E3-111

Strategic Cooperation in First-Person Shooters

DAT6 REPORT
16th July 2004

Reason is the ability of a living creature to perform unreasonable or unnatural acts.

"Roadside Picnic" by Arkady and Boris Strugatsky.

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: Strategic Cooperation in First-Person Shooters

PROJECT PERIOD:
DAT6,
February 1th 2004 –
July 16 2004

PROJECT GROUP:
E3-111

GROUP MEMBERS:
Alex Vendelbo Ringgaard
Martin Guld Thomsen

SUPERVISOR:
Thomas D. Nielsen

NUMBER OF COPIES: 6

REPORT PAGES: 114

APPENDIX PAGES: 33

TOTAL PAGES: 137

SYNOPSIS:

This report presents two agent centered techniques for cooperation among agents. I.e. techniques to support cooperation among agents, without the need for the agents to engage in negotiation. This is for example needed to cooperate with human players in fast paced games like first-person shooters, where the human players only have time to send short messages to the agent.
The two techniques presented in this report is: Prediction, where an agent attempt to determine what actions other agents has or will take. This technique is used as the primary way to support cooperation, as it does not rely on any communication. The second technique, called value of communication, is used to determine whether it is worth to attempt to communicate in a given situation. This means both requesting information from another agent, as well as answering a request sent by another agent. This technique is used as a secondary technique to support the prediction, as it does rely on some message passing between agents.
While both techniques are general enough to apply to most areas of cooperation, this report focuses on strategical cooperation among defending agents in the game "Team Fortress" to show, through an empirical evaluation, that the techniques improve an agent's ability to achieve its goal.
The report concludes that the prediction technique is successful in improving the capabilities of the agents, and that the value of communication technique is capable of determining the most profitable times to request information from other agents.

Acknowledgments

We would like to thank the people at Hugin Expert A/S for proving us with a full license to Hugin Researcher 6.4 free of charge.

Reading Guide

This report is structured into three logical parts. The first part consists of Chapters 1 - 3. Chapter 1 is an introduction to the report and the concept of agents. Chapter 2 describes the agent environment used as test bed for the techniques developed in the report. Chapter 3 presents an agent architecture, called InteRRaP, with three layers (reactive, deliberative and cooperative), which we use in a slightly modified manner as basis for the agent implementation.

The second part of the report consists of Chapters 4 - 7 which describe the techniques used or developed in different layers in the architecture. Chapter 4 gives an description of Bayesian networks and influence diagram, which are the foundation for the implementation of the deliberative layer, described in Chapter 5. Chapter 6 presents the prediction and communication techniques developed for use in the implementation of the cooperative layer, described in Chapter 7.

The last part of the report consists of Chapter 8 and Chapter 9, which are the testing and conclusion, respectively.

When this report talks about agents in plural or an agent in general, they/it will be referred to in neuter, an agent who has assumed a defensive role is referred to in masculine and an agent in an offensive role is referred to in feminine. Furthermore, states of variables will be printed in **bold**.

The implementation can be found at [TR04], as can several pre-compiled versions for ease of use. The web site also contains an ps version of the report, as well as a hyperlink pdf.

Contents

1 Introduction

As the graphical quality of computer games increases, players start to expect more and more from a game. The more real the virtual world of a game seems, the more real the player expects the entities in the world to behave. As computer graphics are steadily nearing photo realistic quality, players are starting to expect entities to behave as they would in real life. Already games as "Black and White™" are using reinforcement learning, perceptrons and decision trees in the attempt to create an artificial intelligence[1] (AI) that learn and behave as one would expect from a domesticated animal.

Within the domain of First Person Shooters (FPS's), especially the team-based ones, there is an increased expectation among gamers to experience intelligent behavior from computer-controlled agents. Already these agents are capable of exhibiting simple cooperative behavior as following a player and obeying simple commands as "defend", "attack" etc. However, in most games, computer-controlled agents are depending on direct orders from human players to exhibit intelligent strategical behavior. If computer-controlled agents where capable of intelligent strategic cooperation, without needing orders from human players, they would appear more intelligent and increase the player's gaming experience. Also, it would allow agents on a team with bots to put up a more challenging fight without resorting to cheating.

1.1 Agent Definition

A popular metaphor is to think of each component of a system as an agent. As a result hereof there exist a lot of different definitions of what an agent is. However, most of these [FG96, WD99], no matter which area they originate from, generally agree on the fact that an agent must consist of the following four basic components:

- The information base. The agent's information about the environment it inhabits. E.g. a software advertising agent might know a list of products for sale, your e-mail address and previous purchases.

[1] The term AI is used in computer games due to historical reasons and is not equal to the general understanding of the term.

- The sensory component. The agent's mean of observing its environment. E.g. the advertising agent might be able to detect your activities on a particular web-site.
- The reasoning engine. This is the agent's means of making decisions, process information and learning from experience. E.g. the advertising agent might decide to send you an ad if you have bought something over the Internet two times this week.
- The actuators. The means of the agent to influence its environment. E.g. the advertising agents might be capable of sending e-mail.

Some also include a fifth component[WJ04].

- The social component. Agents interact with other agents via some kind of agent-communication language. This component is close related to the cooperation of agents, which are described in Section 1.3.1 on page 5.

To distinguish between the class of artificial agents (like robots and programs) and that of all agents. We use the term bot[2] about such agents. Likewise we use the term human controlled agents to when referring to agents under human control.

However, the components described above is not what makes an agent; it is only the components agents have to consist of. When considering the question about what makes a piece of software or robot an agent, people seem to be divided into two different camps. One side advocates the so-called *weak* notion[WJ04, WD99, FM99], which says that an agent should have the following properties:

- Autonomy
- Reactivity
- Pro-activeness

The first property says that the agent must be autonomous; meaning that it must be in control of itself, e.g. a light controlled by a switch is not autonomous, whereas one with a built-in sensor, instead of a switch, is. The second property requires the agent to react to its environment, a traffic light that changes in predefined intervals are not reacting to its environment, but if it were controlled by the actual traffic passing through it, it would be. To satisfy the last property of being pro-active, the system needs to

[2]The word "Bot" is often associated with computer-controlled agents in computer games, but in this report it is used about all computer-controlled agents regardless of domain.

take action by itself, not just react to environment, e.g. an automated dam should increase its production of electricity, causing the water level in its lake to drop if, based on the weather forecast, it looks like the lake might flood.

It is the presence of these three components that define what an (weak) agent is. The other side, however, advocate the *strong* notion[WJ04, WD99, FM99], which, besides the properties from above, also include the following properties and mental attitudes:

- Belief, knowledge, etc. (describing information states)
- Desire, goal, etc. (describing motivational states)
- Intention, commitment, etc. (describing deliberative states)

In the strong notion (often referred to as the BDI[3] architecture[Mül96]), the agents must also poses mental attributes, like knowledge and belief. The latter is divided into orders, if an agent uses first-order belief it means that it has beliefs, desires etc., but no beliefs and desires about beliefs and desires. Second-order belief is more sophisticated; here the agent has beliefs about beliefs and desires (called meta belief), both those of others and its own. Third-order is belief about belief about belief and so on [WJ94]. Secondly the agent should have desires and goals, things that it wants to achieve. Lastly it must be capable of forming intentions and commitments with the purpose of reaching its goals and desires.

Note that even when using the weak notion, described above, a lot of systems that might otherwise be thought to fall within the agent definition are excluded. Take for example a normal automated air condition system; while this has the property of autonomy and reactivity (assuming that it is controlled by the temperature) it lacks the property of pro-activeness, as does numerous other systems controlled solely by rules or scripts.

Throughout this report when the term agent is used, it shall refer to an agent defined under the strong notion. The definition of an agent is based on chapter 4 in [WD99].

1.2 Multi-Agent Environment

Now that the term agent has been defined we can define what a multi-agent environment is. Intuition says that it must be an environment inhabited by more than one agent. But this is not adequate; a room with two agents playing solitaire is not a multi-agent environment. Only if the agents are capable of interaction (possibly only indirectly) can the environment be said to be a multi-agent one. If there is no interaction between two agents, they

[3] Belief-Desire-Intension.

would effectively just consider the other agent a part of the environment and as such not consider its belief, goals and intentions.

[WD99] identifies three key aspects of a multi-agent system.

- The environment occupied by the multi-agent system. E.g. with respect to:
    - Diversity - How many different elements do the environment consist of? E.g. Chess is more diverse than checkers.
    - Uncertainty.
- The agent-agent and agent-environment interactions. E.g. with respect to:
    - Frequency - How often do an agent interact with the world or another agent?
    - Variability - In how many ways can an agent interact?
- The agents themselves

The first aspect is the environment, where some of the most important properties are whether it is deterministic or non-deterministic (ND). In a deterministic environment it is always possible to foresee the consequence of ones actions as opposed to a ND environment where the consequences of an action are unknown. Also, there might exist elements (e.g. other agents) in the world for which we might lack information, if this is the case we say that our agent must reason under uncertainty. Furthermore the world must be either static or dynamic, a static world only changes as a result of an action on our part, whereas a dynamic world can change regardless of us. Another important property is whether events in the environment happen in turns or continuously, if the latter is the case we call the environment "real-time". Existing in a real-time environment demands more of an agent, as the time available to contemplate and analyze a problem might be limited in contrast to a turn-based environment.

The second aspect is the interactions. Every time an agent use an actuator it inevitable mean that the agent is interacting with the environment or another agent. Note that this also covers communication with other agents.

The last of these three aspect gives rise to four different classifications of multi-agent environments as it can either be homogeneous or heterogeneous, with respect to the agents that inhabit the world (i.e. consisting of identical or different agents), and communicating or non-communicating (i.e. are the agents capable explicit communication or not). In this report, whenever "multi-agent environment" is used, it is a heterogeneous communicating one.

There are many reasons for using a multi-agent environment, They are inherently parallel in nature as each bot can be run on separate computer, as a result a multi-agent environment is often both scalable and robust[WD99].

Also, in a multi-agent system complex behaviors, that are very hard to design, might emerge as a result of the agents' interactions.

1.3 Cooperative Agents

If an agent in a multi-agent system share one or more of its goals with other agents, it might be desirable for the agent to coordinate its effort with these agents, as this could increase the agent's chance of success for its own goals. However, as each agent is independent and has its own view of the world, which is likely to differ from that of other agents. It is unlikely that these agents will make decisions that supports each other, even though they might share the same decision process and both chooses the action that is expected to maximize their utility, as they probably have different views of the world.

One way an agent can try to remedy the problem of cooperation under different views is to communicate its intentions to the other agents. The problem is that it is often not feasible to have each agent broadcast information every time its view of the world changes (e.g. due to bandwidth limits or information overflow for the receiving end). It is therefore desirable to minimize the amount of explicit communication between the agents or even completely eliminate it. Another way would be to try to infer what the intentions of other agents are. As a result agents must try to coordinate their actions in an environment with incomplete information and uncertainties.

1.3.1 What is Cooperation?

We will now give a brief discussion on what cooperation actually means in this report.

When looking at cooperation from an action aspect, we might say that two agents are cooperating when their actions satisfy at least one of the following [DFJN97]:

- The agents have a (possibly implicit) goal in common, for which they have individual motivation, and their actions tend to achieve that goal.
- The agents perform actions, which enable or achieve not only their own goals, but also the goals of other agents.

But as [DFJN97] point out this do not require the agents to intend to cooperate - cooperation can happen by sheer chance. We need to strengthen this definition by demanding that the agent is deliberative about the cooperation, it must reflect upon the combination of actions itself and others might perform and then, possible after some negotiation, choose actions the leads to convergence of its and the other agents behavior. To do this the concept of cooperation must be explicitly present in the agents' decision process.

Note that even by this definition there can be cooperation between agents without all of them necessarily knowing about it.

1.3.1.1 Failures

When working with cooperation it is also important that we understand in what ways it may fail, so that, during testing, it is possible to identify possible problems. [DFJN97] gives four general ways in which cooperation can fail.

- No cooperation is possible (i.e. there is no combination of actions that leads to cooperation).
- Cooperation is possible, but the agents cannot gather enough information to decide when and/or what actions to perform.
- Cooperation is possible and the information present, but the decision process is inadequate to process the information and select appropriate actions.
- Cooperation is possible and does indeed take place, but is accompanied by side effects that render the cooperation ineffective.

1.3.2 Level of Cooperation

In this report we focuses on the FPS's as multi-agent systems. Cooperation in a FPS can be divided into two different levels - A tactical level and a strategic level. Tactical cooperation consists of decisions on how to move and act relative to other agents in the immediate vicinity. E.g. for three policemen, about to kick down a door to a suspects house, the tactical decisions might be that the first policeman kick down the door, moves 3 feet into the house and then drops to his knee, expecting that the two others will cover each of his sides. Cooperation on the strategic level, on the other hand, consists of decisions on where to be and what to do in the grand scheme of things. E.g. the three policemen might know that there are another squad who enters from the back door and therefore they proceeds to search the first floor, leaving the ground floor to the second team.

Of course different environments require different decisions, but there is some common ground within the FPS domain. On the tactical level there are always decisions on how to react to a recently discovered enemy, which of the available weapons to use and how to move and position oneself relative to other agents. Strategically, an agent must also decide how to move and position himself, but this time in relation to some overall plan or scheme. Also, in many newer FPS[4] an agent must decide which resources he will

[4] E.g. Counter-Strike, Alien vs. Predator, and Battlefield 1942.

have access to in the game. This covers a wide range of possibilities; what weapons are available, what ”physical” law apply and much more.

In this report we focus on cooperation on the strategic level.

1.4 Project Goals

The goal of this project is to develop techniques that support cooperation between a bot and other agents in a FPS domain. We will focus on agent-centered techniques that will allow an agent to choose its actions according to the expected behavior of the other agents in the game, as opposed to choosing actions based on negotiation between agents, as it is often impossible for a bot to negotiate complex strategies with human players.

We will develop a technique for a bot to estimate what strategic decisions other agents has made, so that the bot itself can take decisions that support these. For the cases where a bot has doubts about the correctness of the estimated decision of another agent, we will present a technique that enables the bot to determine whether asking the other agent for information is expected to be worth the cost of communicating. Likewise we will make a technique to determine if it is worth to answer such a request for information from a friendly agent.

To test if the developed techniques are working, an implementation of a bot, faced with some realistic strategical problems (e.g. dedicated defense) in an actual FPS, will be made. To sum up, the goals for this project is:

- Create techniques that support agent-centered strategic cooperation.
- Implement a bot in a FPS domain using the developed techniques.
- Make an empirical evaluation of the techniques based on the implementation.

2 Agent Environment

One of the first FPS games to offer team multiplayer was Team Fortress. It was a modification to ID Software's Quake2 and it became one of the most popular games of its time. The game software company Valve noticed this and decided to buy Team Fortress and hire some of the people who created it. Valve then created their own modification named Team Fortress Classic (TFC), which was added as a free upgrade to their blockbuster hit Half-Life. But Valve did not stop there and decided to release an SDK-kit for Half-Life to help people create their own modifications for it. This is one of the largest successes ever in the gaming industry; the number of modifications to Half-Life is staggering[1] and Half-Life still has the largest number of people online here 7 years after its release. According to the "GameSpy Arcade" lobby program[2], which is a program used to find game servers, there are currently about 120.000 Half-Life players online, the second highest FPS game only has around 11.000 players. Half-Life is still the only game with a complete SDK for a commercial FPS game although that is likely to be changed when its successor Half-Life[2] is released.

The developers of Half-Life modifications usually release their code, which enabled other people to create bots for the modifications. But some of the largest modifications, like Counter-Strike and TFC, have decided not to publish their code. This made it nearly impossible to create bots for these modifications, for other than the original modification authors. Jeffrey Broome, who reverse engineered the way Half-Life loads modifications and created an interface for a bot, changed this. His work has been the foundation for many bots written to modifications with no released source code (See [bot04] for more information on the interface provided by Jeffrey Broome).

In this project the game TFC is used as a multi-agent test environment and throughout the rest of this report most examples will come from the TFC domain and it is therefore important to be familiar with the terminology used in TFC and understand its specifics, as they influences the model developed

[1] see www.planethalflife.com/community/hosted/mods.shtm and www.half-life.hu/indexeng.php for 100 of the reasonable popular ones.

[2] GameSpy Arcade can be downloaded from www.gamespy.com.

later in the report.

This chapter starts by describing the TFC environment and giving some examples of cooperation in TFC. Next, the environments characteristics are listed and thereafter some restrictions are imposed. Finally, reasons for using TFC as our agent environment are summarized.

2.1 TFC Terminology

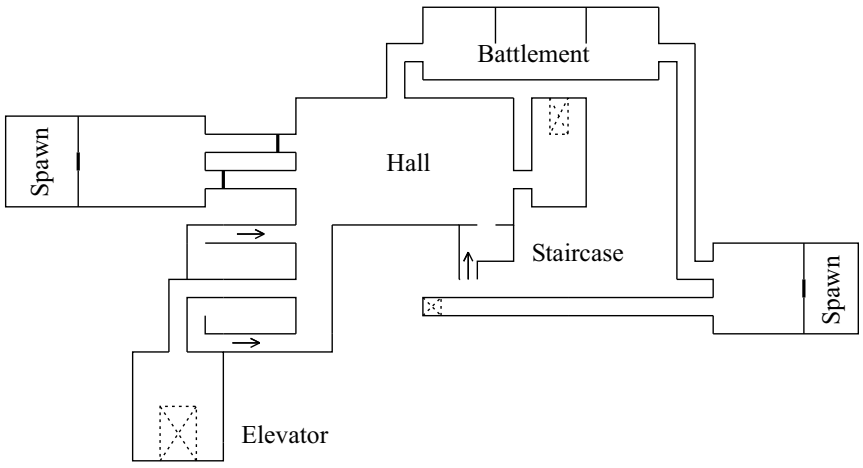
TFC is a team based game where 2 (or in some rare cases 4) teams of agents compete against each other on various maps[3]. On most maps in TFC, a certain portion of it ”belongs” to a specific team; this area is called the team's *base*. Each base has some area where an agent will respawn if killed. These areas are called *spawn points* and are often situated in areas that enemies cannot reach. Most bases also have places (called *resupply areas*) with items that the agents can collect to increase their attributes. Figure 2.1 on the next page shows the layout of a base on the map ”2Fort”.

The maps are played according to different goals called scenarios, some of the most common scenarios in TFC include:

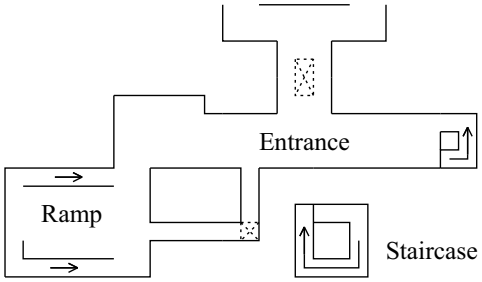
- ”Capture the Flag” (CTF), where each team has a flag they must protect while trying to get the opponents flag. The winning team is the one that captures most enemy flags; a flag is captured when it has been carried from an enemy's base to a predefined capture point.
- ”Attack and Defend”, where the defending team try to hold a number of positions that the offending teams must capture before a timer runs out.
- ”Assassination”, where one team tries to escort a VIP player to some predefined point on the map, while the assassins on the other team try to kill the VIP - the team that succeeds wins the match.
- ”Territorial Control”, where the teams fight to control a number of ”control points” which give the teams points according to varying rules depending on the specific map.

All of these scenarios are good choices for our tests, as they ensure that the agents have some clear goals in common which they can cooperate to achieve. In this report all test are performed using the CTF scenario, as this is the scenario used on the most appropriate map, which is described in Section 2.3.1.

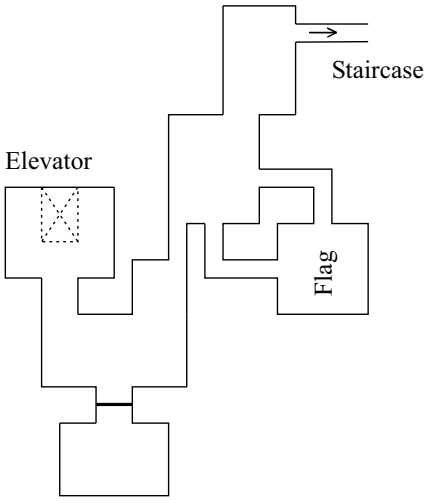
[3] A map is a specific game level, complete with buildings and collectable items.



(a) 1st floor



(b) Ground level



(c) Basement

Figure 2.1: Layout of a base in "2Fort". All arrows show ramps/stairs and points from lower to higher ground, and the dotted crosses show connected places where agents can jump/fall down from a floor to the one below.

2.2 TFC Agents

When an agent joins a TFC game, the first thing it is presented with is the choice of what class the agent wishes to play. The agent must choose one of 9 different classes to play, which influence how he is capable of acting in the world. I.e. what will his actuators be. Each class will grant the agent different characteristics such as, weapons, movement speed, health, armor and special abilities. The choice of class can be changed in-game, but will only take effect after the agent has been killed and respawns (see Appendix A on page 115 for a description of the different classes or www.planethalflife.com/tfc/ for general information). In addition to these actuators, that are dependent on the class choice, all agents can send plain text messages to any other agent. While it is theoretically possible to send any kind of information this way, messages are usually of the form: I am defending this area, I have the enemy flag, I need help etc. Of course, if the messages are to be understood by bots, they must be in some predefined format recognized by the bots. While, as mentioned, it is theoretic possible to send anything, e.g. a bot could send a complete model of its behavior, the general principle enforced is that a bot, in the name of fairness, only must send the same kind of information as human could.

The common sensor input to a bot consists primarily of "visual" information, identical to what a human player gets displayed on the monitor screen. However, the bots do not receive a picture to analyze as a human does, they get data corresponding to what a human can infer from such a picture. The general principal is that bots only has access to the same information as humans have. In addition to information about the position and movement of other agents in sight, it also includes information their class as well as what team they belong to. This information is available since these two factors unambiguously decide the appearance of an agent. However, agents always know the class of any other agents who are on the same team, as this information is available through a scoreboard.

The agent has sensors that tell it how much health, armor and ammunition it has left. Furthermore agents can see the health and armor of another agent if they are on the same team and within visual range of each other. In addition to these common sensors, there are some classes that gives an agent one or more special sensors. An agent who has chosen the Engineer class receives a sensor that enables the agent to see information about the status of its sentry gun (A sentry gun is one of the special abilities of an engineer, see the references above). Another noteworthy sensor comes with the scout class, a scout can always see the general state of the current games objective (See Section 2.1 for more details about the different types of games and their objective). All bots are also equipped with a virtual sense of hearing, but this has little impact in TFC as the "hearing" actually just means that the bot can "see" in a small circle behind itself and is, as such, only useful to

detect agents that are sneaking up behind the bot.

2.3 The Environment

The TFC environment is both real-time and inherently non-deterministic in nature, as is any FPS. This means that an agent only has a limited time to take critical decisions, as what to do if an enemy agent is encountered. The non-determinism means that an agent might not be able to foresee the actions of other agents nor the consequences of its own actions. E.g. if an agent spots another agent, he do not know where the spotted agent will move to next or if an agent fires a weapon it cannot known beforehand if it will hit or not. Also, as with any FPS, the environment is only partial observable, which is a great source of uncertainty as there many things in the world that an agent simply do not see since it can only observe the part of the world in which it is located. Another characteristic of the environment is that all sensors are virtually noise free, except from what is caused by lag, which means that some times an agent is not precisely where the sensor says it is, but as the deviance is usually very small, particular when there are enough available bandwidth, we do not deal with this.

As there, by definition, always will be other agents in a multi-agent environment and as they will always be capable of changing it (an agent that could not change its environment would no fulfill the requirements for being an agent), the environment can, from a single agents perspective, always change without the agent doing anything itself and the environment is, as such, a dynamic one.

2.3.1 The 2Fort Map

A 3D generated overview of the ”2Fort” map can be seen in Figure 2.2.

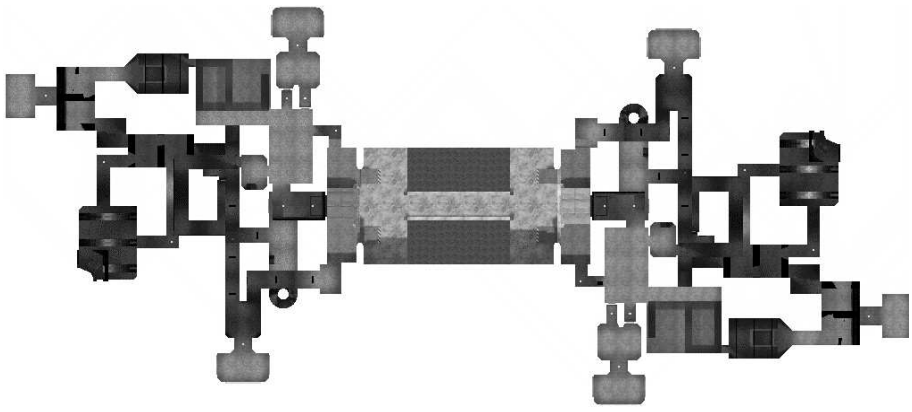


Figure 2.2: 3D generated overview of the ”2Fort” map.

The model, which is going to be the foundation for testing the methods developed in this report, will be based specifically on this map. It is therefore necessary to understand how the map is constructed and what its characteristic are, as they have a big impact on the design of the model and the understanding of the test results.

The 2Fort map is a CTF scenario for two teams. The general layout of the map consists of two bases, one for each team, separated by a body of water that is crossable by a bridge. Each base has a battlement overlooking the bridge and the other base's battlement (see Figure 2.3 and 2.4 on the facing page). This area is usually used as a vantage point for snipers, as it provides both cover and a good overview.

Directly below the battlement is the entrance area, which, besides the actual entrance to the base, consists of a corridor orthogonal to the actual entrance (see Figure 2.5).

After the entrance lies the ramp room (see Figure 2.6). This is a good area to defend in, as it is close to a resupply room and because enemies are very vulnerable when they try to run up the narrow ramps.

The top of the ramp room connects to a hall, which acts as a central hub for the base, as it connects to both the battlement, the entrance area, the aforementioned ramp room and the basement area which contains the team's flag (see Figure 2.7). In addition to the hall area, the ramp room also connects directly to the basement via an elevator. However, the elevator will only transport members of the team that own the base; an intruder that wishes to utilize this route must jump down the elevator shaft, thereby suffering the damage associated with such a fall.

As already mentioned the base has a basement where the team's flag is stored. Like the ramp room, the basement is a fairly good place to defend as it is close to a resupply room, however if enemy forces penetrate the defense it means that they will steal the flag (the room containing the flag can be seen in Figure 2.8).

All tests in this report are done on the map "2Fort" which has been chosen because of its size and design. It relative small size facilitates testing quite well as it does not require a large amount of agents, making it easier to observe the different interactions. Also, its design ensures that there are meaningful strategic decisions to be made both by defenders and attackers, as there clearly exist areas with dissimilar properties, meaning that certain combinations of defenders might be more appropriated in some areas than others. E.g. the battlement might be a better position for a sniper than the enclosed basement.

Figure 2.3: Screenshot of the battlement on the "2Fort" map.

Figure 2.4: Screenshot looking down from the battlement onto the bridge.

Figure 2.5: Screenshot of the entrance corridor. The actual entrance is to the right.

Figure 2.6: Screenshot of the ramp room. The opening in the middle of the picture is the connection to the entrance corridor.

Figure 2.7: Screenshot of the hall. To the left the opening to the battlement can be seen. Straight ahead is the opening to the entrance and to the right the way to the battlement. Further to the right, just outside the bounds of the screenshot, is the opening to the ramp room.

Figure 2.8: Screenshot of the flag room.

2.4 Cooperation in TFC

TFC contains many possibilities for tactical cooperation. In addition to the tactical possibilities normally found in FPSs, i.e. the agents can support each other in combat. TFC contains a host of other options. E.g. the medic class can restore the health of other agents by using his special ability, the engineers can repair other agents' armor and construct machines that dispense ammunition to nearby agents. This report, however, concentrates on strategic cooperation among the agents, but keep in mind that many strategic possibilities have their root in tactical possibilities. E.g. the reason some agents function better together than others can be that their tactical options complement each other.

There exist different possibilities for strategic cooperation in TFC:

- Which of the available classes should the agent choose?
- Should the agent occupy an offensive or defensive role?
- Where in the world should the agent go? For defensive agents this is a choice of position whereas for an offensive agent it is a choice of route to the enemy flag.

None of these choices in themselves constitute cooperation, but the choices must be coordinated with the other agents in the game. E.g. a team with two soldiers might benefit more from having a medic on the team than a third soldier or perhaps an engineer to defend the base while the two soldiers attack. Remember, per the definition of cooperation 1.3.1 on page 5, the agents can only be said to cooperate if the do so intentionally and not by sheer chance.

2.4.1 Examples of Cooperation in TFC

An agent wants to help defend the base and is considering what class to choose. It knows that there are two other agents on the team, what their current class are and it believes that one of them is defending the battlement, but it is unsure about the other agent. To get more information, it sends a message to the agent asking it where it is.

An agent defending the flag receives a message asking where he is currently located. The agent would like the other agent to join him so they can support each other and thus replies that he is defending the flag, believing that this information will increase the probability that the asking agent will join him.

A soldier has just been killed and is about to decide what to do next and what class to respawn as. He knows that there already are 3 soldiers defending the base, so he changes his class to Sniper and runs to the base's

battlements to pick of attacking agents, easing the burden on the three other defenders.

An engineer believes that there is a friendly soldier trying to defend the base's entrance, so he chooses to join him, using his special abilities to support the other agent by building a sentry gun and by repairing his armor.

2.5 Imposed Restrictions on TFC

Communication In TFC the portion of bandwidth needed to send a message to another agent is small in relation to the bandwidth commonly available, so bandwidth is not normally a concern, especially when the messages are between bots, as these are required to be run directly on the server machine and thus can "send" messages to each other instantly. However, in this report we shall impose a limit to the information that can be sent between agents. The limitation is introduced to simulate a multi-agent environment where agents are spread among several different machines, thus they cannot communicate unboundedly, and to simulate that human controlled agents cannot process and generate messages as fast as bots. The actual limitation is enforced by specifying an associated cost, depending on the size of he message, for sending the message (see Section 6.2 on page 69 for information on how the cost limits communication).

As all messengers in TFC are plain text, an agent could potentially send any kind of information to another agent, but as the focus in this report is on strategic cooperation, the agents will be limited to messages stating in which area they are and messages asking other agents where they are.

Classes As mentioned in Section 2.2 each agent in TFC must choose between 9 different classes (see Appendix A on page 115 for a description of the classes). However, this lead to a lot of different configurations of agents that all would have to be tested, so to make testing more feasible, agents are restricted to three of the nine classes: "Sniper", "HWGuy" and "Engineer" for agents in defensive roles and "Scout", "Soldier" and "Medic" for offensive roles. These classes have been chosen based on their disposition towards the role they are confined to. This restriction significantly lowers the necessary amount of test results needed, but without removing the diversity of the TFC environment.

2.6 TFC Advantages as Environment

The arguments for using TFC as a multi-agent test environment is that it is an already working team game and as such designed to support and rewards cooperation. It already contains an implementation of (uncooperative) bots,

parts of which can be substituted and modified to create our own bot. Furthermore, since we have complete access to the source code, we can enforce restrictions on the game, which was not originally there, such as bandwidth limitations.

Also, TFC contains many aspects often found in multi-agent environments. It is non-deterministic, dynamic and consists of heterogeneous agents who must try to cooperate and coordinate with each other based on observation and communication abilities. Also, like any FPS, the environment is only partial observable adding to the agents need to reason under uncertainties.

The last, and perhaps most noticeable, advantage of using TFC, as opposed to another FPS, is that Jeffrey Broome's code can be used to handle all the agent behavior that fall outside of the project's focus, allowing us to focus on the coorperative aspects of the agent.

3 Agent Architecture

The TFC domain described in Section 2 on page 9 is a real time multi-agent environment; in addition, the environment is dynamic and non-deterministic seen from a single agent's point of view. The TFC domain is designed to support and reward cooperation between agents.

In [Mül96], Jörg Müller set up three requirements for agents in a dynamic real-time multi-agent environment. These characteristic describe the TFC domain and we therefore measure agents in the TFC domain by their ability to satisfy the three requirements:

- An agent should select and carry out its actions in real-time.
- An agent should select its actions based on the actions' ability to achieve the agent's objectives.
- The agent should coordinate its actions with the other agents in order to fulfill the second requirement.

The first requirement, henceforth known as the real-time requirement, result in limitations of the resource requirements for the agent's decision process, e.g. limited amount of CPU-cycles or memory usage. The second requirement entails that the agent should act rationally. By acting rationally we mean making decisions that maximizes the expected utility (the so-called normative approach). The third requirement can be seen as an addition to the second requirement. In order to coordinate the agent's actions with other agents' behavior, information about their behavior must be collected. This could e.g. be obtained through observations of other agents or message exchange with other agents.

Different architectures have been developed to fulfill these requirements and a survey of such architectures can be found in [Mül96]. In the sections below three general architectures that each fulfill one of the requirements are described, followed by a short discussion of their strength and weaknesses.

3.1 Reactive Architectures

The reactive architectures are the simplest architectures, as their decision process is based solely on the current sensor input. The decision process classifies the input into one of a predefined number of states that each has an action attached to it, which is then executed. The action for each state is decided at the time of implementation and it is therefore more or less an advanced way of triggering scripting, this also means reactive architectures have the usual disadvantages associated with scripting. E.g. complex environments usually require a large amount of states to adequately classify the input. Each of these states must be anticipated and manually evaluated to attach an appropriate action, thus it scales poorly and as the actions are determined at compile time it does not provide adaptive behavior. The advantage is that the decision process usually has low resources requirements, since it only consist of categorizing the input, which normally is a simple task (although the amount of input can change this).

The advantages of reactive architectures make them well suited to fulfill the real-time requirement. They are also capable of robust behavior[1] in complex environments, however as described this often requires an enormous amount predefined states and manually evaluated analysis of these, in addition it is difficult to achieve an optimal behavior[2], especially in dynamic environments.

3.2 Deliberative Architectures

Deliberative architectures contain mental attributes, e.g. knowledge and beliefs. The decision process is based both on the mental attributes' states and inputs from the environment. A deliberative architecture has the potential to achieve the individual optimal behavior, as its knowledge can include a complete representation of its environment. Often however, this is often not feasible for complex environments, although it gives the best result it also requires enormous amounts of resources.

Deliberative architectures are therefore good at fulfilling the second requirement, but they require more resources, which can give problems with the real-time requirement, this is most apparent in situations that require immediate actions.

[1] Robust behavior is the ability to recover gracefully from the whole range of exceptional inputs and situations in a given environment. I.e. robust behavior should always be able to carry out a reasonable action, although the input is not complete or precisely as anticipated.

[2] Optimal behavior should carry out the most desirable action possible given any inputs and situations in a given environment.

3.3 Cooperative Architectures

Architectures that facilitate interaction between agents have primarily focused on coordination between agents. The coordination can be based on information gathered by observations, predictions and/or communication. The main research areas are techniques to predict other agents' actions, techniques for communication between agents and also coordination between autonomous agents. These architectures can be constructed based on both reactive and deliberative architectures, and therefore share the respective architecture's advantages and disadvantages.

Cooperative architectures are constructed to fulfill the third requirement, but in order to fulfill this requirement they employ additional resources for coordination. However, as they aim to combine the agents' efforts, which in some environments give a larger utility than the agents' separate effort, they have the possibility to further optimize the expected utility.

Discussion The three architectures each address one of the requirements, often however, an agent is measured on its ability to fulfill all three requirements at the same time; in these cases none of them is suitable. Instead it would be desirable to have an architecture that combines the advantages of the three architectures. A logical approach is to try to combine the strength of each architecture in a hybrid architecture. Such an architecture will be described in the next section.

3.4 The InteRRaP Architecture

In [Mül96] Jörg Müller develops a hybrid architecture, called InteRRaP[3], that more or less combines the three described architectures into one. An overview of the architecture can be seen on Figure 3.1 on the next page.

The InteRRaP model consists of three parts: The World Interface Component (WIC), a Knowledge Base (KB) and a Control Component (CC). The WIC controls the interface between the environment and the agent; this includes actuators, communication and sensors.

The KB and the CC are each divided into three layers, where a layer in one part has a corresponding layer in the other part. The basic idea is that each layer handles one requirement and that the following layer represents a higher level of knowledge abstraction than the layer below it.

Knowledge Base The KB is a hierarchical structure. This means that a layer in the CC can access information stored in its corresponding KB layer and the layers below it, but not layers above it. The World Model[4]

[3] **Inte**gration of **R**eactive Behavior and **Ra**tional **P**lanning.

[4] Note that name World Model is a bit misleading as it only contains current inputs.

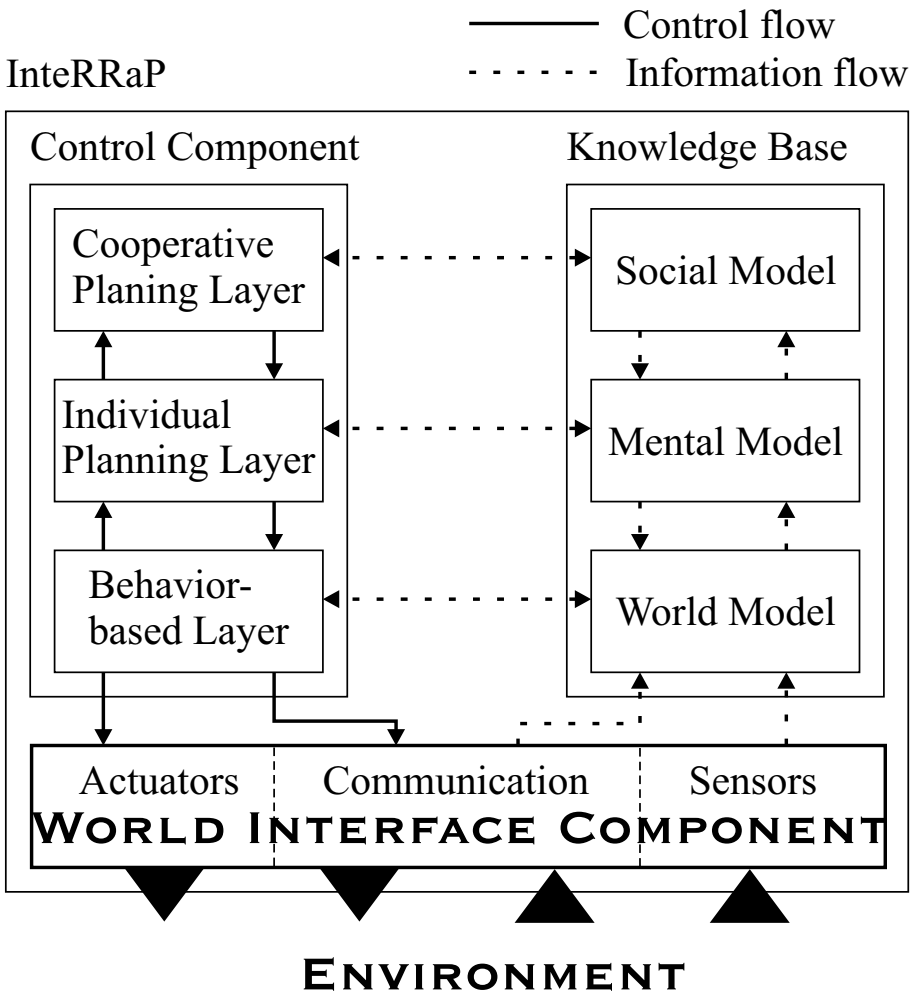


Figure 3.1: The InteRRaP agent architecture, based on [int04] and [Mül96].

is the lowest layer in the KB; it only contains the current inputs from the WIC, and as it is the lowest layer all layers in the CC can access it. The Mental Model contains previous experience and a representation of the environment for a specific agent. The Social Model contains descriptions of negotiation protocols, communication protocols, methods to predict other agents' behavior and belief about other agents.

Control Component The lowest layer in the CC is the Behavior-based Layer, which is a reactive layer that handles all basic situations and situations with a demand for immediate action. In practice the Behavior-based Layer can be implemented as a number of pre-conditions for the World Model, each pre-condition has an action that should be performed when it is fulfilled. An action can either specify a direct interaction with the WIC or be request to the next layer if the situation is to complex, e.g. the situation need to handled based on previous experiences, it require interaction with other agents.

The next layer is the Individual Planning Layer, which is a deliberative layer that is used in situations without any need for coordination between agents. It generates plans based on the Mental Model, which only contains information for this specific agent. If the situation does not require interaction with other agents, it specify a sequence of actions that the Behavior-based Layer must carry out, otherwise it will pass the control to next layer.

The last and highest layer is the Cooperative Planning Layer. It generates plans that involve other agents and handles all interaction between agents. As the Cooperative Planning Layer is the highest layer it cannot pass the control further up, therefore it has to construct a plan based on the Social Model and KB layers below it. The Cooperative Planning Layer generally deals with situations that require knowledge about other agents or coordination with other agents.

Input from either the agent's sensors or communication are handle by the WIC, which update the World Model accordingly. The changes in the World Model can result in a precondition from the Behavior-based Layer being met. This triggers the CC and the decision process is started.

3.5 Architecture for the TFC Domain

The InteRRaP architecture specifies a feasible way to combine the reactive, deliberative and cooperative architectures into a hybrid architecture. The agent (which will be made to fulfill a defensive role) designed in this project will therefore use the fundamental design principles from the InteRRaP architecture. I.e. the principles of having three layers, which each handles one of the three requirements (see page 21). Figure 3.2 on the following page shows the agent architecture used in this project, the difference be-

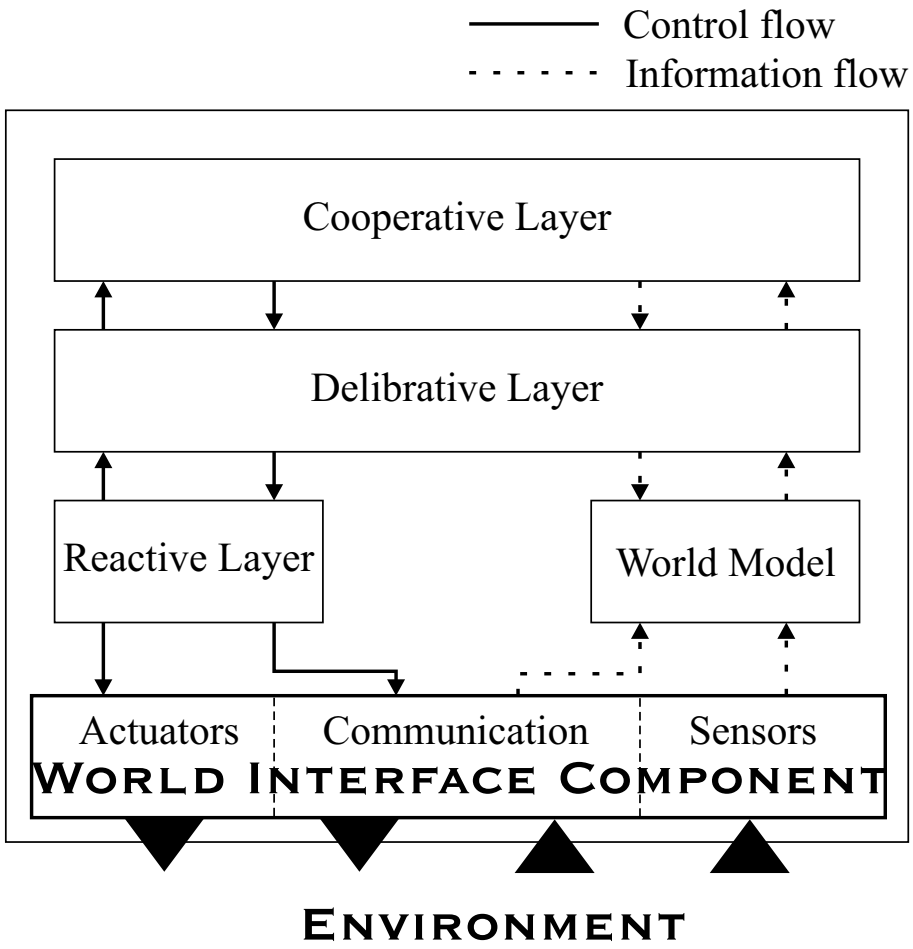


Figure 3.2: The agent architecture employed in this project.

tween our agent architecture and the InteRRaP architecture is that the CC and the KB has been combined for two upper layers, i.e. the Individual Planning Layer and the Mental Model make the Deliberative Layer and the Cooperative Planning Layer and the Social Model makes the Cooperative Layer. The CC and KB merging is caused by the influence diagram technique employed in the Deliberative and Cooperative Layers, influence diagrams combine a representation of knowledge with the decision process, the technique is described in next chapter. The next sections describe each layer's task assignments in the defensive agent developed in this project.

3.5.1 The Reactive Layer

The reactive layer handles situations that require instant action as well as situations that can be resolved based on the current sensor input alone.

In our defensive agent implementation, reactive decisions (e.g. about how to move, what route to take when going to a destination, how to stay near to a specific location etc.) is handled by the module provided by Jeffrey Broome. Likewise decisions on when to invoke special abilities, such as the engineer's ability to repair the armor of nearby friendly agents or a medic's ability to heal other agents, are handled reactively. I.e. if someone is in need of these services and the agent is close to them it will use its abilities if appropriate. Furthermore, decisions about engaging an enemy are also handled as reactive behavior. A bot will always engage a visible enemy, but will not pursue them, if it takes the bot away from its current goal (defending a zone or capturing the flag). E.g. a scout carrying the enemy flag will fire upon enemies, but will not deviate from the route to the capture point. Also, other minor decisions, as how much to look around and how much to strafe[5] etc., are left to the reactive layer.

In effect this means that all kinds of tactical decisions, and thereby tactical cooperation (See Section 1.3.2 on page 6 for the difference between tactical and strategic cooperation), are handled reactively. As the reactive implementation is handled solely by Jeffrey Broome's code, the different techniques to implement reactive behavior will not be discussed in this report.

Note that some of the decision left to the reactive layer, may not be reactive (or tactical) decisions for all kinds of agents. E.g. as we are going to make an implementation of a defensive agent, path finding is left to reactive layer. Had the implementation been that of an attacking agent, this would most likely have been a strategical decision.

3.5.2 The Deliberative Layer

The deliberative layer handles strategic decisions, which are general decisions for an agent's plan in the larger scheme (e.g. decisions to defend specific locations in the map or choosing which resources are available etc.). These decisions do not have to be made within a strict time limit and should strive to achieve optimal behavior, in relation to the available strategical options, which makes them suitable to a deliberative decision process. When the deliberative layer has selected a decision, it should leave the execution to the reactive layer. In Chapter 4 techniques for a deliberative decision process are described, and in Chapter 5 the deliberative layer of the defensive agent developed in this project are constructed.

3.5.3 The Cooperative Layer

In contrast to the cooperative layer in the InteRRaP architecture, the cooperative layer task is not to produce plans for the defensive agent; instead

[5] Moving left or right while still facing forward.

its main task is to produces information about other agents' behavior. This could be based on observations of their behavior, communication and beliefs about their world view. The information gathered from this layer should subsequently be used by the deliberative layer. In Chapter 6 techniques for the cooperative layer are developed, and in Chapter 7 the cooperative layer of the defensive agent are constructed.

4 Bayesian Networks and Influence Diagrams

In the deliberative layer there are two major factors that influences how the bot makes decisions. The first factor is imposed by the requirement that the agent must make rational decisions. In our case that is, the agent must take the actions that are expected to maximise the bot utility. The second factor the shapes that agents deliberative decision process is the environment - the fact that a FPS domain has built-in uncertainties places a demand on the agent to be able to reason under these uncertainties, like determine the value of its possible decisions without knowing for sure what the consequence of each might be.

Bayesian networks are a tool that can be used to help an agent to reason under uncertainties. Bayesian networks use a model of the problem domain in order to determine the probability of different variables of interest being in specific states.

This chapter will describe Bayesian networks and influence diagrams, which is an extension to Bayesian networks that explicit models a decision process. The chapter is mainly based on [Jen01].

4.1 Bayesian Networks

A Bayesian network is a tool for reasoning under uncertainty based on the Bayesian probability calculus. A Bayesian network consists of a directed acyclic graph $\mathcal{G}$ where each node represent a variable in the domain modeled by the network. Each node has a set of mutually exclusive and exhaustive states and is associated with a conditional probability table $P(A|B_1, \ldots, B_n)$, where A is the variable and $B_1, \ldots, B_n$ its parents ($pa(A)$) in $\mathcal{G}$. Note that if A has no parents this reduces to an unconditional probability table $P(A)$.

When construction a Bayesian network, the edges in the graph usually represent causal relations, i.e. cause-effect relations between the nodes. However, this is only a general advice, and not a requirement. There is no rule that demands that edges represent causal impacts, only that they adhere to the d-separation properties implied by the network. The reason for using causal relation, even though it is not required, is that it automatically ensures that the d-separation property holds.

4.1.1 d-separation

To understand the concept of d-separation, one needs to know what evidence on a variable means. Evidence is a statement about the certainty of a variable A being in one or more of its states. If the evidence says that A is certainly in some specific state a_i it is called *hard evidence* and we say that the node has been instantiated. If the evidence only consists of belief about a node (e.g. it is with probability 0.4 in state 1) the evidence is called *soft evidence*.

When evidence is entered into a variable in a Bayesian network it can changes our belief about other variables in the network. E.g. in the network in Figure 4.1 evidence on B will change our belief about C since the probabilities for C depends on B, but it may also change our belief about A as B is dependant on A. However, dependant on the structure of the graph and what nodes in the network that have been instantiated, there are cases where evidence about one variable never can change our belief about another. In these cases the variables in question are said to be d-separated or structural independent.

To determine how information can flow though the network, and thus what variables that are d-separated, there are three types of connections among variables that must be considered:

Serial Connections Figure 4.1 shows the first type of connection, 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 can change the certainty of A through B. However, if B is instantiated, i.e. the state of the variable is known, then variables A and C cannot influence each other and is therefore d-separated given B.

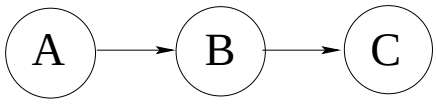


Figure 4.1: Serial connection.

Diverging Connections The second connection type is the diverging connection shown in Figure 4.2. Here evidence about A can change the belief about B which in turn can change the belief about C, and visa versa. However, if B has been instantiated, evidence about A cannot change belief about B, thus evidence on A does not tell anything about C and the variables are d-separated.

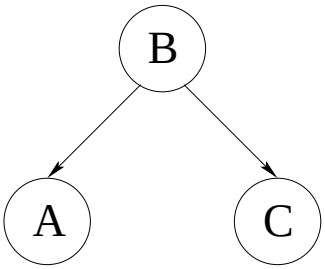


Figure 4.2: Diverging connection.

Converging Connections The last connection type is the converging connection. This concept is illustrated in Figure 4.3. Here, evidence on the variable B can change the belief about A but not about C. If, however, there is hard evidence on A (or any of its children) the evidence about B can tell something about the state of C and thus change the belief about it.

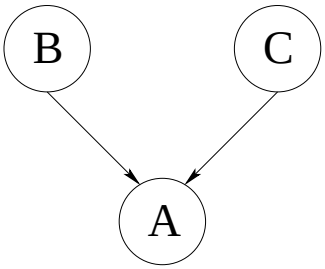


Figure 4.3: Converging connection.

The ways, just described, in which evidence can block the flow of information in a network, are reflected in the concept of conditional independence. A variable A is independant from C given B if $P(a_i|b_j) = P(a_i|b_j, c_k)$, i.e. if the state of B is know, information about C cannot change the belief about A. Note that two variable can be conditional independant without being d-separated.

4.1.2 Joint Probability

The reason for using a joint probability table $P(U)$ over all nodes, is that it enables us to find the probability distribution $P(A)$ of a variable A. For example, if we have the joint probability table $P(A, B)$, we can find the probability for each state a_i in A with the formula:

$$P(a_i) = \sum_{j=1}^{n} P(a_i, b_j) \tag{4.1}$$

where

n is the number of states in B

We call this process marginalization and we say that B has been marginalized out of A. In general marginalization is denoted as follows:

$$P(V) = \sum_{S/V} P(S) \tag{4.2}$$

where

S is the set of variable.
$P(S)$ is the joint probability table for S.
V is the variables for which a joint probability table is wanted.

The joint probability table can be calculated from the conditional probability tables by using the *chain rule*:

$$P(U) = \prod_{i} P(A_i|pa(A_i)) \tag{4.3}$$

The reason for using a Bayesian network, with all it conditional probability table, instead of just having $P(U)$, is that the size of $P(U)$ grow exponentially with the number of nodes in U. E.g. a joint probability table over 10 nodes, with 10 states each, would contain 10 billion probabilities. Compared to this the Bayesian network constitute a much more compact representation of $P(U)$. Of course this would not help if the entire joint probability table had to be computed every time some specific probability are needed. Fortunatly this is not necessary, the example below shows how we can find probability distribution $P(A)$ for a node A by using *Bayes rule*:

$$P(B|A, C) = \frac{P(A|B, C)P(B|C)}{P(A|C)} \tag{4.4}$$

from which we get the *fundamental rule*:

$$P(A, B|C) = P(A|B, C)P(B|C) \tag{4.5}$$

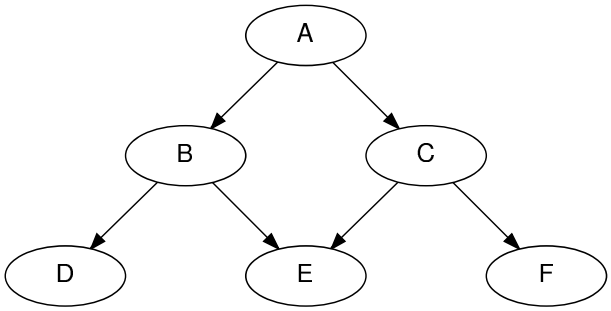


Figure 4.4: Example of Bayesian network.

Example In the network in Figure 4.4, we would like to know the probability distribution $P(D)$.

$$P(D) = \sum_{U/D} P(U) \Rightarrow$$

$$P(D) = \sum_{A,B,C,E,F} P(A)P(B|A)P(C|A)P(D|B)P(E|B,C)P(F|C) \Rightarrow$$

$$P(D) = \sum_{A} P(A) \sum_{B} P(B|A)P(D|B) \sum_{C} P(C|A) \sum_{E} P(E|B,C) \sum_{F} P(F|C) \Rightarrow$$

$$P(D) = \sum_{A} P(A) \sum_{B} P(B|A)P(D|B)$$

The reason $\sum_C P(C|A) \sum_E P(E|B,C) \sum_F P(F|C)$ can be disregarded is when the variables are marginalized out the sum will be 1 and thus it do not influence the result. The probability tables for all relevant nodes can be seen in the three tables below.

$P(D\|B)$	b_1	b_2
d_1	0.2	0.4
d_2	0.8	0.6

$P(B\|A)$	a_1	a_2
b_1	0.3	0.1
b_2	0.7	0.9

$P(A)$	
a_1	0.5
a_2	0.5

To find $P(D)$, we must compute the joint probability table $P(A,B,D) = P(D|B)P(B|A)P(A)$. To do so we first calculate $P(A,B) = P(B|A)P(A)$ by using the fundamental rule:

$P(A,B)$	b_1	b_2
a_1	$0.3 \cdot 0.5$	$0.7 \cdot 0.5$
a_2	$0.1 \cdot 0.5$	$0.9 \cdot 0.5$

$=$

$P(A,B)$	b_1	b_2
a_1	0.15	0.35
a_2	0.05	0.45

and likewise the $P(A,B,D) = P(D|B)P(A,B)$:

$P(A,B,D)$	b_1	b_2
a_1	$(0.15 \cdot 0.2, 0.15 \cdot 0.8)$	$(0.35 \cdot 0.4, 0.35 \cdot 0.6)$
a_2	$(0.05 \cdot 0.2, 0.05 \cdot 0.8)$	$(0.45 \cdot 0.4, 0.45 \cdot 0.6)$

$=$

$P(A, B, D)$	b_1	b_2
a_1	(0.03, 0.12)	(0.14, 0.21)
a_2	(0.01, 0.04)	(0.18, 0.27)

where the parenthesis corresponds to (d_1, d_2). A and B can now be marginalized out of $P(A, B, D)$ by using Equation 4.2, which gives the probability:

$$P(D) = (0.03 + 0.14 + 0.01 + 0.18, 0.12 + 0.21 + 0.04 + 0.27) = (0.36, 0.64)$$

4.2 Influence Diagrams

Bayesian networks are used to calculate probabilities for subsequent decision making. However, as making decisions is the reason for using a Bayesian network, it could make sense to include these decisions directly in the network. This is done with in extension of Bayesian networks, called influence diagrams, that explicitly models the decisions that must be made and the utility of the different situations.

An influence diagram is a normal Bayesian network extended with two new types of nodes: Decision nodes and utility nodes. Where chance nodes represent events beyond direct control of the decision maker, decision nodes represent choices that are under complete control and thus have no probability table. All states in a decision node represent a possible decision and must be mutually exclusive and exhaustive. Decision nodes need to follow a strict temporal ordering, so that it is clear which decision is the first, which is the second and so on. Structurally this means that there must be a directed path in the diagram that includes all decision nodes. See Figure 4.5 for an example of an influence diagram.

A utility node N represents a real-valued function over the node's parents. I.e. a utility node map all parent configurations to a value, called the utility, which specifies how desirable the configurations is. Note that there

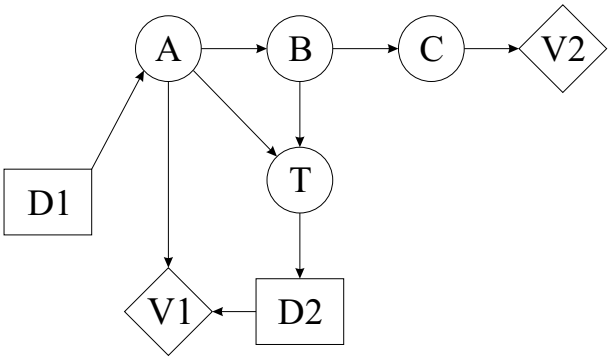


Figure 4.5: Example of an influence diagram.

can never be outgoing links from a utility node, i.e. it can never have any children. Like a Bayesian network is a multiplicative decomposition of the joint probability table, it is possible to make an additive decomposition of the utility function. This means that it is possible to have more than one utility node in an influence diagram. If this is the case, then the final utility of a given configuration of parents is the summed value of all utility nodes. This means that utility nodes, like decision nodes, are not associated with any probability table and, unlike both decision and chance nodes; they do not have any states.

These new types of nodes gives rise to a new type of link in addiction to the (usually causal) links from one chance node to another. These links always comes from a chance node or decision node (Q) and goes into a decision node (W). The link expresses that Q is observed or known before W is to be made. Based on what type of node Q is the link are called either:

- Information links: These are links from a chance node into a decision node.
- Precedence links: These are links from one decision node to another.

The precedence links are used to fulfill the requirement that there must be a directed path that comprises all decision nodes. Note that this does not mean that all decision nodes need to be connected with precedence links. If a chance node that depends on decision node A is observed before decision B, then these two decision node do not need a precedence link as B obviously follows A. Also note that information and precedence links are to be ignored when determining d-separation. So are links to utility nodes that, except for this purpose, functions as a normal links.

4.2.1 Solving an Influence Diagram

Given an influence diagram, the task is to determine what decision to select for each decision node given the past. Before describing how to determine this, some terminology is needed.

- Let the temporal ordering of the nodes be $I_0 < D_1 < I_1 < D_2 < \ldots < I_{n-1} < D_n < I_n$, where D_i is the i'th decision, I_0 is the set of initially observed chance nodes, I_i is the chance nodes observed between D_i and D_{i+1} and I_n are the chance nodes who are observed either after the last decision or never at all. See Figure 4.6 on the following page for an example of the ordering for the diagram in Figure 4.5.
- A policy σ for a decision node D_i is a function, which given any configuration of the past $past(D_i) = I_0, D_1 \ldots, D_{i-1}, I_{i-1}$, yield a decision for D_i. An optimal policy is the policy that given $past(D_i)$ yields the decision with the highest expected utility (EU).

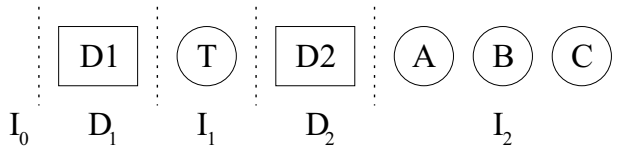


Figure 4.6: The ordering for the influence diagram in Figure 4.5.

- A strategy υ for an influence diagram ID is a set with a policy for each decision in ID. An optimal strategy (sometimes called a solution) is the strategy consisting of optimal policies.

This means that an optimal policy for a decision D_i, in an influence diagram over the chance nodes U_C, the decision nodes U_D and the utility function $V = \sum_i V_i$ where V_i is the ith utility function, can be calculated as follows:

$$\sigma_i(I_0, D_1, \ldots, I_{i-1}) = argmax_{D_i} \sum_{I_i} max_{D_i+1} \ldots max_{D_n} \sum_{I_n} P(U_C|U_D)V$$

To determine the EU for following σ_i, first consider the EU for a decision for D_n given $past(D_i)$ - This is a sum over the probability of I_n given $past(D_i)$ times the associated utility.

$$EU(D_n|I_0, D_1, \ldots, D_{n-1}, I_{n-1}) =$$
$$\sum_{I_n} P(I_n|I_0, D_1, \ldots, D_{n-1}, I_{n-1}, D_n)V =$$
$$\sum_{I_n} \frac{P(I_n|I_0, D_1, \ldots, D_{n-1}, I_{n-1}, D_n) \cdot P(I_0, \ldots, I_{n-1}|D_1, \ldots, D_n)}{P(I_0, \ldots, I_{n-1}|D_1, \ldots, D_n)} V =$$
$$\sum_{I_n} \frac{1}{P(I_0, \ldots, I_{n-1}|D_1, \ldots, D_{n-1})} P(I_n, I_0, \ldots, I_{n-1}|D_1, \ldots, D_n)V =$$
$$\frac{1}{P(I_0, \ldots, I_{n-1}|D_1, \ldots, D_{n-1})} \sum_{I_n} P(U_C|U_D)V$$

which means that the EU for making the optimal decision for the last decision must be:

$$\rho_n(I_0, D_1 \ldots D_{n-1}, I_{n-1}) = \tag{4.6}$$
$$\frac{1}{P(I_0, \ldots, I_{n-1}|D_1, \ldots, D_{n-1})} max_{D_n} \sum_{I_n} P(U_C|U_D)V$$

From Equation 4.6 it follows inductively (although we do not prove so) that:

$$\rho_{i+1}(I_0, D_1, \ldots, I_i) = \tag{4.7}$$
$$\frac{1}{P(I_0, \ldots, I_i|D_1, \ldots, D_i)} max_{D_{i+1}} \sum_{I_{i+1}} \ldots max_{D_n} \sum_{I_n} P(U_C|U_D)V$$

Now the EU for an arbitrary decision D_i can compute, as this must be the sum over I_i times the utility for taking only optimal decisions in the future.

$$EU(D_i|I_0, D_1 \dots D_{i-1}, I_{i-1}) =$$
$$\sum_{I_i} P(I_i|I_0, D_1, \dots, D_{i-1}, I_{i-1})\rho_{i+1}(I_0, D_1, \dots, I_i) =$$
$$\sum_{I_i} \frac{1}{P(I_0, \dots, I_{i-1}|D_1, \dots, D_i)} P(I_i, I_0, \dots, I_{i-1}|D_1, \dots, D_i)\rho_{i+1}(I_0, D_1, \dots, I_i)$$

and if we substitute Equation 4.7 into this, we get that:

$$EU(D_i|I_0, D_1 \dots D_{i-1}, I_{i-1}) = \tag{4.8}$$
$$\frac{1}{P(I_0, \dots, I_{i-1}|D_1, \dots, D_{i-1})} \sum_{I_i} max_{D_{i+1}} \sum_{I_{i+1}} \dots max_{D_n} \sum_{I_n} P(U_C|U_D)V$$

Which means that the maximum expected utility (MEU) for an influence diagram is:

$$MEU(ID) = \sum_{i_0} max_{D_1} \sum_{i_1} max_{D_2} \dots max_{D_n} \sum_{i_n} P(U_C|U_D)V \tag{4.9}$$

While influence diagrams could theoretically be solved using the formulas give above, which essentially unfolds the influence diagram out to a decision tree, there is a problem; namely the joint probability table $P(U_C|U_D)$. As already mentioned in the description of Bayesian networks, joint probability tables grow exponentially in the number of variables. However, there exist methods that allow us to work with much smaller domains. Describing these are beyond the scope of this report, but an interested reader can find more information in [Jen01].

4.3 Divorcing

When working with Bayesian networks and influence diagrams it is necessary to consider the size of the probability tables in the diagram. The larger the state space[1] of any node, the slower the network will be to work with as the probability table's size influences how many computations that is required when computing joint probability tables. Also, the larger the state space the more probabilities will have to be specified, which can be a problem. Consider the situation in Figure 4.7 on the next page. If the variables A, B, C and D have 10 states each, the probability table in variable E will have a total of $10^4 \cdot n$ entries where n is the number of states in variable E itself. If $n = 10$, the number of entries in E will have be $10^4 \cdot 10 = 10^5 = 100.000$.

[1]The state space of a node is the product of its number of states and that of each of its parents.

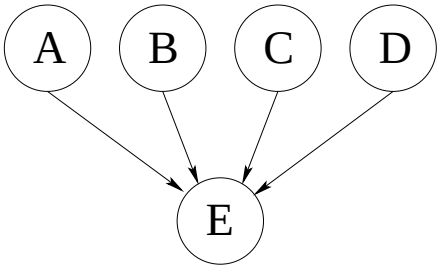


Figure 4.7: Network before divorcing.

Now imagine that E has an extra parent, also with 10 states, this causes the state space to grow to $10^6 = 1.000.000$ states. This huge increase in state space (actually the state space increases exponentially with the number of parents, assuming that each parent must have at least two states) can quickly make any network to computational slow for any use, especially in real-time domains.

A way of reducing the effect of this problem is shown in Figure 4.8. The idea is to introduce a "divorce" variable, thereby reducing the number of parent to the variable E. The idea is that the influence of some of the parent variables is summed up in a new variable. E.g. assume that E is the outcome of hitting on a girl Erica, and A, B, C and D are the length of my hair, my complexion, who wealthy I am and how intelligent I am, respectively. The two variables A and B could be collected into a new variable F (with, for example, 10 states) that representing my physical appearance. This will reduce the number of entries in E's table to 10^4 and, since the number of entries in the table for the newly introduced variable F only will be 10^3, this is a reduction of $10^5 - (10^4 + 10^3) = 89.000$ entries for the network. Of course this technique only works if the number of states in the node F is smaller than the product of the number of states in A and B.

Note that divorcing is not always followed by the loss of information

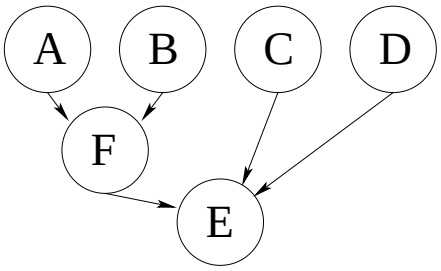


Figure 4.8: Network after divorcing.

normally associated with the reducing the number of states. If some states in two (or more) parent variables "overlap" they can be combined without loosing information. E.g. if E in Figure 4.7 represent whether my dog is happy (states **Yes** and **No**) and A and B is whether or not I give it a bone and a squeaky toy respectively (Also with states **Yes** and **No**). If I give the dog anything it is certain to be happy, so A and B can be summed in the variable F with states **Nothing** and **Something** without losing any information in the network.

4.4 Adaptation

When working with a Bayesian network there is often some (second-order) uncertainty associated the initially specified probabilities, and in many domains the probabilities are under constant change. In most cases the second-order uncertainty cannot be completely avoided. However, trying to adapt the probabilities in the network over time, in an attempt to increase their correctness and thereby decreasing the second-order uncertainty, can, to some degree, counter the problem. It can also be done to adapt to a dynamic environment. A way of performing adaptation is by so-called fractional updating, which is a statistical approach that helps automate the process of specifying the probabilities.

4.4.1 Fractional Updating

The idea with fractional updating is to modify the probabilities gradually when new cases arrive. To make this feasible, two simplifying assumptions must be made:

- Global independence: The second-order uncertainty for variables is independent of each other. This means that the conditional probabilities for the variables can be modified independently.
- Local independence: The uncertainties of the distributions for different parent configurations are 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_1, c_1) and (b_1, c_2); then the second-order uncertainty of $P(A|b_1, c_1)$ is independent of the second-order uncertainty of $P(A|b_1, c_2)$, this makes it possible to modify the two distributions independently.

Now consider $P(A|B, C)$ with the assumption of both global and local independence. The distribution $P(A|b_1, c_1) = (x_1, x_2, x_3)$ can now be viewed as being derived from previous cases, where (B, C) was in state (b_1, c_1). The initialization of a Bayesian network requires starting values for the probabilities, which are viewed as part of the past. The idea is to measure each

state's frequency in the previous cases for the distribution. The certainty of the distribution can be roughly measured by the amount of previous cases, called the sample size. A large sample size makes a probability more resistant to changes, i.e. a new case will have a smaller impact on a probability based on a large sample size than one with a small sample size, and therefore has a smaller second-order uncertainty. Each state in A has a count for the number of previous cases, the sample size is simply all the states' counts summed up. This leads to the counts (n_1, n_2, n_3) and a sample size s such that $s = n_1 + n_2 + n_3$ and

$$P(A|b_1, c_1) = \left(\frac{n_1}{s}, \frac{n_2}{s}, \frac{n_3}{s}\right)$$

For cases where all variables are observed, both s and the count for the state observed is incremented by 1 and the probabilities are calculated again. But not all new cases have evidence on the state of all variables. A new case with $P(b_1, c_1|e) = z$ will add z to s (for the cases where variables B and C is observed to be in state b_1 and c_1, respectively, z will be 1). As $s = n_1 + n_2 + n_3$ the new counts for n_1, n_2, n_3 is multiplied with z. If the new case does not give the state of A, then the current distribution of $P(A|b_1, c_1) = (y_1, y_2, y_3)$ is used. The probability distribution of $P(A|b_1, c_1)$ will therefore be updated by the following formula:

$$P(A|b_1, c_1) = \left(\frac{n_1 + zy_1}{s + z}, \frac{n_2 + zy_2}{s + z}, \frac{n_3 + zy_3}{s + z}\right)$$

This idea is called *fractional updating*. It has a large disadvantage in the instances where new cases only give information about the parent configuration of A but no information about the state of A, e.g. the evidence $e = \{B = b_1, C = c_1\}$ do not give any information about $P(A|b_1, c_1)$, but fractional will still increment s by 1 and use the current distribution as count, and thereby take it as a confirmation of the current distribution. The problem is that this sample size will increase and thereby give the impression of decreasing the second-order uncertainty and make the distribution more resistant to changes.

The following section describes a possible solution.

4.4.2 Fading

Fractional updating will remember all the old cases and they will have the same influence on the probability as the new cases. For each new case added to the network the sample size will be incremented by 1. The result is that for each case, added to the network, the next cases will have a smaller and smaller influence on the final distribution. If the Bayesian network is developed in a dynamic environment, the old counts will become a larger and larger dead weight for the model's ability to adapt to the environment.

The logically approach to remove this drawback is to limit the size of the sample size. An idea for a practical solution that do not require a record over "active" cases and their mutual order will be explained in the following through an example.

Let the variable A have three states and a sample size s, the three states have the corresponding counts (n_1, n_2, n_3) given a specific parent configuration for A. A new case with A in the state corresponding to the count n_1 has been gathered and is ready to be added. In normal fractional updating n_1 and s would be incremented by 1, but in order to prevent the sampling size to grew uncontrollable a fading factor $q \in [0, 1]$ is introduced, the fading factor is multiplied with the counts and the sampling size before the new case is added (the sampling size can also be calculated again). Each update will work this way:

$$s := qs + 1; n_1 := qn_1 + 1; n_2 := qn_2; n_3 := qn_3$$

If all new cases have the same weight, the past will, for each new case, be multiplied with the fading factor and therefore fade away with exponential speed. The maximum sample size is reached when

$$s = \frac{1}{1 - q}$$

Afterwards the sample size will be constant. The fading factor q, corresponding to a desired maximum sample size, can be calculated with the formula:

$$q = \frac{s - 1}{s}$$

The sample size determine how fast the distribution will adapt to new cases, i.e. the smaller sample size, the faster adapting. However, as with normal fractional updating, a larger sample size has the possibility to be more precise.

Example of Fractional Updating Imagine that an inexperienced player of the renowned Rock Paper Scissors game have constructed a simple Bayesian network to advise him. A part of the Bayesian network has two chance nodes; a node models the opponent player's sex and has two states (Male, Female), and a node that represent the hand of the opponent and has three states (Rock, Paper, Scissors). This part of the Bayesian network can be seen in Figure 4.9 on the next page and the table containing the players initial counts for $P(H|S)$ can be seen in Table 4.1 on the following page, the initial sample size is 10 for each distribution.

The player now want to train the Bayesian network with some collected cases, the cases however, are not equivalent as they consist of different information. In the examples below only the distribution for $P(H|S_M)$ is

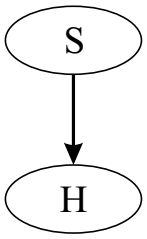


Figure 4.9: A Bayesian network for the Rock Paper Scissors game.

	S = Male	S = Female
H = Rock	5	3
H = Paper	3	5
H = Scissors	2	2

Table 4.1: Initial counts for $P(H|S)$.

calculated, the initial distribution is:

$$P(H|S_M) = \left(\frac{5}{10}, \frac{3}{10}, \frac{2}{10}\right)$$

The first new case that has been collected consists of hard evidence of all nodes ($S = Male, H = Rock$). This will update the distribution as follows:

$$P(H|S_M) = \left(\frac{5+1}{10+1}, \frac{3}{10+1}, \frac{2}{10+1}\right)$$

The sample size has increased from 10 to 11; this makes the distribution more resistant to changes and express that the certainty of the distribution has increased.

The second new case that has been collected has hard evidence on Sex, but only a distribution for the player's hand ($S = Male, P(H|S_M) = (0.2, 0.2, 0.6)$). This will update the distribution as follows:

$$P(H|S_M) = \left(\frac{6+0.2}{11+1}, \frac{3+0.2}{11+1}, \frac{2+0.6}{11+1}\right)$$

The sample size has increased from 11 to 12.

The third new case that has been collected is just the opposite, as it has a distribution for the sex of the opponent and hard evidence for the player's hand ($P(S) = (0.4, 0, 6), H = Scissor$). This updates the distribution to:

$$P(H|S_M) = \left(\frac{6.2}{12+0.4}, \frac{3.2}{12+0.4}, \frac{2.6+0.4}{12+0.4}\right)$$

The sample size has only increased from 12 to 12.4, as there was only 0.4 probability of $P(S_M)$.

The fourth new case that has been collected only has distributions for both chance nodes ($P(S_M) = 0.4, P(H|S_M) = (0.2, 0.2, 0.6)$). This updates the distribution as follows:

$$P(H|S_M) = \left(\frac{6.2 + 0.4 \cdot 0.2}{12.4 + 0.4}, \frac{3.2 + 0.4 \cdot 0.2}{12.4 + 0.4}, \frac{3 + 0.4 \cdot 0.6}{12.4 + 0.4}\right)$$

The sample size has only increased from 12.4 to 12.8.

The last new case that has been collected does not have any information on the player's hand, but does know his sex ($S = Male$). This updates the distribution as follows:

$$P(H|S_M) = \left(\frac{6.28 + \left(\frac{6.28}{12.8}\right)}{12.8 + 1}, \frac{3.28 + \left(\frac{3.28}{12.8}\right)}{12.8 + 1}, \frac{3.24 + \left(\frac{3.24}{12.8}\right)}{12.8 + 1}\right)$$

This example is the most problematic as the sample size has been increased from 12.8 to 13.8, which express that the certainty of the distribution has increased, without any new information. E.g. if 1000 new cases like this one should arrive, then the certainty of the distribution would be extremely high and it would therefore take a huge amount of "real" cases to correct an error. The easy solution is simply to avoid updating with these cases, but it is often not possible to get enough new cases, or any at all, with evidence on all variables. In these situations it can be useful to employ fading, as it put a ceiling on the sample size (unless the fading factor is set to 1). However, bear in mind that until the maximum sample size has been reached, the fading will only reduce the increase of the sample size. It is also worth remembering that a large sample size has the possibility of being more precise and thereby gives a higher certainty of the distribution, which is to some degree reduced by fading. Nevertheless the harm of having an undue certainty of a distribution can often justify fading, as can the existence of a dynamic domain.

5 Deliberative Layer

This chapter describes the implementation of the deliberative layer, which is capable of making rational, strategic decisions. In the chapter an influence diagram for a defending bot on the map "2Fort" is constructed. A description of "2Fort" can be found in Section 2.3.1 on page 13. As mentioned in Chapter 3, the Individual Planning Layer and the Mental Model has been combined into a single deliberative layer. The reason for this is that we use influence diagrams, which in addition to making decisions, is capable of storing knowledge about the world. The tool Hugin™[Hug04] will be used to model and interact with the influence diagram.

Before describing how the influence diagram is constructed and how the techniques presented in Chapter 4 is employed, let us briefly iterate the goal of a defending bot: Prevent the attacking team from taking the flag. To achieve this goal a bot has two strategic decisions to make; which area of the base to defend and which class to play. In theory it does not matter which of the decisions that are made first, but TFC requires the agents to choose a class before commencing play, so in our model this choice is always the first decision. As described in Appendix A on page 115, TFC has 9 different classes, but to reduce the complexity of the model, we confine our defending agents to only 3 of the 9 classes: Heavy Weapon Guy (HWGuy), Sniper and Engineer; which are the classes with the most defensive potential while still possessing different strength and weaknesses.

5.1 Assumptions about TFC

Before the actual defense model for making the strategic decisions are developed, assumptions about TFC, which will influence how the model is constructed, are discussed.

The map "2Fort" is a CTF scenario, so the ultimate goal of a defending agent, from a strategic point of view, is to prevent enemies from obtaining his team's flag. To do so, the defending agent must move to a location[1] from which he can stop oncoming enemies. However, it does not make sense

[1] In TFC a location is an (x, y, z) coordinate.

to consider each and every location in the world individually (there would be an almost infinite number of locations, whose relative worth would have to be learned), since it usually does not matter which location an agent occupies within the same general area. Instead of using individual locations we select sets of related locations with the same characteristics and group them together in *zones* that an agent can use when deciding where to defend. E.g. in "2Fort" the flag that must be defended is situated in a room in the basement of the base, but it does not matter much where in the room an agent stands; if an enemy reaches the room she will be forced into combat with the defender(s) regardless of their position. So instead of working with the actual map, an abstract map, which only consists of zones and how they are connected, is used. An example of an abstract map for a defending team can be seen in Figure 5.1 on the next page. The dotted circle ("Outside") is used to illustrate that an enemy agent can initiate her attack in one of two different ways - it is not a zone that can be defended. As can be seen in the figure an enemy could always start her attack in the "Entrance" zone without ever going through the "Battlement" zone, but the path from her start location (somewhere in the "Outside" zone) is far greater than if she had run through the battlement. So it is quite conceivable that enemies will indeed try to pass through the "Battlement zone". An abstraction of the map that divides it into logical zones, is best constructed either by an experienced player or on the basis of the different guides that are usually available on the Internet. The map in Figure 5.1 on the facing page is based on [tfc04c, tfc04d, tfc04b] and personal experience.

Another assumption about TFC (pertaining only to defending agents) is that death in itself is not a bad thing. In TFC an agent respawns instantly with full health and are usually within 20 (often only 10) seconds travel from the zone he wishes to defend (assuming that he only defends zones within the teams base, and that he uses a shortest path algorithm for navigation. Shortest path is used, as traveling within the bot's own base is relatively safe, and it is important to reach the chosen zone quickly.). But the fact that he has died means that an enemy has killed him, which in turn is troublesome as she must be that much closer to the flag.

In TFCs it is usually suicide for an agent, who as just encountered an enemy, to attempt to flee instead of fighting the opponent. Even though TFC is not a one-shot-one-kill[2] game, an agent can be killed within seconds. We therefore assume that the enemies never retreat. I.e. if an enemy has entered a zone she will never turn back and try another path.

The bot chooses a zone and a class when and only when it respawns. This is justified as changing class can only be done if an agent dies.[3]

[2]One-shot-one-kill games are games where an agent normally is killed by a single or two shots. Examples of such games are "Counter-Strike" and "Rainbow Six".

[3]If an agent that is alive wishes to change class it can commit voluntary suicide, but by doing so it incurs a 5 second penalty, where it is out of play.

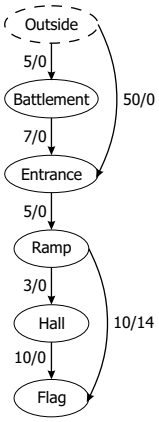


Figure 5.1: Abstract zone map of "2Fort" map. The dotted circle ("Outside") is used to illustrate that an enemy agent can initiate her attack in one of two different zones. The arrows show which zones that are connected and the corresponding number indicates the distance from the start of one zone to the start of the next. So from the start of the "Battlement" zone there are 7 units to the "Entrance" zone. The number after the / is the average environmental damage that is inflicted on an agent that uses this path (e.g. damage from falling a long distance).

To sum up, the assumptions about TFC are:

- Related locations can be merge into groups (called zones) to create an abstract map.
- In itself, it is not bad to die.
- An attacking enemy never retreats.
- A bot only chooses class and zone during respawn (after dying, but before spawning).

5.2 Developing the Defense Model

Losing the Flag First consider which utilities should be used in the diagram - since we have assumed that dying in itself is not bad, the only thing affects the decision is whether the enemy takes the flag. To reflect this we give utility -100 if the enemy takes the flag and utility 0 if she is stopped. As this is the only thing that gives utility it do not really matter what utilities are used as long as the utility for losing the flag is lower than the other. To represent this we let the utility node be a child of a node call "Takes Flag" with the two states **Yes** and **No**, where the first state means that an attacker has taken the flag and the second means that she was prevented from taking it. Figure 5.2 on the next page shows the diagram so far.

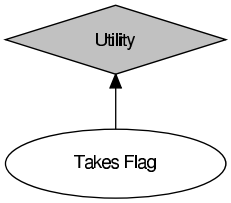


Figure 5.2: The "Takes Flag" node with the two states **Yes** and **No** determines the utility (-100 for **Yes** and 0 for **No**).

It is assumed, that when an enemy reaches the flag she takes it and tries to return it to her capture point. Should she be killed before reaching the capture point, causing her to drop the flag, it will either be picked up be another enemy, who will attempt to capture it, or it will return to the defender's basement (the "Flag" zone) after 60 seconds.

Reaching the Flag Whether the attacker reaches the flag, and thereby takes it, is dependent on whether she reached the "Flag" zone alive, this again depends on whether she reached the "Hall" Zone etc. However, this seems to lack the structure of the abstract map, since the probability of an enemy being alive before entering the "Flag" zone does not only depend on the previous zone, but also depends on the path she chose. If she has chosen to avoid the "Hall" zone by running around it, she is certain to be alive before the "Flag" zone, and if she was dead before, she will remain so (of course, this will never happen - if an attacker is dead she will not enter a new zone.). To account for this we introduce a chance node ("Attack Hall or Flag"), with the two states **Hall** and **Flag** that represent her choice of path (Table 5.1 on the facing page shows the probability for the "Before Flag Zone" node at this stage). For the same reason, we let the chance node "Attack Battlement or Entrance" represent the initial choice of coming through the "Battlement" zone or not. This can be seen in Figure 5.3 on the next page. Note that the structure of the diagram reflects the assumption that an attacker never retreats.

Understanding the semantic for the nodes in the network so far is fundamental for the further development of the network. Each node corresponds to a single attacker's status (**Alive**, **Dead**) at a certain point in an attack. Perhaps the easiest way to understand this is to look at Figure 5.4 on the facing page where the dotted line show the relation between the nodes in the network and the zones (represented by squares).

Stopping an Attacker Now let us consider what influences the status of an enemy (i.e. is she alive or dead) before a zone, i.e. what can cause her to die in the previous zone? It seems natural that the defending bot's

Before Hall Zone	Alive		Dead	
Attack Hall or Flag	Hall	Flag	Hall	Flag
Alive	0.5	1	0	0
Dead	0.5	0	1	1

Table 5.1: Example of the probability table for the ”Before Flag Zone” node. If the attacker was alive before entering the ”Hall” zone, but choose to bypass the ”Hall” zone, she is sure to be alive before the ”Flag” zone, likewise, if she was dead before ”Hall” zone, she certainly still is, and if she was dead before, she will remain so.

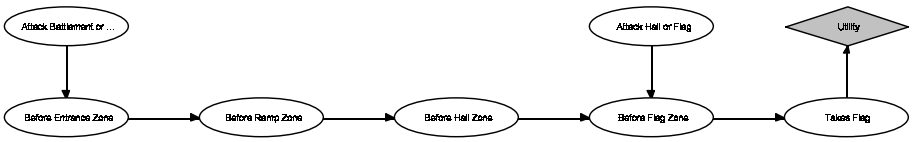


Figure 5.3: The defense diagram extended with nodes for all zones and two nodes that represent the enemy attacker’s choice of path.

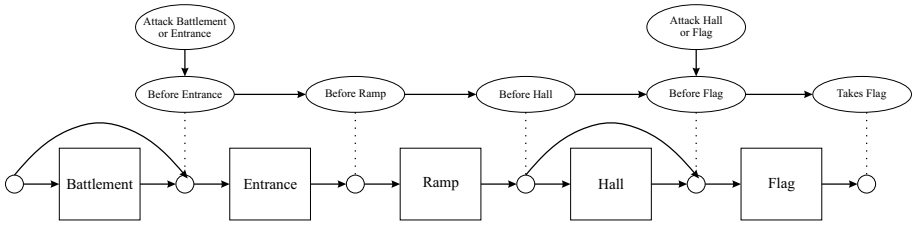


Figure 5.4: The relation between the diagram and the zones.

choice of both zone and class matters - if he has chosen to defend the zone the enemy is passing through, he has a chance of stopping her, which is also influenced by his class. Some zones will be more suited for some classes than others. E.g. the battlement is likely to be better suited for a sniper than an engineer. We try to model this by letting the "Before X Zone" and "Takes Flag" nodes[4] depend on the two decision nodes "Zone" and "Class", which are the defending bot's decisions of what zone to defend and what class to play. However, as a consequence the "Before X Zone" nodes would now require probabilities for the cases where the defending bot has a specific class but is not defending zone X. Intuition tells us that if the bot is not in a zone, his class does not influence the state of an enemy after the zone. So to avoid the unnecessary states we create a mediating node, called "Agent in X", for each zone X, with 3 states that correspond to the agent being in the zone and having a specific class (**Yes - HWGuy**, **Yes - Sniper**, **Yes - Engineer**) and the state **No**, i.e. the agent is not in this zone. Figure 5.5 on the next page shows the diagram extended with the choice of class and zone and Table 5.2 on the facing page show an example the probability table P(Before Flag Zone|Agent in Hall, Before Hall Zone, Attack Hall or Flag).

The Team The defending bot might not be the only agent on the defending team, so the status of an enemy should also depend on all other defending agents' zone and class decisions. However, this is not feasible. Each additional agent has 5 zones and 3 classes to choose between. This causes the state space of a "Before X Zone" node to become 15 times as large for each agent, which would significantly increase the time required to solve the influence diagram. Even if we use mediating variables to combine each agent into a single node, like the one previously introduced for the bot, each additional agent will still quadruplet the state space. Instead we would like the state space to be independent of the number of agents on defense. To archive this we introduce three new nodes for each "Before X Zone" that tells how many agents of each class that are defending the previous zone Y. These nodes are called "HWGuys in Y", "Snipers in Y" and "Engineers in Y" and has the three states **0**, **1** and **2+** meaning that there are none, one or two or more agents of this class defending the zone. The reason for not using e.g. three or more is that the state space starts to become too large to facilitate efficient learning, because of the increased number of agent configurations (The number of agents and their classes in a zone) that must be experienced. A section of the diagram showing only the "Before Hall Zone" node can be seen in Figure 5.6 on page 52. Note that in making this abstraction we lose information about the value of individual agent's (with the same class) in a zone, i.e. we essentially treat all agents as being equally skillful. This is not

[4] From now on whenever the "Before X Zone" nodes are mentioned, they implicitly includes the "Takes Flag" node.

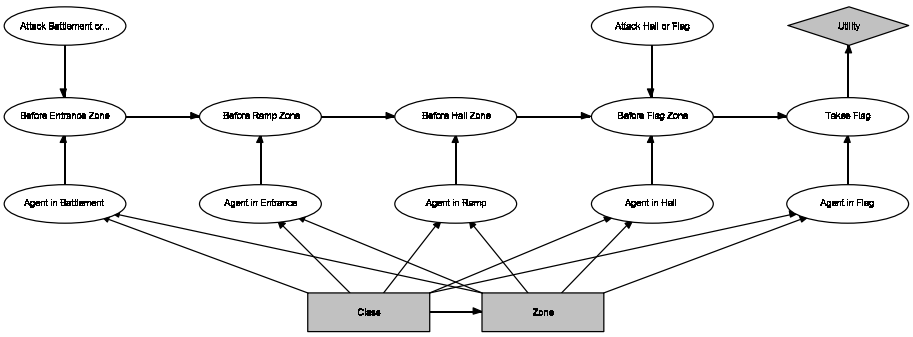


Figure 5.5: The defense diagram extended, so that the status of an enemy before a zone is dependent on whether or not the defending bot are trying to defend the previous zone and if so, what class he is playing.

Agent in Hall	Yes - HWGuy				...	No			
Before Hall Zone	Alive		Dead		...	Alive		Dead	
Attack Hall or Flag	Hall	Flag	Hall	Flag	...	Hall	Flag	Hall	Flag
Alive	0.5	1	0	0	...	1	1	0	0
Dead	0.5	0	1	1	...	0	0	1	1

Table 5.2: Example of the probability table for the "Before Flag Zone" node with its dependency on "Agent in Hall". The part of the table that has been omitted corresponds to the states **Yes - Sniper** and **Yes - Engineer**, which is identical to the "Yes - HWGuy" part of the table (of course, the probabilities might be different from 0.5.

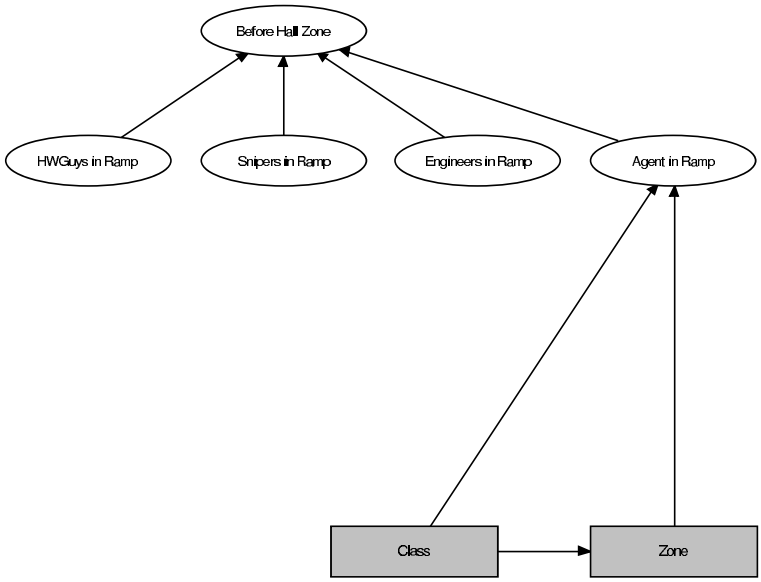


Figure 5.6: Part of the defense diagram, showing the "Before Hall Zone" node and its dependency on the number of agents of specific classes that are defending the "Ramp" zone, which lies before the "Hall" zone.

much of a problem with other bots that, at least given that they are using the same reasoning engines, are identical in skill level, but it does cause inaccuracy when dealing with human agents that inherently vary in skill. Also, we have lost the ability to differentiate between having more than two agents of the same class in the same zone, however in practice this should only constitute a minor problem as, in TFC games, there are rarely more than 6 agents dedicated to defense. Furthermore, the advantage of having more and more agents of the same class in a zone decreases with the number of agents.

Generalize the Agents If we look at the probability tables for the "Before *X* Zone" nodes, we notice that they require us to specify the probabilities for an enemy's status given that the defending bot is not in the zone. This is not something the bot can observe directly, so to avoid this we sacrifice information about the bot's specific worth and just count him among the other agents defending the zone. To do this we create another "HWGuy in *Y*" node for each zone *X*, so that we now have a node for how many of a specific class are defending a zone including the defending bot and one excluding him (These are prefixed with *Ex*). This is done for each of the three classes. An example of this can be seen in Figure 5.7 on the facing page. In addition this generalization results in a reduction of the state space.

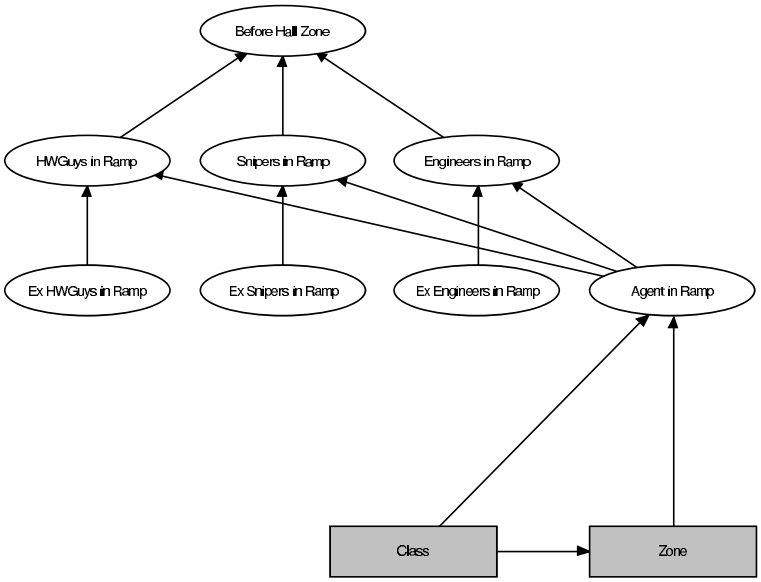


Figure 5.7: Part of the defense diagram, showing the "Before Hall Zone" node where the defending bot is included in the number of agents of specific classes that are defending the "Ramp" zone. The "*C* in *Y*" nodes do not contain any "real" probabilities, they simply add the bot to the other agents.

Estimating the Team When using the influence diagram just developed to make a decision for the agent's class and zone. The chance nodes for what class the other defending agents are playing and where they are located, e.g. "Ex HWGuys in Battlement", is intended to be given hard evidence (i.e. the node is to be instantiated). An example can be seen in Table 5.3.

Ex HWGuys in Battlement	probability
0	0
1	1
2+	0

Table 5.3: An example of the probability table for the "Ex HWGuys in Battlement" node.

However, the bot might not always know where all the other agents are located. This means that sometimes the evidence on the nodes will be the result of an estimation, and therefore only a guess of the states of the "Ex *C* in *Y*" nodes. To account for this, new "sensor" nodes (prefixed with "Guess"), that explicitly model the inaccuracy of the guesses, are introduced. The benefit of this is that when entering the evidence, i.e. the estimations, each guess can be associated with an uncertainty representing the bot's

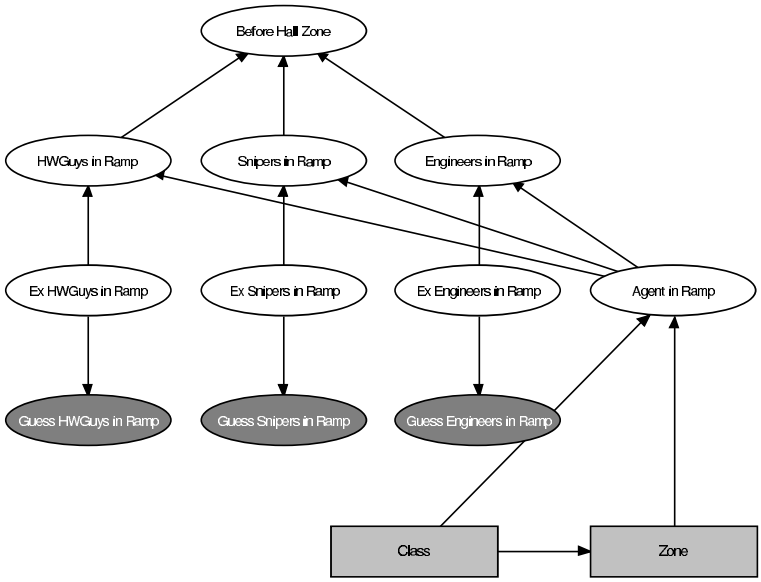


Figure 5.8: A section of the influence diagram showing the sensor nodes, which has been colored gray.

confidence in the estimation. A section of the influence diagram with the sensor nodes can be seen in Figure 5.8.

Note that there should be an information link between each of these sensor nodes and the "Class" decision node, however this would clutter the influence diagram. In fact, they are not needed in this specific diagram, as they are observed before the first decision and there are no observations to be made between the two decisions. Instead the sensor nodes have been given a darker color to indicate that they all receive hard evidence prior to the decisions. The exclusion of the links also constitutes a noticeable performance improvement.

Prior Knowledge Because of the way the defending agents have been generalized into nodes (i.e. there are no node for a specific agent), knowledge about one or more agents cannot simply by entered into the influence diagram. E.g. if the bot B knows for certain, that an agent A_1 is playing a HWGuy in the "Ramp" Zone, and has predicted that the agent A_2 is doing the same, B cannot enter the knowledge about A_1 into the diagram. If B enter it into the "Ex HWGuys in Ramp" node, it will cause the sensor node to become d-separate from the network, eliminating the prediction of A_2. Instead we introduce nodes (called "Known C in Y") that represent how many agents of a specific class B knows are in a zone and attach them as parent to the "C in Y" nodes. An example of this can be seen in Figure 5.9 on the next page. Note that these nodes are colored gray as they,

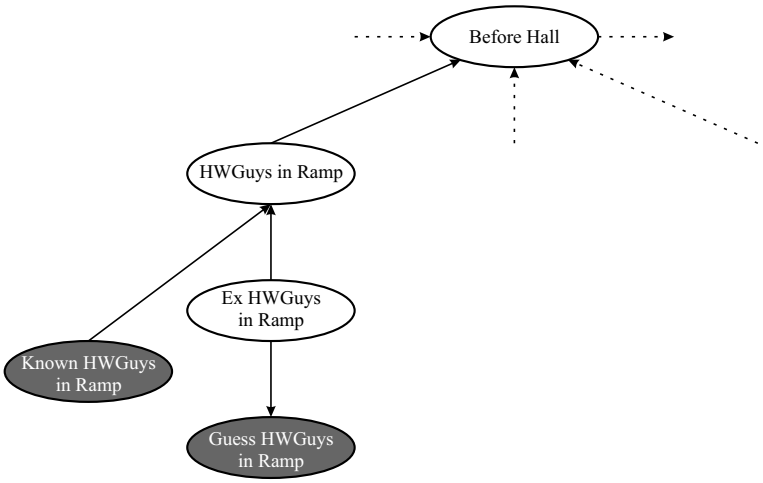


Figure 5.9: Example of knowledge node for HWGuys in the "Ramp" zone.

like the sensor nodes, are observed before the "Class" decision, but have had their information links removed to avoid cluttering the influence diagram, which can be seen in Figure 5.10 on the following page.

The Constraint Node The addition of the uncertainty of the estimation of other agents, introduced with the sensor nodes, create a problem with ensuring that the number of agents in the influence diagram corresponds to the actual size of the defense team in the game. E.g. if the bot estimates that there are two agents in some zone and the uncertainty for this estimate is very high, it would lead to a very low probability of two agents actually being in this zone. If this happens the two agents would be "missing" in the diagram, although they must be somewhere. To solve this, a constraint node is added to the diagram as a child of all the "Ex C in Y" nodes. This node must always receive hard evidence, specifying how many agents that have been predicted, and thus force the correct number of agents. Like the information links, this node clutters the diagram significantly without being very interesting and is therefore not drawn in the diagram show in this chapter. A complete diagram can be seen in Appendix B on page 121.

5.2.1 Disregarded Factors

There are other factors that influence an enemy's status before a zone, which are not modeled in the influence diagram. There are factors for which the bot cannot get a probability distribution and which are never observed, so he cannot even learn the probabilities. Examples of such factors are the average health and ammunition of the attackers. However, a change in one of these

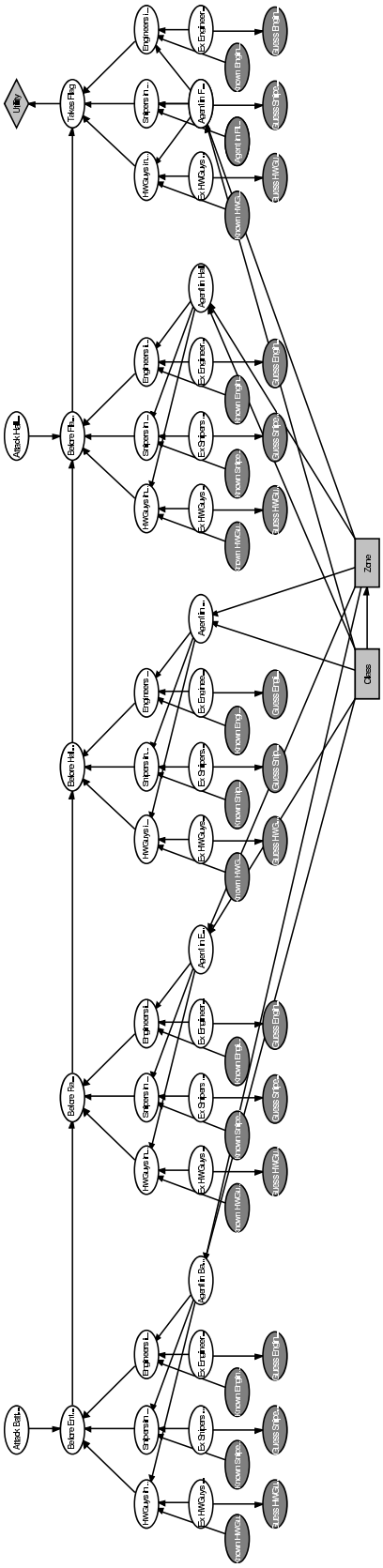


Figure 5.10: The influence diagram. Dark chance nodes (sensor nodes) are observed before the "Class" decision nodes. The information links have been excluded.

factors, e.g. the average health of the attackers, will change the distribution of the cases collected in the future (e.g. an increase in the average health of the attackers will likely increase the share of the cases, where an attacker got through the defenders), and thereby change the distributions in the "Before X Zone" as it is adapted using fractional updating.

Secondly there are factors that could be observed, like the class and weapons of the attackers. These have been excluded to reduce the state space and thereby the time required to solve the influence diagram, which has been based on the factors believed to be the most important from a cooperative point of view.

Number of Attackers At first sight the developed influence diagram only seems to apply to an attacking team with a single attacker. However, with the assumptions/facts that regardless of which attacker that takes the flag, it is equally bad for the defending team, the distributions in a "Before X Zone" can be viewed as an attack wave's probability to defeat the defenders in the zone. This is feasible, since only one of the attackers in group can actually take the flag. It is our estimate, that it is not important enough, from a cooperative point of view, to model the attack team's collaborated strategy to justified the increase in state space.

5.3 Updating the Diagram

The bot collects information through two kinds of cases: Sensor cases and combat cases, which are used to update the influence diagram through fractional updating[5]. Note that the "Agent in Y", the "C in Y" and the "Known C in Y nodes does not to be updated.

5.3.1 Sensor Cases

Sensor cases are used to update the uncertainty of the sensor nodes. A sensor case is generated each time the bot enters a zone and thus gains evidence on the configuration of agents in the zone. The update is done by entering the bot's estimation of the other agents into the fifteen sensor nodes and the observed agent configuration into the nodes corresponding to entered zone. This will result in an update of the uncertainty for all sensor nodes as well as the probability for all "Ex C in Y" node, which are d-connected given evidence on the constraint node.

Sensor cases do not give any information about the probabilities for the "Before X Zone" nodes, so to avoid updating these, we use a set of

[5]Fractional updating in Hugin is not done exactly as described in Section 4.4 on page 39. More information on this can be found in [OJ92].

mediating variables called "Disable Update", which can be seen in Appendix app:fulldiagram.

Example A bot has just respawned and has decided to defend the "Battlement" zone. On his way to the zone he passes through the "Hall" zone, where he observes that there are no HWGuys and snipers but two engineers. He enters this into the "Ex HWGuys in Hall", "Ex Snipers in Hall" and "Ex Engineers in Hall" nodes. He also enters his last estimation of the agent configuration for the five different zones into the sensor nodes, gives evidence on the constraint node and is now ready to make the update.

5.3.2 Combat Cases

There are three types of combat cases: The cases where an attacker is killed by the defenders in a zone, the cases where the bot is killed as the last defending agent in a zone and the cases where the bot is killed and there are more defenders left in the zone.

In the first cases, where an attacking agent is killed, the bot simply enter the known evidence into the diagram, i.e. the defenders in the zone, the status of the attacking agent before this zone (**Alive**) and her status before the next zone (**Dead**).

The next type of case, where the bot is the only defender in a zone and is killed, is virtually identical to the case just describes except that the attackers status before the next zone is **Alive**.

However, there is a problem with the third type of case. If more than one agent is defending a zone and an attacker kills the bot, he cannot know if the attacker got through the zone alive or was killed. Instead he would like to update the probability $P(BeforeXZone|parents)$ of an attacker being alive or dead, based on the attackers probability $P(BeforeXZone|parents')$ for being alive or dead given the remaining agents ($parents'$). This means that we would like to make a soft evidence update on the "Before X Zone" as described in Section 4.4.1 on page 39.

As mentioned, adaptation in Hugin is not exactly as detailed in section 4.4.1, so to get this effect we instead use $P(\text{Before X Zone}|parents')$ in conjunction with a randomly generated number r between one and zero are used to guess if the attacker succeeded in defeating the remaining agents. If $r \geq P(\text{Before X Zone} = \text{Alive}|parents')$, $P(\text{Before X Zone} = \text{Alive}|parents)$ is updated (as if the bot had received hard evidence on the "Before X Zone" node) otherwise $P(\text{Before X Zone} = \text{Dead}|parents)$ is updated. This should be justifiable as, given time and an appropriate effective sample size, this will cause the probability for the agent configuration to converge on the probability for the agent configuration "without" the bot.

Like the sensor cases do not say anything about the probabilities for the "Before X Zone", combat cases do not say anything about the certainty of the estimations, so again we use a set of mediating variable to avoid updating the nodes. These can also be seen in Appendix B on page 121.

Example Assume that:

$$P(B^R|H^E = 1, S^E = 0, E^E = 1, B^E = Alive) = \left(\frac{12}{20}, \frac{8}{20}\right) = (0.6, 0.4)$$

and that:

$$P(B^R|H^E = 1, S^E = 0, E^E = 0, B^E = Alive) = \left(\frac{16}{20}, \frac{4}{20}\right) = (0.8, 0.2)$$

where

B^R = "Before Ramp Zone"
B^E = "Before Entrance Zone"
H^E = "HWGuys in Entrance"
S^E = "Snipers in Entrance"
E^E = "Engineers in Entrance"

If a bot, playing as the engineer, dies in the ramp zone, he generates a random number $r \in [0, 1]$. If $r \geq 0.8$ he will update $P(B^R|H^E = 1, S^E = 0, E^E = 1, B^E = Alive)$ with the evidence $B^R = Alive$, using a fading factor of 0.975 (corresponding to an effective sample size of 40), the new probability will be:

$$P(B^R|H^E = 1, S^E = 0, E^E = 1, B^E = Alive) =$$
$$\left(\frac{0.975 \cdot 12 + 1}{0.975 \cdot 20 + 1}, \frac{0.975 \cdot 8}{0.975 \cdot 20 + 1}\right) = (\frac{12.7}{20.5}, \frac{7.8}{20.5}) \approx (0.62, 0.38)$$

If the bot experiences this situation repeatedly, $P(B^R|H^E = 1, S^E = 0, E^E = 1, B^E = Alive)$ will be updated with $B^R = Alive$ 80% of the time and $B^R = Dead$ 20% of the time, and eventually converge on $P(B^R|H^E = 1, S^E = 0, E^E = 0, B^E = Alive)$. This means that the bot do not help the team at all by standing in this zone, which will eventually cause him to choose another zone.

5.3.3 Unbalanced Cases

When updating the diagram there is an imbalance in the cases that are observed - a bot can only observe cases for the zone where he is. This means that whenever he observes a combat case, an attacking agent was alive before the zone, which will lead to updates of the probabilities for the "Before X *Zone*" nodes that precede the zone the bot occupies. This is as it should be, however the opposite is not true. If an enemy is killed in one of these

zones she will never reach the bot, which will be unaware of the death and cannot make an update. Essentially this causes the bot to overestimate the probability of an enemy getting through zone that lie before the one he is defending. E.g. if an attacker A is killed by bot B in the "Ramp" zone, it will increase B's probabilities for A getting through the preceding zones alive. If, on the other hand, that A are killed (by other defenders) in the zones preceding the "Ramp" zone B will never observe this, thus the probability for an attacker getting through zones preceding the one B occupies can only increase.

To avoid this imbalance we sacrifices the information that an attacker must have been alive before the bot's zone when updating. This is done by using the mediating variable to make sure the preceding nodes are not updated when the bot enters a combat case.

6 Prediction and Value of Communication

In this chapter different methods to support agent-centered cooperation are presented. Cooperation is essentially a matter of coordinating the actions of the involved agents. In order for agents to do this, they need information about the behavior of each other. One method of gaining this information (where discrete observation is not possible) is to attempt to predict what the other agents' are doing. Another technique is to try to communicate by sending and requesting information.

A general assumption in this chapter is that all agents behave rationally in relation to some model, i.e. choose the action that maximizes their individual expected utility. Another assumption is that an agent can have a model that approximates the behavior of another agent; these two assumptions are used repeatedly throughout the chapter.

The lack of information can sometimes cause friendly agents to unintentionally counteract each other. This occurs when two or more agents decide on actions that are individually optimal, but collectively suboptimal, e.g. they choose to perform a task that only needs to be done once thus failing to coordinate. This might happen if the agents use identical models and each take the decision that maximizes their own individual expected utility without considering other agents' action. For example, two taxi customers, at different locations, each order a cab at the same time. There are two cabs at the central who both hears the dispatch and each wants to pick up the nearest customer, as this is more profitable. If they only consider themselves when making their decision, both will try to pickup the nearest customer. But only one of them can actually pickup the customer, the other has just driven in vain and could have received a higher utility if he had gone for the second customer.

6.1 Prediction in Multi-Agent Environment

A bot B's ability to interact with a set of other agents $\mathcal{A} = \{A_1, \ldots, A_n\}$ is a complex and important problem. In order to make decisions that take agent A_i's actions into account (this could also be done to obstruct A_i or simply to avoid unintentional conflicts, instead of cooperating with it.), B can try to predict A_i's actions. A way to do this, is for B to put itself in A's position and predict what A would do in the given situation and then act accordingly. To do this B can use a model of A_i's reasoning to predict A_i's behavior. The precision of a prediction is dependent on the correctness of the model used as well as the available information. Ideally B has an exact copy of A_i's world model and can use this to predict A_i's actions. However, this is normally not possible and B must therefore use an approximation of A_i's decision process. Moreover, even if B knows what model is used by A_i, it does not necessarily follow that B can use it to predict A_i's behavior as A_i's decision process may be based on information that is available to A_i, but not to B. E.g. the model might require evidence about A_i's own location, which A_i properly knows, but B might not.

Given a model of A_i's decision process (or an approximation), B can predict the actions of all other agents and subsequently take these predictions into account when making a decision by using the predicted actions as evidence in B's own model. A description of one approach to implementing this procedure can be seen in the algorithm in Figure 6.1 on the next page. The method described is assumed to be invoked initially by a bot B with its own model M.

There is a problem with the code in Figure 6.1 on the facing page - it has an infinite recursion embedded in line a.1 where it calls itself recursively. This leads to a problem when trying to predict the actions of agents in $\mathcal{A}$: What level of prediction should B consider? I.e. when predicting A_i's actions should B take into account that A_i might be trying to predict B's own actions. However, if B base his prediction on what he predicts A_i has predicted about him, A_i might be doing the same, creating the need for B to base his prediction on this and so on. E.g. when playing checkers my next move is based on what I think my opponents next move is going to be, but his next move is based on what he thinks I am going to do in my subsequent move and so on. Thus, if I do not want to (or more likely cannot) predict the entire game from my current position, I have to base one of my predictions of my opponents move on a heuristic. That is, to avoid infinite reasoning chains like this, B needs to choose how many "levels of recursion" are feasible to his current need. When this is reached B must use a simplified model of A_i that do not use prediction. This is illustrated by the algorithm in Figure 6.2 on the next page.

There is a notable weakness with this technique: If the model and information B uses in his prediction for some agents $\mathcal{A}' \subseteq \mathcal{A}$ is the same, B

Algorithm: Let E be the set of initial evidence and let $\mathcal{A} = (A_1, \ldots, A_n)$ be a set of agents for which the bot B lacks information. In order to calculate a strategy for B with model M do:

a. For $i = 1$ to n
   1. Compute a strategy S for agent A_i by calling the algorithm recursively with B's model for A_i
   2. Add evidence corresponding to S to E
b. Let S be the strategy determined by M given E
c. Return S

Figure 6.1: First draft of a prediction algorithm. All strategies computed are optimal.

Algorithm: Let E be the set of initial evidence and let $\mathcal{A} = (A_1, \ldots, A_n)$ be a set of agents for which the bot B lacks information. In order to calculate a strategy for B with model M do:

a. For $i = 1$ to n
   1. If further recursion is wanted
      Compute a strategy S for agent A_i by calling the algorithm recursively with B's model for A_i
   2. Else
      Compute a strategy S for agent A_i by running a simplified model of A_i
   3. Add evidence corresponding to S to E
b. Let S be the strategy determined by M given E
c. Return S

Figure 6.2: Algorithm for prediction using a cap for the level of recursion.

Algorithm: Let E be the initial set of evidence, let E' be an empty set and let $\mathcal{A} = (A_1, \ldots, A_n)$ be an ordered set of all agents including the bot B. In order to calculate a strategy for B with model M do:

a. For $i = 1$ to n
 1. If the strategy for A_i is known
  Add evidence corresponding to S to E'
 2. Else
  2a. If further recursion is wanted
   Compute a strategy S for agent A_i by calling
   the algorithm recursively with B's model for A_i
  2b. Else
   Compute a strategy S for agent A_i by running
   a simplified model of A_i given E'
  2c. Add evidence corresponding to S to E'
b. Remove predicted evidence for the agents B had initial evidence about, including itself
c. Let $E = E \cup E'$
d. Let S be the strategy determined by M given E
e. Return S

Figure 6.3: Algorithm for prediction with ordering of agents. Notice that the bot B needs three kinds of models: His own, models for all agents $\mathcal{A}$, and simplified models for all agents in $\mathcal{A}$.

will predict the same action for all agents in $\mathcal{A}'$. This can be remedied by introducing a global ordering of the agents. When running the model for the first agent in the ordering (A_1), the simplified model is run just as before. But when running the simplified model for the second agent A_2 we assume that it has made the same prediction about A_1 as B just did. Likewise, A_3 has predicted A_1 and A_2 and so on.

As a consequence of the global ordering, B must also predict the actions of agents he initial had evidence about (including himself) and insert the predicted strategies into the set of evidence when iterating through the set of agents, as the prediction of the next agent's strategy depend on all preceding agent's predicted strategies. However, instead of making predictions of himself and these agents, B simply uses the knowledge he has as predictions of the remaining agents. Of course, when running his own model he will use his knowledge as real knowledge, not as predictions (this happens in line b, where these predictions are removed from E' and line c, where E' is added to the initial evidence E). The prediction algorithm utilizing the ordering of the agents can be seen in Figure 6.3. The technique for ordering just described is inspired by the design conventions and social laws in [Bou96] that expresses the need for an ordering among coordinated agents in a multi-agent decision process.

Example of Prediction The girl Erica has finally agreed to go bowling with me. Unfortunately I have forgotten in which of the town's two bowling halls we are going to meet and I now have to decide where to go. The influence diagram for this decision can be seen in Figure 6.4 on the next page. The node "Bowling Hall" represent my choice of the towns two bowling halls; "Red Pins" and "Balls!". The node "Erica" represents her choice of bowling hall. The probability distributions P(Erica) = (0.5, 0.5). Actually I do not like to bowl, but I do like Erica, so if I choose the same bowling hall as her I get utility 100, otherwise -100. If I simply solve my influence diagram, without using any prediction, I will get:

$$EU(RedPins) = \sum_{Erica} P(Erica)P(Prediction|Erica) \cdot Utility = 0$$

and likewise EU(Balls!) = 0. I.e. both decisions are equally good.

To use the prediction technique, I first modify my influence diagram to include a prediction of "Erica". This node is called "Prediction" and can be seen in Figure 6.5 on the following page. This node is used to assign an uncertainty to the prediction and is only needed if the probability of my prediction being correct is less the 1. Note that the worst probability I can have, given its parent, is that there are an equal probability for predicting any state. The probability table P(Erica) and P(Prediction|Erica) can be seen in Table 6.1.

Erica	
Red Pins	0.5
Balls!	0.5

Prediction		
Erica	Red Pin	Balls!
Red Pins	0.75	0.25
Balls!	0.25	0.75

Table 6.1: Probability tables for the bowling problem.

In this example Erica thinks that we have agreed on a place to meet, therefore it does not make sense to use a model for her, where she makes predictions of me. Therefore, I use a recursion level of 0, which means that I only need a simplified model of Erica. If, on the other hand, we had forgotten to decide where to meet, I would need both the simplified model and a normal model, as she would then try to predict where I would go. The reason why we do not give such an example is that multiple recursion levels should mostly be used for simpler, board-type games, where it is possible to get more accurate predictions than in, for example, FPSs.

My simplified model of Erica can be seen in Figure 6.6 on the following page. I believe that Erica also gets utility 100 if she chose that we should meet at "Red Pins" but only a utility of 80 if we meet at the cheaper and less fancy "Balls!".

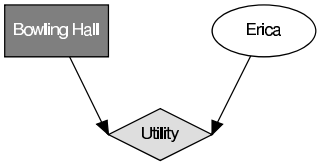


Figure 6.4: The influence diagram for the bowling problem.

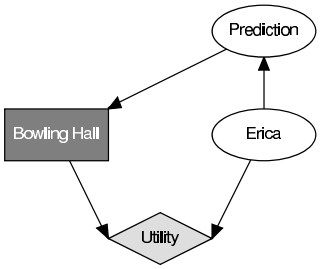


Figure 6.5: Influence diagram for the bowling problem, with the uncertainty of the prediction explicitly modeled.

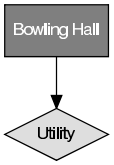


Figure 6.6: Simplified model of Erica.

To use prediction in the decision process, I first predict what Erica will do by solving my simplified model of her. By doing this, I get the result that she has chosen that we should meet at "Red Pins", which I enter into the "Prediction" node in my influence diagram. I now solve my own influence diagram, which says that I should go to "Red Pins" as this has an EU of:

$$EU(RedPins) = \sum_{Erica} P(Erica)P(Prediction|Erica, e) \cdot Utility = 50$$

where

e = the prediction

as opposed to "Balls!" who have an EU of -50.

6.1.1 Ordering of Agents

The prediction technique presented above requires that the set of agents are ordered, but determining the ordering among independent agents in a dynamic multi-agent environment may not be a trivial matter. In some dynamic multi-agent environments agents can disappear (voluntarily or due to crashes) or join at any time. This creates the need for a robust method to agree on the ordering.

The straightforward approach is to have one of the agents act as a centralized master, which periodically broadcast the order of the agents and their unique numbers. If an agent does not receive the list a certain number of times in a row, it starts the process of electing a new master.

There are several algorithms for electing the centralized master [CDK01]. One of the most utilized election algorithm is the "bully algorithm" that is resistant to disappearing agents. The algorithm has two requirements. First it requires a reliable data transfer protocol [KR03]. Secondly it requires that the processes must be synchronized with the master, as each process relies on timeouts to determine if the master has stopped its periodic broadcasts and therefore needs to start the election algorithm. The synchronization requirement is satisfied by the periodical broadcasts by the centralized master.

The bully algorithm requires that each process have a unique identification number (ID). The idea is that the running process with the highest ID is elected as the master. The "bully" comes from the fact that even though a fully functional process is running as master, it will be "bullied" by a new process with a higher ID.

The fundamental principle is that each process only communicates with processes with a higher ID. If a process has been notified that an election is going on or the process itself has detected that the master has stopped broadcasting, it will send an election message to all processes with a higher ID. Each process will respond with an answer message, meaning that it has assumed control of the election. If the process does not know any processes

with a higher ID, or the processes fail to answer, it will take the role of being the master, starting by broadcasting that it has won election and subsequently beginning to periodically broadcast the order of the agents, which the master sovereignly determines, and their unique addresses. These messages are called coordinator messages. The number of messages required by the algorithm to elect a new master depends on the number of processes that have a higher ID than the process, which detected the crash. The number of message is N^2, where N is the number of process' with a higher ID. The worst-case scenario is when the process with the lowest ID detects a timeout. This requires $(P-1)^2$ messages, where P is the total number of processes. The "bully algorithm" was originally presented in the context of computer networks.

Example of the Bully Algorithm The four agents in Figure 6.7 each have unique ID, where A_i has a lower ID than A_{i+1}.

Stage 1: The coordinator A_4 has crashed, A_1 detects this and starts an election. A_2 and A_3 receives an *election* message from A_1 and respond with an *answer* message.

Stage 2: A_2 sends *election* messages to A_3 and A_4 and A_3 sends to A_4. A_3 then sends an *answer* message to A_2. A_1 and A_2 has now received at least one *answer* message and therefore wait for a *coordinator* message. A_3 is waiting for a message from A_4 and is therefore the only agent with an

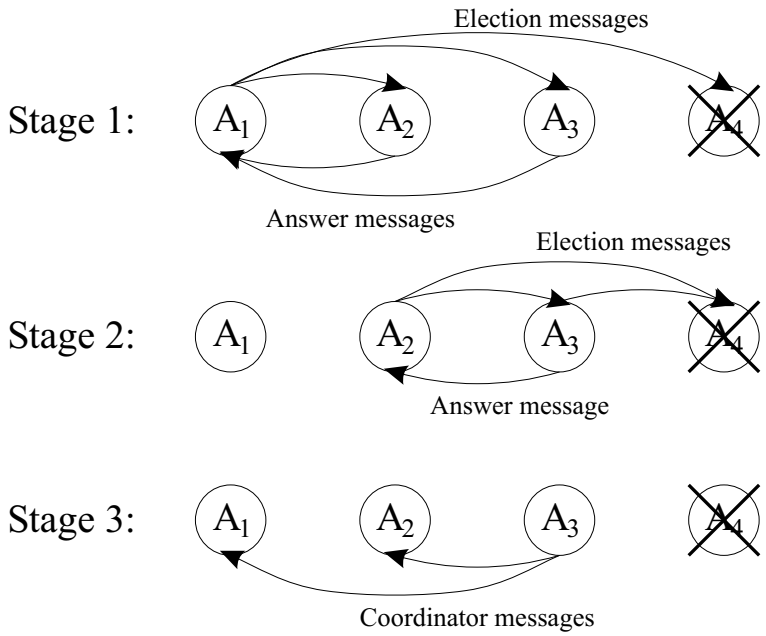


Figure 6.7: An example of the bully algorithm's election process.

active election.

Stage 3: A_3 has through a timeout determined that A_4 is crashed, therefore it elects itself as coordinator and sends a *coordinator* message to A_1 and A_2.

6.2 Value of Communication

The second way to support coordination, and thus cooperation, is through communication among the agents. As mentioned in Chapter 2 on page 9 there are often a cost associated with communication, and because of this the agents need to consider if sending a specific message is worth the associated cost.

Before discussing methods to determine whether or not a message should be send, we consider what kind of messages that are possible:

- **Requests message:** This is a message sent to another agent requesting that it send information back to the sender S. A request do not in itself give S any information, but hopefully the reply will. Therefore, the worth of this type of message must depend on whether the receiver answers and what the answer is.

- **Information messages:** These are messages, containing information about S, that are send to other agents. Sending such a message gives S the knowledge that the recipients know the information contained in the message. Thus sending such a message can change what S predicts the recipients are going to do.

Next the method for calculating whether an information message should be sent is presented. But first the value of information (VoI) technique, on which the method is inspired, is presented. The description is based on [Jen01] where a more detailed explanation can be found.

6.2.1 Value of Information

Observations often have a cost associated with them, and even though an observation adds valuable information to the decision process it might not be worth to make it. Thus, it is interesting to measure the value of the information given by the observation. One way of doing this, assuming that it is possible to make a model of the situation the observation pertains to, is to calculate the Maximum Expected Utility (MEU) with and without the decision of performing the observation. If the gain in MEU is greater than the cost of the observation, the observation should be made.

If there is more than one optional observation, and it is possible to make any number of observations in any sequence. The MEU for all combinations of observations and decisions must be calculated and subtracted the

cost of the performed observations to find the optimal strategy for selecting the observations. As the number of possible combinations of observations is $\sum_{i=1}^{n} \binom{n}{i}$, for n number of optional observations, this quickly becomes impractical.

To avoid this a myopic solution is often used. The myopic solution is a greedy approach: It calculates the MEU of all decisions of performing an observation and then selects the decision with the highest MEU subtracted the cost.

If there are any remaining optional observations the myopic approach can used again, this time based on the new situation where the first observation has been performed. This can be repeated until there are no more observations or none of the remaining observations increases the MEU subtracted their cost. It is important to remember that the myopic approach is not guaranteed to give the optimal sequence of observations, e.g. in the cases where no observation by itself increase the MEU subtracted the cost, but in connection with another observation(s) do increase the MEU subtracted the cost, the myopic solution fails.

6.2.2 Request Messages

In the FPS domain there are rarely optional observations to be made, there are, however, the choice of whether or not to send messages to other agents. But this is a decision to send something, which is not the same as deciding to make an observation. As mentioned an observation always changes a bot's belief about the world because the bot is certain to receive evidence, but this is not necessarily the case when sending requests for information, as the asked agent might not answer. Still, the general principle from value of information might be applicable.

When deciding whether to send a request, B is interested in whether the asked agent will respond and if the respond will result in an increases in the bot's MEU that is higher than the cost (note that this is related to the myopic VoI approach). The value v (increase in expected utility) for sending the request is calculated as the probability for receiving an answer times the sum over the probability for receiving a specific answer multiplied with the difference in MEU. This is done with the formula:

$$v = P(E|e) \cdot \sum_{i=1}^{n} \Big(P(V = v_i|E, e) \cdot \Big(MEU(B|V = v_i, e) - MEU(B|e) \Big) \Big) - \gamma \tag{6.1}$$

where

e is the initial evidence
γ is the cost of sending M
E is the event that A answers

V = the variable in B's model for which information has been requested

The probability $P(E)$ of whether or not A answers, can be simply be based on the frequency of A's previous answers. Another option is to base $P(E)$ on a prediction, where B tries to determine if A will gain an increase in MEU (adjusted for the cost of replying) by replying to B's request, thus making it more probable that A is going to answer. Note that the probability of V being in state v_i ($P(V = v_i|E)$) is used as B's belief about the probability of receiving evidence $e = (V = v_i|E)$.

B can now determine if requesting information will be sensible. If the value v is higher than zero, the request should be send to the agent A.

Note that Equation 6.1 do not take into account that the recipient of the request might have to pay a cost γ' to send an answer. If this is to be taken into consideration, γ' must be subtracted from the summation before this is multiplied with $P(E)$, as the cost is only incurred if the recipient of the request actually answers.

Example of Request Message I am not convinced that my prediction of Erica is correct, so I am considering sending her an SMS asking where we should meet. Doing so however, is going to be pretty embarrassing, causing me to get a -20 utility penalty (the cost of sending the request). To use Equation 6.1, I need P(Erica answering) which I believe is 0.85 (she might not check her phone). I also need the probability for the two different possible answers given that she does answer. For this I use P(Erica).

No matter what she answers (the answer is used as evidence on the node "Erica", which will cause the "Prediction" node to become d-separated from the rest of the network) I have a MEU of 100 and if she does not answer I have a MEU of 50, determined by using the prediction method described in Section 6.1. Now the value v for sending an SMS to Erica can be computed:

$$0.85 \cdot \Big(0.5 \cdot (100 - 50) + 0.5 \cdot (100 - 50)\Big) - 20 = 22.5$$

I conclude that I should call Erica, even though it is embarrassing to admit that I have forgotten where to meet her, as it increases my expected utility by 22.5, in relation to using prediction.

6.2.3 Information Messages

The other type of message a bot B can send is an information message. Whether such a message should be send depends on whether the EU of B, as well as the EU of the agents receiving it, are higher than if the message was not send. I.e. a bot can calculate the value of sending information to another agent by comparing the bot and agents weighted average EU (WAEU) for sending the information with the WAEU for not sending it.

Intuitively the WAEU for a group of agents can be thought of as the average MEU for an agent in the group weighted by its importance. Technically the WAEU for a bot B and a set of other agents $\mathcal{A} = \{A_1 \dots A_n\}$ is a weighted average of B's MEU and the MEU of each agent in $\mathcal{A}$. E.g. the WAEU of three evenly weighted agents, with the MEU of 10, 20 and 30 respectively, is $\frac{1}{3}10 + \frac{1}{3}20 + \frac{1}{3}30 = 20$, but if the first agent were considered to be twice as important as each of the other agents the WAEU would be $\frac{2}{4}10 + \frac{1}{4}20 + \frac{1}{4}30 = 17.5$. The reason for allowing different weights to be used is that, while the WAEU's is used to determine increases in EU there might be some agents that are more important to help than others, e.g. B might be more concerned with its own utility than that of others.

The Weights The specific weights $\{w, w_1, \dots, w_n\}$, where w is the weight for B and w_i is the weight for A_i, could simply be specified by a designer as long as the weights always summarize to 1, however this scheme can make it hard to get the desired ratio between the bot B and the other agents. E.g. if B is to be twice as important as A_1 and A_2 the weight should be $\{\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\}$, but what if A_1 is to twice as important as A_2, while still keeping B twice as important as both? This would require the weights $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\}$. An easier way to control the importance of agents is to specify two things: First the level of altruism that is desired from B, which should be specified as a factor $\alpha \in [0, 1]$. α describes how important B's own utility is in relation to that of the agents in $\mathcal{A}$, where an altruism factor of 0 means that B is completely egocentric and 1 means that it is completely altruistic. An altruism of $\frac{1}{2}$ means that B's MEU are equally important to the average weight of the agents, i.e. B's weight should be $\frac{1}{|\mathcal{A} \cup B|}$. Secondly each agent A_i should simply be associated with a relative importance β_i specifying how important B believe this agent is compared to other agents. E.g. this makes it possible for B to account for another agents' skill by giving higher importance to capable agents. The weight w for B can be computed as follows:

$$w = \frac{(1 - \alpha)}{(1 - \alpha) + (\alpha \cdot n)} \tag{6.2}$$

where

$$n = |\mathcal{A}|$$

The weight w_i for each agent A_i can then be computed as follows:

$$w_i = \frac{\beta_i}{\sum_{j=1}^{n} \beta_j}(1 - w) \tag{6.3}$$

The graphs in Figure 6.8 on the next page shows different weights, based on the agent B's level of altruism, in a four agent scenario where the three agents (A_1, A_2 and A_3) have importance 1, 2 and 4 respectively (i.e. $\beta_1 = 1, \beta_2 = 2$ and $\beta_3 = 4$). Note that $(w + \sum_{i=1}^{n} w_i) = 1$.

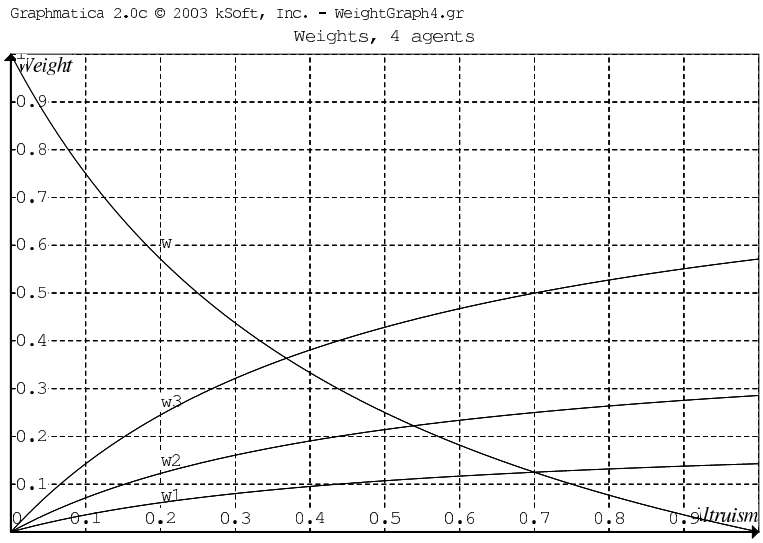


Figure 6.8: Weights in a four agent scenario. The x-axis is B's altruism factor and the y-axis is weight. The graph w correspond to the weight of agent B, the graphs w1 is the weight for agent A_1 and so on.

The Calculations The WAEU of a set of agents $\mathcal{A} = \{A_1 \dots A_n\}$ can now be calculated with the equation:

$$WAEU(B \cup \mathcal{A}) = w \cdot MEU(B|e_B) + \sum_{i=1}^{n} w_i \cdot MEU(A_i|e_{A_i}) \qquad (6.4)$$

where

e_B is B's initial evidence
$e_{A_i} \subset e_B$ is the evidence B knows A_i has
$MEU(A_i|e_{A_i})$ is B's estimation of the MEU for the agent A_i.

B also needs to compute the WAEU for a set of agents given that the message $M = e'$ is sent, which is done with the equation:

$$WAEU(B \cup \mathcal{A}|M) = w \cdot MEU(B|e_B, e') + \sum_{i=1}^{n} w_i \cdot MEU(A_i|e_{A_i}, e) \quad (6.5)$$

where

$e \in e_B$ is the information B has sent to the agents
e' is the information that the agents in $\mathcal{A}$ know e

The reason for calculating the new MEU for B, is that sending the message M generates the evidence $e'' =$ The agents in $\mathcal{A}$ knows e'. This may change B's prediction of what the agents receiving M are going to do. E.g. if B sends a message to A_i containing B's current position, it can change B's prediction of A_i's future actions because B now knows, that A_i knows B's position.

Now B can determine if it should send a message $M = e'$ to the set of agents $\mathcal{A} = \{A_1, \ldots, A_n\}$ by calculating:

$$WAEU(B \cup \mathcal{A}) - WAEU(B \cup \mathcal{A}|M) - \gamma$$

This gives the weighted average increase in expected utility for the agents in $\mathcal{A} \cup B$, and B can now determine if sending a message M will result in an increase in WAEU that is higher than the cost γ for sending the message M. Note that this assumes that M can be broadcasted to the agents, if this is not the case, γ must be multiplied by $n = |\mathcal{A}|$.

Example of Information Message Instead of sending a request to Erica and admit that I have forgotten where to meet her, I am now contemplating to just send her an SMS with the text "See you tonight at Red Pins". The cost of this message is only 10, as, even if it might be wrong, she would just think that I am changing our agreement.

To use the technique just described, to determine if I should send this message, I need a model of Erica. This can be seen in Figure 6.9 on the facing page. Her utility for meeting me at the two bowling halls are as before, but if I do not show up (i.e. I have chosen the wrong place) she will get mad, especially if she has gone to "Red Pins" where it is more expensive to get in. The utilities can be seen in Table 6.2 on the next page. I also need to determine what weights to use and, since I am a nice guy, I give Erica a weight of 0.7 and myself 0.3.

First I compute:

$$WAEU(Me, Erica) =$$

$$0.3 * MEU(Me|Prediction) + 0.7 * MEU(Erica) =$$

$$0.3 * 50 + 0.7 * 15 = 25.5$$

Next I compute:

$$WAEU(Me, Erica|SMS) =$$

$$0.3 * MEU(Me|Prediction, EricaknowSMS) + 0.7 * MEU(Erica, SMS) =$$

$$0.3 * 50 + 0.7 * 100 = 85$$

I subtract the first from the second and get $85 - 25.5 = 54.5$, which is higher than the cost of 10 for sending the SMS, so I conclude that sending the SMS is a good idea.

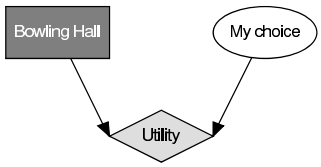


Figure 6.9: Model of Erica.

Bowling hall	Red Pins		Balls!	
My choice	Red Pins	Balls!	Red Pins	Balls!
Utility	100	-100	-50	80

Table 6.2: Table for the "Utility" node from the influence diagram in Figure 6.9.

7 Cooperative Layer

The cooperative layer's purpose is to enable the deliberative layer to makes decisions that coordinate the bot's actions with other agents' behavior. The cooperative layer uses the two techniques described in Chapter 6 on page 61 (Prediction in Multi-Agent Environment and Value of Communication) to get information about the other agents' behavior. The prediction technique is the primary technique, as it is certain to produce a result based on the influence diagram and the world model. "Value of Communication" is used as a secondary technique as it is not guarantied to return useful information, since it depends on communication between the agents.

The cooperative layer receives control from the deliberative layer in two cases: When the bot is spawning and need to choose a class to play and a zone to defend, and when another agent asks the bot about his location. In the first case the cooperative layer uses predictions and the VoC method for request messages. In the second case it uses predictions and the VoC method for information messages.

7.1 Spawning

At the beginning of the game and whenever the bot is killed, he must choose which class he will respawn as (restricted to the three classes choose in Chapter 5 on page 45) and subsequently which zone he will defend. To do so he first makes a prediction of all the agents for which he lacks information and secondly he considers, for each agent individually, whether the agent should be asked about its location.

7.1.1 Prediction

The prediction technique is use to generate evidence for the sensor nodes introduced in Figure 5.8 on page 54; they are depicted as dark chance nodes that are observed before making the decisions in the influence diagram seen in Figure 5.10 on page 56. The problem with this diagram is that a bot can only observe the other agents' class, but not where they are located on the map. To predict in which zones the other agents are located, the bot uses an influence diagram identical to his own as his simplified model for the other

agents (used in line 2b in the algorithm in Figure 6.3 on page 64). The bot exist in the same environment and with the same objectives as the agents he is trying to predict; it is therefore reasonable to assume that the bot's influence diagram will be a good estimation for the other agents' decision process.

The prediction technique requires that the set of agents is ordered. The ordering used for the agent's in the test environment is dictated by the sequence in which they join the server (in the TFC environment even bots that are running on the same host as the server, must logon to the server). As the server only accepts one logon at a time, it means that the ordering is well defined. A more general method for creating an ordering of agents can be seen in Section 6.1 on page 62.

Example of Prediction Assume that five agents with the ordering $\{A_1, A_2, A_3, A_4, A_5\}$ are defending the base. The engineer A_3 is killed in the "Flag" zone and is about to decide what to do now. This is a decision that should support the other agents on the team, so control is passed from the deliberative layer to the cooperative. A_3 already knows the classes of the agents, which he gets from the scoreboard, and just before he died he saw that A_4 were defending the "Flag" zone. A_3 now needs to predict which zones the other two agent are defending, so he can compute his optimal strategy.

First agent A_3 takes an influence diagram ($\mathcal{M}$) identical to his own, which is used as his approximation of the other agents. A_3 then gives evidence on all the "guess" nodes (the gray nodes in Figure 5.10), guessing that there are no agents anywhere, sets the constraint node to state **0** and enters A_1 class into the "Class" decision node. The zone corresponding to the state with highest EU in the "Zone" decision node is saved as the prediction of A_1. The predictions can be seen in Table 7.1.

	Class	**Zone**
A_1	Sniper	Ramp
A_2		
A_3		
A_4		
A_5		

Table 7.1: Predictions after A_1.

The evidence on the sensor nodes in $\mathcal{M}$ is now modified to account for the prediction of A_1, i.e. if A_3 predicted that A_1 is in the ramp zone and A_3 knew that A_1's class is "Sniper", the evidence on the "Guess Snipers in Ramp" node will be change from the state **0** to the state **1**. Furthermore the

evidence on the "Class" decision node is changed so it corresponds to A_2's class and the constraint node is set to **1**. Now the zone decision with the highest EU can be saved as the prediction of A_2's decision. The predictions can be seen in Table 7.2.

	Class	**Zone**
A_1	Sniper	Ramp
A_2	Sniper	Battlement
A_3		
A_4		
A_5		

Table 7.2: Predictions after A_2.

A_3 should now predict himself - however, this is not necessary as A_3 know where he was located just before he died, which is used as the other agents prediction of his location. The predictions can be seen in Table 7.3, where the prediction of A_3 is marked in italic to show that it is not the result of an actual prediction.

	Class	**Zone**
A_1	Sniper	Ramp
A_2	Sniper	Battlement
A_3	*Engineer*	*Flag*
A_4		
A_5		

Table 7.3: Predictions after himself.

The same goes for A_4, as A_3 just saw that it was defending the "Flag" zone, this is simply added to the set. This can be seen in Table 7.4.

	Class	**Zone**
A_1	Sniper	Ramp
A_2	Sniper	Battlement
A_3	*Engineer*	*Flag*
A_4	*Engineer*	*Flag*
A_5		

Table 7.4: Predictions after A_4.

All sensor nodes are updated to accommodate the latest predictions and a prediction of A_5 is obtained. Finally A_3 remove the prediction about

himself and replaces the prediction of A_4 with the evidence, which is entered into the node "Known Engineers in Flag" instead of the "Guess Engineers in Flag" sensor node. The final set of evidence can be seen in Figure 7.5. A_3 can now, based on $\mathcal{M}$, choose the class and zone with the highest EU.

	Class	**Zone**
A_1	Sniper	Ramp
A_2	Sniper	Battlement
A_3		
A_4	*Engineer*	*Flag*
A_5	HWGuy	Ramp

Table 7.5: Final evidence.

7.1.2 Request Message

When all the predictions have been made, the bot uses the VoC technique described in Section 6.2.2 on page 70 to consider if any of the predicted agents should be asked about its location, instead of using the prediction of the agent. To use Equation 6.1 on page 70 for a specific agent A, the bot need to know the following things:

- e, the initial evidence. This contains B's knowledge about other agents and his predictions.
- $P(E|e)$, the probability that A will answer. In our implementation we use a simple frequency over previous times A has been asked. I.e. $P(E) = \frac{r}{n}$, where r is the number of times A has replied and n is the total number of times A have been asked. The initial frequence is $\frac{20}{20}$ and is updated every time an agent is asked (using a fading factor of 0.975, just like the influence diagram).
- $P(Answer = Z|E, e)$, the probability of A answering that it is in zone Z, given that it answers. There are different ways of estimating these probabilities. As mentioned in Section 6.2.2, the probability of the variable, for which information is being requested, could be used. However, our influence diagram do not have an explicit variable that represent A, since specific agents have been grouped together in general nodes. Instead we use the frequency $P(Answer = Z|E) = \frac{z}{r}$, where z is the number of times A have replied that it were in zone Z (A fading factor of 0.975, as described above).
- γ, the cost of sending the request. γ will be set to different values during the testing, depending on the circumstances.

A flowchart of the spawning process can be seen in Figure 7.1.

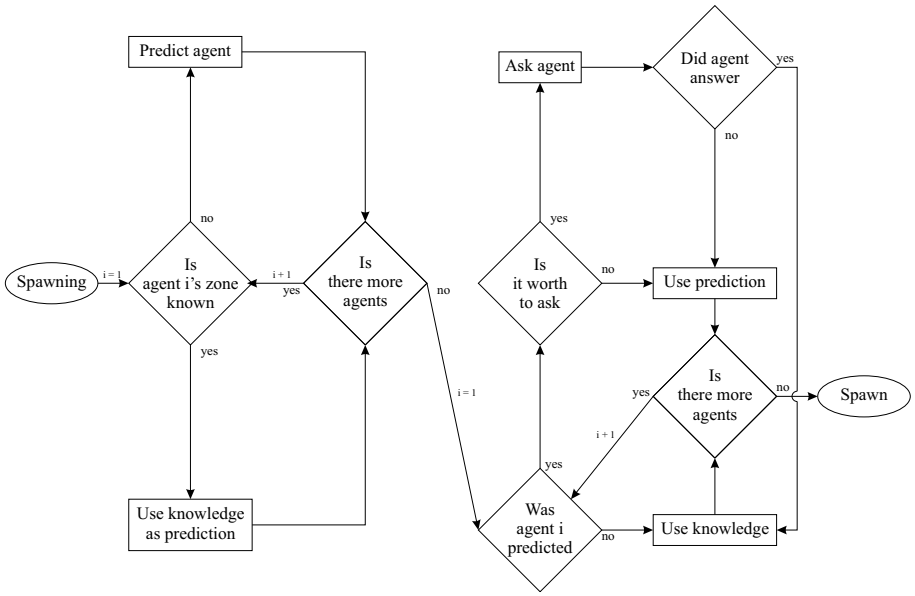


Figure 7.1: Flowchart of the spawning process.

7.2 Sending Information

When a bot B receives a request for information, it must determine if it is worth the cost to answer the agent A who send the request (Note, that this means we never consider sending information to more than one agent). To use the method described in Section 6.2.3 on page 71, B first computes WAEU($B \cup A$), the WAUE of itself and A by using Equation 6.4. This equation requires a set of weights (we always use equal weights i.e. 0.5 for both agents) and the following evidence:

- e_B, B's initial evidence. This contains B current knowledge, as well as a prediction of all agents.
- e_A, B's evidence for A. This is the part of e_B that B uses when computing the MEU for A. This includes the part of B's knowledge that B knows A know, as well as the predictions of other agents. Actually B should make new predictions for all agents, seen from A's perspective. However, as these would be made with the same models as the ones' B has already made, the predictions are simply reused.

When this is done B computes WAEU($B \cup A|M$), the WAEU for sending the message $M = e$ to A. This is done with the Equation 6.5, which requires the following sets of evidence:

- e, the evidence corresponding to the message M. E.g. I (B) is defending the "Ramp" zone.
- e', the evidence that B knows that A knows e.
- e_B, B's initial evidence. But this time with a new prediction for $A|e$.
- e_A, B's evidence for A. This is the same set as before.

Now B can compute the value of sending the message M by subtracting WAEU($B \cup A$) and the cost γ from WAEU($B \cup A|M$) and determine if replying to the request is worth the cost.

Example of Information Message The agent A has asked the bot B where he is located and B is considering if he should reply with message $M = e$. Both A and B has a weight of 0.5.

B must first compute WAEU($B \cup A$). This is done in the following way:

1. Compute a set of predictions for all agents.
2. Compute $MEU(B|e_B)$.
3. Estimate $MEU(A|e_A)$. This is done by entering e_A into B's own model (which is used as approximation of A) and giving evidence on the decisions that correspond to B's prediction of A.
4. Compute WAEU($B \cup A$)

Next B must compute WAEU($B \cup A|M$), which requires the following:

1. Compute a new prediction for A, this time with the evidence e', i.e. A knows e.
2. Compute $MEU(B|e_B, e')$.
3. Estimate $MEU(A|e_A, e)$.
4. Compute WAEU($B \cup A|M$)

B now computes the weighed average increase in EU for sending the reply and if it is higher than 0, he sends M to A.

8 Test

With the implementation of the three layers in the architecture, the agent implementation is complete and ready to be tested in the TFC environment. Since the lowest layer in the architecture, the reactive layer, has not been developed in this report, there will not be conducted any test of it. Therefore, the chapter starts with tests of the deliberative layer, as this is the foundation for the cooperative layer, and finishes with the cooperative layer. The test of the cooperative layer will test the effects of the two techniques developed to support cooperation, prediction and VoC, to see how much they improve the performance of the implemented bots.

However, before commencing with the testing, a test strategy is planned in order to ensure that the tests are performed in a logical order, and that the important aspects of the agent implementation are tested. The test strategy should also ensure efficient utilization of the test resources, which is an important aspect, as each test takes between 12 and 24 hours. Finally the test setup is described before the actual testing can begin.

In Section 8.1 the test strategy are presented, Section 8.2 describe the test setup and Sections 8.3 - 8.6 contain the performed tests and a discussion of these. At the beginning of each test section is a table with the test parameters (the abbreviation HW, SO, SN and EN are used for the classes HWGuy, Soldier, Sniper and Engineer respectively).

8.1 Test Strategy

The test strategy is based on a bottom-up principle, i.e. simple tests are performed first followed by more and more advanced ones. Due to the amount of test data, it is necessary to present the data in a compact and comprehensible manner which, in this report, this is done by using graphs. Excel sheets containing all test data can be found at [TR04].

The first set of tests will only have one defending bot, and will be performed with a bot that select its strategic decisions randomly and by a bot that is controlled by the deliberative layer. These tests should show that the bot that is controlled by the deliberative layer adapt to the situation, and end up performing better than the bot that selects its decisions randomly.

In addition, it should be checked that the bot explore the available agent configurations, and that various runs of the same test will produce similar results.

The second set of tests will be performed with two defending bots, and will have four tests: The first test is, again, a test where the bots select their strategic decisions completely random. The second test is performed with the deliberative layer enabled, but without any cooperative techniques. The third test is performed with the deliberative layer enabled and with prediction in the cooperative layer enabled. The last test, which is a reference test, should confirm the results obtained in the previous tests. This test set should show the advantage of prediction, that the agents explore the available state space, and that various runs of the same test have similar results.

The third set of tests will be performed with four defending bots. The purpose of this tests set is to examine the effect of estimating combat cases when an bot is killed (see Section 5.3.2 on page 58) and to determine the effect of VoC. The test set will consist of tests performed with no communication, tests where bots are force to exchange information, and lastly tests that use the two methods from VoC.

At the end of the chapter, the empirical standard deviation of all test are shown and briefly discussed. Note that many of the graph presented in this chapter shows the cumulative number of captures in a test, which are good to show tendencies over time, but makes it hard to see irregularities in the different tests. Therefore, graphs showing the average capture rate, in fixed 30 minutes interval, is shown for all tests.

8.2 Test Setup

The tests are performed on an installation of the original Half-Life installed on Windows2000™. The Half-Life installation is patched with patch 1.1.1.0, which adds TFC to the game. In addition, the third-part software metamod 1.17[1] is installed, before our agent implementation for TFC is installed.

In all tests, unless otherwise specified, the defense agents starts with an untrained version of the influence diagram developed in Chapter 5 on page 45. The fading factor for all nodes are set to 0.975, corresponding to an effective sample size of 40, and the initial experience count are set to 20 for the nodes that are updated with fractional updating.

The initial probabilities for "Ex C in Y" have uniform distribution with a probability of $0.\bar{3}3$ for each state, and the "Guess C in Y" nodes have an initial probability of 0.95 for a guess being correct.

[1] Metamod is a plugin/DLL manager that sits between the Half-Life Engine and an HL Game mod, allowing the dynamic loading/unloading of mod-like DLL plugins to add functionality to the HL server or game mod.

The initial probabilities for "Before X Zone" and "Takes Flag" nodes for the parent configuration where an attacker is alive and there is at least one defender are all set to 0.001 and 0.999 for the "Alive" and "Dead" state respectively. The reason for the high starting probabilities for an attacker to die in an encounter is that it makes the defenders explore the available state space. E.g. if a defender decides to defend a specific zone, sooner or later an attacker will get through the zone alive, this will decrease the probability for the attacker to die in the zone. The defender will therefore select to defend a zone with a higher probability for the attacker to die. In this way the defender will cycle through the possible decisions and decrease the high starting values. In the end the defender will reach a state where the probabilities do not decrease any further, and the evolution stop. The untrained version of the influence diagram[2] can be found at [TR04].

Note that, as the initial values are independent of the number of defenders in a zone, the defenders will try to avoid each other in the beginning, this have the result that the defenders mainly get cases for configurations with one defender in a zone, when the initial high probabilities for defenders stopping the attackers has dropped a bit, they will start exploring the configurations with two defenders in a zone and so on. This is important as the configurations with more than one defender utilize the configurations with fewer defenders when a defender dies in a zone with remaining defenders.

8.3 Tests with One Defender

	Test 1	Test 2
Defenders		
Number	1	1
Available Classes	HW, SN, EN	HW, SN, EN
Attackers		
Scouts	1	1
Soldiers	1	1
Medics	1	1
Techniques		
Random	no	yes

Table 8.1: Test setup for tests with 1 defender against 3 attackers.

In Table 8.1 the test setup for the two test can be seen. The first test is performed with random behavior for strategic decisions, i.e. the bot chooses a random class and location to defend each time he spawns. The second test is performed with the agent developed in the preceding chapters, where the deliberative layer select the bot's strategic decisions. Note that, as there is only a single defending bot, the cooperative layer is never activated.

[2]The influence diagram is saved in Hugin's .net v5.7 format.

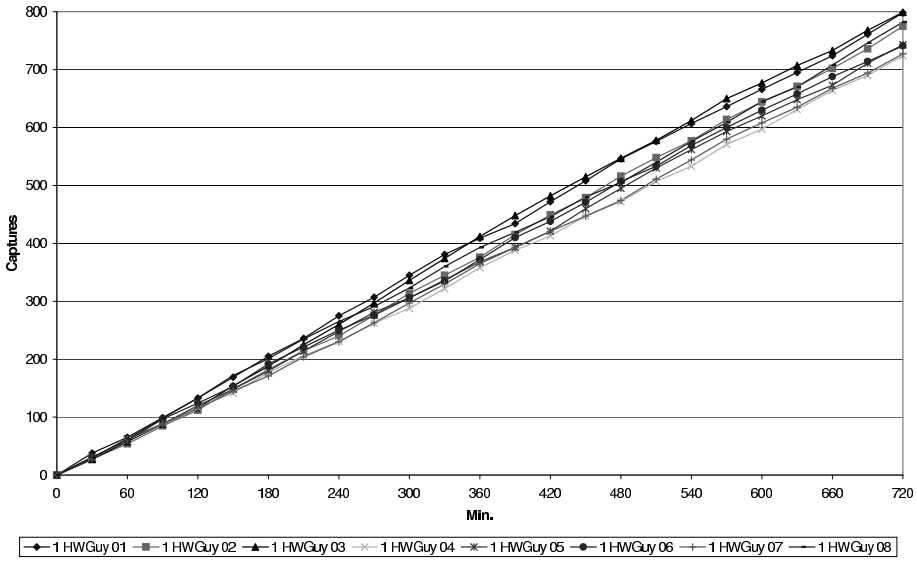


Figure 8.1: 1 defender against 3 attackers, test runs with random behavior.

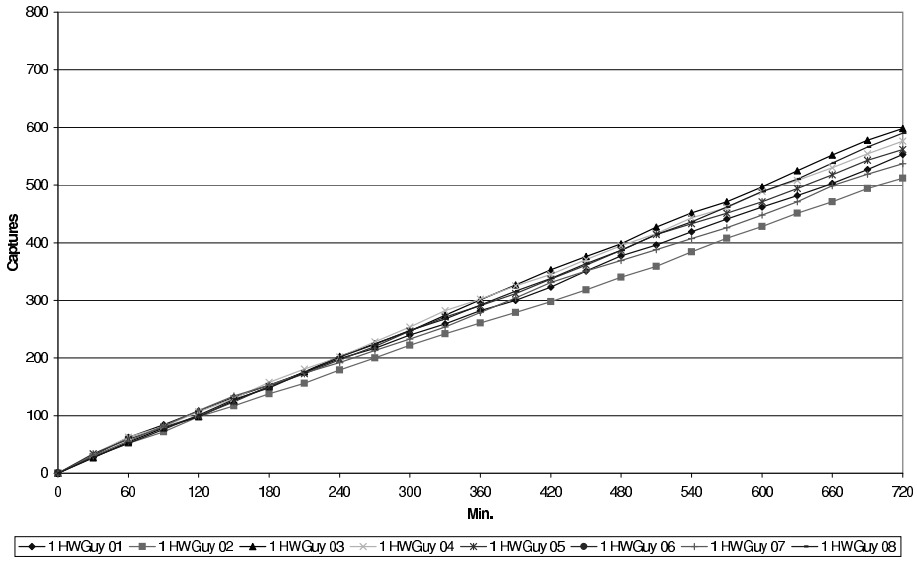


Figure 8.2: 1 defender against 3 attackers, test runs with model.

The results of the individual test runs of the two tests can be seen in Figure 8.1 and Figure 8.2 on the facing page. These graphs is only used to check that the test runs delivers consistent results and will not be shown for any of the subsequent tests in this chapter. However, a examination of the deviance in all test sets can be found in Section 8.7 on page 97 and graphs with individual test runs for all tests can be seen in Appendix C on page 123, which also contains all graphs shown in this chapter.

The graph shows the cumulative number of times the attacking team has managed to capture the flag. The more captures, the worse the defending team is doing. In the first test the number of captures vary between 700 and 800 after 12 hours, and in the second test between 500 and 600. The graph also shows that the individual test runs follow the same pattern to reach the end results. We judge that the deviance between the individual test runs are acceptable (for more details see Section 8.7).

To compare the two test set directly, the averages of the 8 individual test runs in each test set has been depicted in Figure 8.3 on the next page. The graph shows that the defender with random behavior has a considerably worse performance than the defender controlled by the model.

A graph over the average number of captures for each 30 minute interval can be seen in Figure 8.4 on the following page. Note that the graph's y-axis starts at 20 instead of 0, as it makes it easier to see the difference between the two tests.

From this graph it can be seen that, on average, the flag has been captured around 30 times the first 30 minutes in both test sets, but after 60 minutes the test with the model has decreased the capture rate to about 27, and after another 30 minutes the capture rate is down to 22, however the improvement stop after 90 minutes, as the capture rate oscillates between 20 and 25 for the remaining time of the test. The explanation of this behavior is that in the beginning the defender tests his options. Eventually he reach the conclusion that he should be play a HWGuy and that it do not really matter where he stands as long as it is in one of the zones that an attacker must pass through (an example of an influence diagram showing this can be seen at [TR04]). When he has reached this strategy, after an hour or so, the improvement stops. In the test with random behavior the capture rate oscillates between 29 and 34 throughout the test, this is as anticipated. These results indicates, that the implementation of the deliberative layer successfully models aspects of the domain, and that the defending agent's performance increase over time.

When looking at the networks resulting from the test, we can see that all parent configurations to the "Before X Zone" and "Takes Flag" nodes, that are possible to achieve with one agent, have been tried several times (determined by looking at the experience count, which has increased from the initial count of 20). This indicates that the bot do try out all different configurations.

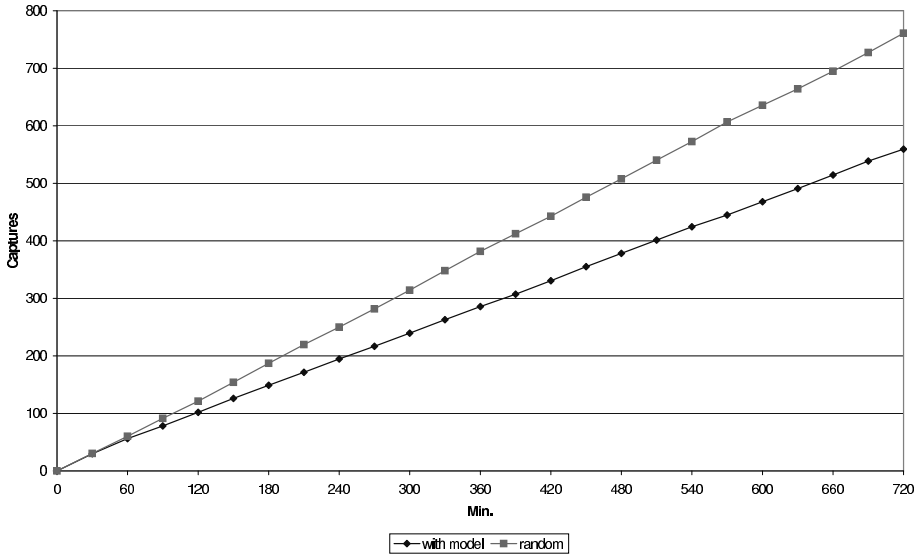


Figure 8.3: 1 defender against 3 attackers, average of the test runs with random behavior and with model.

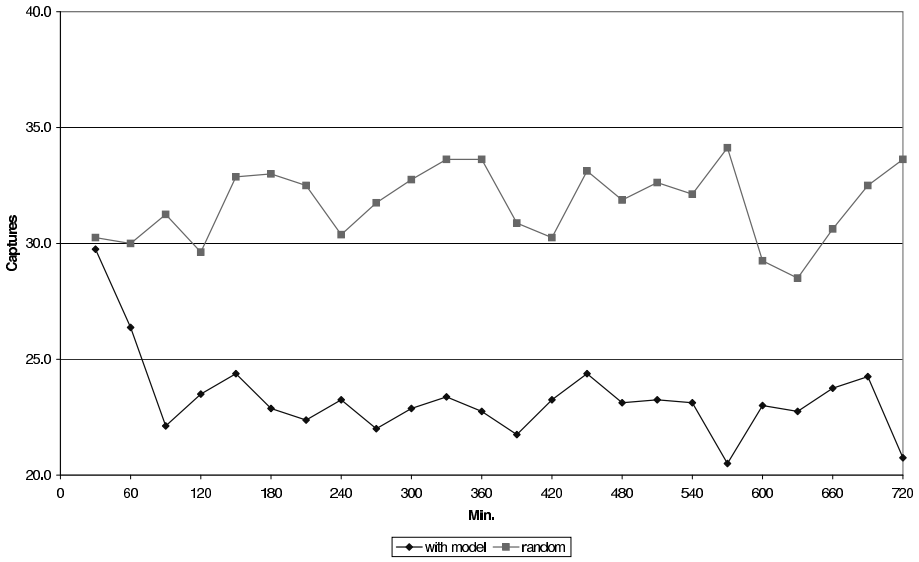


Figure 8.4: 1 defender against 3 attackers, average captures for each half hour with random behavior and with model.

8.4 Tests with Two Defenders

	Test 1	Test 2	Test 3	Test 4
Defenders				
Number	2	2	2	2
Available Classes	HW, SN, EN	HW, SN, EN	HW, SN, EN	HW, SN, EN
Attackers				
Scouts	2	2	2	2
Soldiers	2	2	2	2
Medics	2	2	2	2
Techniques				
Random	Yes	No	No	No
Prediction	No	No	Yes	Yes
Trained Diagrams	No	No	No	Yes

Table 8.2: Test setup for tests with 2 defenders against 6 attackers.

In Table 8.2 the test setup for the four tests can be seen. The first test is performed with random behavior for strategic decisions, i.e. the two bots chooses random classes and locations to defend. The second test is performed with the deliberative layer enabled, but without any cooperative techniques and the third test is performed with both the deliberative layer and with the prediction technique enabled in the cooperative layer. The fourth test is a reference test, which is a continuation of the third test with prediction enabled, i.e. it has the same setup, but the bots are started with the trained influence diagrams from the third test.

The average cumulative captures for the three tests can be seen in Figure 8.5 on page 91. The test with random behavior perform worst with over of 450 captures after 12 hours. The second test ended with around 260 captures and the third test with 230 captures. From this it can be concluded that the deliberative layer enable bots to perform substantially better than bots with random behavior, which conform to the findings from the tests with one defender. When the captures for second and the third test are compared, it can be seen that two bots actually benefit from the prediction technique, however the difference is rather small.

In Figure 8.6 on page 91 the three tests' average capture rate per 30 minutes is shown. The test with random behavior has a fairly stable capture rate throughout the test, which is expected, as the bots do not have any adaptive behavior. The second and third test both starts with a capture rate around 25, which during the tests falls to around 6-7 captures per 30 minutes in the end. However, the graph indicates that the capture rate in the test with prediction enabled drops a bit faster than the test without prediction, however the difference is rather small and should not be over-interpreted, especially since the capture rate at the end of the two test are almost identically.

Notice, that the test run with random behavior does considerably better

than the other two tests during the first 30 minuets. The reason for this is that, as described in the test setup, the initial probabilities in the influence diagram has been set so that, at the start of the test, it is bad for agents to be in the same zone. The random agents have no such inhibitor, as they do not use the diagram, their probability for having more agents in the same zone is the same at the beginning of the as at the end.

In Figure 8.7 on page 92 the cumulative captures are depicted, the defense team with trained diagrams (Test 4) performs considerably better than the defense team with untrained diagrams. The graph in Figure 8.8 on page 92 shows that the decrease in capture rate at the beginning of all previous test, with the deliberative layer enabled, are indeed caused by the fact that the bots learn and is not a general effect of the environment or implementation.

Like the previous test, the networks, resulting from this test, show that all parent configuration that correspond to one agent in a zone, as well as the ones corresponding to two agents, have been tried several times. Thus we conclude that the bots do try to explore all possible agent configurations.

8.5 Tests with Four Defenders

	Test 1	Test 2	Test 3	Test 4
Defenders				
Number	4	4	4	4
Available Classes	HW, SN, EN	HW, SN, EN	HW, SN, EN	HW, SN, EN
Attackers				
Scouts	5	5	5	5
Soldiers	15	15	15	15
Medics	5	5	5	5
Techniques				
Prediction	No	Yes	No	Yes
Gen. Cases	Yes	Yes	No	No

Table 8.3: Test setup for tests with 4 defenders against 25 attackers.

In Table 8.3 the test setup for the four tests can be seen. The first test is performed with the deliberative layer enabled, but without any cooperative techniques, the second test is performed with the deliberative layer and with the prediction technique enabled in cooperative layer.

Test 3 and 4 are identical to test 1 and 2 except that they do not use the technique to estimate combat case when a defending agent dies in a zone with remaining defending agents. If a defending bot in test 3 or 4 are killed alongside other defending agents, they instead get a cases where the attacker succeeds in getting through the zone.

The first graph, seen in Figure 8.9 on page 94, shows that, like the previous tests with 2 defenders, the cooperative prediction technique improve the

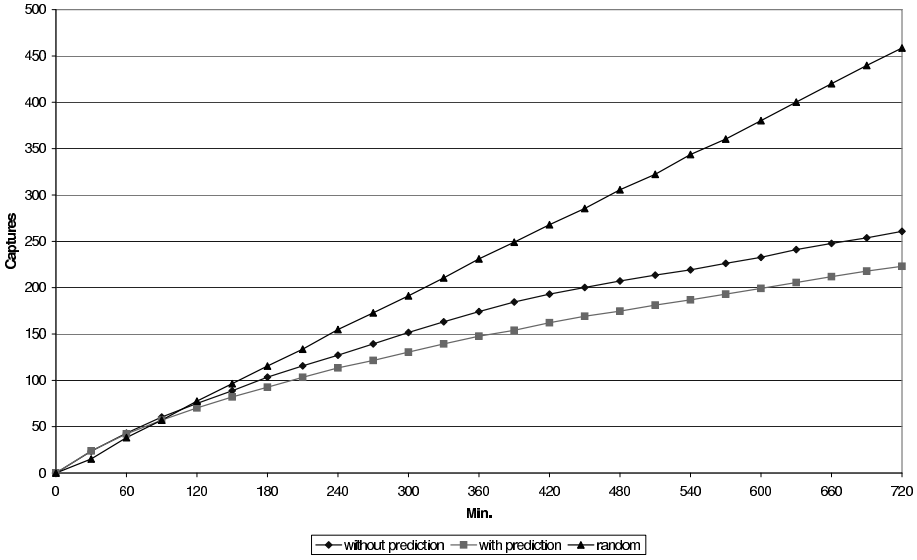


Figure 8.5: 2 defenders against 6 attackers, averages of the first three tests.

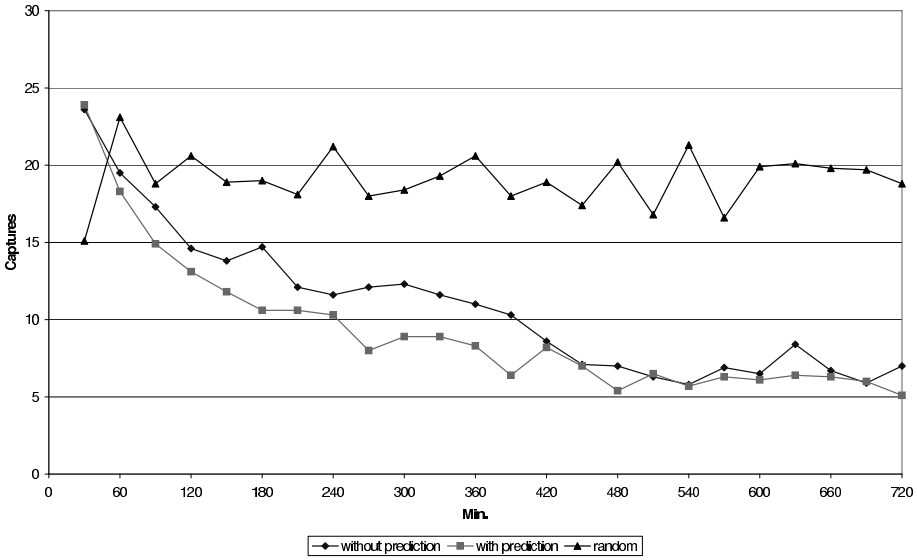


Figure 8.6: 2 defenders against 6 attackers, average captures for 30 minutes of the first three tests.

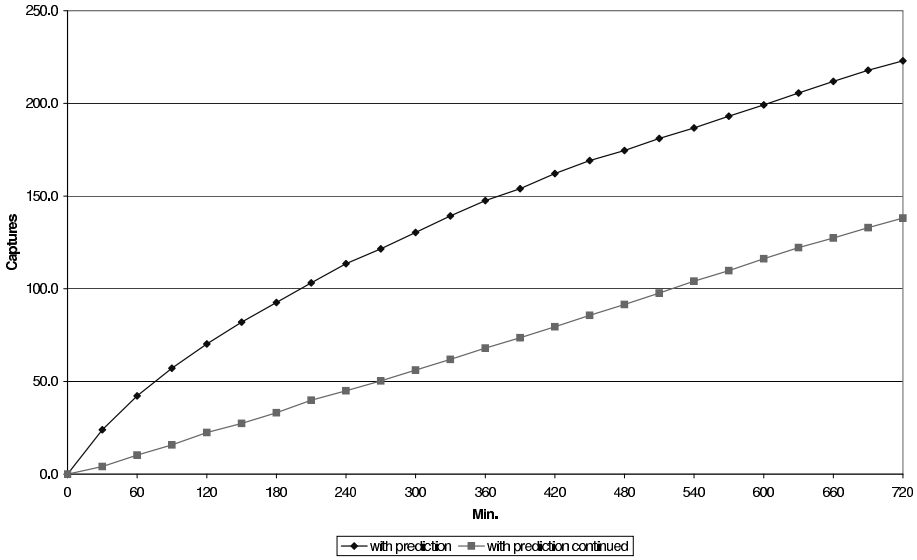


Figure 8.7: 2 defenders against 6 attackers, averages of the tests with trained and untrained influence diagram.

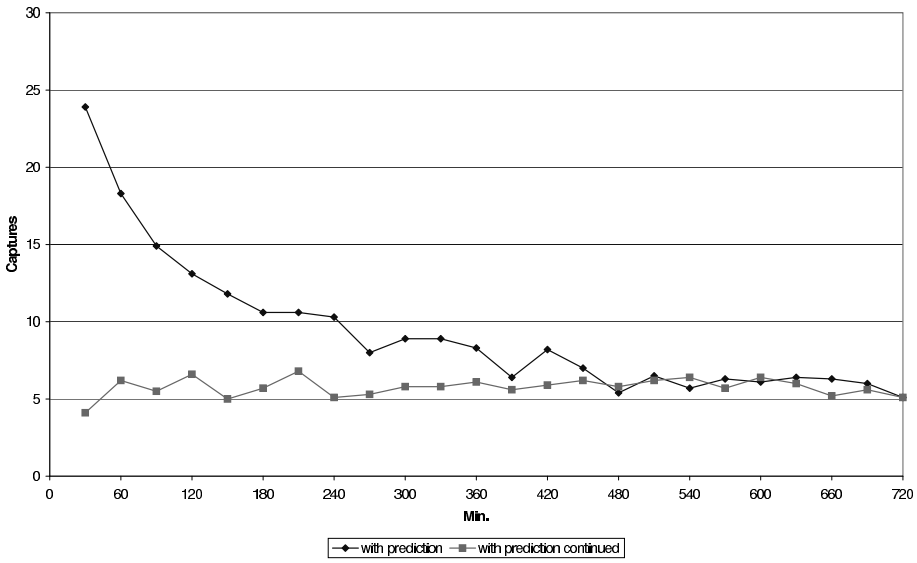


Figure 8.8: 2 defenders against 6 attackers, average captures for 30 minutes of the tests with trained and untrained influence diagram.

defending team's performance, but the disparity is more pronounced with 4 defenders. When comparing the tests with and without the technique for generating cases, it can be seen that generating cases do not seem to lead to a better strategy, it just seem to reach the same behavior faster. In addition, it can be seen in Figure 8.10 on the following page that the defense team using prediction reach a better end strategy than the bots without prediction.

The second test, with 4 defenders using prediction, shows that the defenders make it almost impossible for the attackers to capture the flag. The test setup is therefore not useful to demonstrate any potential advantage of the "Value of Communication" cooperation technique. In addition, another drawback has been observed through this and the previous tests: The defenders more or less always end up playing HWGuy, which kills the interaction between the different classes. It has therefore been decided to switch the HWGuy and Soldiers classes between defense and attack respectively. In the remaining test sets the defenders can therefore switch between Soldier, Sniper and Engineer class.

8.6 Tests with Four Defenders - Revised

	Test 1	Test 2	Test 3
Defenders			
Number	4	4	4
Available Classes	SO, SN, EN	SO, SN, EN	SO, SN, EN
Attackers			
Scouts	5	5	5
HWGuys	15	15	15
Medics	5	5	5
Techniques			
Prediction	No	Yes	Yes
Request Com.	None	None	100% forced
Answer Com.	None	None	100% forced

Table 8.4: Test setup for tests with 4 defenders against 25 attackers.

In Table 8.4 the test setup for the first three tests can be seen. The first test is performed with the deliberative layer enabled, but without any cooperative techniques, the second test is performed with the deliberative layer and with the prediction technique enabled in the cooperative layer. The third test is performed with the deliberative layer enabled and with both the prediction technique and 100% forced communication between the agents, i.e. the agents always ask and receives answers from other agents (effectively eliminating prediction). This test is performed to get a reference point for comparison with other forms of communication - a sort of best case, as all agents always know where all other agents are located.

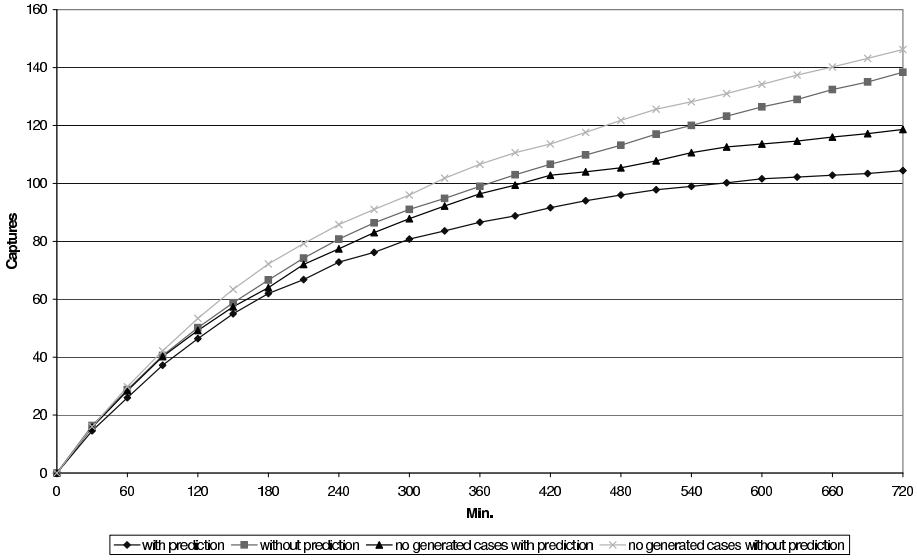


Figure 8.9: 4 defenders against 25 attackers, averages of the three tests.

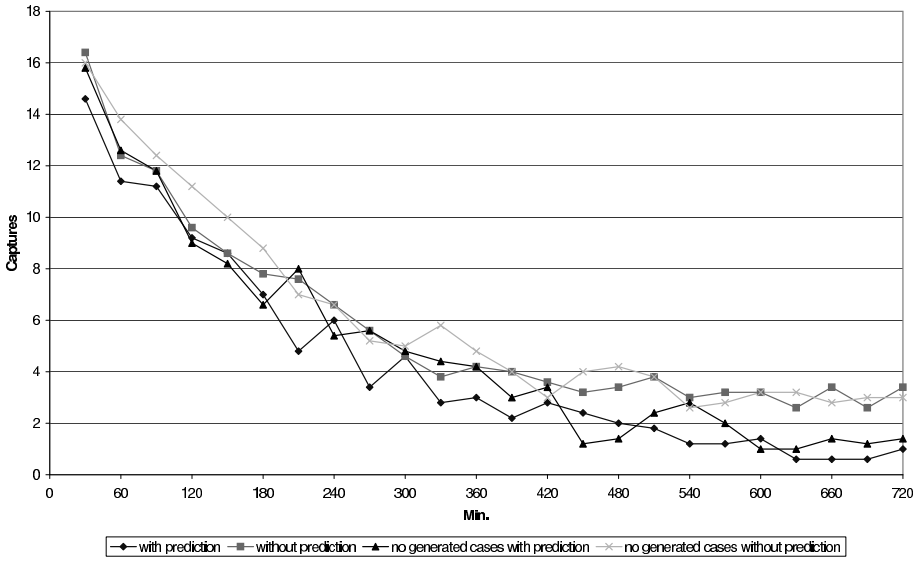


Figure 8.10: 4 defenders against 25 attackers, average captures for 30 minutes of tests.

	Test 4	Test 5	Test 6	Test 7
Defenders				
Number	4	4	4	4
Available Classes	SO, SN, EN	SO, SN, EN	SO, SN, EN	SO, SN, EN
Attackers				
Scouts	5	5	5	5
HWGuys	15	15	15	15
Medics	5	5	5	5
Techniques				
Prediction	Yes	Yes	Yes	Yes
Request Com.	50% forced	48%	100% forced	64%
Answer Com.	100% forced	100% forced	53%	74%

Table 8.5: Test setup for tests with 4 defenders against 25 attackers.

The graph containing the average of the individual test sets can be seen in Figure 8.11 on the following page. The test with only the deliberative layer enabled, ends up with around 420 captures, which makes it the worst of the three tests. The test with prediction, but no communication, ends up with little over 250 captures. The test with the best result is the test with 100% forced communication with around 100 captures.

In Figure 8.12 on the next page the three tests' average capture rate per 30 minutes is marked. It can be seen that test 2 (using prediction) perform worse than the corresponding test in the previous section that used HWGuys instead of Soldiers. The average capture rate oscillates between 3-5 versus 0-2, which leaves more room to examining the effects of VoC. It is also worth noticing that test with 100% forced communication almost end up blocking the attacking team.

In Table 8.5 the test setup for the last four tests can be seen. Test 4 is performed with 50% forced communication (the agents randomly asks 50% of the time and always receives an answer). This test is used to judge the effect of all other kinds of communication. In test 5 the agents uses VoC for evaluating if it is worth to send a request for information, which other agents are forced to answer. The cost of sending a message set to 2.9, which has been set to get a message exchange as close as possible to 50% and has been obtained through trial and error. In test 6 an agent will request information all the time, but other agents uses the method from VoC to evaluate if it is worth answering the requests. The cost of sending a message set to 3.8. This value has been set to get a message exchange as close as possible to 50% and has been obtained through trial and error. Test 7 is performed using VoC for both requests and answers, the cost of sending a message set to 2.2 for request messages and 2.7 for answering a message. These values has again been set to the message exchange as close as possible to 50%.

Figure 8.13 on page 98 contain the cumulative captures for test 2-7. Test 3 and 4 (Full communication and 50% Communication, respectively) are used as reference points for the three VoC tests (test 5-7), which have had

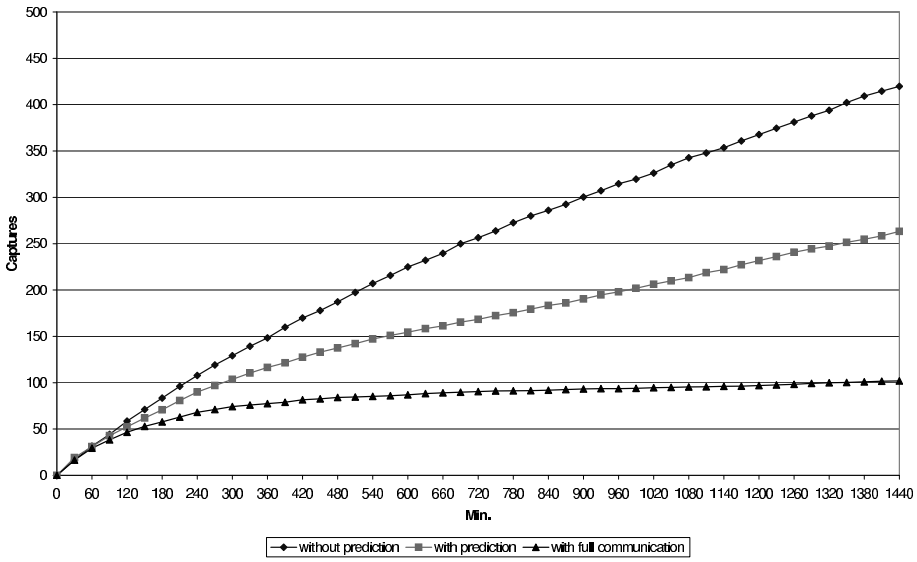


Figure 8.11: 4 defenders against 25 attackers, averages of the first three tests.

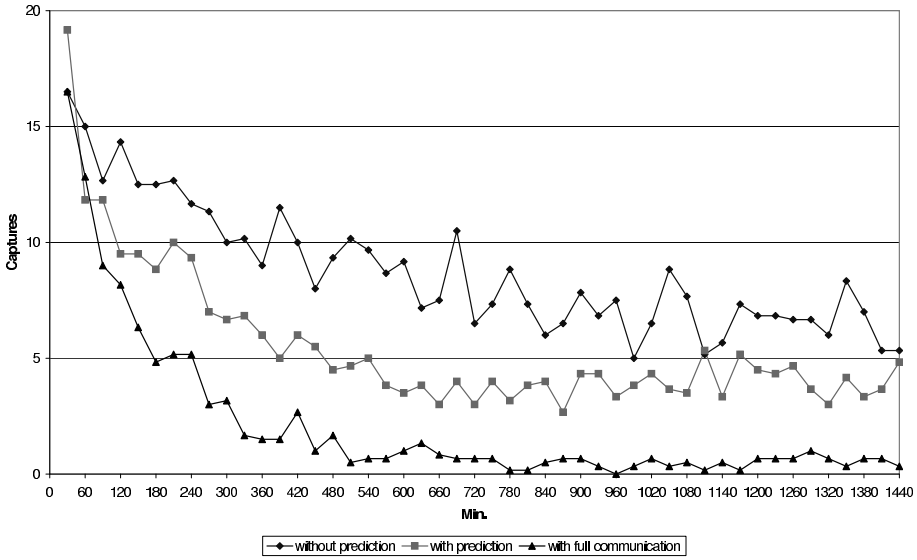


Figure 8.12: 4 defenders against 25 attackers, average captures for 30 minutes of the first three tests.

their parameters set to achieve a message exchange rate of approximately 50% (i.e. 50% of all predictions will be replaced with knowledge).

Test 5 (VoC request) is, with around 125 captures, significant better than the 50% random communication, this indicates that the technique is capable of determining when it is profitable to ask another agent for its location instead of using the prediction of the agent. However, the result of test 6 (VoC answer) is almost completely identical to the test with 50% forced communication, which means that for some reason the technique is incapable of computing when answering a request is going to improve the team's performance. Given the result of test 6, it is not surprising to see that the last test, with agents using VoC for both requests and answers, performs slightly worse than the test with only VoC for request. As, agents will answer rather arbitrarily, it undermines the agents' ability to determine when to send requests.

It can therefore be concluded that VoC for requests is actually able to select the cases where communication can be performed with an advantage, but VoC for answers is not working.

The final graph shows the average captures for each 30 minutes. The graph can be seen in Figure 8.14 on the following page (A section of this graph, showing the last 14 hours in increased detail, can be seen in Figure 8.15 on page 99.). In the graph it can be seen that the 4 defenders with 100% forced communication or 47% VoC are almost capable of preventing the huge attack team from taking the flag after 510 minutes. This is quite a feat, considering that they are outnumbered 6 to 1 and that the HWGuy class has been exchanged with the Soldier class.

8.7 Deviance

The empirical standard deviation for the end result of the different tests sets are computed with the following equation:

$$deviance = \frac{\sqrt{\frac{\sum_{i=1}^{n}(t_i - a)^2}{n}}}{a} \cdot 100 \tag{8.1}$$

where

a is the average number of capture for the performed tests
t_i is the captures for i'th test
n is the number of performed tests

The deviance for all performed test can be seen in Table 8.6 on page 99. In general all test have a deviance of 9% or less. The two notable exceptions are the test with 100% forced communication and the test with VoC. An likely explanation of why these test have a higher deviance is that, as they

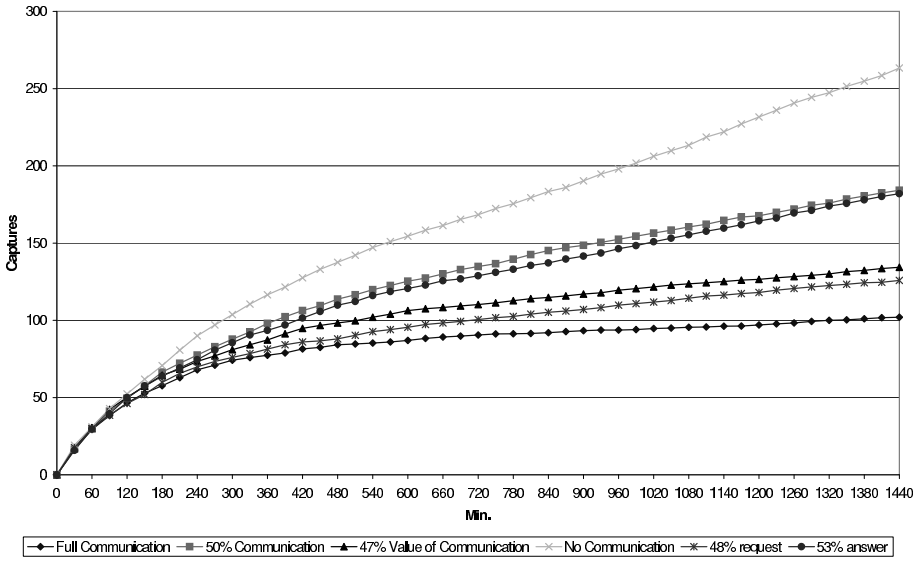


Figure 8.13: 4 defenders against 25 attackers, averages of the six VoC tests.

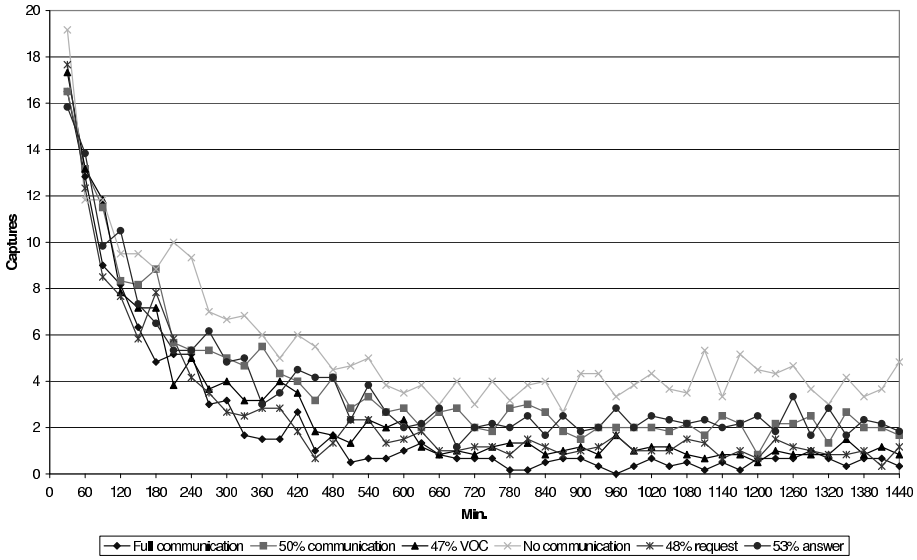


Figure 8.14: 4 defenders against 25 attackers, average captures for 30 minutes of the six VoC tests. A section of this graph showing only the last 14 hours can be seen in Figure 8.15 on the next page.

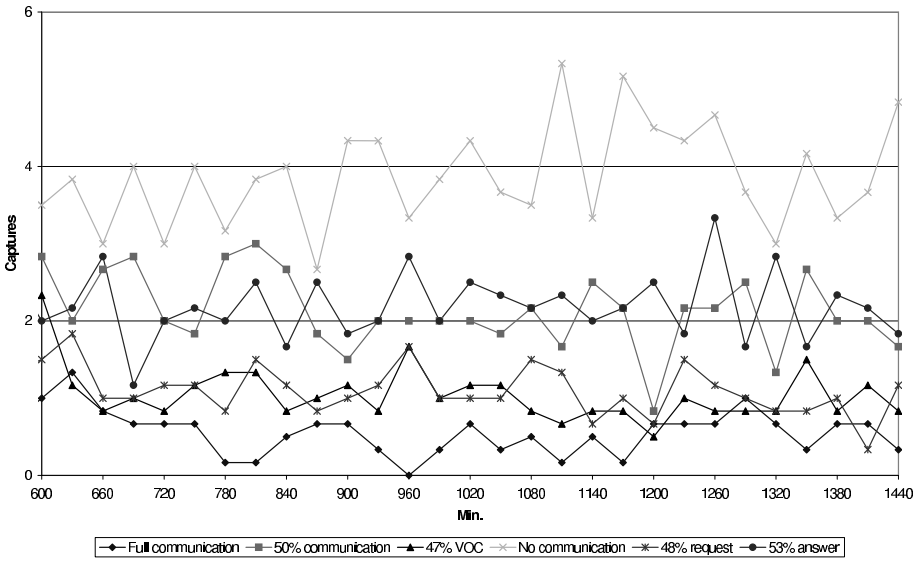


Figure 8.15: A section of the graph in Figure 8.14 showing the last 14 hours in greater detail.

1 Defender	
Random	3.80%
With model	4.74%

2 Defenders	
Random	5.52%
Without prediction	7.35%
With prediction	6.77%
Prediction cont.	8.29%

4 Defenders	
Without prediction	4.49%
With prediction	6.02%
No case generation	6.58%
No case generation, pred.	6.50%

4 Defenders - Revised	
Without prediction	9.00%
With prediction	5.80%
100% Forced	16.22%
50% Forced	5.16%
47% VoC	9.92%
48% Request	6.86%
53% Answer	4.48%

Table 8.6: The empirical standard deviation for all test sets.

eventually becomes capable of almost completely stopping the attackers, their final results are very sensitive to when they reach the best strategy.

9 Conclusion

This chapter concludes the project with a review of the accomplished results and by putting them into perspective. Finally, some of the possibilities for future work is presented.

The gaming industry has so far mainly focused on advancement in graphics and sound technologies. However, as these have reached a fairly high level it is becoming increasingly harder to sell games on new graphics and sound features. Game developers are therefore starting to look towards new areas to differentiate their game from the competitors. One of these areas is AI for bots that, in present games, mainly have been simple, scripted agents with behavior that often decreased the player's suspension of disbelief.

One the possible feasible techniques, which could be used to create more advanced AI is Bayesian networks, has already been employed in two projects at Aalborg University to demonstrate its applicability for FPS death match[1] games, both projects focused on constructing an adaptive agent and demonstrate the effect of adaptation, the results of these project can be found in [Ben04] and [LOT+04]. Agents with adaptive behavior has the possibility of extending a game's duration, as the player find counter-tactics to the agent's new tactics, thereby making the game more intriguing.

The recent batch of FPS games has focused on cooperative team-based multiplayer game play. This poses new demands to the bots, as the gamers now expect intelligent cooperative behavior from the bots. The most advanced bots in these games are only are capable of exhibiting simple cooperative behavior as following a player and obeying simple commands as "defend", "attack" etc. from the human players. If computer- controlled agents where capable of intelligent strategic cooperation, without needing orders from human players, they would appear more intelligent and increase the player's gaming experience.

9.1 Results

This report has presented two techniques to support agent-centered cooperation. These as been implemented in a adaptive bot and tested in a FPS

[1] The objective in FPS death match games is simply to kill the other agents.

domain, where agent-centered cooperation is especially needed as human players have little or no time to communicate with the agent.

The first of the techniques was prediction, which was used as the primary techniques, as it does not rely on communication. The conducted tests have shown that, in relation to both random behavior and normal adaptation, the prediction technique causes a noticeable improvement in the implemented bot's performance.

The second technique, called VoC, was used by the bots to determine if it was worth attempting to communicate with other agents in a given situation. The technique actually consists of two different methods - one for messages that requests information from other agents, and one that sends information to other agents. The tests of VoC have shown that, when used to support prediction, the method for request information can successfully determine the best instances to ask other agents for information. In fact, the test shows a significant improvement in the bot's performance seen in relation to only using prediction and also in relation to using a similar percentage of random communication. However, the test also shows that the second part of VoC - the method for sending information to other agents, seems incapable of determining whether it is most profitable for the bot to send information.

One reason why the second method from VoC has failed, could be that the domain or the model of the domain simply contains too many uncertainties or is to inaccurate for the method to estimate the MEU of other agents. An implementation in a less complex environment, like the sheep hunting caveman found in [VJ04], could probably determine this. Unfortunately, we have not had the time to do so.

9.2 Further Work

Active Information Sending The techniques presented in this report have been focused on agent-cantered ways to support cooperation. A notable drawback of this is that the techniques are vulnerable to local minimum and maximum strategies, where that team finds locally optimal behaviors. E.g. there could be a zone where one single agent will perform horrendously, but two agents will perform very well. The agent configuration with two agents in the zone can be very hard to achieve, since no agent want to be the first to position itself in the zone.

To solve this kind of problem, one could attempt to merge the VoC techniques with regular negotiation between the agents. However, this would undermine the agent's ability to function with human players. Remember, that human players in FPS may not have time or the ability to negotiate with other agents. Another idea could be to more actively try to influence other agents by sending them information. E.g. if a bot thinks that there is a zone where there ought to be two agents, it could tell another agent, that

the bot is going to the zone and hope that the agent will choose to support it.

Support for Human Players The bot implementation created in this report, where implemented with the intention of testing it versus other bots. Therefore, it does not support humans as players - only as observers that can interact with the game world. It might be interesting to test the bots' ability to support a human player.

Bibliography

[Ben04] Rimantas Benetis.
Rbot - The Intelligent Agent in Unreal Game Environment.
www.cs.auc.dk/library/cgi-bin/detail.cgi?id=1026515339, July 14, 2004.

[bot04] Jeffrey Broomes's Homepage.
www.planethalflife.com/botman/, March, 2004.

[Bou96] Craig Boutilier.
Planning, Learning and Coordination in Multiagent Decision Processes.
www.cs.toronto.edu/~cebly/, 1996.

[CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg.
Distributed Systems Concepts and Design, Third edition.
Addison-Wesley, 2001.

[DFJN97] J. E. Doran, S. Franklin, N. R. Jennings, and T. J. Norman.
On Coorperation in Multi-Agent Systems.
The Knowledge Engineering Review, 12(3), 1997.

[FG96] Stan Franklin and Art Graesser.
Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents.
Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[FM99] Roberto A. Flores-Mendez.
Towards the Standardization of Multi-Agent System Architectures: An Overview.
ACM Crossroads, Special Issue on Intelligent Agents, Association for Computer Machinery, Issue 5.4, pp. 18-24, 1999.

[Hug04] Hugin's Homepage.
www.hugin.com, Juli 2. 2004.

[int04] Slides for InteRRaP from University of Alberta.
www.cs.ualberta.ca/~ree/courses/cmput652/classmaterials/InterRap.slides.pdf, 2004.

[Jen01] Finn Verner Jensen.
Bayesian Networks and Decision Graphs.
Springer-Verlag, 2001.

[KR03] James F. Kurose and Keith W. Ross.
Computer Networking, Second edition.
Addison-Wesley, 2003.

[LOT+04] Jacob Larsen, Henrik Hvergel Oddershede, Martin Guld Thomsen, Alex Vendelbo Ringgaard, and Niels Cristian Nielsen.
Advanced AI In Computer Games.
www.cs.auc.dk/library/cgi-bin/detail.cgi?id=1073308615, July 14, 2004.

[Mül96] Jörg P. Müller.
The Design of Intelligent Agents, 1996.

[OJ92] S. L. Olesen, Kristian G.and Lauritzen and Finn V. Jensen.
aHugin: A System Creating Adaptive Causal Probabilistic Networks.
Uncertainty in Artificial Intelligence, Proceedings of the Eighth Conference, p.223-229, July 17-19, 1992.

[pla04] Team Fortress Classic Guide.
www.planethalflife.com/tfc/guide/classes.shtm, 2004.

[tfc04a] Team Fortress Classic Installation.
www.game-revolution.com/download/pc/action/half-life_team_-fortress.htm, 2004.

[tfc04b] Blufive's Pages.
www.gavncal.demon.co.uk/blufive/tfc-index.html, March, 2004.

[tfc04c] Expert TFC's TFC guide.
http://website.lineone.net/~chutney_boy/ExpertTFC/-Map_Guides/2_Forts/2_forts.html, March, 2004.

[tfc04d] PlanetHalfLife's TFC guide.
www.planethalflife.com/tfc/strategies/maps/2fort.shtm, March, 2004.

[TR04] Martin Guld Thomsen and Alex Vendelbo Ringgaard.
Project Homepage.
www.cs.auc.dk/~guld80/dat6.html, July 15, 2004.

[VJ04] Andreas S. Værge and Henrik Jarlskov.
Using Bayesian Networks for Modeling Computer Game Agents.
www.cs.auc.dk/library/cgi-bin/detail.cgi?id=1055315209, July 14, 2004.

[WD99] Gerhard WeiSS and Pierre Dillenbourg.
Collaborative learning. Cognitive and computational approaches.
Pergamon Press, 1999.

[WJ94] Michael J. Wooldridge and Nicholas R. Jennings.
Agent Theories, Architectures, and Languages: A Survey.
ECAI-Workshop on Agent Theories, Architectures and Languages, 1994.

[WJ04] Michael J. Wooldridge and Nicholas R. Jennings.
Intelligent Agents: Theory and Practice.
www.csc.liv.ac.uk/~mjw/pubs/ker95/ker95-html.html, March, 2004.

List of Figures

List of Tables

A TFC Classes

This appendix is a compilation of the official guide for TFC, which comes with the installation files[tfc04a], and the fan made guide from planethalflife's TFC section[pla04].

A.1 General Description

Scout - The perfect flag capturer

Scouts are the best flag capturers, so they are utterly essential in any map which involves a flag (maps with objectives like Capture the Flag or Territorial Control). Scouts can be effective even without firing a single shot, since their strength lies in their speed rather than in their combat skills. If you like moving twice as fast as everyone else, and screaming for an escort as you escape the enemy base with the Flag and the entire enemy team pursuing you, then the Scout is your class.

Sniper - The ultimate long-range fighter

The Sniper is the only class that can kill an enemy on the other side of the map with almost no warning at all. Be careful though, for if the enemy engages you at close range you're weaker than most of the other classes. If you like settling down into a good sniper position and methodically killing every enemy who exits his base with a single shot to the head, then the Sniper is for you.

Soldier - The best all-around fighter

The workhorse in any TF team, soldiers perform well at any task and specialize in none. If you like killing lots and lots of enemies in straight man-to-man combat, this is the class for you. Soldiers are a core class in any offensive or defensive squad.

Demoman - A lethal close-range fighter

Carrying more explosive weaponry than any other class, the Demoman clears whole rooms in seconds. He's essential on maps where the use of high explosives can destroy barriers, allowing access to more entrances into the enemy base. His remotely-detonable bombs make him a vicious defender in enclosed spaces. If you like watching the smoke clear and seeing bodies everywhere, the Demoman's your class.

Medic - The most indirectly lethal class in the game

A solid close-to-medium range fighter, the Medic moves swiftly and carries a gun that enables him to mow down any of the weaker classes without too much trouble. His real strength lies in his ability to heal himself and his teammates with his medikit. If you love seeing eternal gratitude in the eyes of your teammates, or can't aim at all and still want to be incredibly useful to your Team, go for the Medic class. You might not kill a lot of enemies, but the teammates whose lives you've saved sure will.

Heavy Weapons Guy - A walking tank

The name says it all. A slow moving, heavily armored beast toting an enormous assault cannon, the HW Guy can mow down any enemy in seconds. Not a lot of finesse in this one, since the cannon spits out so much fire you barely need to aim it. If you like picking a defensive position in a map and saying "No-one, and I mean no-one, is getting past here", then the HW Guy's the class you want.

Pyro - Set them on fire, and watch them burn

A great first line of defense, the Pyro can easily knock chunks out of the enemy with his flamethrower. He does not have much in the way of killing weaponry, but he can make sure no-one gets into the base without being set on fire, and being burned alive tends to upset people. If you like running into a room, firing wildly, and then watching enemies run around screaming as they roast, then the Pyro is perfect for you.

Spy - The perfect infiltrator and terrorist, all in one

A class who can disguise himself to look like anyone, the Spy can often walk into the enemy base, right past the defenders. Able to kill an enemy in a single hit with his knife, the Spy is someone you definitely don't want to turn your back on. After a few of their members have been knifed in the back, teams tend to get fairly paranoid. You know you're winning when they start to shoot each other for fear they're Spies. If you like matching your wits with the enemy instead of your reflexes, then the Spy is your class.

Engineer - The perfect remote defender

When your base needs some heavy defense, the Engineer comes to the rescue, building automatic sentry guns that track enemies and kill them. Able to repair his teammates' armor and build machines to dispense ammunition, the Engineer is often the core player in large defensive squads. If you need to defend your base against all those hardcore players out there with much better aim, the Engineer's the perfect class.

A.2 Statistics

Scout

Special: Displays the status of each team's flag on CTF maps
Speed: Very fast
Health: 75
Armor: 50
Weapons: Single-barrel shotgun, nail gun and crowbar
Grenades: 3 caltrop canisters and 3 concussion grenades
Abilities: Disarms detpacks set by enemy demomen by touching them; Unmasks spies by touching them

Sniper

Special: Toggles the sniper rifle zoom
Speed: medium
Health: 90
Armor: 50
Weapons: Nailgun, auto rifle, sniper rifle, and crowbar
Grenades: 2 hand grenades

Soldier

Special: Reloads your currently selected weapon
Speed: Slow
Health: 100
Armor: 200
Weapons: Single-barrel shotgun, double-barrel shotgun, rocket launcher and crowbar
Grenades: 4 hand grenades and 1 nail grenade

Demoman

Special: Detonates all your pipebombs
Speed: Medium

Health: 90
Armor: 100
Weapons: Single barrel shotgun, grenade launcher, pipebomb launcher, and crowbar
Grenades: 4 hand grenades and 4 MIRV
Abilities: Can set large explosive "detpacks" to clear new entrances to enemy base

Medic

Special: Selects the medkit
Speed: Fast
Health: 90
Armor: 100
Weapons: Medikit, single-barrel shotgun, double-barrel shotgun, and super nailgun
Grenades: 3 hand grenades and 2 concussion grenades
Abilities: An heal teammates with the medikit; Can automatically heals himself over time

Heavy Weapons Guy

Special: Selects the assault cannon
Speed: Very slow
Health: 100
Armor: 300
Weapons: Single-barrel shotgun, double-barrel shotgun, assault cannon, and crowbar
Grenades: 4 hand grenades and 1 MIRV grenade
Abilities: Doesn't get blown back as much by explosions

Pyro

Special: Selects the flamethrower
Speed: Medium
Health: 100
Armor: 150
Weapons: Single-barrel shotgun, flamethrower, incendiary cannon, and crowbar
Grenades: 1 hand grenade and 4 napalm
Abilities: Wears flame retardant armor, making him impossible to set on fire

Spy

Special: Shows the disguise/feign menu
Speed: Medium
Health: 90
Armor: 100
Weapons: Tranquilizer gun, double-barrel shotgun, nailgun, and knife
Grenades: 2 hand grenades and 2 hallucination gas grenades
Abilities: Can disguise himself to look like any class, team or both; Can feign death loudly or quietly; Unmasks other spies by touching them

Engineer

Special: Shows the build menu
Speed: Medium
Health: 80
Armor: 50
Weapons: Railgun, double-barrel shotgun, and wrench
Grenades: 2 hand grenades and 2 EMP grenades
Abilities: Can build automatic sentry guns; Can build ammunition and armor dispensers; Can repair teammates armor by beating them with the wrench; Can create ammunition

B Full Influence Diagram

Figure B.1: Full influence diagram.

C Test Results

C.1 Tests with One Defender

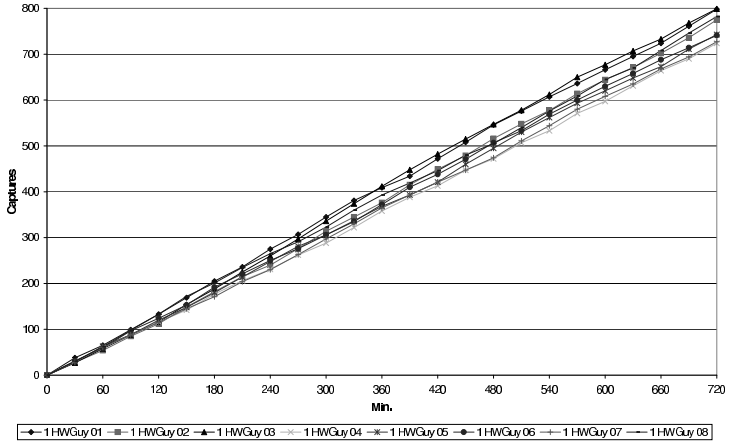


Figure C.1: 1 defender vs 3 attackers, test runs with random behavior.

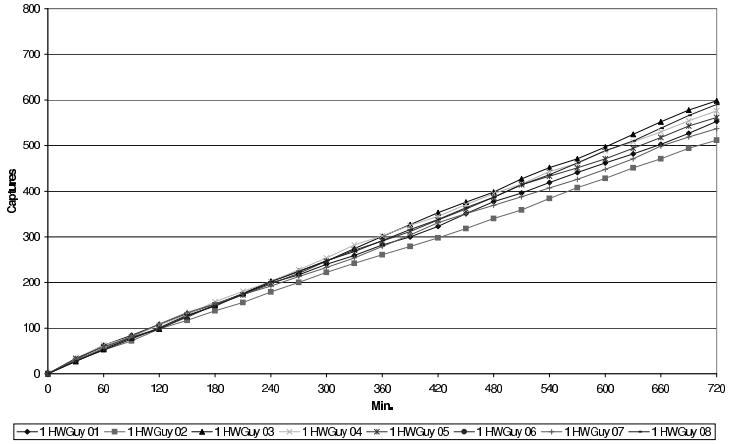


Figure C.2: 1 defender vs 3 attackers, test runs with model.

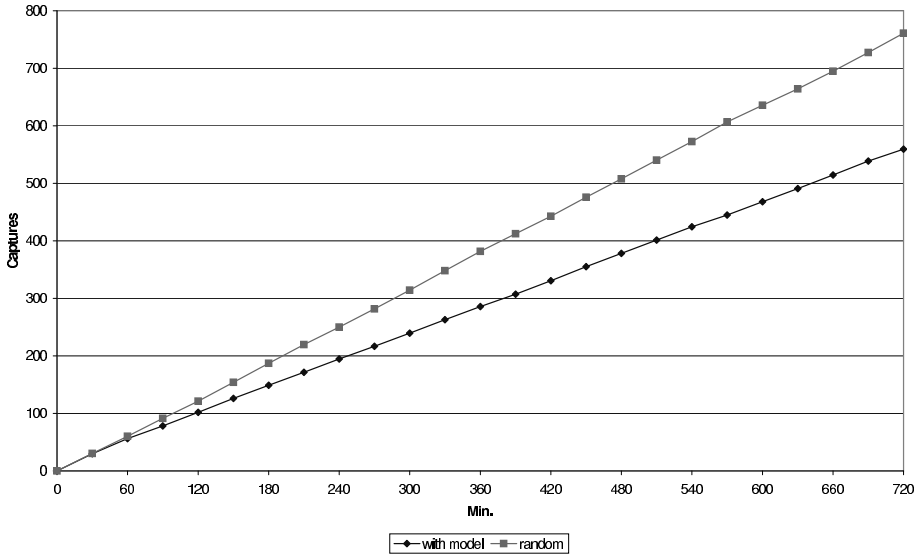


Figure C.3: 1 defender vs 3 attackers, average of the test runs with random behavior and with model.

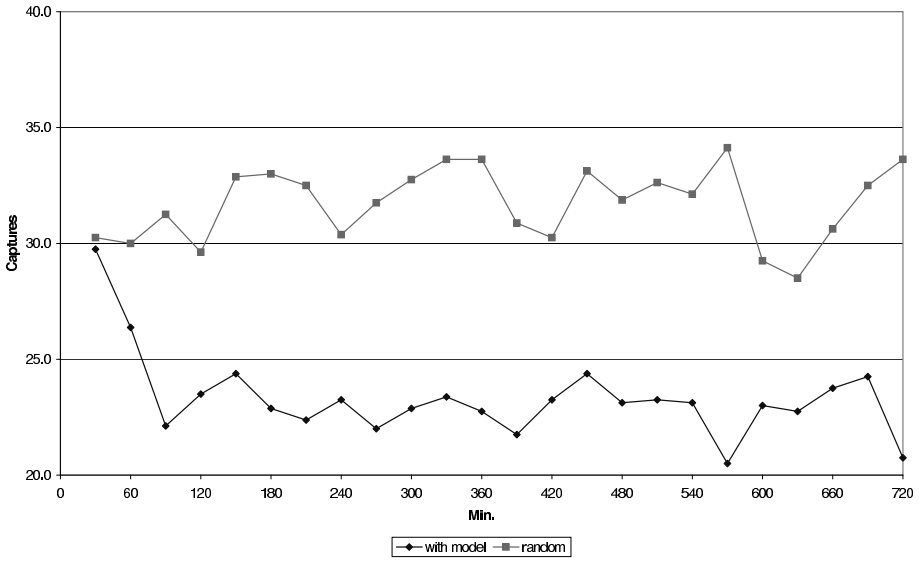


Figure C.4: 1 defender vs 3 attackers, average captures for each half hour with random behavior and with model.

C.2 Tests with Two Defenders

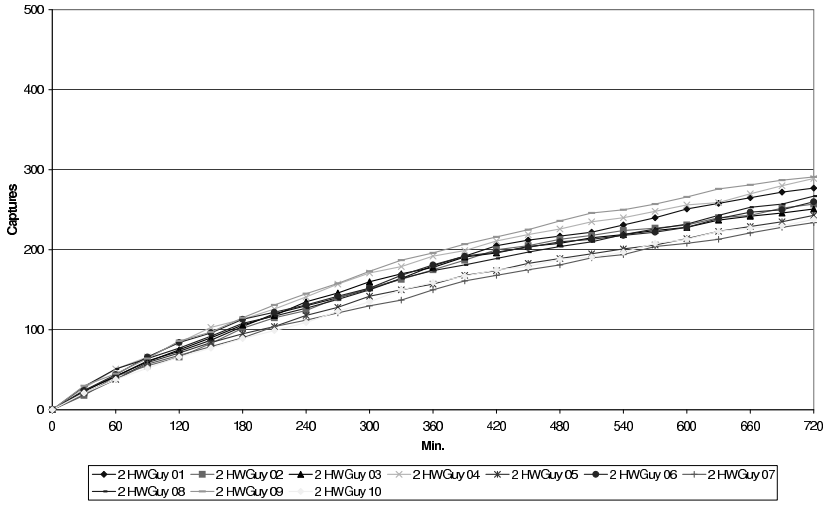


Figure C.5: 2 defenders vs 6 attackers, without prediction.

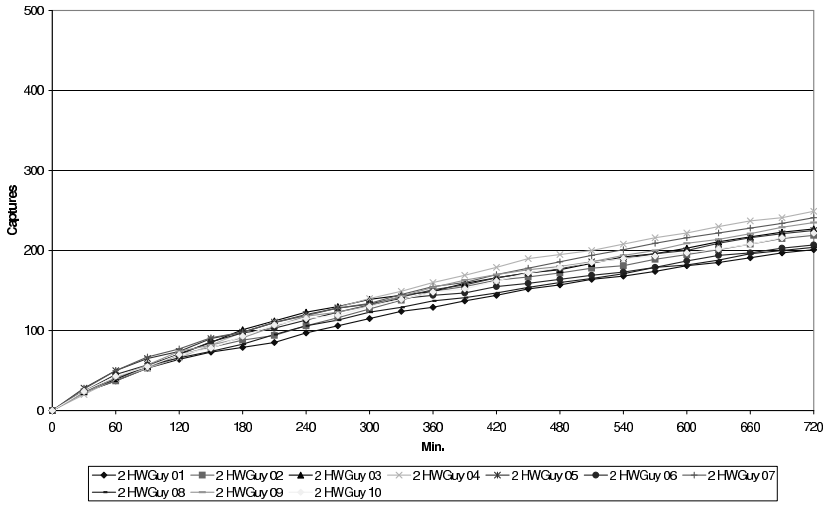


Figure C.6: 2 defenders vs 6 attackers, with prediction.

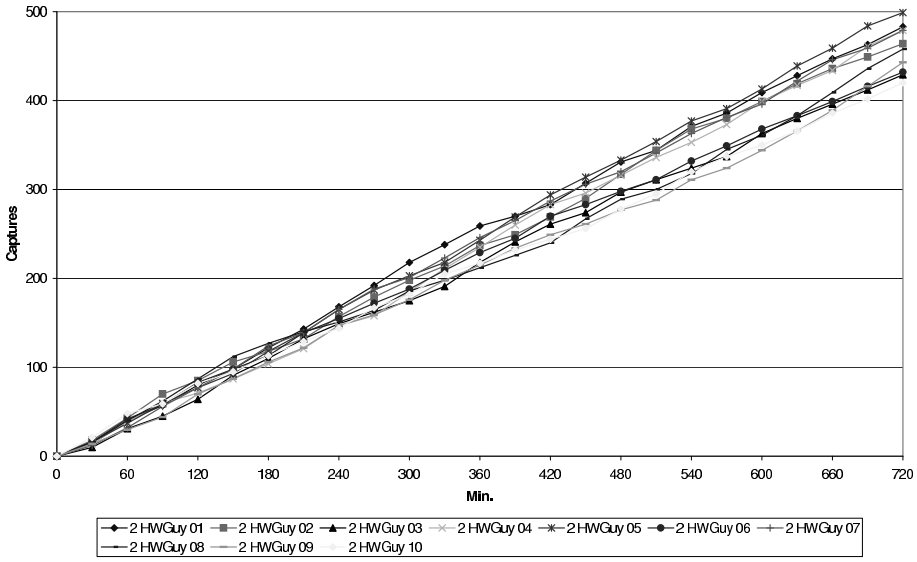


Figure C.7: 2 defenders vs 6 attackers, random.

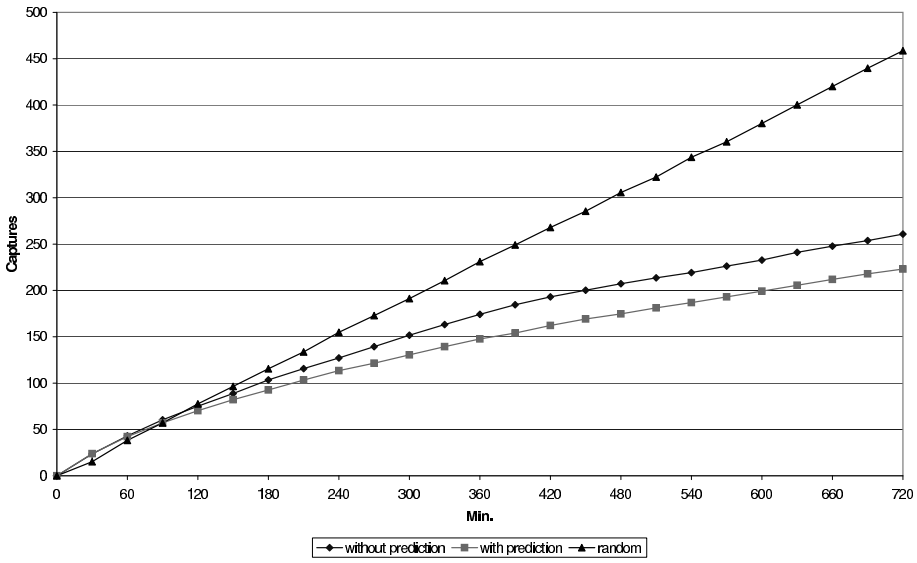


Figure C.8: 2 defenders vs 6 attackers, averages of the first three tests.

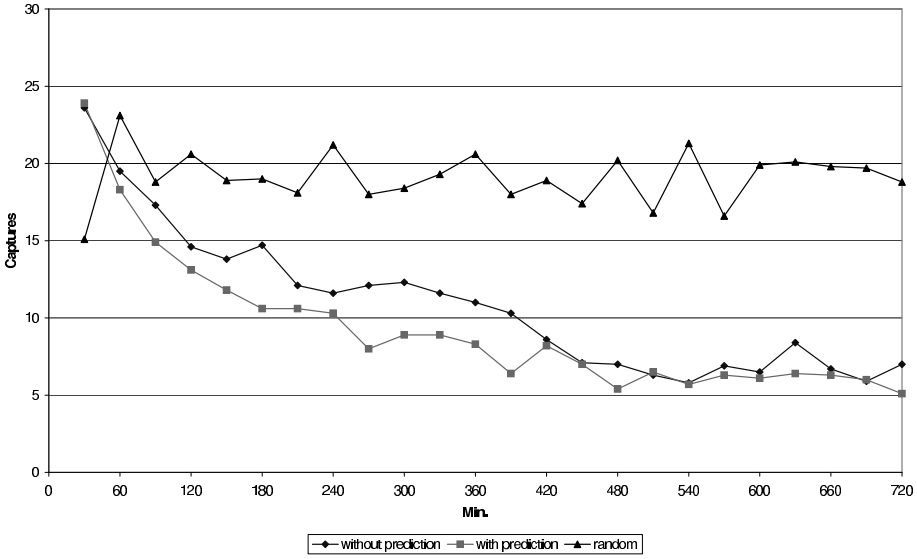


Figure C.9: 2 defenders vs 6 attackers, average captures for 30 minutes of the first three tests.

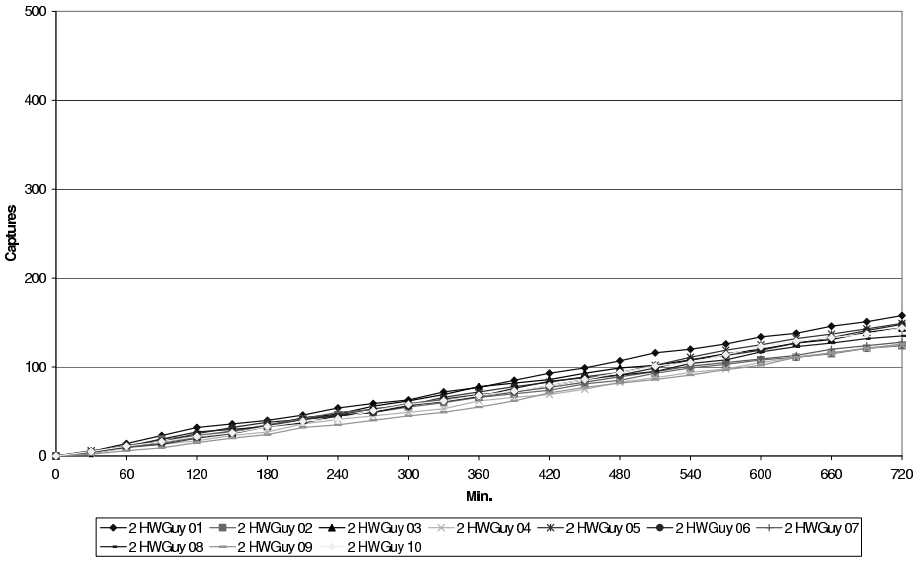


Figure C.10: 2 defenders vs 6 attackers, with trained influence diagram.

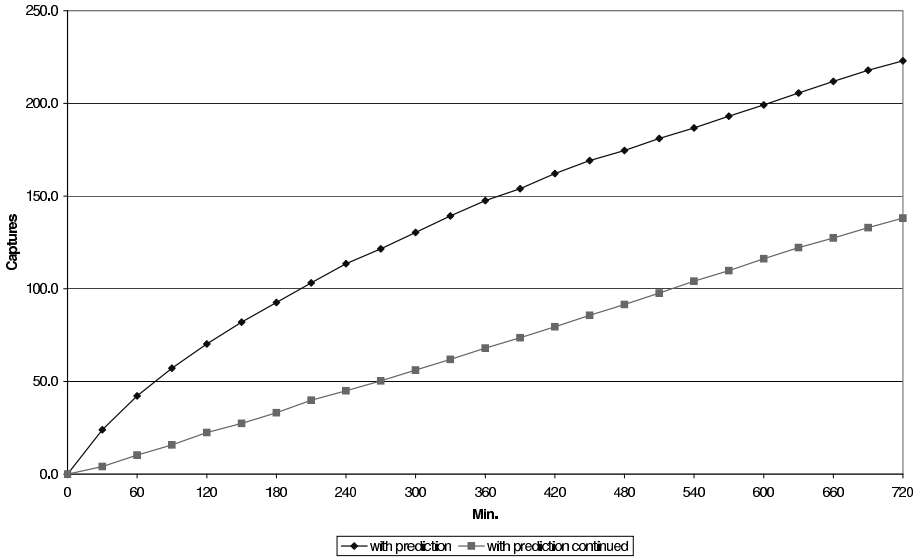


Figure C.11: 2 defenders vs 6 attackers, averages of the tests with trained and untrained influence diagram.

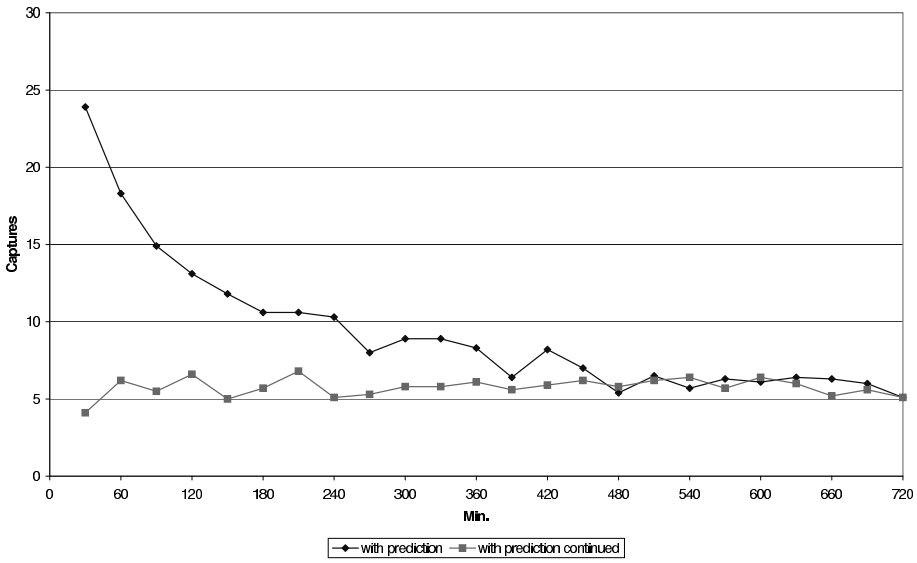


Figure C.12: 2 defenders vs 6 attackers, average captures for 30 minutes of the tests with trained and untrained influence diagram.

C.3 Tests with Four Defenders

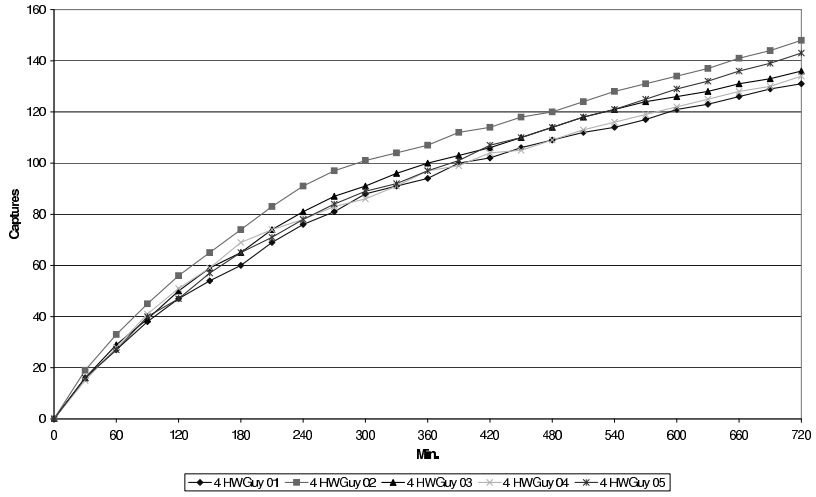


Figure C.13: 4 defenders vs 25 attackers, without prediction.

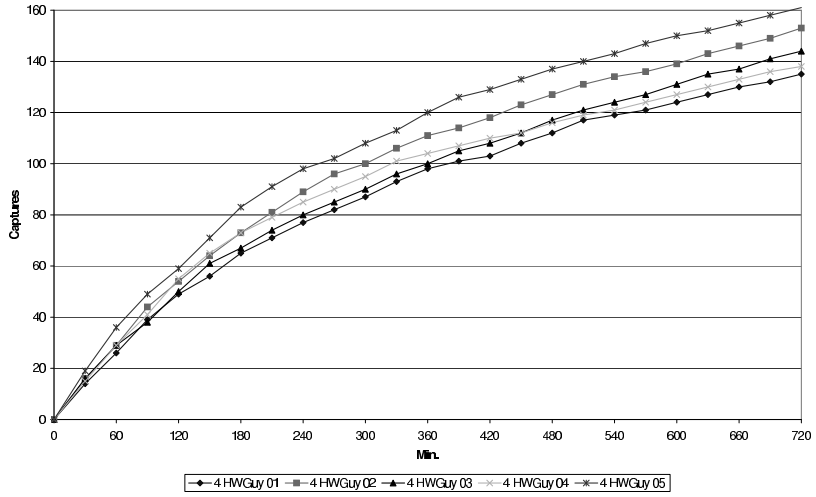


Figure C.14: 4 defenders vs 25 attackers, without prediction and no case est.

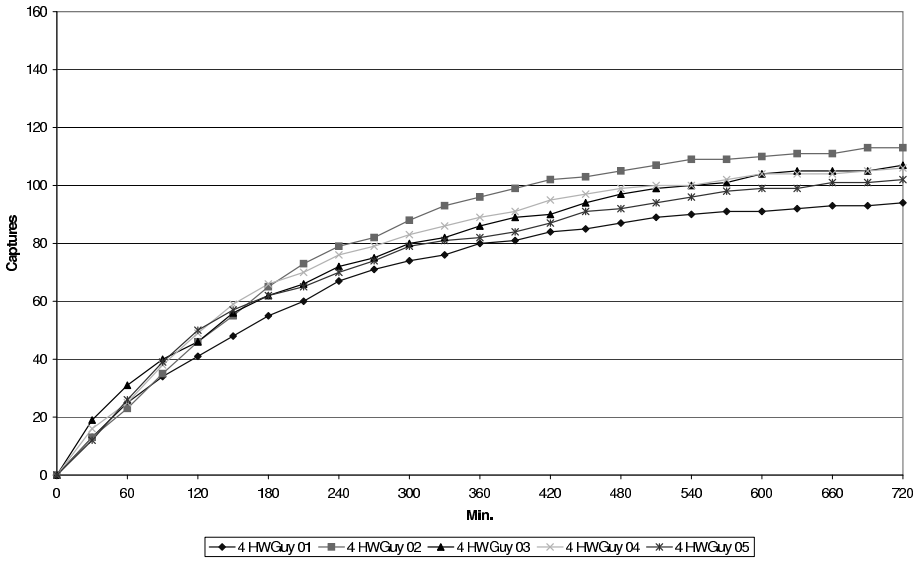


Figure C.15: 4 defenders vs 25 attackers, with prediction.

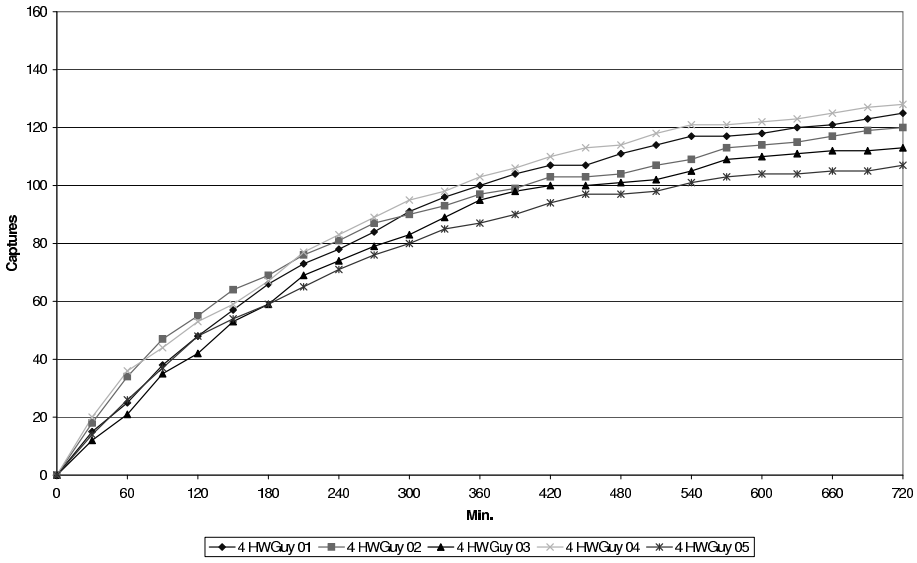


Figure C.16: 4 defenders vs 25 attackers, with prediction and no case est.

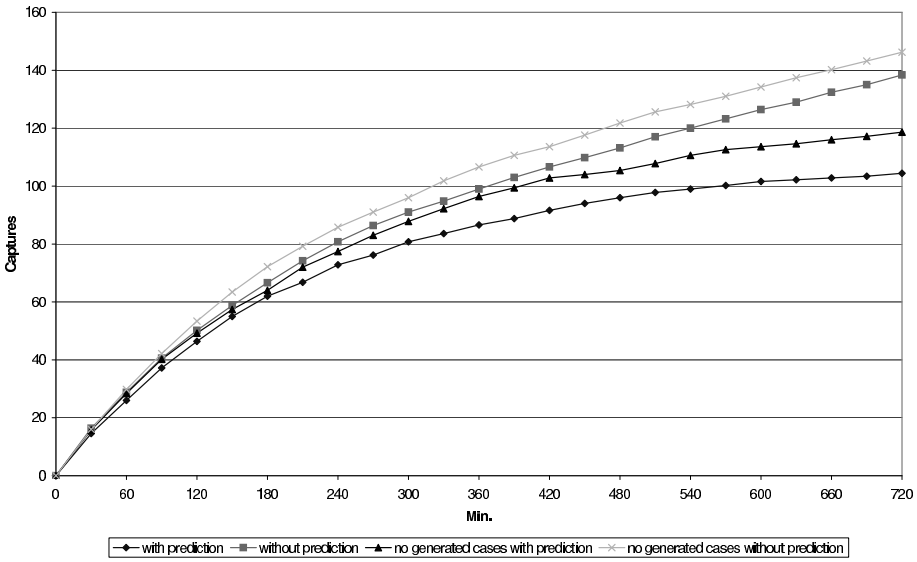


Figure C.17: 4 defenders vs 25 attackers, averages of the four tests.

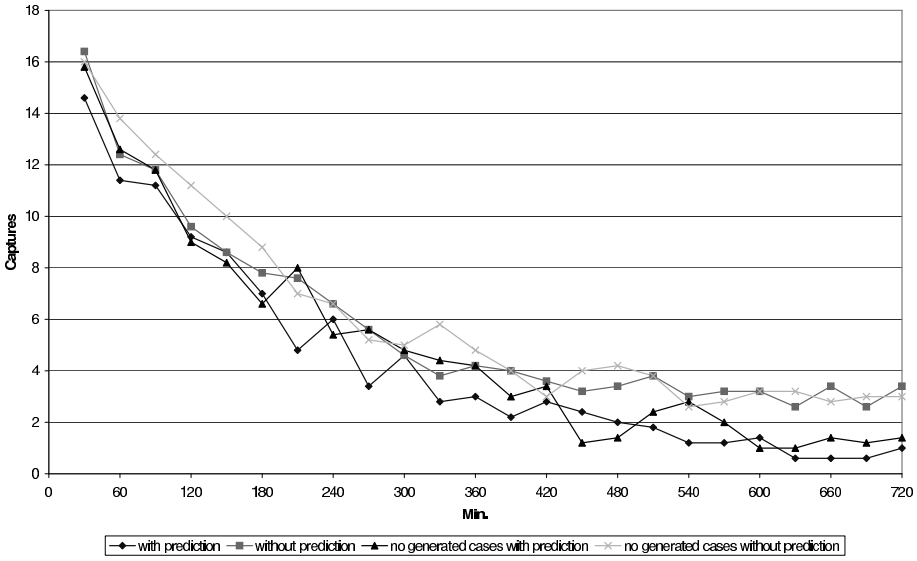


Figure C.18: 4 defenders vs 25 attackers, average captures for 30 minutes of tests.

C.4 Tests with Four Defenders - Revised

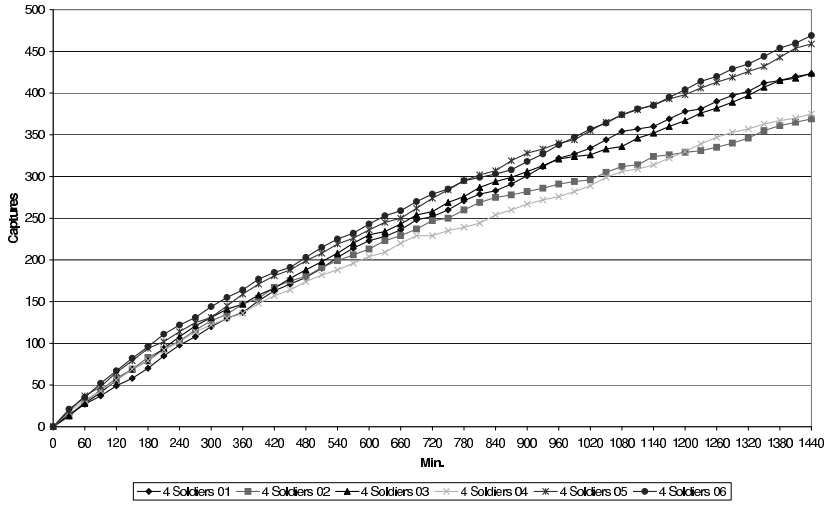


Figure C.19: 4 defenders vs 25 attackers, without prediction.

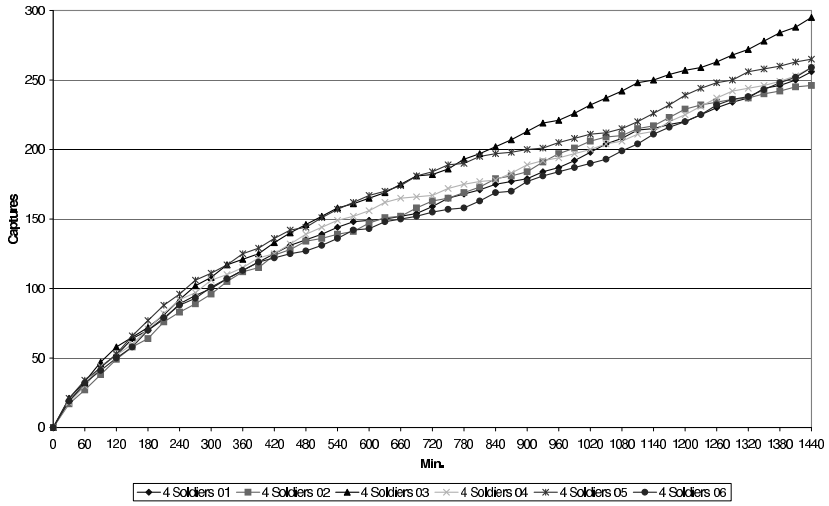


Figure C.20: 4 defenders vs 25 attackers, with prediction.

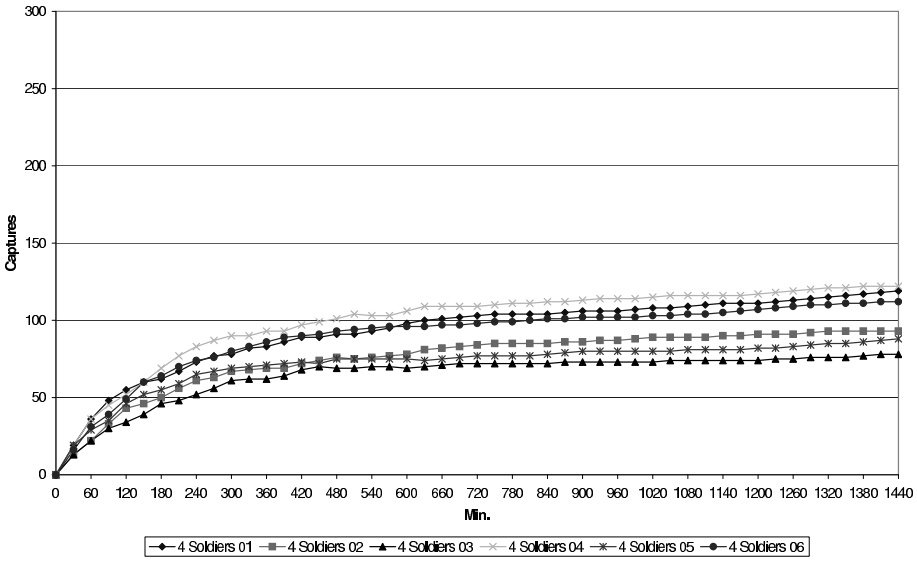


Figure C.21: 4 defenders vs 25 attackers, 100% communication.

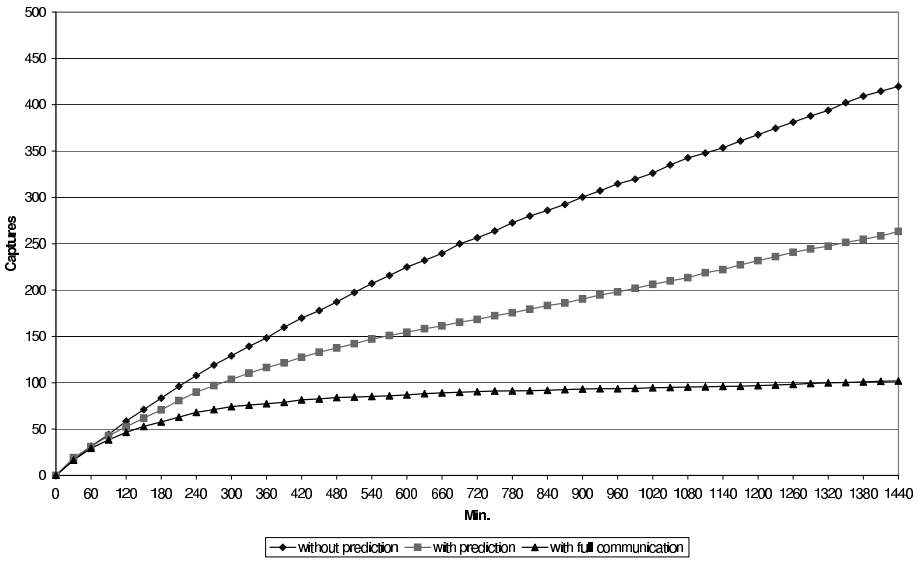


Figure C.22: 4 defenders vs 25 attackers, averages of the first three tests.

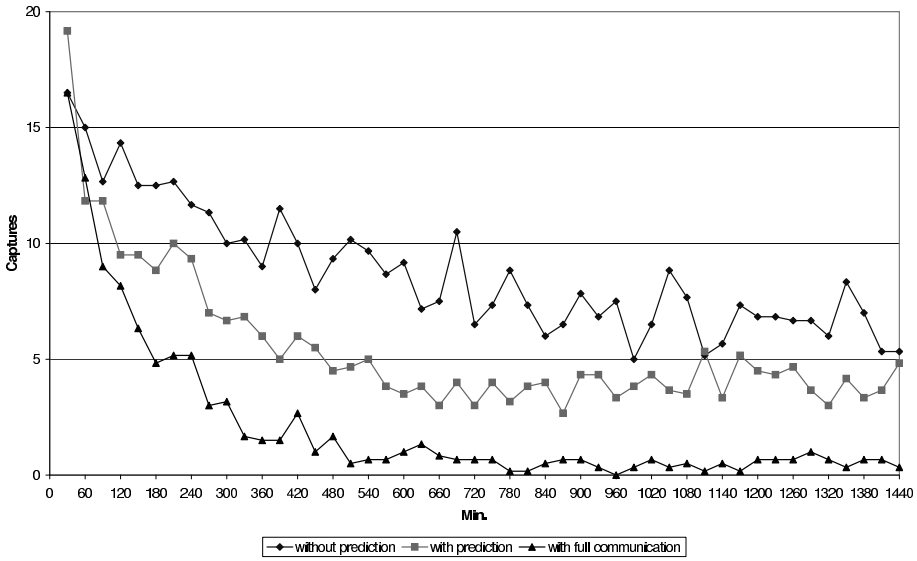


Figure C.23: 4 defenders vs 25 attackers, average captures for 30 minutes of the first three tests.

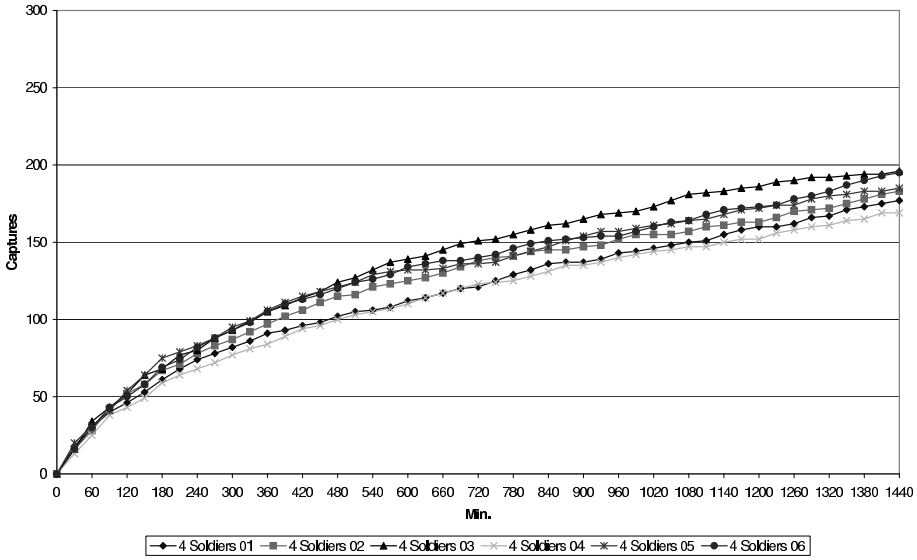


Figure C.24: 4 defenders vs 25 attackers, 50% random communication.

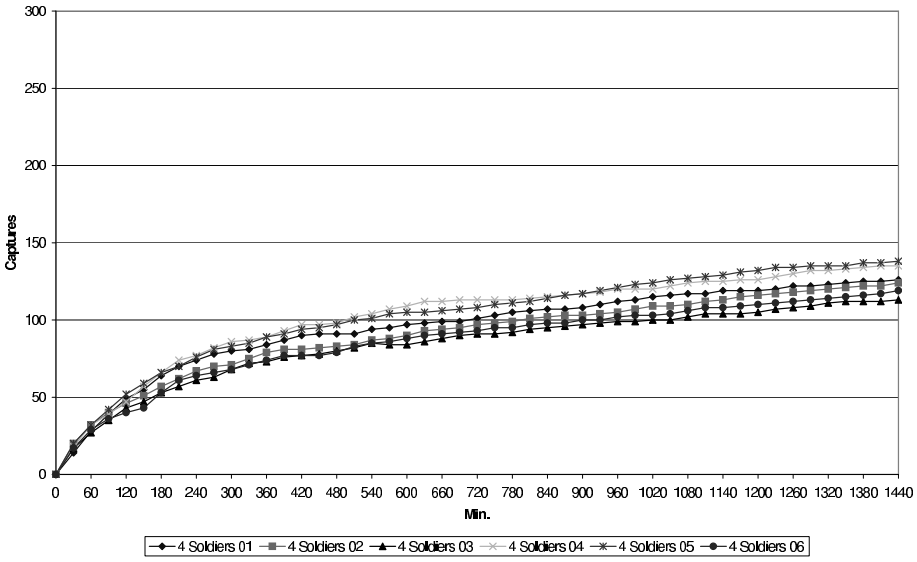


Figure C.25: 4 defenders vs 25 attackers, 48% request.

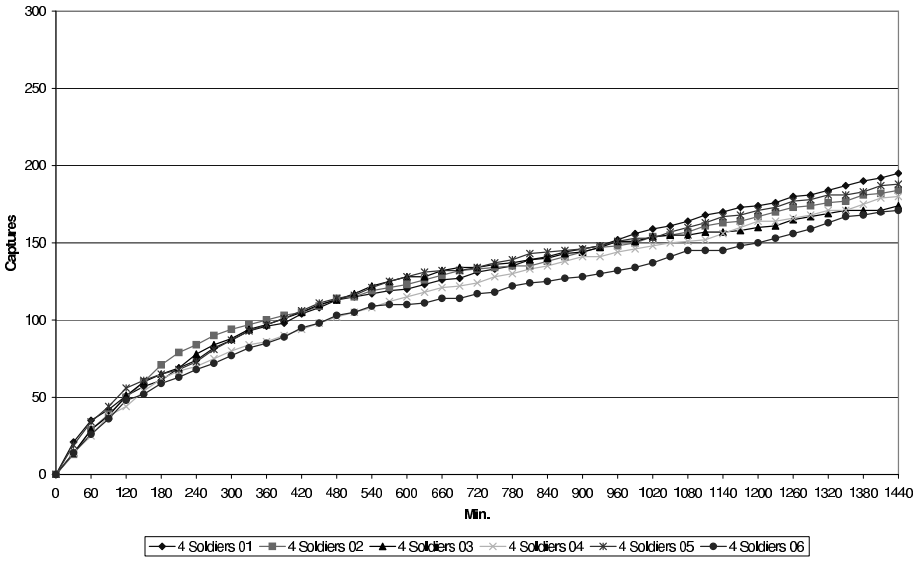


Figure C.26: 4 defenders vs 25 attackers, 53% answer.

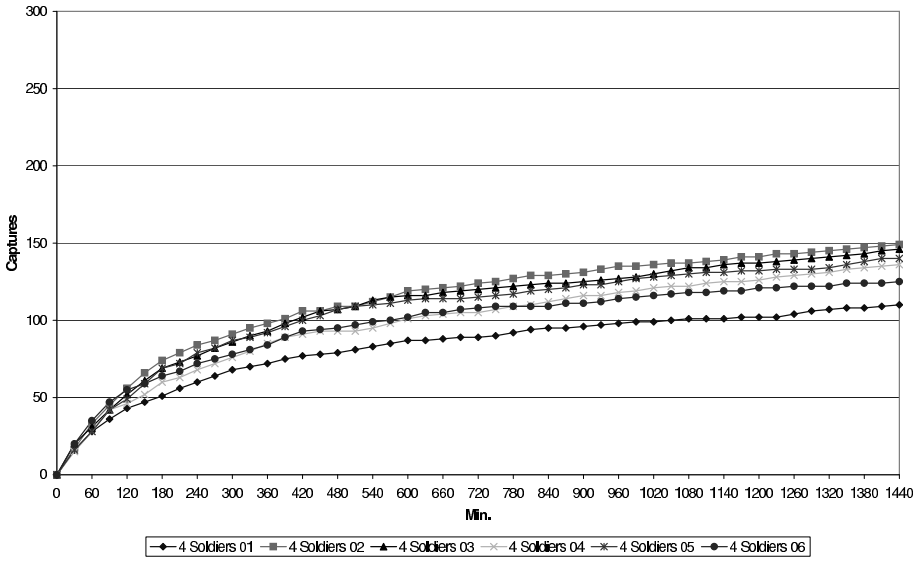


Figure C.27: 4 defenders vs 25 attackers, 47% VoC.

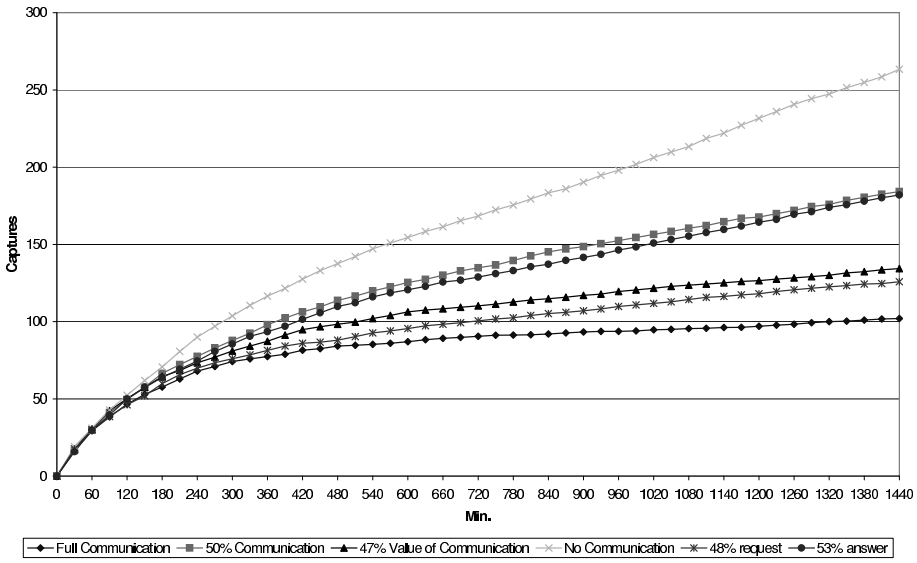


Figure C.28: 4 defenders vs 25 attackers, averages of the six VoC tests.

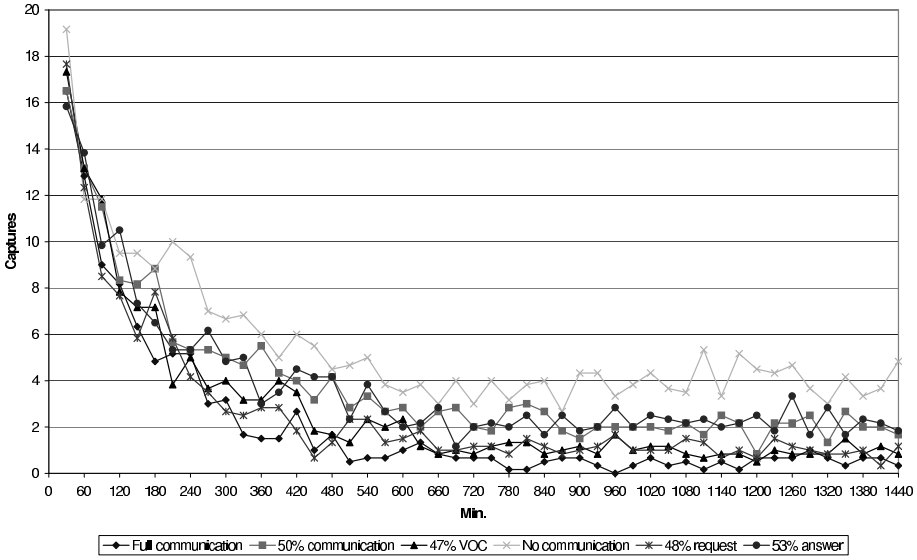


Figure C.29: 4 defenders vs 25 attackers, average captures for 30 minutes of the six VoC tests. A section of this graph showing only the last 14 hours can be seen below.

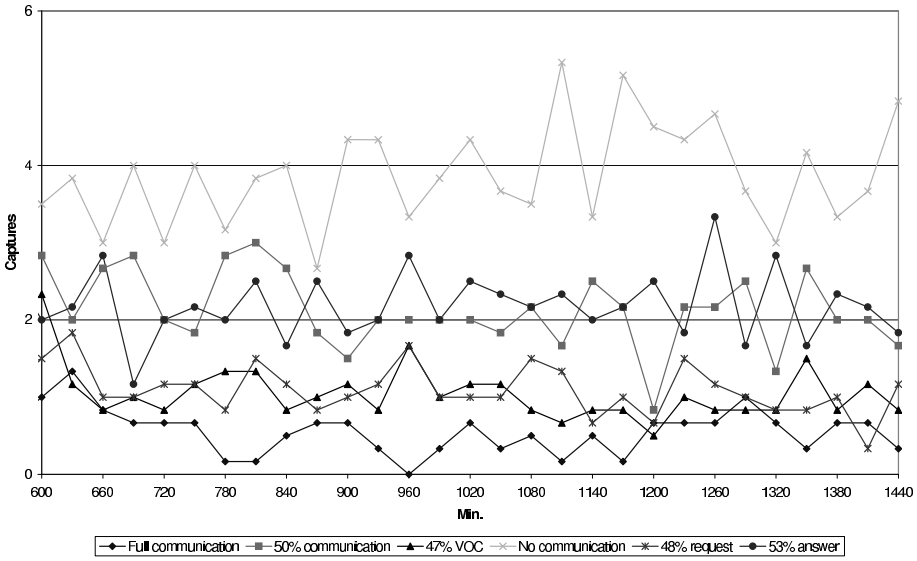


Figure C.30: A section of the above graph showing the last 14 hours in greater detail.