INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Decentralized and Emergent NPC Coordination using Behavior Trees

Tiago Nunes Henriques

Dissertation submitted to obtain the Master Degree in

Information Systems and Computer Engineering

Jury

Chairman:	Nuno João Neves Mamede
Supervisor:	Mário Rui Fonseca dos Santos Gomes
Co-Supervisor:	Marco Paulo de Sousa Correia Vala
Members:	João António Madeiras Pereira

May 2012

Agradecimentos

Ao professor Marco Vala por me ter orientado ao longo das duas cadeiras de tese de mestrado. Foi desde o início até ao fim quem me ajudou a escolher o melhor rumo a dar ao meu trabalho e a focar-me no essencial, de maneira a fazer o trabalho o mais coeso possível.

Ao professor Mário Rui Gomes pela sua disponibilidade para analisar e corrigir o que fosse necessário nos documentos escritos.

Aos elementos do grupo GAIPS que me ajudaram com a construção e análise estatística dos questionários feitos.

A todos os que se disponibilizaram para fazer as sessões de testes e responder aos questionários.

Aos meus colegas que me fizeram companhia no GAIPS nos últimos meses enquanto trabalhava na tese.

A todos os meus outros colegas por termos juntos passado pelos vários desafios que ao longo do curso nos foram chegando, principalmente ao Diogo, à Sara, ao Tiago, ao Rui e ao João, pois foram mais do que colegas.

À minha família, por terem sempre confiado em mim e acreditado que mais tarde ou mais cedo acabaria o curso.

E principalmente aos meus pais, pois é graças a eles que aqui estou, por nunca terem pedido nada em troca e mesmo com algumas dificuldades terem pago por estes seis anos de universidade.

A quem me criou, educou, moldou
e tudo me deu,
Aos meus pais,
Muito obrigado por tudo.

Abstract

Artificial Intelligence in videogames is being more used nowadays than it was before and is notorious that producers are investing more on this subject. An area that needs to have better AI is definitely coordination between individual characters, because it makes players to identify and be more immerse with the Non Player Characters in the game. With this in mind, we wanted to experiment, in a real videogame, a different approach of doing coordination between NPCs. So, the aim of this project is to use Behavior trees, a recent structure introduced in the game industry, to enable coordination in a decentralized way. With this approach we are giving more focus on the characters behaviors and more importance to each character individually and consequently create more interesting characters for the players. The StarCraft commercial game was used to show and evaluate our implementation.

Resumo

A inteligência em videojogos está a ser mais usada nos dias de hoje do que foi no passado e é notório o investimento que os produtores têm feito nesse sentido. Uma área que necessita de ter melhor IA é sem dúvida a coordenação entre cada personagem, isto porque faz com que os jogadores se identifiquem mais com as personagens-não-jogador (NPC) do videojogo. Com isto em mente, quisemos experimentar em ambiente real de um videojogo, uma abordagem diferente para coordenar os NPCs. Portanto, o objectivo deste projecto é usar Árvores comportamentais, uma estrutura que foi introduzida recentemente na indústria dos videojogos, para permitir a coordenação de forma descentralizada. Com esta abordagem nós estamos a dar mais foco aos comportamentos de cada personagem e mais importância a cada personagem individualmente que por sua vez criam personagens mais interessantes para o jogador. O jogo StarCraft foi usado para mostrar e avaliar a nossa implementação.

Keywords

Behavior tree

Emergent coordination

Non Player Character

Behavior

Game AI

StarCraft

Palavras-chave

Árvore de Comportamentos

Coordenação emergente

Personagem-não-jogador

Comportamento

IA em jogos

StarCraft

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

In the videogame industry, a game to succeed should have good graphics, animations and gameplay. For some types of games, those aspects can be enough, but nowadays, almost every game needs to have some kind of artificial intelligence (AI). Although the scenario nowadays is more cheerful, AI has been one aspect of game development in which the producers usually did not wanted to risk too much on and therefore AI in videogames have been increasing slowly. However, some AI technics are already being used in videogames for a long time, for instance the A* algorithm (1), used for characters path finding and Finite State Machines (FSM) (2), used to control characters' behaviors in a game. For instance Lionhead Studios makes the difference with their games[1] because of the AI they have, which is the case of Black and White and Fable series.

Fact is that when the players see more realistic graphics they also expect more realistic behaviors. For instance when Crytek released Crysis[2], it had the most amazing graphics, but even though it was not bad, the character's AI was not at the same level.

More recently, in an attempt to improve AI in game development, **Behavior Trees** (3) were introduced in the industry. This new structure is an attempt to improve FSM and they seem to be earning an important place in the industry, by simplifying and making easier to implement behaviors for in-game characters. And this is a good improvement because it helps a lot the behavior designers, because is a lot easier to make new behaviors when using behavior trees, especially if a good editor is used.

However, there are still some issues that are not trivial to resolve with behavior trees. One of these problems is the **coordination** (4) between two or more Non Player Characters (NPC), which current implementations do not handle well, because they always require an additional structure to deal with the problem.

Having in mind that the industry is increasingly using behavior trees and coordination is something that the players want to see more and more in the videogames they play, it would be ideal to achieve an implementation of behavior trees, which maintain its simplicity and modularity but also be able to handle decentralized coordination. Moreover knowing that almost every approach taken, about the coordination of NPCs, is based on centralized entities and it would be interesting to see if a decentralized approach could give interesting results and maybe even better results.

1.2. Goals

Our goal is to achieve a behavior tree implementation that controls the actions of one NPC and make it cooperate with other NPCs without external structures to support it. In other words,

[1] Http://lionhead.com/
[2] Http://crytek.com/games/crysis/overview

we want to have decentralized coordination and at the same time create interesting behaviors that rely only on the knowledge of the unit itself without receiving explicit orders from another character or entity. This decentralization is important because it makes the coordination more flexible and oriented for each NPC enabling each character to have control over itself. This can lead each character to worry about things that are not usually seen in videogames where the AI is centralized, for instance the health points of the unit, which makes the unit to behave accordingly.

The big advantage of including coordination in the behavior trees is that they are already a decentralized approach and therefore we can make decentralized coordination easily.

It is also our intention to show that this approach can give the players a different and more satisfying game experience then the AIs that we usually see in games like real time strategy games. In order to do that, we will experiment our implementation in a well-known video game in an array of scenarios and also do some play testing to take conclusions at the efficiency level and the entertainment level.

1.3. Outline

This document is structured as follows.

In section 2 a background introduction will be made, where it will be explained what behavior trees are, how they work, what can be done with them and also some base knowledge of coordination and collaboration in agents.

In section 3 an overview of the technology and previous work will be made where it will be explained how behavior trees are being used, especially in videogames and what is already being done about coordination with them.

In section 4 we will introduce the StarCraft game, why we choose it to make our experiments and what features we will utilize. It will be explained how the original AI acts and what options we have to compare our AI with. It will also be explained how the connection between StarCraft and our code is made.

In section 5 is presented the architecture of the whole solution, the structure of the behavior tree and its implementation details, including the behaviors used and also the auxiliary classes that were made.

In section 6 it is explained how our solution was evaluated, including the scenarios between our implementation and the original game AI and the tests with people playing against those AIs.

Finally in section 7 an overall conclusion of the work done is made and also some ideas for what can be done in the future.

2. Background

Since this work is about behavior trees and coordination, a background introduction to these subjects must be done.

2.1. Behavior Trees

A behavior tree is a form of hierarchical logic, mostly based in Hierarchical Task Networks (HTN) (5), in which the root is the overall behavior, the non-leaf nodes represent simpler behaviors and the leaf nodes are atomic tasks that work as the interface between the behavior tree and game engine. Behavior trees have two types of leafs:

- **Conditions:** used to check something in the game world, for instance checks if the character has a long range weapon.
- **Actions:** correspond to the actual actions inherent from the game, for instance shoot the enemy.

The non-leaf nodes are therefore composite tasks which can be composed by atomic tasks and other non-leaf nodes. They are used to add complexity to the behavior tree by choosing and managing which branches of the tree should be executed. To do this, there are two basic nodes that can be used:

- **Sequences:** this kind of node is used when we want the child tasks to be executed one by one in a specific order, i.e. the first child runs and if it succeed, the next child is executed, if not, the node fails. If the last child runs and succeeds the node also succeeds.
- **Selectors:** this type is used when we want to have more than one child to choose from, depending on the situations, so the selector chooses one of the children to execute, if it succeed the node also succeeds, if not, the selector chooses another child to execute, if there are no child left, the node fails.

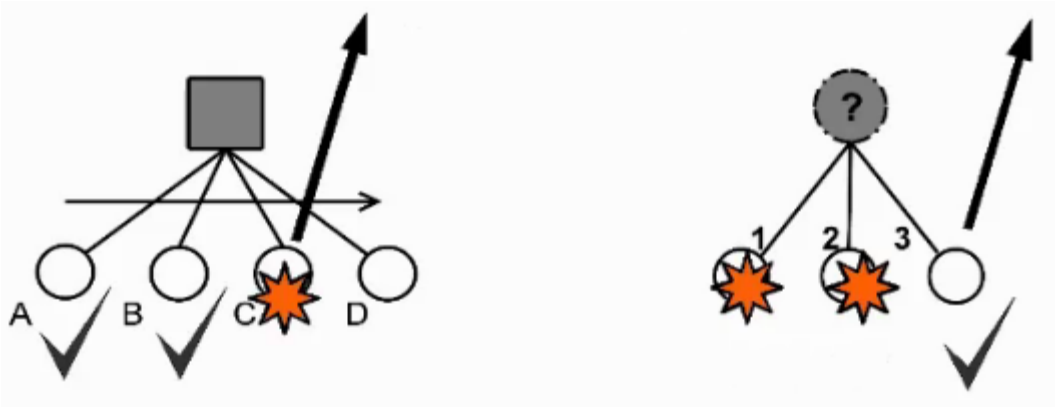

Figure 1 - Sequence (left) and Selector (right) nodes

As we can see, these two nodes are basically and-or nodes with which we can do very complex behaviors.

When the behavior tree starts running, it will be doing a depth first search through the tree, starting from the overall behavior (the root node) and ending when it succeeds. A condition succeeds if what it is checking is true. An action succeeds if the correspondent function in the game engine returns true.

In **Figure 2** we can see a behavior tree in a middle of its execution, where the root node (a selector) chooses to run its left child (a sequence node). Then this child has to execute both of its children one at a time. The left one chooses one action which succeeds, the right one chooses another sequence node which starts by executing its first action but fails, so the sequence also fails and the selector has to choose the other child to run. If this action succeeds, the selector succeeds and propagates up the tree to the root, which also succeeds.

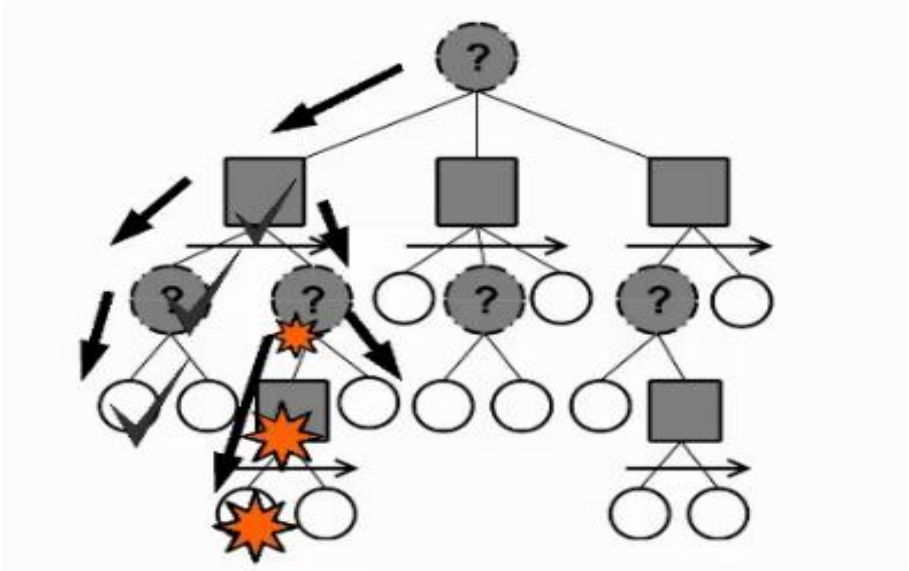

Figure 2 - Behavior tree execution eaxample

To do even more sophisticated behaviors, different types of selectors and sequences can be created, by choosing different orders of child execution. For instance, sequences execute their children in order, which can be left to right, can be ordered by different priorities which can change over time, or can even be random. In conclusion, it is possible to create any kind of selectors and sequences just by tweaking the order by which each child is executed.

And actually there is another type of composite node, the **parallel node**. This node is used if we want all the children to execute at the same time in parallel and we can have different configurations in order to this mode succeed or fail. This node can either succeed if any child succeeds or only if every child does and the same thing with failure, can fail if one child fails or only if all child fails. This type of node is particularly suited to check for dynamic conditions while another action is executing, or for example, just to play more than one action at the same time, for instance play an animation and a sound at the same time.

Besides these types of nodes and leafs there is another type of node that can be used in behavior trees. These nodes are called **decorators** and like the design pattern with the same name, this node just adds some functionality to that sub-tree without actually change its code. These decorators can do all sorts of things to the behavior tree, and it is flexible in the way that

we can anytime implement another decorator which does something different. The most common used decorators are:

- **Filters:** decides whether this sub-tree should execute or not according to some dynamic condition, for instance:
    - Counter: Limits the number of times that this sub-tree can be executed, i.e. when the given counter reaches zero, this sub-tree will never be executed again.
    - Timer: Only executes the sub-tree if the given timer expires, preventing the sub-tree being constantly executed.
- **Loops:** executes the sub-tree a given number of times.
- **Wrappers:** even if the sub-tree fails, the node will always succeed.

With these types of nodes it is much simpler to construct behaviors because we can program high level behaviors without thinking in little details like if it will be possible for this behavior to run, because those details are already programmed in the low level sub-trees.

One big advantage of using behavior tree is that they are easily understood even for non-programmers people, because they are goal oriented. With this in mind, it is possible to structure the tree in smaller sub-trees and even use the same sub-tree in more than one branch of the overall tree. This is done using a special decorator node, called **lookup node** and a **lookup table**. The lookup node searches in the lookup table the sub-tree that is designated to fulfill the desired goal (**Figure 3**). The lookup table has to be previously filled with every sub-tree, with each entry of the table being a goal that corresponds to a sub-tree. This sub-trees should have as root node a selector, so it can have multiple ways of achieving that goal.

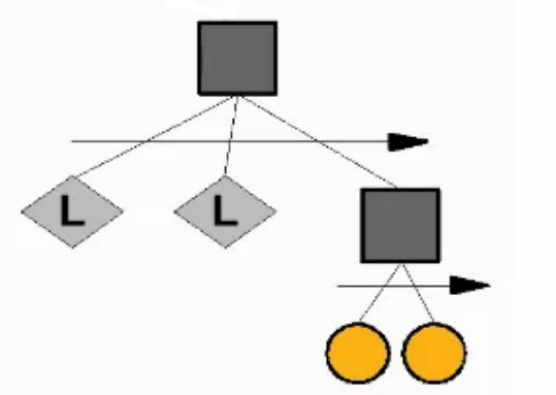

Figure 3 – sequence node with three children, two lookup nodes and another sequcence

In fact behavior trees can be seen as a hierarchy of task's goals, where the root node is the overall goal and the branches can either be different ways to accomplish that goal, for instance, we can think in a behavior tree which goal is "kill opponent", and the branches can be "kill with pistol", "kill with machine gun", "kill with grenade", or can be a sequence of tasks to accomplish that goal, for instance the branches of "kill opponent" can be "find opponent", "select weapon", "shoot opponent".

Although they are not used much until now, behavior trees are becoming increasingly used in videogame industry, not only because they are easy to understand and to work with, but also

because they are powerful enough to handle a large set of problems and difficulties normally encountered in game AI for characters. They are so powerful that they can be used in very different types of game, like first person shooters and real-time strategy and can even be easily used outside the videogame industry, in other researches.

Behavior trees were created to try to bridge the gaps that other hierarchical logic approaches used in games have, like FSM or scripting and less common HTN planner. The problem with scripting is the fact that introspection is difficult, so if something goes wrong it is not trivial what should be done. In other hand FSM takes a lot of work to edit, because of the big number of states and transitions. Planners are very heavy computationally because there is no easy way to re-plan. With behavior trees we not only manage to avoid other approaches cons as we can also avail their pros in one single structure.

2.2. Coordination

Agent coordination is a vast and complex subject and there have been many attempts to solve this problem even more nowadays because it is very common to see games where NPCs work together with each other or with the player itself.

Coordination is a very important subject in videogames (6) because the more times it happens and the better it is done more realism it gives to the scenes and more immersive will be the player experience.

Examples of scenarios where NPCs need to cooperate can be:

- **Movement coordination:** two or more NPCs have to move in the terrain in a squad formation.
- **Possible conflict tasks:** two or more NPCs need to go through the door and are trying to do so at the same time, but there are space just for one at a time.
- **Tactic maneuvers:** in a battle two or more NPCs have a strategy to flank the player.
- **Social interactions:** a handshake between two human NPCs or a conversation where one participant talks while the other listens and vice versa.

All of these examples are already present and can be seen in video games, but most of the times this is accomplished by scripting these behaviors manually, i.e. there is no real coordination because everything is preprogrammed.

Coordination can be thought at two different levels:

- **High level coordination:** this level is related to a strategic coordination that can exist between agents, i.e. coordinates what actions each agent should perform (e.g. tactic maneuvers).
- **Low level coordination:** this level is related to movement and “physical” coordination, i.e. coordinates the movements of the agent’s body so it does not enter in conflict with other agents (e.g. movement coordination).

In order to ensure coordination between agents it is necessary that each participant recognizes the behavior it is about to execute and knows what must be done at each moment. There are some approaches that have been developed to handle these problems and there are two basic ways of coordinating NPCs in a game:

- **Centralized:** there is a unique planner or another structure that tells each one of the agents what to do. This is very common in real-time strategy games, because typically they are more focused on groups of characters rather than on a single character.
- **Decentralized:** Each agent makes its decisions autonomously without receiving orders from an outside entity about what to do. An example of this technic is Craig Reynolds' BOIDS flock behavior (7).

In a decentralized approach each agent has to, in a certain way, **communicate** with the others. They can do it in an **explicit** way or not. Explicit means sending a message to a particular agent or to everyone. **Implicit** means to leave marks in the environment so other agents can sense and act based on them, which is also called **stigmergic coordination** (8). Usually the explicit way is more used, but if it would make sense in the context of the game, an implicit approach may be used.

A centralized approach has the advantage of not being too complex and be easy to program because there is just one entity controlling every NPC, but on the other hand it cannot handle well some specific situations like a NPC not being able to execute a determined order, for instance collisions with the scenario. Besides that with a centralized approach we are giving focus to a group, so if we have a game where we want to focus on individual NPC's behaviors it is necessary to give it specific behaviors, and let the coordination emerge. In other hand, with emergent behaviors we introduce another problem, which is the communication between the agents.

What is usually seen in video games is a mixture of both approaches, particularly squad based games, where each NPC has to follow the squad's goal and orders but it has its own decision making to avoid obstacles for example.

Interruptible and **Resumable** behaviors may be an important addition to help in coordination. When we have behaviors which can be performed in collaboration with other agents and when one of them asks another agent to participate in one, it is necessary to see if this agent is already doing something. If it does, we may want to interrupt it and resume it later.

Parallel behaviors should also be taken into account, because in the situation, if it is possible we may want to do both behaviors at the same time. Most of the times, both resuming and parallel behaviors are not done in current games, sometimes because it is not very important but other times because producers think it is not worth it.

Another aspect which can influence how the coordination is made is the **number of participants** wanted in each coordination behavior. When the behaviors just have two participants, they can be made as pairs so that each one is assigned to one of the participants. This does not scale well, thus for more than two participants, behaviors must be flexible, so they can coordinate with any number of characters.

2.3. Summary

In this section an overview over behavior trees and coordination was given, this is important to achieve our goals in the way that we need to understand the structures and concepts involved.

Behavior trees are a relative new addition in the videogame industry and it has already several practical uses in successful video games. It is a structure which is very simple to understand and to work with, has simple concepts and besides having just sequence, selectors, parallel and decorator nodes, behavior trees are very powerful and very complex behaviors can be achieved with them.

Coordination is something that is present in video games for longer time. Centralized approaches are more common, with some emergent coordination when it is really needed. But to give more flexibility it is better to decentralize the coordination. Some situations like interrupting a collaborative behavior or doing more than one thing at the same time are sometimes forgotten, but it can be interesting to see it happening more often.

As behavior trees are being more used in the industry and games more in need of getting better agent coordination and specially the video game industry in need of easier ways to put everything working together, perhaps a credible and valuable coordination can be done using a decentralized approach with help of behavior trees.

3. Related work

The purpose of this work is to improve behavior trees, which implies that an analysis to the current uses and implementations must be made. For that, an analysis will be done to different approaches taken to try to understand how behavior trees are being used, what nodes are being used, the difficulties that are being encountered and how they are being surpassed. It will be taken into special account the coordination issue that each work accomplishes to solve.

State of the art in the videogame industry has been studied, such as Halo 2 AI (9), Spore AI[3], a third-party improvement of Defcon AI (10), Façade AI (11) and Crysis AI (12). An additional work that is not about behavior trees will be analyzed because it has some interesting ideas of an architecture which solve coordination problems and in some way can be seen as an evolution of behavior trees (13).

3.1. Projects

3.1.1. Halo 2 AI – impulses and stimulus

The first reference we encounter from behavior trees being used in games is to Bungie in their game HALO 2. Damian Isla call it Behavior DAG (directed acyclic graph) rather than a Behavior tree because each sub tree can occupy several positions in the structure. In their trees they originally thought to just use selectors in non-leaf nodes for decision making and leafs which are the actions that the agent will perform. To decide on which child we should proceed, they thought on relying in the children's desire-to-run, with each child providing a floating point number to the selector and it selecting the higher one. But they ended up by using a simpler approach, which consists of a binary test and in five different types of deciders.

- **Prioritized-list:** the children are listed in order of priority, and the first that can run, is the one that runs.
- **Sequential:** run every child in order, skipping those that not pass the relevance test.
- **Sequential-loping:** a sequential node that will restart after the last child finishes.
- **Probabilistic:** randomly choses one of the relevant children to run.
- **One-off:** randomly or in a prioritized way picks one child but never repeat that choice.

As we can see, they used two types of sequences and three types of selectors. However this was not enough for what they intended to achieve, because they want the possibility for their priorities to change. To accomplish that, they introduced a **behavior Impulse**. These impulses are put in the tree like a regular behavior, but they are merely a reference to a real behavior. So, when an impulse wins the child competition, the current stack of the tree is redirected to the location of the referenced behavior and continues from that position, or else, the referenced behavior is simply run in the impulse's position. They also used these impulses in another interesting way. If they put an impulse which only returns negative relevancies, obviously will never be picked, but can be useful for debugging, for instance, printing something to the console.

[3] Chris Hecker. My Liner Notes for Spore. http://chrishecker.com/My_Liner_Notes_for_Spore

As we can easily predict, when the trees start to increase in size, they will increase in relevance runtime checking too. And in Halo 2, those relevancy conditions were almost the same across the various candidates checked over and over again, what result in too many unnecessary checks. To deal with this, they introduced **behavior tagging**, which consists in encoding this conditions as a bitvector in a tag for the behavior, representing the necessary conditions to enable that behavior. In runtime this tag is compared with another bitvector representing the actual AI's current state. This way, only the behaviors whose conditions are satisfied will be available, as if the tree originally just has those behaviors, ignoring completely the other ones, acting like a mask. With this mechanism they can do even more, for instance by locking and unlocking behaviors, just by specifying tags to the behaviors and with that specify behaviors for different types of characters, excusing having to do different behavior trees.

Another problem came up when they wanted to deal with event handlers. They could use an impulse, but this implies having to be tested every tick, while ideally this would be event-driven. For this they came up with a **stimulus behavior**. Those stimulus are simply behaviors or impulses which are dynamically added to the behavior tree at a specific position at runtime. This way, when an event happens, a stimulus behavior is added to the tree and for a given period of time, that behavior will be considered for execution along with the originally behaviors.

3.1.2. Spore – improvement of halo's approach

Maxis's developers look at Bungie's work with Halo game series and tried to improve their behavior tree architecture to use in Spore's AI. Unlike Bungie, they enter in more details on how there deciders behave and how the behavior tree ticks work and also explains some differences for their own implementation, but they really do not add anything new to behavior trees structure, as they say, their implementation is a 1.5 version of the Halo 2 implementation.

3.1.3. Defcon – random generation of sub-trees and generic operations

Defcon is multiplayer real-time strategy game and Chong-U Lim, Robin Baumgarten and Simon Colton (10) made use of behavior trees to develop an AI-controlled player that could be able to outperform the game's original AI-bot. They used common behavior tree architecture with **sequences**, **selectors**, **decorators**, **conditions** and **actions**, but they also include some techniques to improve them.

First they added some randomness to the behavior sub-trees without having to put all the possible outcomes in a selector as it would be expected. Instead, what they do is to put a node in a sequence before the actual action that just computes random parameters to be used in the action. Using their example (**Figure 4**), the goal of this sub-tree (left tree in the figure) is to place silos in random locations, so before the action "PlaceStructure" they put an action that calculates the x and y locations to be used in the next action. The "IsDefcon(5)" condition node is there just to ensure that the game state is at Defcon 5. The resultant tree is seen on the right side of the figure, which results in a sequence of all the possible locations. This can be tweaked by restricting the number of silos and the positions.

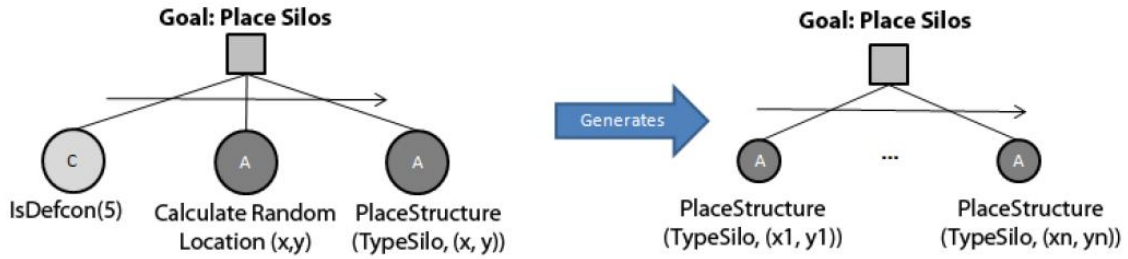

Figure 4 - Random sub-tree generation example

Other interesting thing they did was to use **genetic** operations in the tree, with crossovers on branches and mutation on nodes. This is useful when used with the random generated trees, because this way, throughout the various generations the AI will become more efficient.

3.1.4. Façade – parallel and joint behaviors

Façade[4] is a game which goal is to give to the player an interactive drama between the player and two agents in a 3D environment. The player can interact by entering free text in natural language using the keyboard, walking through the scenario and perform physical actions like picking an item or kissing an agent.

The agent characters maintain their own state and pursue their own goals. They use in Façade ABL (A Behavior Language) (14) and in it they use a form of behavior tree to organize the currently active behaviors which they call ABT. The trees can be composed by **goal nodes** and **behavior nodes**, which can be **sequential** or **parallel**. The goal nodes have only one child which is a behavior node representing the currently selected behavior to reach the goal.

Unlike typical behavior trees, they do not use selectors to choose the behaviors, because this tree is not static. The tree only **represents the active behaviors at each moment**, so it is dynamically changed with the behaviors wanted at each time. When the system chooses what the agent should do, it puts in the tree the corresponding goal or goals, then a behavior to accomplish that goal must be chosen. For that, behaviors have **preconditions**, which must have been fulfilled in order to make possible for this behavior to be chosen, and specificity numbers, which represents how special the behavior is, i.e. the higher the specificity number, more special the behavior is and less usable because it only makes sense in a very specific situation.

They use a tree instead of a stack just because they need **parallel behaviors** and with a tree it is simpler to represent it. Besides the parallel behaviors, what make the ABL language so interesting are the joint goals and behaviors that enables agents to do synchronized behaviors with other agents.

The keyword “joint” can be used in a behavior, making it a joint behavior and because of that, it cannot execute unless all the team members for this **behavior agree** to participate. These members need to be specified in the behavior structure and a request to participate in the behavior is sent to them. If an agent receives a request for a joint behavior, it checks if have a

[4] http://www.interactivestory.net/

matching behavior on its behavior library and if so, it replies affirmatively. If one participant of the joint behavior wants to abort, because a higher priority behavior needs to be executed, this agent has to send an abort request to all participants and only when it receives the confirmation from all of them, it can abort. Instead of aborting the behavior, it can also just be suspended and resumed later after the other high priority behaviors finish executing.

In Façade, because there are only two characters, each one as a joint behavior which is a complementary behavior of the other, thus they can make the coordination more real. In **Figure 5** we can see a pair of joint behaviors in which we can see that they are the same behavior but internally they have different sub goals for each character.

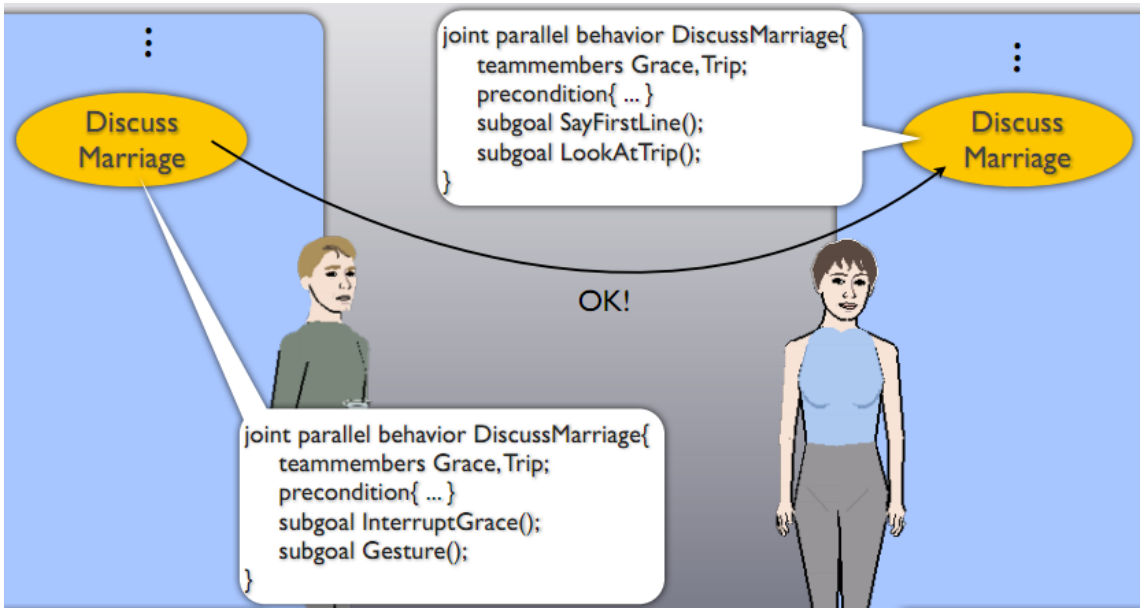

Figure 5 - joint behavior example [5]

The algorithm for executing these behaviors is:

1. The agent chooses the joint behavior for the joint goal wanted.
2. If a behavior is found, the goal is marked as negotiating and the request is broadcasted to all desired participants, otherwise the goal fails.
3. If all team members reply affirmatively, the joint behavior is added to the ABT under the correspondent goal node.
4. At this point, every participant is running the joint behavior and if any of them at any point encounter a condition which causes it to exit or suspend, either because it ended successfully or failed or a higher priority enters the ABT, it has to send an exit request to the other members and mark the joint goal as negotiating.
5. Each member that receives the exit request marks its own joint goal as negotiating and also sends the exit request to the other members. When an agent receives the exit intention from every member it exits.

ABL language supports more than two agents participating in the same joint behavior at the same time, but in Façade it was only tested with two.

[5] Http://studies.rehmsen.de/page_attachments/0000/0008/slides_facade.pdf

3.1.5. Crysis – coordinated behaviors

Crytek used behavior trees in his game Crysis. They used a typical behavior tree for their NPC's AI, but when they started to look at squad/group management they feel the need to create something else in order to simplify this management. So, they created a manager that will be monitoring the behavior tree and also a "tactic", which is a sub tree that they put along with the other nodes of the behavior tree, in a place they consider a good location to identify a good candidate to participate in that tactic. Each tactic sub tree has a name and one or more conditions, for instance the number of participants required.

With these additions, when the behavior tree is running for a certain agent, when it passes in one of this tactics and the conditions that verify that this agent is a good candidate are checked positively, the manager marks this agent as able to participate, but the access to the tactic's branch will be denied. When the manager detects that the minimum number of participants was reached, he opens that branch for the participant agents and reevaluate the behavior tree for those agents.

For instance, we want a coordinative behavior to flank the enemy with two or three NPC. First we need to know where to put the tactic "flank" in the tree. In this example we will consider that a NPC ready to fire a weapon is a good candidate, so we add the node "Flank" under "shoot" and before the original possibilities (**Figure 6**). This way, instead of just throw a grenade or use his weapon against the enemy, this NPC has the option to enter in a flank maneuver with another NPC.

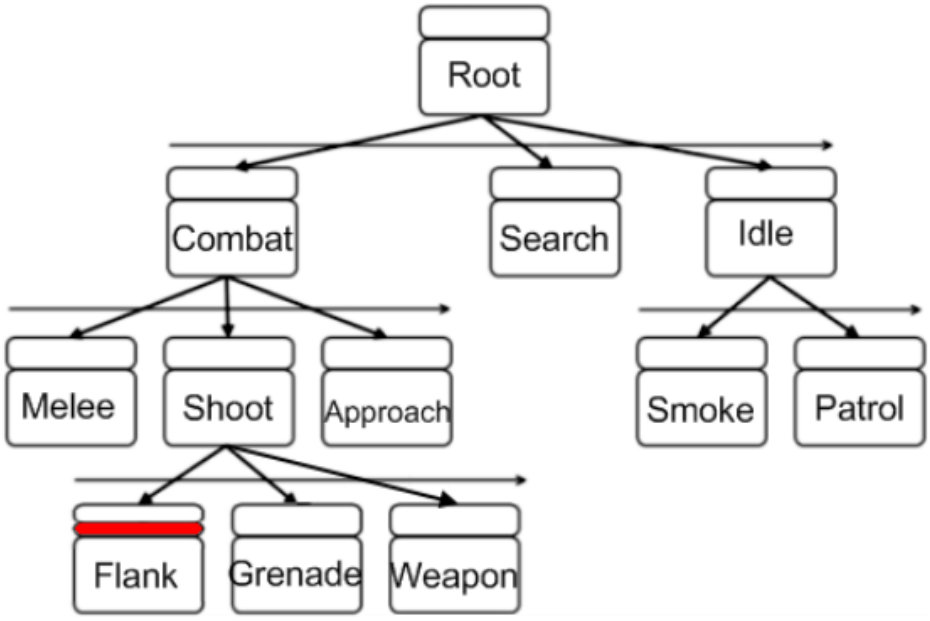

Figure 6 - "Flank" tactic added to the behavior tree. Each non-leaf node represents a selector node

3.1.6. Behavior Multi-Queues

Maria Cutumisu and Duane Szafron (13) made an AI behavior architecture that supports responsive, collaborative, interruptible and resumable behaviors using behavior queues.

According to the authors, there are five important features that a behavior should exhibit, which are:

- **Responsive:** a behavior should react quickly to the environment.
- **Interruptible:** a behavior may be suspended by other behaviors or events.
- **Resumable:** a behavior should be capable to continue execute a previous suspended behavior from where it was interrupted.
- **Collaborative:** a behavior should be able to initiate or respond to joint behaviors requests.
- **Generative:** a behavior should be easy to be created by non-programmers.

None of the most popular AI architectures in games is capable of supporting all these five features, being the most difficult to integrate the collaboration and resuming suspended behaviors. Focused on these subjects, they made an architecture so-called "Behavior Multi-Queues" and implemented in Neverwinter Nights 2 game[6] to demonstrate its viability.

The behavior architecture they developed is based on **three sets of queues**, with each one supporting a different type of behavior, one set for **proactive independent behaviors**, one for **collaborative** and one for **latent behaviors**. A behavior is composed by a sequence of other behaviors or tasks. A task represents an action in the game.

An independent behavior is a behavior that is performed alone, while a collaborative behavior is performed with another NPC. A behavior can as well be proactive, latent, or reactive which represents how the behavior is initiated. **Proactive behaviors** are spontaneously initiated by a NPC when they do not have any active behavior. **Latent behaviors** are initiated if an event cue fires for that NPC. **Reactive behaviors** are initiated when another NPC is trying to perform a collaborative behavior with it. Each behavior must be tagged with these classifications so when they are used they can be put in the correct queue. Collaborative behaviors also have another parameter, the topic in which it fits, so that two NPC can only collaborate if they both have a behavior in the same topic.

The current implementation of the architecture just supports collaborative behaviors for two NPCs, however there is nothing in the architecture making this limitation, so they can easily re-implement it in order to enable collaboration between more than two NPC.

In order to a collaborative behavior starts executing, apart from having to meet all the initiator's behavior conditions, the reactor NPC must satisfy two requirements:

- It must have a **reactive behavior** in the same topic as the initiator's collaborative behavior.
- It must make **"eye-contact" with the initiator**. Eye-contact happens when an NPC is trying to start a collaborative behavior and it succeeds if the reactor NPC is not already in another collaborative behavior or in a latent one.

If all these requirements are met, collaboration begins and both participants start executing their own collaborative behavior, the initiator a proactive or latent behavior and the reactor a

[6] Http://www.nwn2.com/

reactive behavior. A collaborative behavior is divided by phases, with each phase representing a pair of tasks, one for each NPC. This is important for synchronism between the two behaviors, because when a NPC finishes its task, it has to wait for the other NPC to finish the corresponding pair too. The behavior proceeds to the next phase only when both NPC complete its task. It may happen that the proactive and the reactive behavior have different number of tasks. In this situation it is automatically added the appropriate number of default "wait" tasks in the end of the shorter behavior, which simply waits for the corresponding pair task to finish.

3.2. Discussion

Behavior trees are being used in a wide range of situations. They are used in different types of games with different aims and in all situations it is notorious that behavior trees were an added value to those projects. For that reason it is difficult to compare one project to the other. What we can do is analyze each work for what they did regarding the problematic of coordination.

With this in mind and based on characteristics of both behavior trees and coordination seen in the background, we can summarize the different works in some key aspects (**Table 1**).

	Interruptible and Resumable behaviors	**Parallel Behaviors**	**High / Low level Coordination**	**Centralized / Decentralized Coordination**	**Number of participants in coordination**
Halo 2	Only interruption	No	High	Centralized, uses a blackboard	Multi NPCs
Spore	Only interruption	No	N.A.	N.A.	N.A.
Defcon	Only interruption	No	N.A.	N.A.	N.A.
Façade	Both achieved by the ABL language	Enabled by using parallel nodes in the ABT	Both	Decentralized, each NPC communicates with the other	Two NPCs
Crysis	Only interruption	No	High	Centralized, uses an external manager	Multi NPCs
Behavior Multi-Queues	Both enabled by using multi-queues	No	High	Decentralized, each NPC communicates with the other	Two NPCs

Table 1 - Summary of some characteristics of the analized works

Halo 2 and Spore approaches are identical; the difference between them is that in Spore it is not obvious how the coordination is done while in Halo 2 it is known that it is done using a blackboard. In Crysis an external structure to handle the coordination is also used and

although not specified how, it seems to be something similar to a blackboard approach. In any of these games, a behavior which is interrupted cannot be resumed later.

Defcon in other hand does not implement behavior trees for a NPC but for an entire game AI bot, which means that they do not need any coordination, but nevertheless it was interesting to see that behavior trees can also be applied to a bot which simulates a real player in control of a game, not only to NPCs. This project does not support resumable nor parallel behaviors, but in the context of the game it was not that much necessary.

Façade is an interesting case in the videogame AI history because it managed to coordinate two NPCs with each other and also with the player in a very credible way. However it was not tested in a large scale, it does something that none of the others does, which is parallel behaviors, i.e. each NPC can do more than one thing at the same time. The coordination that exists in the game is achieved because there are only two NPCs and each joint behavior has two versions, with each one being assigned to one NPC, thus limiting to those two NPCs which is not scalable. Another problem of this project is the complexity of the ABL, language they used for the NPC's AI, which only a small part uses behavior trees.

Behavior Multi-Queues perhaps is the more complete of the analized works, because it not only managed the coordination problem, but also can resume early interrupted behaviors, however it does not handle parallel behaviors and it is not based in behavior trees. The architecture was tested with a large set of NPCs in a specific scenario in NeverWinter Nights and with that they could use the internal language to help with their behavior implementations and with low level coordination.

4. StarCraft and connection with BWAPI framework

Now that we have defined our goals, have seen a background on behavior trees and coordination and have looked to how those things are being used in the videogame industry and in other related projects, it is time to choose a platform to sustain our project. Since we want to show coordination behavior in NPCs in a videogame scenario, we first thought using a RPG game, specifically the Neverwinter Nights 2. But then we decided to go in another direction and picked a different game genre, in this case the famous RTS, StarCraft. We choose this game not only because it has vast and distinguished types of units for us to work with, but also because it is not usual the AI in this genre of games to be decentralized and based on each unit individually and so, we wanted to try it and see the results.

4.1. StarCraft

StarCraft[7] is a real time strategy game from Bllizzard, which besides being released in the year of 1998, is still played by many. It is one of the best games of its genre being one of its strong points the fact that it is very well balanced. Fact that helped it becomes a professional sport, mainly in South Korea. These reasons were sufficient to make us choose this game for us to experiment our behavior tree implementation.

StarCraft is a vast game. It has three complete different races to play, the Terran, the Protoss and the Zerg. In our implementation we only used a limited set of StarCraft units, because it would be impractical in a short time to tweak the AI for all of them. Nonetheless we chose units that can still demonstrate the dynamics and behaviors we wish to demonstrate. Our race of choice was the Protoss, for the simple reason that they have two types of heath point, the shields, which can regenerate with time and hit points which not. This is crucial to our work since we want the units to be aware of their health status and with this kind of heath that regenerates over time, units can choose to attack or retreat accordingly. Within the Protoss race, we restricted our units to five types only, including one building, a worker and 3 military types:

- **Nexus:** The main building of the Protoss race. The roles of this building are to produce probes (Protoss workers) and receive resources brought by those probes. These buildings have 750 shield points and 750 hit points.
  The Nexus building will be used as a representation of the player base with more importance in the “spawn 2” scenario (see section 6.1.1.11) where the match only finishes when the units destroy it. The “complete game test” scenario is the only one where the Nexus is used at its full potential, as this scenario represents a full game scenario.
- **Probe:** The worker unit type of the Protoss race. Their purpose is mainly to gather resources and construct buildings, however they still can be used to attack, although they are very weak for that. They have 20 shield points and 20 hit points and its attack power is of 5 attack points.

[7] http://eu.blizzard.com/en-gb/games/sc/

In our tests, Probes are just used in the “complete game test” scenario and are only used to gather resources and construct buildings, i.e. they are never to attack anemy units.

- **Zealot:** They are the main Protoss force. These units have 100 hit points and 60 shield points. They only attack ground units and when in contact with the target, which does a double strike with 8 attack points each.
  Zealots are the main unit we use in our scenarios, just because it is the principal unit of the Protoss race. In our implementation it is considered as a military unit.
- **High Templar:** A weak, but very powerful unit. They have only 40 hit points and 40 shield points and cannot attack in a regular way. However they are spell casters, i.e. units which have special abilities that consume energy to use them. They have 200 energy points, which regenerate over time, that can be spent in the “psionic storm” ability. This is an attack that covers an area for a short period of time in which any unit that is caught under it will continuously suffer damage over the time it stays under the storm. This ability cost 75 energy points. High Templars also have the ability to merge with another High Templar into one more powerful unit.
  High Templars are very important on our experiments because the way they attack is completely different from zealots, besides the possible targets that each can attack be also different. We this in mind we were able to able to show cooperation between these different types of units and how they can work together in order to accomplish their objectives faster and more efficiently.
- **Archon:** When two High Templars merge, one Archon appears in their places. This unit, unlike High Templars, does not have any special ability, but have a very strong attack and great shield points. These units have 10 hit points and 350 shield points. They attack at a short range with a power of 30 attack points and do splash attack, i.e. it attacks the enemies around the target with fraction of the original attack.
  Archons were chosen because they are the unit in which the high Templars transform when they merge. They are important because we can show how the units try to adapt to the different scenario circumstances. In our implementation it is considered as a military unit.

Figure 7 – In game representation of Protoss units: Nexus, Probe, Zealot, High Templar and Archon (left to right)

For the enemies, depending on the scenario, we choose Protoss or Terran race. For the scenarios were we wanted to have the same units in each side we used the Protoss, whereas for other scenarios we choose the Terran race. This choice was basically because we wanted to

use a very particular unit present in this race which is the “Bunker”. Including this unit we used five unit types from Terran race:

- **Bunker:** This unit is not really a unit, is a building which makes it a non-walking unit. A characteristic of Terran buildings is that when they have only one third of its hit points left, they start to lose hit points over time, even when not being under attack. These building have 350 hit points and have the ability to store up to four infantry units. Alone these buildings are not a threat, but when they have units inside those units can attack enemies in their weapon range without getting hurt.
  The bunker will be very important on our scenarios because will show a different attack approach from our units, mostly when we use high Templars as they cannot attack buildings. The bunkers will always be used with four Firebats inside.
- **Firebat:** It is one of the Terran infantry units. They have 50 hit points and their attack consists in a double strike doing 8 attack points each. This attack is a short range attack and do splash damage to nearby enemy units and only affects ground units.
  Firebats were chosen as the main enemy unit mostly because their use in the bunkers. We choose Firebats over Marines because we wanted to have short range units so our units could prepare the attack near the bunker without being attacked.
- **Marine:** The most basic unit of the Terran force and also an infantry. They have 40 hit points and their weapon permits a middle range of 6 attack points affecting ground and air units.
  Marines were introduced in some scenarios because we wanted some diversity on attack range and life points.
- **Goliath:** A non-infantry unit with 125 hit points. They have two different weapons one for air units and the other to ground units. The ground weapon does 12 attack points and has middle range.
  Goliaths were only used in some defensive scenarios just to have a more powerful unit on the enemy side.
- **Missile Turret:** Another Terran building used to detect enemy units approaching and attack air units. They have 200 hit points.
  Missile turrets were introduced to show the priorities of our units when choosing their targets.

Figure 8 - In game representation of Terran units: Bunker, Firebat, Marine, Goliath and Missile Turret (left to right)

In the game there are several orders that can be issued to a unit or a group of selected units. However in our implementation we use only four types of orders:

- **Move:** Order a unit to move to specific position.

- **Attack:** Order a unit to attack a specific unit.
- **Use Tech:** Used by units with special abilities like High Templars to use one of its abilities. In our case it is used only by the high Templars to use psionic storm and merge with another high Templar.
- **Stop:** Whatever the unit is doing it stops doing it.

These orders are directly mapped to the actions that each unit's AI agent has.

Besides StarCraft being a very balanced game, there is a lot of randomness in battle. Namely the cool down between consecutive attacks, which can vary between minus one or plus 2 frames and the chance to the attack hit the target, which depends on the ground level in which the units are. For instance, on the same ground the chance of an attack misses the target is 1 on 256. Those factors make that in the same situations the outcome can vary a little bit from one time to another.

4.2. Original StarCraft AI

StarCraft AI is originally made to play in a complete game scenario, where it has to manage not only the attack, but everything inherent to an RTS game, like buildings, upgrades, units and resources. This is not the case of our scenarios, as our goals are only focused on unit attack and survival management. But as we which to compare our implementation with the original, we had to solve this little problem.

Originally StarCraft have different types of AI[8] prepared to be used in the scenarios. There are custom AIs and campaign AIs. Campaign AIs are used for the campaign scenarios, where the objective is not to overwhelm the player but in fact, to let the player win easily. For this type of AI there are different levels of difficulty, going from easy to insane. Custom AIs are more like what we expect as an AI opponent for a head-to-head scenario, in which it tries to really win the game.

For our interests, the Custom AIs are not suitable because they expect to have a starting base, workers, i.e. units that construct new buildings and gather resources, and resources to collect, so we decided to use the Insane Campaign AI. We choose the insane one, because unlike the others campaign AIs, it really wants to win the game and it gives all it got, expecting nothing from the scenario.

For the most scenarios the behaviors of this AI is not enough. In order to make a right and useful comparison the AI needs to start an attack against the opposite player, which it does not since it stays still waiting for the player to attack. Fortunately there is a special script that we can use to complement the AI, which orders all units to make a suicide attack, i.e. it chooses an enemy unit and instructs them to attack it until all the opposite units are dead.

[8] http://classic.battle.net/scc/faq/aiscripts.shtml

4.3. BWAPI framework

Since StarCraft is a commercial and closed game it is not open for anybody who wants to make AI experimentations on it. Fortunately there is an open-source framework which enables anybody to experiment their AI implementations. BWAPI[9] is a free C++ framework with which we can retrieve information about the game and issue commands like a human player would do in the game. In addition to BWAPI we also used bwapi-mono-bridge[10], a wrapper for everybody who wants to use BWAPI, but prefers to program in C# instead of C++. In order to use the mono-bridge, it is also necessary to install the mono .Net framework [11] version 2.8.

To make the connection between BWAPI and StarCraft it is also needed a program that injects that code in the game. The program used is called "Chaos Launcher". Basically this program injects a library (.dll) which contains the BWAPI and the AI implementation into the game, assuming the place of the human player when the game starts.

So that we can use our AI in the game, we must have a **StarCraftBot** class that implements the BWAPI framework interface called **IStarcraftBot**. Is through this class that StarCraft can interact with our code, by calling events when something specific happens in the game. The events we will use are:

- **onEnd:** Called when the game match comes to an end. It receives as an argument, the information if the player won or lost the match. It is used in our implementation just for debug information.
- **onFrame:** Called every logic frame of the match. It is the heartbeat of our AI, where every frame the "execute" method of each agent is called in order to give an order to its unit.
- **onStart:** Called at the beginning of the match. It is used to initialize everything needed for the rest of the match, namely the creation of the list of agents.
- **onUnitDestroy:** Called when a unit is withdrawn from the match. It receives as an argument, the reference to the unit in question. If the unit belongs to the player, it is used to remove the correspondent agent form the list of agents.
- **onUnitMorph:** Called when a unit morphs into another type. It receives as an argument, the reference to the unit in question. If the unit is an Archon it means that two High Templars merged and therefore one was destroyed and the other was morphed. If that is the case, the correspondent agent, which was a High Templar agent, is replaced by an Archon agent that will control the same unit.
- **onUnitShow:** Called when a unit appears. It receives as an argument, the reference to the unit in question. If the unit belongs to the player, then an agent of the correspondent type is created and added to the list of agents. If the unit is neither a Zealot nor a High Templar, the agent created is a general agent.

There are other events in the class, but only these six were necessary. BWAPI also provides a "bwapi" object, which is used to retrieve all the information from the game. This information

[9] http://code.google.com/p/bwapi/
[10] http://code.google.com/p/bwapi-mono-bridge/
[11] www.go-mono.com

can be in game information about players, units or map. With that object we can also retrieve references to the units in our control and through them give the orders we want to give.

Using BWAPI was not an easy task. Documentation is not very clear, which made us to do mistakes that we were not expecting and that were difficult to understand their origins. For instance, a method that should give us the information if a certain unit is under attack, only works correctly if the unit is in our control, otherwise it does not give the expected information. To give another example, it would be logical that when a unit dies it would have zero hit points, but in reality, in the frame in which the unit dies it still has some hit points left, it is only updated to zero a few frames later. But many more pitfalls we had to overcome to achieve our goals.

5. Architecture

In order to test and evaluate our behavior trees, we decided to implement it in the StarCraft videogame. This section will explain in practice how that link was made and what else was necessary to be implemented to support the behavior trees.

From this moment on, every time we want to reference an in game unit, we will use the designation **unit**. When we want to reference the Unit class that is controlling the in game unit, we will use the designation **agent**.

5.1. Solution Architecture

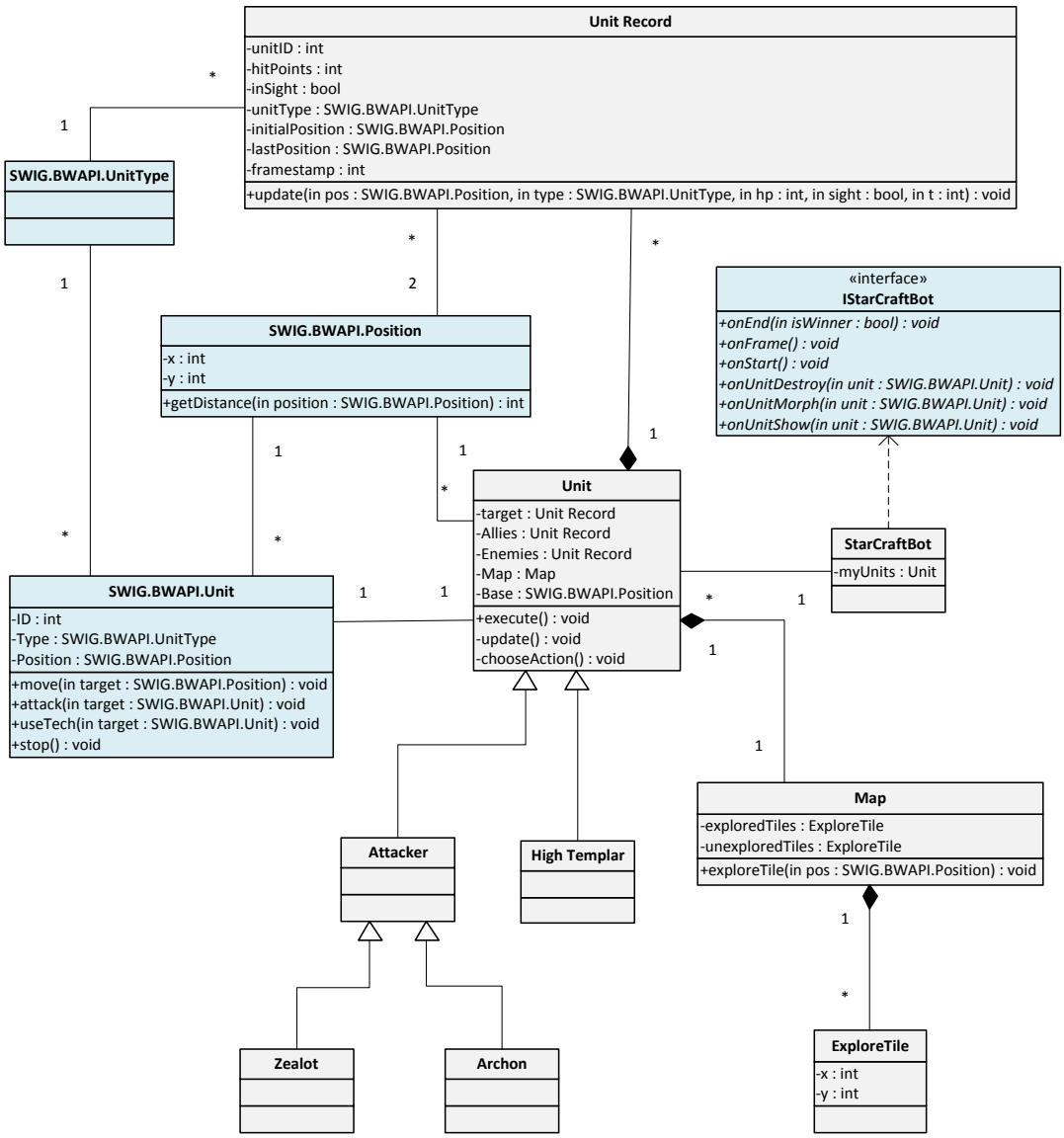

Figure 9 - UML of the entire solution Architecture (blue classes represent BWAPI classes, gray classes represent created classes)

In **Figure 9** we can see all the classes implemented and there connection with some classes provided by the BWAPI framework. The entire BWAPI architecture can be seen in the BWAPI web site[12].

The StarCraftBot class is the representation of the player in the game and therefore is essential to make the connection with the game. Basically it implements the IStarCraftBot interface provided by BWAPI and it is through the onFrame event that it calls the **execute** method on every unit present in **myUnits** list.

5.1.1. Unit class implementation

This is the class that represents each agent controlling each ally unit in the game that belongs to the player. Each of which has several data stored that is updated every logical frame in order to help it make its choices:

- **Target:** The target unit that the agent is currently attacking or preparing to attack;
- **Allies:** A list of all the ally units that this agent knows about. It is used so the agent can know with how many allies he can count with and there types;
- **Enemies:** A list of all enemy units this agent knows about. It is used to know where the enemies are located, there types and if are almost dying or not;
- **Map:** An object containing lists of known explored and unexplored Explore Tiles. It is used to avoid exploring already explored locations;
- **Base:** The position of its base. It is used when the unit wants to retreat from battle or wait until it is recovered.

Besides these properties, the Unit class also has important methods, two of which being directly called by the execute method:

- **chooseAction method:** This method is where the behavior tree is executed and therefore where an action for the unit perform in the game is chosen. The behavior tree is detailed in section 5.2;
- **update method:** before the action choice, in each frame the agent updates its internal variables and state and this is where it is done. The update phase consists in:
    - Update internal map knowledge by moving the Explore Tile corresponding to the unit position and the eight tiles around it, from the unexplored tiles list to the explored tiles list;
    - Update or Add information about units in sight range. If the unit is an ally and in previous frame was not visible, then it sends a message with its knowledge to it;
    - Update as invisible or dead the units that were visible in the last frame but not in this one. If the unit position changes radically it means that it was removed from play, i.e. the unit is dead;
    - Considers dead all ally units which are not seen for more than a certain period of time.

[12] http://i.imgur.com/lppnE.png

The messages sent in the update process contain a list with the known unexplored map tiles and the list of updated allies and enemies. When the unit receives this message, it moves all Explore Tiles present in the list received, from its unexplored tile list to the explored tile list. For the enemy and ally lists, it checks the frame stamp of each unit record present in it and if the frame stamp of the correspondent unit record of its own is smaller, than it updates its information with the new one because it is more recent. If the agent does not have any unit record for that particular unit, it does not need to check the frame stamp, just creates a new record and copy the information. Also, if the sender is a military unit and it previously decided that it would be necessary to ask a High Templar to merge into an Archon and the receiver happens to be a High Templar, then it does it.

All the other methods are auxiliary methods that support the decision making from the behavior tree, which are further explained in section 5.2.2.

The Unit class has two subclasses associated, the HighTemplar and the Attacker, which in turn also have two subclasses, the Archon and the Zealot. The differences between each one of them are the implementation of the virtual methods present in the original class and each one is used according to the unit type.

5.1.2. Unit Record class implementation

As explained in the previous section each Unit object has reference for other units, but those references are not for the actual units but instead, for a small representation of them that we created and are named **Unit Record**. In each Unit Record we have the following information:

- **Unit ID:** The game ID of the unit. It is used to identify the real unit in which this object is representing;
- **Hit Points:** The hit points plus shield points of the unit at the last update. It is used to help choose the unit that this unit will target and to calculate if the attack can be done. This is further explained in section 5.2.2. This also tells if the unit is dead or alive, if the unit has 0 hit points, than it is dead;
- **In Sight:** Tells if the unit is visible, i.e. is in sight range, at the moment;
- **Unit Type:** The type of the unit. It is used to know if it is a dangerous unit or for instance if can be attacked;
- **Initial Position:** The position where it was first seen. It is used to search the enemy if it was not in the last position recorded;
- **Last Position:** The last position where it was seen. It is used to know the position where it was last seen, if the unit intends to attack it;
- **Frame stamp:** The number of the last frame when all the previous information was updated. It used to compare the information with new values and therefore update them or not.

Every time the execute method is called or something else requires so, the corresponding agent has to update its list of ally and enemy units. This means that the unit records must be updated. For instance, the agent do not know any unit, but in this frame an ally unit appears in the unit's sight range, so in the update phase, the agent will notice that and since it is a new unit, it will create a new unit record that represents that unit and add it to Allie list. In the next

frame the same ally unit moves to a new position, therefore the agent in its update phase, since it already has a unit record for that unit, will update it with the new position. In order to do this, the unit record class has a method **update**. This method receives all the new values to update and also the frame stamp in which those values are verified. Then, if the frame stamp received is higher than the frame stamp previously registered, the update can be done, but if it is lower, the values stay unchanged because this means that the new information is in reality older than the values it already has. This can happen when the agent receives a message from other agent containing information about units that it saw before, but the unit that is receiving the message have seen those units more recently than the sender. However, when a unit enters in sight range of other unit, the other unit also enters and then they both send messages to each other and therefore both end with the same and more recent information.

This class is very important because it represents the memory of the agent. Every time the agent wants to know information about its enemies or allies is to this object that it will gather that information.

5.1.3. Map class implementation

The Map object contains at the start of the game an empty list of explored tiles and an unexplored tiles list which contains all the possible Explore Tiles that can be explored by the unit. An Explore tile is equal to 128x128 in game pixels. This list is filled when its agent is initialized, where the map size is divided and is created one Explore Tile for each section. The borders of the map and tiles where the unit is not able to walk on, are not included into the list. The ExploreTile class just has the X and Y coordinates, which represents the map location of that tile.

This class has an important method, which is the **exploreTile**. This method is used whenever the unit enters in a new explore tile. Then if it is the case that tile is still in the unexplored list, It is removed and added to the explore list, if not it is done nothing. It also does the same for the eight adjacent tiles that are around the unit. It is important to have a list with the explored tiles so the agents can send that information to the others in order for them to remove those tiles from there unexplored tiles.

5.2. Unit Behavior Trees

At least as far as we know, every implementation previously tested on coordination between agents in video game scenarios was based on some centralized entity, mostly blackboards where all the agents store their knowledge and load new information based on the knowledge of the others. Examples of that are the Marc Cavazza (15) experiments in The Dropship Game or the Choreographic Coordination scenario by Lima (16). Another common solution is to use a super-agent which commands all the other agents in the world or even have some hierarchy between the units, which is somewhat the case with what Tan and Cheng (17) did. On their experiments, they came out with a framework capable of having agents cooperating with each other in a decentralized by, but for the strategic attacks they have a hierarchy of agents that

give orders to the agents above. Nevertheless, it is the project that most resembles to ours in terms of decentralized agent coordination.

As it is one of our goals to achieve a solution without any kind of external entity, our implementation consists only in a large behavior tree made of smaller trees and nothing more. Therefore, all the information that each agent have is only known by it, until it is able to share it with the others. This communication between the agents occurs only when the correspondent units are in sight range.

All the decisions the agents take are based only on information received from others and on their own, not receiving direct orders from any other agent.

5.2.1. Conceptual Model

In each behavior tree diagram, a rectangle leaf represents a sub tree, an in game action or a composition of taking a decision and then making the actual in game action. The circle leafs represent a condition, which some are self-explained whilst the others are explained in section 5.2.2.

5.2.1.1. Choose Action

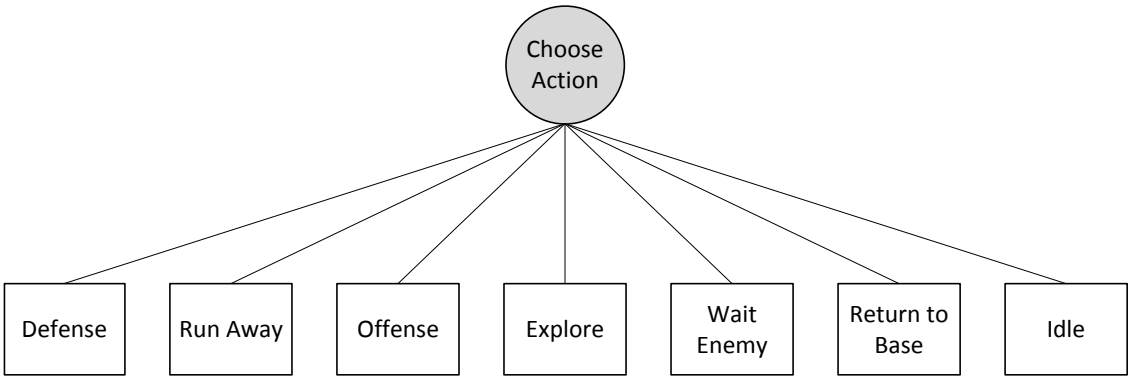

Figure 10 - "Choose Action" Behavior tree

The tree in **Figure 10** is the overall behavior tree of each agent. It is a selector with seven other sub trees each of which representing seven different types of behaviors that each unit can perform in the game. The priority for each child to run is from left to right and since this is a selector node, only one of them will run successfully.

Before it starts, it is first needed to find the enemy unit that should be attacked. In our implementation the choice of the enemy unit to attack is based on the type of the units, the hit points they have and their distance to the unit. More precisely, if there are dangerous enemies in sight range, the agent chooses as a possible target a unit that is targetable and is the closest of them. If there are no enemy units in sight, the agent chooses the targetable unit that has the less hit points of all the known alive enemies. In case the agent is controlling an Attacker type unit, it first sees if it knows any Terran bunker and if so, the target to attack would be the bunker with less hit points.

5.2.1.2. Defense

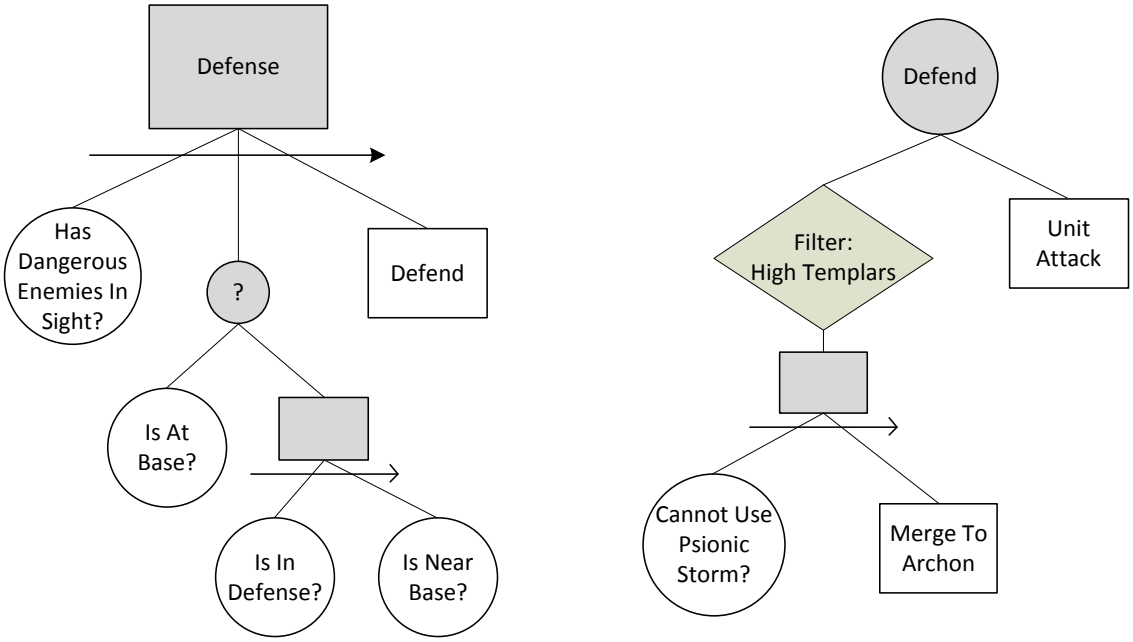

Figure 11 - "Defense" Behavior tree and "Defend" sub tree

On the left tree in **Figure 11** we have the most priority behavior for our agents. Basically this tree succeeds when the unit is near or at the base and has dangerous enemies in sight range. The unit is near the base when the distance between the unit and the base is two times the sight range at maximum and the defense tree succeeds only if the unit was previously defending.This behavior has a high priority then the “Run Away” behavior because in the way it is implemented, the unit runs away towards the base and therefore a higher priority behavior is necessary for when the unit is already at the base. Otherwise the unit would stay still and be killed after a while.

On the right side of **Figure 11** is represented the Defend behavior, which basically decide if the unit will attack the attacking enemies or in case the unit be a High Templar, will merge with another High Templar to create an Archon. If the unit is a High Templar but it still has sufficient energy to cast the psionic storm ability, it will not merge, but will attack the enemy units instead.

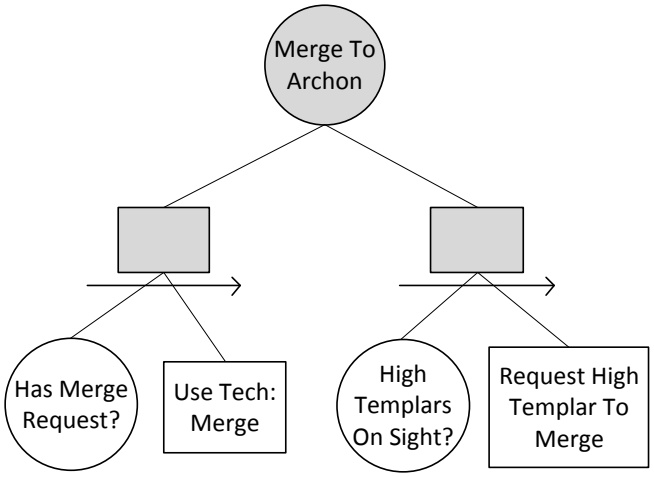

Figure 12 - "Merge To Archon" Behavior tree

The High Templar merge behavior is represented in **Figure 12**. The unit will merge with another, only if the other High Templar accepts it. For that sake, each agent controlling a High Templar has an extra property that tells if the unit has previously received a request or not. So, if the unit is a High Templar and wants to merge, it either already has a request from another High Templar and therefore can use the action “Use Tech” to merge with it, or see if there are no other High Templar in sight range to make the request.

In this case, the “Request High Templar To Merge” action is not an in game action, but only a communication between the agents representing the units, which happens outside the game environment.

5.2.1.3. Run Away

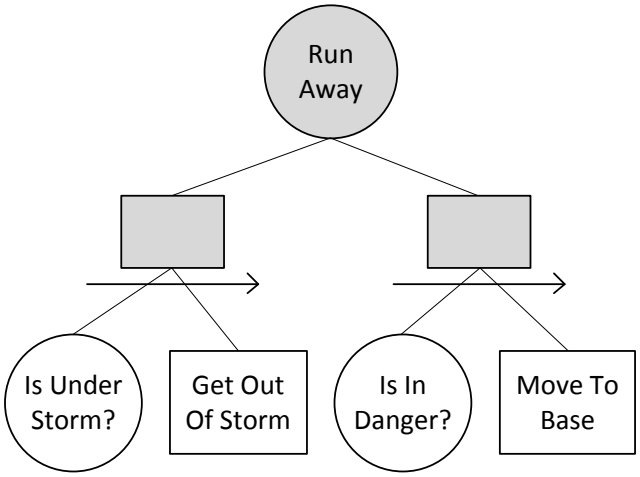

Figure 13 - "Run Away" Behavior tree

The tree in **Figure 13** represents the Run Away behavior, which is the second most priority after the Defense behavior. Essentially a unit must run away from battle when it is in danger. In this case we differentiate being in danger and being under storm. So, if the unit is under a psionic storm, it must get out of the storm by going to nearest position that is not affected by the storm. So the “Get Out of Storm” action consists in choosing the best position to run away

and then move towards it. If it is in danger due to another factor then it must run away towards the base.

The decision to have a distinct way to run away when the unit is under a psionic storm then when the unit is in danger, was taken because when a unit is under a storm it should evade it the soonest possible. In order to make it happen, the nearest position outside of the storm is calculated and the unit goes to it, instead of going directly to the base.

5.2.1.4. Offense

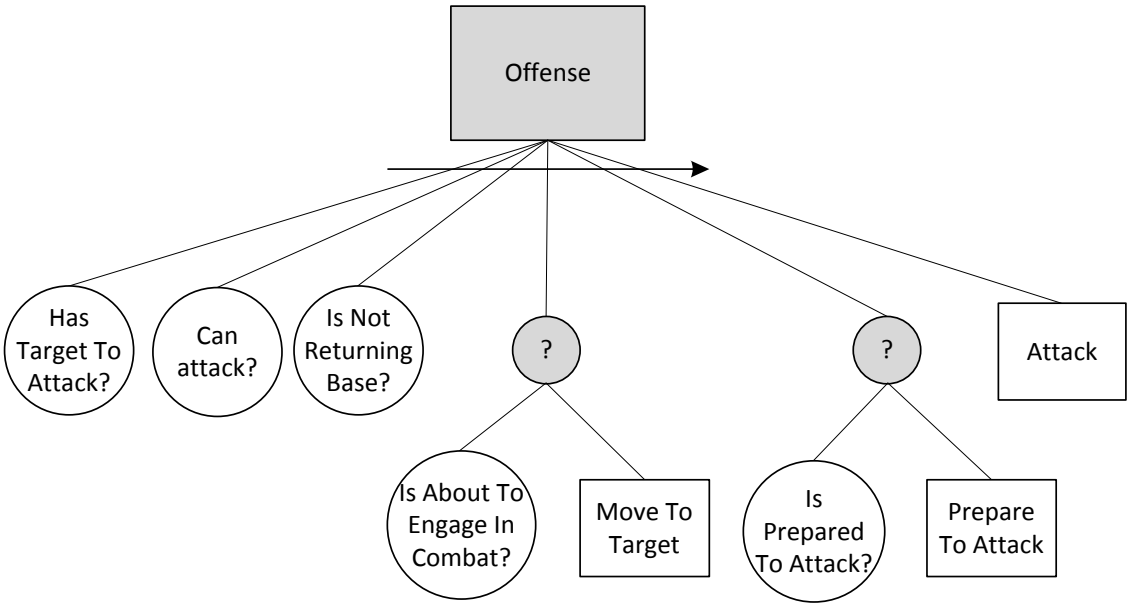

Figure 14 - "Offense" Behavior tree

If the unit is not in danger and it do not need to defend at the base, then the unit must try to take the offense. The tree in **Figure 14** shows how that behavior works. Before anything else it is necessary to check three things. If the agent knows any enemy unit that can be attacked by it, if the unit can attack and if it is not returning back to base. After that, the agent enters in a sequence of tasks. First, if the unit is not close to the enemy unit, it has to move towards its position. Then when the unit is about to engage in combat, i.e. is already near the enemy unit, it has to check if the unit is prepared to attack, i.e. if it thinks it has conditions to kill the enemy unit without being killed. If not, the unit must do whatever is necessary to be prepared. Otherwise it is now time for the unit to attack.

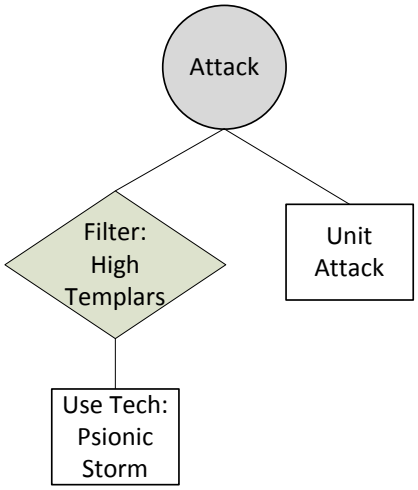

Figure 15 - "Attack" Behavior tree

When it is finally time to attack, the agent enters in the attack behavior, which is seen in **Figure 15**. This tree is there basically to filter the unit type, because if the unit is a High Templar it has to attack by using the action “Use Tech” to cast the psionic storm on the enemy unit, but if not, the unit can just use the action “Attack”.

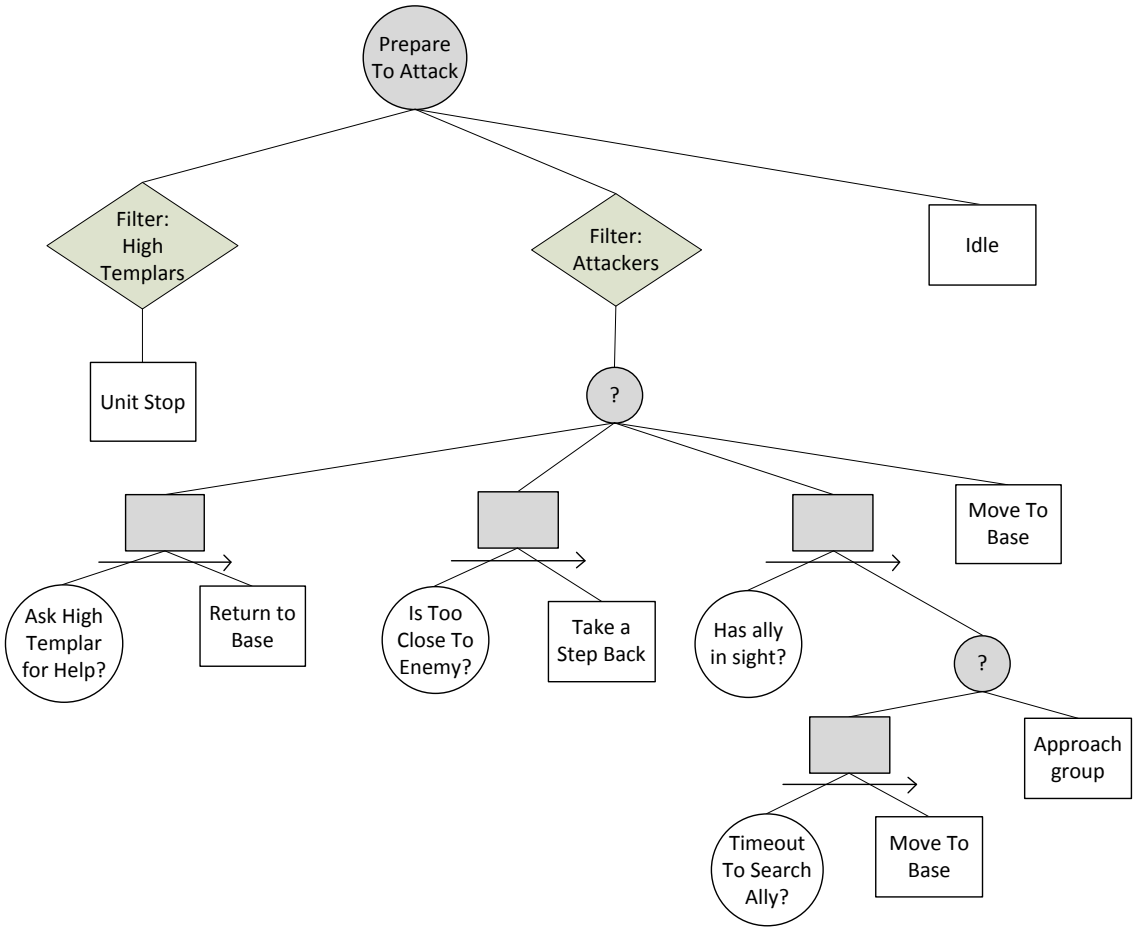

Figure 16 - "Prepare To Attack" Behavior tree

On the other hand, if the unit is not yet prepared to attack, it must prepare itself. In **Figure 16** we have the tree that represents that behavior. If the unit is a High Templar then it just needs to wait until it is prepared, because it is just a matter of time. Although if the unit is an Attacker, i.e. a Zealot or an Archon, it must prepare according the situation.

- If the unit realizes that it needs the help of the High Templars, it returns to the base hoping to find a High Templar at it or in the way. If it happens to meet one, then it makes the request for it to transform into an Archon.
- If the unit is excessively close to the enemy unit but should not attack it, which is the case, then it must step back, i.e. move away a little bit by moving the unit to a close but further position from the enemy.
- If the unit has allies in sight range it can either approach them in order to form a bigger group, or if a timeout expires must move towards the base, searching for more ally units.
- If none of the three situations above is the actual situation, then the unit must move to the base.

The “approach group” action consists in calculate the centroid of the units around it and move towards it.

Finally, if the unit type is neither a High Templar nor an Attacker, it should do nothing, but it is opened for possible new implementations of other types.

5.2.1.5. Explore

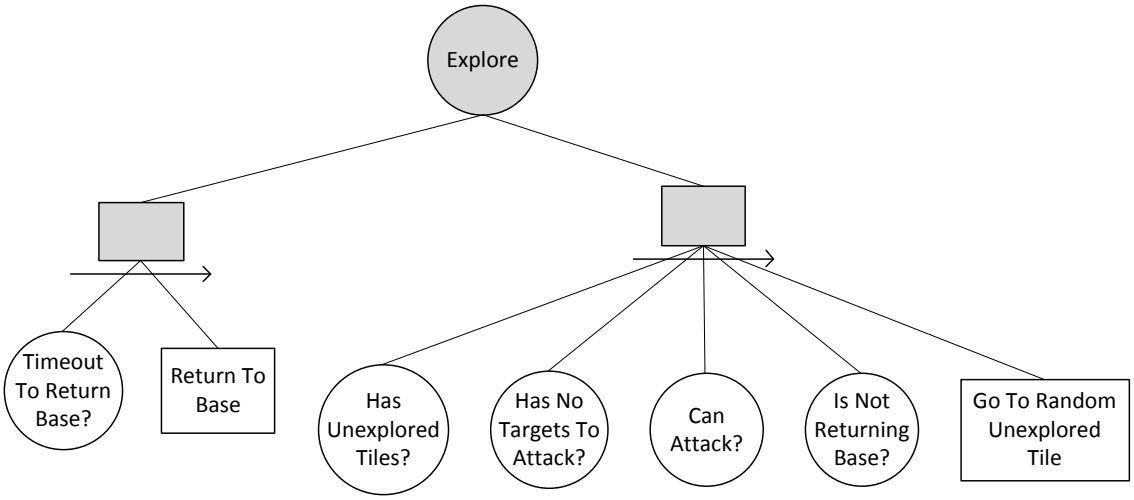

Figure 17 - "Explore" Behavior tree

In **Figure 17** we have the representation of the explore behavior. When the agent does not know any enemy unit alive that cannot be attacked by the unit, it cannot attack anything. Consequently it needs to find one and the way to do such thing is by exploring the map. More precisely, if there are unexplored tiles left to explore, if do not have targets to attack but could do so if there were and it is not returning to base, then the unit goes towards one of the unexplored tiles chosen randomly. However the units should not keep exploring forever, so a timeout is used and when it runs out, the unit must return to base to share its knowledge with

the others. This timeout is reset every time the unit passes sufficiently near the base and it enters in its sight range.

When an agent has the unexplored list empty, it means the map was fully explored and therefore the unit will stay at the base or enemy beacon, if exists, forever.

5.2.1.6. Wait Enemy

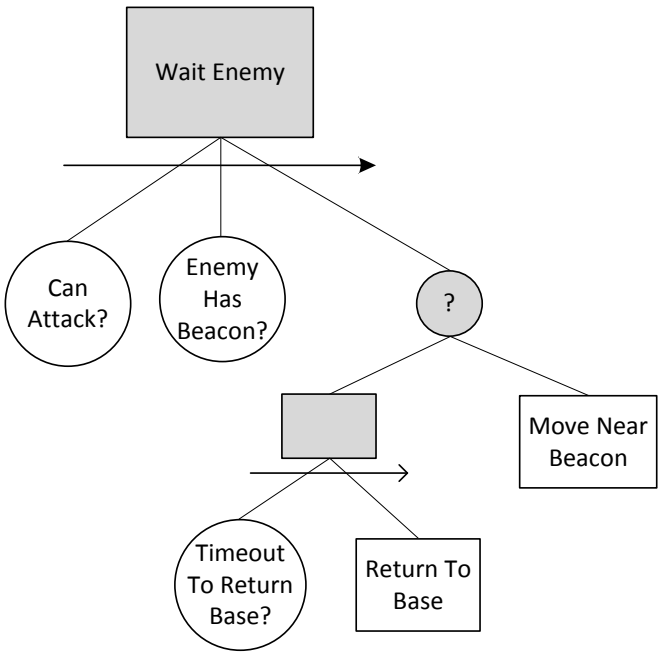

Figure 18 - "Wait Enemy" Behavior tree

After the unit had explored the entire map but still do not have enemies to attack, it has two possible behaviors. One of each is the wait for the enemy to appear represented in **Figure 18**. This behavior can only be done if the unit can attack and the enemy player has a beacon, i.e. if the unit found an enemy beacon in its explorations. Basically this behavior tells the unit to stay close to the enemy beacon waiting for an enemy unit to appear. However we do not want the unit to stay there forever, so the return to base timeout is also used here and therefore if it runs out, the unit must return to base.

5.2.1.7. Return To Base

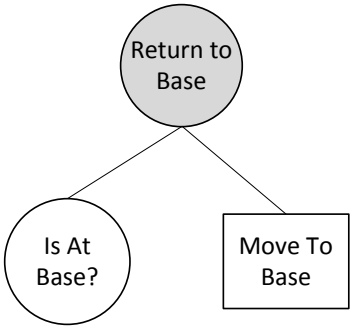

Figure 19 - "Return To Base" Behavior tree

The other possible behavior that each the agents have when the map is completely explored, is wait at the base. This is a very simple behavior where if the unit is not yet at the base, it must move to its position and is represented In **Figure 19**. As seen in the previous trees, this behavior is also called from other places, namely where there is a reference to the timeout that makes the unit return to base.

5.2.1.8. Idle

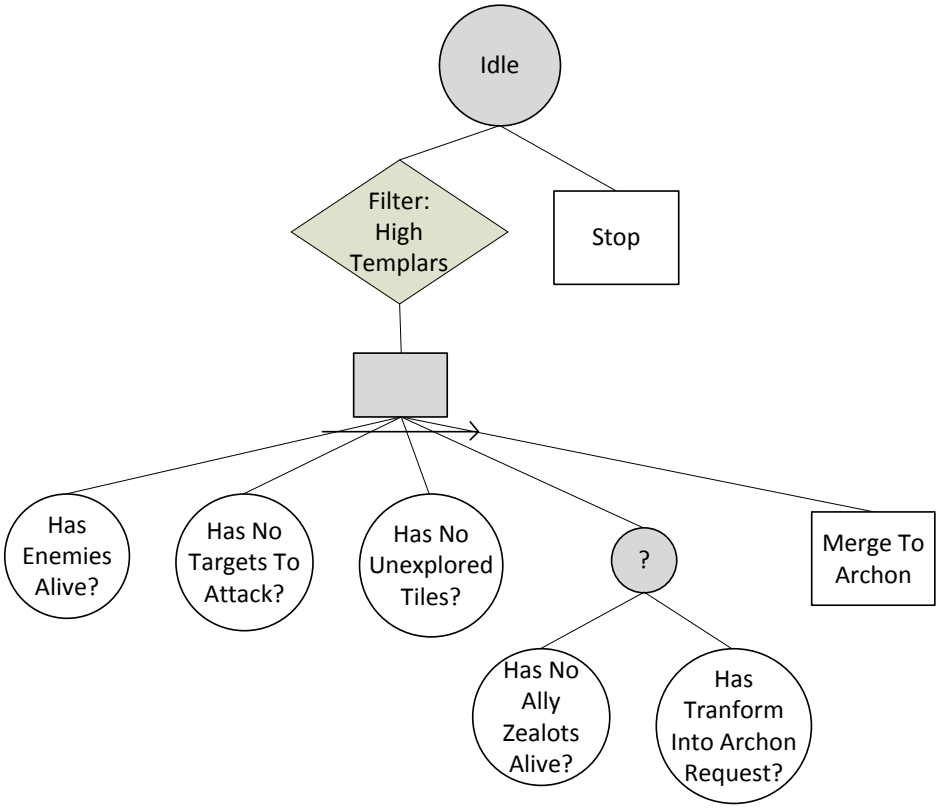

Figure 20 - "Idle" Behavior tree

Initially, the idle behavior used to be an empty tree and was there just for the case when no other behavior would successfully run. But because of the High Templar ability to merge, we had to enable that when there are only High Templars alive and there are units that cannot be killed by them, e.g. enemy buildings. So, in the idle behavior, which is represented in **Figure 20**, the High Templars will check if they know any enemies alive, if all of them cannot be attacked, if the entire map is explored and either if there are no more ally zealots alive or if it has previously received a merge request. If all of those conditions are true, then it does the "Merge To Archon" behavior, explained in section 0 In case the unit does not be a High Templar or some of the conditions do not verify, the unit merely use the action "Stop", waiting for something to happen.

5.2.2. Implementation details

In this section the details about the conditions we have in each behavior tree diagram are explained. Some of the conditions are self-explained and do not need to be explained, however others are more complicated. These conditions are in practice methods belonging to the Unit class that returns true or false according to the situation. Each condition method uses

auxiliary functions, each of which gathers the necessary information needed to the condition method decide and return true or false. There are functions to gather the following information:

- **Dangerous Units:** a unit is considered dangerous if it belongs to an enemy player and one of the following criteria is true:
    - it is a Terran Bunker;
    - it has a ground weapon;
    - it is a spell caster.
- **Unit in Danger:** the unit is considered in danger if one of the following is true:
    - the unit is not attacking but is being attacked;
    - the unit is not attacking, cannot attack and have dangerous enemies in sight;
    - the unit is attacking but its shields are lower than 10 points;
    - the unit is under a psionic storm. In practice is considered Under Storm;
    - the unit is an Archon and its shields are lower than 75 points.
- **Targetable Units:** a unit is considered targetable if the unit that wants to target it can inflict damage to it with a regular attack or special ability. A unit of a military type can attack a unit if it is not invincible (e.g. a Beacon is invincible). A unit of type High Templar can attack a unit if it is not invincible or not a building.
- **Unit About to Engage in Combat:** a unit is about to engage in combat when a target enemy is in sight range but is not yet attacking it. In the case of the military units the condition is a bit different. It is only when our unit is in weapon range to the target enemy, with a minimum distance of half of the unit sight, that the unit is about to engage. This was made to help these units prepare better their attack, because this way they are able to approach a little more closer to the enemies and realize if there are more units than the target.
- **Unit Can Attack:** a generic unit and the units of type Attacker, cannot start an attack only if its shields are not at its maximum power. The High Templars cannot start an attack if they do not have sufficient energy to cast their ability or if the ability weapon is in cool down.
- **Unit Prepared to Attack:** a generic unit is always prepared to attack, but the Attackers and High Templars do not and the situations in which they are not prepared are different. Considering the Attacker type, the unit is prepared to attack if one of the following is true:
    - the enemy unit is a worker type;
    - the real total attack power is greater than the attack power needed to kill the enemy unit, or if the allies in sight are less than 2/3 of the entire known army, the real attack power must be two times greater than the power needed.
    - If the attack power is lower and there are no more alive ally units of type Attacker known.

  To calculate the power attack needed, the following formula is used:

$$PowerNeeded = \frac{EnemyHealthPoints \times EnemyPower}{AllyHealthPoints}$$

Where:

$$Hit\ Points = HP, \qquad Shield\ Points = SP$$
$$Allies\ in\ sight = AIS, \qquad All\ alive\ allies = AAA$$
$$Dangerous\ Enemies\ in\ sight = DEIS$$
$$Firebat\ weapon\ damage = FWD, \qquad weapon\ damage = WD$$
$$Terran\ Bunker = TB, \qquad Protoss\ Zealot = PZ$$

$$EnemyHealthPoints = \begin{cases} TargetHP + TargetSP, & AIS > \frac{2}{3} AAA \\ \sum_{E \in DEIS} EHP + ESP, & AIS \leq \frac{2}{3} AAA \end{cases}$$

$$EnemyPower = \sum_{E \in DEIS} \begin{cases} FWD \times 2 \times 4, & E = TB \\ EWD \times 2, & E \neq TB \end{cases}$$

$$AllyHealthPoints = \sum_{A \in AIS} ASP$$

The real total attack power is:

$$AllyPower = \sum_{A \in AIS} \begin{cases} AWD \times 2, & A = PZ \\ AWD, & A \neq PZ \end{cases}$$

These calculations are not very accurate because two reasons. First of all because StarCraft have some random values associated with attack parameters like the cool down of the weapons. Secondly because there are so many variables to have in count that it would generate in a big equation. Since our goals are not perfection and efficiency, we decided to simplify and created a more common sense, human like calculation.

The reason for us to have two different calculations when we have more or less than 2/3 of the army in sight, is because we want the units to attack with the little risk possible and since the calculations are not very accurate, we want our units to wait for more allies if they exist.

On the other hand, the unit is not prepared to attack the target enemy if one of the following is true:

- If the target is a Terran Bunker and its hit points are below than 1/3 of its maximum hit points. This is just to avoid losing shields, since the building will be automatically destroyed;
- If the attack power is lower. However if there is no more ally units alive known of type Attacker but still are High Templars, the Ask HighTemplarForHelp property is enabled and the unit will try to do it.

Now considering the High Templar type, the unit for default prepared to attack, but it is not if one of the following is true:

- the High Templar has the information that its target is already going to be attacked by another High Templar;

    - the target is already under a psionic storm;
    - if the target is alone, i.e. do not have other enemy units nearby, and there is an ally unit of type Attacker that is closer to the enemy.
- **Unit is too close to enemy:** this happens when the distance between the unit and its target is lower than half of it sight range.

Every timeout in our implementation is counted based on the frame counting since the start of the game, which is the same used for the frame stamps.

5.3. Summary

The BWAPI framework architecture can be a bit complex, but fortunately it is not that difficult to use. Thanks to the IStarCraftBot interface, the interaction with the game is enabled via event handlers, namely the onFrame event, which we use to give the orders to the in game units, represented in by the BWAPI.Unit class.

To make the choices about what actions the unit should perform, the Unit class was created and to support it there is the Unit Record class, which gathers all the needed information about the ally and enemy units, such as their positions and health points.

To make the actual decisions of the actions the unit controlled by the Unit class must do, it has implemented seven principal behaviors which are represented as behavior trees.

Some behaviors are perhaps not the best choices or do not lead to the optimal outcome, but we are aware of that and it was not a goal that we defined at the beginning. Therefore, every decision about each behavior was taken, considering the time restraints of this project, the limitations of the game and the possible actions of the units only and in some situations also considering the more practical solutions. For instance, the defense behavior is not perfect, far from that, since the High Templars always attack the enemy, if they have energy, even if there are allies in the way. But it was a practical solution the resolve the defense situation. If the project was meant to continue this behavior would be revised.

6. Evaluation

In order to know if our solution has achieved the proposed goals, we made two types of evaluations:

- **Measurements at the end of each match:** Represented by the outcome of the corresponding game match. We recorded the time spend to the end of the match and the number of dead ally and enemy units. It was also taken note of the behaviors of each unit, namely how they searched their enemies, which enemy they chosen to attack, how they attacked (e.g. if they used always the same method), if they used all of their potential, i.e. used all their abilities and if they coordinated with each other, with same and different types of units.
- **Tests with users:** With a specific scenario we gathered a group of individuals to play against our AI and the original StarCraft AI, so they could compare each other and tell their opinions on their performances. At the end, we asked the testers to answer to a questionnaire regarding the experience they had, in order for us to evaluate them and take our conclusions about both AI's performances.

6.1. Measurements at the end of a match

With this type of tests we intend to gather some quantitative information to compare our AI with the original. This is important because we can have tangible information on the performance of each AI and thus make comparison much easier and believable.

6.1.1. Scenarios

Every scenario consists in two teams (player 1 and player 2), not necessarily with the same units, in an open area and there objective is to win the match, i.e. eliminate all opponent units. Player 1 always has its base at the bottom of the scenario and is controlled either by our implementation or the original AI. Player 2 always has its base at the top of the scenario and is controlled by the StarCraft AI with different behaviors depending on the scenario. Player 1 always controls Protoss units, whilst player 2 controls Protoss or Terran units.

Regarding what is known by each player, our AI only knows beforehand the static map information, i.e. the terrain, and does not know anything about the opposite player, i.e. its location and number of units. Because of this, it needs to explore the map to know what exists in the world, even ally units are unknown. On other hand, the original StarCraft AI always has all map information, like static map information and the position of every unit in the map. This applies if it is controlling either player 1 or player 2. Unfortunately we cannot do anything to change this, so we have to carry out with this disadvantage.

So we could evaluate every behavior we implemented and see how they work in practice, all the scenarios were made with unit's abilities and those behaviors in mind with each one of them testing something different. Every scenario was made with the help of the StarCraft in game tool "StarCraft Campaign Editor". The maps dimensions are indicated as walking tiles which correspond to a 32x32 in game screen pixels.

6.1.1.1. Scenario “1 bunker”

- **Map dimension:** 64 x 64 walking tiles
- Player 1
    - **Starts with**: 7 Zealots, 1 High Templar
    - **Behavior**: search and eliminate opponents
- Player 2
    - **Starts with**: 1 bunker, 4 Firebats inside
    - **Behavior**: wait and attack enemies on sight

In this scenario we try to see how the zealots cooperate with each other to destroy the bunker with the minimal possible casualties and what influence the High Templar has on the outcome of the play.

6.1.1.2. Scenario “27 zealots vs 27 zealots”

- **Map dimension:** 22 x 64 walking tiles
- Player 1
    - **Starts with**: 27 Zealots
    - **Behavior**: search and eliminate opponents
- Player 2
    - **Starts with**: 27 Zealots
    - **Behavior**: wait and attack enemies on sight

Here we want to see how a larger scale of units handles the battle with and against equal units, mainly to see how many units player 1 can end up with at the end of each match.

6.1.1.3. Scenario “27 zealots vs 27 zealots kamikaze”

- **Map dimension:** 22 x 64 walking tiles
- Player 1
    - **Starts with**: 27 Zealots
    - **Behavior**: search and eliminate opponents
- Player 2
    - **Starts with**: 27 Zealots
    - **Behavior**: rush attack the opponent units

This is basically the same scenario as the last one but as player 2 units rush to player 1’s base we want to see how well player 1 can handle with this massive attack and how it responds to it.

6.1.1.4. Scenario “30 protoss vs 30 protoss”

- **Map dimension:** 22 x 64 walking tiles
- Player 1
    - **Starts with**: 27 Zealots and 3 High Templars
    - **Behavior**: search and eliminate opponents
- Player 2

- **Starts with**: 27 Zealots and 3 High Templars
- **Behavior**: wait and attack enemies on sight

In this scenario we added three different units to the previous scenarios and see if the cooperation between the different units helps to accomplish the objective.

6.1.1.5. Scenario "30 protoss vs 30 protoss kamikaze"

- **Map dimension:** 22 x 64 walking tiles
- Player 1
  - **Starts with**: 27 Zealots and 3 High Templars
  - **Behavior**: search and eliminate opponents
- Player 2
  - **Starts with**: 27 Zealots and 3 High Templars
  - **Behavior**: rush attack the opponent units

Like the 27 zealot kamikaze scenario, in this one we want to see what happens when player 2 rushes, but this time with the High Templars in game to see if they make the difference.

6.1.1.6. Scenario "scenario 3"

- **Map dimension:** 64 x 64 walking tiles
- Player 1
  - **Starts with**: 21 Zealots and 4 High Templars
  - **Behavior**: search and eliminate opponents
- Player 2
  - **Starts with**: 28 Marines, 27 Firebats, 4 bunkers with 4 Firebat inside each and 2 Missile Turrets
  - **Behavior**: wait and attack enemies on sight

This scenario intends to show how Zealots and High Templars choose their attack priorities and see if a smaller number army can still win.

6.1.1.7. Scenario "defense 2"

- **Map dimension:** 22 x 64 walking tiles
- Player 1
  - **Starts with**: 8 Zealots and 2 High Templars
  - **Behavior**: search and eliminate opponents
- Player 2
  - Rounds:
    1. 1 Firebat, 15 seconds
    2. 1 Firebat, 1 Marine, 30 seconds
    3. 2 Firebats, 3 Marines, 40 seconds
    4. 3 Firebats, 3 Marines, 50 seconds
    5. 3 Firebats, 3 Marines, 1 Goliath, 60 seconds
    6. 6 Firebats, 60 seconds
    7. 3 Goliaths, 60 seconds

8. 5 Firebats, 2 Goliaths, 60 seconds
9. 6 Marines, 60 seconds
10. 3 Firebats, 2 Marines, 2 Goliaths, 60 seconds

- **Behavior**: rush attack the opponent units in rounds. Each round starts when all units form previous round are dead and the time for that round had expired.

In this scenario, player 2 instead of starting with all the units, they will appear round after round at the player's base. The objective of this scenario is to see how many rounds it takes for player 1 to lose all its units, or eventually win after the last round is completed.

6.1.1.8. Scenario "defense 3"

- **Map dimension:** 22 x 64 walking tiles
- Player 1
  - **Starts with**: 5 groups of 4 Zealots in different spots
  - **Behavior**: wait and attack enemies on sight
- Player 2
  - Rounds:
    1. 1 Firebat, 15 seconds
    2. 1 Firebat, 1 Marine, 30 seconds
    3. 2 Firebats, 3 Marines, 40 seconds
    4. 3 Firebats, 3 Marines, 50 seconds
    5. 3 Firebats, 3 Marines, 1 Goliath, 60 seconds
    6. 6 Firebats, 60 seconds
    7. 3 Goliaths, 60 seconds
    8. 5 Firebats, 2 Goliaths, 60 seconds
    9. 6 Marines, 60 seconds
    10. 3 Firebats, 2 Marines, 2 Goliaths, 60 seconds
    11. 7 Firebats
  - **Behavior:** rush attack the opponent units in rounds. Each round starts when all units form previous round are dead and the time for that round had expired.

Unlike the other scenarios, in this one player 1 will not search and attack the enemy, will instead stay idle in defense, waiting for the enemies to come. With this we try to see if they cooperate with each other to survive all rounds.

6.1.1.9. Scenario "defense continuous" offensive

- **Map dimension:** 22 x 64 walking tiles
- Player 1
  - **Starts with:** 5 groups of 3 Zealots in different spots
  - **Behavior:** search and eliminate opponents
- Player 2
  - **Rounds:** each round is composed by 2 Firebat units
  - **Behavior**: rush attack the opponent units in rounds. Each round last until the two units are both dead and at least 10 seconds has passed since they appeared.

The objective of this scenario is to know for how many rounds player 1 can survive until all of their units are dead.

6.1.1.10. Scenario “defense continuous” defensive

- **Map dimension:** 22 x 64 walking tiles
- Player 1
  - **Starts with:** 5 groups of 3 Zealots in different spots
  - **Behavior:** wait and attack enemies on sight
- Player 2
  - **Rounds:** each round is composed by 2 Firebat units
  - **Behavior:** rush attack the opponent units in rounds. Each round last until the two units are both dead and at least 10 seconds has passed since they appeared.

The difference from this scenario to the offensive one is the behavior of player 1. With that change in the behavior, we try to see if it has any influence on the outcome of the match.

6.1.1.11. Scenario “spawn 2”

- **Map dimension:** 22 x 64 walking tiles
- Player 1
  - **Starts with:** 13 Zealots. Every 20 seconds another zealot appears at the base.
  - **Behavior:** search and eliminate opponents
- Player 2
  - **Starts with:** 13 Zealots. Every 20 seconds another zealot appears at the base.
  - **Behavior:** rush attack the opponent units

In this scenario each player starts with the same number of units and every 20 seconds another one appears. The objective is to destroy the enemy building situated in the enemy base. With this scenario we want to see how the units behave when new units appear and what difference it makes from scenario “27 zealots vs 27 zealots” where the units are the same just in a bigger number.

6.1.1.12. Scenario “complete game test”

- **Map dimension:** 128 x 96 walking tiles
- Player 1
  - **Starts with:** A Nexus, 4 probes and 50 mineral resources
  - **Behavior:** Human player controls the workers, manages the resources and construct buildings and units. The AI search and eliminate opponents with military units like it does in the other scenarios.
- Player 2
  - **Starts with:** A Nexus, 4 probes and 50 mineral resources
  - **Behavior:** The Protoss insane campaign AI is controlling everything.

This scenario is completely different from the others because this is a complete game in which it is necessary to collect resources and construct units. As our AI was made only to control the

military units, all the other things needed are controlled by a human player. This test is only meant to our implementation because it is impossible to test it in these conditions where the AI cooperates with the human player by dividing tasks. The map is based in one that comes originally with the game which is the "(2)Astral Balance" scenario but with some tweaks. The differences are in the location of the resources and limitations in unit and building constructions. In the original scenario the resources are scattered all over the entire map, whereas in ours, the resources are only located near the base to avoid unnecessary complexity about base expansions. The limitations in terms of units are necessary since we did not have sufficient time to create decent behaviors for all types of units and therefore our AI is not prepared to deal with other types of units with different characteristics. For instance, our AI is not prepared to handle a combat with air units. So, the units were limited to the five already introduced units, plus the "Dragoon", a Protoss long range attack unit.

The objective of this scenario is to see if cooperation between the human player and the AI is practical but especially to enrich the results of the other scenarios with a more common type of match.

6.1.2. Tests and Results

For these tests the specified scenarios were used and all the information can be seen in the tables on the attachments section. Each scenario was played ten times for both AIs, with exception for the last scenario which was only tested three times.

6.1.2.1. Scenario "1 bunker"

- **Our implementation:** All units start to explore the map. When they find the bunker the zealots start grouping near the bunker and wait until they "think" they have sufficient power to destroy it, while the high Templar continues to explore. The zealots cannot destroy the bunker in a single attempt, since they retreat after their shields are almost gone to evade being killed. After the bunker is finally destroyed, the four firebats that were inside it appear and if the zealots can still engage in combat, they attack them, otherwise retreat. If the high Templar happens to know that firebats appeared, it attacks them, together with the remaining zealots.
  Due to the fact of the zealots retreat to recover their shields, they take much time to destroy the bunker, however more units survive at the end. This AI always won the match with an average of 2.50 zealots dead and zero high Templars within an average of 7:18 minutes.
- **Original AI:** As this AI cheats, by always knowing where all the units are, the zealots go straight to the bunker and attack it until it is destroyed, while the high Templar stays at a safe distance waiting. As the high Templar stays back and never attacks, this scenario in ten tries only finished three and that was because the zealots managed to kill all enemy units by themselves. On average they managed to kill 2.30 units, always destroyed the bunker and lost 6.50 zealots, while the high Templar always survived. It took in average 28 seconds to finish or stop acting (in cases which not finished).

6.1.2.2. Scenario “27 zealots vs 27 zealots”

- **Our implementation:** Due the limitations of the map, the exploration is almost absent and consequently they encounter the enemy very quickly. Like the bunker case, they only attack if they “think” they can win, thus they wait at the periphery of the enemy line, approaching each other with the intention of making a group which have sufficient power to defeat the enemy units. After an attack, units with low shield retreat, so it can recover it and later, when their shield is fully restored they come back to battle.
  In 10 tries, it always won the match with an average of 0.80 units dead and 4:22 minutes.
- **Original AI:** All 27 zealots gathered and focused immediately on the attack which only finished when all enemies were killed. It took only 51 seconds in average to finish, but lost 11.50 units in battle.

6.1.2.3. Scenario “27 zealots vs 27 zealots kamikaze”

- **Our implementation:** This time all 27 enemy units rush to our units and only stop when all of them are dead. On the other hand, our units at first try to attack back, but when their shields go down they begin to retreat. By retreating, the units go back to their base and gather around it, but since the enemy units follow them continuing the attack, they have no other alternative but to fight. However as they retreat they keep being attacked and therefore losing life.
  As a result this AI did not manage to win a single match, killing in average 10 enemy units and survived for 55 seconds.
- **Original AI:** In this case, the AIs controlling both players are the same, so as expected, the matches were balanced and therefore the correspondent results.
  On average player 1 lost 22.50 units whilst player 2 lost 24.40. The match took 45 seconds in average.

6.1.2.4. Scenario “30 protoss vs 30 protoss”

- **Our implementation:** In this scenario all 27 enemy zealots stay still, waiting for their enemies, on the other hand the three high Templars go forward and attack. Since that the high Templars are slower than player 1’s zealots, they are easily killed. Then the match goes much like the previous scenario, but this time with the help of the high Templars, making player 1 to finish the match more easily and faster.
  On average it lost 0.30 zealots and 0.50 high Templars and won all 10 matches in 4:14 minutes.
- **Original AI:** As in scenario “1 bunker” there was 7 matches that did not finish because the high Templars do not attack isolated units and all the zealots were killed. The time in average that took the matches was 7:26 minutes with a standard deviation of 6:47. We got a high value in the standard deviation because in some matches this AI was very quick to eliminate the enemies but other ones took too much time, were the zealots died very quickly, giving all the work to the high Templars which take a long time to restore their energy power needed for their psionic storm attacks.

On average player 1 lost 26.60 zealots, 1.20 high Templars and killed 25.50 zealots, 2.00 high Templars.

6.1.2.5. Scenario "30 protoss vs 30 protoss kamikaze"

- **Our implementation:** In the scenario where the player only has zealots available, this AI did not win a single match, but in this case, with the help of the three high Templars it turned out to be the opposite, winning all ten matches. What made that difference was, when player 1's zealots start retreating and their enemies pursue them, the high Templars attack, causing a great deal of damage. Then when they reach the base and start to attack again, their enemies do not have the advantage they had before, this time player 1's units have the advantage. We can see another thing happening, which is the high Templars at the base which cannot attack derived from the fact lack of energy to perform their attack, they merge into a new unit named Archon, which sometimes was proven crucial for the player to win the match.
  On average ten matches took 1:41 minutes to finish with 17.00 allied zealots, 0.00 high Templars and 0.10 Archons dead.
- **Original AI:** Again, as the AIs for both players are the same, the matches and the results were balanced, as expected. Both players rushed into each other and with the help of the high Templars the matches finished quicker than the "27 zealots vs 27 zealots kamikaze".
  It took on average 39 seconds with player 1 having 23.00 zealots and 1.90 high Templars dead and killed 24.40 zealots and 2.30 high Templars. None archon was created.

6.1.2.6. Scenario "cenário 3"

- **Our implementation:** As usual the units started the match by exploring the map until they found an enemy. At first, in the case of the zealots, they try to attack the opponent they encounter, but after they retreat and return to base and gather with their allies, they gain knowledge of the rest of the enemies and choose accordingly there next target. As a result we can see the zealots choosing to attack the bunkers at the left, and the high Templars attacking the Marines and the Firebats at the right. In the majority of the matches (8 out of 10) the zealots were not able to destroy all the bunkers by themselves, because almost everyone was killed, staying alive only 2, 3 or 4 zealots. What happens then is that they went back to the base and asked for help to the high Templars. If the high Templars are not busy, i.e. have enemy units which can be attacked, they merge in pairs to create an archon.
  On average it took 19:34 minutes to finish the match with 19.00 dead zealots and 0.10 high Templars. Also 1.40 archons were created and none died.
- **Original AI:** With this AI, player 1 failed to win a single match, although five of them have not finished due to the same reason why other scenarios did not finish as well. In this ten tries, this AI took two different approaches. Sometimes they group all the zealots and attacked the bunkers at the left one after another until all the zealots died, while the high Templars attacked the Firebats. Other times it group all the zealots and

attacked the Firebats, ignoring the bunkers. One way or the other, all the zealots end up dead.

On average each match lasted 6:32 minutes to finish or to stop acting, all zealots were killed and 2.70 of the high Templars too. They managed to kill zero turrets, 1.60 bunkers and 39.50 units.

6.1.2.7. Scenario "defense 2"

- **Our implementation:** In this scenario a Beacon was introduced. A beacon in this game is where new units appear. When player 1's units notice the existence of an enemy beacon and the map is all explored, instead of waiting in their base, they stay near the beacon waiting for the enemy to appear. Since the map is quite small comparing with the number of units, they explore it entirely very quickly and so, at the end of round 2 or 3, player 1 already have units waiting near the beacon. This behavior proves useful because the units can attack or step back accordingly, faster than if they waited at the base. However the player never succeeded in defeating all the ten rounds.

  On average it managed to reach round 8.30 where it killed 19.50 firebats, 9.60 marines and 4.90 goliaths, surviving for 4:06 minutes. In 8 matches the two high Templars merged and created an archon.
- **Original AI:** Since this AI always knows the position of its enemy units, they go directly towards them and therefore towards the beacon. But contrary to what our AI does they do not retreat, giving no time for the units to recover their shields. Without any coordination whatsoever, the player confined itself attacking the incoming units one by one and being killed one by one until they are all dead and therefore got worse results than our AI.

  On average it reached round 7.00 and killed 15.30 firebats, 9.00 marines and 2.30 goliaths in 3:19 minutes long.

6.1.2.8. Scenario "defense 3"

- **Our implementation:** This time, the behaviors to explore and wait near the beacon were removed, thus making it behave more like the original AI. What we can see is, as the enemy units attack, player 1's zealots begin to retreat slowly and as they found new ally units, which are positioned at their backs, they inform them about the existence of new enemy units. Those units then go forward in order to find and try to attack them. At the end of the 3rd or 4th round almost all of player 1's units are near the base. In the end, the player never managed to win, i.e. overcome all 11 rounds, but it reached the last round once.

  On average it reached round 10.00, killed 22.10 firebats, 15.10 marines and 6.50 goliaths within 5.16 minutes.
- **Original AI:** for the reason that they stand still at the same position waiting for the enemy units to enter on their sight range and since ally units would only help if the enemies were also on their sight, the units were not capable of giving a better resistance then our AI.

  On average it managed to reach round 9.30, kill 21.30 firebats, 12.90 marines and 5.80 goliaths and survived for 4:36 minutes.

6.1.2.9. Scenario “defense continuous” offensive

- **Our implementation:** In this scenario, the rounds are always composed by the same number of units which appear on regular time intervals and never stops. At first, when player 1’s units discover the enemy beacon they try to be close to it, but as they are attacked by the enemy rounds, some of them get their shields lowered and when a new group of enemy units arrive they retreat, resulting in fewer zealots staying near the beacon. And since the time needed for the zealots to completely recover their shields is bigger than the time it takes for a new group of enemy units appear, they no longer have the capacity to return near the beacon. For that reason they do not have sufficient time to counter the enemy attack without being constantly losing life. Nevertheless, they have a good performance and on average they survived for 13:30 minutes, killing 92.80 units.
- **Original AI:** With this AI in control, all the units, except for 2 zealots which stay back at the base, gather and attack the incoming enemies as they appear. In practice, it is always the same units that are closer to the enemies that attack and therefore are also those that take damage. As a result, the units that stay in the back only enter in action when the others in front are dead.
  Since the units in the back do not help the ones in the front, player 1 loses units very quickly, what caused it to last only 3:10 minutes on average, killing 35.70 units.

6.1.2.10. Scenario “defense continuous” defensive

- **Our implementation:** The same as the previous scenario, but without the explore behavior we can see that performance decreases a little. This happens because the units stay still waiting for the enemies to appear in their sight range, limiting their movement. Whereas when the units can explore, each unit can go near the beacon, as long as they have full shield, or go near the base when not, resulting in a better distribution of the units throughout the map and giving to each unit more time to regenerate its shield.
  On average player 1 survived for 12:49 and killed 81.20 units.
- **Original AI:** Unlike ours, the original AI increases the quality when their units stay in their positions waiting. That happens because in the offensive scenario the units rush to the enemy units, not giving any time for them to regenerate its shields, while in this they not. However it cannot surpass our implementation results.
  On average it survived for 6:20 minutes and killed 44:80 enemies.

6.1.2.11. Scenario “spawn 2”

- **Our implementation:** This scenario starts the same way as the “27 zealot vs 27 zealot” where the enemy charges against player 1 and their units retreat to the base. But as new units keep arriving, player 1’s units manage to kill the enemies and strikes back. However the enemy receives new units too, but those units stay at the base. As the time goes by player 1 manages to kill more enemies whilst not losing any and so eventually it will have sufficient power to eliminate all enemy units and destroy the enemy nexus.

In ten tries the objective was always accomplished taking on average 7:20 minutes to do it and killed 45.90 units.

- **Original AI:** This case was only tested three times because, as expected, the match do not finished. The first 13 units from both sides managed to kill each other, but then, the units that kept appearing stayed still in their own bases making the number of units increase without end and the match did not finish.

6.1.2.12. Scenario "complete game test"

- **Player 1 actions:** As the game starts without any military units, the single intervenient on Player 1 side is the human player, whom orders the workers to gather resources and the Nexus to build another probe. This behavior keeps going until there are sufficient resources income to construct the "Gateway", the building which is used to build Zealots and High Templars. As soon as the first military unit is constructed and comes into play, our AI takes control of it and starts exploring the map, whilst the human player continues with its duty to build more units and Gateways. Meanwhile Player 2, is doing the same as the human player, plus the management of its military units and when it has a certain number of units, it launches an attack to player 1's base. This attack is the main cause for player 1 to lose many units, but eventually all the enemy units are destroyed. At this point of the match the incoming resources increased a lot and therefore the human player starts building the necessary auxiliary buildings to enable the construction of High Templars and its ability to cast the psionic storm spell, while continuing to produce zealots. Units that come into play and the ones which recovered its entire shield points, start to govern the map and take the offense against the enemy base. Sooner or later player 1's units manage to outnumber the enemy force and win the match.

- **Player 2 actions:** At the beginning, when the limitations do not interfere with the intents of the AI, it performs normally, but after a while, it seems to lose interest in winning the match, creating very few units. It starts by creating a big group of zealots and when it has 12 of them ready, it orders them to attack player 1's base. Meanwhile it continues to produce units for the next attack wave, which is composed by 10 zealots and 5 dragoons. When the group is ready it orders them to attack the enemy base too. But at this moment, since the next wave would need another unit type that is not available, it almost stops acting by constructing too few units.

In the three matches done the human player took three different approaches in terms of construction speed. In the first match the human player tried to be perfect, by collecting resources and constructing units as fast as he could. The results were overwhelming as seen in **Table 2**. In the second match the human player tried to mimic the actions of player 2 in order to try to eliminate the human factor. For that it was used an in game cheat to allow seeing the entire map and therefore see the actions made by player 2. The results show that much more units were created but player 1 still has a great performance. In the third match the human player made the worst performance of the three, what caused player 1 to almost lose the game in the first attack wave. However it managed to recover and even won the game with good results.

	Player	Units Produced	Units Killed	Units Lost	Structures Lost
1st Match	1	88	91	6	0
	2	91	3	91	15
2nd Match	1	126	230	56	1
	2	230	38	230	23
3rd Match	1	102	88	33	6
	2	88	33	88	21

Table 2 - "complete game test" scenario results

6.1.3. Summary

Looking at the measurements made in all scenarios, including the information in **Table 3**, we learned interesting things. It is noticeable how our implementation outperformed the original AI in terms of unit survival. On the offensive scenarios, our AI managed to end their matches with more than half of its units alive, while the original AI had 80% of dead units in the end. This subject is most evident in scenarios where the enemy units do not rush against the ally units such as the "30 protoss vs 30 protoss", where in the majority of the tries there were no casualties for player 1.

However, this survival advantage comes with a cost of time consumption. We can see that the original AI completes its goal faster in every scenario and on the overall it lasted more than twice then the original AI to finish all the matches. Nevertheless there are scenarios where the original AI does not finish its goal because it stops acting or just because it lost all units, whereas our implementation always finishes the matches, almost every time being the winner. In fact all the losses of our implementation were in the same scenario, the "27 zealot vs 27 zealot" scenario. The main motive for our AI not perform better in this scenario, is due the collisions between the units that are running away, with the ones that are going in the direction of the enemies. In more recent RTS games, collision avoidance technics are used (e.g. in StarCraft 2) but in StarCraft they are not. Therefore, every time groups of units are going on a collision course with each other, they lose too much time redefining their routes, giving time to the enemy units to attack them.

What makes the original AI not finish some scenarios in some occasions is the fact that, High Templars often stay in the back and do not attack. This happens because with this AI, these units only attack if there are a large group of enemy units together or If they are being attacked and are going to die.

Offensive scenarios							
	Dead Allies		Time	Wins		Finished Matches	
Our AI	673	46%	06:20:32	50	83%	60	100%
Original AI	1169	80%	02:46:45	29	48%	41	68%
Max	1470		∞	60		60	

Table 3 – Totals from scenarios 5.1.1 to 5.1.6

From the scenarios “27 zealots vs 27 zealots kamikaze” and “30 protoss vs 30 protoss kamikaze” we can see how much helpful it is to have two different units cooperating with each other to eliminate their enemies. If in the first scenario our implementation was unable to win a single match, in the second it won all matches, while the original AI it was not seen much difference.

Looking at the totals on the defensive scenarios (**Table 4**), one more time our implementation managed to perform better than the original AI. As expected though, our implementation lasted more time in play and in consequence killed more units. This shows us that independently of the scenario, if both AIs have the same conditions, our implementation has better control over the units than the original AI. This is mostly caused by the fact that the units retreat when they have their shield low and also because they communicate with the ally units that are waiting near to the base, when they enter in their sight range, which makes them to know of the existence of the enemies. These behaviors result in an emergent cooperation between these units, where new units take the place of the units retreating, thereby maximizing the total life points of all of them. This is most obvious in the last scenario “spawn 2” where new units appear every 20 seconds, which makes this process occurs with more frequency.

Defensive scenarios		
	Dead Enemies	**Time**
Our AI	2403	05:56:49
Original AI	1390	02:56:01

Table 4 - Totals from scenarios 5.1.7 to 5.1.10

In the case of the last scenario, due to a strong influence that the human player has in the outcome of the match, we could not take very reliable results from it. However it can be used as a support for the results of the other scenario’s and consequently reinforces them. For that reason only three tests were made, which was sufficient for us to see that the results are consistent with the rest of the scenarios. Another reason that made us not continue the tests with this scenario was the behavior of the opponent AI. Due the limitations we made in the scenario, the opponent AI did not perform as well as expected, this happens because the original AI is completely scripted and therefore is not well prepared to act when its building choices are limited, which is the case.

6.2. Tests with users

To complement the results of the measures made in the previous section, it is necessary to have a more qualitative evaluation. For that reason, a play testing session was prepared and we let 20 players play against our AI and the original StarCraft AI, so that in the end they could respond to a questionnaire about their game experience. The reason that made us choose play testing over showing some videos for people analyze was because we wanted to have the opinions from their interaction with the AI. By seeing videos of gameplay, the person would

not be sufficiently immersed in the game to analyze the scenario as a player would, which is the objective of these tests.

6.2.1. Play testing

6.2.1.1. Scenario

To make these experiments, a scenario based in the “30 protoss vs 30 protoss” scenario was used, but with a little modification. The map dimensions were increased to 64x64 and the players have their bases at opposite corners of the map, specifically player 1 in the right lower corner and player 2 in the left upper corner. Since the units have more free space to walk, it enables the players to understand better some differences between the AIs, namely the explore capabilities of our units.

6.2.1.2. Test sessions

So the testers can have a baseline of comparison, they have the possibility to play against our AI and also the original AI. But since it is important that the testers do not know if they are playing against the original AI or our AI, they are referenced as AI A for the original and AI B for ours. It is great that the testers have this possibility, but to do it we need to be careful with the order in which each is played, because the first will undoubtedly influence the strategy of the tester when he plays against the second. To surpass this problem a commonly used solution is to divide the testers in two groups, where the sequence in which each AI is played is inverted, so group 1 plays against AI A first and AI B second and group 2 plays against AI B first and AI A second. In our case we were intercalating the testers making the group of the testers with even numbers and other group with the odd numbers.

To introduce the game controls, the scenario and the units and their abilities, an extra phase is added before the actual test begins. In this way we can make the player more acquainted to game and consequently when he is in the real test, he can focus more of his attention on the behaviors of the AI opponent.

To really evaluate and make conclusions on their experiments, two questionnaires were elaborated. One is about the tester itself about their gaming habits, experience and opinion. The other is about the match experience and their opinions on the AI opponent performance. This questionnaire is answered two times, one for each AI.

In practice each session was held as follows:

1. A summary of how the session will be held is made and the participant answer to a Player questionnaire;
2. It is made a quick tutorial about the game controls, unit types and capabilities and the scenario;
3. The participant can test what he learned in the tutorial and practice in the same scenario that the tests will be made;
4. The participant plays the first match against either the AI A or AI B according to his player number;

5. The participant answers to a questionnaire about the behaviors of his opponent concerning the experience he just had;
6. The participant plays the second match against the other AI that he did not play earlier;
7. The participant answers to a second questionnaire about the behaviors of his opponent concerning the experience he just had;
8. End of the session.

These sessions were made at the work laboratory at the university. We had two PCs, each one prepared for one IA. For AI A, i.e. the original AI, it was only necessary to have the StarCraft installed. For AI B, i.e. our AI, it was necessary the game installed and also a virtual machine with the StarCraft installed and the BWAPI framework, the mono and the chaos launcher. Since our AI runs like if a human player is playing, to play against it is necessary to play in multiplayer mode, thus the need of the virtual machine. What happens then is our AI is played in the virtual machine while the human player plays directly in the PC, as it would be expect. It is important that the scenarios are prepared when the testers sit in front of the PC, to minimize as much as possible the influence it can make. For that reason the testers only see the AI running when is there time to play against it.

6.2.2. Questionnaires

The first questionnaire about the player is important to introduce the tester to the subject and at the same time, is important for us to learn if the tester is experienced in StarCraft or at least to this type of games. We can also see the importance the tester gives to the AI in RTS games and how interesting they think it would be to have the AI helping the human player in different situations of a RTS game. All this questions are important to correlate with the other questionnaire about the AI and to take some conclusions about the differences between the AIs.

In order to make a good questionnaire about artificial intelligence we looked at the **Godspeed** questionnaires (18) to take some ideas. The article mention questionnaires to test robots and therefore most of them are useless to our work, except for the **perceived intelligence** questionnaire, which has exactly the questions we were looking forward.

This questions are a five point scale answer and on each one is asked to rate the impression of the robot on its Competency, Knowledge, Responsibility, Intelligence and Sense. In our case we decided to split the Knowledge question into six, because we wanted to have a more detailed opinion. So, we made our questionnaire with the Godspeed questions being the core questions, where the Knowledge question was split in knowledge about the location, the hit points and the capabilities of the ally and enemy units.

According to the article, it is important to have more questions mixed within to mask the intention. For that we looked into GameFlow (19), a model for evaluating player enjoyment in games and we saw that an important element is that a game should be challenging. Since one of the challenges in games is the AI opponents, we added five more questions about gaming, such as if the AI was challenging and if it surprised the player.

6.2.3. Results

20 people attended to our test sessions and with them we were able to have 10 testers for each group, which is a reasonable number of testers for this case. Each session took about 25 minutes to be completed.

To see if these results are statistically significant, we imported the results into the IBM SPSS Statistics 20[13] program.

The questionnaires can be found on the attachments section along with some charts that summarizes all the results.

6.2.3.1. Player questionnaires

A fast analysis of the testers based on the Player questionnaire shows us that:

- Almost every participant is in the age of 21 to 30 years old and is male;
- The majority plays games every day;
- The majority thinks that AI is important or very important in a videogame;
- Almost every participant have already played a RTS game;
- From those who already played an RTS:
    - Many participants had never played StarCraft or StarCraft 2 in the past or has little experience with it, but almost every participant is somewhat experienced in RTS in general;
    - Almost every participant thinks it is important that the opponent's AI should be difficult to defeat;
    - The majority thinks that it is very important that the AI acts like a human player would and should adapt its behaviors according the situation it is in;
    - The majority also thinks it is very important that the human player have access to the same options and conditions as the AI;
    - On average, the participants would like to have the option to have the AI help them with the map exploration, construction of units and manage the resources gathering. But on the other hand, they would not want help to construct buildings nor to manage the attacking and defending units. But in general, they would like to have these kind of options enabled in future RTS games.

[13] http://www-01.ibm.com/software/analytics/spss/

6.2.3.2. Match Results

As we have done in the test scenarios specified in section 6.1, at the end of each match we took note of the number of ally and enemy units killed.

	Units Lost			Units Killed			Matches Won	
	average	%	σ	average	%	σ	total	%
Original AI	27,30	91	4,17	24,15	80.5	7,02	9	45%
Our AI	28,65	95,5	3,59	13,05	43.5	9,92	3	15%

Table 5 - Results from all 20 tester's matches

By looking to the results detailed in **Table 5**, we can see that either against the original AI and our AI, on average the testers lost almost the same number of units. But if we look to the number of matches won, we see that almost half of the testers managed to win the match against the original AI, whilst only 3 managed to win against ours. This happens because, if we look at the units killed, we see that the testers managed to kill much more units while playing against the original AI. It is also true that the standard deviation is quite big, but nevertheless it is a great difference.

In order to evaluate the statistical significance of these results a paired difference test was made, where both AIs make a pair for each measure, which are the units killed, units lost and matches won. To know which test we must use, we had to analyze the results to see if we have a normal distribution or not and to do so we made the Kolmogorov-Smirnov test. Since all three pairs have at least one value under the 0.05 significance, the test to make must be the Wilcoxon signed-rank test. These values can be seen in the attachments section.

	Units Lost	Units Killed	Matches Won
Significance (2-tailed)	0,122	0,001	0,034

Table 6 - Wicoxon signed-rank test significance results for units measures

As we can see in **Table 6** that the number of units lost is not significant. This was expected since the results are very similar. But if we see the units killed and matches won, they are statistical significant.

An interesting result is that the testers that won against our AI have different videogame experiences. Two of them had never played StarCraft before, however one is very experienced in RTS while the other is experienced in RTS and somewhat experienced in StarCraft 2. On the other hand, the other tester was a little experienced in StarCraft but as much experienced in RTS in general as the other two.

	Original AI		Our AI	
	Units Lost	Units Killed	Units Lost	Units Killed
Original -> Ours	27,5	25,4	27,3	17,4
Ours -> Original	27,1	22,9	30	8,7

Table 7 - Comparison between the order of the AIs with which each tester played against

With the information in **Table 7** we see that splitting the groups, when the testers played against the original AI, the values are very close, but against our AI there are great differences in the number of units killed. We can even see that nobody managed to win our AI in the first match.

6.2.3.3. AI questionnaires

As mentioned in the session guide, the testers after playing a match, answered to a questionnaire regarding their thoughts about the opponent AI they just played against.

			Original AI		Our AI	
			average	σ	average	σ
Resistance			3,85	1,27	4,50	0,76
Surprise			3,50	1,10	4,20	0,83
Competency			3,80	1,10	4,55	0,76
Knowledge	**Location**	**Enemies**	4,35	1,04	4,00	0,92
		Allies	3,35	1,09	4,00	1,03
	Hit Points	**Enemies**	3,10	1,07	3,50	0,89
		Allies	3,10	1,21	4,00	0,86
	Capacities	**Enemies**	3,20	1,10	4,00	1,08
		Allies	3,65	0,87	3,75	0,91
Responsibility			3,45	1,23	4,35	0,67
Intelligence			3,35	1,14	4,30	0,73
Sense			3,25	1,07	4,35	0,74
Individual Behavior			2,95	1,54	2,80	1,40
Greatest Difficulty			40% - Opponent AI		55% - Opponent AI	
Cheated			80% - No		100% - No	

Table 8 - Results from the AI questionnaires (the higher the value the better)

In **Table 8** we have the average answers to the questions made, including the Godspeed questions. In the individual behavior scale, values closer to 1 means that the orders were given to a group of units and values closer to 5 means the units make their own decisions and do not receive orders from anything.

As we can see, our AI has bigger values in practically all the categories, only the knowledge of the location of the enemy units and the individual behavior is the opposite. In the knowledge of the location of the units, it is easily explained with the fact that all the units with the original AI attacked directly the player. Unfortunately, that fact only happens because the original AI cheats on that knowledge and therefore the results are unfair, although the results are not very different.

To the question about the most difficult aspect of the game, testers also answered more times that was the AI when played against our AI. Unlike what we expected, only 20% realized that the original AI cheated because they attacked them directly, which should not be possible. But

this result is possible due the fact that it is normal in this type of games the AI cheat and the testers know that subconsciously.

One measure we wanted to obtain was the perceived intelligence mentioned in the Godspeed questionnaires. To do that, we combined the six different scales regarding the knowledge questions and created the scale Knowledge. Then we did the same thing but this time with the five perceived intelligence questions results, in order to create one single scale, which is the Perceived Intelligence scale.

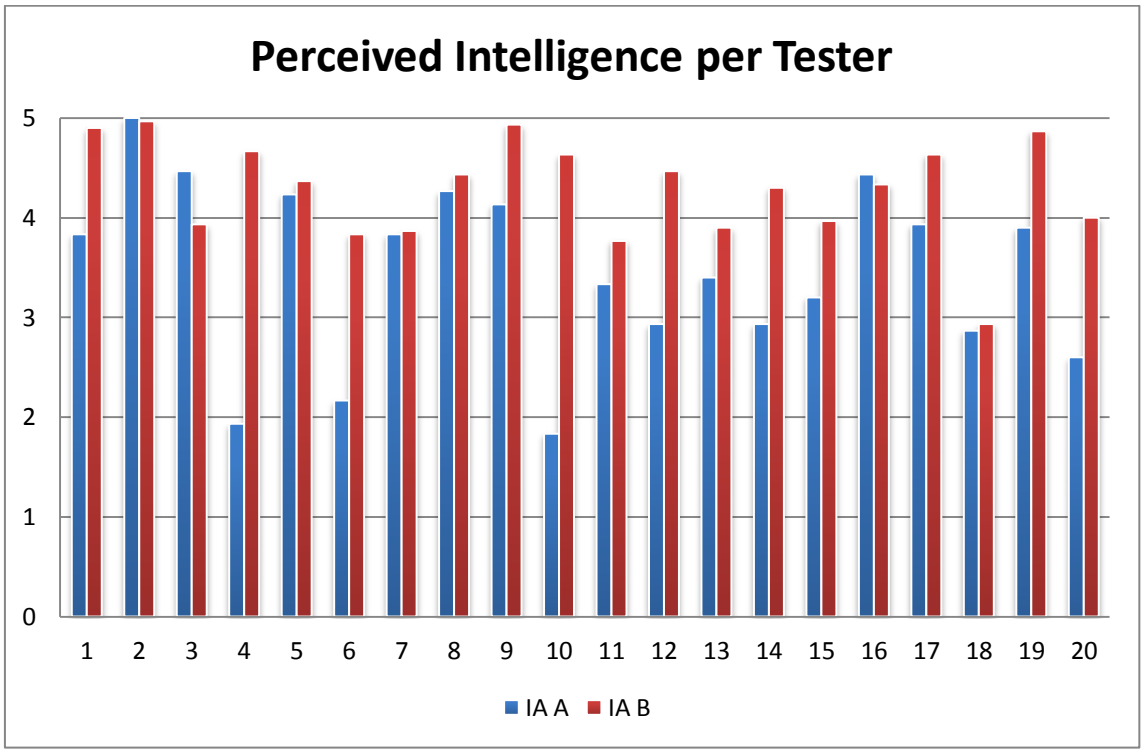

Chart 1 - Perceived intelligence values per tester

By looking to **Chart 1** we can see that almost every tester thinks our AI has more perceived intelligence than the original AI. However there are three testers that think the opposite, but the differences are very low and above or near the value 4, which is good.

	Average	σ	Cronbach's Alpha
Original AI	3,46	0,89	0,89
Our AI	4,28	0,51	0,78

Table 9 - Perceived Intelligence results

With a Cronbach's Alpha over 0.7 seen in **Table 9**, which Nunnally (20) recommends being the minimum value, the results from both AIs are internally consistent. However both AIs are above the middle value 3, it is obvious the difference between them, with our AI being almost one value higher in the scale than the original AI and also with the standard deviation being lower.

In order to evaluate the significance of these results another paired difference test was made, where this time, both AIs make a pair for each question. The Kolmogorov-Smirnov test, made

with the results from the questionnaires, shows that all the pairs have at least one value under the 0.05 significance, except for the single scale Knowledge and the perceived intelligence. These values can be seen in the attachments section. So, for these two measures, the test to do must be the Paired Student's t-test, for the other ones, the test to make must be the Wilcoxon signed-rank test.

			Significance (2-tailed)	
Resistance			0,051	0,025
Surprise			0,018	
Competency			0,021	
Knowledge	**Location**	**Enemies**	0,273	0,137
		Allies	0,080	0,040
	Hit Points	**Enemies**	0,163	0,081
		Allies	0,012	
	Capacities	**Enemies**	0,073	0,037
		Allies	0,793	0,396
	Single scale		0,032	
Responsibility			0,003	
Intelligence			0,003	
Sense			0,002	
Individual Behavior			0,729	0,365
Perceived Intelligence			0,000	

Table 10 - Paired difference test significance results for the questionnaire meausres

In **Table 10** we have the significance results from the Wilcoxon test and it confirms that almost every pair differences are significant, because they are under the maximum value, 0.05 derived from the 95% confidence interval. The results with two columns means that the result was above the 0.05 value, but since the results are 2-tailed we can divide the value by two, and the second column represents that value. This means that the individual behavior and the knowledge of the enemy location, hit points and the capacities of the allies are not statistically significant. Nonetheless, the most important measure we wish to see is the perceived intelligence and that one is undoubtedly significant.

6.2.4. Summary

Looking at the match results we can see that:

- The fact that the number of units lost with both AIs is slightly the same, but the number of matches won is not, clearly means that our AI has better unit control then the original AI;
- The standard deviation in the units killed against our AI can be explained with the strategy that each tester used against the AI, a more offensive strategy lead to more units being killed, while a more defensive one lead to less;

- The testers who have won against our AI have experience playing RTS games but not necessarily in StarCraft. But if we look at the other testers that did not win, there are some that are more experienced in StarCraft or StarCraft 2. This means that our AI can be very challenging for those who are familiar to the game and at the same time more accessible to the less experienced ones. This gains more consistency when we see individually each tester that not won, but killed many units and noticed that they are also more or less experienced in RTS. In the end, the ones that are more experienced in StarCraft and because of that were more likely to win, did not.
- The differences in the order of the AIs in which each tester played against, can be explained with the fact that, when they fought against our AI in second place, they began the match more alert, because the original AI attacked very quickly with every unit. Because of that they were more prepared then the other way around. Nevertheless, the results have statistical significance.

It was interesting to see that some testers would like to have the option, in RTS games, to have an AI helping them with some in game subjects. However, in their opinion the AI helping with the offensive strategies is not that good.

Regarding the AI questionnaires answered by the testers, the individual behavior was perhaps the question that the testers took much time to answer. Something we already expected since the subject is very difficult to evaluate, which explains the results obtained that are very similar in both AIs and very close to number 3, the middle of the scale.

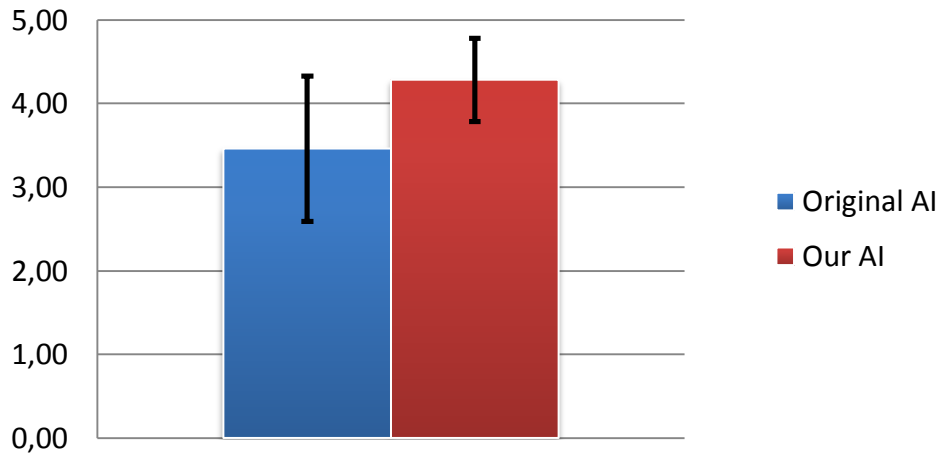

Chart 2 - Perceived Intelligence

By looking to the mean values obtained in the questionnaire and there statistical results, we can say that the testers think our AI is more competent, knowledgeable, responsible, intelligent and sensible. Those five characteristics are part of the Perceived intelligence Godspeed IV questionnaire[14] and with a Cronbach's Alpha over 0.7, the results are coherent. Since the statistical Paired Student's t-test results shows the questionnaire results are significant, is notorious the perceived intelligence that our AI has, compared to the original StarCraft AI (see **Chart 2**).

[14] http://www.bartneck.de/2008/03/11/the-godspeed-questionnaire-series/

6.3. Conclusions

After all the experiments made and measures taken we gain knowledge about our AI that we could not obtain by any other way.

Looking at the scenarios and to both AIs, the biggest conclusion we made was that when the units coordinate with each other and have notion of their heath points, they can finish more scenarios being the winners and with less casualties. Although it was expected that our units would run away, since the AI was programmed to make the unit retreat if its shields were lower, it was great to see that a good strategy emerged and it is an efficient way to win a match. This behavior also proved to be the core of the strategy that ended to emerge in the overall attack process.

We can also compare our AI with the original by making an analogy to the evolution of the real battles over the centuries. The original AI is similarly to what used to happen in the past in those great and epic battles where the main weapons were swords, bows and arrows. On those battles the soldiers were in line formations and the ones in the back could not do anything until the soldiers in front were killed. Basically they were all like cannon fodder and each soldier was just a number. While our AI is more like what happens in more recent wars, where each soldier is the most important thing and everything must be done to win a battle with the fewer casualties possible.

From the results of the questionnaires, units lost/killed and commentaries made by testers about both the original AI and our AI, we conclude that the testers were very surprised with the behaviors of our AI and showed much more satisfaction playing against our AI then the original. It was also notorious that the testers understood the self-preservation “feeling” that the units have.

Unfortunately, the testers did not realized that our units were being controlled by a single entity, since the results to the question about their individual behavior was very close to the middle value and statistically it did not had significance. However, the results for the original AI were identical, so we cannot take great conclusion on this one, except that this is perhaps very difficult to analyze and also the behavior of the units can be deceiving.

In general we can say that our AI is superior to the original AI in almost every way. The results from the scenario tests between AIs show that is true. The play testing results and the confidence degree given by the statistical results based on the questionnaires, also tell us that it is true.

7. Final Thoughts

7.1. Conclusions

As the videogame industry has been evolving, the different subjects of gaming have not being growing equally. That is the situation of game AI, which has been evolving much slowly than graphics and only more recently it started to have more weight. Yet, there are some evolutions and one good addition to industry is the behavior trees.

As we have seen in the results of the questionnaires made, people are giving more importance to AI in games and above that, they think that it should have a behavior similar to one of a human being. Since coordination is something that we humans do all the time, most times without noticing, coordination is an important feature that games should have.

Some attempts to use coordination with behavior trees have been done, as we saw in the related work section, and even other approaches using different structures. But most of the solutions we see support their coordination in a centralized entity, mainly blackboards. With our solution we have decentralized unit coordination where each unit behaves only based on its own knowledge.

A great characteristic of behavior trees is that they are agent oriented, so they are the right choice to use in this project. Looking to the behavior diagrams in section 5.2.1, we can see that behavior trees are also very useful to show and explain agent's behaviors and in fact, that is one of its great qualities.

To demonstrate our solution we used the well-known commercial game StarCraft. With it we created test scenarios and compare the performance of our AI against the best original game AI. The results were interesting, showing that our AI was almost every time better than the original AI, mainly in the number of surviving units.

That happened because a very interesting behavior emerged. It was accomplished with simple instructions, namely the instruction to retreat from battle when the unit is getting too much damage, keeping score of the position of the enemies and their hit points and pass that information to the allies that it encounters in the way back to the base or when exploring the map. The results of the AI questionnaires reinforced this idea and the testers confirmed that our AI is more intelligent and has a more interesting and more human like behavior than the original StarCraft AI.

The "complete game scenario" also helped to understand that cooperation between the human player and the AI in this type of games is valid and helpful. However the testers answered that they would not be interested in having the AI controlling the units for them, preferring that the AI would help with other things like gathering resources.

The decentralization turned out to be very important in our solution. Not only was easy to implement the behaviors for the units, but also enabled the emergence of the overall offensive strategy. However the testers did not realize that and thought that that behavior was a preprogrammed strategy. But it is acceptable because the behavior is very deceiving and also we can think that because we are in a game environment, what matters to the player are the

behaviors that he perceives the opponent is taking. They do not care if the units are being controlled or not by another entity.

At the end the goals of the project were accomplished and the implementation gave good results, quantitatively and qualitatively.

We hope that the quality of AI in videogames continues to grow more and more and that in the future we can see NPCs acting more like human beings, not necessarily in RPG games but also in RTS games, or even other genres.

7.2. Future Work

After the conclusion of this project, much more work can be done, mainly to bring the results scale to a higher level.

This implementation was made with a great amount of limitations at the StarCraft level. Few types of units were used and therefore much of the game itself was lost in our experiments. However it would be very interesting to tweak this AI implementation to make use of the other types of units. Following that, a larger test could be made, based on the “complete game test” but this time much closer to a real StarCraft custom match, without limitations of any kind.

In terms of implementation, performance was never a goal of this project, however if it was to be continued, many implementation details should be revised, mainly because when the number of units start to grow a lot, the FPS begin to be too low. Although not being the focus of this project, one thing that could be done would be to have different Unit Record classes for the ally and enemy units, because most of the information that this objects have is just needed for the enemies, thus the ally units should have a simplified version of the class.

If we look at each behavior in detail, we realize that almost every unit behavior has strategic flaws, some more serious than others. If the project would to be continued the behaviors should be improved, so the results in both scenario tests and play testing could be improved too. For instance the explore behavior, in current version the units explore randomly. This can be significantly improved, taking advantage of the communication that our agents already have, they could take some decisions on where to explore based on information about ally and enemy locations.

8. References

1. **Peter Hart, Nils Nilsson, Bertram Raphael.** A Formal Basis for the Heuristic Determination of Minimum Cost Paths in Graphs. *IEEE Trans. on Systems Science and Cybernetics, Vol. SSC-4, No. 2, pp 100-107.* 1968.

2. **Ryan Houlette, Dan Fu.** The Ultimate Guide to FSMs in Games. *AI Game Programming Wisdom 2, Charles River Media.* 2003.

3. **Alex Champandard.** Behavior trees for next-gen game AI. *AiGameDev.com.* 2008.

4. **Michael Schumacher.** *Objective coordination in multi-agent system engineering: design and implementation, Springer-Verlag.* 2001.

5. **Dana S. Nau, Stephen J. J. Smith, Kutluhan Erol.** Control Strategies in HTN Planning: Thoery versus Practice. *Proceedings of AAAI/IAAI-98.* 1998.

6. **Ian Millington, John Funge.** *Artificial Intelligence for Games, Morgan Kaufmann.* 2009.

7. **Craig Reynolds.** Steering Behaviors For Autonomous Characters. *Game Developers Conference.* 1999.

8. **R. Beckers, O.E. Holland, J.L. Deneubourg.** From Local Actions to Global Tasks: Stigmergy and Collective Robotics. *Artificial Life IV, MIT press.* 1994.

9. **Damian Isla.** Handling Complexity in the Halo 2 AI. *Proceedings of Game Developers Conference.* 2005.

10. **Chong-U Lim, Robin Baumgarten and Simon Colton.** Evolving Behaviour Trees for the Commercial Game DEFCON. *doc.ic.ac.uk.*

11. **Michael Mateas, Andrew Stern.** Façade: An Experiment in Building a Fully-Realized Interactive Drama. *Game Developers Conference, Game Design track.* 2003.

12. **Ricard Pillosu.** Coordinating Agents with Behavior Trees. *Paris Game AI Conference.* 2009.

13. **Maria Cutumisu, Duane Szafron.** An Architecture for Game Behavior AI: Behavior Multi-Queues. *Proceedings of the Fifth Artificial Intelligence for Interactive Digital Entertainment Conference.*

14. **Michael Mateas, Andrew Stern.** A Behavior Language: Joint Action and Behavioral Idioms. *Life-like Characters: Tools, Affective Functions and Applications.* 2004.

15. **Whittaker, Marc Cavazza and Steven J. Mead and Alexander I. Strachan and Alex.** A Real-Time Blackboard System for Interpreting Agent Messages. *GAME-ON'00.* 2000.

16. **Lima, Rodrigo B V and Tedesco, Patrícia A Restelli and Ramalho, Geber L.** Dance One Scenario for Choreographic Coordination of Intelligent Agents in Entertainment Applications. *Anais do Simpósio Brasileiro de Jogos de Computador e Entretenimento Digital SBGAMES.* 2006.

17. **Tan, Chek Tien and Cheng, Ho-lun.** A combined tactical and strategic hierarchical learning framework in multi-agent games. *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games.* Los Angeles, California, 2008.

18. **Bartneck, Christoph and Kulic, Dana and Croft, Elizabeth A. and Zoghbi, Susana.** Measurement Instruments for the Anthropomorphism, Animacy, Likeability, Perceived Intelligence, and Perceived Safety of Robots. *I. J. Social Robotics.* 2009.

19. **Sweetser, Penelope and Wyeth, Peta.** GameFlow: a model for evaluating player enjoyment in games. *Comput. Entertain.* 2005.

20. **JC, Nunnally.** *Psychometric theory, 2nd edn.* McGraw-Hill, New York, 1978.

A. Scenario Tests Measures

		Time			Units Lost					Units Killed			
Simulation		Our AI	Original AI		Our AI		Original AI			Our AI		Original AI	
					Zealot	High Templar	Zealot	High Templar		Firebat	Bunker	Firebat	Bunker
				begin:	7	1	7	1	begin:	4	1	4	1
1	-->	06:07	00:26	-->	0	0	7	0	-->	4	1	2	1
2	-->	08:05	00:29	-->	4	0	7	0	-->	4	1	1	1
3	-->	06:42	00:29	-->	4	0	7	0	-->	4	1	2	1
4	-->	05:56	00:26	-->	3	0	7	0	-->	4	1	0	1
5	-->	12:25	00:28	-->	5	0	7	0	-->	4	1	4	1
6	-->	07:17	00:26	-->	2	0	5	0	-->	4	1	4	1
7	-->	07:35	00:29	-->	1	0	7	0	-->	4	1	2	1
8	-->	06:42	00:27	-->	3	0	7	0	-->	4	1	3	1
9	-->	07:52	00:28	-->	3	0	4	0	-->	4	1	4	1
10	-->	04:21	00:27	-->	0	0	7	0	-->	4	1	1	1
	mean:	07:18	00:28	mean:	2,50	0,00	6,50	0,00	mean:	4,00	1,00	2,30	1,00
	σ:	02:00	00:01	σ:	1,63	0,00	1,02	0,00	σ:	0,00	0,00	1,35	0,00

Table 11 - 1 bunker scenario measures

Simulation		Time			Units Lost			Units Killed	
		Our AI	Original AI		Our AI	Original AI		Our AI	Original AI
				begin:	27	27	begin:	27	27
1	-->	05:28	00:56	-->	0	17	-->	27	27
2	-->	04:29	00:51	-->	2	8	-->	27	27
3	-->	03:30	00:46	-->	0	15	-->	27	27
4	-->	02:39	00:51	-->	1	9	-->	27	27
5	-->	04:10	00:51	-->	0	12	-->	27	27
6	-->	04:22	00:55	-->	0	14	-->	27	27
7	-->	05:42	00:48	-->	3	13	-->	27	27
8	-->	04:04	00:51	-->	1	11	-->	27	27
9	-->	04:25	00:50	-->	0	10	-->	27	27
10	-->	04:51	00:47	-->	1	6	-->	27	27
	mean:	04:22	00:51	mean:	0,80	11,50	mean:	27,00	27,00
	σ:	00:50	00:03	σ:	0,98	3,20	σ:	0,00	0,00

Table 12 - 27 zealot vs 27 zealots scenario measures

Simulation		Time			Units Lost			Units Killed	
		Our AI	Original AI		Our AI	Original AI		Our AI	Original AI
				begin:	27	27	begin:	27	27
1	-->	00:51	00:42	-->	27	19	-->	14	27
2	-->	00:55	00:41	-->	27	16	-->	16	27
3	-->	00:56	00:45	-->	27	27	-->	13	19
4	-->	00:47	00:44	-->	27	21	-->	2	27
5	-->	00:48	00:54	-->	27	27	-->	12	23
6	-->	01:04	00:45	-->	27	24	-->	6	27
7	-->	00:56	00:46	-->	27	21	-->	10	27
8	-->	00:56	00:44	-->	27	19	-->	11	27
9	-->	00:55	00:40	-->	27	27	-->	8	13
10	-->	00:58	00:50	-->	27	24	-->	8	27
	mean:	00:55	00:45	mean:	27,00	22,50	mean:	10,00	24,40
	σ:	00:05	00:04	σ:	0,00	3,69	σ:	3,92	4,57

Table 13 - 27 zealot vs 27 zealot kamikaze

Sim.		Time			Units Lost					Units Killed				
		Our AI	Original AI		Our AI		Original AI			Our AI		Original AI		
					Zealot	High Templar	Zealot	High Templar		Zealot	High Templar	Zealot	High Templar	
				begin:	27	3	27	3	begin:	27	3	27	3	
1	-->	04:05	11:09	-->	0	1	27	2	-->	27	3	25	1	
2	-->	03:58	07:48	-->	1	0	27	1	-->	27	3	26	2	
3	-->	06:50	06:22	-->	0	2	27	1	-->	27	3	23	2	
4	-->	04:48	09:28	-->	1	1	27	1	-->	27	3	25	2	
5	-->	04:10	22:55	-->	0	0	27	2	-->	27	3	23	2	
6	-->	03:24	12:58	-->	0	0	27	1	-->	27	3	26	0	
7	-->	03:51	00:54	-->	0	0	24	1	-->	27	3	27	3	
8	-->	03:17	01:03	-->	0	0	27	1	-->	27	3	27	3	
9	-->	03:58	00:52	-->	0	0	27	0	-->	27	3	26	2	
10	-->	03:57	00:52	-->	1	1	26	2	-->	27	3	27	3	
	mean:	04:14	07:26	mean:	0,30	0,50	26,60	1,20	mean:	27,00	3,00	25,50	2,00	
	σ:	00:57	06:47	σ:	0,46	0,67	0,92	0,60	σ:	0,00	0,00	1,43	0,89	

Table 14 – “30 protoss vs 30 protoss” scenario measures

Sim.		Time			Units Lost					Units Killed					Archons			
		Our AI	Original AI		Our AI		Original AI			Our AI		Original AI			Our AI		Original AI	
					Zealot	HT	Zealot	HT		Zealot	HT	Zealot	HT		Created	Killed	Created	Killed
				begin:	27	3	27	3	begin:	27	3	27	3	begin:	0	0	0	0
1	-->	01:18	00:48	-->	22	0	27	3	-->	27	3	14	0	-->	1	0	0	0
2	-->	00:47	00:31	-->	6	0	15	2	-->	27	3	27	3	-->	1	0	0	0
3	-->	02:31	00:30	-->	24	0	23	1	-->	27	3	27	3	-->	1	0	0	0
4	-->	00:50	00:51	-->	10	0	27	3	-->	27	3	25	2	-->	1	0	0	0
5	-->	04:57	00:31	-->	27	0	24	0	-->	27	3	27	3	-->	1	1	0	0
6	-->	01:09	00:44	-->	5	0	24	1	-->	27	3	27	3	-->	1	0	0	0
7	-->	00:51	00:25	-->	16	0	27	3	-->	27	3	16	0	-->	1	0	0	0
8	-->	01:57	00:29	-->	26	0	12	2	-->	27	3	27	3	-->	1	0	0	0
9	-->	00:54	00:41	-->	15	0	24	1	-->	27	3	27	3	-->	1	0	0	0
10	-->	01:35	00:59	-->	19	0	27	3	-->	27	3	27	3	-->	1	0	0	0
	mean:	01:41	00:39	mean:	17,00	0,00	23,00	1,90	mean:	27,00	3,00	24,40	2,30	mean:	1,00	0,10	0,00	0,00
	σ:	01:13	00:11	σ:	7,60	0,00	5,02	1,04	σ:	0,00	0,00	4,76	1,19	σ:	0,00	0,30	0,00	0,00

Table 15 – "30 protoss vs 30 protoss kamikaze" scenario results

Sim.		Time			Units Lost					Units Killed							Archons			
		Our AI	Original AI		Our AI		Original AI			Our AI			Original AI				Our AI		Original AI	
					Zealot	HT	Zealot	HT		Unit	Bunker	Turret	Unit	Bunker	Turret		Created	Killed	Created	Killed
				begin:	21	4	21	4	begin:	71	4	2	71	4	2	begin:	0	0	0	0
1	-->	18:15	06:37	-->	19	0	21	4	-->	71	4	2	29	3	0	-->	0	0	0	0
2	-->	16:27	06:22	-->	19	0	21	4	-->	71	4	2	34	3	0	-->	2	0	0	0
3	-->	21:47	04:48	-->	16	0	21	4	-->	71	4	2	39	3	0	-->	2	0	0	0
4	-->	20:51	08:03	-->	21	0	21	4	-->	71	4	2	37	2	0	-->	2	0	0	0
5	-->	15:18	04:55	-->	15	0	21	0	-->	71	4	2	40	0	0	-->	0	0	0	0
6	-->	17:05	01:47	-->	20	0	21	0	-->	71	4	2	40	0	0	-->	1	0	0	0
7	-->	22:20	04:46	-->	20	0	21	4	-->	71	4	2	35	2	0	-->	2	0	0	0
8	-->	28:52	03:11	-->	20	0	21	1	-->	71	4	2	41	0	0	-->	2	0	0	0
9	-->	17:54	18:32	-->	19	0	21	3	-->	71	4	2	60	3	0	-->	2	0	0	0
10	-->	16:48	06:22	-->	21	1	21	3	-->	71	4	2	40	0	0	-->	1	0	0	0
		mean:	19:34	06:32	mean:	19,00	0,10	21,00	2,70	mean:	71,00	4,00	2,00	39,50	1,60	0,00	mean:	1,40	0,00	0,00	0,00
		σ:	03:50	04:21	σ:	1,90	0,30	0,00	1,62	σ:	0,00	0,00	0,00	7,68	1,36	0,00	σ:	0,80	0,00	0,00	0,00

Table 16 – "cenário 3" scenario measures

Sim.		Time			Units Lost					Units Killed							Archons			
		Our AI	Original AI		Our AI		Original AI			Our AI			Original AI				Our AI		Original AI	
					Zealot	HT	Zealot	HT		Firebat	Marine	Goliath	Firebat	Marine	Goliath		Created	Killed	Created	Killed
				begin:	8	2	8	2	begin:	0	0	0	0	0	0	begin:	0	0	0	0
1	-->	03:01	03:10	-->	8	0	8	2	-->	16	9	1	16	9	1	-->	1	1	0	0
2	-->	03:58	02:42	-->	8	2	8	2	-->	17	9	5	14	9	1	-->	0	0	0	0
3	-->	04:00	02:35	-->	8	2	8	2	-->	21	9	5	11	9	1	-->	0	0	0	0
4	-->	04:32	04:00	-->	8	0	8	2	-->	21	9	6	17	9	4	-->	1	1	0	0
5	-->	03:57	03:22	-->	8	0	8	2	-->	19	9	5	16	9	1	-->	1	1	0	0
6	-->	04:37	04:02	-->	8	0	8	2	-->	21	13	6	20	9	4	-->	1	1	0	0
7	-->	03:58	02:36	-->	8	0	8	2	-->	20	9	5	11	9	1	-->	1	1	0	0
8	-->	03:52	04:05	-->	8	0	8	2	-->	18	9	4	16	9	4	-->	1	1	0	0
9	-->	04:30	02:35	-->	8	0	8	2	-->	21	9	6	11	9	1	-->	1	1	0	0
10	-->	04:34	04:02	-->	8	0	8	2	-->	21	11	6	21	9	5	-->	1	1	0	0
	mean:	04:06	03:19	mean:	8,00	0,40	8,00	2,00	mean:	19,50	9,60	4,90	15,30	9,00	2,30	mean:	0,80	0,80	0,00	0,00
	σ:	00:28	00:38	σ:	0,00	0,80	0,00	0,00	σ:	1,80	1,28	1,45	3,41	0,00	1,62	σ:	0,40	0,40	0,00	0,00

Table 17 - "defense 2" scenario measures

		Time			Units Lost			Units Killed					
Simulation		**Our AI**	**Original AI**		**Our AI Zealot**	**Original AI Zealot**		**Our AI**			**Original AI**		
								Firebat	**Marine**	**Goliath**	**Firebat**	**Marine**	**Goliath**
				begin:	20	20	begin:	0	0	0	0	0	0
1	-->	05:14	05:14	-->	20	20	-->	21	15	6	23	16	6
2	-->	05:16	04:00	-->	20	20	-->	23	15	6	19	9	4
3	-->	05:54	04:31	-->	20	20	-->	25	17	8	21	9	6
4	-->	05:26	04:37	-->	20	20	-->	22	17	6	21	12	6
5	-->	05:20	04:37	-->	20	20	-->	22	15	7	21	12	6
6	-->	05:11	04:40	-->	20	20	-->	22	15	7	21	14	6
7	-->	05:15	05:15	-->	20	20	-->	23	16	6	23	15	6
8	-->	05:15	05:11	-->	20	20	-->	21	15	6	21	15	6
9	-->	04:35	05:12	-->	20	20	-->	21	11	6	22	15	6
10	-->	05:17	04:36	-->	20	20	-->	21	15	7	21	12	6
	mean:	05:16	04:47	mean:	20,00	20,00	mean:	22,10	15,10	6,50	21,30	12,90	5,80
	σ:	00:18	00:24	σ:	0,00	0,00	σ:	1,22	1,58	0,67	1,10	2,39	0,60

Table 18 - "defense 3" scenario measures

Simulation		Time			Units Lost			Units Killed	
		Our AI	Original AI		Our AI	Original AI		Our AI	Original AI
				begin:	15	15	begin:	0	0
1	-->	14:05	02:39	-->	15	15	-->	96	28
2	-->	12:11	02:53	-->	15	15	-->	82	33
3	-->	13:37	03:06	-->	15	15	-->	98	36
4	-->	12:00	03:15	-->	15	15	-->	83	34
5	-->	11:57	02:36	-->	15	15	-->	81	29
6	-->	10:11	03:43	-->	15	15	-->	68	46
7	-->	14:48	03:20	-->	15	15	-->	104	36
8	-->	15:08	03:39	-->	15	15	-->	102	38
9	-->	19:52	03:13	-->	15	15	-->	138	35
10	-->	11:08	03:18	-->	15	15	-->	76	42
	mean:	13:30	03:10	mean:	15,00	15,00	mean:	92,80	35,70
	σ:	02:37	00:21	σ:	0,00	0,00	σ:	18,81	5,16

Table 19 - "defense continuo (ofensivo)" scenario measures

Simulation			Time				Units Lost				Units Killed	
			Our AI	Original AI			Our AI	Original AI			Our AI	Original AI
						begin:	15	15		begin:	0	0
1	-->		11:58	06:01	-->		15	15	-->		76	40
2	-->		16:24	06:16	-->		15	15	-->		104	44
3	-->		08:19	06:18	-->		15	15	-->		53	44
4	-->		14:48	05:17	-->		15	15	-->		93	39
5	-->		12:15	06:50	-->		15	15	-->		78	48
6	-->		13:23	06:03	-->		15	15	-->		84	43
7	-->		11:20	06:09	-->		15	15	-->		73	44
8	-->		15:23	06:17	-->		15	15	-->		99	44
9	-->		12:50	07:23	-->		15	15	-->		82	54
10	-->		11:30	06:43	-->		15	15	-->		70	48
		mean:	12:49	06:20		mean:	15,00	15,00		mean:	81,20	44,80
		σ:	02:13	00:32		σ:	0,00	0,00		σ:	14,18	4,09

Table 20 - "defense continuo (defensivo)" scenario measures

B. Questionnaires

Questionário sobre o Jogador

Este questionário refere-se a perguntas mais pessoais sobre videojogos
*Obrigatório

Nº de Jogador *

Qual é a sua idade? *

- De 0 a 12 anos
- De 13 a 20 anos
- De 21 a 30 anos
- De 31 a 40 anos
- De 41 a 50 anos
- Mais de 50 anos

Qual é o seu sexo? *

- Masculino
- Feminino

Com que frequência costuma jogar videojogos? *

- Diariamente
- Dia sim dia não
- Uma vez por semana
- Uma vez por mês
- Raramente
- Nunca

Classifique em termos de importância que dá aos seguintes componentes de um videojogo: *

	1 - Nada importante	2	3	4	5 - Muito importante
Grafismo					
Jogabilidade					
IA (Inteligência Artificial)					
História					
Música e efeitos sonoros					
Interacção com outros jogadores					

Já jogou algum RTS (jogo de estratégia em tempo real)? *

- ◯ Sim
- ◯ Não

Este conjunto de perguntas diz respeito apenas ao tipo de jogo RTS

Qual é a sua experiência nos seguintes jogos: *

	1 - Nunca joguei	2	3	4	5 - Muito experiente
StarCraft	◯	◯	◯	◯	◯
StarCraft 2	◯	◯	◯	◯	◯
Outro RTS	◯	◯	◯	◯	◯

Segundo a sua preferência, classifique a importância das seguintes características que uma IA adversária deve ter *

	1 - Nada Importante	2	3	4	5 - Muito Importante
Dificuldade em vencê-la	◯	◯	◯	◯	◯
Facilidade em vencê-la	◯	◯	◯	◯	◯
Ter um comportamento idêntico ao de um jogador humano	◯	◯	◯	◯	◯
Adaptar o seu comportamento consoante a sua situação no jogo	◯	◯	◯	◯	◯
Ter o mesmo acesso e as mesmas opções que o jogador humano tem	◯	◯	◯	◯	◯

Se estivesse disponível, utilizaria ajuda da IA para o ajudar nos seguintes aspectos? *
por exemplo: utilizar mudanças automáticas num jogo de corridas.

	Sim	Não
Gerir a exploração do mapa	◯	◯
Gerir as construção de unidades	◯	◯
Gerir as construções de edifícios	◯	◯
Gerir a recolha de recursos	◯	◯
Gerir o ataque aos inimgos	◯	◯
Gerir a defesa aos inimigos	◯	◯

Questionário sobre a IA

Tomada em consideração a experiência de jogo que acabou de ter, responda às seguintes questões
*Obrigatório

Nº de Jogador *

A IA adversária ofereceu resistência? *

	1	2	3	4	5	
Nenhuma	○	○	○	○	○	Muita

A IA adversária surpreendeu-o: *

	1	2	3	4	5	
Negativamente	○	○	○	○	○	Positivamente

Classifique as unidades adversárias acerca da sua competência: *

	1	2	3	4	5	
Incompetentes	○	○	○	○	○	Competentes

Classifique as unidades adversárias acerca do conhecimento que cada uma tem sobre os seguintes aspectos: *

	1 - Ignorantes	2	3	4	5 - Conhecedores
Localização dos inimigos da IA	○	○	○	○	○
Localização dos aliados da IA	○	○	○	○	○
Os pontos de saúde das unidades inimigas da IA	○	○	○	○	○
Os pontos de saúde da IA	○	○	○	○	○
As capacidades do inimigo da IA	○	○	○	○	○
As capacidades da IA	○	○	○	○	○

Classifique as unidades adversárias acerca da sua responsabilidade: *

	1	2	3	4	5	
Irresponsáveis	○	○	○	○	○	Responsáveis

Classifique as unidades adversárias acerca da sua inteligência: *

	1	2	3	4	5	
Não Inteligentes	◯	◯	◯	◯	◯	Inteligentes

Classifique as unidades adversárias acerca da sua sensatez: *

	1	2	3	4	5	
Tolas	◯	◯	◯	◯	◯	Sensatas

Classifique as unidades adversárias acerca do seu comportamento individual: *

	1	2	3	4	5	
Recebem ordens de grupo	◯	◯	◯	◯	◯	Cada unidade decide por si

Que aspecto do jogo lhe ofereceu maiores dificuldades? *

- ◯ Os controlos do jogo
- ◯ Pensar numa boa estratégia para ganhar
- ◯ A IA adversária

Achou que a IA adversária fez batota? justifique *

C. Play Testing and Questionnaires results

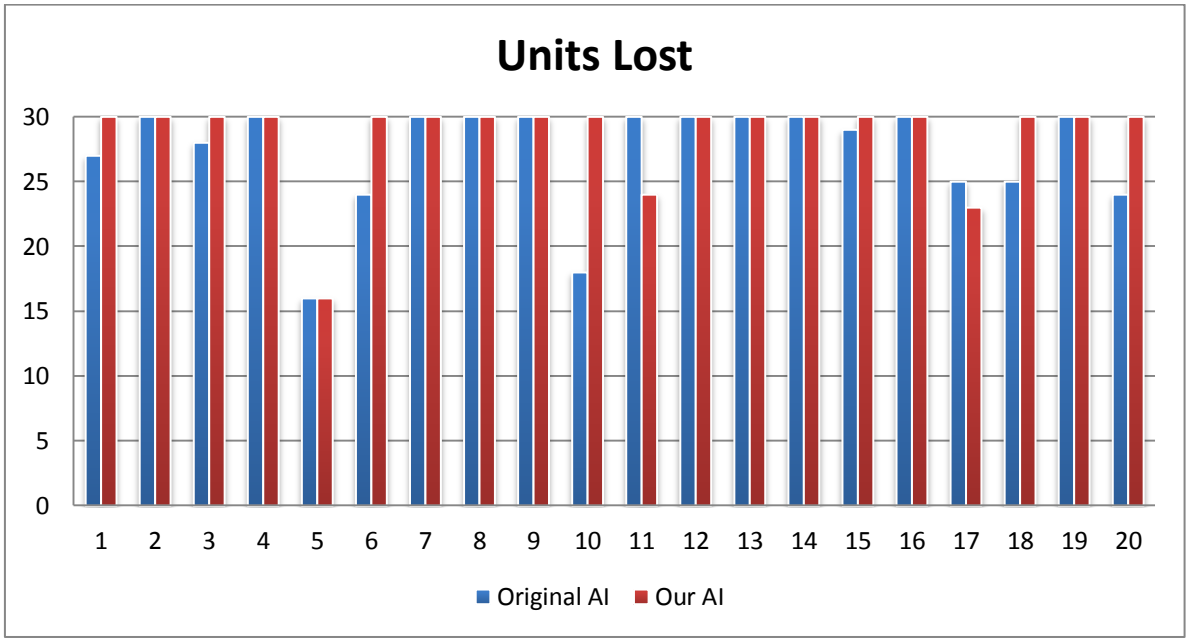

Chart 3 - Testers matches results: Units lost per tester

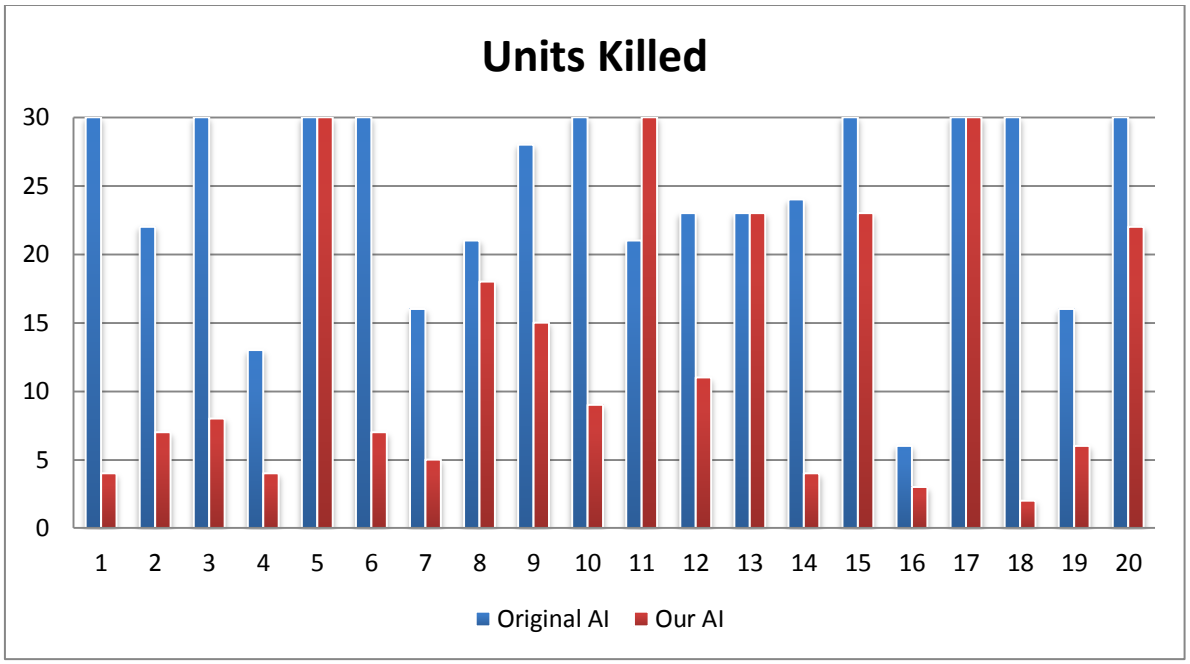

Chart 4 - Testers matches results: Units Killed per tester

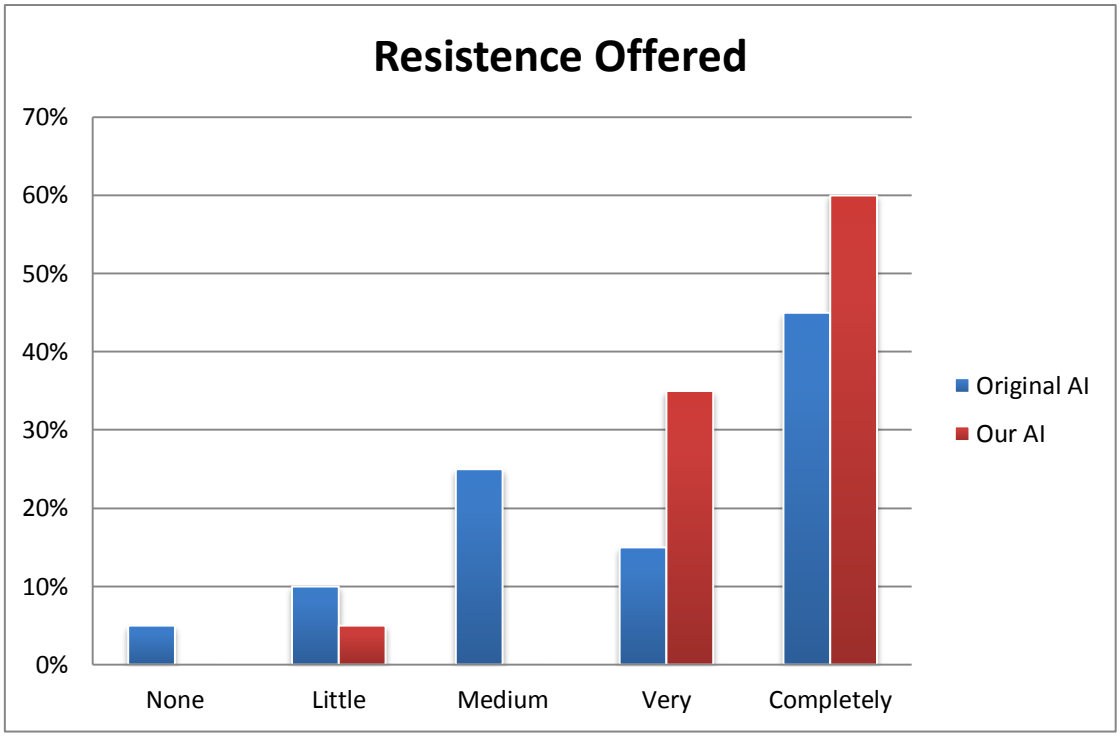

Chart 5 - Comparision of answers about the Resistance of the AI units

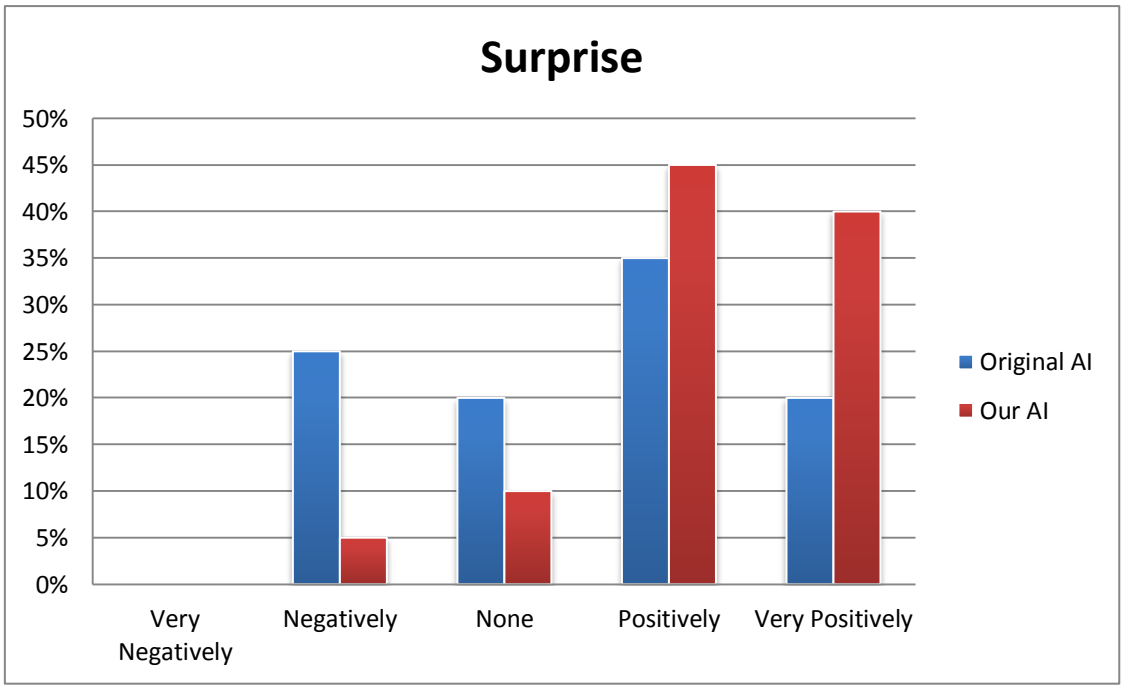

Chart 6 - Comparision of answers about the Surprise effect by the AI units

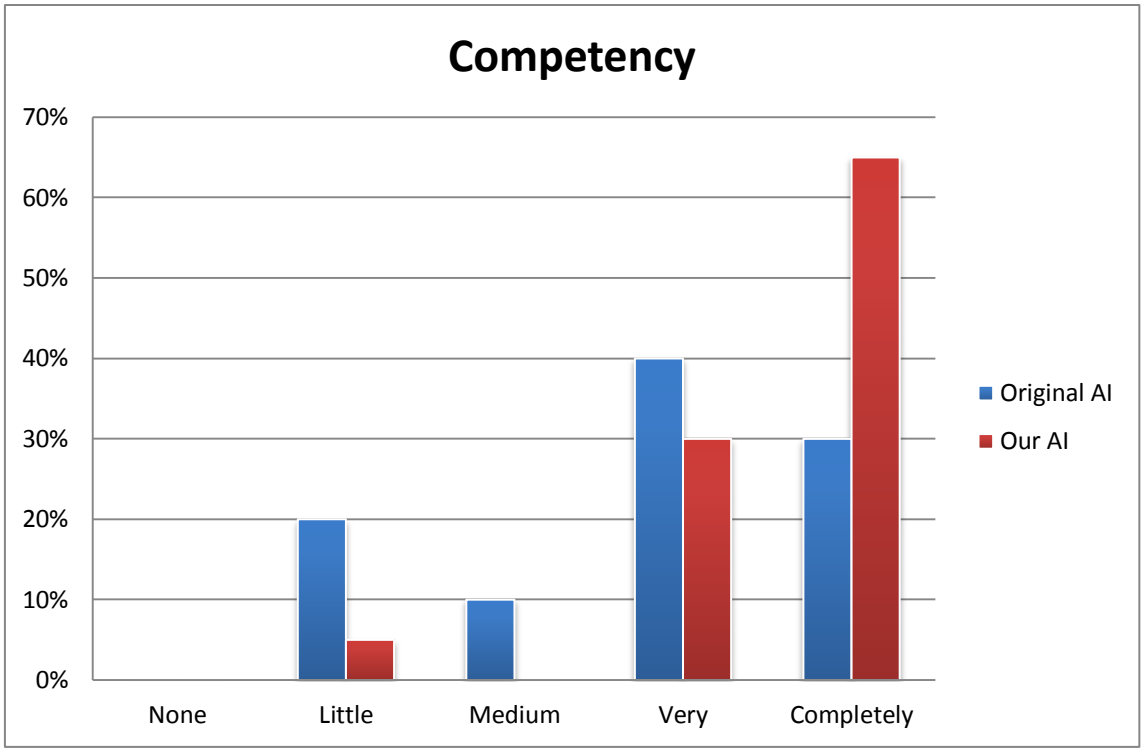

Chart 7 - Comparision of answers about the Competency of the AI units

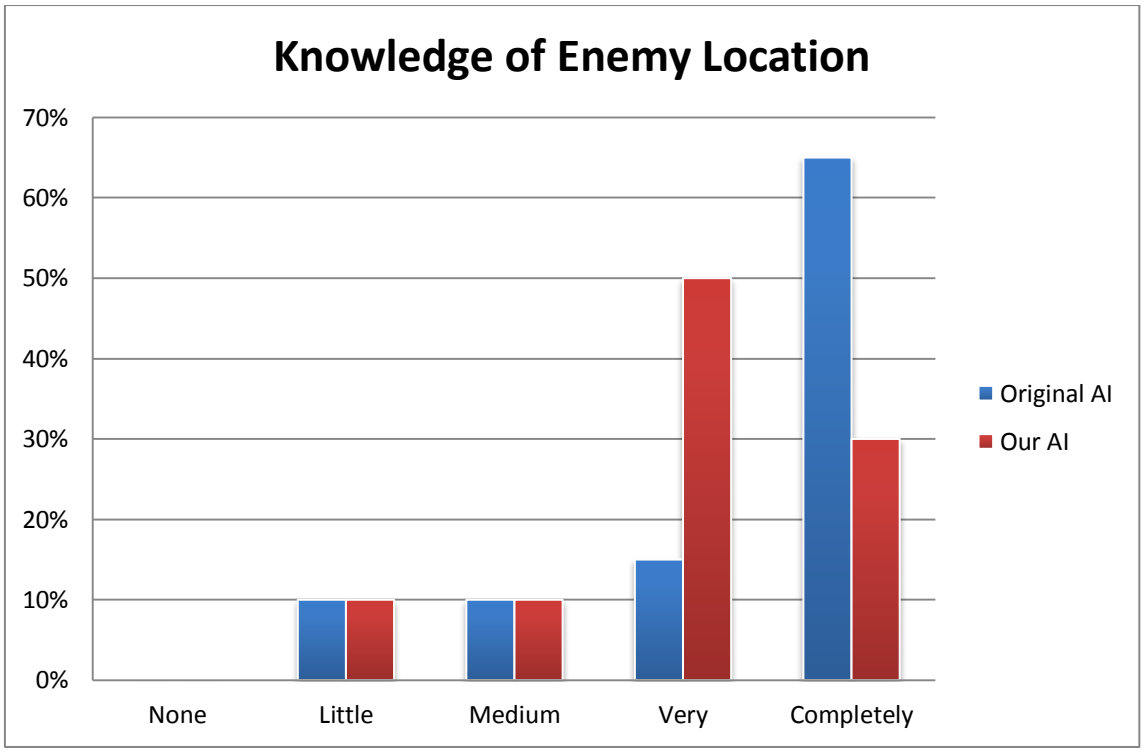

Chart 8 - Comparision of answers on the Knowledge that the AI units have about the Location of their Enemies

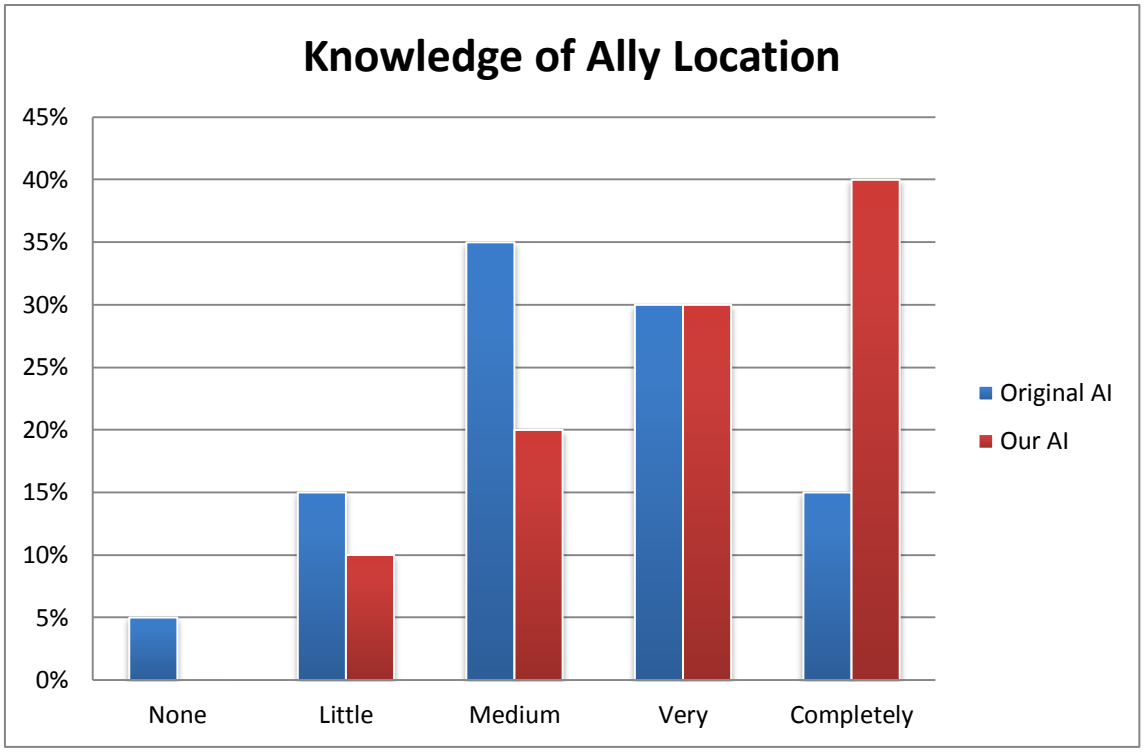

Chart 9 - Comparision of answers on the Knowledge that the AI units have about the Location of their Allies

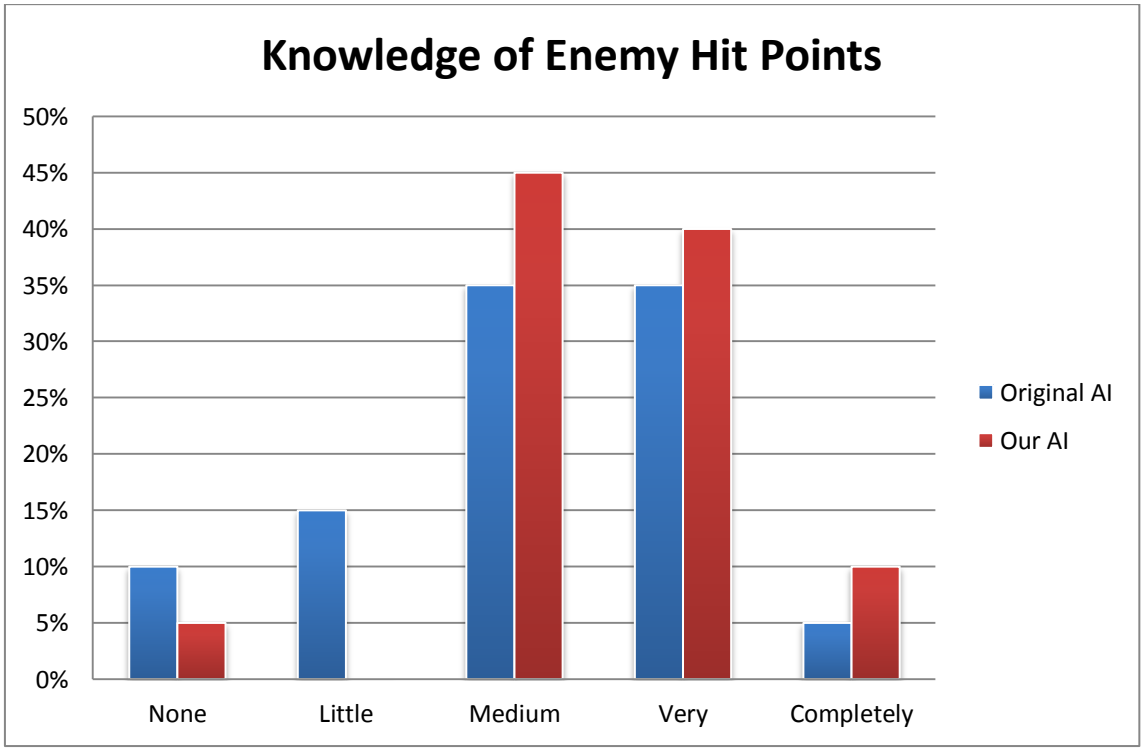

Chart 10 - Comparision of answers on the Knowledge that the AI units have about the Hit Points of their Enemies

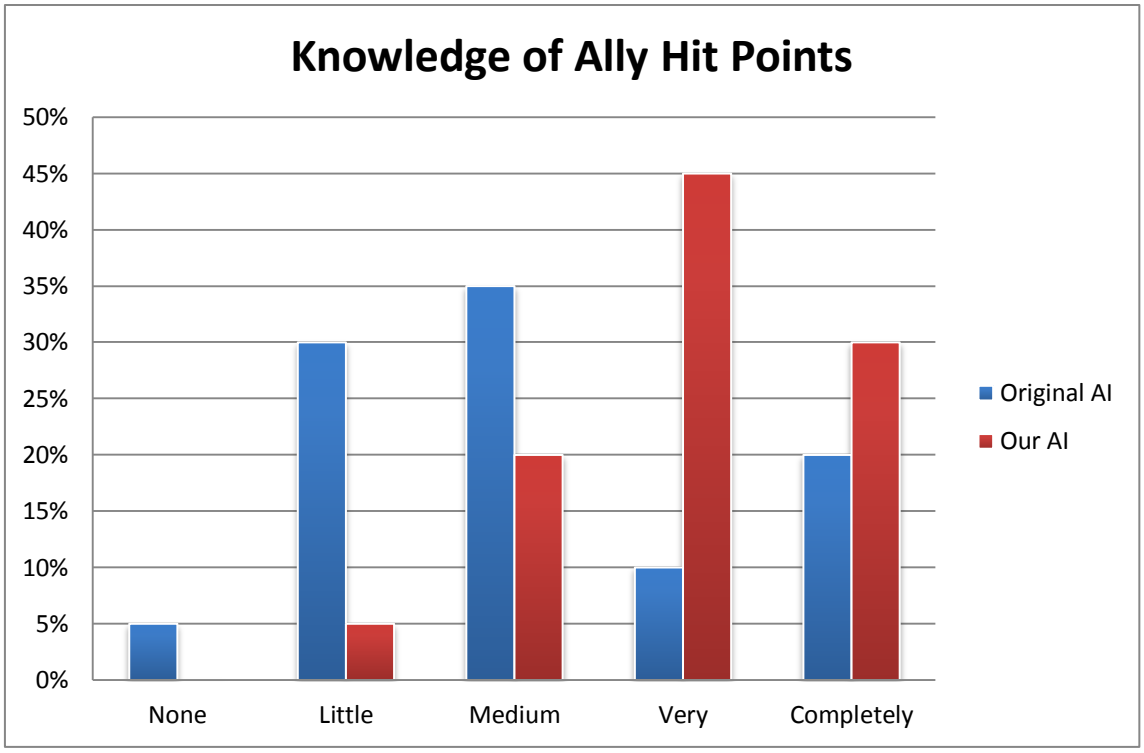

Chart 11 - Comparision of answers on the Knowledge that the AI units have about the Hit Points of their Allies

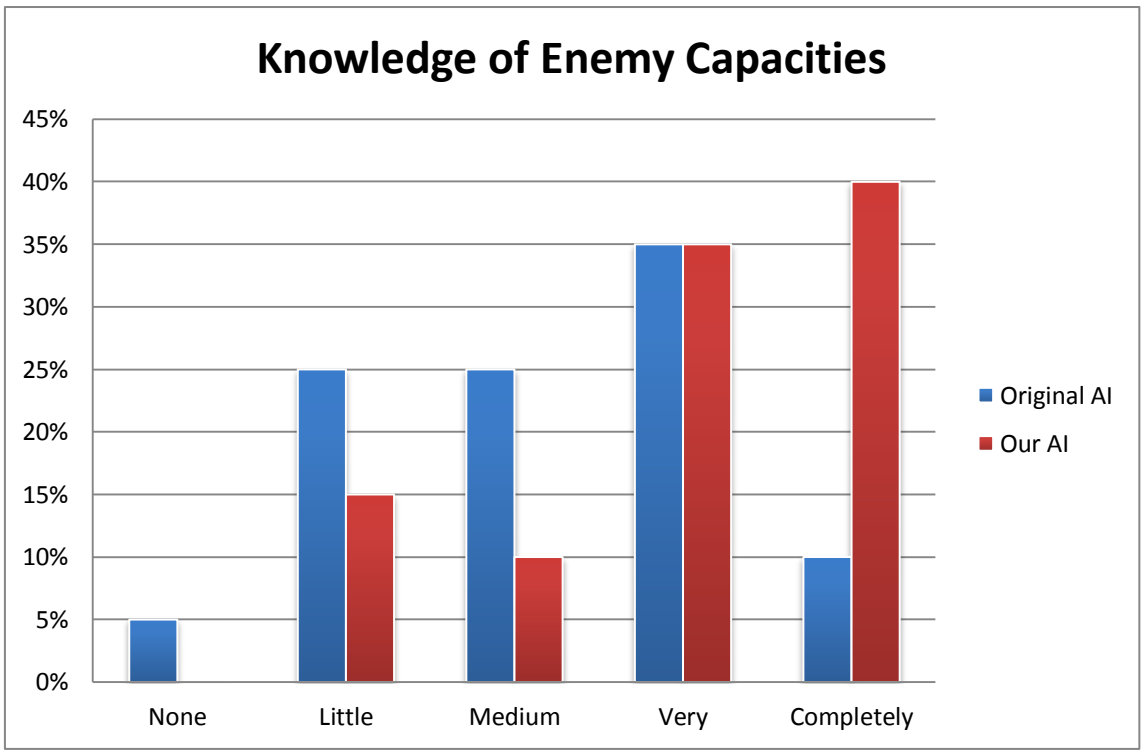

Chart 12 - Comparision of answers on the Knowledge that the AI units have about the Capacities of their Enemies

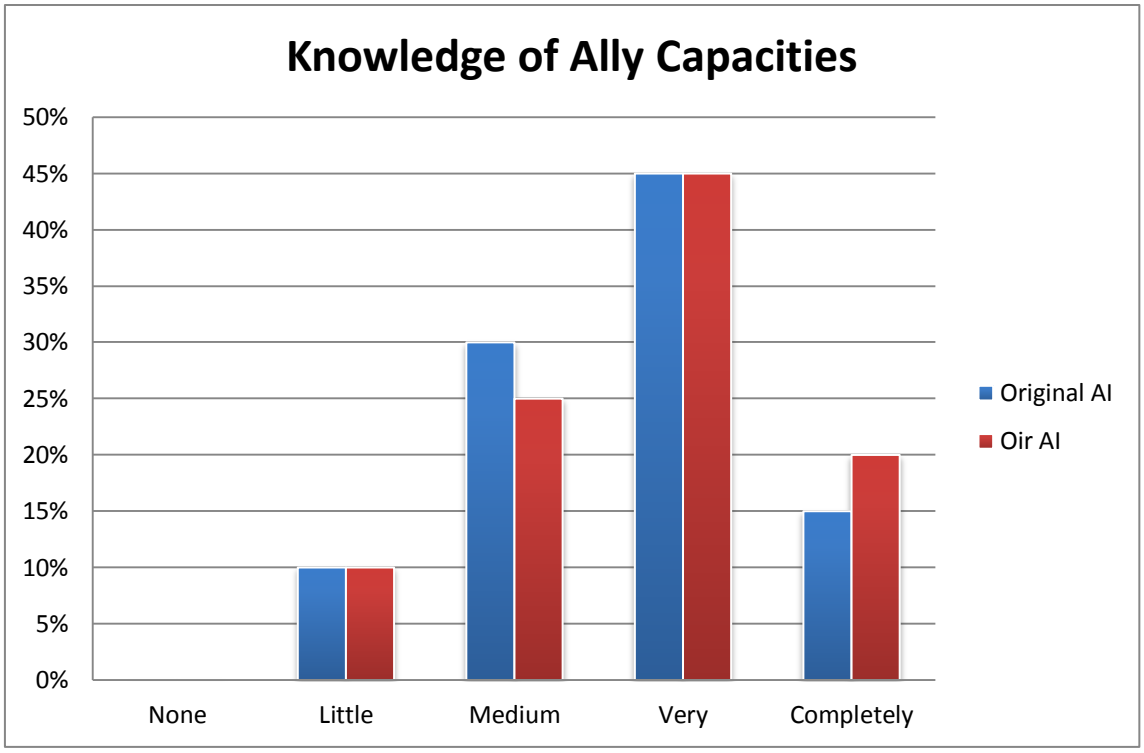

Chart 13 - Comparision of answers on the Knowledge that the AI units have about the Capacities of their Allies

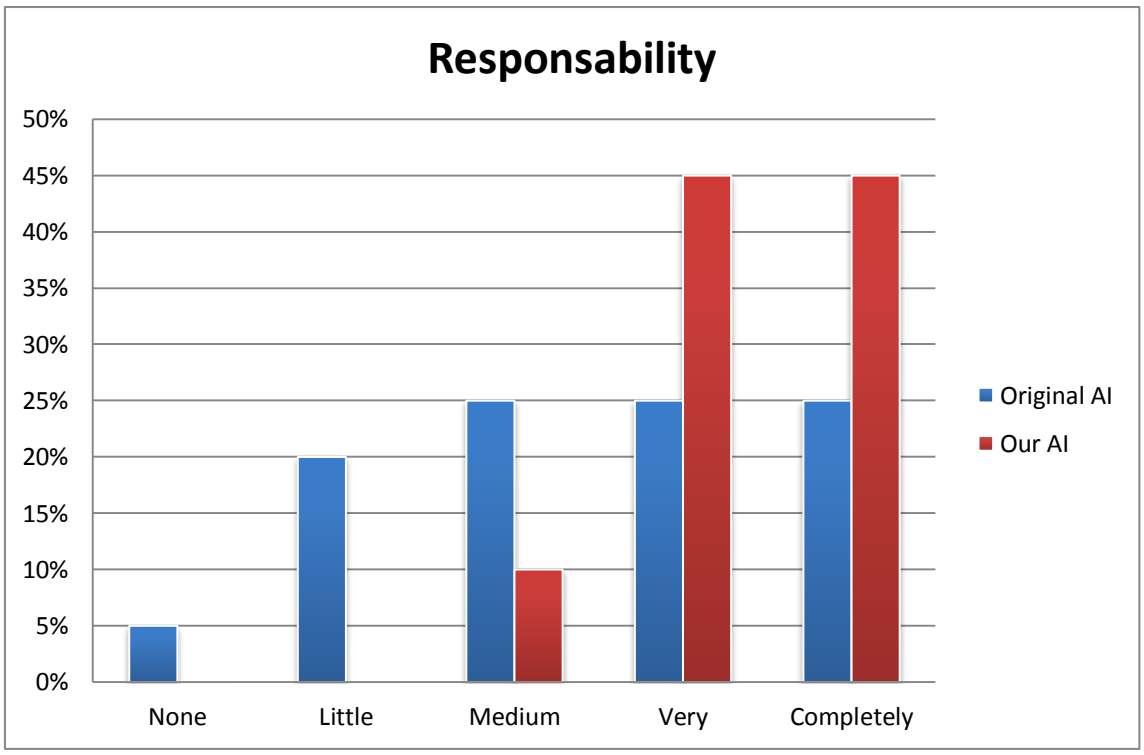

Chart 14 - Comparision of answers about the Responsability of the AI units

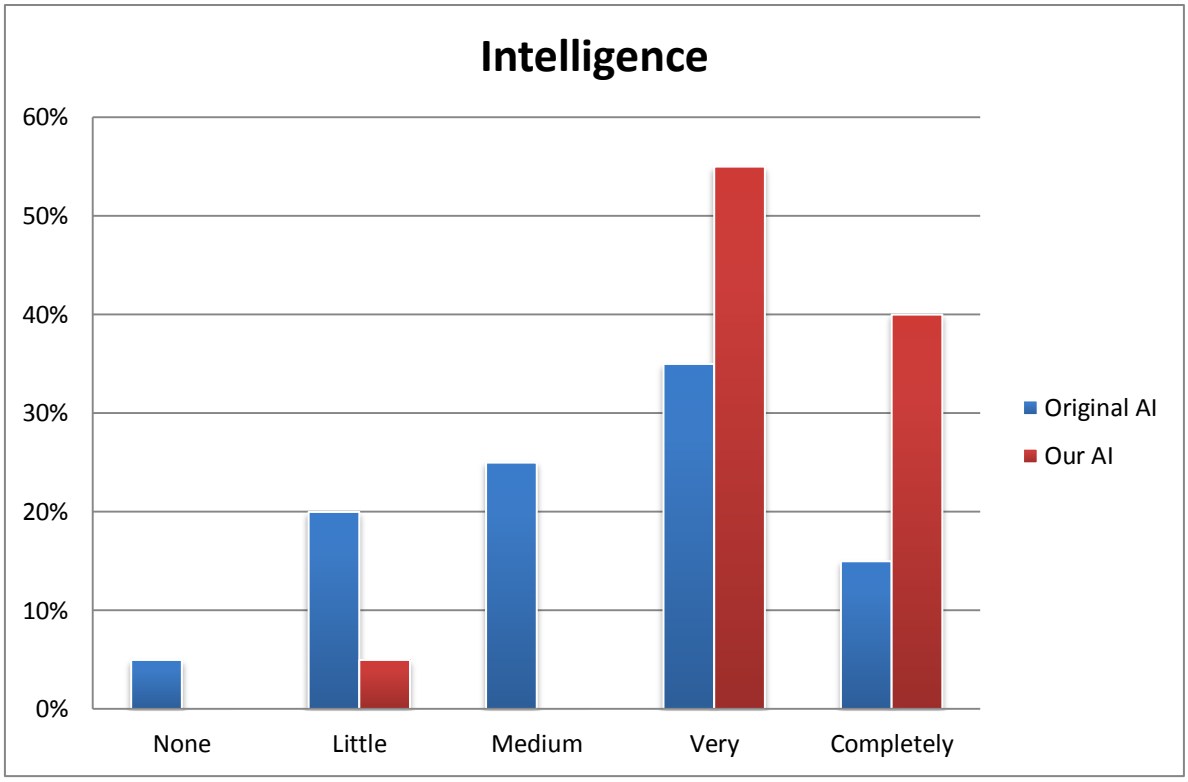

Chart 15 - Comparision of answers about the Intelligence of the AI units

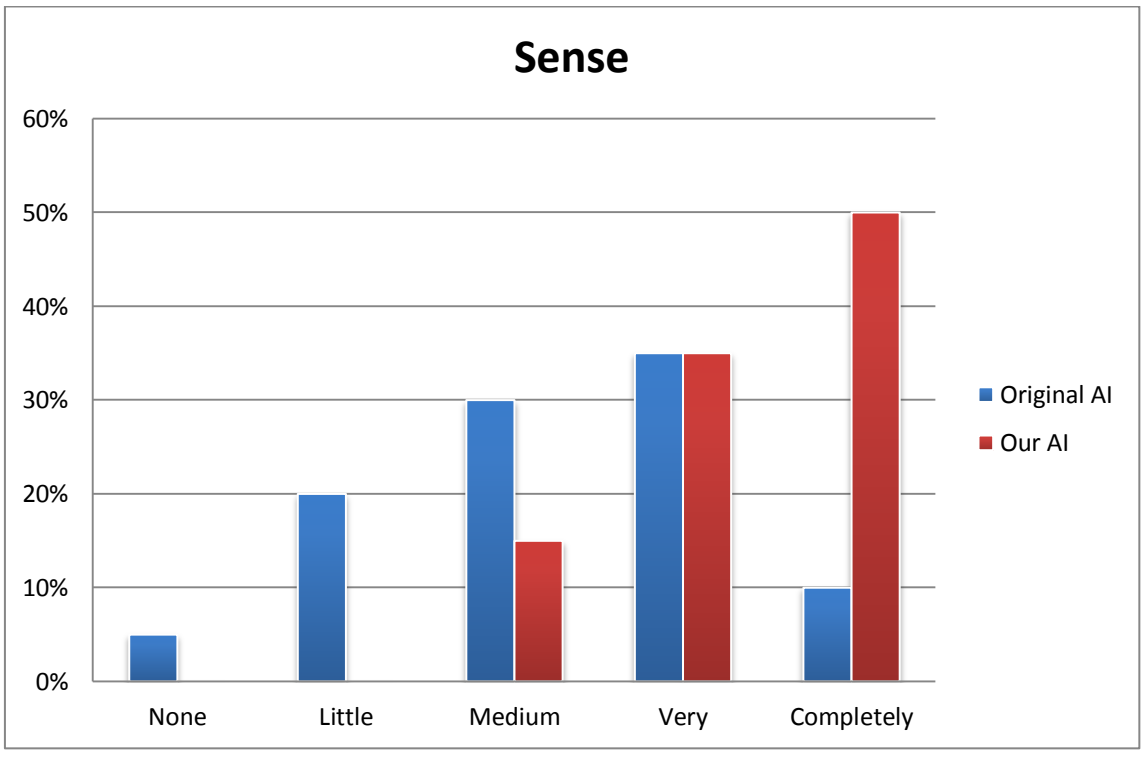

Chart 16 - Comparision of answers about the Sense of the AI units

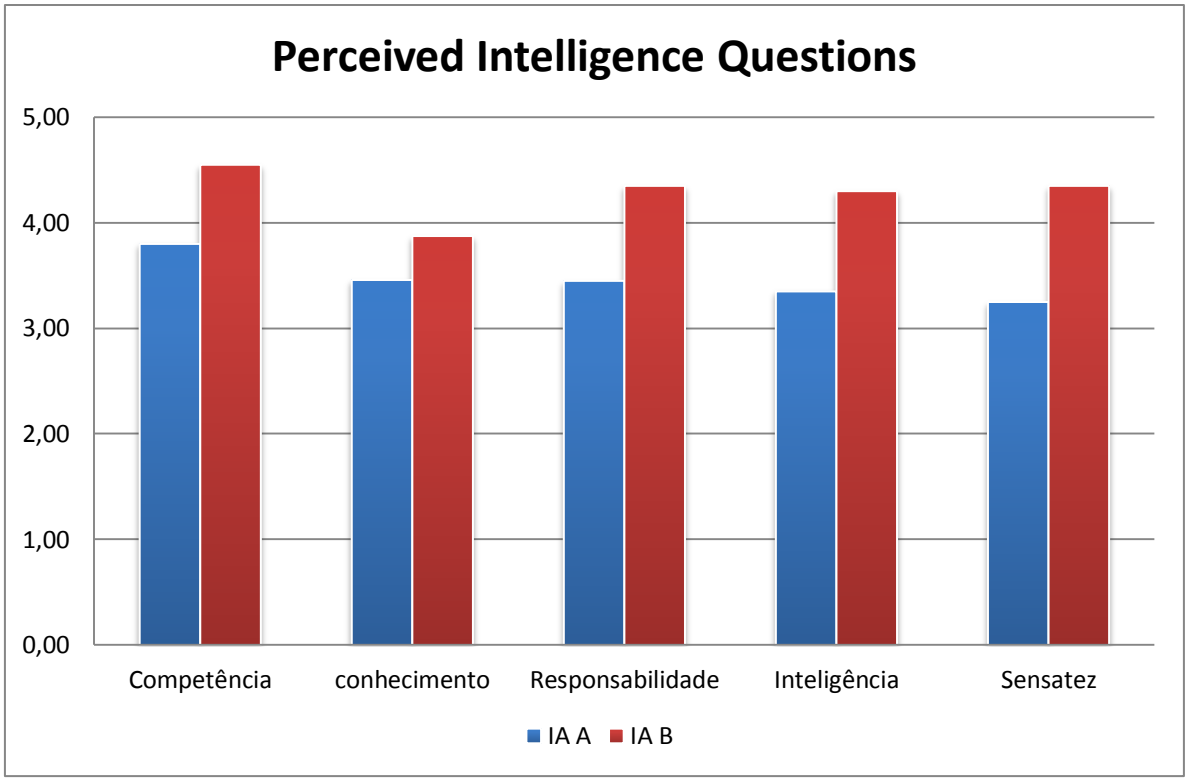

Chart 17 - Comparision of the mean values from the five preceived intelligence answers

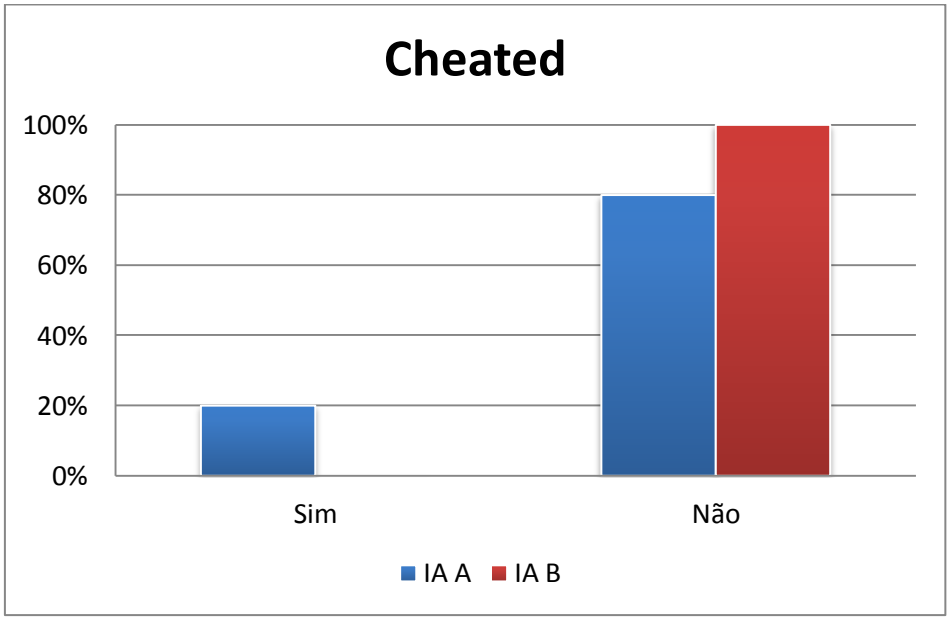

Chart 18 - Comparision of answers about if the AIs cheated

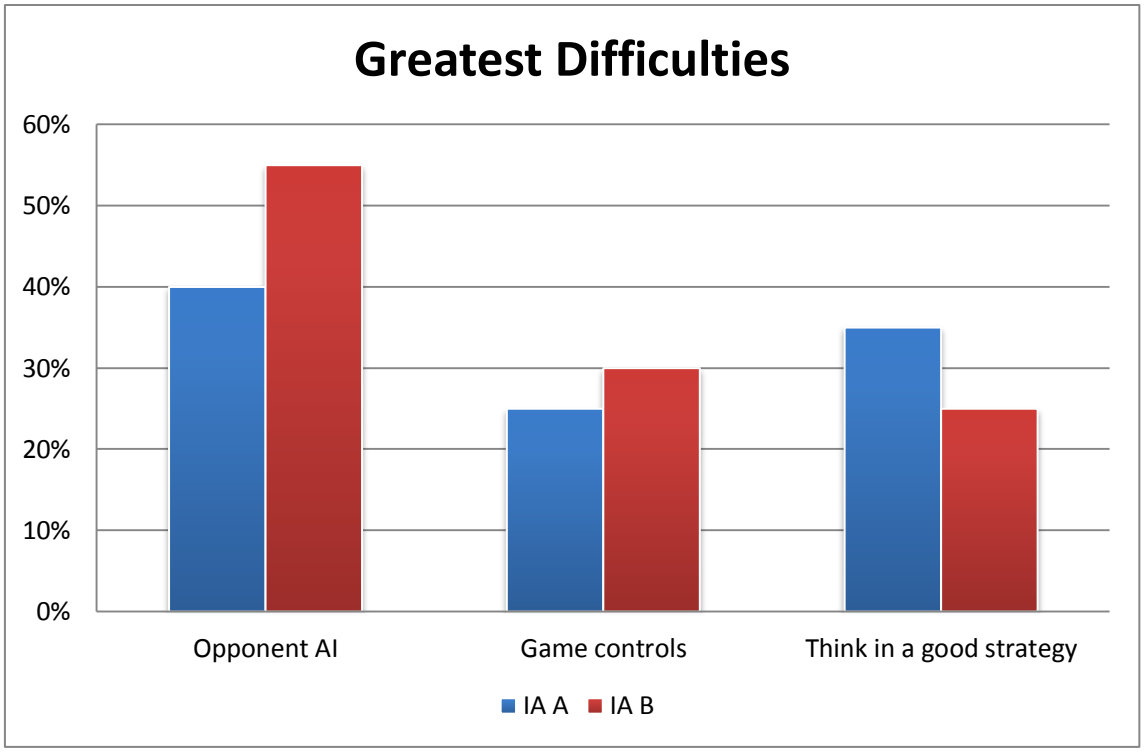

Chart 19 - Comparision of answers about the difficulties encountered

D. Kolmogorov-Smirnov test results

Tests of Normality

		Kolmogorov-Smirnov[a]			Shapiro-Wilk		
		Statistic	df	Sig.	Statistic	df	Sig.
Units Killed	Original	,248	20	,002	,820	20	,002
	Our	,208	20	,023	,855	20	,006
Units Lost	Original	,291	20	,000	,706	20	,000
	Our	,497	20	,000	,443	20	,000
Matches Won	Original	,361	20	,000	,637	20	,000
	Our	,509	20	,000	,433	20	,000

a. Lilliefors Significance Correction

Table 21 - Normality test for units measures

Tests of Normality

		Kolmogorov-Smirnov[a]			Shapiro-Wilk		
		Statistic	df	Sig.	Statistic	df	Sig.
Resistance	Original	,268	20	,001	,830	20	,002
	Our	,344	20	,000	,653	20	,000
Surprise	Original	,225	20	,009	,866	20	,010
	Our	,255	20	,001	,804	20	,001
Competency	Original	,272	20	,000	,828	20	,002
	Our	,373	20	,000	,622	20	,000
Knowledge of Enemy Location	Original	,384	20	,000	,672	20	,000
	Our	,300	20	,000	,817	20	,002
Knowledge of Ally Location	Original	,176	20	,105	,922	20	,108
	Our	,235	20	,005	,837	20	,003
Knowledge of Enemy Hit Points	Original	,213	20	,018	,900	20	,042
	Our	,237	20	,005	,832	20	,003
Knowledge of Ally Hit Points	Original	,233	20	,006	,884	20	,021
	Our	,250	20	,002	,856	20	,007
Knowledge of Enemy Capacities	Original	,215	20	,016	,915	20	,078
	Our	,250	20	,002	,807	20	,001
Knowledge of Ally capacities	Original	,255	20	,001	,881	20	,018
	Our	,258	20	,001	,877	20	,016
Knowledge (single scale)	Original	,134	20	,200*	,962	20	,592
	Our	,140	20	,200*	,969	20	,739
Responsibility	Original	,172	20	,123	,905	20	,052
	Our	,284	20	,000	,773	20	,000
Intelligence	Original	,216	20	,015	,916	20	,083
	Our	,291	20	,000	,705	20	,000
Sense	Original	,208	20	,023	,920	20	,098
	Our	,308	20	,000	,765	20	,000
Individual Behavior	Original	,232	20	,006	,858	20	,007
	Our	,216	20	,015	,893	20	,030
Perceived Intelligence	Original	,162	20	,182	,961	20	,554
	Our	,112	20	,200*	,926	20	,132

*. This is a lower bound of the true significance.

a. Lilliefors Significance Correction

Table 22 - Normality test for Question results