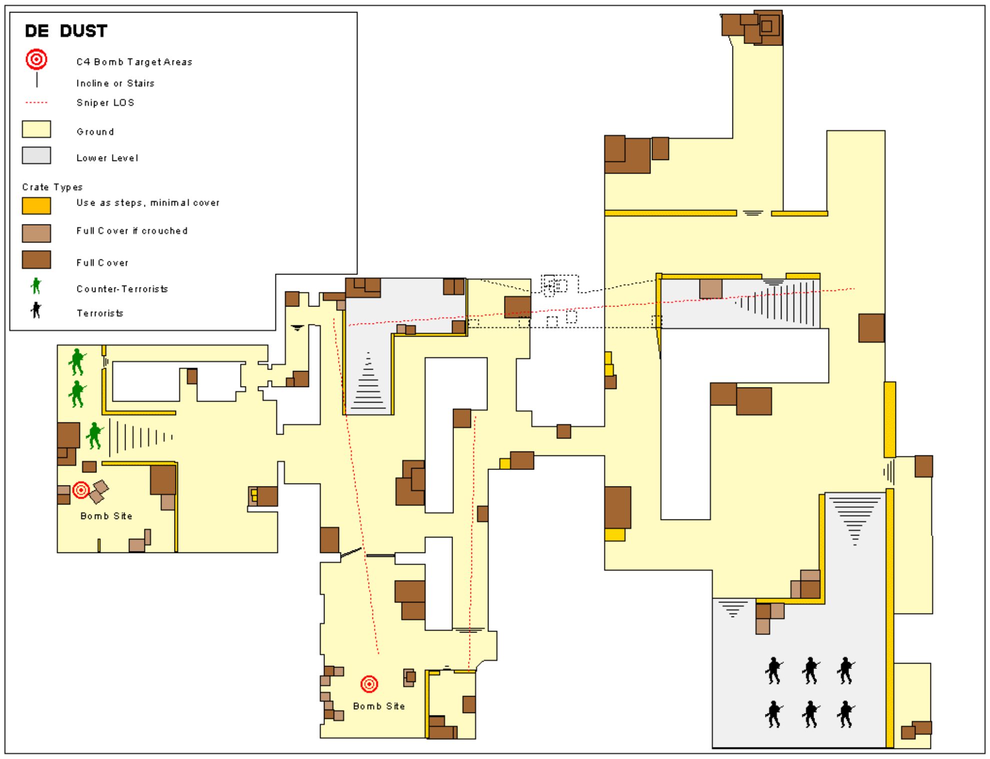


Figure 6.1: Overview of the de_dust map. Terrorists start in the lower right corner and counter-terrorists start in the mid left

The map contains two bomb sites and numerous ways to reach them from the terrorist's starting point. Figure 6.2 shows 19 points used to build the different routes the terrorists may take from their starting point in order to reach either one of the two bomb sites.

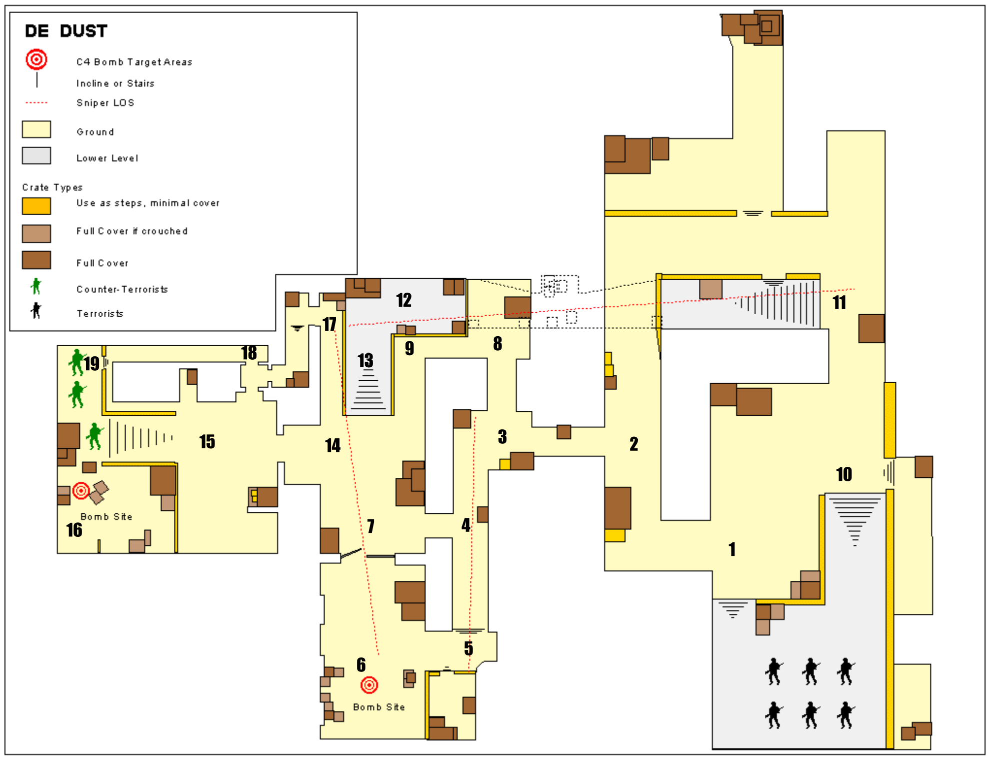


Figure 6.2: Points used to construct routes to the two bomb sites on the de_dust map.

Please note that the points in the figure are not equivalent to the waypoints actually used in the implementation but are only for explanatory purposes. The real number of waypoints used in the implementation is about 500.

There are a total of 10 routes to choose from, some being more "sneaky" than others and some being more fast and direct. This diversity in routes makes it virtually impossible for the counter-terrorist team to determine where the terrorist team will emerge next. This is the case even if the counter-terrorist team has human players on it. The route is chosen using a weighted random function as this is not the focus of the project. For information on how to implement and design a route decision model and more, consult our previous work on the topic (see [8]).

Table 6.2 shows the different routes implemented in the game on the de_dust map, by connecting the points from Figure 6.2. Some routes which are possible to construct in the map are not available as they are simply not considered feasible and a number of 10 routes is surely adequate for a sensible test environment.

Bomb site 1 is the one located near point number 6 and bomb site 2 is the one located near point 16 just by the position where the counter-terrorists start.

Destination	Route
Bomb Site 1	- 1-2-3-4-5-6 - 1-2-3-4-7-6 - 1-2-3-8-9-7-6 - 10-11-12-13-7-6
Bomb Site 2	- 1-2-3-8-9-14-15-16 - 10-11-12-13-14-15-16 - 1-2-3-4-7-14-15-16 - 1-2-3-4-7-14-17-18-19-16 - 10-11-12-13-14-17-18-19-16 - 1-2-3-8-9-14-17-18-19-16

Table 6.2: Implemented routes for CO-Bots in de_dust.

Chapter 7

The Cooperative Model

This chapter is concerned with the creation of a cooperative decision model. It is based on influence diagrams, and optimizes decisions taken by a group without consideration to any individual bot's needs and thus optimizes the group's behavior.

7.1 Motivation

The motivation for modeling the decision process of any bot by the use of influence diagrams is to let the artificial intelligence of the bots go beyond the scopes of ordinary scripting. Using influence diagrams in an implementation makes the bots perform actions based on the learned values, and modeled correctly they can adapt to any player type. Another motivation is that incorporation of team work makes for much more interesting opponents.

7.2 Cooperative Model Design

This section describes a simple decision model, which enables CO-Bot to select the best strategy and the best combination of roles to fulfill the selected strategy.

Figure 7.1 illustrates the model for selecting the best strategy. The figure can only select the best strategy for a team of three bots. This is reflected several times in the model: One example, is the decision node, *Strategy*, which contains the states described in Table 6.1 under *3 bots*. Another is the three decision nodes *Weapon_1*, *Weapon_2* and *Weapon_3*.

As illustrated in the figure, the selection of a strategy is ultimately based upon an assessment of the fighting capabilities of the enemy group versus the fighting capabilities of the friendly group. In order to assess a group, the nodes *Avg_health* and *Avg_armor*, are calculated programmatically as an average of all its members. For instance, a group's *Avg_health* is calculated

as
$$\frac{healthBot1 + healthBot2 + ... + healthBotN}{N}$$

These two nodes both have three states; 1-33, 34-66 and 67-100, and together with the number of individuals in the group, represented by *Friendly_number_of_group_members*, they constitute the combined *Friendly_danger_level* of the group. The states of the node *Friendly_number_of_group_members* are: 1, 2 and 3 and the states of the node *Friendly_danger_level* are: low, medium and high. The nodes *Avg_health* and *Avg_armor* are easily combined because they both correspond to an exact in-game representation – namely a number between 0 and 100.

After having chosen a strategy, the bots must be distributed on a number of roles. The best distribution depends on the number of bots in the group. In fact the number of roles that needs to be occupied corresponds exactly to the number of bots in the group. Therefore, there are several influence diagrams for distributing the roles among them. For each strategy, there is one influence diagram for each possible number of bots. That is, if the number of bots is limited to five and the number of strategies is limited to three, there are fifteen different influence diagrams for distributing the roles among them.

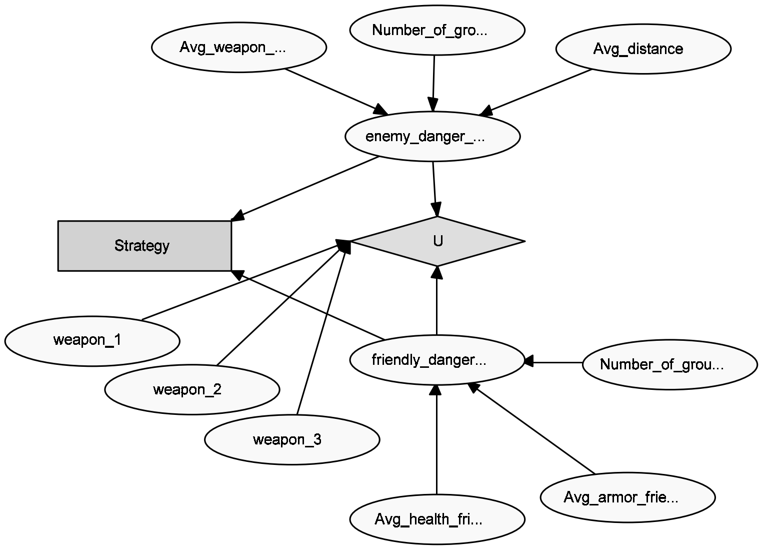


Figure 7.1: Desired influence diagram for selecting a cooperative strategy.

Figure 7.2 suggests a possible model for distributing three roles. Instead of combining the friendly group's nodes (as was the case in Figure 7.1), each individual bot's nodes are now considered. It is worth noticing, that the utilities in the utility node *U* still optimizes the group's chances for success and never favors a particular bot.

As can be seen in Figure 7.2, the utility node depends on eleven nodes each with three states: Four probability nodes: *danger_level_1*, *danger_level_2*, *danger_level_3*, and *enemy_danger_level* all with states low, medium and high, three decision nodes: *Storm_1*, *Storm_2* and *Suppress* with states Bot1, Bot2 and Bot3, three probability nodes: *weapon_1*, *weapon_2* and *weapon_3* with states short_range, medium_range and long_range and finally the probability node *Friendly_number_of_group_members* with states 1, 2 and 3. This amounts to 177147 different utilities that need to be set – either by hand or through a utility-learning technique (see [11]). The former is a completely impossible task, while the techniques for the latter is still undergoing heavy development. Even if the techniques were fully developed, they would undoubtably require an unacceptably large number of cases in order to achieve a meaningful result.

Common to both models is the divorcing of *health* and *armor* into *danger_level*. One could argue, that *weapon* nodes should also be divorced into a group's *danger_level*. However, it was chosen to keep explicit control over utilities on a per-weapon basis while health and armor were deemed more coherent.

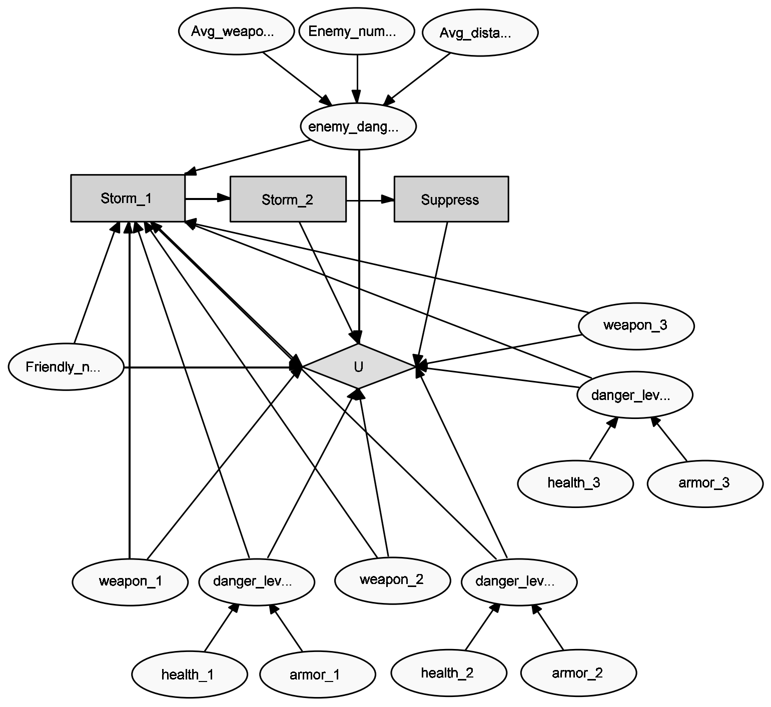


Figure 7.2: Desired influence diagram for selecting a distribution of roles.

Since the utility nodes of Figures 7.1 and 7.2 contain many entries, setting these utilities by hand would require a great deal of work and understanding of each situation. The structures of Figures 7.3 and 7.4 are approximations that remedy this situation.

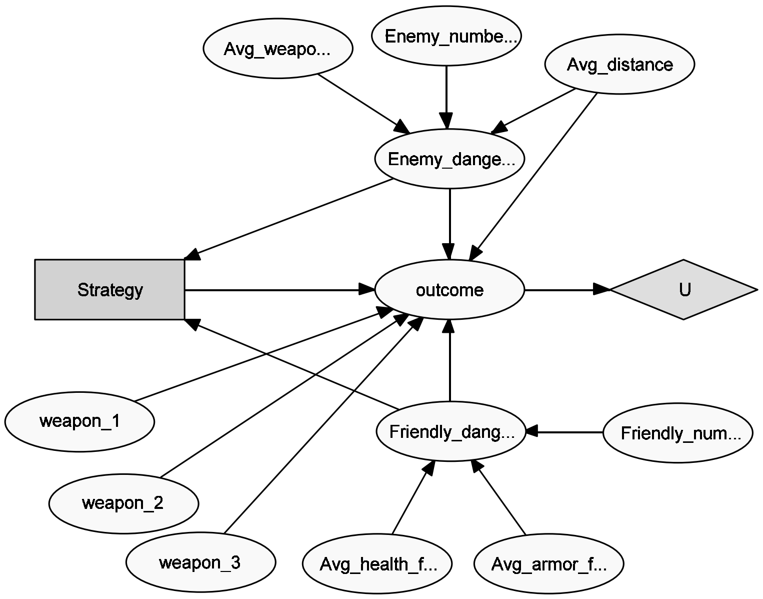


Figure 7.3: Approximation of the original desired network. An outcome has been introduced, in order to allow fractional updating.

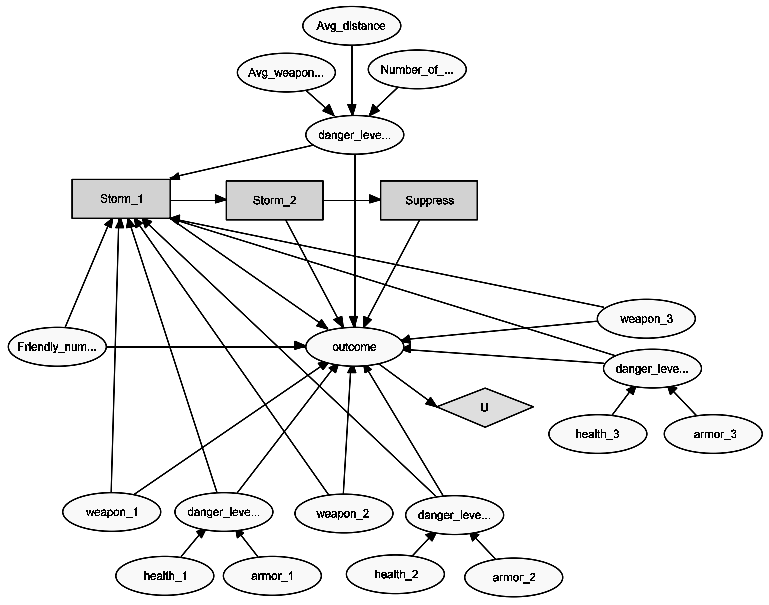


Figure 7.4: Approximation of the original desired model for distributing roles. An outcome has been introduced, in order to allow fractional updating.

The difference is the replacement of the utility nodes with chance nodes which contain a set of states representing outcomes of an encounter. The ”outcome” chance node has a utility node attached to it giving each possible outcome a utility. This approximation makes it possible to learn the networks using fractional updating as described in [4].

7.3 Testing the Cooperative Model

The purpose of testing the cooperative model is to determine if the concept of bots working together as a team in order to solve a common goal is superior to just having bots follow their own preferences. The test setup consists of three CO-Bots on the terrorist team and three PODBots on the counter-terrorist team.

Learning Values During Game Play

The learning scenario ran over 724 rounds where the CO-Bots won the 416 rounds and the PODBots won the remaining 308 rounds.

Learning the networks during game play led to an interesting observation regarding the dependencies in Counter-Strike. When engaging in combat it seems that variables other than weapon and distance have only little or no influence on the outcome of the encounter. This means they can be eliminated in future models without causing a degeneration in the usefulness of the model.

Testing Using Prelearned Values

Clearly the probability node *outcome* in both of the influence diagrams of Figures 7.3 and 7.4 contains too many entries to be learned during the course of any single game. Before conducting additional tests, a team of learning CO-Bots played approximately 1000 rounds against a team of PODBots. Although not enough to completely learn how PODBots play, it was enough to reveal a tendency. The tendency was assumed to continue and thus it was intensified through a huge number of similar cases generated by a small program.

A new test was conducted, using these prelearned values. The aim of the test is to determine whether the cooperative CO-Bots given sensible values in their influence diagrams beats the non-cooperative PODBots and to use the test result for comparison in later tests.

In the test, the CO-Bots have been prelearned and do not learn further during game play as this is not important for the topic in question. Furthermore the fact that probabilities are static eliminates any uncertainties in the test results caused by changes in the distribution of probabilities during the execution of the test.

The test was conducted over 1014 rounds. The PODBot team won 132 rounds and the CO-Bots won the remaining 882 rounds. Whether the superiority is due to the cooperative nature and attention to a common goal incorporated in the CO-Bots or due to their ability to learn accurately or both is not important. What is important is the fact that CO-Bots are far superior to the bots commonly used today.

PODBot Performance

In both tests the PODBot team performs quite badly against the CO-Bot team. Given the circumstances this is quite understandable as the cooperative nature of the CO-Bot is clearly more similar to the way experienced teams of players in Counter-Strike plays than the non-cooperative nature of the PODBots is. Any non-cooperative team facing a cooperative one is bound to have a disadvantage regardless of the fact that the individual abilities of the players might be the same.

Test Environment

Counter-Strike is not as tactically based a game as first believed, as only weapon and distance determines the role a bot should take, regardless of how any team mates behave during the encounter.

Chapter 8

The Personalized Model

In this chapter an extension to CO-Bot is devised. This extension provides CO-Bot with a personality and the ability to trick other bots on the same team into taking actions that are to the benefit of itself. CO-Bots that exhibit a tricking personality are called tricksters. Modeling such a trickster is interesting because in a real life game, having bots with human-like abilities makes the bots far more interesting for players to interact with. The personalized model presented in this chapter is based on the cooperative model described in Chapter 7.

After describing the motivation for this chapter, the concept of communication is introduced and a conceptual comparison of the cooperative and the personalized model is provided. After this the personalized model is presented, and a comparison of the two designs is provided. Later on it is explained in detail how the personalized model is integrated into Counter-Strike. Finally the chapter is concluded by presenting relevant test scenarios and test results.

8.1 Motivation

A human player in Counter-Strike is quite different from a bot based on the cooperative model, in that the bots have only one purpose: To ensure the victory of the group, regardless of the price to the bot itself. Humans, on the other hand, are sometimes selfish and as a consequence they are not always willing to sacrifice their life, even if they know that the overall benefit of the group is great. Actually, some human players would provide false information to their group mates in order to trick them into performing certain actions that optimize the players' situation instead of the group as a whole.

Consider a human player on a team of three. The player spots three enemies, but instead of telling his group mates the actual number of enemies, he says that there is only one. That way, the group mates think that they can easily overcome the adversaries, and hence they choose to storm the enemy. The deceitful human knows that the actual risk is much higher, and therefore stays back and provides suppression fire, which is a less dangerous task in that situation. The result is that the deceitful human survives more combats than his group mates.

A natural extension to the cooperative model is to add personalities to CO-Bot, such that it mimics the human behavior exemplified above. For that to be worthwhile, teams mixed with both CO-Bots and humans should be introduced, since CO-Bots tricking each other is not interesting from the point of view of human players. CO-Bots could communicate with the human players through text messages, whereas the communication in the opposite direction, could be facilitated by some predefined messages, available in a quickly accessible menu system.

8.2 Conceptual Design Issues Associated with Extending the Cooperative Model

In order to extend the cooperative model described in Chapter 7 to incorporate trickery, several design issues must be considered. This section discusses these issues by comparing the cooperative and the personalized model at a conceptual level. Before doing so, however, the concept of communication is introduced.

A Word On Communication

CO-Bot is assumed to have perfect communication abilities.

Definition 5 If all sent messages are received by the correct recipient, preserving the order in which they were sent, the **communication** is **perfect**.

This is not an unreasonable assumption compared to human capabilities. Consider a team of human players engaged in a game of Counter-Strike. Most likely they will have some sort of communication device (e.g. a headset) that provides for instantaneous communication. In general humans do not transmit every single piece of relevant information, since this would be an extremely time consuming task. Thus only the most relevant information is transmitted. However, usually humans are able to infer any missing information. And even if this is not the case, they can easily request elaboration. Any reasonable team of human players will develop a language that allows them to easily distinguish one recipient from another. Any normal messaging device ensures that messages are received in the order they were sent. Therefore communication between human players is next to perfect. It is assumed that CO-Bot must have possibilities no less than humans, and as a consequence CO-Bot communicates perfectly.

Conceptual Comparison

Conceptually all CO-Bots share the same goal in the cooperative model, namely to optimize the group's situation. Furthermore, they communicate perfectly. Therefore, all CO-Bots share the same information at all times and agree on all actions given any information. Thus they need not have a separate instance of the influence diagram and instead they share a single instance.

In contrast, CO-Bots utilizing the personalized model do not share the same goal. A trickster's first priority is to ensure its own survival, whereas a non-trickster's first priority is to ensure the well-being of the group. Thus all CO-Bots may not agree on all actions given the same information and no CO-Bots share instances of influence diagrams; instead they maintain a private instance of the influence diagrams of the personalized model with their own probability distributions learned from cases in which the bot has taken part. Furthermore the use of private instances of influence diagrams ensures that CO-Bots can easily circulate among several groups and bring their own experiences to that group. In this context it is possible for a trickster to lie about the cases it has collected. That is – it may suggest a role division far from the role divisions suggested by non-tricksters. While (naive) non-tricksters believe the different role divisions to be due to difference in collected cases, they are in fact due to difference in goals and ultimately due to differences in personalities.

8.3 Personalized Model Design

This section discusses the design of the personalized model. There is one major change in the personalized model compared to the cooperative one. The cooperative model was built upon two influence diagrams: One for selecting a strategy and one for distributing the roles given by the selected strategy. In the personalized model there is just one influence diagram. Instead of distributing the roles given by a strategy, the influence diagram is used to directly distribute roles among all bots in the group. This approach implies a strategy.

There is one influence diagram for each possible group size. The influence diagram used when a group consists of only one bot can be simplified compared to other group sizes. This is because the extension aims toward bots tricking other bots into taking actions from which the trickster benefits and the single bot in groups consisting of only one bot can not trick anyone. Therefore, two influence diagrams are discussed: The influence diagram for groups consisting of only one bot, and the influence diagram for groups consisting of three bots.

Figure 8.1 depicts the influence diagram for groups consisting of three bots. The most prominent difference compared to the influence diagram of

UBot	total_loss	partial_loss	partial_win	total_win
	-90	-45	45	90
UGroup	total_loss	partial_loss	partial_win	total_win
	-10	-5	5	10

Table 8.1: Utilities of a trickster. The trickster is almost indifferent to group losses

UBot	total_loss	partial_loss	partial_win	total_win
	-10	-5	5	10
UGroup	total_loss	partial_loss	partial_win	total_win
	-90	-45	45	90

Table 8.2: Utilities of a non-trickster. The non-trickster is almost indifferent to its own losses.

the cooperative model is the introduction of a second outcome – now there is a *group_outcome* and a *bot_outcome*. *group_outcome* has states *total_loss*, *partial_loss*, *partial_win* and *total_win*. *total_loss* represents the entire group being killed without killing any enemies, *partial_loss* represents the group killing all enemies but with losses, *partial_win* represents the entire group being killed, but not before killing some of the enemies, and *total_win* represents killing all enemies with no losses. *bot_outcome* has states *total_loss*, *partial_loss*, *partial_win*, and *total_win*. To a particular bot, *total_loss*, represents the bot being killed without having killed an enemy, *partial_loss* represents it being killed, but not before having killed at least one enemy, *partial_win* represents a bot that has not been killed, but that did not kill an enemy either, and *total_win* represents a bot that has killed at least one enemy and that has not been killed itself.

The two outcomes each have a utility attached, *UBot* and *UGroup*, respectively. The outcome nodes introduce the same type of approximation as described in Section 7.2 to remedy not being able to set all utilities in a large utility node. Utilities are influenced by a *personality* with states *trickster* and *non-trickster* in such a way that a trickster is almost indifferent to group losses and non-tricksters are almost indifferent to its own losses. More precisely this is done by setting the utilities of a trickster to those specified in Table 8.1 and the utilities of a non-trickster to those specified in Table 8.2.

As explained in Section 7.3 testing the cooperative model revealed that in this particular test domain (Counter-Strike) factors such as health and armor was of little or no importance. Thus all that is important to a particular outcome are the friendly team's combination of weapons represented by the probability nodes *weapon_1*, *weapon_2* and *weapon_3*, the friendly group's distance to the enemy (*avg_enemy_distance*) and the enemies average weapon strength (*avg_enemy_weapon*). *avg_enemy_weapon* has states *short_range*, *medium_range*, *long_range* and *mixed*. *Avg_enemy_distance* has states *nearby* and *far_away* and the three weapon nodes all have states *short_range*, *medium_range* and *long_range*.

The influence diagram can be used to decide a distribution of roles for a team size of three bots. Therefore there are three decision nodes, *bot_1_role*, *bot_2_role*, and *bot_3_role* each with states *attack* and *suppress*.

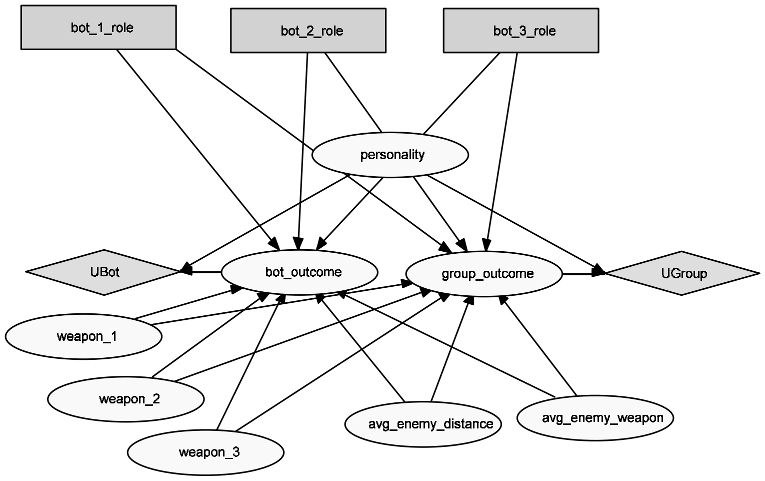


Figure 8.1: Influence diagram for three bots. There is both a group outcome and a bot outcome, each with a utility node attached.

Figure 8.2 depicts the influence diagram of the personalized model for groups consisting of only one bot. The single bot has only itself to consider and thus does not have a *group_outcome*. There is no difference between a trickster and a non-trickster and therefore there is no need for a *personality*.

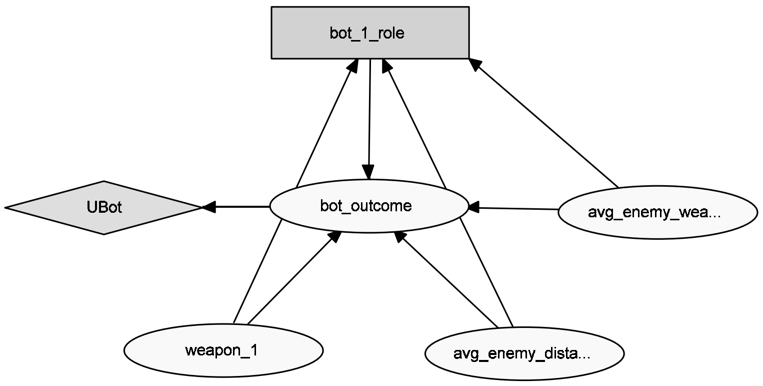


Figure 8.2: Influence diagram for one bot. There is no group outcome node or personality node.

Calculating Decisions

The influence diagrams used in this chapter and in the preceding are all of a nature that permit the calculation of decisions to be done only once, as described in [4], and then use the calculated table to look up utility values later. This increases speed greatly as a simple table look-up is very fast compared to calculating a new table every time. Consider the influence diagram handling only one bot, as depicted in Figure 8.2, the nodes *weapon_1*, *avg_enemy_distance* and *avg_enemy_weapon* have all been instantiated. As a consequence, it is sufficient to look only at one column of the conditional probability table of the *bot_outcome* node, and thus no propagation of evidence is necessary when a calculated table is already present.

The same principle holds for the networks handling two and three bots - the only difference is that there are more instantiated nodes, and two outcome nodes.

8.4 Integration of the Personalized Model Into Counter-Strike

Up until now bots have been given the ability to calculate the best distribution of roles in accordance to their individual goals. However, this ability is just part of the complete decision process. As already explained, compared to the cooperative model, bots no longer necessarily agree on a distribution of roles, and thus they must negotiate and agree on an acceptable one. To support this, a communication system and a negotiation protocol must be developed and used in conjunction with an influence diagram to form a complete decision process.

The Decision Process Using the Personalized Model

When faced with a specific combat situation, a team of bots have a chance to share opinions on how to distribute roles. Using an appropriate influence diagram (depending on the number of team mates), they each decide what they believe to be the best distribution.

A list consisting of expected utilities of each possible combination of distribution of roles is communicated from all bots to all other bots in the group. When a bot has received the lists from its team mates, a combined expected utility of each combination is calculated by summation, and the combination with the highest combined expected utility is chosen. Using the above decision process, all bots arrive at the same conclusion about the distribution of roles.

A trickster may provide false expected utilities for any combination. Providing high expected utilities for combinations deemed good for itself and low expected utilities for undesirable combinations, may trick the team mates into choosing a combination that is more desirable to the trickster.

As an example consider a situation with three bots: B_1, B_2, and B_3. B_1 and B_2 are cooperative bots while B_3 is a trickster. Say there are 4 different distributions of roles to choose from, D_1, D_2, D_3, and D_4. The cooperative bot B_1 sends out a list of expected utilities calculated using its private influence diagram. The values are $\{D_1 = 60, D_2 = 53, D_3 = 20, D_4 = 87\}$. B_2 being a cooperative bot as well has quite similar values namely $\{D_1 = 58, D_2 = 51, D_3 = 23, D_4 = 88\}$. B_3 on the other hand is a trickster. The values sent by B_3 are $\{D_1 = -100, D_2 = -100, D_3 = 100, D_4 = -100\}$. This is because B_3 wants D_3 to be chosen as the distribution of roles. The combined expected utilities for the distributions are reached by summation and is $\{D_1 = 18, D_2 = 4, D_3 = 143, D_4 = 75\}$. Thus the two cooperative bots, B_1 and B_2, were tricked into choosing D_3 as the best distribution of roles even though they individually clearly considered it to be the worst choice for the group.

8.5 Testing the Personalized Model

The test of the personalized model is performed in the same environment as the test of the cooperative model. The terrorist team consists of three bots where one of them is a trickster and the rest are regular, naive bots. As with the cooperative model, the bots have been prelearned such that they know, for example, that storming the enemy with a sniper rifle does not pay off.

The differences between the cooperative model and the personalized model are expected to affect the result of the match. First, the trickster is expected to survive more often than its non-trickster team mates, since it insists on laying suppression fire while it manipulates the team mates to storm the enemy. Second, since some team mates can be tricked into storming the enemy with an unsuitable weapon (e.g. a sniper rifle), they will have difficulties making a kill. Therefore, non-tricksters are expected to kill less opponents than tricksters. Finally, the overall performance of the team is expected to degrade compared to the performance of the team using the cooperative model, as the personalized model does not provide for optimal cooperation from the point of view of the team.

To be able to accurately compare a game containing a trickster to a game without one, a test for both of these situations was conducted. The tests were conducted over approximately 1000 rounds. The result of the test without a trickster can be seen in Table 8.3. As can be seen in the table, the results are close to those obtained in the test of the cooperative model as shown in Section 7.3.

In Table 8.4 the test result of the personalized model with a trickster, is depicted. The result matches the expectations.

	Score	Kills	Deaths
Counter Terrorists(CT)	111		
CT bot1		117	802
CT bot2		109	822
CT bot3		98	831
Terrorists(T)	876		
T bot1		830	101
T bot2		817	108
T bot3		808	115

Table 8.3: Results of the test of the personalized model without a trickster. 987 rounds were played.

	Score	Kills	Deaths
Counter Terrorists(CT)	206		
CT bot1		187	817
CT bot2		171	840
CT bot3		166	843
Terrorists(T)	825		
Trickster		1077	120
T bot2		730	197
T bot3		693	207

Table 8.4: Results of the test of the personalized model with a trickster. 1031 rounds were played.

Chapter 9

Extending CO-Bot with Trust Management

As described in Chapter 8, extending CO-Bots with personalities, making some of them more selfish and thus more human-like, caused the group as a whole to function less optimal. The test results of Section 8.5 showed that non-tricksters were tricked by tricksters into taken actions that they would normally not. In that test, of course, the non-tricksters were completely naive and had no way of detecting tricksters. This chapter concentrates on the development of a counter-measure to tricksters - namely the development of a trust management system. Implementing such a feature will provide the bots with the ability to detect tricksters to some extent and disregard their information when determining which roles should be chosen for which bots.

9.1 Motivation

Incorporating trust management into CO-Bot makes it aware of untrustworthy players and bots. The suspension of disbelief is assumed to decrease if a player consistently succeeds in tricking bots into doing what he wants them to. Based on this assumption, incorporating trust management into CO-Bot is expected to render the bots more interesting opponents and team mates.

9.2 Basics of Trust Management

[16] discusses several issues concerning trust management. In the realm of trust management, there are two key concepts: Trust and Reputation.

Definition 6 A bot's belief in attributes such as reliability and honesty of another, based on previous interactions, is called the **trust** in the other bot.

Definition 7 The expectation about another bot's behavior based on prior observations reported from other bots, is called the **reputation** of the other bot.

These concepts are used to determine whether a bot is trustworthy to interact with. Consider the situation where there are two bots, bot A and bot B. If bot A does not have any direct interaction with bot B to build its assessment of trust on, it may make decisions based on the reputation of bot B. Once bot A have interactions with bot B on which to develop its trust in it, it may do so according to its level of satisfaction with the interactions and use this trust assessment to make decisions for future interactions.

Centralized versus Decentralized Trust Management

A trust management system can be implemented using either a centralized or a decentralized structure.

Centralized systems are mainly used in the area of e-commerce, such as the rating system used on eBay. In such a system, one could imagine a search-agent or a shopping-agent, acting on behalf of a user, as is the case in [16]. In this case the trust and reputation mechanisms are relatively simple. Common characteristics in these systems are listed below.

- A centralized node acts as the system manager responsible for collecting ratings from both sides involved in an interaction.
- Agents' reputations are public and global. The reputation of an agent is visible to all the other agents.
- Agents' reputations are built by the system. There is no explicit trust model between agents.
- Less communication is required between agents. An agent only communicates with the centralized node to know other agents' reputations.

A problem using a centralized structure for trust management could be that agents are reluctant to give negative ratings because the receiving party can see the rating and thus might give another negative rating as payback.

Another problem is that an agent in an e-commerce context may discard its old identity at any time and make another identity with an empty rating history.

A third problem is that an agent may create fake identities (and thereby fake agents) and make these new agents give itself high ratings to artificially increase its reputation.

In the context of FPS gaming these issues might not be as apparent as they are in the context of e-commerce.

The mechanisms used for dealing with trust and reputation in decentralized systems, like cooperative FPSs, are more complex than the ones used in centralized systems.

- There is no centralized manager to govern trust and reputation.
- Subjective trust is explicitly developed by each bot. Each bot is responsible for developing its own trust in other bots based on their direct interaction.
- No global or public reputation exists. If bot A wants to know bot B's reputation, it has to proactively ask other bots for their evaluations of B's reputation. The reputation of bot B developed by A is personalized because bot A can choose which bots it will ask for evaluations of B, its trustworthy friends or all known bots. Bot A decides how to combine the collected evaluations to get bot B's reputation. For example, it can combine the evaluations coming from trusted bots, exclusively. Or it can put different weights on the evaluations from trusted bots, unknown bots, and even untrustworthy bots when it combines them.
- A lot of communications is required between bots to exchange their evaluations.

In decentralized systems it is hard for a bot to increase its reputation artificially because a bot (A) gets the reputation of any other bot (B) by basing the calculation on its own knowledge on the truthfulness of the other bots that make recommendations for bot B. Using this structure bots can express their true feelings about other bots without having to fear for revenge, because only the bot who requests the information is able to see it. The tradeoff of using a decentralized approach as opposed to a centralized one is a substantial increase in the amount of communication and computation needed for each bot.

In this project a decentralized approach was chosen as the foundation for the trust management design and implementation for two reasons. First, it is the most interesting from a theoretical point of view, as bots mimic the human approach to cooperation by letting each bot contain its own specific information about any other bots. Second, if human players are to join a team of trust managing bots, the bots must be able to cope with the fact that human players always maintain their own trust and reputation values anyway.

Design and Implementation Issues

When implementing a trust management system some issues have to be addressed. Although originally written in a peer-to-peer file sharing network context, these issues have been modified from [16] to better fit an FPS context.

- How is an interaction to be evaluated? Trust is built on a bot's direct interactions with other bots. For each interaction, a bot's degree of satisfaction of the interaction will directly influence its trust in the other bot involved in the interaction. Sometimes, an interaction has multiple aspects and can be judged from different points of view.
- How does a bot update its trust in another bot?
- When will a bot ask for recommendations about another bot that it is going to interact with?
- How does a bot combine the recommendations for a given bot coming from different references? Since the recommendations might come from trusted bots, non-trusted bots or strangers, a bot has to decide how to deal with them.
- How does a bot decide if another bot is trustworthy enough to interact with, according to its direct experiences or reputation, or both?
- How does a bot develop and update its trust in a reference that makes recommendations?

In addition to the above mentioned list, modified from [16], some other design issues are relevant in this context:

- Humans evaluate a statement set forth by a certain other person by consulting their own previous experiences with that person (their trust in that person), by consulting other people, be it trusted, untrusted or complete strangers (reputation) and finally they evaluate the statement itself. Therefore completely untrusted people can still earn some trust if the statement they put forth is sensible to the person evaluating them.
- In real life, trust is an asymmetric concept, in that it is harder to earn trust than it is loosing it. To see this, consider the situation where a salesman has sold a buyer a completely useless product. Unless the salesman has former satisfying sales, either directly to the buyer or to friends of the buyer, the buyer is not likely to buy anything from the salesman again. Even if the salesman gets another chance, the buyer will pay even more attention to the degree of satisfaction of the sale.
- The degree of satisfaction with an interaction does not depend on the quality of the interaction alone. In a decentralized system, bots have different perceptions of the world and thereby different perceptions

of the truthfulness of a statement. Returning to the example with the salesman, the salesman can actually believe a sold product to be the best product in the world, while at the same time the buyer is unsatisfied with the product.

The extension to the design issues, set forth by [16], consists of evaluating the actual contents of the statements and not just the trust and reputation values of the reputation provider. Furthermore, to mimic human behavior, trust is considered asymmetric.

9.3 The CO-Bot Trust Management System

As already stated, when faced with a specific combat situation, bots receive information from all other bots. They must each decide who they can trust and who they can not. In the following, we present a trust management system as a solution to this problem that takes into account some of the design issues described in Section 9.2. Other works, such as [16], has suggested solutions to creating such a trust management system. However they are specifically designed for different domains than that described in Chapter 4.

The basis of any implementation is the exchange and processing of information between bots. Precisely what information is exchanged and how it is processed will be clarified momentarily. A method is presented that can detect tricksters by evaluating interactions. The method mimics human behavior in that it is based on trust, reputation and an evaluation of the information itself. In most domains that require trust management, a particular agent needs only make a particular decision for itself. This is the case in [16] where agents decide for themselves (on behalf of a user) which other agents are trustworthy. This domain is different in that it requires bots to agree on a single group decision. To ensure this, a voting algorithm is suggested. Finally test results are presented as produced using the test environment described in Section 6.3.

Decision Process Overview

When faced with a combat situation, bots exchange opinions. Each bot sends out a list to all other bots, consisting of expected utilities of each possible combination of distribution of roles (see Figure 9.1). This list needs not be the same for all receiving bots.

Upon receiving such lists, each receiving bot decides whether it trusts each sending bot by evaluating them according to trust and reputation. Furthermore, the receiving bot may decide that an (otherwise trusted) bot is untrustworthy based on the contents of the list that was received, and disregard the list. In both cases, the trust in that particular bot is updated accordingly.

The receiving bot then combines all trusted lists (including its own) into a single list which is normalized in order for the values not to be dependent on the number of trusted bots. This revised list represents a bot's revised opinion.

Again each bot sends out a list - this time, the newly constructed revised list. The list no longer contains expected utilities, but scores of each possible combination of distribution of roles.

Normally, in a decentralized system, each bot needs only to decide for itself the best distribution of roles, based on trust and reputation. However, in this situation, upon receiving the revised lists, the bots must agree on a (final) single distribution of roles. This issue is new to the field of trust management theory and requires an innovative solution.

We suggest a voting system to solve this problem. The voting system represents the group excluding single bots who are collectively deemed untrustworthy.

All bots must cast votes on all other bots, based on their trust in each bot and based on an evaluation of the contents of the original list sent out by each bot. They vote either in or out. If a particular bot has fewer in-votes than a certain threshold, they are excluded from influence on the final cooperative decision.

It is assumed that all bots submit to the rules of the voting system. When the votes have been cast and the revised lists have been sent from all bots to all other bots, everybody - even untrusted bots - combines the revised lists correctly (everybody excludes untrusted bots from influence) and takes on the role dictated by the resulting distribution of roles. The resulting distribution of roles is the combination of roles with the highest score.

Using the voting system in this way is completely decentralized as bots get to decide for themselves who will influence their private decision, before the group deems any bots untrustworthy. In this way a bot, even though it has been deemed untrustworthy by the group, may already have influenced the final distribution of roles through individual bots that do trust it. This is not a problem for the group because a single bot that is influenced too many times or too much by tricksters will itself become untrustworthy over time, or it too will detect that the tricking bot is in fact a trickster.

The voting system can be used earlier in the process to create a more restricted and idiot-proof solution.

In this solution, each bot votes out untrusted bots, before it combines the remaining (trusted) bots' lists into a single revised list. No bots that have been deemed untrustworthy by the group can ever influence the final cooperative distribution of roles through any single (naive) bot. However, although the group has deemed a particular bot trustworthy, other, less tolerant, bots do not have to trust it. Thus by using the voting system prior to combining lists into revised lists the entire process has become more restricted and fewer tricksters are allowed influence.

The first solution was chosen for test and implementation purposes because it is more decentralized in that bots with a naive personality are no longer protected by a group censorship. This is in consistency with how human players act, in that some humans have naive personalities.

Calculating Trust

Section 9.2 discusses several design issues. In particular humans evaluate each other by means of trust, reputation and evaluation of the information that has been received. To mimic the human behavior, a similar approach to creating a trust management system is used in the following.

Each bot maintains a level of trust, T, of all other bots. Bot_k's trust in Bot_i, $T_{k,i}$, is based on the total number of interactions with Bot_i, $I_{k,i}$, and the number of times Bot_i was deemed trustworthy, $TW_{k,i}$. The relationship is the following:

$$T_{k,i} = \frac{TW_{k,i}}{I_{k,i}} \tag{9.1}$$

Note that $T_{k,i}$ is the trust of Bot_k in Bot_i. It is not based on reputation, which is all other bots' trust in Bot_i. Note also that it is a number between 0 and 1.

$CT_{k,i}$ is k's collected trust in Bot_i calculated from use of trust and reputation. It is defined as,

$$CT_{k,i} = \sum_{j=1}^{N} (T_{j,i} \cdot T'_{k,j}) \tag{9.2}$$

where N is the number of bots. In order for Bot_k to calculate its collected trust in Bot_i, $CT_{k,i}$, it consults all bots in the group (itself included). Each bot, Bot_j, sends back its trust value in Bot_i, $T_{j,i}$. To Bot_k this is known as reputation. At some point Bot_k collects the trust value $T_{k,i}$. To Bot_k this is known as trust (its own trust in Bot_i). The information received from each Bot_j (and $T_{k,i}$) is weighted by a function $T'_{k,j}$ calculated using the trust value Bot_k has in Bot_j, as follows

$$T'_{k,j} = \begin{cases} 0 & \textbf{if } T_{k,j} < h \\ T_{k,j} & \textbf{else} \end{cases} \tag{9.3}$$

where h is a threshold. Formula 9.3 discards untrusted bots. The reason for doing so is that without it, even though a bot is considered untrustworthy, it can still have tremendous impact simply by sending sufficiently huge positive or negative numbers in L_i. Note that the trust value $T_{k,k}$ is always 1 and thus the function $T'_{k,k}$ always becomes 1.

Each bot sends out a list, L_i, consisting of expected utilities of each possible combination of distribution of roles. A new list, L_{new}, which is L_i

modified according to trust, reputation and an evaluation of L_i, is calculated as follows

$$L_{new} = \frac{\sum_{i=1}^{N}(L_i \cdot CT_{k,i} \cdot Eval(L_i))}{\sum_{i=1}^{N}(TBC_{k,i} \cdot Eval(L_i))}^{1} \tag{9.4}$$

where N is the number of bots and $Eval(L_i)$ is an evaluation of L_i.

In consistency with the design issues discussed in Section 9.2, $Eval(L_i)$ is an evaluation of the contents of L_i. First, it detects whether too many elements are to be considered untrustworthy, by evaluating the number of untrustworthy elements against a threshold-function H_1. Second, it detects whether a single element is obscure enough, that it would single handedly obscure the result, by evaluating each element against a threshold oh.

$Eval(L_i)$ is defined as

$$Eval(L_i) = \begin{cases} 0 & \textbf{if} \begin{array}{l} (\sum_{m=1}^{R}(Element(L_i[m])) > H_1) \vee \\ (\exists q : |L_i[q] - L_k[q]| > oh \wedge 1 \leq q \leq R) \end{array} \\ 1 & \textbf{else} \end{cases} \tag{9.5}$$

where $L_i[m]$ is the m'th element in L_i and R is the number of elements in L_i. $Eval(L_i)$ uses $Element(L_i[m])$ to evaluate whether a single element is untrustworthy. This is conducted by taking the difference between the element in L_i and the corresponding element in L_k (Bot_k's list) and evaluating this difference against a threshold-function, H_2, as in the following

$$Element(L_i[m]) = \begin{cases} 1 & \textbf{if } |L_i[m] - L_k[m]| > H_2 \\ 0 & \textbf{else} \end{cases} \tag{9.6}$$

$\sum_{m=1}^{R}(Element(L_i[m])) > H_1$ in 9.5 ensures that no more than a certain number of elements in L_i, depending on the function H_1, can be obscure.

$\exists q : |L_i[q] - L_k[q]| > oh \wedge 1 \leq q \leq R$ in 9.5 ensures that no single element in L_i can ever be too obscure by checking the difference between each element in L_i and the corresponding element in L_k against oh.

H_1 is a function that determines the threshold for discarding elements according to Bot_k's trust in Bot_i, defined as

$$H_1 = T_{k,i} \cdot h_1 \tag{9.7}$$

where h_1 is some threshold for Bot_k, as defined by Bot_k's personality. Weighting h_1 by $T_{k,i}$ adjusts Bot_k's error tolerance towards Bot_i according to Bot_k's trust in Bot_i. This is similar to human behavior in that a person

[1] A multiplication of a list and a number is conducted by multiplying all elements in the list by the number. A division of a list with a number is conducted by dividing all elements in the list by the number. Two lists are summed by summing each element in the first list with the corresponding element in the second list.

is more willing to accept information, on which he is uncertain, if it comes from a trusted associate.

H_2 in 9.8 is similar to H_1:

$$H_2 = T_{k,i} \cdot h_2$$

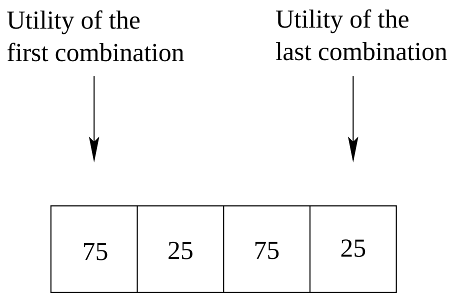

Utility of the first combination
Utility of the last combination
75
25
75
25

Figure 9.1: An example list of expected utilities.

In Formula 9.4 the list calculated in the numerator is divided by the number of trusted bots that tell the truth,

$$\sum_{i=1}^{N}(TBC_{k,i} \cdot Eval(L_i))$$

where

$$TBC_{k,i} = \begin{cases} 0 & \textbf{if } CT_{k,i} = 0 \\ 1 & \textbf{else} \end{cases} \tag{9.8}$$

This is done in order to make sure L_{new} in Formula 9.4 is normalized. The denominator counts the number of trusted bots in this calculation using $Eval$ and TBC. TBC is the Trusted Bot Count, which counts trusted bots based on collected trust values. If this normalization is not done a bot which trust many team mates will have larger values in L_{new} than a bot which believes there are fewer trustworthy team mates thus giving the former a larger influence.

Updating Trust

$Eval(L_i)$ forms the basis of evaluating interactions with other bots and is thereby applied when updating $T_{k,i}$ in 9.1. A bot is deemed trustworthy if $Eval(L_i)$ becomes 1 and then $TW_{k,i}$ and $I_{k,i}$ are both incremented by 1. Otherwise only $I_{k,i}$ is incremented.

Recall from Section 9.2 that trust is asymmetric. If it was not, $I_{k,i}$ would be incremented by 1 when $Eval(L_i)$ becomes 0 - that is, when Bot_i

is deemed untrustworthy. To facilitate asymmetric trust, when $Eval(L_i)$ becomes 0, $I_{k,i}$ is instead incremented by an arbitrary number depending on Bot_k's personality. The higher the number, the harder it is for Bot_i to regain lost trust.

The Voting System

Bot_k's vote on Bot_i, $V_{Bot_{k,i}}$, is

$$V_{Bot_{k,i}} = \begin{cases} 0 & \textbf{if } T_{k,i} \leq h \vee Eval(L_i) = 0 \\ 1 & \textbf{else} \end{cases} \tag{9.9}$$

$V_{Bot_{k,i}}$ taking on the value 1 represents Bot_k voting in Bot_i and $V_{Bot_{k,i}}$ taking on the value 0 represents Bot_k voting out Bot_i.

Bot_i is excluded from influence, if

$$\sum_{k=1}^{N} V_{Bot_{k,i}} \leq gvh \tag{9.10}$$

where gvh is a group voting threshold

9.4 Testing CO-Bot With Trust Management

In order to verify that the trust management system works as intended, a test is performed. The following test setup is used: The counter-terrorist team consists of three PODBots and the terrorist team consists of three CO-bots, all of the latter equipped with trust management, where one of them is a trickster. All the CO-Bots have been prelearned just as in earlier iterations. Since the setup is very similar to the one used in earlier tests, the test results are assumed to be comparable.

The trust management system is expected to have a profound influence on the test results, since the tricksters, as described thus far, does not perform sophisticated tricking. Instead they exaggerate the lies to such an extent that it is easy for the trust managing bots to detect the tricksters and vote them out accordingly. As a consequence, the tricksters will have no influence, and thus only the expected utility of true group-minded bots are considered. Therefore, the test results are expected to be similar to the results of testing the personalized model with tricksters, to be found in Table 8.3.

	Score	Kills	Deaths
Counter Terrorists(CT)	91		
CT bot1		101	940
CT bot2		88	958
CT bot3		84	970
Terrorists(T)	956		
T bot2		968	96
Trickster		957	95
T bot3		943	82

Table 9.1: Results of the test of the CO-Bot with trust management. 1047 rounds were played.

As can be seen from Table 9.1, the PODBot looses a bit more than in the initial test described in Section 8.5. This difference is expected to be caused by statistical inaccuracy.

In Figure 9.2 a graph shows two bots' trust value over a number of encounters. As can be seen, the trickster's trust in the non-trickster quickly climbs to a high value, whereas the opposite is the case for the non-trickster's trust in the trickster. When the non-trickster occasionally is deemed more trustworthy, as in encounter 9 and 10, it is because the two bots agree on the best distribution of roles.

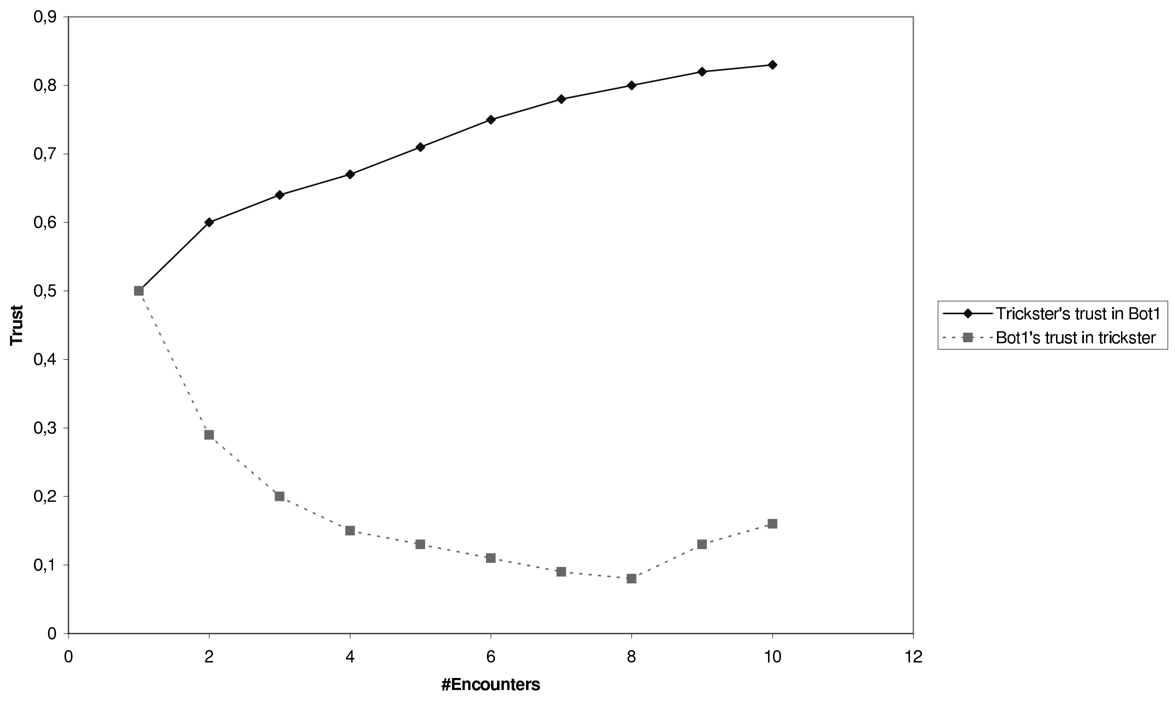


Figure 9.2: Two bot's trust in each other over a number of encounters. The graph shows that it takes only a small number of encounters for a trickster to be deemed untrustworthy.

All in all the test results match the expectations, and thus it is concluded that the trust management module works as intended.

Chapter 10

Trickery Management

As seen in Chapter 9 the addition of trust management enables CO-Bot to detect tricksters. This chapter focuses on improvements to the CO-Bot in order to make it perform better against bots containing trust management.

10.1 Motivation

In the preceding chapter, CO-Bot has been equipped with a mechanism for detecting deceitful group mates. This renders a trickster bot almost useless as it has no clever strategy of tricking: It consistently exaggerates the lies to an extent that makes it trivial for the trust managing bots to detect. Apart from being useless for tricking, it is also very far from human behavior. The next natural step in the development of CO-Bot is to add trickery management, such that it will try to trick others in a more sophisticated manner.

As an example of trickery management, consider two human players – let us call them Skinken and Potter. Skinken is a trickster without any prior knowledge of Potter. Skinken wants to find out how naive Potter is, so when he lies to Potter, he makes sure to exaggerate the lie more and more. When Potter finds out that Skinken lies, he will ignore the false information – something that Skinken detects. Thereby Skinken has found an estimate of how naive Potter is. Surely, Potter trusts Skinken less than before, but if Skinken consistently tells truths for a while, Potter's trust in Skinken will eventually be restored. Skinken may now use the estimate of how naive Potter is to determine how much to lie. In this situation, Skinken's exploitation of Potter is at its maximum, for the drawback of the group as a whole, as well as for Potter, but for the benefit of Skinken.

10.2 Tricking Trust Managing Group Mates

Being detected as deceitful makes it harder for a trickster to influence the choices about distribution of roles made by team mates. Providing selfish information when it is evident that it will be detected as being deceitful and thus discarded is of no use to the trickster and will only add to the belief of the bot in fact being a trickster, making the situation for the trickster even worse.

The challenge for the sophisticated trickster is to determine the threshold values introduced in Formula 9.5 (h_1 and oh). If the trickster succeeds in determining these values, it can make qualified guesses on when a lie of a certain size will pass unnoticed and when it will not. Based on this, it can make qualified guesses on whether it will succeed in influencing team mates into choosing a certain distribution of roles - one that it will benefit from. Consider Formula 9.4 from Section 9.3:

$$L_{new} = \frac{\sum_{i=1}^{N}(L_i \cdot CT_{k,i} \cdot Eval(L_i))}{\sum_{i=1}^{N}(TBC_{k,i} \cdot Eval(L_i))}$$

This is where any received lists are combined into a single list, and thus where some lists are rejected. As is evident from the formula, there are just two reasons for a bot, Bot_s, being rejected by another bot, Bot_r: Either Bot_r does not trust Bot_s and thus $CT_{r,s}$ becomes 0 or Bot_r believes Bot_s's particular list to be an attempt to lie and then $Eval(L_s)$ becomes 0 (or both). If Bot_s has no prior knowledge of Bot_r, determining which of the two is the reason for its potential rejection is almost impossible. However, Bot_s is not without prior knowledge. In fact it is Bot_s that decides when to tell a lie and when not to. And thus Bot_s can control Bot_r's collected trust in itself to some extent, simply by telling the truth to all bots in the group. The only uncertainty is in the difference in world perception of each bot, a world perception based on individually gathered cases. Under the assumption that all bots will gather somewhat similar cases over time, this is not an issue.

The procedure is for Bot_s to eliminate the possibility of Bot_r not trusting it, and thus leave only the possibility of being rejected due to an evaluation of the list Bot_s sends to Bot_r. That is, Bot_s leaves only the possibility of being rejected due to $Eval(L_s)$ becoming 0. It may do so by telling the truth for a while, to raise its trust. When Bot_s registers that Bot_r votes it in (i.e. when $VBot_{r,s}$ becomes 1), Bot_s is certain that neither the trust of Bot_r in Bot_s nor the list sent by Bot_s to Bot_r caused a rejection. In other words, Bot_s knows that Bot_r trusts it (and furthermore that telling the truth did not exceed Bot_r's threshold). Since Bot_s is trusted, the next potential rejection must be caused by Bot_r's threshold being exceeded.

Now consider the function $Eval(L_s)$ (Formula 9.5 from Section 9.3):

$$Eval(L_s) = \begin{cases} 0 & \textbf{if} \begin{array}{l} (\sum_{m=1}^{R}(Element(L_s[m])) > H_1) \vee \\ (\exists q : |L_s[q] - L_r[q]| > oh \wedge 1 \leq q \leq R) \end{array} \\ 1 & \textbf{else} \end{cases}$$

There are two reasons for $Eval(L_s)$ becoming 0. Either the threshold oh is exceeded or the value returned by the function H_1 is exceeded. H_1 (for Bot_r) depends on $T_{r,s}$ and h_1, however Bot_s assumes (as before) $T_{r,s}$ to be under its control, and thus it can only be that the threshold h_1 is exceeded. The following presents a method for Bot_s to determine the threshold values of Bot_r.

Determining Threshold Values

Bot_s will guess the threshold values oh and h_1 of Bot_r one at a time, starting with the threshold, oh, for detecting obscure single elements in the received list. Later it will be clear why this threshold has to be determined first.

Determining oh

The strategy for determining oh is to send a list, solely consisting of sane expected utilities except for one, which is somewhat higher. If Bot_r votes Bot_s trustworthy (i.e. when $VBot_{r,s}$ becomes 1 in the voting system, as described in Section 9.3), then Bot_s knows that Bot_r's oh has not yet been exceeded. As a consequence, Bot_s chooses a higher expected utility the next time. This is repeated until $VBot_{r,s}$ becomes 0, thus revealing to Bot_s the approximate value of oh.

Several ways of deciding on which false expected utility to send, can be applied. Inspired by the mechanism of avoiding congestion control on computer networks, Bot_s applies a slow-start approach as described in [13]. The problem is similar in that a maximum value for sending packets while avoiding congestion, has to be found. The principle is that as long as an exceeding false expected utility has not been found, the expected utility is doubled, thus making it grow exponentially. When an exceeding false expected utility has been found, that value is halved and from that point it continues to grow linearly instead of exponentially. When the false expected utility exceeds again, the search for oh ends.

Determining h_1

When the value of oh has been determined, it is possible to determine h_1. This threshold determines how many minor lies will pass unnoticed. As can be seen in Formula 9.7, H_1 is a function of trust and the actual threshold, h_1. The task is to guess the latter. The approach is to start lying about one expected utility, and if this passes unnoticed, twice as many expected utilities are lied about, thus utilizing the slow start approach once again. This continues until Bot_r votes Bot_s out, at which time the search continues in a linear fasion. The completion of the linear search marks the end of the search for h_1.

In the process of finding h_1, it is important for Bot_s to choose false expected utilities that it is certain that Bot_r sees only as minor deviations and not huge lies (lies which exceed oh). Therefore the false expected utility is set just below the value of oh. This is the reason why oh has to be determined first.

Using the Threshold Values

Having found the threshold values h_1 and oh, Bot_s is fit to perform sophisticated trickery. From this point, Bot_s tries to trick all other bots every time a decision upon distribution of roles has to be made. Since it does not expect to be caught lying, it can only benefit from it.

The first step in tricking is for Bot_s to settle for a distribution that it wants to be chosen. This would be the distribution that has the highest expected utility. Bot_s's goal is to increase the probability that this distribution is picked and decrease the probability of all others. It tries to fulfill this goal by calculating the number of utilities, n, Bot_r would accept a minor deviation on, using Formula 9.7. This is possible since the constituents of Formula 9.7, h_1 and $T_{r,s}$, are known.

If $n = 1$ (i.e. only one element can be changed), the expected utility of the desired distribution is incremented by Bot_r's oh. If $n > 1$ then, apart from increasing one expected utility, Bot_s has the opportunity to dampen $n - 1$ utilities. The question is which distributions to dampen. In order to find out, Bot_s calculates the expected utilities of all decisions about distributions from a non-trickster's point of view (this is done by altering the state of the *personality* node as depicted in Figure 8.1). The $n - 1$ highest expected utilities, except for those representing distributions of roles also desired by Bot_s, are decremented by Bot_r's oh. Thereby Bot_s has maximized its trickery of Bot_r. The same is carried out for all other bots.

As a clarifying example consider a situation where Bot_s is to trick Bot_r. All threshold values of Bot_r have been determined by Bot_s. $h_1 = 4$, $oh = 50$ and $T_{r,s} = 0.5$. Bot_s settles for a desired distribution by calculating all expected utilities. Say the distribution *Suppress, storm* has the highest expected utility and thus is the desired. Next, the number of utilities, n, Bot_r will accept a minor deviation on, is calculated: $n = T_{r,s} \cdot h_1$. This amounts to two, which means that Bot_s can increment the utility of the desired distribution by oh and decrement one of the utilities of the undesired distributions also by oh. As described above, Bot_s calculates all expected utilities from a non-trickster's point of view. These can be viewed in Table 10.1. As can be seen in the table, the distribution *Storm, storm* has the highest expected utility (80), and since it is not Bot_s's desired distribution its utility is decremented by *oh* giving the value $80 - 50 = 30$. The utility of the desired distribution (15) is incremented by *oh* giving the value $15 + 50 = 65$. All calculations have now ended and Bot_s's list of utilities sent to Bot_r is depicted in Table 10.2.

Distribution	**Utility**
Storm, storm	80
Storm, suppress	23
Suppress, storm	15
Suppress, suppress	1

Table 10.1: Expected utilities from a non-trickster's point of view, calculated by Bot_s

Distribution	**Utility**
Storm, storm	30
Storm, suppress	23
Suppress, storm	65
Suppress, suppress	1

Table 10.2: The final list of utilities sent from Bot_s to Bot_r. The utility of the distribution *Suppress, storm* has been artificially increased, whereas the utility of *Storm, storm* has been artificially decreased.

10.3 Testing CO-Bot With Trickery Management

In order to verify that the trickery management system works as intended, tests of the implementation are conducted. The test setup is as in the earlier tests (see e.g. Section 9.4), except the CO-Bots are now equipped with trickery management. Two tests have been conducted, each with different threshold values for the trust managing bots. Besides the usual test results (i.e. the score for each team and each team member), the number of times the team chooses the trickster's desired distribution of roles is recorded.

In the first test, *oh* was set to 20 and h_1 was set to one. This corresponds to bots that are very hard to trick. The team chose the trickster's desired distribution 11.7 % of the total number of decisions about distributions. Note that this does not mean that the trickster bot succeeded in tricking the group mates in 11.7 % of the decisions – sometimes the group mates actually agree with the trickster.

In the second test, *oh* was set to 80 and h_1 was set to four. This renders the trust managing bots easier to trick. The test showed that the team chose the trickster's desired distribution 89.6 % of the total number of decisions.

Figure 10.1 shows how successful the trickster is in a part of the test run, with one plot for each test with different threshold values for trust managing bots. In the 44th encounter (in the plot for the high threshold) the trickster has determined all threshold values and therefore begins to perform trickery, with frequent success. An occasional glitch is caused by difference in experience between the bots.

Table 10.3 and Table 10.4 shows the end scores for each of the two tests. The results of the test where non-tricksters have low thresholds are quite similar to the test results of the trust managing bots with one naive trickster (see Section 9.4). The reason is that even though the tricking bot is now more sophisticated, the victims are very hard to influence using trickery, as a consequence of their small threshold values. The results of the test where non-tricksters have high thresholds are close to the test results of the personalized model as shown in Table 8.4. This is because the trust managing bots are quite naive, due to their relatively high threshold values.

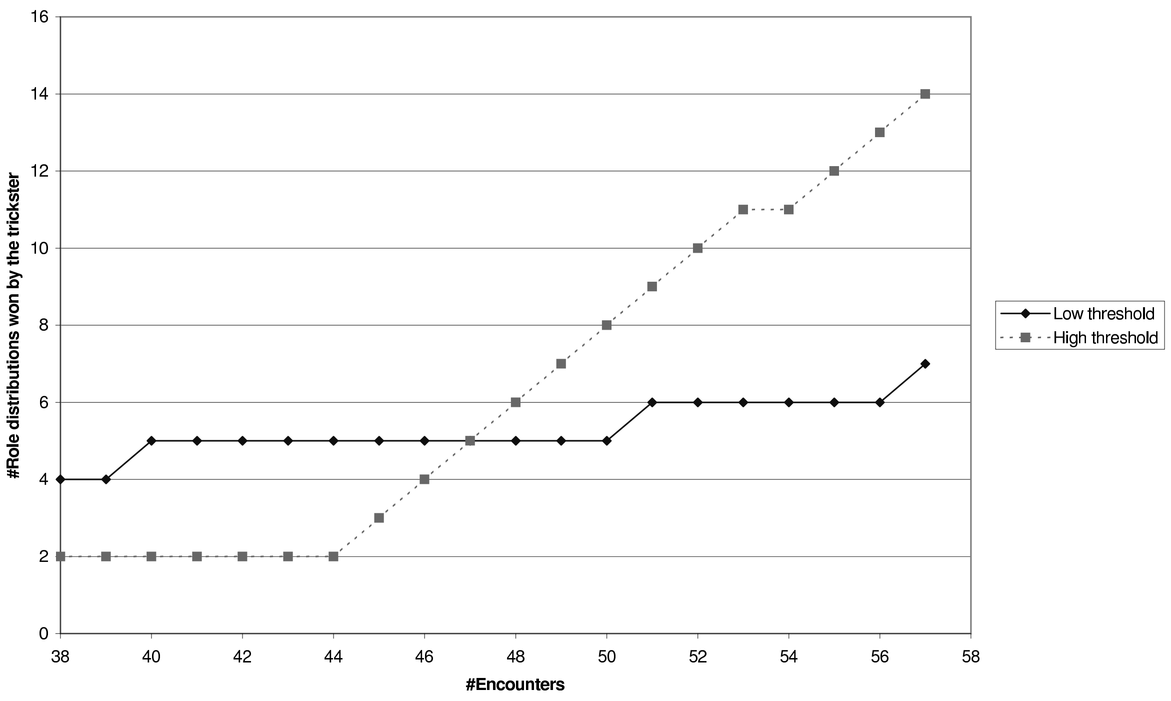


Figure 10.1: Graph showing how many times the trickster got the desired distribution. The two plots mark tests with different threshold values for trust managing bots. In the 44th encounter (in the plot for high threshold), the trickster bot has finished determining the threshold for fellow bots, and in the subsequent decisions it performs sophisticated trickery with success.

	Score	Kills	Deaths
Counter Terrorists(CT)	105		
CT bot1		111	825
CT bot2		106	834
CT bot3		98	837
Terrorists(T)	916		
T bot2		847	99
Trickster		828	105
T bot3		821	111

Table 10.3: Results of the first test of the CO-Bot with trickery management. The trust managing bots (T bot 2 and T bot 3) are hard to cheat because the threshold values are low. 1021 rounds were played.

	Score	Kills	Deaths
Counter Terrorists(CT)	185		
CT bot1		191	801
CT bot2		183	816
CT bot3		181	822
Terrorists(T)	853		
Trickster		895	138
T bot2		788	213
T bot3		756	204

Table 10.4: Results of the second test of the CO-Bot with trickery management. The trust managing bots (T bot 2 and T bot 3) are easier to cheat because of the higher threshold values. 1038 rounds were played.

Chapter 11

Conclusion and Future Work

This chapter concludes on the project described in this report by reviewing what was accomplished in terms of contributions to the theory of trust management and by putting these into perspective. Closingly some ideas for relevant future work is presented.

11.1 Conclusion

In the later years, trust management has become a hot topic within the decision support systems community. It is usually discussed in an Internet context, where shopping or download agents use trust management to govern particular interests of a person by automatically deciding which other agents to trust in the context of these interests. Although usually discussed in an Internet context, the project described in this report seeks to apply trust management to cooperative multiplayer FPSs.

In doing so, problems arise that are new to the theory of trust management. Due to the group oriented nature of cooperative FPSs, they demand a single cooperative decision to be reached. When discussed in other contexts, trust management governs the interests of a single person. For example, trust management is applied to protect a user of an email service by filtering received emails, to sort peers in peer-to-peer systems, to choose the fastest source of download or to find the most trusted online store for a particular product. FPSs are different in that trust management does not exist to protect anyone. It exists to increase the suspension of disbelief, by adding human-like character traits to bots. In fact, to stress this, a concept new to the theory of trust management is introduced - namely trickery management. Trickery management is the diametrically opposite of trust management. While trust management seeks to minimize the trickery of tricksters, trickery management seeks to optimize it.

Normally trust management is applied to a number of services, where it is natural to base a filtering of the service providers on trust and reputation. However, the introduction of trickery management implies that false information can be communicated from one to another. To mimic human behavior, trust management is extended to include an assessment of the soundness of what is actually communicated.

Many times before have human players been astonished by the creativity and resourcefulness of game developers, usually due to impressive graphics, sounds, ideas or overly dynamic virtual worlds that allow for a virtually unlimited number of paths through a game. Although in many cases deceitful and intolerant themselves, human players of FPSs have grown accustomed to bots that do not imitate human behavior by being deceitful or intolerant. While hard to measure, bots with deceitful and intolerant personalities are assumed to increase the suspension of disbelief of human players. Based on this assumption the report at hand documents the development of a cooperative bot for the popular FPS Counter-Strike. The bot called CO-Bot demonstrates and renders probable the applicability of trust and trickery management as a means to incorporate deceitful and intolerant personalities into FPSs.

Before implementing personalities or trust and trickery management, basic cooperative abilities and game strategic knowledge were incorporated into CO-Bot through a cooperative decision model. In a test setup, where a team of CO-Bots played against a team of PODBots, tests showed that CO-Bot by far outperformed the PODBot which does not utilize cooperation to the same extent.

Built on the cooperative model, the personalized model added personalities to CO-Bot. Thus tricksters were introduced, in an effort to make the bots more human-like, as team mates with a hidden agenda to be awarded specific roles no matter if this was the best decision for the team. As expected, tests of the implementation of tricksters showed that CO-Bot was naive and vulnerable to trickery at this stage. Furthermore, the tests showed that tricksters benefit in terms of in-game stats from their trickery.

The cooperative and the personalized decision model alike is based on influence diagrams. Being that CO-Bot is relatively simplistic and made for demonstration purposes only, creating bots for commercial use, possibly incorporating a whole range of personalities, is expected to entail a considerable increase in complexity. Given the requirement of FPSs for virtually instant decision making, the inherent complexity of solving influence diagrams and considering the simplicity of the influence diagrams constructed for supporting CO-Bot's decision process, it is recommended that other methods than influence diagrams are researched.

In response to trickster bot's deceitful nature, CO-Bot was subsequently equipped with a trust management system that rendered it capable of detecting tricksters. Inspired by human nature, the trust management system was designed to take trust, reputation and an assessment of the soundness of the statement to be evaluated into account. With this extension CO-Bot was able to detect tricksters and, by deploying a voting system, prevent these from affecting the single cooperative decision required by cooperative FPSs. As expected, tests revealed that tricksters were easily detected and excluded from influence. This of course, is due to the consistently exaggerated lies of tricksters. In terms of human behavior, such lies are considered unrealistic.

To redeem the situation, CO-Bot was equipped with trickery management. Trickery management is applied to determine threshold values of trust managing CO-Bots, revealing to what extent it is feasible to lie about expected utilities of outcomes. Threshold values are found by varying the number and severity of the lies being sent and then evaluating the result. The newly found threshold values are used to construct future lies making sure the receiving CO-Bot does not detect the trickster's trickery. As expected, tests showed a substantial increase in tricksters' influence on the cooperative decision when trickery management was used against team mates using trust management.

Contrary to influence diagrams, trust and trickery management, as it is presented here, is not inherently complex. It can easily accommodate the strict requirements of commercial FPSs, and if applied to trick human players teaming with CO-Bot, sophisticated trickery through trickery management is expected to increase the suspension of disbelief considerably.

11.2 Future Work

Central statistics server

The usefulness of influence diagrams, in an FPS context, is inhibited by lack of statistics on which to train the networks. A way of solving this problem would be to implement a central statistics server which collects information (cases) from every computer which runs the bot code and distributes the new probability among all the participating computers. This implementation would entail general security issues but could also contain influence diagrams (or some other element of Decision Support Theory) itself. These elements could be useful for classifying players giving the statistics for each player different weights and maybe using specific statistics only in specific contexts. The server could of course itself make use of trust management theory to discard information received from suspicious sources.

Support for human team players

The work explained in this report focuses on adding human-like behavior to bots. The theory of trust management could be expanded to model humans, and thereby detect untrustworthy players in larger online game communities. This may be used by other humans to determine trustworthiness of others, and by bots to make the interaction between these and humans playing the game more interesting.

Trust management in large online game communities

In this project trust management was used to distinguish tricksters from group oriented bots in predefined groups, in order to discard the information coming from the selfish bots when choosing roles. It would be interesting to examine the use of trust management in a large online game communities where bots could choose with whom to form groups from a large set of agents using a combination of trust and reputation values.

Trickery Alliances

Since more than one bot on a team may be a trickster, one could imagine some form of collaboration between these in order to make their own situation better. The bots would not have the same goal, as they would each only want to increase their own chances for survival, but providing false positive recommendation through reputation about each other to non-tricksters could make them both look more trustworthy thus bettering both their situations. If a number of tricksters could take part in such a trickery alliance it would take longer for them to be deemed untrustworthy by their non-trickster team mates.

Consider the situation illustrated in Figure 11.1.

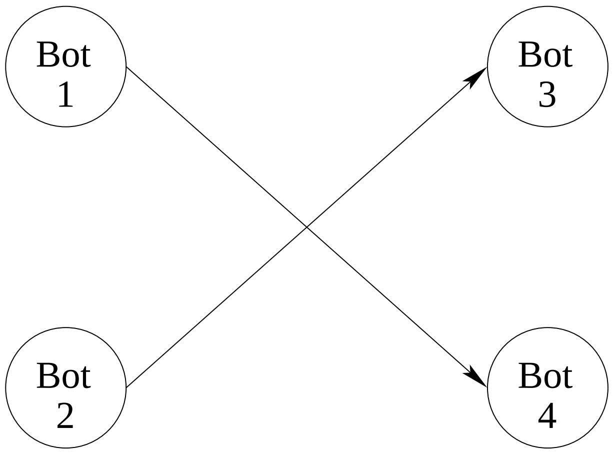


Figure 11.1: An example of a trickery alliance. Bot 1 and Bot 2 are tricksters. Bot 1 tricks Bot 4 and Bot 2 tricks Bot 3.

There is a total of 4 bots. Bots numbered 1 and 2 are tricksters and bots numbered 3 and 4 are non-tricksters. The arrows represent which bots trick which other bots. Bot 1 tricks Bot 4 and Bot 2 tricks Bot 3. The two tricksters are in a trickery alliance where they each trick only one of the two non-tricksters. Furthermore they provide each other with false positive recommendation through reputation when asked and provide false negative recommendation of Bot 3 and Bot 4.

In this way Bot 3 has good experience with Bot 1 and thus provides positive recommendations of Bot 1 to Bot 4. Bot 2 is a trickster and thus provides false positive recommendations of Bot 1 to Bot 4 and negative recommendation of Bot 3 and Bot 4 to all others. Now Bot 4 has two sources which have both provided positive recommendations for Bot 1. This will impact Bot 4's trust value in Bot 1 in a positive direction, even if Bot 4 already has had bad experience with bot 1.

As illustrated in the example above, an alliance of tricksters equal in size compared to non-tricksters, can completely dominate the non- tricksters by providing false positive or negative recommendations.

Now, when Bot 4 learns it is being tricked by Bot 1 in this situation, it decreases its trust in Bot 1. If trust values in the trust management implementation are updated based on the reputation provided, both Bot 2 and Bot 3 are percieved as being untrustworthy because both gave a good recommendation of Bot 1 even though it turned out to be a trickster. In this case Bot 3 is actually being mistreated as it gave a true recommendation based on its actual values.

The problem of discovering trickery alliances is not a trivial one and would require a great deal of examination of the problems described.

Trickery using false actions

As a final thought on future work one could imagine a trickster which does not only lie about what distribution of roles is most preferable but also does not perform the action it is told to by the group. This brings up a whole new set of problems about how the group should react to the behavior of such a bot and how its preferences and desires should be taken into consideration when distributing roles.

Part III

Appendix

Appendix A

Tests

A.1 Transcript

This is a transcript that exemplifies a scenario where Bot0 is a trickster and Bot1 and Bot2 are non-tricksters.
This first encounter shows how the trickster tries to determine thresholds, by exaggerating the expected utilities.

```
*** NEW ENCOUNTER ***
Trying to determine thresholds
Li_bot0[0]: 96
Li_bot0[1]: 131
Li_bot0[2]: 131
Li_bot0[3]: 196
Li_bot0[4]: 81
Li_bot0[5]: 146
Li_bot0[6]: 146
Li_bot0[7]: 211

Li_bot1[0]: 66
Li_bot1[1]: 131
Li_bot1[2]: 131
Li_bot1[3]: 196
Li_bot1[4]: 81
Li_bot1[5]: 146
Li_bot1[6]: 146
Li_bot1[7]: 211

Li_bot2[0]: 69
Li_bot2[1]: 128
Li_bot2[2]: 121
```

```
Li_bot2[3]: 189
Li_bot2[4]: 85
Li_bot2[5]: 151
Li_bot2[6]: 153
Li_bot2[7]: 200
```

The trickery of Bot0 is not yet detected, so Bot0 is voted IN by both Bot1 and Bot2.

```
** VOTES START **
Bot0 votes Bot1 IN
Bot0 votes Bot2 IN
Bot1 votes Bot0 IN
Bot1 votes Bot2 IN
Bot2 votes Bot0 IN
Bot2 votes Bot1 IN
** VOTES END **
```

Trust is updated accordingly.

```
** TRUST START **
** Bot0's trust in Bot1: 0.600000 **
** Bot0's trust in Bot2: 0.600000 **
** Bot1's trust in Bot0: 0.600000 **
** Bot1's trust in Bot2: 0.600000 **
** Bot2's trust in Bot0: 0.600000 **
** Bot2's trust in Bot1: 0.600000 **
** TRUST END **

Final utilities:
Lfinal[0]: 95
Lfinal[1]: 164
Lfinal[2]: 164
Lfinal[3]: 245
Lfinal[4]: 101
Lfinal[5]: 182
Lfinal[6]: 182
Lfinal[7]: 264

Chosen distribution of roles: Suppress, suppress, suppress
```

```
*** ENCOUNTER ENDED ***
```

In the second encounter, the trickster exaggerates even more.

```
*** NEW ENCOUNTER ***
Trying to determine thresholds
Li_bot0[0]: 126
Li_bot0[1]: 131
Li_bot0[2]: 131
Li_bot0[3]: 196
Li_bot0[4]: 81
Li_bot0[5]: 146
Li_bot0[6]: 146
Li_bot0[7]: 211

Li_bot1[0]: 66
Li_bot1[1]: 131
Li_bot1[2]: 131
Li_bot1[3]: 196
Li_bot1[4]: 81
Li_bot1[5]: 146
Li_bot1[6]: 146
Li_bot1[7]: 211

Li_bot2[0]: 69
Li_bot2[1]: 128
Li_bot2[2]: 121
Li_bot2[3]: 189
Li_bot2[4]: 85
Li_bot2[5]: 151
Li_bot2[6]: 153
Li_bot2[7]: 200

** VOTES START **
Bot0 votes Bot1 IN
Bot0 votes Bot2 IN
Bot1 votes Bot0 IN
Bot1 votes Bot2 IN
Bot2 votes Bot0 IN
Bot2 votes Bot1 IN
** VOTES END **
** TRUST START **
** Bot0's trust in Bot1: 0.666667 **
** Bot0's trust in Bot2: 0.666667 **
** Bot1's trust in Bot0: 0.666667 **
** Bot1's trust in Bot2: 0.666667 **
** Bot2's trust in Bot0: 0.666667 **
** Bot2's trust in Bot1: 0.666667 **
** TRUST END **
Final utilities:
Lfinal[0]: 125
Lfinal[1]: 191
Lfinal[2]: 191
Lfinal[3]: 286
Lfinal[4]: 118
Lfinal[5]: 213
Lfinal[6]: 213
Lfinal[7]: 295

Chosen distribution of roles: Suppress, suppress, suppress

*** ENCOUNTER ENDED ***
```

In this encounter, the trickster exaggerates even more.

```
*** NEW ENCOUNTER ***
Trying to determine thresholds
Li_bot0[0]: 156
Li_bot0[1]: 136
Li_bot0[2]: 198
Li_bot0[3]: 198
Li_bot0[4]: 141
Li_bot0[5]: 141
Li_bot0[6]: 203
Li_bot0[7]: 203

Li_bot1[0]: 74
Li_bot1[1]: 126
Li_bot1[2]: 126
Li_bot1[3]: 179
Li_bot1[4]: 77
```

```
Li_bot1[5]: 130
Li_bot1[6]: 130
Li_bot1[7]: 183

Li_bot2[0]: 77
Li_bot2[1]: 132
Li_bot2[2]: 120
Li_bot2[3]: 171
Li_bot2[4]: 72
Li_bot2[5]: 138
Li_bot2[6]: 136
Li_bot2[7]: 189
```

This time the trickster is detected and voted out.

```
** VOTES START **
Bot0 votes Bot1 IN
Bot0 votes Bot2 IN
Bot1 votes Bot0 OUT
Bot1 votes Bot2 IN
Bot2 votes Bot0 OUT
Bot2 votes Bot1 IN
** VOTES END **

** TRUST START **
** Bot0's trust in Bot1: 0.714286 **
** Bot0's trust in Bot2: 0.714286 **
** Bot1's trust in Bot0: 0.444444 **
** Bot1's trust in Bot2: 0.714286 **
** Bot2's trust in Bot0: 0.285714 **
** Bot2's trust in Bot1: 0.714286 **
** TRUST END **
Final utilities:
Lfinal[0]: 73
Lfinal[1]: 124
Lfinal[2]: 124
Lfinal[3]: 180
Lfinal[4]: 76
Lfinal[5]: 128
Lfinal[6]: 128
Lfinal[7]: 176
```

The trickster has lost its influence and in this case it gets the worst imaginable distribution of roles.

```
Chosen distribution of roles: Storm, suppress, suppress

*** ENCOUNTER ENDED ***

*** NEW ENCOUNTER ***
Trying to determine thresholds
Li_bot0[0]: 162
Li_bot0[1]: 126
Li_bot0[2]: 180
Li_bot0[3]: 183
Li_bot0[4]: 126
Li_bot0[5]: 129
Li_bot0[6]: 183
Li_bot0[7]: 187

Li_bot1[0]: 66
Li_bot1[1]: 131
Li_bot1[2]: 131
Li_bot1[3]: 196
Li_bot1[4]: 81
Li_bot1[5]: 146
Li_bot1[6]: 146
Li_bot1[7]: 211

Li_bot2[0]: 69
Li_bot2[1]: 128
Li_bot2[2]: 121
Li_bot2[3]: 189
Li_bot2[4]: 85
Li_bot2[5]: 151
Li_bot2[6]: 153
Li_bot2[7]: 200

** VOTES START **
Bot0 votes Bot1 IN
Bot0 votes Bot2 IN
Bot1 votes Bot0 OUT
Bot1 votes Bot2 IN
Bot2 votes Bot0 OUT
Bot2 votes Bot1 IN
```

```
** VOTES END **

** TRUST START **
** Bot0's trust in Bot1: 0.750000 **
** Bot0's trust in Bot2: 0.750000 **
** Bot1's trust in Bot0: 0.333333 **
** Bot1's trust in Bot2: 0.750000 **
** Bot2's trust in Bot0: 0.200000 **
** Bot2's trust in Bot1: 0.666667 **
** TRUST END **
Final utilities:
Lfinal[0]: 63
Lfinal[1]: 126
Lfinal[2]: 126
Lfinal[3]: 188
Lfinal[4]: 78
Lfinal[5]: 140
Lfinal[6]: 140
Lfinal[7]: 202

Chosen distribution of roles: Suppress, suppress, suppress

*** ENCOUNTER ENDED ***
```

Bibliography

[1] Bikramjit Banerjee, Anish Biswas, Manisha Mundhe, Sandip Debnath, and Sandip Sen. Using bayesian networks to model agent relationships. 2001.

[2] Jeffrey Broome. Botman's bots. http://www.planethalflife.com/botman/, 2004.

[3] Tom Clancy. Tom clancy games. http://www.tomclancygames.ubi.com/main.php, 2001.

[4] Finn Verner Jensen. *Bayesian Networks and Decision Graphs*. Springer-Verlag, 2001.

[5] Markus Klinge. Podbot. http://podbot.nuclearbox.com/, 2004.

[6] John E. Laird. It knows what you're going to do: Adding anticipation to a quakebot. In *Proceedings of the fifth international conference on Autonomous agents*, pages 385–392. University of Michigan, ACM Press New York NY, USA, 2001.

[7] John E. Laird and Michael van Lent. Human-level ai's killer application interactive computer games. Article from AI Magazine Summer 2001:15-25, 2001.

[8] Niels Christian Nielsen, Henrik Oddershede, Martin Thomsen, Alex Ringgaard, and Jacob Larsen. Advanced ai in computer games. http://www.cs.auc.dk/library/files/rapbibfiles1/1073308615.ps, 2004.

[9] Major NME. Swat tactics. http://www.nme.de/cgi-shl/nme/swat.cfm, 2001.

[10] Raymond Padilla. Counter-strike. http://archive.gamespy.com/e32003/preview/xbox/1001699/, 2004.

[11] Gabriela Serban. A new reinforcement learning algoritm. 2003.

[12] Dicky Suryadi and Piotr J. Gmytrasiewicz. Learning models of other agents using influence diagrams. In *Proceedings of the seventh international conference on User modeling*, pages 223–232. Department of Computer Science and Engineering, University of Texas at Arlington, Springer-Verlag New York, Inc, 1999.

[13] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, 1996.

[14] The CS Team. Counter strike official website. http://www.counter-strike.net/, 2003.

[15] William van der Sterren. Squad tactics: Team ai and emergent maneuvers. In Rabin, S., editor, AI Programming Wisdom, chapter 5 Tactical Issues and Intelligent Group Movement, pages 233-259. Charles River Media, inc., first edition., 2001.

[16] Yao Wang. Bayesian networks-based trust model in peer-to-peer networks. 2003.

[17] Steven M. Woodcock. The game ai page. http://www.gameai.com, 2004.