AALBORG UNIVERSITY

Department of Computer Science

KDE master group

RBOT - An Intelligent agent in the Unreal game environment

4th Semester project (final project)

Rimantas Benetis

10 July 2002

Aalborg University

Department of Computer Science

Title:

RBOT (Intelligent agent in Unreal game environment)

Subject:

Decision Support Systems

Project group:

E1-121

Participants:

Rimantas Benetis

Supervisor:

Tomas Kocka

Time of Writing:

28 April 2002 – 12 July 2002

Copies:

4

Pages:

90

Synopsis:

This project presents an intelligent agent called Rbot. The Rbot is an Unreal Tournament game opponent, which uses the Bayesian Networks for making decisions. The agent is implemented using the Java language. Two design approaches of creating influence diagrams presented. The performance results against a built-in Unreal Tournament bot and human player are presented. Conclusions about usability and performance of Bayesian networks in real-time game environment presented.

Preface

This project is submitted for a course in Decision Support Systems at Department of Computer Science at the faculty of Engineering and Science, July 2002 by Knowledge and Engineering master group. The main purpose of this project was to explore the use of Bayesian networks in real-time games. The game chosen was Unreal Tournament (UT). In order to test Bayesian networks in UT game an agent utilizing Bayesian networks was created.

Rimantas Benetis

Abstract

The real time environment can be very complex and it could be impossible to predict all of the possible states of the environment. In order to act effectively agents in such environments need to be able to adapt to the changes. This project addresses agents in the real time environments. This project makes a contribution to the field of Decision Support Systems.
First the project introduces the environment, which was created for an agent to be tested in. The whole platform for testing agents in real time environment is presented.
Second the description of the first semester work is presented. It introduces the first approach to create the influence diagram for an agent. Then it shows the performance results of the agent, which uses influence diagram created by using first approach. The problems that could arise by using such approach are presented.
Third the project introduces a second approach of creating influence diagrams. It gives a description how the agent can be created and what problems might arise and how they can be solved. This project also includes the detailed description of the agent that was created for an Unreal Tournament game environment.
The test results prove that the influence diagrams can be used efficiently in real time environment. This is proved by applying the methods described in a real time environment.

Acknowledgements

I would like to thank many people for their support, encouragement during my years as a graduate student in Aalborg University.
First of all this project took a great deal of time not only for myself but also for my two supervisors Jiří Vomlel (previous semester) and Tomas Kocka (last semester). They both helped me with the research and gave me many advices and helped this project to be as it is now.
I would also like to thank my censors that also gave me valuable comments on my previous projects.

Contents

Table of Figures

Index of Tables

Chapter 1 Introduction

During the past decade the games evolved together with the hardware. They got more complicated and so had to evolve the computer opponents. In order to make computer opponent more like a human player the artificial intelligence was used. As in the old games the behavior of the computer opponents could be coded only using the if-then rules then the current games needs them to adapt to the play of the human player to provide the interesting game play. Several kinds of game types can be pointed out (you can find more detailed description in [1]):

- Turn based strategy games. The example of such a game could be the checkers or any other board game where the moves are made in turns and the game rules are known for every player. The alpha-beta search is usually used in such games.

- Real time strategy (RTS). Such games are more complicated then the turn based strategy as the movements are done while time elapses and their order is not known to the opponent. Also the resource management and a path finding algorithms are often used in such games.

- First person shooter (FPS). These games try to model the environment itself. Only the goal is given and there are no rules how the players reach their goal. The AI in such games has to be very advanced to be able to compete with humans.

We will concentrate on the FPS type of game as it gives a very wide area of the research. The game is very dynamic and the changes are made in real time. The first games of such type (Wolfenstein, Doom, etc.) did not have any complicated AI and the difficulty of the game depended only on the number of opponents and their damage caused to you. The newer games such as the Unreal Tournament (UT) and Quake 3 already tries to provide computer opponents (bots), which are capable to mimic the human behavior. They are able to go for the powerups, get better weapons and even try to ambush you. Although even having such complicated scripts bots is not a match to the human player. The problem is that they do not try to evaluate their moves and do not adapt to a changing environment. This means that after a human player discovers the weak spot of a bot it will be able to successfully exploit it for the rest of the time. The ability of a bot to judge its own actions would enable it to change its strategy to the different one and prevent human player exploiting the weak points. The adaptability could also help in the areas where users are able to change the environment as the script creators are not able to foresee all of the possible changes that could be done to the environment and code them into the scripts. There is a good description of different games and what kind of AI they use at [2].

So the task of this project is to explore the Bayesian networks (BN), which are a very powerful tool for decision-making, and try to apply them in the domain of FPS. The goal also would be to suggest some kind of general approach when addressing such kind of problems. In order to prove that the BN does help to improve the efficiency of the bot the tests has to be carried out. To be able perform those tests the testing platform has to be done.

Chapter 2 Related Work

This chapter contains the description of the related work that has been done in the similar area of research to this project. First we give the description of a Quakebot [3] project. Then we present the project, which shows that the opponent can be modeled by using the influence diagrams.

2.1 *Soar Quakebot*

The Quakebot uses Soar engine (Soar is a general cognitive architecture for developing systems that exhibit intelligent behavior) for making and executing decisions – selecting the next thing the system should do and then executing it. The basic objects in Soar are called operators. An operator consists of primitive actions (the actions that can not be divided into smaller actions, such as move, turn, shoot or wait), internal actions (remembering something), or more abstract goals to be achieved (get-item) that must dynamically be decomposed into primitive actions. Those actions are implemented by multiple if-then rules, which are fired parallel and sequence to implement single operator.

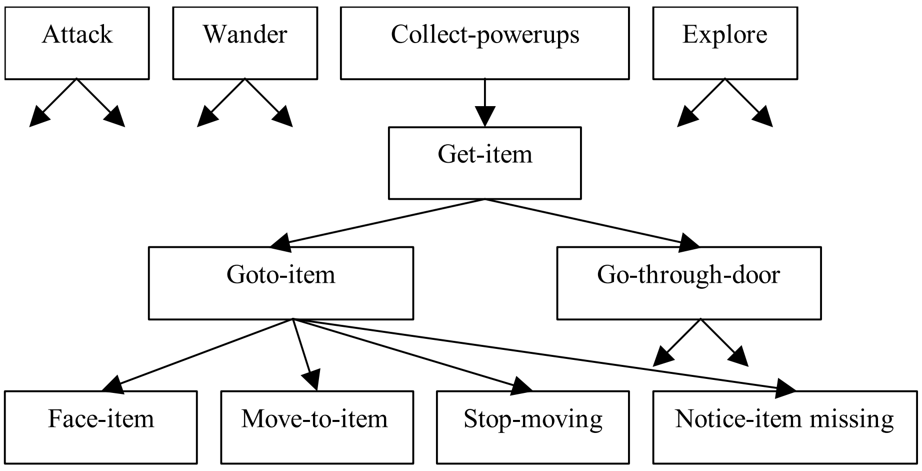


Figure 1 Partial operator hierarchy

Figure 1 shows organization of operators that were decomposed into primitive actions.

The Soar engine does not use any predefined ordering to determine which operator should be selected or applied. Instead all the operators that match defined rules are fired in parallel to change working memory by either adding or removing declarative structures.

2.1.1 Quakebot prediction

The Quakebots approach to prediction is to create an internal representation that mimics enemy's internal state based on bots observation of the enemy. Then the bot uses the information about enemy and tries to make a decision what it would do if it were in enemy's place.

Using this approach the bot may get either useful information or it could get uncertain information. Useful information may be that the enemy bot will pass through a certain point and the uncertainty appears when several such points exist (the Quekebot will not know which point the enemy will pick to pass through).

Prediction is used to set ambush or deny enemy power-up. The bot did not try to predict enemy continuously as it takes too much time and it may happen that by the time the prediction is made it becomes useless. Bot did not predict when no information about the enemy was available or it already knew what to do. Quakebot starts prediction only when it sees enemy bot and enemy is facing away from Quakebot. When enemy bot faces Quakebot then Quakebot stops its predictions so that is not caught napping (making a prediction).

The Quakebot creates an internal representation of enemy's state and then it uses its own knowledge what it would do if it were in the enemy's state to predict enemy's actions. It is assumed that Quakebot's and enemy's sets of actions are the same. A prediction terminates when a bot comes to a point where uncertainty appears (for example, several exits exists in a room and the Quakebot does not know which one will the enemy choose).

2.1.2 Quakebot prediction application

The prediction is useful only when there is ability to use it (the information received from a prediction may put bot in a better situation then the opponent). Quakebot concentrates on getting to a certain room first, setting an ambush and getting to a specific powerup first. To make these predictions three operators are used: hunt, ambush and deny-powerups. When a prediction is made that the enemy will be in a certain room where Quakebot can get faster the hunt mode is applied and the bot is sent to the correct room. If it predicts that enemy will exit a certain door the ambush is selected and it move a bot towards that door to set an ambush (Quakebot waits around the corner and tries to shoot enemy in the back). Deny-powerups is selected when enemy is predicted to pick up a powerup that the bot can get first.

Quakebot also makes use of Soars engines mechanism called chunking. Chunking creates rules that test the aspects of the situation that were relevant during the generation of a result. In other words it tries to remember which conditions should hold in order to get the same result. It means that when the Quakebot gets into the same situation where calculations were already performed and saved the prediction will be calculated instantly.

One of the biggest weaknesses of Quakebot is that it assumes that enemy would behave the same way as Quakebot would under equal conditions. In a real game scenario such an assumption is not true as the enemy may take different actions. In such a case it might be a good idea to adapt to the enemy's actions.

Another point is to use the probabilities in cases where there is an uncertainty (when several exits exist in same room). The fact that an enemy bot can be predicting and adapting to your own bots actions should also be taken into account. In such cases we need to make decisions under uncertainty because enemy state and strategy are not fully disclosed or observed. One of the possibilities is to use Bayesian networks as they have good performance under uncertainties.

The approach of introducing some prior strategies as hunt, ambush will also be used in Rbot. The strategies will be used in a prediction and in a decision of Rbot.

2.2 *Learning Models of Other Agents Using Influence Diagrams*

Some work was done with using influence diagrams in multi-agent system (MAS). The MAS is the system where several agents try to reach some defined goal. The knowledge about the other agent might not be known or it can change during the time. This means that the agents, which are capable to adapt to the changes, has the advantage against the agents with fixed strategies. The paper "Learning Models of Other Agents Using Influence Diagrams" [4] makes use of the influence diagram to model other agents. They use influence diagram to model agents in following scenario: two bases defend against missile attack. There are two incoming missiles. Each of defense bases can launch only one interceptor. The purpose is to minimize the damage of incoming missiles when there is no communication between two defense bases. The influence diagram in such system helps the agent to predict another agent's actions and choose the action, which will minimize the damage inflicted by a missile. They also give an influence diagram, which can be used for selecting a best action (Figure 1).

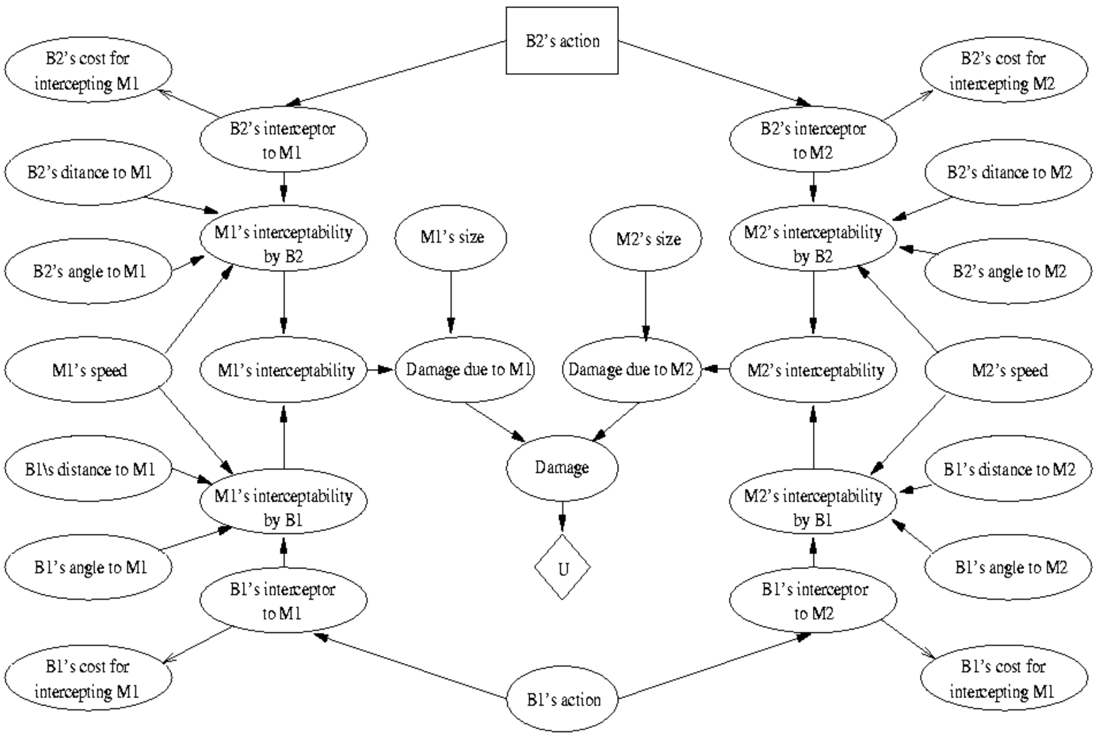


Figure 2 Decision model of defence unit B2

The authors also present what kind of possible problems we might encounter when environment changes and the influence diagram must be updated three main inaccuracies might occur:

- The utility function of another agent might change. Then we only need to learn the new utility function of another agent.

- The probability distributions in the model might be inaccurate. We must learn the new probability distributions from the observations.

- The structure of the influence might be incorrect. We must learn the new structure that fits the observation data.

2.3 *Other projects*

There are also some other projects, which proposes different learning techniques in a real time environment. One of them is using information from an expert to learn the actions and the conditions under which those actions should be carried out [5]. Another project is an agent interaction in real time environment when communication is uncertain [6]. The learning process in that paper is described as learning by stages. First we learn the easy personal tasks (performing some movement) then we learn more complicated tasks (as a teamwork).

Chapter 3 Theory

3.1 *Bayesian networks*

First I would like to give a short introduction to Bayesian networks. You can find a good description in [7].

Definition of Bayesian network:

Bayesian network consists of the following.

- A set of variables and a set of directed edges between variables.
- Each variable has a finite set of mutually exclusive states.
- The variables together with directed edges form a directed acyclic graph (DAG). (A directed graph is acyclic if there is no directed path $A_1 \rightarrow \cdots \rightarrow A_n$ s.t. $A_1 = A_n$).
- To each variable A with parents $B_1,\ldots,B_n$ there is attached the potential table $P(A \mid B_1,\ldots,B_n)$.

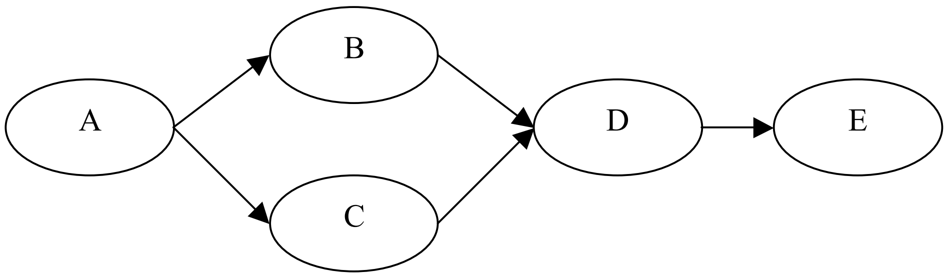


Figure 3 Simple Bayesian network example

In Figure 3 a simple example of Bayesian network is presented. It has nodes *A*, *B, C, D*, *E* and it has relationships that *A* influences node *B* and *C*, *B* and *C* influences *D*, *D* influences *E*. Each node may represent some property from the real world and may have several mutually exclusive states. A directed edge may represent a relationship between nodes. The relation is expressed as a conditional probability table assigned to a node. In Figure 3 conditional probability tables are assigned to nodes *B*, *C*, *D* and *E*. The list of probability tables for model in Figure 3 is:

$P(A), P(B \mid A), P(C \mid A), P(D \mid B,C), P(E \mid D)$.

An example of conditional probability table for node *B* is:

$P(B\|A)$	a_1	…	a_i
b_1	$P(b_1\|a_1)$	…	$P(b_1\|a_i)$
...	…	…	…
b_j	$P(b_j\|a_1)$	…	$P(b_i\|a_i)$

Table 1 An example conditional probability table

Probability table contains values of probability $P(b_j \mid a_i)$ for all conditions of states where a_i denotes the *i*-th state of node *A* and b_j represents *j*-th state of node *B*. If a node (node *A* in Figure 3) does not have any parents a prior probability table is assigned to it with values $P(a_i)$.

Let us consider situation, which is reflected by *B*, *C*, *D* and *E* nodes. As we can see the nodes *B* and *C* influences *D* which then influences node *E*. So we can say that nodes *B* and *C* influence the node *E* only through node *D*. But if a state of node *D* is known then the nodes *B* and *C* no longer have influence on node *E*. In such case we say that node *E* is *d-separated* from *B* and *C* given *D*. The other situation is nodes A, B and C and it is called diverging connection. Nodes *B* and *C* will be *d-separated* only when node *A* is instantiated (when we know the state of variable than we say it is instantiated). And the nodes *B*, *C* and *D* for a situation called converging connection. In converging connection nodes *B* and *C* are *d-separated* when there is no evidence for node *D* or its descendants else they are not *d-separated*.

Definition of d-separation:

Two variables *A* and *B* in a causal network are *d-separated* if for all paths between *A* and *B* there is intermediate variable *V* such that either

- The connection is serial or diverging and *V* is instantiated

Or

- The connection is converging and neither *V* nor any of *V*'s descendants have received evidence

The importance of *d-separation* is that it reduces the number of calculations that has to be carried out in a Bayesian network.

A Bayesian network is constructed to reflect some part of real world prior to any observations. Information, which can be derived from the network, is called belief and it is a probability that a certain node is in a certain state. Having a network we may observe some events, which could change our beliefs. These observations are called evidence. Evidence may be entered into the Bayesian network and propagated (which means recalculation of the probabilities. The new probabilities of variables given evidence are calculated).

3.2 *Fractional updating*

A way to model uncertainty about a conditional probability table is to modify the table every time new evidence is received. Fractional updating [7] can be used for this purpose.

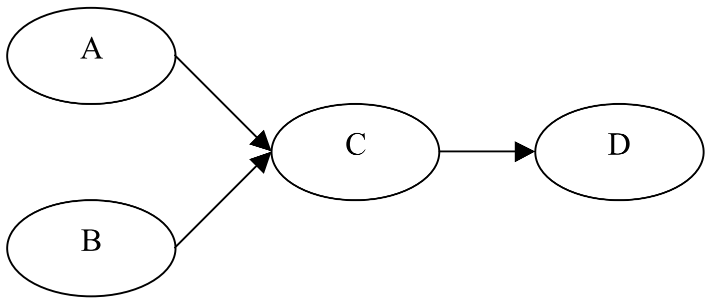


Figure 4 Node C with parents A and B

Suppose we have the network given in Figure 4. Node C has conditional probability distribution $P(C \mid A, B)$. First assumption we have to make is that the uncertainty about probability values of different variables is independent (global independence). And that uncertainty about probability values of the same node for different parent configurations is independent (local independence). Since the local independence assumption is made, we may consider each parent configuration (a_i,b_j) for node C independently. If node C has k states, then the current probability distribution would be

$$P(C \mid a_i, b_j) = (x_1, x_2, ..., x_k)$$

Let s be a positive number, which expresses our certainty about the distribution, it is called *sample size*. Then the probability distribution can be written as

$$P(C \mid a_i, b_j) = \left(\frac{n_1}{s}, \frac{n_2}{s}, ..., \frac{n_k}{s} \right)$$

here $x_i = \frac{n_i}{s}, i = 1,...,k$.

If we get a new case e with $A = a_i$, $B = b_i$ and C = c_1. Then $n_1 := n_1 + 1$ and $s := s + 1$, and probabilities are updated as follows

$$x_1 = \frac{(n_1 + 1)}{s + 1}; x_2 = \frac{n_2}{s + 1}; \cdots; x_k = \frac{n_k}{s + 1}; \text{ (1)}$$

If we get a case with $A = a_i$, $B = b_i$, but for C we only have distribution $P(C \mid e) = P(C \mid a_i, b_j, e) = (y_1, \ldots, y_k)$ then we can not work with integer counts and we update $n_k := n_k + y_k$ and $s := s + 1$ then we get

$$x_k = \frac{n_k + y_k}{s + 1} \text{ (2)}$$

In general we get a case with $P(a_i, b_j \mid e) = z$. Then $s := s + z$. To update the counts we use the distribution $P(C \mid a_i, b_j, e) = (y_1, y_2, ..., y_k)$. As the sample size is only increased with *z* we take $n_k := n_k + zy_k$ and we get

$$x_k = \frac{n_i + zy_i}{s + z} \quad (3)$$

By using this approach and receiving evidence our network adapts to the evidence received. After some time new evidence with different finding will not change our belief much (as we can see from formula (3) the larger is the count *s* the larger must be *n* to cause a noticeable change.). In order to evade past experience influencing our model too much we use *fading factor*. The fading factor can take values from interval (0,1]. Before updating our beliefs we multiply sample size *s* and case count n_i by fading factor *q* so that we get

$$s := sq + 1; n_i := n_i q + zy_i$$

The fading factor is useful where environment is dynamic and changes are fast. It is not useful where environment is static and we do not get incorrect data.

3.3 *Decision diagrams*

A Bayesian network represents a model for a part of the world. It consists of certain variables and relationships between them. Sometimes we want Bayesian networks to help us make some decisions. To represent decision problem graphically we must extend Bayesian network with new types of nodes.

1. Decision nodes. It has rectangle shape.

2. Utility nodes. It has diamond shape.

Decision node has states that represent actions that we can take. Utility node has outcomes (utilities) of a given parent configuration.

Let us present an example of extended Bayesian network.

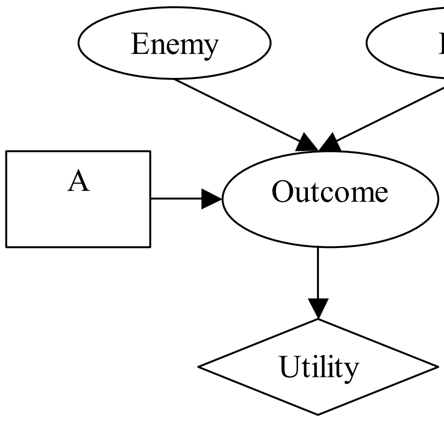


Figure 5 Simple decision diagram example

As we can see from Figure 5 there are two new nodes. One of them is decision node (labeled A) and second one is utility node (labeled Utility). Now suppose Figure 5 represents bots reasoning. The bot can observe its own state and partially observe enemy state and then it should choose the action (let us say bot has two options either to *retreat* or *fight*), which would maximize its survival probability. The utility node represents possible outcome in real values (let us say that if bot survives its utility is 1 if it dies then it is 0). Having this information we can calculate the expected utility of action a_i using the following formula:

$$EU(a_i) = \sum_{Outcome} U(Outcme)P(Outcome \mid a_i, evidence)$$

After evaluating expected utilities for all possible actions that bot can take we choose an action, which yields the highest utility.

3.4 *Decision strategies*

A classical way of representing decision scenarios with several decisions is a tree. Let us give an example:

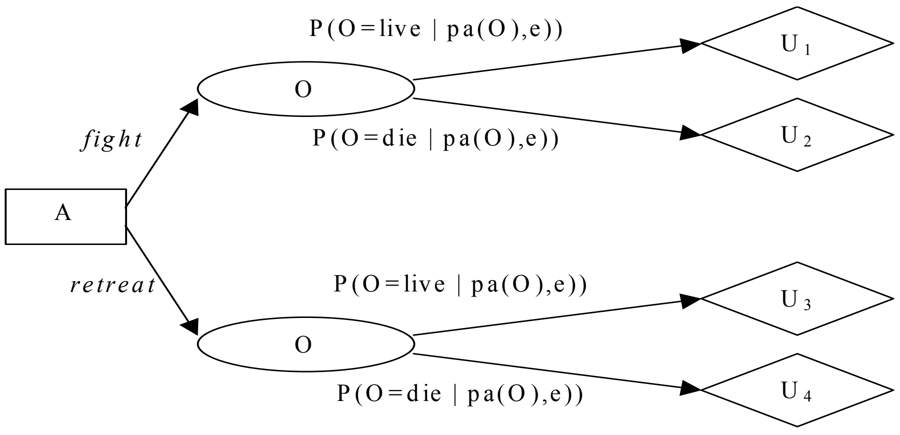


Figure 6 An example of a tree describing a simple decision strategy

In Figure 6 a decision strategy of a previous example is presented (the variable Outcome is abbreviated O). As we can see here we can take only two actions *fight* or *retreat*. EU(fight) should present the expected utility of action *fight*. That is:

$$EU(fight) = U(live)P(O = live \mid A = fight, e) + U(die)P(O = die \mid A = Fight, e)$$

The *e* in this formula is the evidence that may change our belief of Outcome. In Figure 5 the variables Enemy and Bot may change probability of Outcome so *e* represents evidence received from those variables.

To better understand the way the calculation of decision strategy we will use an example.

Let us assume that tables for Figure 5 are

P(Outcome|Action,e)	*fight*	*retreat*
Live	0.65	0.4
Die	0.35	0.6

Table 2 Conditional probability table

U(Outcome)	
live	1
die	0

Table 3 Utility values

The decision strategy is solved by "rolling back". We start with the nodes, which has only leaves as their children. If node *A* is a chance node, the expected utility for *A* is calculated:

$$EU(A) = \sum_{C \in children(A)} U(C)p(A \to C)$$

Here $p(A \to C)$ denotes probability of a link from node *A* to node *C*. And *C* is a child of A. And $U(C)$ is a utility value attached to a child *C*.

If the node is a decision node *D*: each child of *D* has a (expected) utility attached, then we choose the child with maximal expected utility and attach the value to *D*.

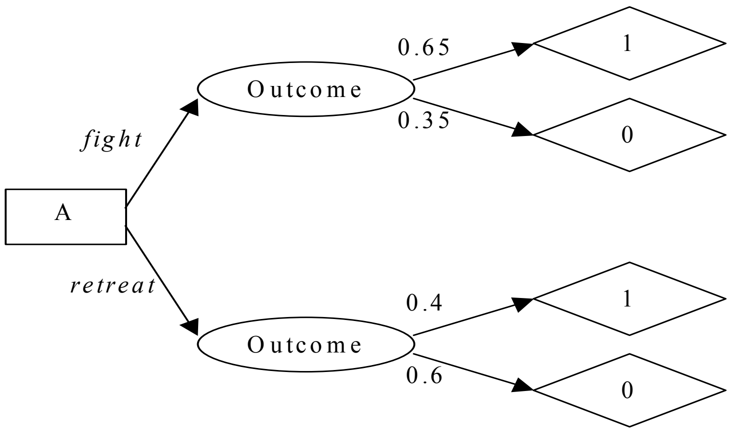


Figure 7 tree representing decision strategy with values entered

After having a tree and parameters for nodes we can start calculating from bottom up.

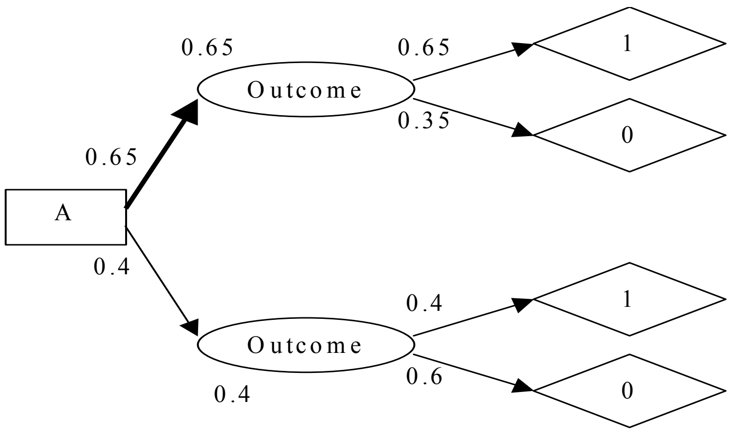


Figure 8 Results when solving a tree from Figure 7

Because the tree in Figure 7 is very simple only few calculations has to be done. After having results we should pick the path from tree, which yields highest expected utility. The path in Figure 8 is highlighted and it yields the highest expected utility of 0.65.

There is also possibility to have several decision nodes in our decision model. An example of such model is presented in Figure 9 (truncated tree branches are marked with dots). In Figure 9 the bot models following situation. A Bot noticed an enemy bot but enemy it is too far to observe its position vector (where bot is facing) so our bot must make a decision if it wants to approach or stay at current location. The decision it makes will influence the correctness of the observation of an enemy. After the observation is done the bot must choose the action to take against the enemy. After all actions are done the outcome can be observed.

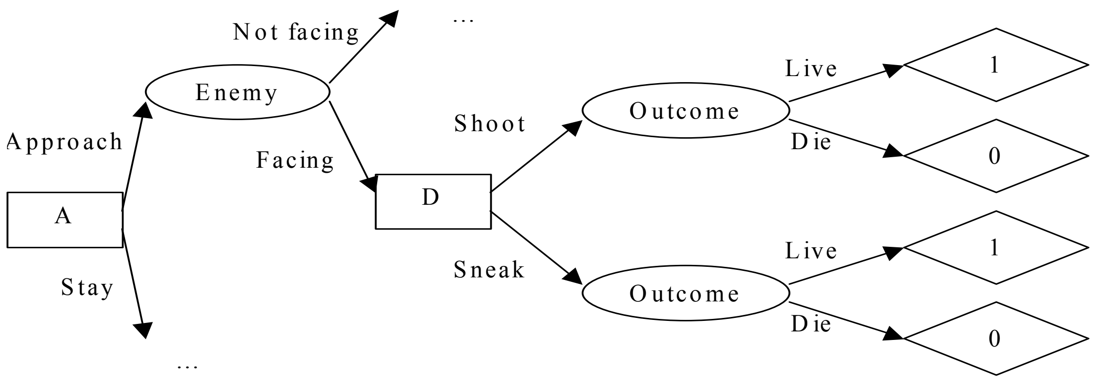


Figure 9 Example decision diagram with two decisions

3.5 *Influence diagrams*

Some of the decision scenarios can form a symmetrical tree for some decision strategy. That is the decision order must be the same in all paths. Sometimes some information is received prior of making a decision. To represent which observations can be made before a decision we extend the graph with information links. In similar way we extend the graph with precedence link (we add a link between decisions were the edge would be directed from earlier decision to later one). The resulting graph is called *influence diagram* and an example is shown in Figure 10. Influence diagram is a compact representation of decision strategy for symmetric decision scenarios.

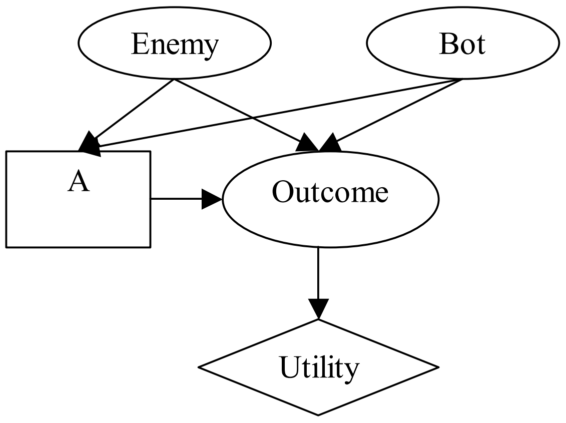


Figure 10 An extended example of Figure 5 (information links added)

The links from Enemy and Bot to A means that information about Bot and Enemy are known before taking decision A and are called information links.

An influence diagram may have several decision nodes. An example of influence diagram with two decision nodes is presented in Figure 11.

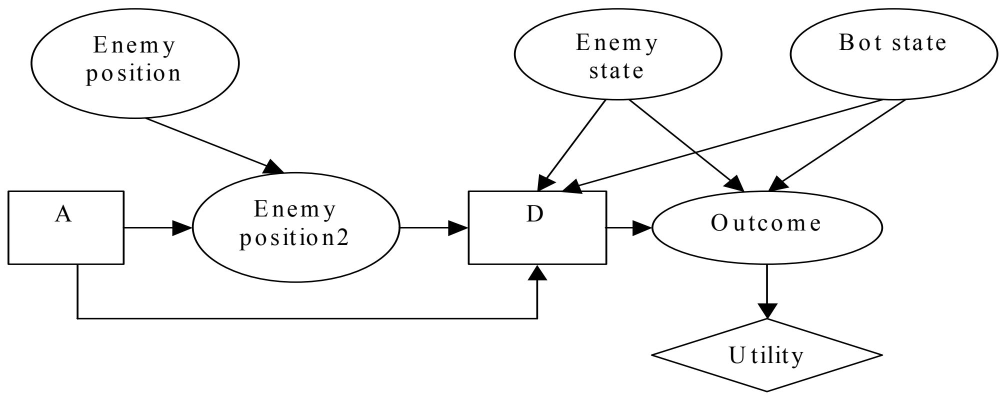


Figure 11 Example of a simple influence diagram with two actions

The formal definition of influence diagrams is presented below. This definition is taken from [7].

Syntax

An *Influence diagram* consists of a directed acyclic graph over chance nodes, decision nodes, and utility nodes with the following structural properties

- there is a directed path comprising all decision nodes,
- the utility nodes have no children.

For the quantitative specification we require that

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

Definitions:

- A *policy* for a decision D_i is a mapping σ_i, which for any configuration of the past of D_i yields a decision for D_i. That is

  $$\sigma_i(I_0, D_1, \ldots, D_{i-1}, I_{i-1}) \in D_i$$

  here I_0 is a set of chance nodes observed prior to any decision, I_i is a set of chance nodes observed after D_i is taken and before D_{i+1} is taken, and D_i is a decision node.

- A *strategy* for an influence diagram is a set of policies, one for each decision.

- A *solution* to an influence diagram is a strategy maximizing the expected utility.

There may be a need to find a strategy while time elapses. In such cases an influence diagram with several time slices must be created. In Figure 12 the example of an influence diagram with three time slices is presented. In the example S stands for state, O for observation, A for action and U for utility. In this example a bot may observe its enemies state and then take some actions that may change the state of an enemy. Those actions are made while time elapses and are relevant for future.

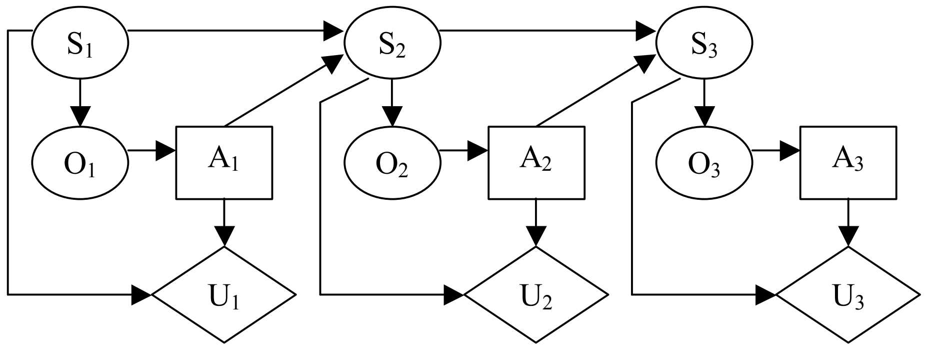


Figure 12 Example of influence diagram with three time slices

The calculation of influence diagram is similar to the calculation of decision strategy. In principle we can unfold the influence diagram out to a tree representing decision strategy and then use a technique for calculating those trees.

3.6 *Parent divorcing*

Sometimes by creating a model we may get a model similar to the one in Figure 13. It may also contain more parents for node Y. If we would need to specify parameters of $P(Y \mid X1, X2, X3, X4, X5)$ we would have to gather a lot of knowledge about this dependence. You might also use a database to extract the parameters but then you will need a lot of cases describing this relationship.

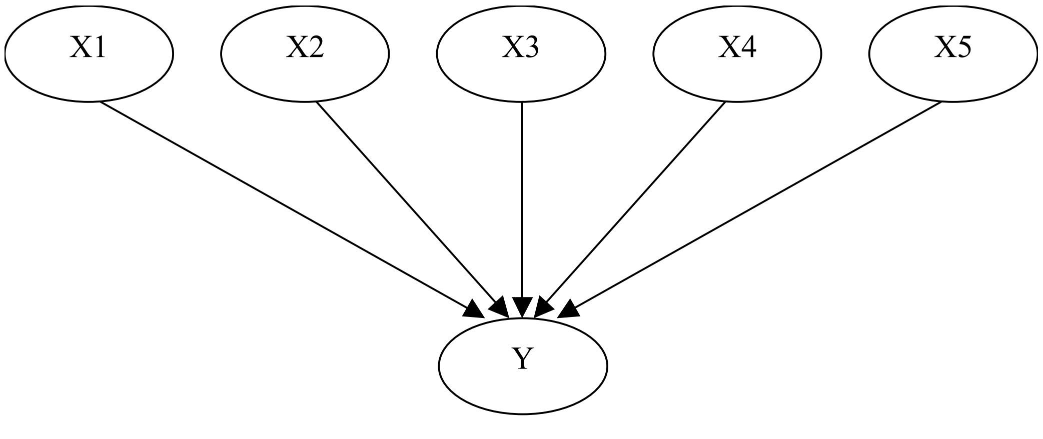


Figure 13 Network without divorcing

Suppose each of the nodes in Figure 13 has 3 states then the size of conditional probability table of a node Y becomes $3^6 = 729$. It becomes for expert an impossible task to specify such a probability table. Suppose we take distribution from database then we need a database to have around 7000 cases. To handle such problems we *divorce* parents.

Parent divorcing is possible only in some cases when we can encode the same probability distribution $P(Y \mid X1, X2, X3, X4, X5)$ by the other model in Figure 14.

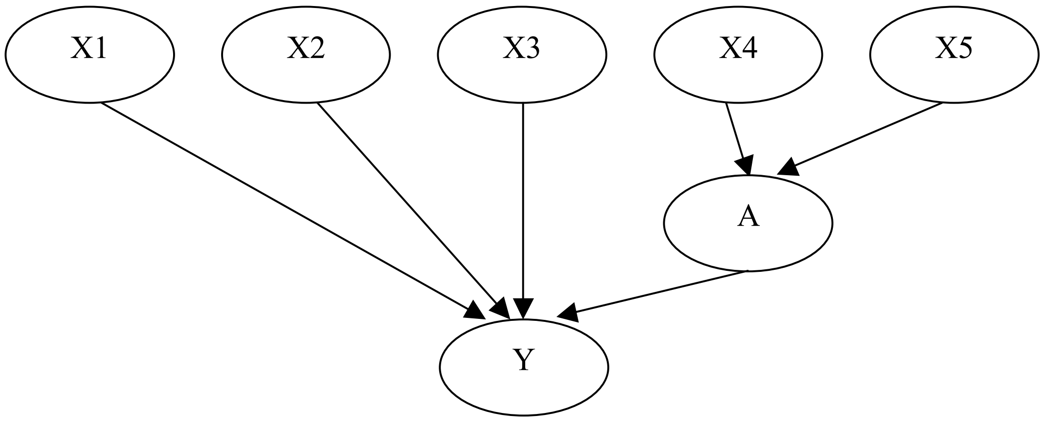


Figure 14 Network with parents X4 and X5 divorced

In this model we divorced parents X4 and X5 of a node Y and introduced a new (hidden) node A. This model can always encode the same $P(Y \mid X1, X2, X3, X4, X5)$ if the $|A| = |X4| \cdot |X5|$ (cardinality of $|A|$ is same as the product of $|X4|$ and $|X5|$ cardinalities). In such case the $P(Y \mid X1, X2, X3, A)$ will still have the same size as $P(Y \mid X1, X2, X3, X4, X5)$. However in some cases the lower cardinality of |A| can be enough. This means that we can use less states in node A and reduce the size of conditional probability table of a node Y.

This method was described in [7]. It is often used in the context of noisy OR, AND or similar functions where it reduces the cardinality of |A| to 2. It works best in cases where this technique can be applied many times and it would result in a graph where each node has at most two parents. Then the Figure 13 could be changed into Figure 15.

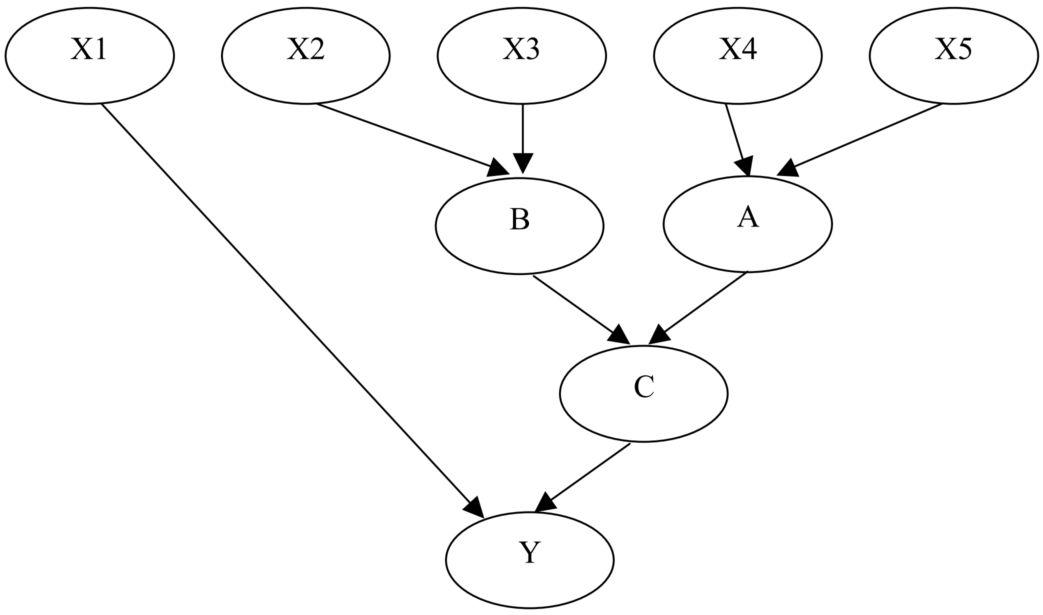


Figure 15 Network with 3 hidden nodes

In such case (if all the nodes has 3 states) the number of configurations for which the expert must give (or must be learned from database) a probability distribution is reduced to 27 + 27 + 27 + 27 = 108 which is much smaller then the previous one of 729.

3.7 *Structure learning*

Sometimes when creating the structure we might already posses some test data. In order to test some properties of data or prove some dependencies we might want to find these properties or dependencies from the data we already posses. For such cases structure learning is used. This approach enables us to learn the structure that best represents the given data. Two main approaches are used for structure learning. One of them is by using scoring function and another is by testing the independencies in data.

The first approach is to use a scoring function, which evaluates the structure (how well it fits the data given). To find the best structure we must try to evaluate different kinds of the structures with the same evaluation function. The structure with the best score describes the given data best. Usually methods that reduce the search space are used [8].

The second approach is to construct the network. One way is to use the PC algorithm described in [9]. This algorithm is used in Hugin API, which is used in this project.

Both of these approaches are NP complete problems. And it is proved that both of them would find the best solution given that the data is consistent and all of the independencies can be recognized from the data.

3.7.1 PC Algorithm

The PC algorithm is very similar to the SGS [9]. This algorithm takes the data as an input and then produces the structure, which reflects the data given. The algorithm can be divided into three steps:

1. Produce the complete graph on vertex set given by data.
2. "Thin" the graph by testing for d-separation.
3. Orient the graph.

Suppose we have a data received from graph (Figure 16 (a)). If we run PC algorithm following actions will be carried out.

In the first step we make a complete graph. That is the graph should contain the edges from any vertex to any other (Figure 16 (b)).

In the next test we must test for d-separation of the variables. The procedure starts "thinning" the graph by removing edges with zero order conditional independence, then first order conditional independence and so on. The set of variables conditioned on need only to be a subset of the set of variables adjacent to one or the other of variables conditioned. In Figure 16 (b) the following independence test should be carried out (an order of independence means the size of conditioning set. 0 order means that the conditioning set is empty):

0 order:
$A \perp\!\!\!\perp B \backslash \{\emptyset\}$
$A \perp\!\!\!\perp C \backslash \{\emptyset\}$
$B \perp\!\!\!\perp C \backslash \{\emptyset\}$

After zero order independence tests we will not be able to remove any edges so the graph will stay the same.

1st order:
$A \perp\!\!\!\perp C \backslash \{B\}$ – this independence must be true if the data is correct and enable to remove edge A-C
$A \perp\!\!\!\perp B \backslash \{C\}$
$B \perp\!\!\!\perp C \backslash \{A\}$

After performing first order independence tests we will get a graph that is showed in Figure 16 (d).

After having the final undirected graph we must orient the edges. In order to orient them we must find the "colliders". This means finding the converging connections. This test must be carried out for all triples of vertices A, B, C such that pairs A, B and B, C are adjacent to each other in a resulting graph but pair A, C is not adjacent. If A and C becomes dependant given B then we must orient edges towards B else we must orient edges in such way that they do not collide on B. The independence tests are not performed again in PC algorithm. We use the tests, which were performed in previous step.

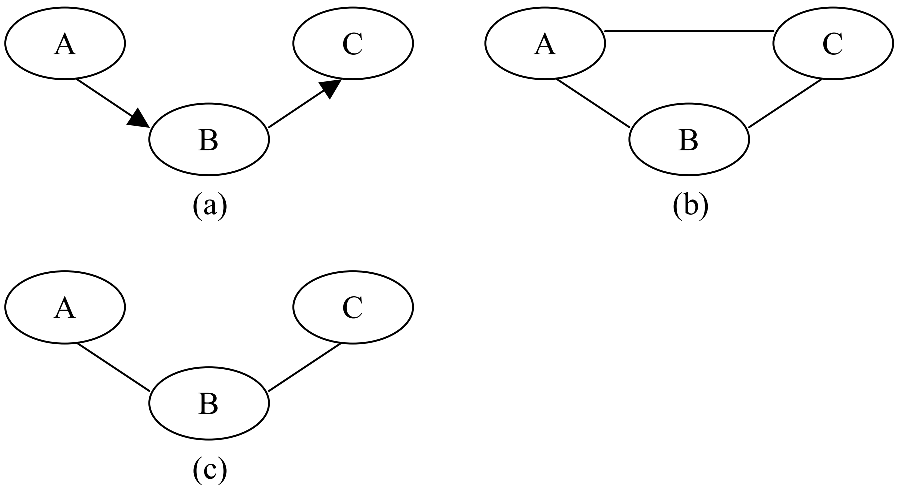


Figure 16 Traces of step 1 and step 2

The following meta code for the PC algorithm is taken from [9]:

Let **Adjacencies**(C, A) be the set of vertices adjacent to A in a directed acyclic graph C. (In the algorithm, the graph C is continually updated, so **Adjacencies**(C, A) is constantly changing as the algorithm progresses.)

```
1) Form the complete undirected graph C on the vertex set V.
2) n = 0
   repeat
          repeat
                 select an ordered pair of variables X and Y that are adjacent in
                 C such that Adjacencies(C, X)\{Y} has cardinality greater that
                 or equal to n, and subset S of Adjacencies(C, X)\{Y} of
                 cardinality n, and if X and Y are d-separated given S delete
                 edge X – Y from C and record S in Sepset(X, Y) and Sepset(Y,
                 X);
          until all ordered pairs of adjacent variables X and Y such that
          Adjacencies(C, X)\{Y} has cardinality greater than or equal to n and
          all subsets S of Adjacencies(C, X)\{Y}of cardinality n have been
          tested for d-separation;
          n = n +1;
   until for each ordered pair of adjacent vertices X, Y, Adjacencies(C, X)\{y} is
   of cardinality less than n.
3) For each triple of vertices X, Y, Z such that the pair X, Y and pair Y, Z are
   each adjacent in C but pair X, Z are not adjacent in C, orient X – Y – Z as
   X -> Y <- Z if and only if Y is not in Sepset(X, Z).
   repeat
          If A -> B, B and C are adjacent, A and C are not adjacent, and there is
          no arrowhead at B, then orient B – C as B-> C.
```

```
            If there is a directed path from A to B, and an edge between A and B,
            then orient A – B as A -> B.
        until no more edges can be oriented.
```

In theory, the PC algorithm is not stable in both steps 2) and 3) but in practice step 2) proved to be more reliable than 3).

Lets discuss some example and show where errors may occur. Suppose we have a graph presented in Figure 17 after performing 0 order independence tests.

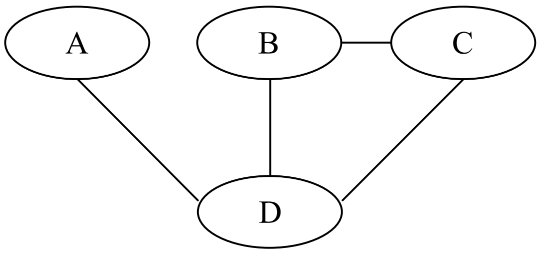


Figure 17 error type 1 example

By testing 1st order independence test in such graph will need to test the independence $D \perp\!\!\!\perp C \backslash \{A\}$. If this test would give the result that it is true then the algorithm will remove the D – C edge. As we can see in the graph the A cannot separate D and C. This would lead to the incorrect removal of the edge.

The other kind of error can occur in the edge orientation step (3). Suppose we get the undirected graph presented in Figure 18 after performing step 2 of the algorithm. And we have independencies which are $A \perp\!\!\!\perp B \backslash \{\emptyset\}$, $A \perp\!\!\!\perp C \backslash \{\emptyset\}$ and $B \perp\!\!\!\perp C \backslash \{D\}$.

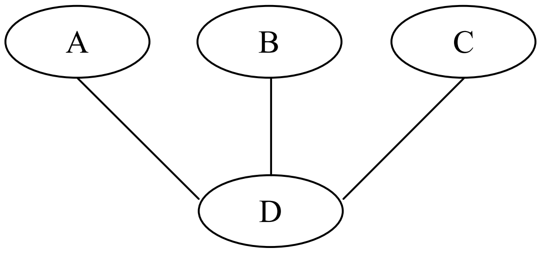


Figure 18 error type 2 example

When orienting edges we have 3 triples BDC, ADB and ADC. When performing 3rd step of the algorithm we find out that AB should converge on D, AC should converge on D but BC should not converge. As we can see there is no way to satisfy those conditions.

To solve first kind problems the PC* algorithm could be used. Suppose we have two adjacent variables A and B. If they are conditionally independent given Pa(A) or given Pa(B) then they are independent given a subset of Pa(A) or given a subset Pa(B) consisting only of variables lying on undirected paths between A and B. Then it is enough only to test for conditional independence of A and B given subsets of variables adjacent to A or B that are on undirected paths between A and B. The drawback of this algorithm is that it must keep the record of all undirected paths in the graph it considers at that stage. Usually it is unfeasible for large graphs as the number of undirected paths is typically large. But the PC algorithm could be used until the graph becomes sparse enough and then switch to PC* algorithm.

Chapter 4 Environment Description

In this chapter a short description of the Unreal Tournament (UT) game, Gamebots platform and architecture of the Rbot is presented. Only main aspects that are important for the project are described. More information about UT game or Gamebots platform can be found at [10] and [11].

4.1 *The Game - Unreal Tournament*

The Unreal Tournament is a game that belongs to a 3D shooter class. In such games a player controls some kind of an object (may be a human, robot, vehicle) ant the information to a player is presented as if it was looking thought the objects "eyes". The players itself is residing in a 3D world. The example of a players view is presented in Figure 20 and the example of 3D world surrounding player in Figure 19. The UT is a complex and high dynamic game. The game has different types of games, which are described in section 4.1.1. The 3D environment where player resides is encoded as a map. Maps can be changed by a player and they can vary in their size and their complexity. To be able to play a game the opponents are required. The opponents can be computer controlled (AI) or human controlled.

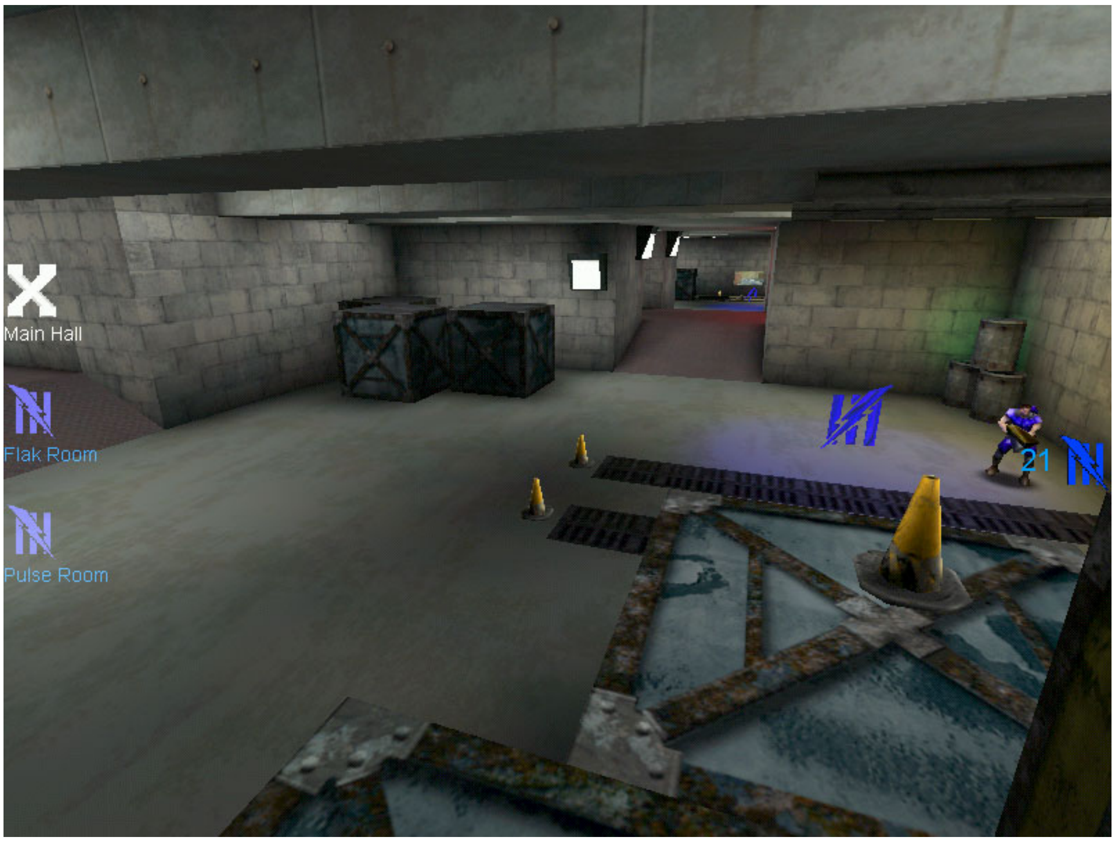


Figure 19 The view of 3D environment

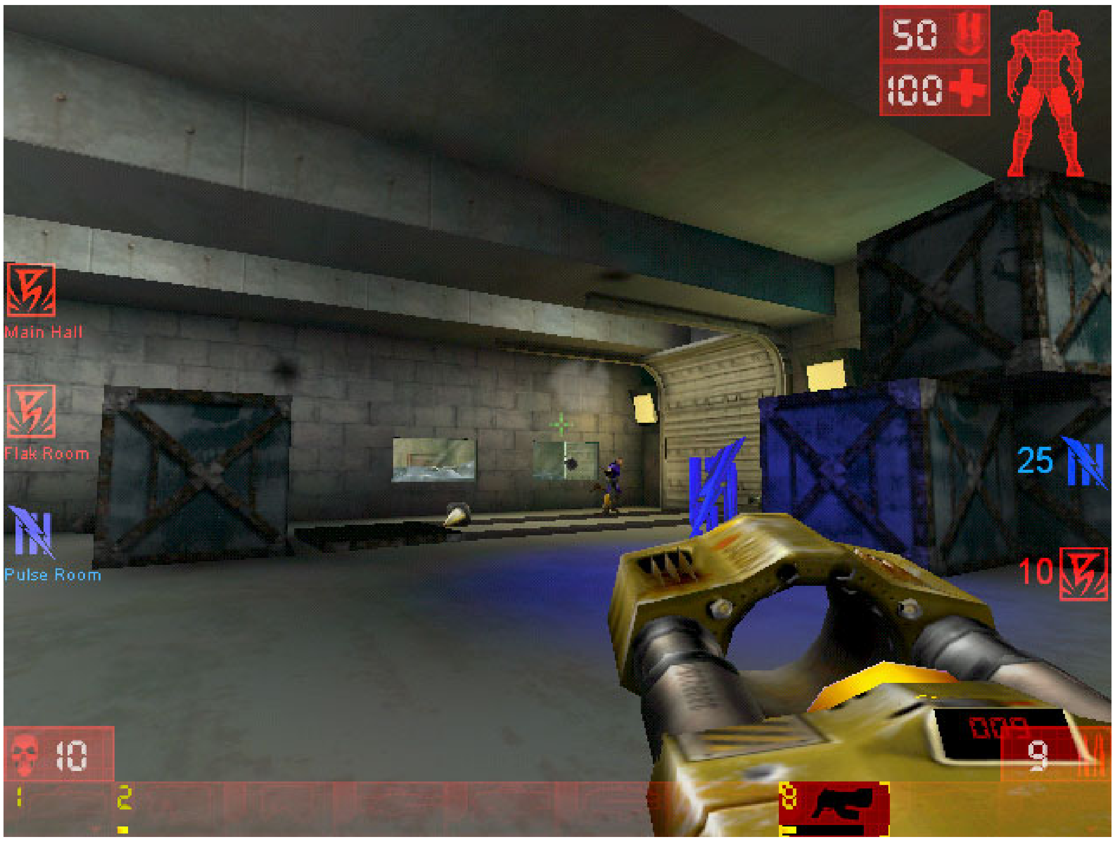


Figure 20 The view from player's perspective

4.1.1 Game types

There are several types of games in UT. There are three basic types that differ in goals. Most of other game types are usually derived from these three, which are described below:

Death match: The purpose in this game type is to be able to kill opponents as many times as possible. The limiting factor can be either the number of kills (the kill are also called frags) or the time limit can be specified. The winner becomes the player that has most kills.

Domination: In this game type players are divided into teams. Each teams purpose is to capture and defend certain points in the map. Each such point controlled is generating points for the team which posses it. The team which collects the defined number of points or which has the most of them when time runs out wins.

Capture the flag (CTF): In this game type the players are divided into two teams. The point is given when a team gets enemies flag and brings it back to their own home base. To get a point their own flag must be in their home base (as the other team can also steal the flag and try to bring to their home base). The team that first gets the defined number of points wins.

4.1.2 Opponents

The opponent in UT games is either the computer controlled (AI) players or the other human players connected through LAN or Internet. The AI controlled bots in UT can be considered to be a strong opponent and make a game challenging. The AI of UT bots is state automaton but still sometimes the behavior can be very similar to the behavior of a human player. Such behavior is achieved by "cheating" as the bot may use the information that is not available to a human player (positions of items, positions of other players, the exact aiming). The different AI difficulty can be set which changes the field of view (FOV) angle, the aiming error, and speed of a bot.

4.1.3 Items

The certain properties describe the state of a player. Those properties are health, armor, weapon and ammo. In order to be able to kill an opponent player must be in a good state. The more health and armor it has the more damage it can withstand before death. To cause more damage player must gather different types of weapons and then use them. Each weapons has different characteristics such as speed, damage, rate of fire, recoil and etc. Exact specifications of weapons and powerups (health, armor, etc.) will be described in sections 4.1.4 and 4.1.5.

4.1.4 Weapons

As it was mentioned before the weapons in UT have different properties. Each of those properties makes a weapon useful in different situations. One of the classification property could be the speed of a projectile the weapon fires. Those could be classified as:

Instant damage: Weapons belonging to this class would cause damage as soon as it is fired. So if the aiming is correct the damage is inflicted to a target.

Non-instant damage: The weapons in this class usually fires some kind of a projectile, which is traveling at some given speed. The target may perform some evasive actions while projectile is traveling thus avoiding it.

The other classification could be done by the damage the weapon makes.

Explosive damage: The weapons of this class usually are non-instant damage, which fire a projectile that explodes upon impact with an object (like opponent, wall, crate) and the damage done depends on a distance from explosion center. (Real world example would be grenades).

Non-explosive damage: These kinds of weapons must hit the target directly to cause damage. (Real world example would be bullets).

Each of the weapons may have a secondary fire mode, which may be different from the primary (example could be Flak cannon which is able to fire like shotgun or launch grenades). The short description of various weapons available in UT is given below.

Impact hammer: A close combat weapon. Primary fire mode inflicts medium damage. If targeting the wall player may be damaged due to recoil. The secondary damage is fatal to an enemy if the weapon is loaded enough (player must hold a certain amount of time the fire key until the weapon reaches its maximum power). This weapon does not have ammo and is always available to a player.

Enforcer: This is the initial weapon of a player. It has a good accuracy but makes low damage. It has also a slow rate of fire. The secondary fire mode increases the fire rate but decreases accuracy, so this fire mode is useful only in short ranges. The damage done buy this weapon is low.

Double enforcer: If player already posses one enforcer it is able to pick up second one and use them both at the same time.

Shock rifle: The fire rate is slow but it inflicts a medium damage. It is instant damage weapon. The secondary mode shoots plasma balls, which are slow but inflict high damage.

Bio rifle: This weapon fires toxic waste, which inflicts damage after some player touches it (also the shooter). The projectile speed is slow, but it does not vanish if missed. It stays on the ground (or wall) for some time and then explodes damaging nearby players. The secondary fire allows accumulating those projectiles and firing several of them at once. After secondary fire those accumulated projectiles are spread in an area.

Pulse blaster: This weapon is a rapid-fire weapon. Each its projectile is causing only low damage but the high rate of fire compensates for it. The secondary fire can be used as a beam weapon, which causes damage to anyone in its path, but the distance of the beam is limited.

Sniper rifle: An instant fire weapon, which has low rate of fire but is fatal if head is hit.

Ripper: This weapon shoots razor sharp blades, which may bounce off the wall and hit the shooter. The speed of those blades is high and if head is hit instant death accrues. The secondary fire makes the discs explode upon impact.

Minigun: This weapon uses the same ammo as enforcer, but the fire rate is extremely fast. The secondary fire does not differ from the primary fire.

Flak cannon: The weapon shoots the particles as a shotgun. Particles are fast and cause high damage. But the particles disperse with the distance making this weapon effective only from close ranges. The secondary fire shoots the grenades, which explode on impact causing damage. Flack cannon can be seen in Figure 20.

Rocket launcher: This weapon fires rockets, which has a high damage and high medium speed. The secondary fire mode enables shooter to drop those rockets as grenades, which explode after some time.

The following tables will summarize the properties of these weapons.

Weapon	Damage	›peed*	Explosive	State of eWeapon and Weapon variables
Impact hammer	24	Infinite	No	0
Enforcer	17	Infinite	No	0

Double enforcer	2x17	Infinite	No	0
Shock rifle	40	Infinite	No	1
Bio rifle	20	840	Yes	1
Pulse blaster	20	1450	No	1
Sniper rifle	45, 100, kill**	Infinite	No	2
Ripper	30	1300	No	1
Minigun	17	Infinite	No	2
Flak cannon	6x16	2500	No	2
Rocket launcher	75	900	Yes	2

Table 4 Weapon properties

* Speed is given in UT measure values. ** Depending on a place you hit (legs, torso, head)

Ammo type	Ammo number	For weapon	State of Ammo variable
Shock core	10	Shock rifle	There are three states of ammo which are: Low, Medium and Full They are calculated as follows: (Ap - Ammo percentage, Ba – Ammo that bot carries, Ma – Max amount of ammo that bot can have) Ap = (Ba / Ma) * 100 Then: if Ap is less than 10% then the state is Low if Ap is less than 50% but more then 10% than the state is medium if Ap is between 50% and 100% then the state is full
Biosludge ammo	50	Bio rifle	
Pulse cell	25	Pulse blaster	
Rifle rounds	25	Sniper rifle	
Razor blades	25	Ripper	
Bullets	50	Minigun, enforcer	
Flak shell	10	Flak cannon	
Rocket pack	12	Rocket launcher	

Table 5 Ammo types, ammo number and which weapon uses them

After picking up a weapon it will never be loaded with maximum number of ammo so the player must search for ammunition and pick it up. The types of ammo are presented in Table 5.

4.1.5 Powerups

The player dies when its health reaches 0. In order to increase health player must search for health packs. The main list is presented below:

Health pack: Increases players' health by 20 points to no more than 100 points.

Health vial: Increases players' health by 5 points to a maximum of 199 points.

Keg O' health: Increases players' health by 100 points to a maximum of 199 points.

In additional to health player may have armor. It provides protection from some of the damage received. Armor is also counted in points and each time the damage occurs the number of points is reduced.

Below is the list of the main armor types:

Thigh pads: Provides 50 points of armor for a player. Protects from only 50 percent of damage.

Body armor: Provides 100 points of armor for a player. Protects from only 75 percent of damage.

Shield belt: Provides 150 points of armor for a player. Protects from 100 percent of damage.

In the beginning or after the death players are given initial status, which is given in the table below.

Weapons	Enforcer and Impact hammer
Health	100
Ammo	50
Armor	0

Table 6 Players initial state

4.1.6 Actions of a bot

The player in UT may perform certain actions. Humans control its player by using keyboard and mouse. The movement in a level can be performed only on solid surfaces such as floor, crate and barrel. The player may damage or kill itself if it falls from a certain height. There are two criteria for movement: one of them is moving vector and another is focus vector. Focus means where player is looking and targeting its weapon. Example would be movement backwards, when bot is moving backwards but looking into opposite direction of the movement. A few main actions can be pointed out:

Move forwards/backwards: The movement is performed forward or backwards from current players position.

Move right/left: The movement is performed right or left from current players position.

Rotate right/left: The player rotates to the left or to the right.

Start shoot/stop shoot: The player starts shooting its current weapon until **stop shoot** action is received. The weapon fires to the focus location of a player.

The actions of a movement can be combined. For example the player may move forward and turn at the same time. The figure explaining actions is given below.

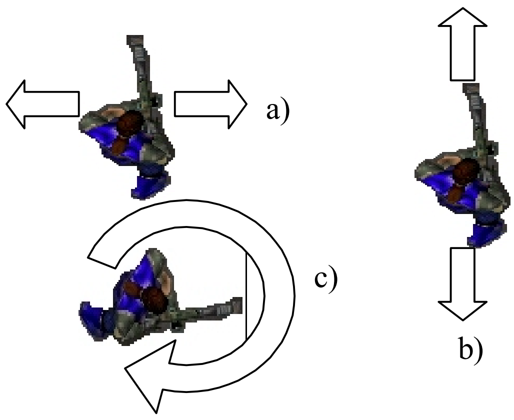


Figure 21 Movement of a player; a) move right/left; b) move forward/backward; c) rotate left/right (picture shows a player from above);

4.2 *Gamebots platform*

The Gamebots platform was created to provide AI researchers with flexible, dynamic, real time environment and to enable to use the UT engine without knowing how to program in the Unreal script language. The platform supports multiple agents. The Gamebots platform is a modification for UT game and it supports main game types such as *death match*, *domination* and *capture the flag*. It enables to create AI agents for a very popular game called UT (thus using AI agents for real applications).

The Gambot platform enables to control a player through TCP/IP sockets. This feature enables designers to use any kind of programming language that are capable of handling TCP/IP sockets. The Gamebot platform “wraps” the messages from UT game sent to a player and redirects them through sockets. This enables for player to receive information about environment it is in. It will also listen on a socket for an action to perform (the designer is responsible of sending which actions to perform).

The messages that are passed to and from Gamebot platform are text strings. The Gamebots platform has three kinds of messages:

Synchronous: These messages are generated at some time intervals (the interval may be set in properties of the game). Mostly they include sensory information about current environment. The interval used for Rbot was 0.1s (interval may fluctuate when a computer is slower).

Asynchronous: These messages can be received at any time during the game. They usually contain information about some events that happen (bot hears the noise, bot receives damage, etc.)

Action messages: These messages are sent to the Gamebot platform at any time. They contain the actions that bot should take.

The Figure 22 presents the organization of Gamebot platform. The human players are handled by UT engine itself but Bot application is serviced by Gamebot platform. From the diagram we can see that Gamebot platform must reside in the same computer where UT engine is run, but Bot applications may be run on any computer that has a network connection to a computer, which is running UT engine with Gamebots platform.

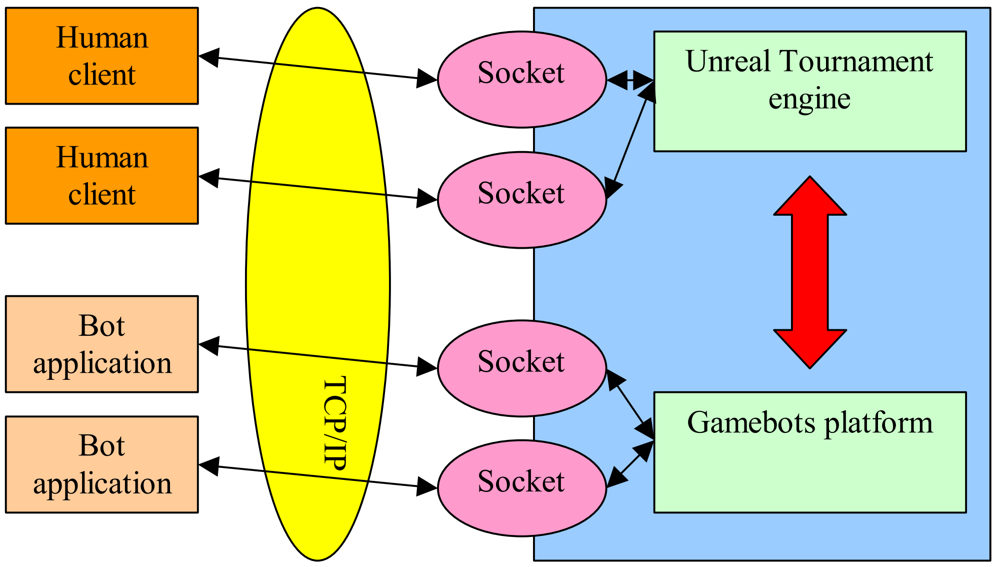


Figure 22 The organization of the Gamebot platform software.

4.3 *Architecture of Rbot*

In this section we describe the main components of the Rbot client program. We can distinguish three main components that make up Rbot:

1. Connection module.

2. Bot module.

3. Brain module (the whole brain module structure is given in Figure 24).

All of these modules were developed with extensibility and reusability in mind. Each of those modules can be reused in other client (Simple example is given in appendix A). The hierarchical structure is presented in Figure 24.

The most important part of the Rbot is its brain module as it is the part, which enables our bot to make sensible actions. The Rbots brain is itself divided into several modules, which enables us to extend brains capabilities easily.

The structure of Rbot is presented in Figure 23.

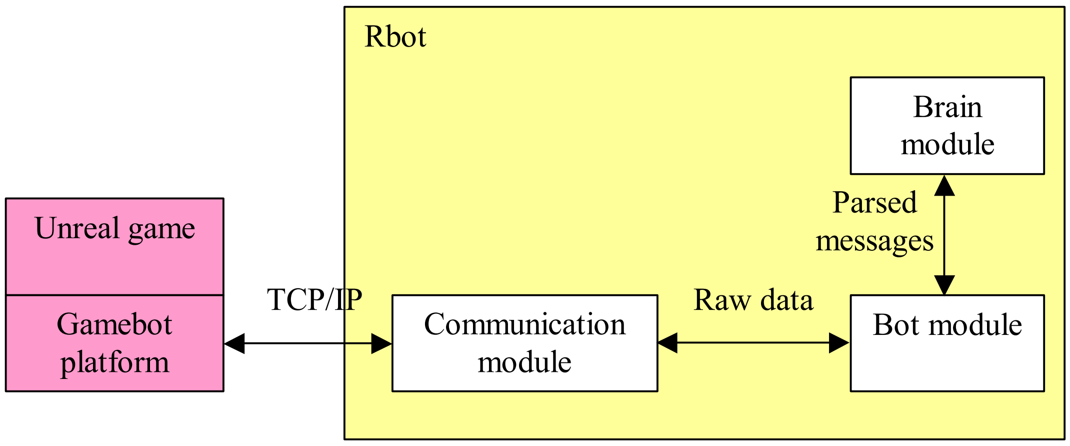


Figure 23 Structure of Rbot

4.3.1 Communication module

This module is responsible for all communication processes. It makes a connection to the Gamebot platform, makes sure the connection is open. Communication module waits for the messages to come and after receiving them it feeds those messages (they come as text strings) to a Bot module. This part of Rbot can be reused in other clients, which use network connections. Other clients do not necessarily need to connect to the Gamebots platform.

4.3.2 Structure of Bot module

This module is responsible for retrieving and classifying messages received by the communication module from the gamebots platform. It is also responsible for creating messages, which will be sent to the gamebots platform (this ensures, that we cannot send a message that is not supported by the gamebots platform. New types of messages are easily updateable). Bot module also works as a dispatcher of newly received messages to other modules, which are registered as receivers. This approach enables messages to be received in a distinct way. For example module A (Brain module in Figure 23) wants to receive only message PLR (PLR is a message type which carries information about a player) from the gamebots platform, then module A registers as a receiver of PLR message and when that message is received by the bot module, message PLR will be sent to module A and others which are also registered as receivers of PLR message. The bot module can be reused in all gamebots platform clients. In Rbot case Rbot brain module is responsible for receiving and interpreting messages from gamebots platform.

4.3.3 Structure of Rbot brain

In this section we present the structure of Rbot brain. The main task of winning the game is distributed into smaller tasks. The part of software responsible for decisions taken by Rbot is called the Rbot brain. First, Rbot's brain was divided into modules, which more or less are responsible for different tasks, such as the path finding, target notification, evasion. The split of Rbot's brain into modules enables us to concentrate on each part of the brain without thinking about the whole problem. Another advantage of such structure is to save the computing time by switching modules on or off when they are not needed. For example, if we have no targets there is no need to do evasive tricks or search for a path for the retreat. There is one important element in this structure, which is the Rbot reflex. The Rbot reflex module is responsible for all low level functions such as walk to a location, or shoot at a location. In the reflex module we code the behavior that is carried out when a decision from the brain comes (if a task is complicated it can take some time to find a decision). This kind of structure ensures that at least some measures will be taken until the decision arrives.

Rbot's brain structure is shown in Figure 24.The detailed description how the Rbot chooses its actions will be described in Chapter 6. Bayesian networks will be used for the decision-making.

The structure of Rbot brain was inspired by [12].

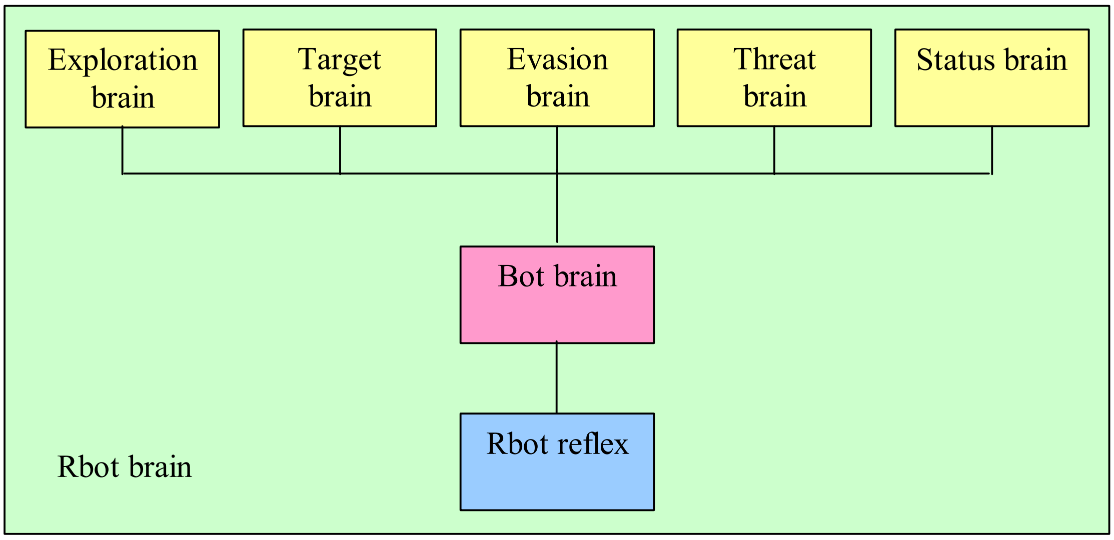


Figure 24 Structure of Rbot's brain

Event driven module is a module that is activated by a certain type of a message from the Gamebot platform. The module does not perform any computation while no messages are received. The modules that are event driven:

- Target brain
- Threat brain

Other modules are turned on or off when needed by Bot brain module.

The following sections will explain what each module is responsible for.

Exploration brain module

The exploration brain module is one of the most important parts of Rbots brain. As the modules name implies it is used to explore and memorize the environment our Rbot is in. This module takes care of finding nodes to explore. The module also creates paths to various items and domination points. This module itself does not make the Rbot to move, but it provides a place where to move for Rbot brain module, which handles movement of the Rbot.

For path finding my version of the A* algorithm is used [13]. The pseudo code of the algorithm is presented below. The input is two nodes source and destination and the output is the shortest path from the source node to the destination node.

```
1. We have empty R and Q sets (R is a set for the closed nodes and Q is a set for the open nodes).
2. We have nodes source and destination.
3. Add all nodes reachable from the source node to Q set. Mark their distance from source node (Euclidian distance is used as the nodes can be reach by moving in straight line).
4. Assign currentNode to null (no current node).
5. Add source node to R list.
6. While Q set is not empty
7. {
8.     Select the next node from Q list that has the lowest Euclidean distance from the destination node plus the current distance from a source node assigned to it. Assign that node to the current node. This means that the nodes that are nearest to the destination point will be explored first and as soon a shorter path appears algorithm will pick the node with shorter path.
9.     If currentNode is equal to the destination node return the path (we keep a record of a node from which this node is reached so we only need to backtrack back to the source node and we'll get a path) and break from while loop.
10.    Add current node to R list and remove it from Q list.
11.    Add all the reachable nodes from current node to Q list and mark their distances from source node. For example we have nodes A and B and A has marked distance from source node, so the distance of B node from the source will be d(source, A) + d(A,B) = d(source, B)
12. }
13. We did not find the path (path may not exist at all).
```

A very good description of A* algorithm was written by Amit J. Patel from Stanford University [14].

The module is also responsible for finding nearest weapon spot, health spot, and domination point.

Target brain module

The target brain module is responsible for selecting a target. Targets are selected only if there is any in Rbots field of view. If there are multiple targets in the field of view then the random target that is received from the Gamebots platform is selected (in our case only one opponent is used so it is selected every time it gets into field of view of the Rbot). After the target is selected module notifies Rbot brain module and provides it with the coordinates of a target. This module is event driven and it notifies Rbot brain module only when information about target is received.

Evasion brain module

The evasion brain module is responsible for performing evasion moves, when the target is available and Rbot decides to attack it. The evasion is also used when the target is in sight and Rbot is retreating. Given the destination and current Rbots location this module returns a spot near its current location leading towards the destination. It means that Rbot will not run in a direct line but rather deviate from course randomly or it will move around one point if the destination is the same as the current location. These movements enable Rbot to evade some of projectiles fired at it. In our case the evasion brain module is only used when fighting.

Threat brain module

The threat brain module is responsible for notifying the Rbot brain module when there is a possible threat near. Threat brain also provides the Rbot with the vector pointing from the Rbot to the direction of a threat. Such threats can be received by examining messages like (HRN – hear noise). Threat module notifies Rbot brain module when it hears noise made by other bots (direction vector of coming noise is given) and if the Rbot is shot at (the vector facing the opposite direction from current Rbots pointing vector is given). Threat brain module was used only for the first semester project.

Status brain module

The status brain module is responsible for gathering statistical data, as how many times the Rbot died, how much damage it made, what is the score, etc. This information is saved every 10 seconds for first semester project and every 25 tics for the second semester project to synchronize with actions. Later this data are used for performance results. The information from the status brain module is not used in decision-making. This information may be used in a future version of Rbot.

Chapter 5 Previous Semester Work

This chapter includes the work done during the first semester. We present a short description of a previous semester project and then present some results. Later the discussion follows which describes which are the weak parts that should be changed. Some points are given which were solved in this report.

This chapter includes only the Idea, which was behind, and the results from previous report. The theory part is shared between these projects. The Rbot in this chapter is assumed to be the previous version of Rbot.

5.1 Idea behind

Because the UT environment is highly dynamic and the observation of all the environment variables or the opponent is not complete and not always possible some methods for predicting the environment state had to be used. Also as the behaviour of the opponent is not known and is not fully observed it needs to be predicted. The Bayesian networks were chosen to perform those observations and predictions.

As the time elapses during the game the opponent might change its strategies and the ability to adapt is also required. Also the Rbot should be able to choose a best action based on the predicted state of the environment and the opponent. Influence diagrams can handle all of these tasks (of predicting and choosing the appropriate action).

The task was divided into two subtasks: prediction and decision.

Because the UT is a real time game the prediction and decision parts has to be performed in certain intervals.

In order to be able to predict enemies' action, the environment should be observed (and Rbot should have some kind of memory about previous observations). Also the prediction should not take place all the time, as it would consume computational resources. Rbot should not try to predict the enemy when it has no knowledge about it. The prediction should also be skipped when Rbot already knows what to do (for example, enemy attacks Rbot as soon as it sees it). So the following conditions should be met before Rbot starts making a predicion:

- The enemy is in Rbots sight
- The enemy does not engage Rbot

If while Rbot is observing the enemy starts attacking Rbot then the observation is abandoned and Rbot engages enemy too. If all the conditions for prediction are met then Rbot gathers information about enemy (observes) and use the information to make a prediction.

Rbot also makes a decision which action it should take. For making a decision it does not necessarily require an enemy to be in sight. If there is no enemy in sight the decision will be made without entering information about enemy (except enemy controlled domination points as this can be observed always). In Figure 25 the decision cycle of Rbot is presented. Let us explain it more deeply.

First of all the Rbot senses the environment. It collects the information about its current state and the state of the visible environment. If the enemy is present and it meets the conditions for prediction then Rbot observes it for 15 game tics (during 1 game tick the Rbot receives information about environment 1 time). If observation was successful (observation was not cancelled due to an attack) then Rbot enters the evidence it collected into the prediction network and propagation is done. Then the result from prediction network is entered into the decision network. After the result from prediction network is entered into decision network the prediction network is adapted. The action that yields the highest utility in the decision network is selected. After the action is selected the Rbot performs it and the state of a bot after the action is recorded and stored. If while performing the action Rbot engages the enemy it will resume the same action after a fight. After the game finishes all the information about the outcomes of an actions is entered to a decision network and the network is adapted.

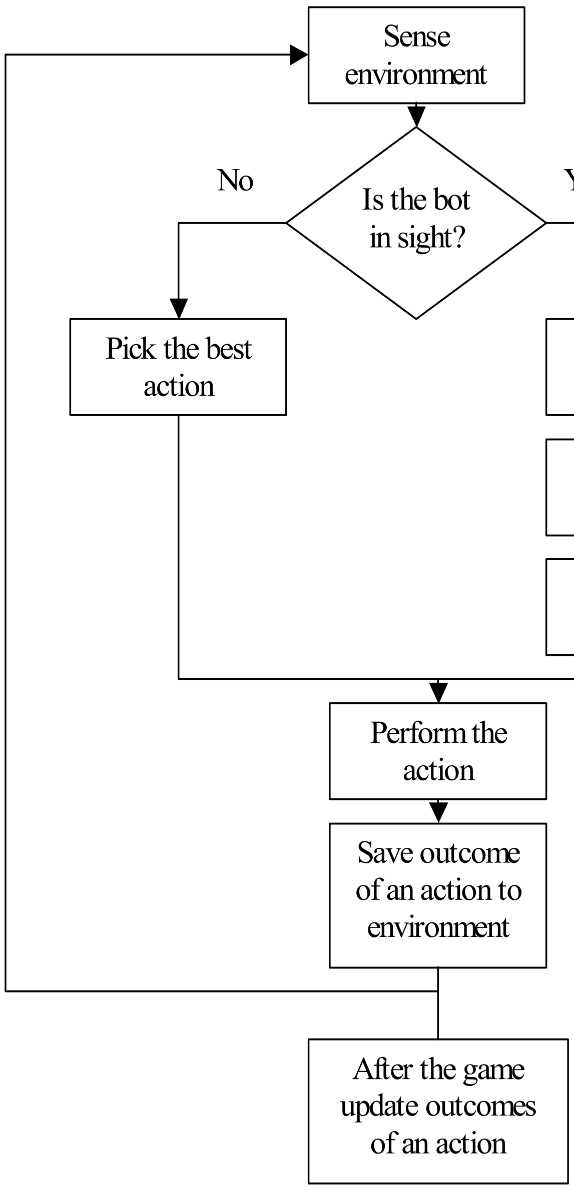


Figure 25 Rbots decision cycle

5.2 *Model*

For each task (prediction and decision) the Bayesian network is created. In this description those networks will be described separately but in the real application to ease up the coding both of the networks are merged and used as one.

5.2.1 Prediction of an enemy action

The enemy actions are assumed to be a high level actions (the example could be "pick the inventory near me"). There is no possibility to guess what the enemy is doing only looking at its current state. We need to remember what it was doing some time ago. In order to be able to enter that information we use time slices. In our prediction part three time slices were used (experiments showed that three slices are enough to be able adequately predict the action of the enemy. They approximately take 1.5 second time interval.). As the evidence seven variables are observed which are presented in

Table 7. The variables "distChng", "event", and "Rotation" are the ones that describe which action the enemy bot is performing. The actions of the enemy are represented in variable "eStrategy". The variables "eStrategy" probability table is adapted (the probability tables of other variables are not adapted), because we are interested how environment influences enemies' actions. The adaptation takes place each time the successful (15 of data about enemy is received) observation is made. The index to a node name means the time slice number (the higher the number the later observation is done. Example A and A_1 would mean that first A was observed and then A_1).

Node name	Meaning	States
domCtrl	Number of domination points controlled by the enemy bot	0, 1, 2, 3
eWeapon	Type of the weapon the enemy is carrying	0, 1, 2
Dist2Dom	Enemy's distance to the domination point	Low, High (low means closer than 500 units)
Dist2Inv	Enemy's distance to the inventory point	Same states as Dist2Dom
distChng	Enemy's distance change (the distance between last location observed and the current location)	No change, small change (walking), large change (running)
event	Did any event happened (such as the item pickup or the domination pickup)	No event, Dom change, Item pickup
Rotation	Did the bot rotate or is it steady	Stable (moving without rotation), random (the enmy bot is turning)
eStrategy	The strategy of the opponent bot	Hunt, Ambush, Camp, Roam, Unknown

Table 7 The meaning of observable variables

Three time slices are presented in Figure 26. As the adaptation takes place the only interesting conditional probability table is attached to a node "strategy_2". Only tables attached to "eStrategy", "eStrategy_1" and "eStrategy_2" is adapted. The tables used in "eStrategy" and "eStrategy_1" should be equivalent to the table used in "eStrategy_2" because we want same conditional probabilities in all time slices. To make tables equivalent first we must marginalize "eStrategy_1" out for use of the table in node "eStrategy_1"and then marginalize "eStartegy" for use in first slice. When adapting "eStrategy_2" the fading factor of 0.9 is used.

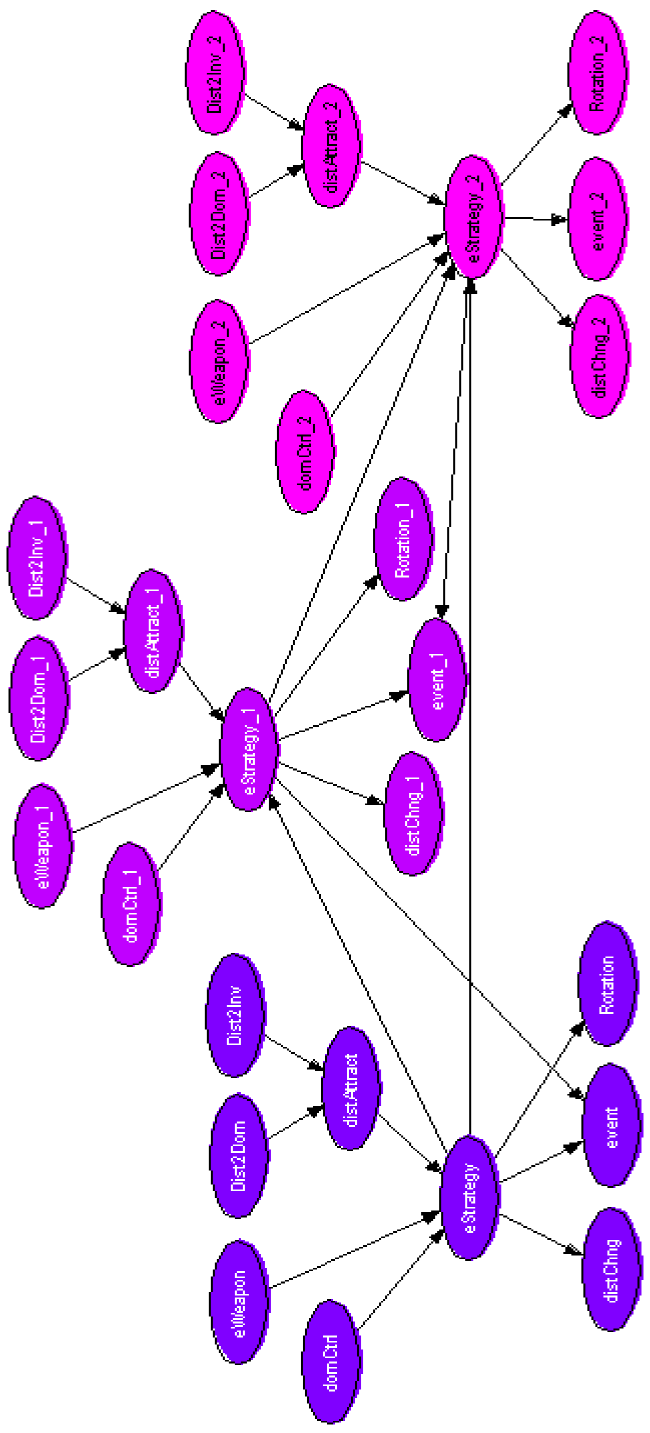


Figure 26 Three tine slices for prediction (prediction network)

5.2.2 Decision making

The part of the decision network that is used to make a decision which action Rbot should take can be represented as an influence diagram and is presented in Figure 27. The node "eStrategy_2" is a copy of a node "eStrategy_2" in prediction network and thus it contains the same table as the one in prediction network. The current Rbot state is entered into the decision network and the action, which yields the best utility, is chosen. After the chosen action is completed the state of Rbot is taken and saved for a later use. The adaptation of this part is made after the game has ended and the information that is received is used only in a next game. Another possibility would be to update as soon as we get the information, but then it would be hard to find out how the learning affects Rbot's performance.

Node name	Meaning	States
myDom, myDomAfter	Number of domination points controlled by the Rbot	0, 1, 2, 3
eWeapon_2	Type of the weapon the enemy is carrying. This node is copied from prediction part.	0, 1, 2
domCtrl_2	Number of domination points controlled by enemy bot. This node is copied from prediction part	0, 1, 2, 3
Kill	Did the Rbot kill the enemy	Yes, no
scoreChange	How did the score change during the action execution	Increase, none, decrease
Action	What action the Rbot should make or what action the Rbot performed.	Hunt, Camp, GetDomination, GetWeapon, GetHealth, Roam, Ambush
Health, Armor, Weapon, Ammo	All of these nodes has 3 states and describes the current state of Rbot. The nodes with index _n mean the state of the Rbot during next time slice.	0, 1 and 2 0 = poor 1 = normal 2 = good
Strategy_2	The strategy of the opponent bot. This node is a copy from prediction part.	Hunt, Ambush, Camp, Roam, Unknown
DominationUtility	The table for this domination is: myDomAfter | 0 | 1 | 2 | 3 DominationUtility | 0 | 0.5 | 1 | 1.5	
StaminaUtility	The table for stamina utility is: StaminaAfter | Poor | Normal | Good StaminaUtility | 0 | 0.2 | 0.3	
KillUtility	Table for kill utility is Kill | Yes | No KillUtility | 0.2 | 0	
StrengthUtility	Table for strength utility is: StrengthAfter | Poor | Normal | Good StrengthUtility | 0 | 0.1 | 0.2	
ScoreUtility	Table for Score utility is: ScoreChange | Increase | Decrease | None ScoreUtility | 0.5 | 0 | 0.3	

Table 8 The meaning of nodes in decision part influence diagram

Bellow the influence diagram of Decision part is given.

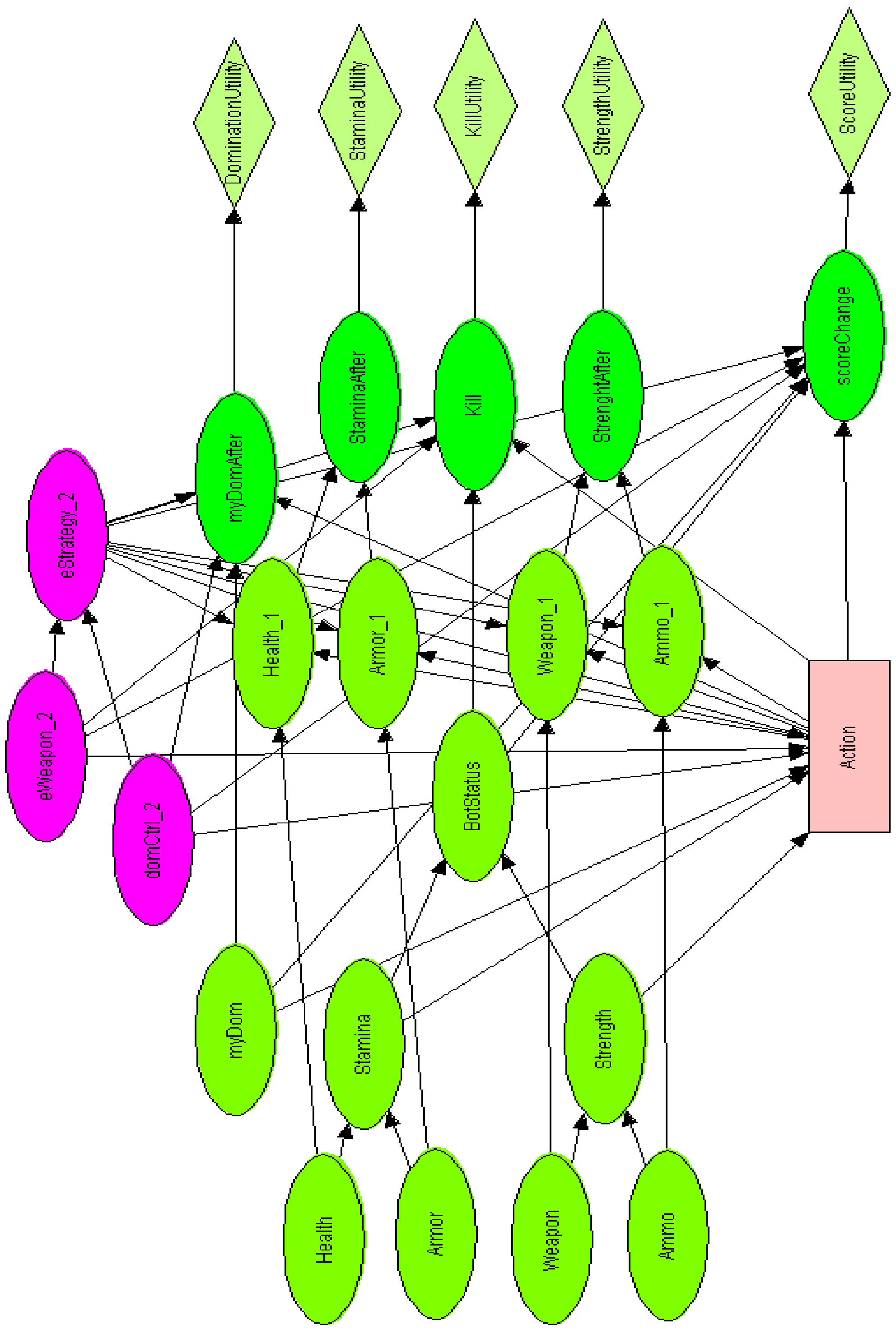


Figure 27 influence diagram used in the Rbot's decision making

5.3 *Tests and Performance of Rbot*

We carried out a number of experiments to see whether adaptation of Bayesian networks improves Rbots results. First we tested the Rbot against the built in UT bot. Then we tested Rbot against Rbot and at the end we tested Rbot against Human player, to see how does Rbot adapt.

5.3.1 Rbot against UT bot

The UT bots are well known to the 3D shooter players of their strength (ability to fight) and human like acting. The UT bot does not learn or adapt to an enemy so its strength is constant. This gives us a good opponent for testing our rbot's ability to adapt and as the UT bot does not evolve during the game play it can be used as a benchmark. First of all the Rbot with no knowledge about the UT bot is run to see how it performs against UT bot. We expected it would perform worse than the UT bot because its fighting routine is not efficient and some information is not available to Rbot. Later we expect Rbot to improve its performance by using more efficient strategies thus enabling to accumulate more points even when it is much weaker at the combat. By watching the games that Rbot played against the UT bot we noticed that after few generations the Rbot's strategy changed. Instead of involving in a fight with the UT bot, Rbot preferred to get a domination point. Such behavior could be explained that Rbot was bad at fighting the UT bot. As we can see the latest Rbot (generation 3) was clearly ahead of UT bot.

The choice of a correct action depends on a number of samples available (The sample is one of the specific observation cases. It contains information about bots initial state and the state after a certain action was chosen). Usually the bot receives average of 75 samples during one game. The table bellow shows how many samples were available for the bots of each generation.

Rbot type	Sample size
Generation 0 (no learning)	0
Generation 1	75
Generation 2	160
Generation 3	241

Table 9 Rbots sample size

The table bellow shows the final score of the Rbot. Mean was calculated as:

$$\mu = \frac{\sum_i x_i}{n}$$

The variance was calculated as:

$$\sigma = \sqrt{\frac{\sum_i (x_i - \mu)^2}{n-1}}$$

Here x_i is the sample result and μ is average result and n = 10.

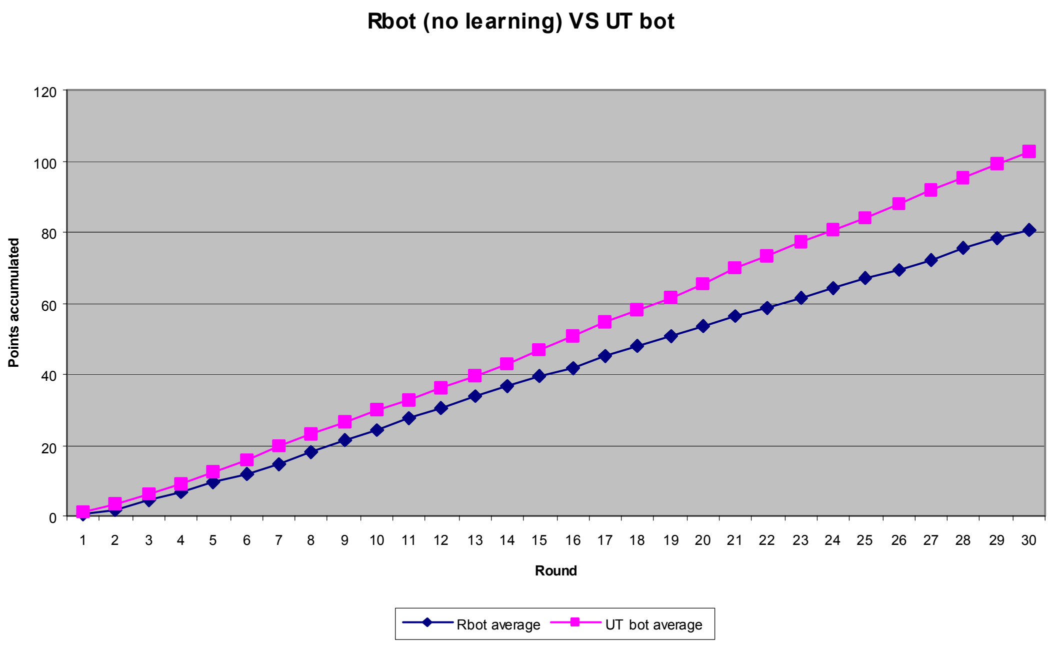
Rbot type	μ±σ
Generation 0 (no learning)	80.58 ± 12.52
Generation 1	94.78 ± 13.68
Generation 2	100.57 ± 13.82
Generation 3	100.70 ± 13.36

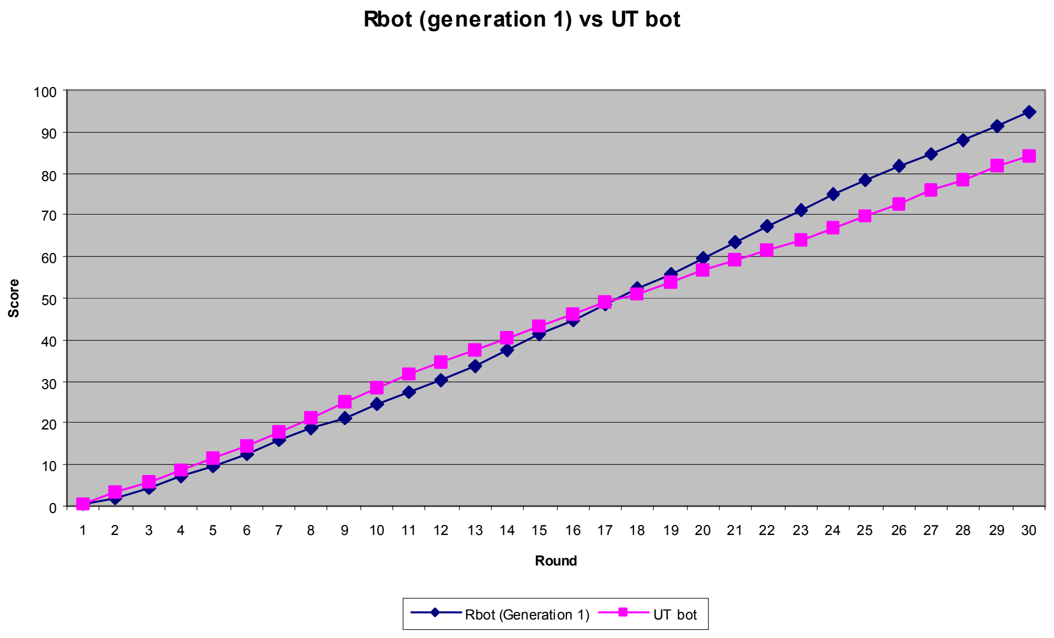
Table 10 The final score of the Rbot

Below the graphs representing Rbots and UT bots score during the game. The options for the game were:

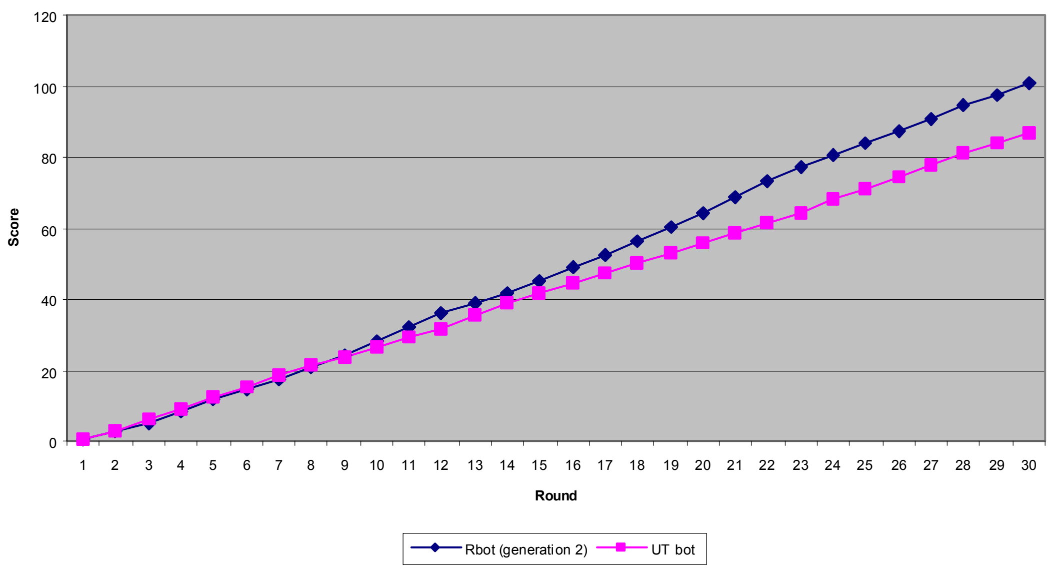
- Score limit 999
- Time limit 5 minutes
- Map: DOM-Stalwart

Ten tests for each generation Rbot were carried out.

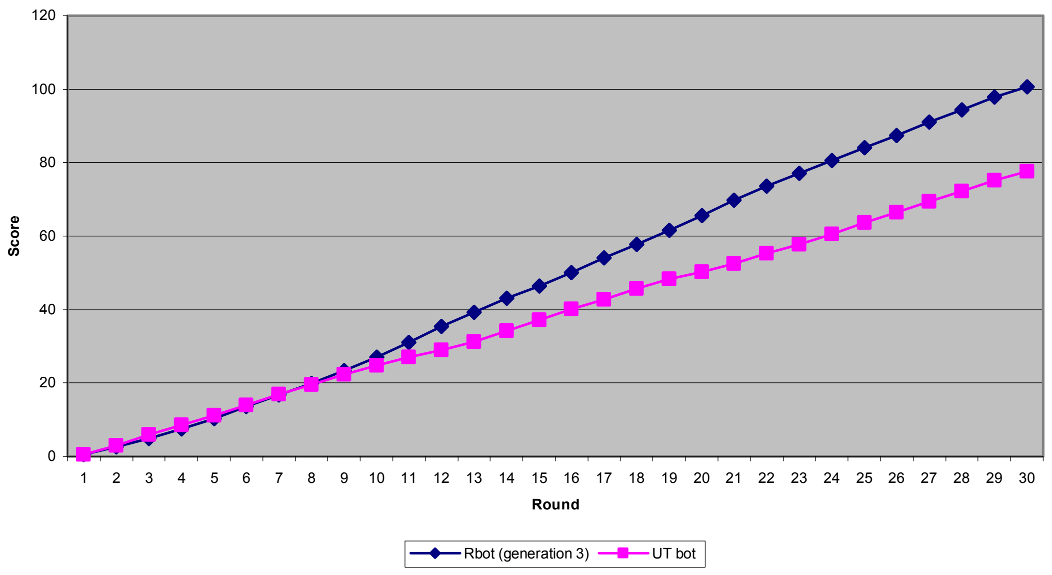




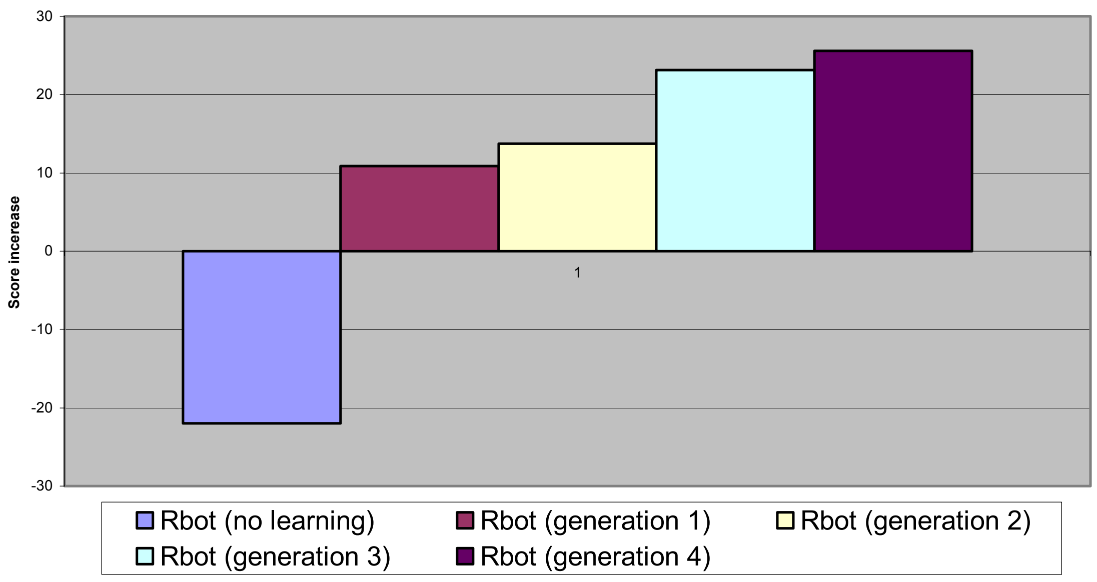
Rbot (generation 2) vs UT bot



Rbot (generation 3) vs UT bot



Final score difference between Rbot and Ut bot



The prediction accuracy (number of correct predictions about enemy's strategy) of Rbot during this test was 69%. This shows that we should use more variables to describe enemy's actions to improve the prediction accuracy. This result is not very accurate as it is received only from 20 observations.

5.3.2 Rbot against Rbot

The next test was done Rbot versus Rbot. This was carried out to ensure that Rbot is capable to adapt not only to a UT bot but also to other player. The probabilities of enemy's strategy and the probabilities of which action would yield best-expected utility were monitored in this test. Ten tests for each generation Rbot were carried out. The score limit and the time limit were the same as for the previous tests.

The states of variable "eStrategy_2" are different than the states for the "Action". This means some of the actions are described by the same variable state in "eStrategy_2". The actions "getHealth", "getDomination", "getWeapon" and "Roam" are reduced to the enemy strategy "Roam". All of them have the common behavior such as moving toward some location (where domination point, health, weapon, etc. exists). This movement by Rbot is identified as the "Roam" action.

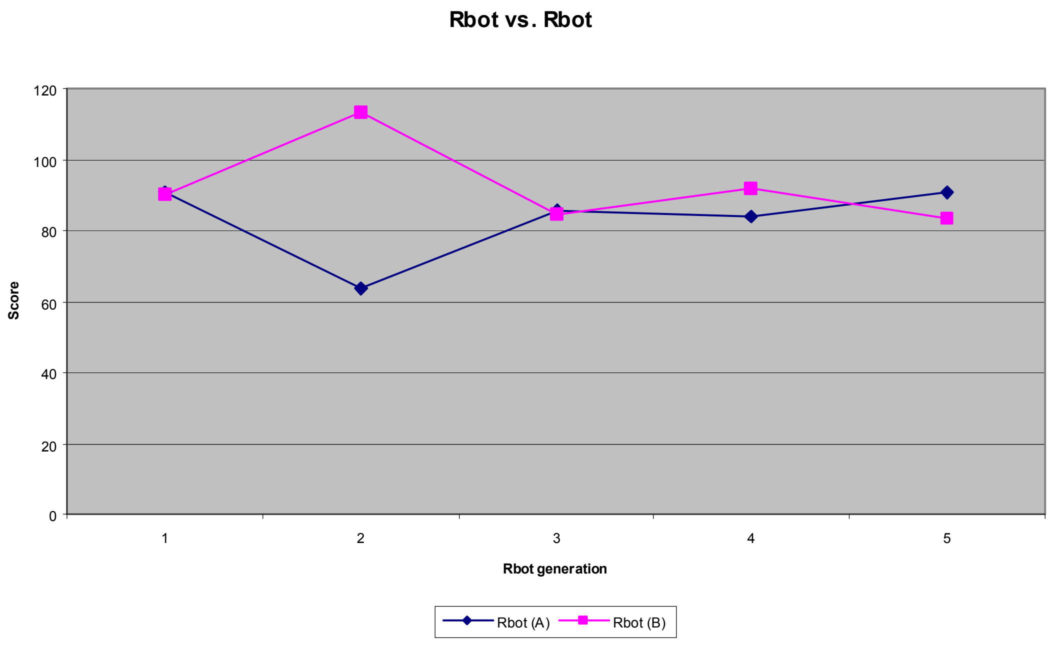


Figure 28 Rbot versus Rbot

As we can see in Figure 28 the Rbots are adapting to each other. The second generation Rbot(A) was loosing with a large score difference but after adaptation it was capable to reduce the difference and even win. The score fluctuates as the Rbots are adapting to each other. The Rbot(A) during first generation was able to successfully kill Rbot(B) several times. This increased the probability of the "Hunt" action and by using it in a second generation was not able to stop Rbot(B) from controlling domination points.

In Figure 29 one of the Rbot was allowed to adapt and the other was always using the same conditional probabilities (uniform distribution). As we can see from the graph the learning Rbot gains advantage over non-learning and then maintains it.

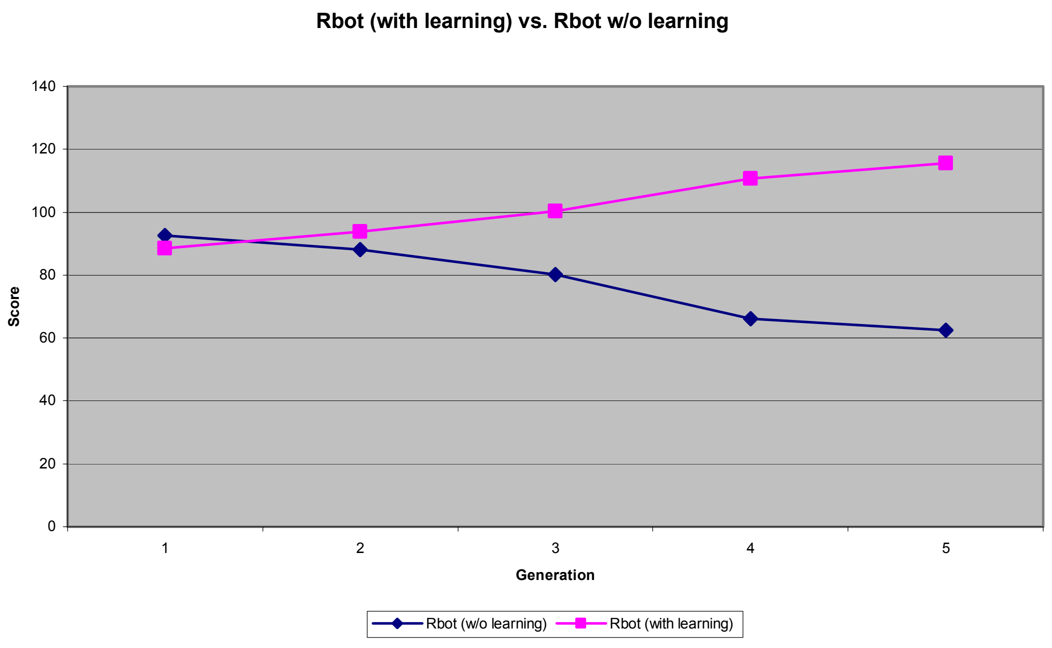


Figure 29 Rbot with learning versus Rbot without learning

In Figure 30 the change of expected utility of Rbot as it adapts is shown. The Rbot may have different strategies for different state configurations so we will present only some of them. The fixed values of Rbots states are taken (Health=4; Armor=0; Ammo=0; Weapon=0; domPoints=1; enemyDomination=0). As we can see the utility of action get domination does not change much as it is the primary goal of the Rbot and at given state Rbot owns only 1 of 3 domination points but the utilities of other actions does change (we can see from the graph that hunt utility decreases and roam increases).

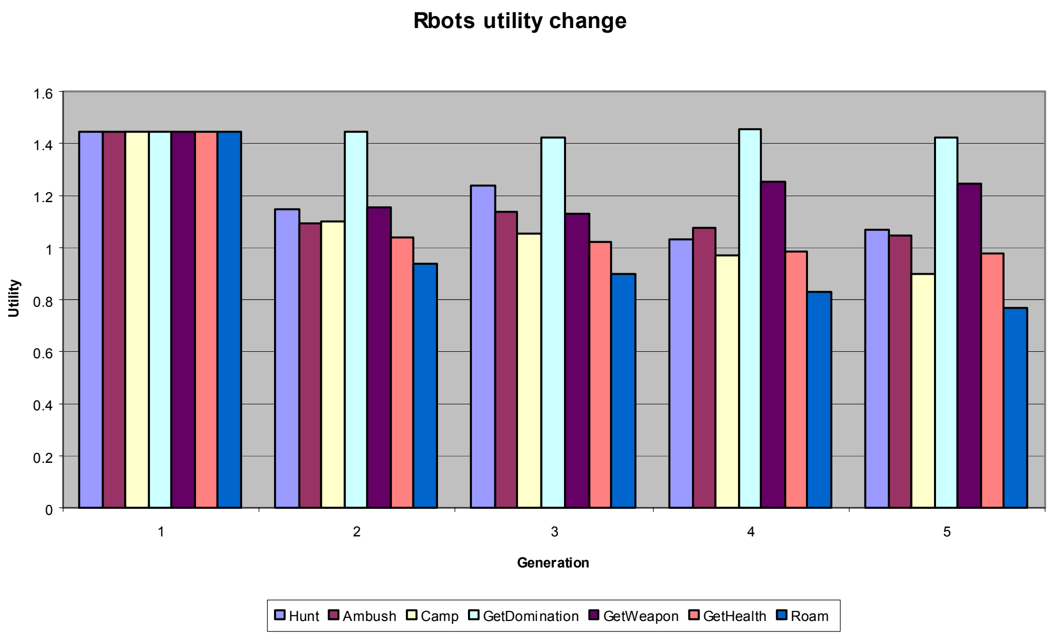


Figure 30 Utility change of the Rbot

	Enemy's strategy				
Rbot's action	Hunt	Ambush	Camp	Roam	Unknown
Hunt	1.863	1.463	1.565	1.756	1.797
Ambush	1.601	1.654	1.594	1.794	1.624
Camp	1.584	1.746	1.689	1.846	1.616
GetDomination	1.599	1.524	1.548	1.628	1.799
GetWeapon	1.574	1.548	1.566	1.537	1.462
GetHealth	1.850	1.885	1.850	1.757	1.783
Roam	1.801	1.821	1.816	1.764	1.735

Table 11 Rbot's actions expected utility depending on enemy's strategy

Table 11 shows how the enemy's strategy affects Rbot's decision. Here the following state is taken:

Health = 4; Armor =0; Weapon = Type 2; Ammo=Low;Domination points=2; Enemy domination points = 1;

The prediction accuracy of Rbot during this test was 49%. Such low number was caused due to a technical error in the Gamebots platform, because the Rbot is not capable of slow movement and action "camp" depends of the speed of movement. The prediction result is not very accurate as it is received only from 20 observations.

5.3.3 Human against Rbot

In this test The Rbot was playing against Human. As the Human player was the project author and possessed the good knowledge about UT bot and Rbot. Human player due to its ability to adapt to an enemy and to employ new strategies won all of the games. After playing a several games it could be observed that UT bot was much more hard to kill but easy to win due to its poor strategy on the other hand the Rbot was much easier to kill (the fight routine of the Rbot was not the main issue) but it was harder in a strategic sense (the Rbot tried to claim more domination points rather than fight this made human player also to go for domination points). Ten games with each bot type were played. As we can see in Table 12 neither the Rbot nor the UT bot are able to beat humans yet.

Enemy	Human score	Enemy score
UT bot	122	42
Rbot (generation 5)	117	49

Table 12 human player against the Rbot and the UT bot

5.4 *Discussion*

This section will describe the problems encountered with previous version of Rbot. The main issues will be mentioned and in the next chapter most of the solutions will be given and then the new performance results presented.

5.4.1 Prediction

One of the problems was observed with the prediction. It was not very accurate. The problem is that the tables in nodes "eStrategy_1" and "eStrategy_2" have too many parents. The table size for “eStrategy_1” is 600 and the table size for “eStrategy_2” is 3000. This means that the huge amount of information is needed to change the beliefs. It seems that the Rbot was making a prediction of an enemy only from prior probabilities and the data that was received during the game did not affect the probability distributions much.

The second problem with the prediction is that usually the enemy is encountered only several times during one game. This means only small amount is available to Rbot. Also the encounters are very short (we used ~1.5sec for information collection) and during this short period it is very hard to predict the real strategy of an enemy. It may happen that we detect only the small part of some strategy, which may reassemble totally different strategy thus making a wrong prediction.

Also as it was mentioned before the number of nodes describing some enemy strategy was too small. This means that the strategies of an enemy are not described fully. This inaccurate representation causes wrong predictions.

5.4.2 Decision

There could be several problems with the decision part pointed out. One of the problems is the utility functions. There are five utilities used in making the decision. The only clear utility function is for domination points as domination points directly affect the final score. All of the other functions are not known. Each of those utility functions is defined by an expert and do not change over the time. There could be that those utility functions are not correct or change over time (if another bot is learning). In such cases the Rbot will start making bad decisions based on the wrong utility functions.

There is also a problem of a bot learning its own utility function. Learning opponent’s utility function can be done by observing its actions. The different problem arises with learning your own utility function as the actions has to be selected before outcome can be observed. This means that to learn your own utility function you must try different combinations of the actions several times for different environment configurations. This could be a whole research to find the efficient way of learning your own utility function as the human player can learn its own utilities rather quickly.

5.4.3 Implementation

The next problem could be encountered in implementation of the actions. The Rbot has the defined set of actions, which are high level. Those actions were taken by observing the human players play and generalizing what strategies are most frequently used. The action is performed until it is finished. For example the action “getDomination” would make the Rbot to go to the nearest uncontrolled domination point. The distance to it may vary and the state during this action may change, but the implementation would not allow the Rbot to change its current action and it will be carried out until it is finished. Also the engagement of the enemy was also hard coded. This means that the Rbot will always attack the enemy if it is in sight and close to the Rbot or if an enemy decided to engage. Such approach disables the ability for the Rbot to change its action.

Chapter 6 New Rbot

This chapter describes the work done in the second semester of the project. It starts by a short review of the first approach and its problems. Then we introduce a new more principled approach, which bases the decision part on influence diagrams as well as the previous approach. However the influence diagram has a structure of Dynamic Bayesian network which is learned from data and further optimized using expert knowledge. Moreover it uses just one kind of utility node, which is directly related to the goal of the game. Thus there is no uncertainty in this utility.

6.1 *First semester approach*

In previous semester Rbot decision were made based on influence diagram. Because the Rbot was taking into account only the current state of the game several utility functions had to be used to enable the Rbot to pick different actions under different conditions. Several main influencing factors were introduced such as killing the opponent, getting more health or getting better weapon. An expert introduced all of these functions. All those functions except the scoring function are not clear in UT environment. An expert presented only averaged functions, which might not necessarily be true. Usually in such cases some kind of utility function learning is used to be able to change the utilities.

The positive thing about the previous approach was that it used a prediction of the enemy but later it showed to be not precise and problems with prediction are described in 5.4.

6.2 *New approach*

6.2.1 Introduction

In this semester project the decision was made to leave the same overall structure. In order to cope with the problems, which arose in previous semester, some changes had to be made. We took the more principal approach to construction of the influence diagram. Because the scoring in the game is well known we base the utility only on it. The approach, which was used in previous project, would not be good in this case as the Rbot would only chose action which increases the utility (action go to domination). In order to enable the Rbot to "look" into the future we have to use dynamic Bayesian networks. The Figure 31 shows a simplified version of the influence diagram with two time slices (slices are separated by dashed line). This means that when making the decision the Rbot will think about the next action as well. This would enable the Rbot to pick action with lower payoff if the next action could yield large payoff.

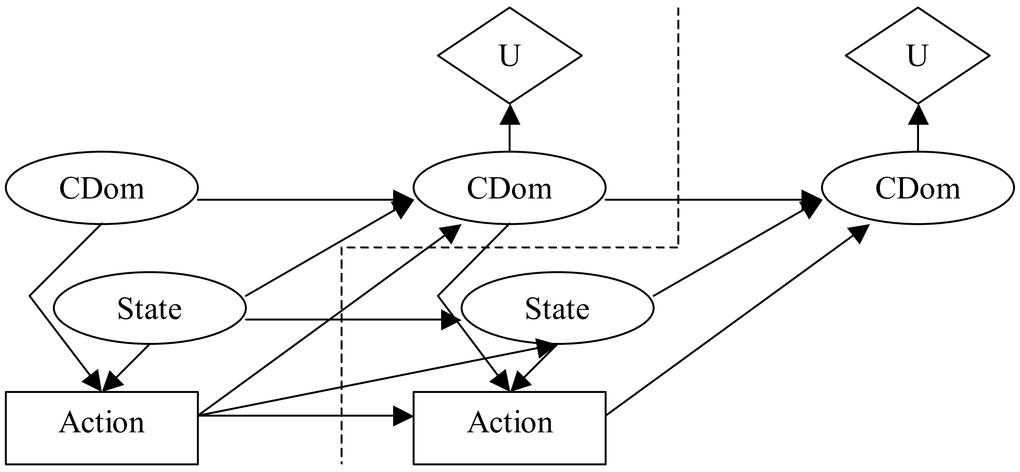


Figure 31 Influence diagram with two time slices

In order to support such influence diagram the execution of the actions had to be changed. We also learned the structure of the influence diagram from the data and then used expert knowledge to improve it considering practical aspects of the implementation as well.

Because some work was already done during the previous semester the problems that arisen were also addressed. The whole design was divided into several parts, which include:

- Implementation of the Rbots actions and perception of the game
- Construction of the influence diagram which will be used in decision making
- Optimization of the influence diagram

Each of these parts will be described in different sections.

The examination of the results from the previous projects revealed that the decision part has the most impact on the Rbots performance. In this project the main focus will be the decision part.

6.2.2 Implementation

In the UT game there are only basic actions that include movement to some direction, turning and shooting. All of these actions are low-level actions and cannot be used as encoded actions in the influence diagram. In order to be able to encode a sensible action, which could give us a certain changes in the environment some basic strategies had to be implemented. The human behavior was observed and several main high-level actions were extracted:

1. Hunt. The strategy, which is used only when opponent, is visible and includes making evasive maneuvers from incoming projectiles from opponent and shooting at it.
2. Get health. This strategy makes bot to move and pick up appropriate health items (different health items act differently).

3. Get weapon. This strategy includes going and picking up weapons or ammo for those weapons.

4. Camp. This strategy includes walking about a certain point (usually domination) and preventing enemy opponent to reach it.

5. Ambush. This strategy makes a bot to move to a certain point, which is not easily visible, but near the path to some important point (such as domination point) and wait there until enemy tries to pass then engage it.

6. Get domination. This strategy simply moves a bot to the uncontrolled domination point.

There are several ways to implement such high level actions. One of them was used in previous project. It included performing the action until it is successful. This means that the actions could be of the different length as the items can be in different places and the time to get to them would be different. To know those lengths of the actions in advance is impossible. Because the use of time slice approach for creating influence diagram such method could not be used any more as the intervals between time slices must be equal.

In order to make all actions of the same lengths they had to be synchronous. For measuring the time of each action the UT clock was used. The UT provides Rbot with synchronous messages at a certain intervals, which could be used as a clock "tick". In such case each action would be allowed to perform only a certain amount of time. The time interval has to be chosen such that the Rbot should be capable to perform some change on the environment. The time interval of the actions for Rbot was chosen to be of 25 "ticks" and it was determined by changing its length and observing the actions of the Rbot. It approximately takes 2.5 seconds. If the interval chosen would be too short the information received in the beginning of an action would be the same and if it were too long then some of the information could be overlooked. The other reason for choosing the fixed time actions was to enable the Rbot to change its mind when environment changes. The example could be when the Rbot is traveling to the certain point and suddenly it senses that the health or weapon is nearby that it could change its action to pick that item and continue on a previous course. This scenario would not be possible with previous implementation as the actions are performed until they are successful (this would make the Rbot to ignore any changes in environment and continue to its destination point).

The other implementation change was to remove the prediction part from the design. The reason for that is that the encounters are not that common in UT game when playing two players. Second reason was that the data that can be gathered about an enemy is not able to define the action of that enemy because the enemy may have totally different set of possible actions and the set of actions provided by the UT game is very small and low level (such as turn left, turn right, move forward and etc.). This means that the prediction cannot be carried out correctly. The possibility to access some currently hidden data could give at least some generalized bot state by observing the exterior of a bot (such that it is bleeding or very good shape) and could enable us to start predicting the enemy bot state. This is not implemented in the Gamebots platform. Instead of the opponent modeling we added the simple information when the opponent is in sight and close or near. The other possibility for creating the prediction part could be made from a long term observations of an enemy such as encounter rate, places of encounter and similar information.

Figure 32 shows how the information flows from environment to the Rbot. The decision making is carried our each 25 cycles.

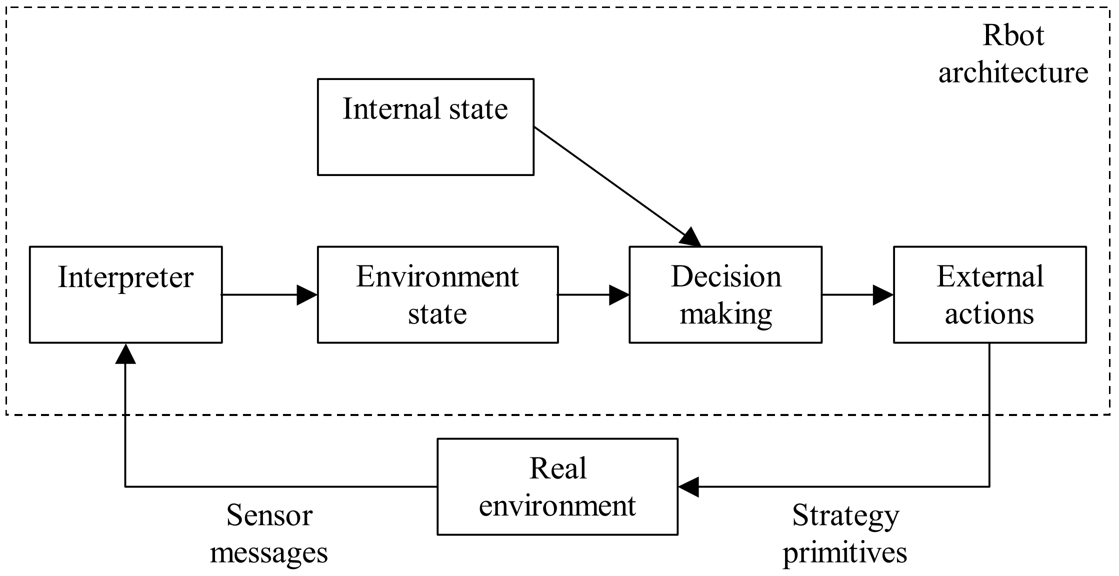


Figure 32 Data flow from environment in the Rbot

6.3 *New decision cycle*

Because the implementation of the execution of actions in the Rbot changed so did the decision cycle. The prediction part was removed and the decision part was changed. Currently the Rbot makes a decision for its action and executes it but only for a fixed period of time. This period is 25 tics (~2.5sec). After the time for current action elapses the Rbot select the action again. The only exception is if the Rbot encounters the enemy bot. Then the current action is stopped immediately even it still has some time and the Rbot chooses again. The reason this is done is that the Rbot could engage enemy at once if it decides so and not wonder for some time and only then decide to fight. This gives the Rbot better chance for killing enemy if it decides to fight as soon as the enemy bot appears in sight.

The information about the Rbots state and the action with its outcome is saved. The fractional updating of the influence diagram is performed only after the game ends.

The decision cycle of the Rbot is presented in Figure 33.

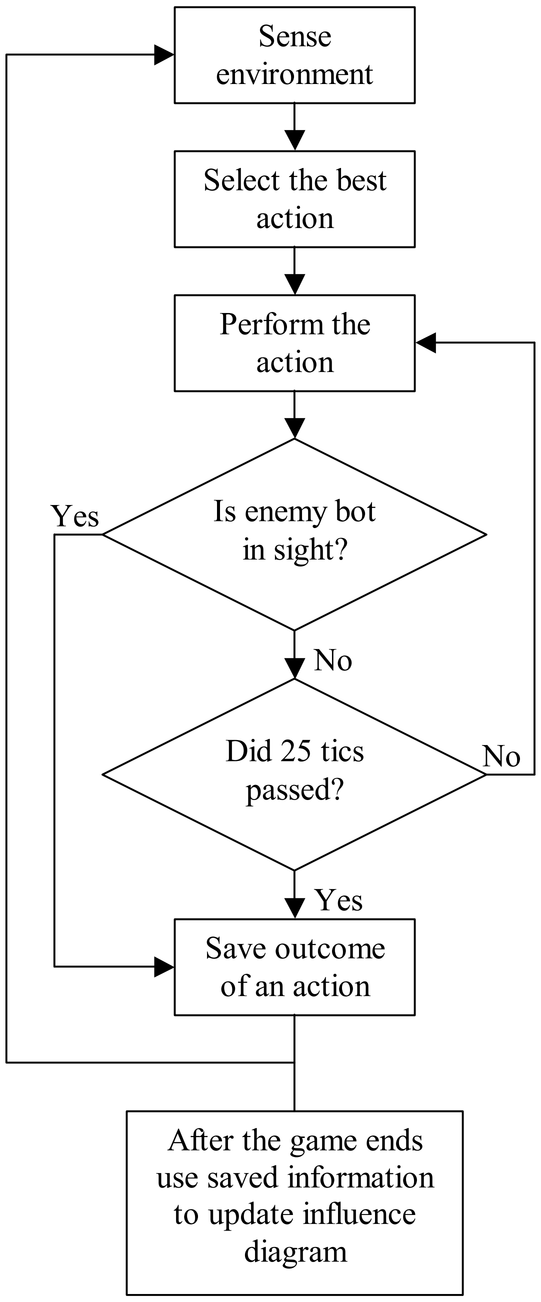


Figure 33 the new Rbots decision cycle

6.4 *Construction*

There are several ways to construct an influence diagram. One of them is that the expert is able to determine the relationships between variables and creates the influence diagram representing the given problem. Another way is to learn the structure only from data that is given. Both of these ways have same drawbacks. In the first approach the expert might not know all of the aspects of the problem and miss some important relationships. The next problem with the expert is that he is not able to cover the very complex problems where the variable number is very big and it becomes very hard to identify all of the relationships between them. The problem with learning the structure from the data is that the data might not contain the complete set of possible cases. This means that the structure learned will have some links missing.

The third approach would be to combine previously described methods together. The expert could examine the structure that was learned from the data and add or remove some relationships based on its experience. Expert would also be able check whether some relations hold true by using the structure learning.

The approach to create the structure only based on expert was used in the first semester project. For the second semester the learning of the structure combined with the expert knowledge was used.

6.4.1 Learning

First of all before learning took place some important relationships that had to be tested were noted. They included the following relationships:

- Bots state affecting bots score
- Bots actions affecting bots state

This means that the Rbots score must depend on the actions taken by the Rbot and the state of the Rbot. If there would be no relationship between the Rbots actions and the score so it means that the Rbot can make the random actions and get the same score. The next relationship that the score also depends on the state of the Rbot is also important because if were not important then the Rbot would always make one action which affects the score. And by observing the human play we can see that several strategies are used (for example the human would pick the health if the state is bad or go for the better weapon if its near). If the Rbots state is also affecting the Rbots score then the actions of the Rbot should also influence state of the Rbot. Such relationship would make the Rbot to carry out some different actions than the one that influences the score.

So here are the necessary conditions to enable the second approach to work:

a) The Rbots actions should have an influence on the final score
b) The Rbots state should also have the influence on the final score

First of all we needed to make sure that these conditions are really enough to enable the Rbot to pick different actions. To test them we created a simple network only with a few nodes. This network is given in Figure 34.

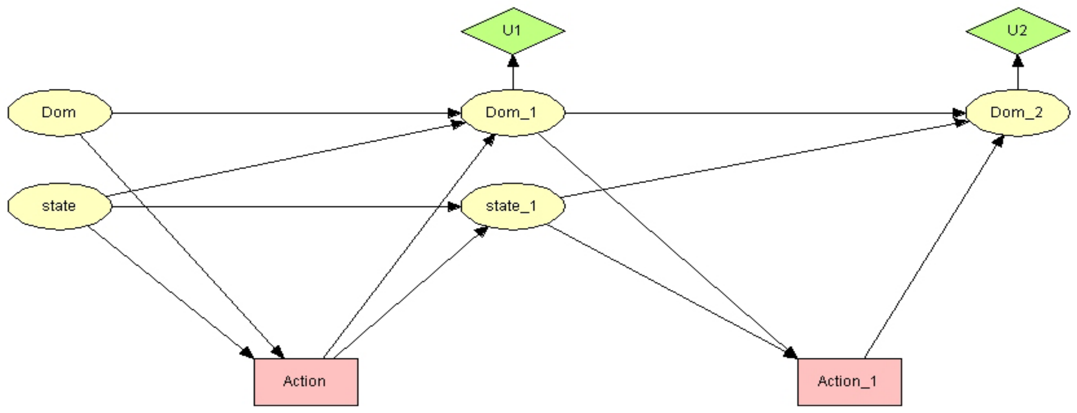


Figure 34 a simple influence diagram to test the conditions

Then we adapted the network with some typical behavior of the Rbot and checked whether changing the Rbots state or the actions the expected utilities change.

After performing some test on the simple influence diagram (Figure 34) we found that both of the conditions presented above were fulfilled because:

a) After changing the action the expected utilities changed and so did the probabilities for a state (in second time slice).
b) Changing the state of the Rbot (first time slice) had effect on the expected utilities.

To test these conditions and probably to find some other relationships the learning algorithms on a real data acquired from the Rbot were used. Two tools were used. B-Course [15] and Hugin [16]. The B-Course came up to be not useful in our case as the time limitation for calculation prevented to find the good model.

The Hugin proved to be useful and confirmed that the relationship between bots state and number of domination points it possesses exists. It also proved that certain actions influences Rbots state (for example if the action would be go for weapon then the probability that the weapon in next time slice will be higher). Also some additional links were found. Those include a link between health and armor (This can be explained that usually the armor is reduced to 0 before health decreases to 0. This means that if we have some armor then it means that we probably have good health). Also the links if the Rbot die then its state is affected were found.

After using the structure learning some edges relevant to the UT game were found. They included the dependence of the distances between health powerups and weapons or domination points. Such edges were removed by expert because they are true only for the one map used in our experiments. Some of the links like dependence of the armor given action were not present due to the incomplete data and they were added by an expert.

After performing structure learning and applying expert knowledge the initial influence diagram was constructed. The initial structure included only two time slices. No more time slices were introduced because it took much time to propagate the network.

6.4.2 Initial influence diagram

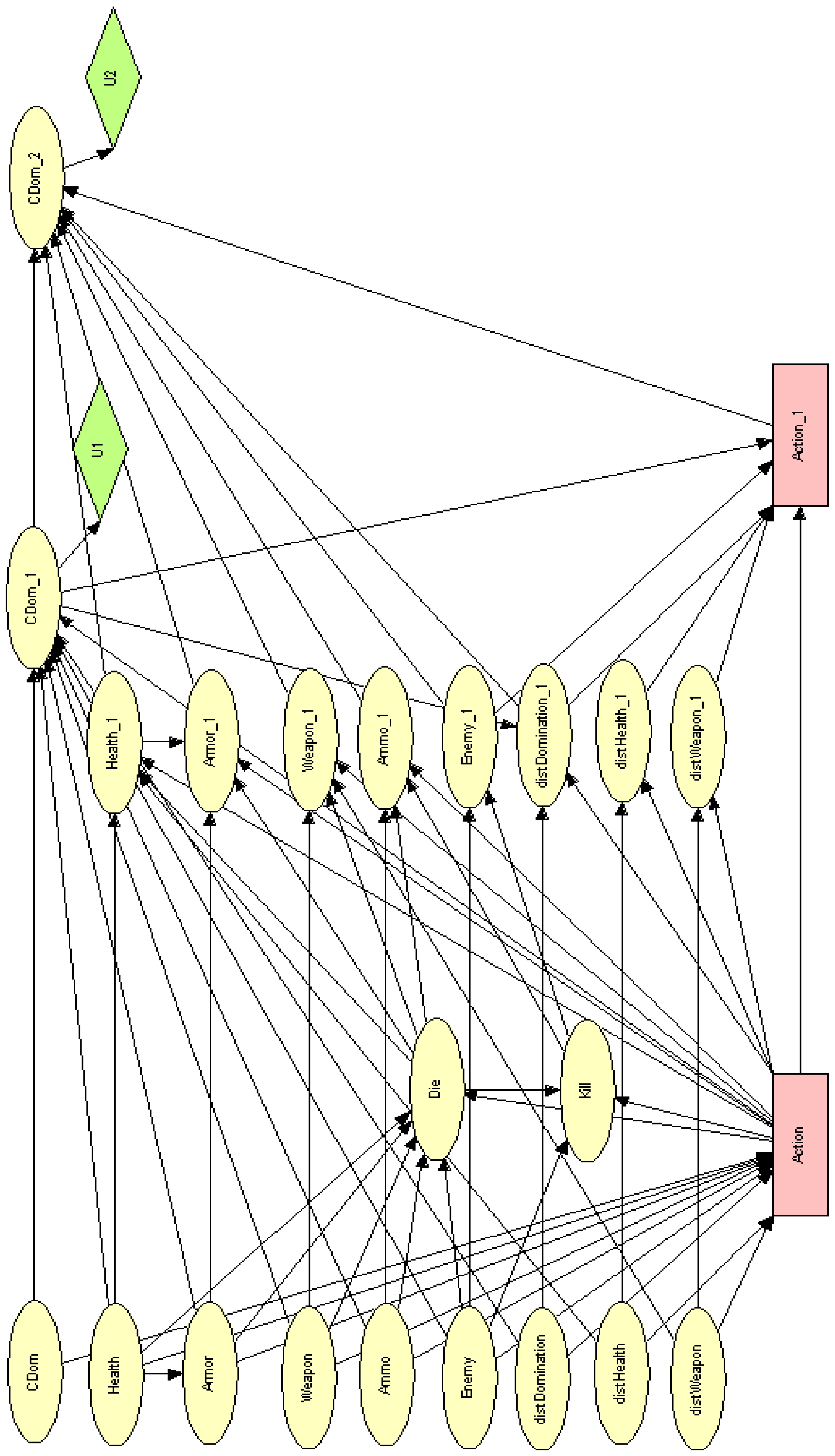


Figure 35 Initial influence diagram of the Rbot

6.5 *Optimisation*

After creating an initial diagram the test of propagation showed that it takes too much time to complete it. In order to reduce the time taken to compute such diagram some optimization was done. First we should concentrate on the nodes with the most parents. As we can see in Figure 35 the node Cdom_1 and Cdom_2 have the most parents. The technique called parent divorcing (3.6) can be used to reduce the size of parents. We divorce health and armor nodes and weapon and ammo. Both of these node pairs are related to each other and describe the Rbots strength and stamina. As the armor is counted as the percentage of the hit points it can be treated as hit points. This means that a state were bot has a good health is equal to the state when bot has normal health and some armor. This enables us to divorce health and armor nodes by introducing hidden node stamina. The same holds for the weapon and ammo. If the bot has a good weapon but low ammo it can be treated as weak weapon with a lot of ammo as the amount of possible damage to the enemy is the same. This enables us to divorce weapon and armor nodes by introducing hidden node strength. As both strength and stamina describes the state of the bot they can also be divorced with hidden variable BotState. The divorcing is shown in Figure 36.

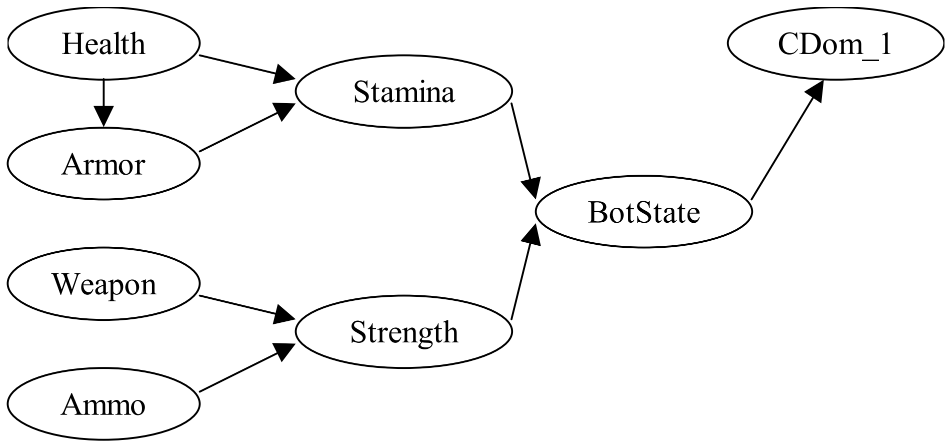


Figure 36 divorcing of health, armor, weapon and ammo nodes

The reason is that the bot with good stamina but poor strength is almost at the same condition when it has a normal stamina but a good strength. This divorcing reduces the number of “CDom_1” and “CDom_2” parents from 8 to 5 and reducing table size from 82944 (in some cases the Hugin would throw an out of memory exception) to 1152. The values for hidden nodes were learned using EM algorithm. This reduction does not only reduce the size of conditional table but it also increases the speed of adaptation.

After reducing parent number of “CDom_1” and “CDom_2” still the propagation time was not acceptable. The usage of such model made the Rbot just to stand and calculate the actions instead of doing them. To reduce the computation time of an influence diagram the parent number of “Action” and “Action_1” node was reduced. Because the children of “Health”, “Weapon”, “Armor” and “Ammo” has fixed conditional probability tables and so does the their child node “BotState” we can use the link from “BotState” to “Action” and thus reduce by 3 parents. This reduction made the propagation to last less than 1sec, which is low enough.

The following table shows the time taken to compute the influence diagram after optimizations.

Influence diagram	Time
No optimisation	2.7s
Parent divorcing	2.6s
Parent reduction	120ms

Table 13 time taken to propagate evidence

6.5.1 Future optimisation

For future optimization the nodes with most parents should be addressed first. An observation from a data can be made that the Cdom_1 is very dependant from Cdom. We can also note that usually the domination point increases only by 1 point. This enables to make following change where part of the change (for slice 1) is presented in Figure 37. In this figure the node Increase can have three states for increase in score, decrease of score and for no change. The data for Increase node can be received by observing environment. This would save the number of cases from 1152 to 912.

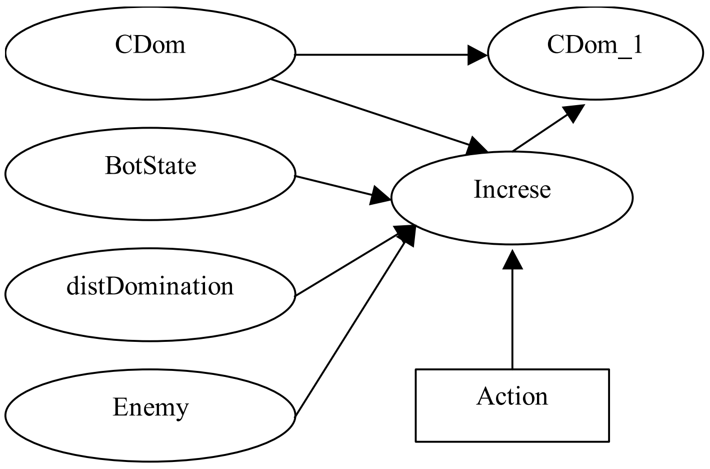


Figure 37 future optimisation

The other optimisation should be done to the Health_1, Armor_1, Weapon_1 and Ammo_1 (in all time slices) as they have the most parents. The parent divorcing could be used (3.6).

6.6 *Detail description*

This section will have a detailed description of each node in the network and a figure of the final influence diagram used in the Rbot.

6.6.1 The description of variables used in final influence diagram of the Rbot

In the Table 14 we explain the meaning of each node of the final influence diagram. It also contains the names of the node states and short description what it means. The index "_n" means that those nodes belong to the different time slice. The nodes with no index means the current time.

Node	States	Description
Health, Health_1	Good Normal Poor	This node represents the state of Rbots health. The health values are translated in the following way. If health state is: (0..50] = Poor (50..100] = Normal (100..199] = Good
Armor, Armor_1	Good Normal Poor	This node represents the state of Rbots Armor. The armor values are translated in the following way. If armor state is: 0 = Poor (0..25] = Normal more than 25 = Good
Weapon, Weapon_1	Good Normal Poor	This node represents what kind of weapon Rbot is carrying. The weapons are translated in the following way. Low damage weapons (like enforcer, impact hammer) = Poor Medium damage or rapid fire (like pulse gun, shock gun) = Normal High damage or explosive (like flak cannon, UT_Eightball) = Good
Ammo, Ammo_1	0 1	This node represents Rbots ammo state and is translated as: Less than 10% of max available ammo = 0 Else = 1
CDom, CDom_1	0 1 2 3	This node represents how many domination points the Rbot is controlling. The state name represents the number of domination points. Because the Rbot was tested in the environment where max number of domination points is 3 it has only 4 states (1 state for 0 domination points)
distDomination, distDomination_1	Near Far	This node represent the Euclidian distance measured in UT units to the nearest domination point and is translated as follows. Less than 500 units = Near Else = Far The reason for choosing 500 units is that the experiments showed that this is the average distance that can be covered by Rbot during the time while strategy is executed.
distHealth, distHealth_1	Near Far	Same as distDomination except the distance to nearest health or armor (see Action node for more information)

distWeapon, distWeapon_1	Near Far	Same as distDomination except the distance to nearest weapon or ammo for that weapon (see Action node for more information)
Enemy, Enemy_1	Far Near Unknown	This node represents the distance to the enemy if it is in sight. If the enemy is not in sight state unknown is used else if Distance to enemy > 500 = Far Distance to enemy <= 500 = Near
Die	Yes No	This node represents if the Rbot died.
Kill	Yes No	This node represents if the Rbot killed its enemy
Action, Action_1	Camp GetHealth GetWeapon GetDomination Ambush Hunt	This node represents the actions that the Rbot is capable to make or already made. More detailed explanation how actions are carried out see Table 15.
U1, U2	Utility node	This node represents the utility function of Rbot. The utility function is very simple. It depends only on number of domination points the Rbot possesses. Dom: 0, 1, 2, 3 Utility: 0, 1, 2, 3

Table 14 the explanation of the influence diagram nodes

The nodes “Stamina”, “Strength” and “BotState” were introduced after performing parent divorcing in optimisation step. Those nodes are fixed and do not change over time (no fractional updating is performed on them).

6.6.2 Explanation of actions that are implemented for the Rbot

The Table 15 gives the explanation how each action is performed in the Rbot.

Action name	Description
Camp	While performing this action the Rbot picks a random node from a set, which includes all controlled or neutral domination points and moves there. After arriving to the point that was selected the Rbot starts walking around that point. In case the camp node is reached and enemy is in sight it also starts firing upon enemy.
GetDomination	This action picks the closest uncontrolled domination point and moves the Rbot there. If no uncontrolled domination points are present no action is taken (the Rbot stands still).
GetHealth	During this action the Rbot goes picks up the nearest health or armour. The nearest health is picked. The health must be of such type that it would increase health of the Rbot (for example if Rbot has 50 health the health vial and medbox and armour will be considered but if Rbot has 100 health only health vial and armour will be considered as medbox is able to increase the Rbots health

	to 100 units only).
GetWeapon	During this action the Rbot will try to get a better weapon or ammo for a current weapon whichever is closer.
Ambush	During this action the Rbot will try to go to the spot which is not visible by enemy or if the enemy is not in sight the Rbot will go to a spot that is near domination point but not reachable directly (the point will be near domination point but not visible from that point). After reaching that point the Rbot will wait for an enemy and as soon as it appears the Rbot will start shooting.
Hunt	The hunt action starts fight with an enemy if it is in sight else the Rbot will wait for an enemy to appear at the location where hunt action started.

Table 15 explanation of the possible Rbots actions

6.6.3 The final influence diagram used in the Rbot

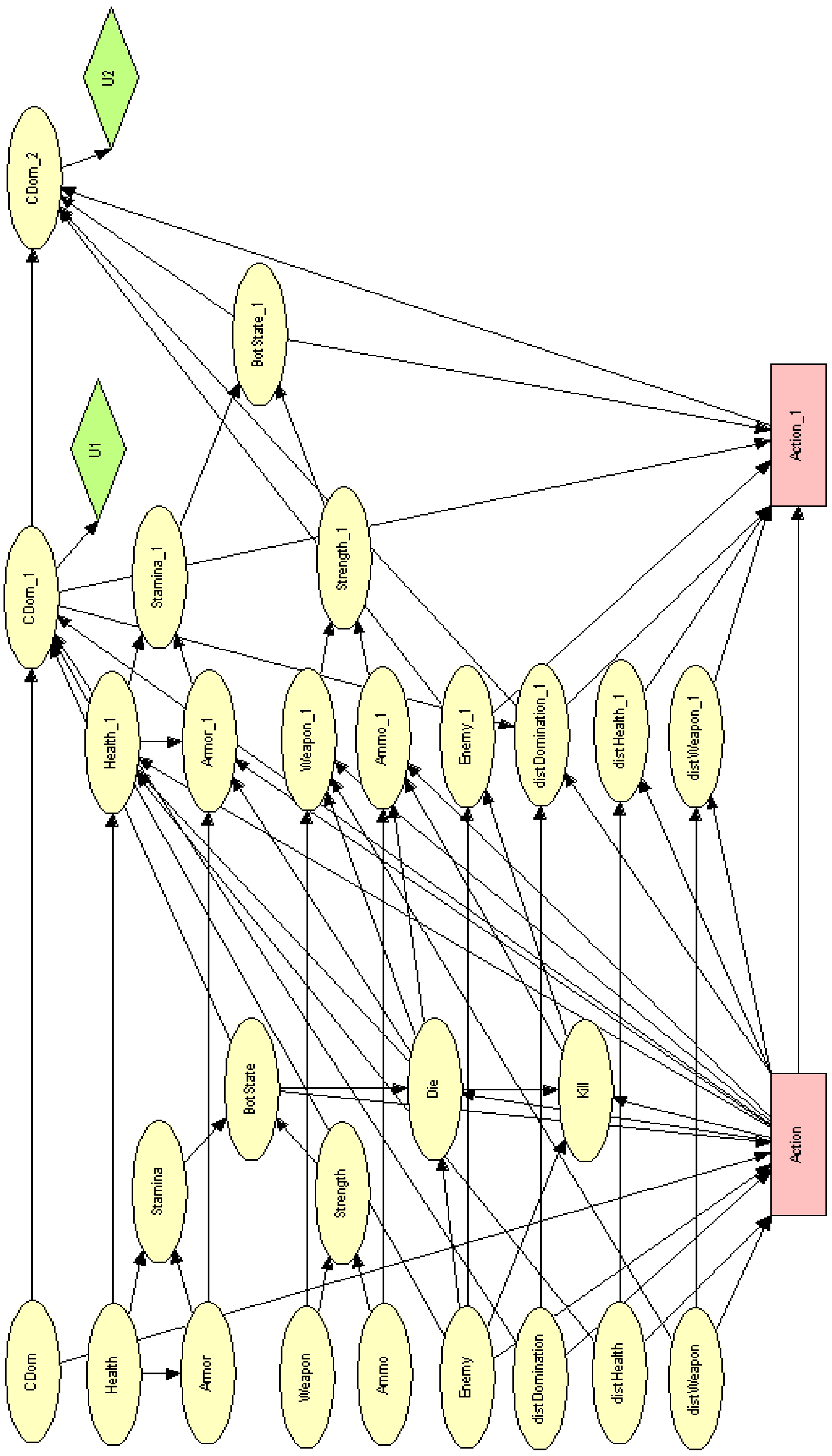


Figure 38 the final influence diagram used in the Rbot

Chapter 7 Test Results

This chapter contains the description of the experiments that we performed. First we describe the test setup and then we present the results of experiments. We compare the Rbot to the UT bot and human using the achieved score and we investigate in detail two main strategies developed by the Rbot. We end with conclusions.

7.1 *Test setup*

The tests were run on 566Mhz Intel Celeron machine. All of the software was run on a single machine. The software needed includes:

- Unreal Tournament game – an application which provides us with the testing environment
- Gamebots platform (modified) – a modification for UT which provides a communication with UT application
- Java 1.3 – a programming language and a run time environment
- Hugin API 5.3 – a tool for handling the influence diagram operations like updating, finding optimal decision.

The DOM-Stalwart map for UT was used for each game. The time limit for the game was set to 5 minutes and the score limit was set to 999. The rules are the same as in UT domination game type.

All of the results are the averaged from several games. The mean is calculated using following formula:

$$\mu = \frac{\sum_i x_i}{n}$$

The variance was calculated as:

$$\sigma = \sqrt{\frac{\sum_i (x_i - \mu)^2}{n-1}}$$

Here x_i is the sample result and μ is average result.

The number of games (n) will be stated in each test.

7.2 *Rbot versus UT bot*

The UT bots are well known to the 3D shooter players of their strength (ability to fight) and human like behaviour. The UT bot is written in UT scripting language and does not include any special AI routine. It is not capable of learning thus it can be considered that its strength is always the same. This property of UT bot gives us a good opponent for testing Rbots ability to adapt to the environment and opponent. Because the UT bot does not evolve it can be used as a benchmark.

This test should show that the Rbot is capable to adapt to the environment after receiving information about its own actions. In order to show this ability several tests must be run. First test includes the Rbot with uniform prior distributions (all the actions gives the same utility). Such probability distributions will make Rbot to make random actions. By making the random actions the Rbot is not expected to win the game. Later the information received from a game will be used to update the Rbots network and we will call that Rbot generation 1 (first Rbot will be called generation 0). The test should be carried out until there will be no improvement in difference between the Rbots and UT bots score.

Several approaches of the getting the next generation Rbot were used. The following approach was selected:

- Perform 10 games with the generation 0 Rbot
- Use information from 10 games to update the network and get generation 1
- Perform 10 games with the generation 1 Rbot
- Use the information from each game from previous step to create 10 generation 2 Rbots. Perform games with 10 generation 2 Rbots
- The information received from the games is used to create 10 next generation Rbots.
- 4 Generations showed to be enough

The reason for using all the information from 10 games in the beginning is that the data received from 1 game is not enough to correctly update initial network. The observation was made that some Rbots were making the wrong decision. Of cause the wrong decision is fixed during the later games but then it takes more time to adapt. If we provide the Rbot with enough prior data its decision become more reasonable. Around 1300 cases were used to update the initial Rbot. Later each game produced around 132 cases.

The other approach was to use always all the information received from 10 test games. This approach causes faster adaptation to the environment but the scores are similar.

Figures below will show the scores of each generation Rbot versus UT bot.

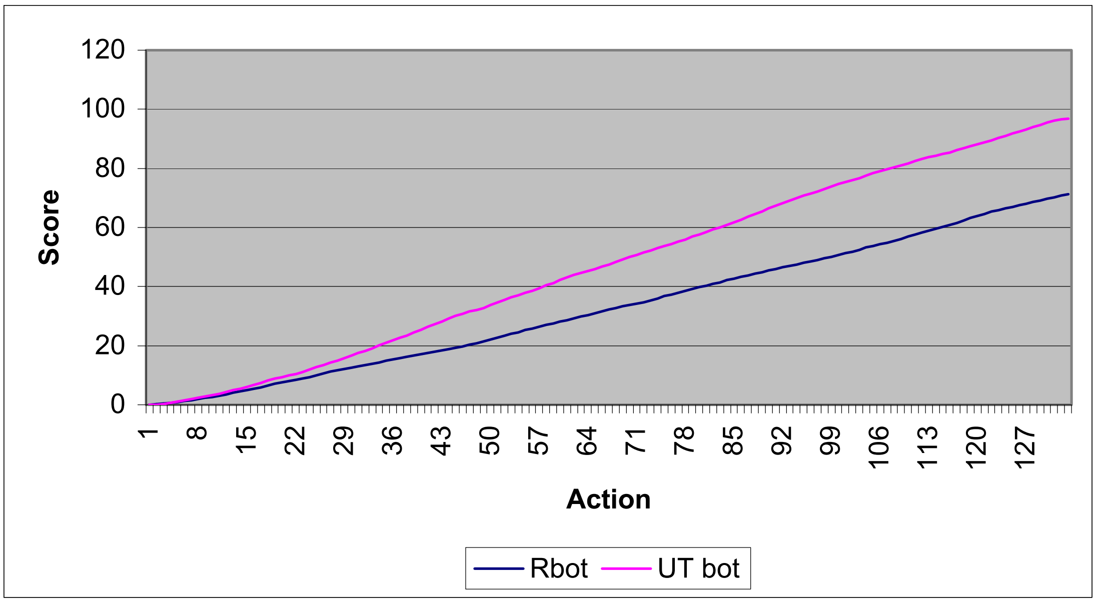


Figure 39 generation 0 Rbot versus UT bot

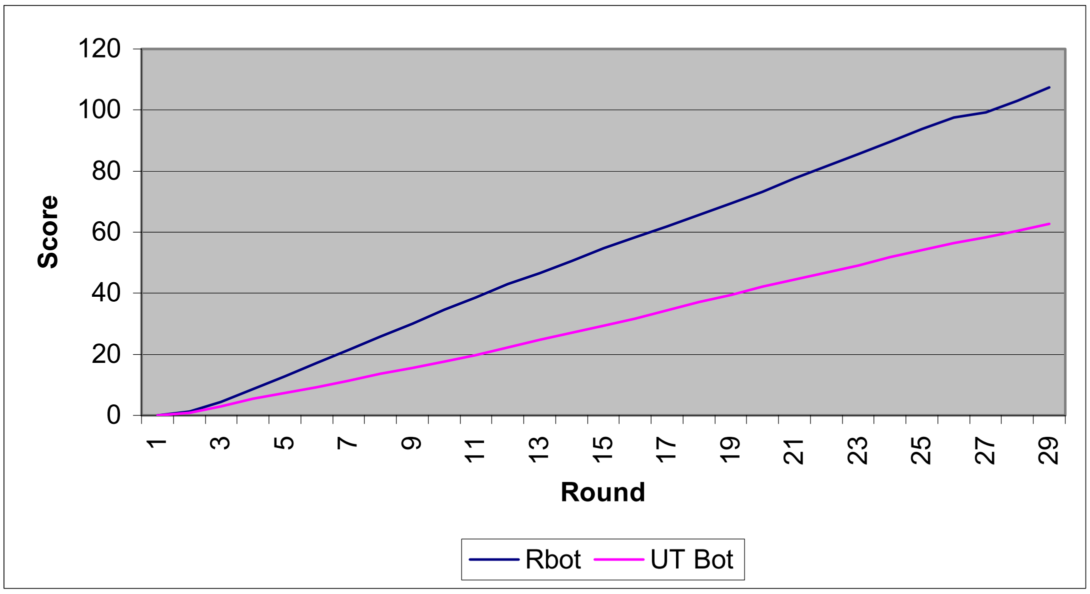


Figure 40 generation 1 Rbot versus UT bot

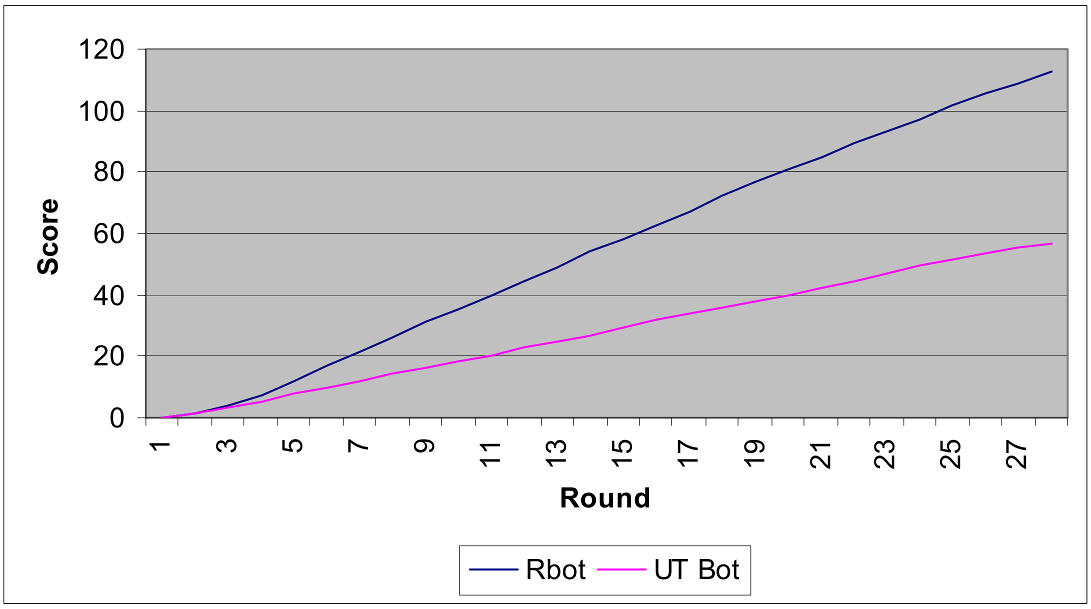


Figure 41 generation 2 Rbot versus UT bot

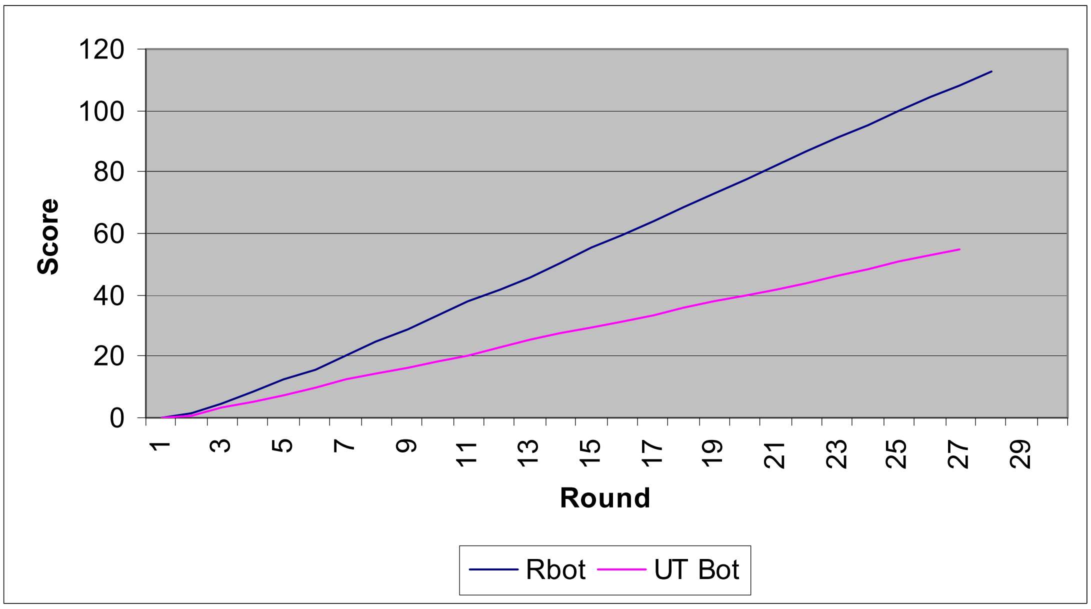


Figure 42 generation 3 Rbot versus UT bot

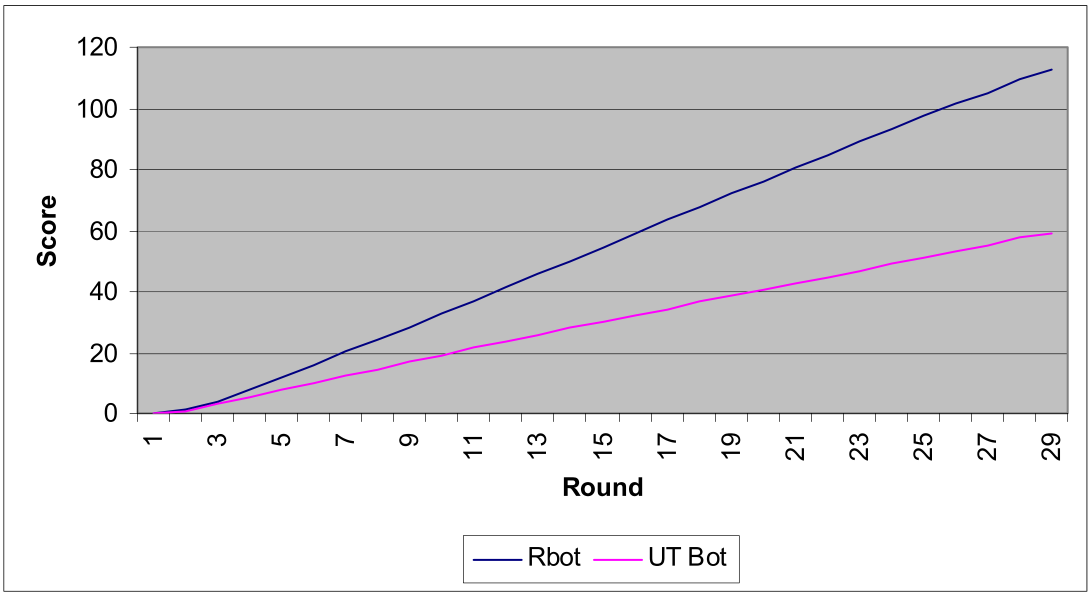


Figure 43 generation 4 Rbot versus UT bot

The decision quality depends on the number of samples that are available for updating the influence diagram. The more cases Rbot gets the better its performance and the Table 16 shows that.

Rbot type	Sample size used
Generation 0	0
Generation 1	1321
Generation 2	1453
Generation 3	1585
Generation 4	1717

Table 16 Rbots sample size

The reason why the later generations does not need such a high number of samples than the generation 1 is that after getting some insight about good decisions Rbots makes more sensible decisions and only few situations causes to change its belief about a best action.

The final Rbots results with their variance are presented in Table 17.

Rbot type	μ±σ
Generation 0	71.2799 ± 20.3961
Generation 1	107.3195 ± 10.9483
Generation 2	112.6794 ± 6.6833
Generation 3	112.7794 ± 5.7952
Generation 4	112.6594 ± 8.3169

Table 17 final score of the Rbot

The next picture will show how the Rbots score changes compared to the UT bots score.

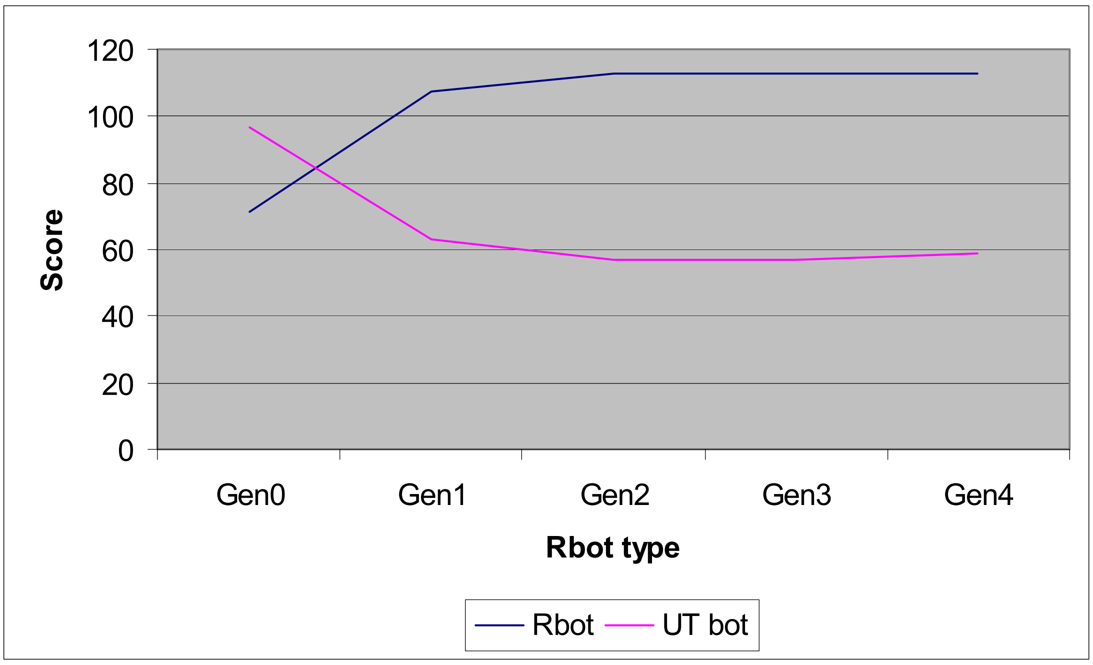


Table 18 Score change for Rbot and UT bot

The next chart shows how did the score difference of Rbot and the UT bot changes as Rbot evolves.

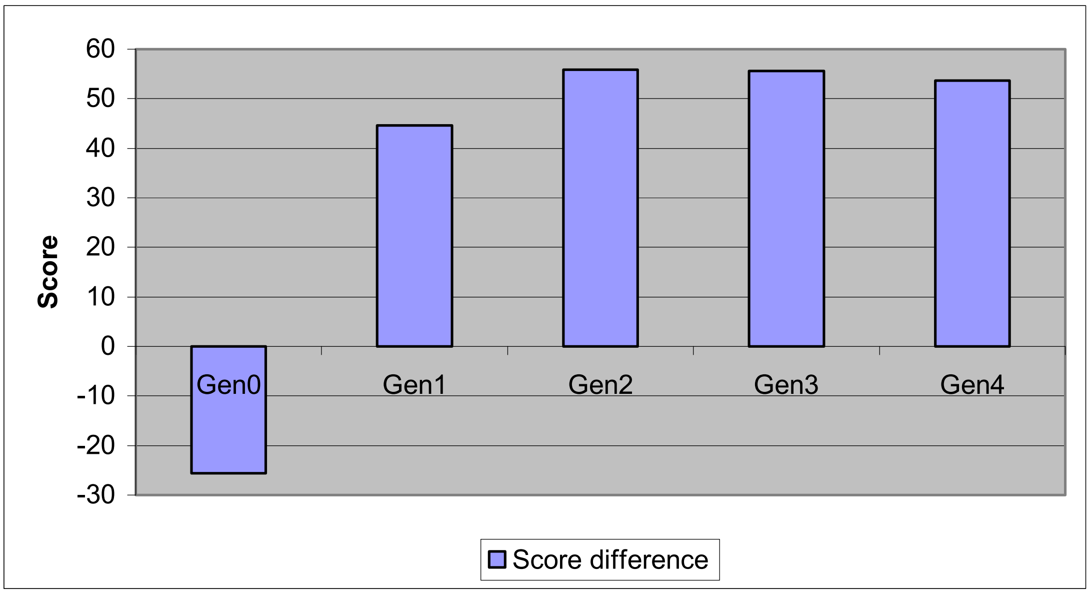


Table 19 final score difference between the Rbot and UT bot

As we can see from the charts presented above the Rbot is capable to win against the UT bot already after it gets information about the first game. Later only some improvement can be detected and then the score difference stabilises and has only small fluctuations. The reason for reaching the stable point is that the Rbot is not capable to keep all three points all the time and even killing opponent he reappears in another place where he can take a domination point.

The results also show that the experience is very important in the beginning. The reason for this is that the UT bot does not change its strategy and the more information the Rbot has the better choice of the decision it can make. No fading was used when testing against the UT bot but later the results will show that fading has some influence on the results when playing against opponent, which changes its strategy.

7.3 *Detailed game analysis of the Rbot vs. UT bot*

In order to check whether the Rbot is making sensible decision a detailed observation of several games were performed. The actions taken by the Rbot were also recorded. The results showed that Rbot is not just trying to get a domination point all the time.

By watching the games two different strategies evolved.

The figure Figure 44 and Figure 45 presents the frequency of an actions that are performed during the game by generation 4 Rbot.

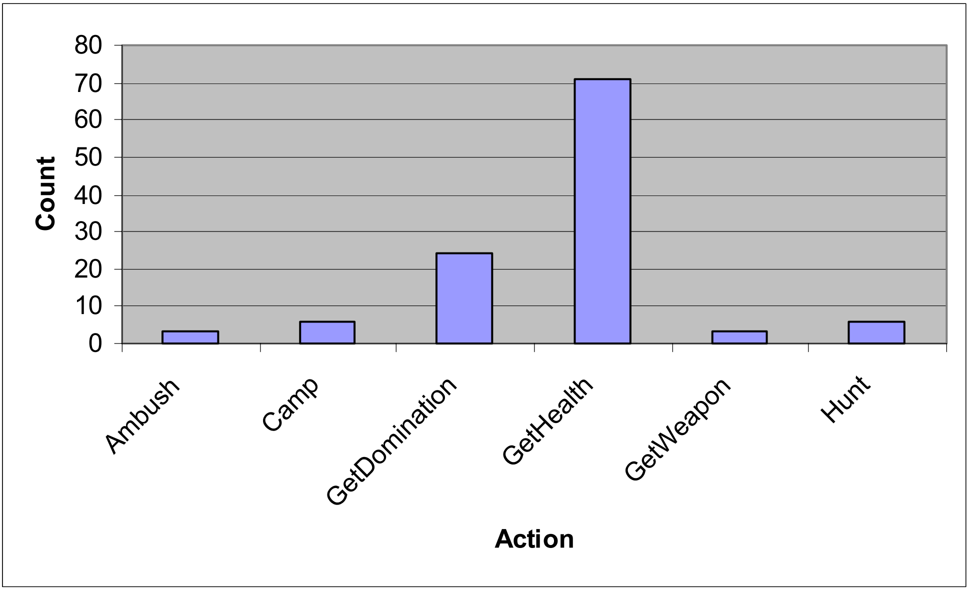


Figure 44 strategy 1

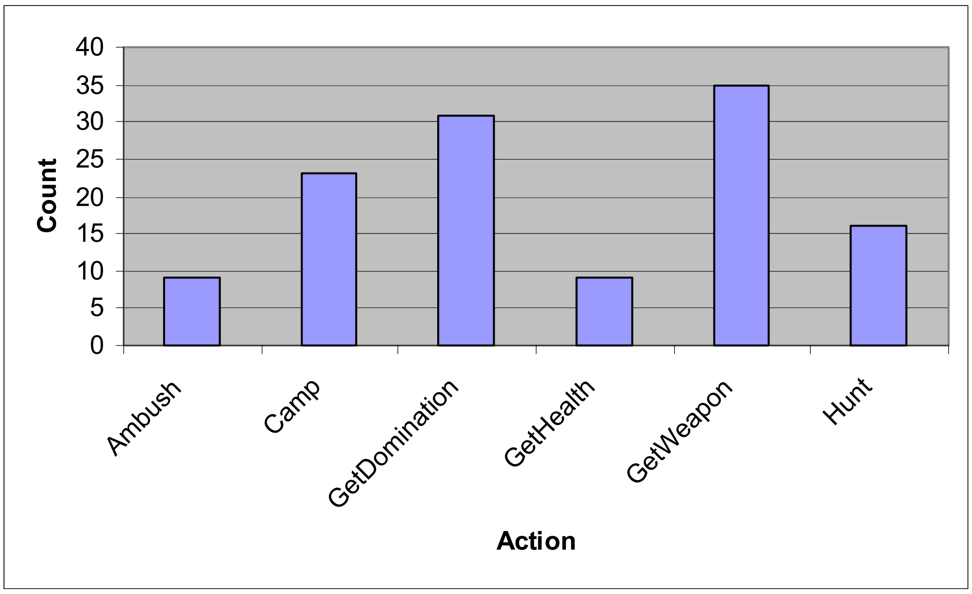


Figure 45 strategy 2

7.3.1 Strategy 1

In Figure 44 the frequencies of actions of the first strategy are presented. By observing the Rbot the strategy includes following main actions:

- Go for domination until Rbot has two of them
- After controlling 2 domination points go for a health.

This causes Rbot to pick two of domination points and then pick up the items that increases its health. The other actions are caused only when the opponent is seen. What happens is that usually the UT bot would protect only 1 domination point and will not wonder around. After the Rbot gets 2 domination points by going to the health it manages to stay clear of the enemy and accumulate points.

7.3.2 Strategy 2

In Figure 45 the frequencies of actions of the second strategy are presented. By observing the Rbot different behaviour was found. In this strategy Rbot has following main actions:

- Go for domination points until 2 acquired
- Go for third domination point if no enemy in sight
- If enemy is seen and weapon is close get the better weapon
- If enemy is seen and no weapons or health around fight the enemy
- If close to domination point just go for it

This strategy makes Rbot to go for two domination points and then try to get the third one. It usually involves fighting so the action to pick better weapon if it is near is reasonable. If Rbot is lucky then it will be able to pick the third domination point.

The average scores received by using strategy 1 and strategy 2:

Strategy	Score
Strategy 1	112.0794
Strategy 2	113.2394

Table 20 average score using different strategies

However by checking the final scores using those strategies (presented in Table 20) we can see that the difference between them is very small. This is because when the Rbot does not engage the opponent there is no chance to kill him and the score steadily increases. In strategy 2 the Rbot is more likely to engage another bot and possibly kill him. This makes opponent to appear in another position of the map and get the nearest domination point thus again decreasing Rbots number of controlled points. Such behaviour would also cause opponent to move more thus get more domination points.

The figures below show how the score increase between Rbot and UT bot changes using different strategies.

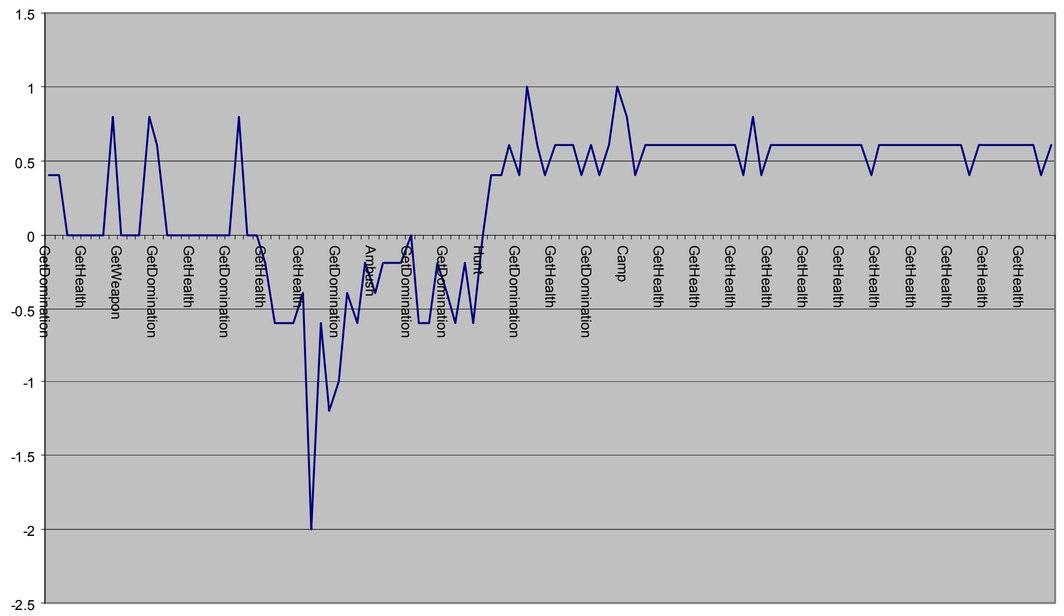


Figure 46 The change of score increase using strategy 1

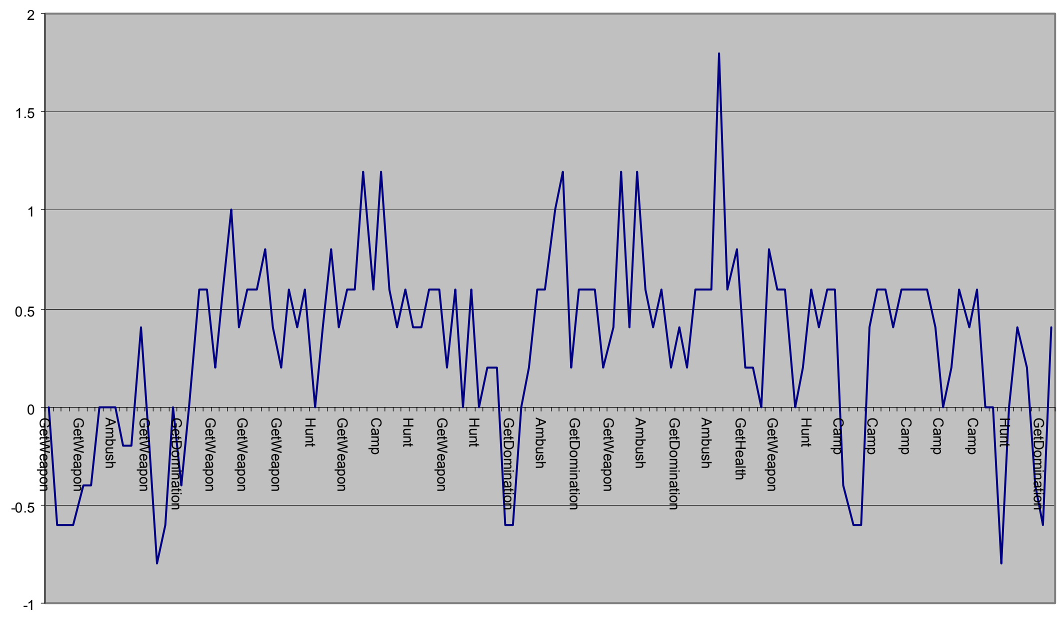


Figure 47 the change of score increase using strategy 2

As we can see from Figure 46 and Figure 47 that the strategy 1 gives a steady increase in points after two domination points are captured and that the strategy 2 does not give a steady increase but is capable of giving higher increase during some points (capture of 3 domination points).

The Rbot, which used all of the information from previous games, did not learn strategy 1 but it learned strategy that is similar to strategy 2.

The number of action counts while the Rbot evolves is presented in Figure 48.

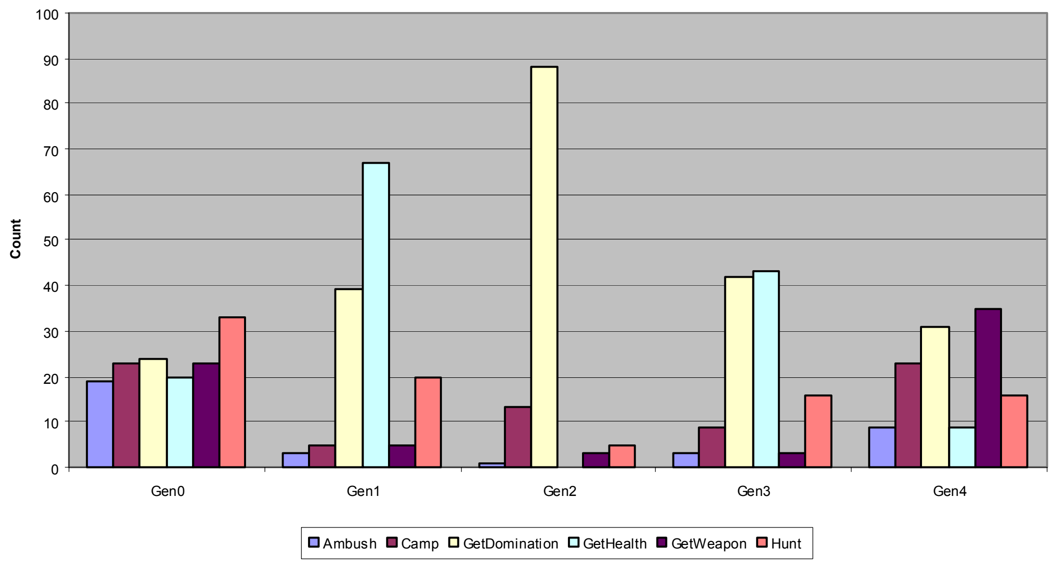


Figure 48 action count of different generation Rbot

7.4 *Rbot versus Human*

This test was carried out against human player. The human player was the author of this project and posses a good knowledge about the UT bot and a good knowledge about the Rbot. The human player should be able to adapt to the enemy strategy quickly. Five games were played against UT bot and against the Rbot. The Rbot was taken of the 5th generation. That is first the human player played against 1st, 2nd … until 5th generation then 5 games against 5th generation Rbot. Also the tests were done without using fading and with using fading of 0.9.

During those games it was observed that the Rbot was more strategic player than the UT bot and made a game more difficult. Although the UT bot was much more advanced when fighting. The results of those tests are presented in (scores are average from 5 games).

Enemy type	Human score	Enemy score
UT bot	122	42
Rbot (5th generation no fading)	111	54
Rbot (5th generation with fading)	105	57

Table 21 human player against the Rbot and UT bot

As we can see from final result that the fading did have influence on the score. The reason is that the initial probability tables were learned from ~1320 cases of the games that were played by the Rbot against UT bot and not the human player. As the strategy is different the fading was able to reduce the influence of those cases and enabled the Rbot to choose better decisions.

7.5 *Detailed analysis of the game the Rbot vs. Human*

The following figures show the frequency of the different Rbot actions when playing against the human player.

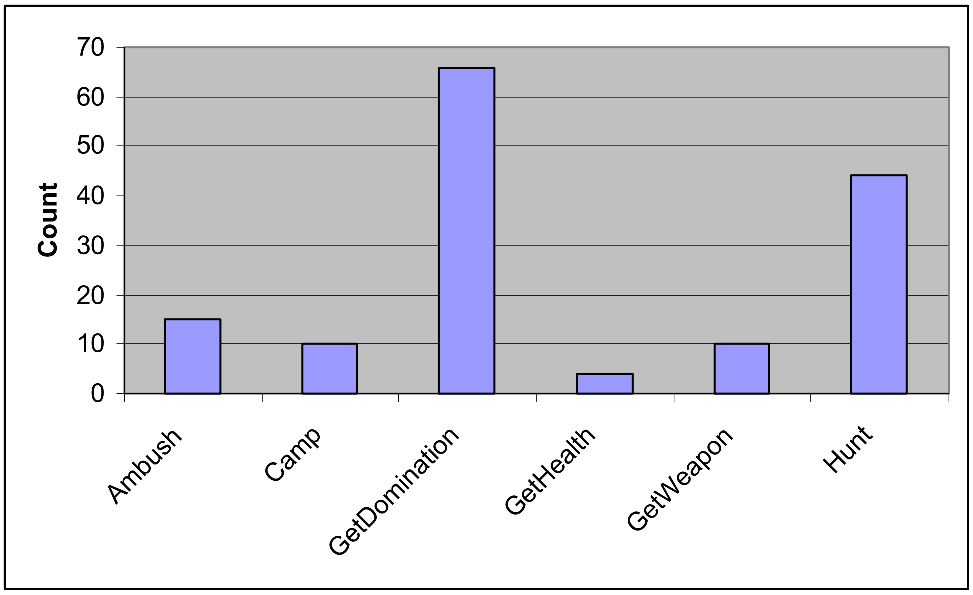


Figure 49 Rbot (no fading) against Human player

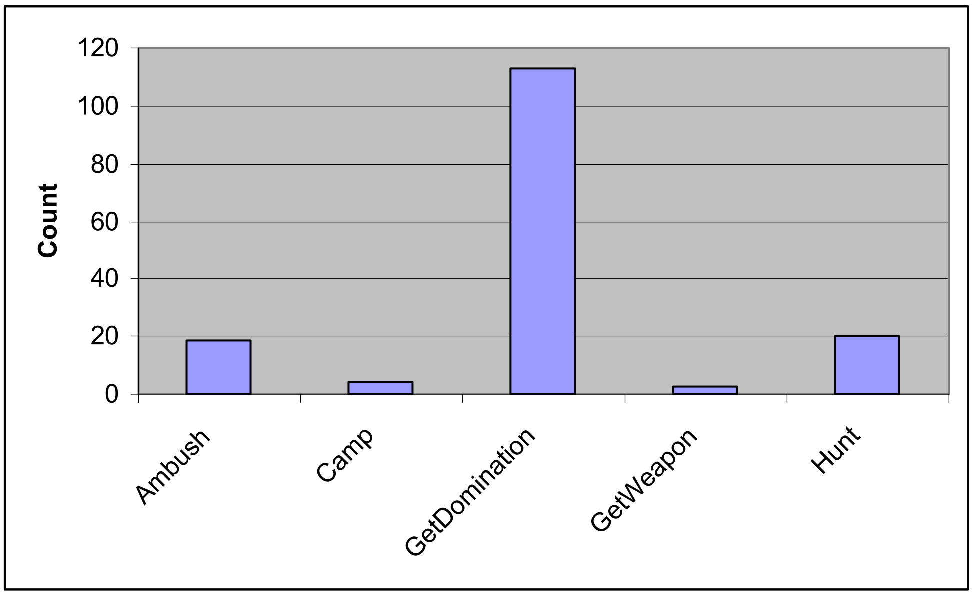


Figure 50 Rbot (with fading) against Human player

After examining the play of the Rbot one strategy could be pointed out and it is:

- Go to the domination point.
- If enemy is in sight then try to kill it.

The other actions of the Rbot could not improve its performance as the Human player was much stronger in fight and did not stand in one place. Increasing the health or getting the better weapons did not increase Rbots chances to survive considerably thus this strategy to go for domination point and shoot when possible (shooting does stop the human player from recapturing the domination point for a while) seem to be reasonable. And from Figure 50 we can see that the Rbot with fading did learn such strategy faster than the Rbot without fading.

7.6 *Result conclusions*

The results showed that the new type of the Rbot does outperform the older one in all aspects.

After performing the test of Rbot against UT bot we can see that the current Rbot (generation 4) won the game 112.65 against 58.96 (previous semester Rbot (generation 4) 101.80 against 76.20). We also described that typically one of two strategies evolved. Both of those strategies make sense in the context of the considered game. But after examining results we can say that the 1st strategy evolved only specifically for the UT bot. The strategy 1 exploits the weak point of UT bots play, which is to guard one domination point and rarely go for others. The different action execution than the previous Rbot showed that the strategies of the Rbot can be more complicated and change as the environment changes (previous semester Rbot would choose the strategy and would continue with it even if it becomes not the best one).

After tests against Human the current Rbot also performed better than the previous Rbot (current Rbot 57 against 105 and the previous semester Rbot 49 against 117). We could also observe that the Rbots strategy learned against Human is similar to the second strategy learned against the UT bots except some actions like get weapon and camp were very rare. This could be explained that the better weapon did not influence the fight against the Human and camping action also includes some fighting. It was also harder to play against the current Rbot that the previous version of the Rbot.

Further improvement of actions (low level implementation like path finding, fighting, evasion) could lead to even better results. Moreover the Rbot is currently not capable to use all the information that is available to the human player, thanks to limited capabilities of the UT game. Also adding the capability to remember the position of the enemy could improve the evasion and improve score. The additional timeslices could also enable the Rbot to learn even more complicated strategies.

Chapter 8 Conclusions and Future Work

8.1 *Conclusions*

Fully functional agent for UT game was created. Although the Rbot is not capable to win against the human player it proved to be better than the currently built in UT bot. The results also show that the Rbot is capable to learn a strategy and use it successfully and in case of a bad choice it is capable to learn a different one. Even without predicting the enemy the Rbot is capable to show intelligent behavior. It also proves that Bayesian networks are very powerful tool for a decision making if used correctly. Because the Bayesian networks are not very frequently used in real time environment it is very hard to compare with other works done. The problem addressed in this project was to explore how would it be possible to apply Bayesian networks in the real time environment. Although the Bayesian networks does not fit in such places as the fighting in the game because the behavior is very erratic and short term it still fits very well in creating the long term strategy of the agent. We think that the aim of the project, which is to show that it is possible to apply the Bayesian, networks even in highly dynamic environment and that an agent exploiting Bayesian network is able to adapt to the environment and make sensible actions, was achieved. Because the project was done for two semesters two creation techniques and their results were presented. The results also proved that the second design was more successful and is more general for such kind of problems.

Some problems were also observed when creating the Bayesian networks. They tend to become very complicated very quickly so some optimization is necessary. Also the selection of observable variables must be chosen very carefully as there are too many variables to take into account in games like UT. The correct selection of the variables will give a structure, which is more sensitive to the environment and gives better decisions.

8.2 *What was done*

In order to be able to communicate with the UT environment the whole structure based on the Gamebots had to be created. The communication between Gamebots platform was created with reusability in mind. The whole platform for testing AI based on any Bayesian network was created.

The next target was to explore the way to put the influence diagram to work. Two ways of constructing the influence diagram for the Rbot decision making were proposed. One approach was to use the information about the current situation of the Rbot and use several utilities to make the Rbot choose different strategies under different conditions. The second approach was more general which also employed the structure learning algorithms to help the expert to create the influence diagram. The dynamic Bayesian network approach was chosen and only one well-defined utility function used. The problems that arose were discussed and their solution given.

After introducing the two different influence diagrams the tests were carried out to test their performance. Several tests were run to check the efficiency of those two approaches. The conclusions that can be done from the tests were presented in previous chapter. The tests were done against the built in bots of the UT game and against the human player. The ability to adapt to the non-learning opponent was presented. Also the test results shows that the Rbot is much better than the UT bot. Also the results show that the Rbot is capable to reach similar results against the UT bots as the human player (human score 122 against 42 and the Rbot result 112 against 58). The further improvement of low level actions such as path finding or fighting algorithm could reduce the score difference even more.

Also the author learned how to address such problems and explored the drawbacks or the advantages of the different approaches such as the usage of dynamic Bayesian networks, application of the structure learning, Bayesian network optimization.

8.3 *Future work*

After making this project some ideas were still left unimplemented. Some of them are easy to apply and some of them need further investigation. The straightforward ones are:

One of the improvements could be done to the path finding algorithm. The observation of the human play shows that by executing some strategy the human player while moving to a certain point is capable to route its path through nearby items and pick them up without being distracted from the strategy he is performing. One way to do this is to use some distance modifier in the path finding algorithm to make the path to go through the nearby items. Also the Rbot should not use always the same path, as the human is capable to learn which way the Rbot will move and set up a successful ambush.

The other improvement could be done in the fighting algorithm. Currently the implementation is very simple and does not make a good challenge. It would be nice to use the UT fighting routine, as it is fairly good (however it is currently impossible) or possibly applying approach suggested by [17].

The other future work would need bigger changes and more research to be done. First of all the drawback of the Rbot is that it does not try to predict the enemy. The first semester work tried to predict the enemy but the efficiency was not very good. Some other approach then the one presented in the first semester work must be used. The opponent should be monitored through out the whole game to find out if it is moving around all of the level or it likes some certain spots. This could be done by recording the last places it was seen and use that information to predict its strategy.

The next observation that was done by testing the Rbot was that if it selects some action as the good one (mistakenly) it will continue to do it for some time (until the fractional updating with fading will show that it is not the best action). In order to prevent such problems some randomization into the Rbots actions can be added. It would also make it more unpredictable. One way of doing it could be to use some probabilities based on the expected utility of that action. The other way could be instead of using fixed probability number in the conditional probability table to use the distribution for it, which would cause the expected utility to change, and different actions would be selected. The problem with the randomization could arise in the execution of the actions. Because the action is performed for some fixed time too high randomization would prevent the Rbot to complete the action. Thus it would be desirable to have this random changes correlated in time in order to achieve consistent (and utility gaining) behavior.

Bibliography

[1] John E. Laird and Michael van Lent (2001). Human-Level AI's Killer Application Interactive Computer Games. Article from AI Magazine Summer 2001: 15-25.

[2] Game AI Page, http://www.gameai.com, 2002.

[3] John E. Laird (2000). It Knows What You're Going to Do: Adding Anticipation to a Quakebot. Agents 2001 conference. An earlier version first appeared in the AAAI 2000 Spring Symposium Series: Artificial Intelligence and Interactive Entertainment, March 2000: AAAI Tech.Report SS-00-02.

[4] Dicky Suryadi and Piotr J. Gmytrasiewicz (1999). Learning Models of Other Agents Using Influence Diagrams. Department of Computer Science and Engineering, University of Texas at Arlington 1999.

[5] Michael van Lent, John Laird (1999). Learning Performance Knowledge by Observation. International Conference on Machine Learning, June, 1999.

[6] Peter Stone (1998). Layered Learning in Multi-Agent Systems, Ph.D. Thesis, Computer Science Department, School of Computer Science, Carnegie Mellon University, December 1998.

[7] Finn V. Jensen (2001). Bayesian Networks and Decision Graphs. Springer Verlag, New York.

[8] David Maxwell Chickering (2002). Optimal Structure Identification with Greedy Search. Microsoft Research, 2002.

[9] Sprites Peter, C. Glymour, and R. Scheines (1993). Causation, Prediction and Search. Lecture Notes in Statistics 81. Springer Verlag, New York.

[10] Planet Unreal Page, http://www.planetunreal.com/, 2002.

[11] Gamebots platform page, http://www.planetunreal.com/gamebots/, 2002.

[12] Brian Mayoh on behalf of Eric Bonabeau, Swarm Intelligence, EVALife workshop 2001 (http://www.evalife.dk).

[13] N. J. Nilsson (1982), Principles of Artificial Inteligence, Springer Verlag. Berlin, 1982.

[14] Amit's Thoughts on Path-Finding (2002), http://theory.stanford.edu/~amitp/GameProgramming/

[15] B-Course (2002). Complex Systems Computation Group, Department of Computer Science, University of Helsinki (http://b-course.cs.helsinki.fi/), Version 1.0.2.

[16] Hugin (2002). Hugin API reference manual. Hugin Expert A/S (http://www.hugin.com), Aalborg, Version 5.3.

[17] Jens Dalgaard Nielsen and Jørn Holm (2002). Genetic Programming – Applied to a Real Time Game Domain. Aalborg University, 2002.

Appendix A

The whole code for communication with the Gamebot platform was written.
A simple code for creating a bot that uses gamebots platform is given below.

```
import rbot.client.*;

public class Yourbot extends Bot implements DistinctMessageListener {

   public Yourbot(String name, int team) {
      super(name, team);
      init();
   }
   public Yourbot() {
      super();
      init();
   }
   public void init() {
         addListener( this, Message.GET_ALL);       //Shows that we want to receive
                                                    //all messages from gamebots
                                                    //platform
         //Add initialization of your bot here
   }
   public void distinctMessage(Message msg) {
         //Add handling of messages here
         if (Message.getType() == Message.ASYN_HRN) {
                  //Code below will be executed after receiving message HRN
                  //For sending a message do the following
                  Properties prop = new Properties();
                  prop.addProperty("Location","1,1,1");
                  Message ourmsg = new Message(Message.SHOOT, prop);
                  sendMessage(ourmsg);
                  //The obove 4 lines would make bot to shoot at location 1,1,1
         }
   }
}
```

This is all code that you need to write in order to be able to receive parsed messages from Gamebots platform. Sending messages to the Gamebots platform is made by calling the Bot class method (it is inherited by Yourbot class) sendMessage(Message msg);