Universitat
Autònoma
de Barcelona

Master in Computer Vision and Artificial Intelligence

Report of the Master Project

Option: Artificial Intelligence

Multi-Reactive Planning for Real-Time Strategy Games

Author: **Alberto Uriarte Pérez**
Advisor: **Santiago Ontañón Villar**

Acknowledgements

I would like to thank my supervisor Dr. Santiago Ontañón for invaluable support and guidance throughout the work. I also would like to thank my friend Albert Puértolas for trying to make me a better C++ programmer and valuable comments and corrections.

ABSTRACT

Real-time strategy games are a challenging scenario for intelligent decision making algorithms and human gameplay simulation. In this thesis we examine the state of the art of useful AI techniques to apply in RTS games such us: steering behaviours, potential fields, decision trees, finite state machines, threat maps, real-time multi agent systems and the blackboard architecture. For that purpose, we designed and implemented an AI bot (Nova) for Starcraft: Brood War, a military science fiction real-time strategy video game developed by Blizzard Entertainment. Nova uses a multi agent system to divide tasks in micro management tasks which controls a large-size army with different abilities (tactical decisions) and macro management tasks, which take decisions about economy, building construction and technology research (strategy decisions).
We want to prove that by performing an immersion in the knowledge of the domain and implementing some of the latest AI techniques, we can improve the built-in AI of the game, beat other bots and be a real challenge for a human expert player.

Keywords: *Real-Time MAS, Working Memory, Balckboard, Threat Map, Potential Fields, FSM, Flocking, Pathfinding, RTS*

Contents

Chapter 1

Introduction

In the past decade the popularity of computer games has increased greatly and the revenue of the gaming industry exceeds now multiple billions of dollars[1]. In most games, the player expects the game AI to be able to work towards long-term goals by performing intelligent decision making. In the genre of real-time strategy (RTS) games, AI agents must reason at several levels of coordination, making intelligent high-level strategic decisions while simultaneously micromanaging units in combat scenarios, acting individually and/or cooperatively. Designing a real-time AI is complex, and this complexity is increased if we want to apply it to non-deterministic domains like RTS games.

In RTS games, playing against other humans is much more popular than playing against the artificial opponents. One of the main reasons is because commercial producers of RTS games have not been able to create games with a challenging enough artificial intelligence without spending a large amounts of resources.

The goal of this thesis is to design a real-time multi agent systems-based AI that can combines several state of the art AI techniques for different aspects of the game. Specifically, we created an AI bot for a popular commercial RTS, which improves the built-in AI and is a real challenge for human players.

The main tasks performed to achieve our goal have been:

- Select an appropriate RTS game as out testbed.
- Analyze all the different techniques and the state of the art in game AI.
- Select the most useful techniques and algorithms for an RTS games.
- Do an extensive study of the existing domain knowledge for the RTS of choice.
- Design a multi agent system that controls the different aspects of the game.
- Implement the designed AI, called Nova.

- Evaluate the performance of our AI by running a set of experiments against the built-in AI, and participating on different international competitions of RTS AI bots.

The rest of this monograph is organized as follows. Section 2 briefly summarizes the history of the Real Time Strategy genre, and gives the reader a context of our research domain and the main mechanics of this type of games. We also remark the challenges in this scenario for AI research. In Section 3 we present a literature review of the research related to game AI techniques that can be applied in RTS games. This state of the art shows how to solve some problems present in all modern RTS games. Then, in Section 4 we present the design of our real-time multi agent system, Nova, and how to assemble most of the techniques seen on the previous section. Furthermore, we explain how to exploit the well known strategies used by professional players. Then, in Section 5 we present our experiments and the results of our bot in an international competition for RTS AI bots. We finish the master thesis with conclusions and how this research can serve as the starting point for future work.

Chapter 2

Introduction to Real Time Strategy Games

Real Time Strategy (RTS) is a video game genre where a player plans a series of actions against one ore more opponents to achieve victory, and where actions are executed in real-time. This means that the action in the game is continuous, so the player must think and act quickly in a constantly changing world.

In most strategy video games, the player is given a godlike view of the game world and takes control of all their game units. These games challenge the player's ability to explore, manage an economy and command an army. Additionally, the player's attention is divided into many simultaneous tasks, thus, RTS games are considered hard to master.

In a RTS game the opponents can be other humans or computer AIs, and it is in the latter where we will focus our interest: the process of developing a computer AI player.

2.1 History

The concept of Real Time Strategy game is generally traced back to Dune 2, released in 1992 by Westwood Studios and based on the Dune universe from the series of science fiction novels by Frank Herber. Certainly, this is not the first RTS game, but Dune 2 defined many of the major elements that now make up the RTS genre. Figure 2.1 shows the game with 2D graphics and a point-and-click graphical user interface.

Figure 2.1: Dune 2 game screenshot

It was the first to introduce a true real-time aspect in a moment where all strategy games implemented a turn-based system like Civilization. As ign.com described

Dune II put a ticking clock into turn-based action and changed strategy gaming forever[2].

Some other key features introduced by Dune 2 are the ability to control the construction of a base, the need to gather resources and the inclusion of three races, each with specific and unique features.

2.2 Gameplay

There are some gameplay concepts present in all RTS games. Here we will describe the main characteristics most RTS games share.

- **World representation**. The game plays in a two-dimensional grid of tiles called map. We do not have a fixed map size, and depending on the game we can find several different map sizes. Another important concept of RTS games is the Fog of War. Unexplored regions of the map for a given player are hidden behind darkness. Each unit has a range of vision and a zone is considered explored if it is in range of vision for any of the player's units. Explored regions become fully visible and they do not go back to darkness again even if they become out sight of friendly units, instead, a lighter fog covers that area, allowing the player to see the map, but not any enemy units present in that area. Figure 2.2 illustrates the Fog of War concept.

Figure 2.2: Fog of War

- **Obtaining resources**. In this type of games we need to harvest on-map resources in order to purchase units. We can gather one or more types of resources (like gold, minerals, gas, wood, etc.).
- **Building bases**. Another typical concept is the possibility of building a mini city where each building has special functions and/or are needed to train units for the army.
- **Researching technologies**. We have the option to improve our units doing research. Often we will find a technology tree with dependencies between each research line. Performing research typically requires a long time or a large amount of resources, but it gives access to important upgrades for the units, or even to new unit types.
- **Producing units**. The last concept is the ability of produce different type of units with different unique characteristics. Normally we can choose between several races and each race has different types of units. This variety adds a lot of design complexity, since each race must be balanced and all units of the game must follow the pattern "rock, papers, scissors" to have a natural equilibrium (i.e. there should not be any dominant type of unit, in order for interesting strategies to arise).

2.3 Challenges for AI

Over many years, game AI research has been focused on turn-based games like Chess. RTS games, however, offer a more complex scenario with many different problems, where an elegant solution is needed. One of the biggest challenges is how to deal with a non-deterministic world, since we do not know what our opponent is doing, in a real-time context. RTS games pose several interesting questions to the research community:

- How to build a multi-agent system capable of reasoning and cooperating in a real-time environment. Building a multi-agent system can be a a complex task if we want each agent to take

decisions and perform tasks as soon as possible within a time constraint. Executing tasks in real-time, means that we do not have the time we want to search the best decision for every case, we must respond quickly to any external stimulus. And all of this in a cooperative way, so a clear and efficient communication is very important.

- Concurrent and adversarial planning under uncertainty. In order to win the game, each agent has to take the right decision in a high level plan. This kind of planning becomes hard when we have uncertainty about our opponent, and tasks like learning and opponent modeling can help.
- Spatial and temporal reasoning. Not all the knowledge is generated on-line, we can analyse the map before the game starts and make tactical decisions off-line. But during a game, we have to adapt some decisions under situations where the questions 'when' and 'where' are the key to win.

2.4 Case of Study: Starcraft

We did our experiments on the popular military science fiction RTS game Starcraft: Brood War, released in 1998 by Blizzard Entertainment. Starcraft is set in a science-fiction based world where the player must choose one of the three races: Terran, Protoss or Zerg. The good work done by the people of Blizzard makes this game one of the most extremely well-balanced RTS games ever created.

- Terrans, human exiled from Earth, provide units that are versatile and flexible giving a balanced option between Protoss and Zergs.
- Protoss units have lengthy and expensive manufacturing process, but they are strong and resistant. These conditions makes players follow a strategy of quality over quantity.
- Zergs, the insectoid race, units are cheap and weak. They can be produced fast, encouraging players to overwhelm their opponents with sheer numbers.

With more than 9 millions of copies sold worldwide in early 2008, it is the best-selling RTS game ever. Stracraft is extremely popular in South Korea, where 4.5 millions of copies have been sold only in this country. The influence of Starcraft is so big that in South Korea are professional leagues with sponsored teams and professional gamers wearing trendy F1-style leather sportswear. The average salary for a 2010 Starcraft player is $60,000, where the average South Korean earns $16,296[3]. Korean players are famous for their high speed control of the keyboard and the mouse, achieving an average APM (actions per minute) of about 360 through a single game, that is an incredible six unique orders every second.

Chapter 3

State of the Art on RTS AI

In this section we will review the current state of the art on RTS game AI. Since there is a huge number of different techniques available on game AI today, we only cover those relevant to RTS games.

3.1 Terrain Analysis

Terrain analysis supplies the AI with chunks of abstract information about the map in order to help in making decisions. Terrain analysis can provide automatic detection of regions, choke points and other areas of interest. This analysis is usually performed off-line, in order to save processing time during gameplay.

The final goal of terrain analysis is having all the useful tactical information of the map. To do this we can use different algorithms, one of the most used, coming from computer vision, being *skeletons by influence zones*, also known as *Voronoi diagrams*[4]. Through this kind of diagrams we will be able to generate a graph with the centroid of the regions and the choke points of each region. Figure 3.1 illustrates the final graph of regions and choke points in a Starcraft map. Red dots are the centroid of the regions, blue dots are the center of a choke point and green lines represent the connected regions; the different tonalities of browns are the different levels of the map.

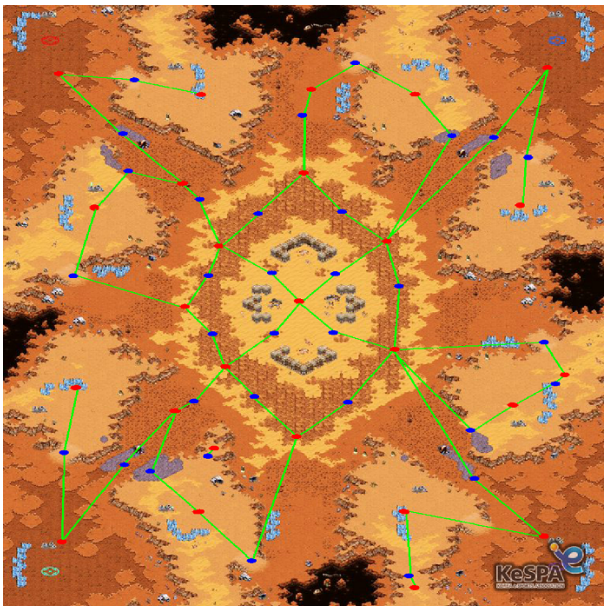

Figure 3.1: Starcraft map with regions and choke points

Another common technique is the utilization of an influence map to detect and mark relevant zones of the map as resources location. More info about influence maps on section 3.4.1

3.2 Unit control

Unit control involves the ability to move units in a way that makes them more effective individually, whether in dealing damage, avoiding damage or achieving effective movement. In RTS games often we have to deal with a large number of units. For an optimum motion, they need to coordinate their movement behavior while ensuring safe navigation in complex terrain. In this context, units will avoid obstacles and other units.

3.2.1 Pathfinding

The most common pathfinding algorithm is A*, which can be a great improvement over Dijkstra algorithm with the right heuristic. The key idea of A* is the use of a best-first search algorithm to find the least-cost path given an initial node and a goal node. The path cost between nodes is the distance plus an heuristic function. Figure 3.2 illustrates the node cost and the final path to go from green tile to red tile. Figure 3.3 represent the same exercise in the Starcraft domain.

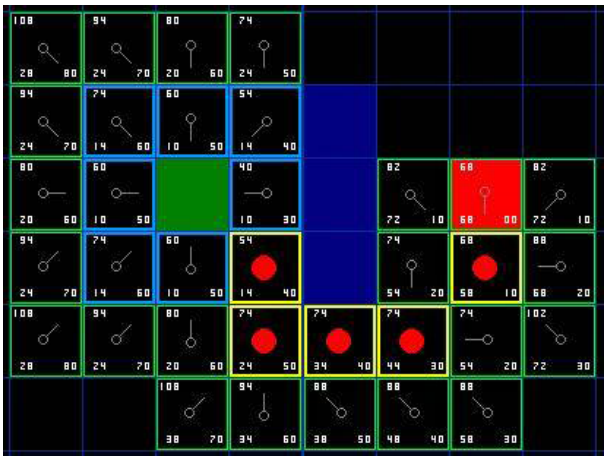

Figure 3.2: A* example

Figure 3.3: A* in Starcraft

But the big problem of this algorithm is the time and memory consumption, a show-stopper in a real-time environment.

To solve this the search space needs to be reduced as much as possible. One common technique is reducing the tile map search into a navigation graph map. To do this some polygonal triangulations are calculated and abstracted on a grid-based map and this is the TRA* (Triangulation Reduction A*) algorithm[5]. Figure 3.4 represent a a navigation mesh and a navigation graph with a path connection points A and B using TRA*.

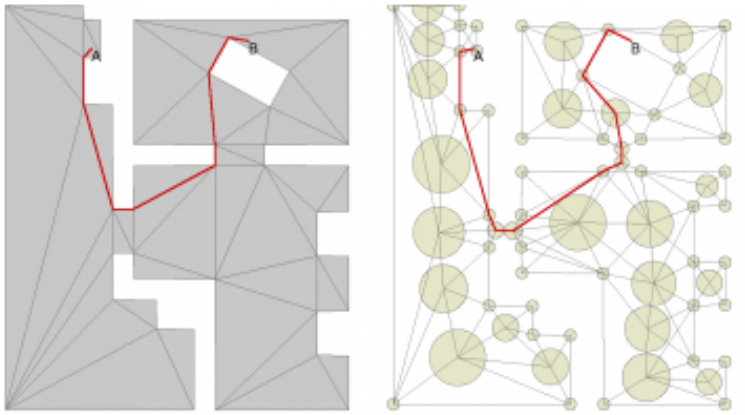

Figure 3.4: Navigation mesh and navigation graph examples

3.2.2 Flocking

Craig Reynols introduced algorithmic steering behaviors to simulate the aggregate motion of swarms using procedural animation[6]. His model describes how a system of multiple agents , each steering according to simple local rules, can produce the collective movement found in flocks of birds. That model consists of three simple rules for each agent: Separation (keep a distance to neighboring flockmates), Alignment (adjust the facing direction and speed to match with local flockmates), and Cohesion (steer towards the average position of neighboring flockmates). Figure 3.5 represents each of these rules.

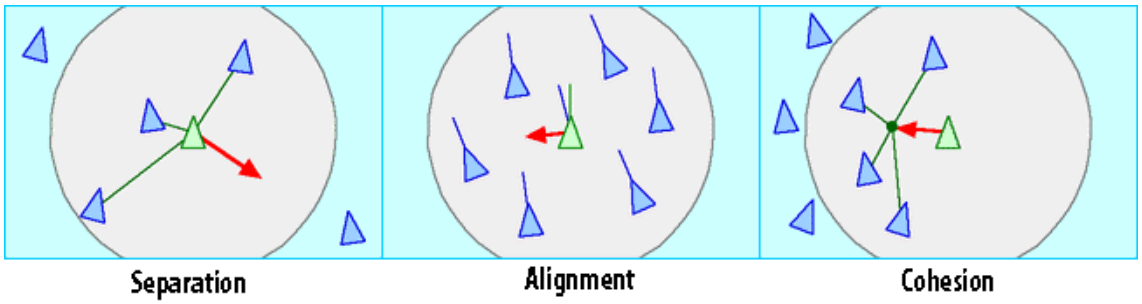

Figure 3.5: Flocking rules examples

With this behaviours we are able to simulate intelligent squad movements and do tactical moves to flank the opponent in a RTS game[7], improving the survival of our units.

3.2.3 Reciprocal velocity obstacles

A good alternative to Flocking is using Reciprocal Velocity Obstacles (RVO)[8], since it is easier to control the behaviour and they guarantee to prevent collisions. The key idea of RVO is to share the work

required to avoid a collision among participating agents, this means that we are implicitly assuming that the other agents will make a similar collision avoidance reasoning. For example, imagine we have two agents A and B heading to each other, then the agent A will do only half of the effort to avoid a collision to B, and assumes that agent B will take care of the other half. Figure 3.6 shows agents avoiding collision with a 50% of effort for each one.

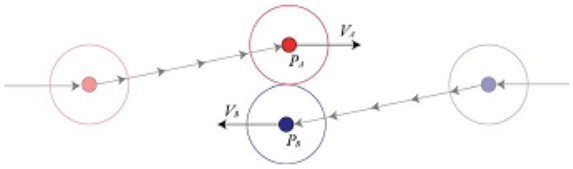

Figure 3.6: RVO example of 50% of effort for each agent

With this algorithm we can move a large squad crossing another squad and avoid all possible collisions between units. This type of techniques are already applied in commercial RTS games like Starcraft 2[9] and Supreme Commander 2[10] with great success.

3.2.4 Potential fields

Basically, Potential Fields work like a charged particle moving through a magnetic field. The idea is to put a charge at an interesting position in the game world and let the charge generate a field that gradually fades to zero with distance. Each force field either has a positive (attractive) or a negative (repulsive) electric charge. That is a reactive approach, where the agent will constantly have to remain extremely vigilant to new changes in the environment and therefore the entire route from start to the goal is not planned ahead.

The process of tuning and optimizing the parameters of potential field charges can be time reduced applying learning algorithms such as Q-Learning[11]. Another common issue we have to deal with when using potential fields is the local optima problem that can stuck the unit away from the goal. Some examples of the utilities of potential files in RTS games are avoiding obstacles (navigation), avoiding opponent fire, or staying at maximum shooting distance[12].

3.3 Decision Making

When a marine is attacked by an enemy unit, it can choose to retreat or retaliate. A human player or some sort of AI agent, must make this decision. In order to take the best action we have to consider all the information in the current game state.

The following sections present some of the most common decision making techniques in game AI.

3.3.1 Decision Trees

A decision tree is a decision support tool that uses a tree-like graph where a root is a decision point and leafs nodes describe the actions to be carried out. To decide which action to perform, we start at the root and follow the path led by each decision point's answer. When we ultimately arrive at a leaf node, we have found the desired action to perform.

Some of the advantages of the decision trees are: simple to understand and interpret acting like a white box where we can explain why a decision is taken, and they are compatible with other decision techniques.

For complex decisions we have many algorithms to generate an optimum decision tree like ID3 or C4.5, depends on the problem we will use one or another. We call a decision tree which needs the least information to take the right decision an optimum decision tree.

3.3.2 Finite State Machines

A FSM is a model which is composed of a finite number of states associated to transitions. While an agent is in a state, it can work on a set of inputs and either perform some actions defined on the state, or transition to a new state. The idea behind finite state machines is to decompose an object's behavior into easily manageable "chunks" or states, such as "attacking an enemy", "gathering resources" or "repairing" and establish the conditions that trigger transitions between them.

Usually, the state transition occurs when a set of specific input conditions, called triggers, are met. This is like flicking a switch for an ordinary light bulb. If the light is on when the switch is flicked, it allows electricity to flow through the system (input), the light is turned on (action) and the current state is changed to "On" (state transition). Figure 3.7 illustrates this example of a light behaviour.

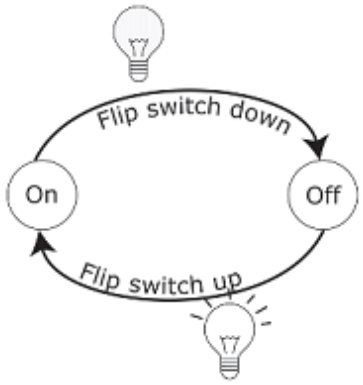

Figure 3.7: FSM of a light behaviour

3.4 Opponent modelling

Opponent modelling is the process of keeping track of what the enemy is doing during a game in order to estimate the probability of it using a specific strategy. The key point here is to predict what units the enemy will build, when he will build them, when and where he will attack and so on. This will give us information about what kind of opponent it is allowing us to adapt our strategy and select the best counter.

Science we are dealing with imperfect information, the best solution is to build a Bayesian model which can deal perfectly with the uncertainty. The difficult part is building that model. Some approaches use traces of game logs to analyze the state of the world, but those traces have all the information of the game, so we have to add some noise to emulate the real use of the model in a non-deterministic world.

3.4.1 Threat map

A Threat Map or Influence Map is a way to store relevant information about the opponent and keep track of the conflict areas or the secure ones. Statical threat maps can be designed where the threat is always "real" or concepts like heat map over time can be applied to make predictions.

Another decision we must take designing this type of maps is the spatial partition. A map with a high definition can be computationally non-viable on real-time environments, but a low definitions can fall in a poor information precision making it useless for taking decisions. Figure 3.8 shows two ways of spatial partition on threat maps, the left part is a 2D tile partition while the right part is an area graph partition.

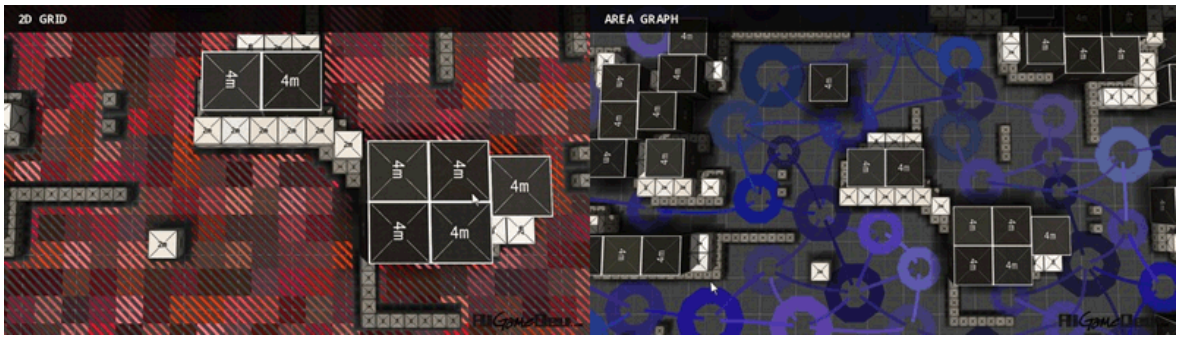

Figure 3.8: Different ways of spatial partition on threat maps

3.5 Real-time multi agent architecture

There is a lot of literature about multi agent system, but we have the constrain of a real-time environment, therefore we will focus in systems capable to deal with that. In such environments, agents need to act autonomously while still working towards a common goal. These agents require real-time

responses and communication among agents must be controlled to avoid performance penalties. So an efficient communication layer is our priority.

3.5.1 Working memory

The working memory is a shared place with a collection of facts. When a sensor's agent discovers something about the world it deposits a fact in the working memory. Then if an agent is looking for information about the world, it searches all valid working memory facts looking for the best fact which is used to help in making a decision or execute an action. The benefit of using the working memory is that it allows the storage of results (caching) so sensors may not need to be run every frame; they can be delayed (distributed processing).

A fact can have a degree of belief that represents the level of confidence that something exists in the world. For instance, if an enemy unit is in the agent's range of view then a working memory fact of type Enemy would be created with a belief of 100%. If the enemy was obstructed or not seen in some time the belief of the fact would be lower.

3.5.2 Blackboard

A blackboard helps us in the process of agent communication, instead of a complex message passing between all agents, a blackboard centralizes all the useful information that agents want to exchange. So, in a blackboard anyone can read information at any time and all agents can write their intentions or desires. This definition is that of dynamic blackboards, where the information changes at run-time, but static blackboards can also be build.

In conclusion, the main advantage of the blackboard is that it provides modularity; the different agents do not depend on each other, instead they just exchange the data via the blackboard. This reduces the level of coupling between the different behaviors, which makes the system easy to work with. Now, when an agent needs querying for a certain information, it does not need to ask several agents, but just search directly the answer in the blackboard making the communication process more clean and sustainable in a real-time process.

Chapter 4

Implementing a Starcraft Bot: Nova

In this section we will describe how Nova has been designed. First we will give a global view of the architecture and agents we used. Second we will describe the complexity of each agent in more depth.

4.1 How to interact with Starcraft

Since Starcraft is a commercial game, we do not have access to the source code in order to modify or extend it. Fortunately there is a solution called Brood War Application Programming Interface (BWAPI) that makes the dirty job for us. BWAPI can be loaded from the Chaos Launcher application which launches a Starcraft instance and injects the BWAPI into the game's process. Then we can extend BWAPI with our own DLL to interact directly to the Starcraft game. Figure 4.1 illustrates this process.

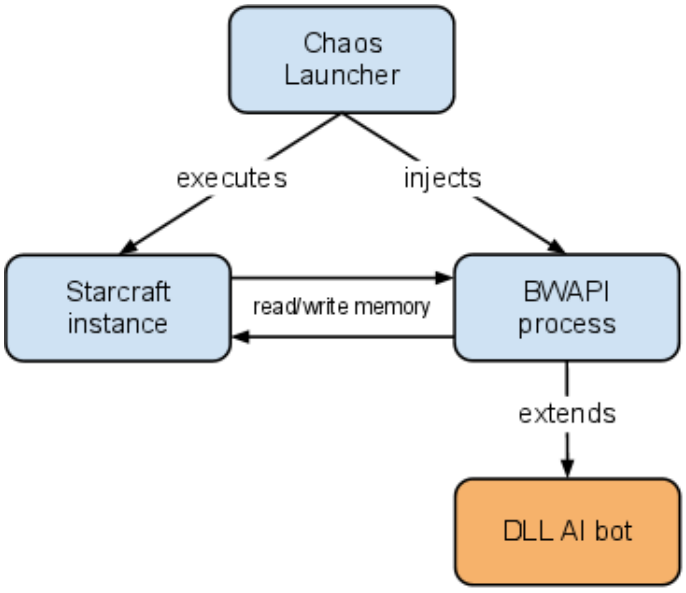

Figure 4.1: How BWAPI works

BWAPI also provides a modified Chaos Launcher which can execute multiple instances of Starcraft (with BWAPI injected), which is perfect for test different bots in the same computer.

4.2 Architecture overview

The main tasks in Starcraft are micro management tasks and macro management tasks, and in order to make the right decision the bot needs the maximum information possible about the enemy. Having this in mind, we designed Nova as a multi-agent system with two main types of agents: agents and managers.

- **Managers**: managers are in charge of persistent tasks, like build-order, for which we only need one instance of each type manager type in Nova.
- **Agents**: regular agents are in charge of military units and squads, and new agents can be spawned or killer when Nova creates or loses military units.

Each agent has a specific knowledge of the domain and they are capable to do atomic tasks. Some agents can make decisions about higher level concepts, while others are lower level. One of the big advantages of having a multi-agent system is the capacity to process different problems in parallel. In a RTS game, unlike in a turn-based game like Chess, we have to do actions in real-time. This means we are always performing tasks, in a more or less planned way, but we do not want to stay idle. The cooperation between the agents makes planning and execution of complex behaviours possible.

One of the issues designing a multi-agent system is the way in which agents communicate. Selecting the wrong structure can lead to a unsustainable and complex communication architecture. To address this problem the best way possible, we decided to build a blackboard architecture to make the communication easier. To achieve this, we included an Information Manager, which has the Blackboard and Working Memory functionalities. Nova uses the blackboard to store all the intentions (plans) of each agent, while the working memory is for saving all the information about the state of the game. Most agents communicate through this blackboard, however, there are some exceptions with some hierarchical agent messages. This is so, because the micro agents have a military organization, so they have a natural hierarchical organization: A Combat Agent (soldier) is commanded by a Squad Agent (captain) and this is commanded by the Squad Manager (general). However, even those agents also access the blackboard and working memory.

Figure 4.2 illustrates the multi-agent architecture with the blackboard and working memory. On the right hand side of the figure there is a Starcraft instance, on the left hand we can see Nova with two blocks of agents that interact with the Information Manager and the sensors of each agent saving information directly to the Working Memory.

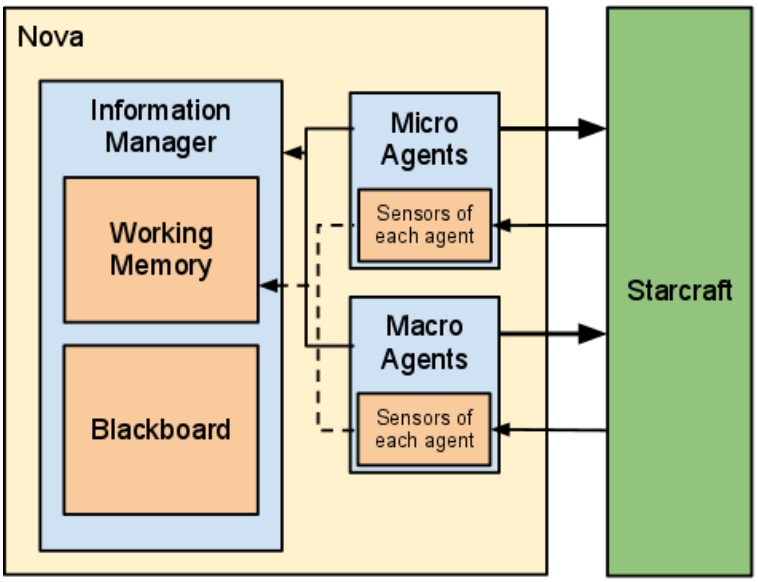

Figure 4.2: NOVA global architecture

4.2.1 Managers and Agents

We defined different generic modules, composed of managers and agents. Figure 4.3 shows the modules of Nova and the area to which belongs.

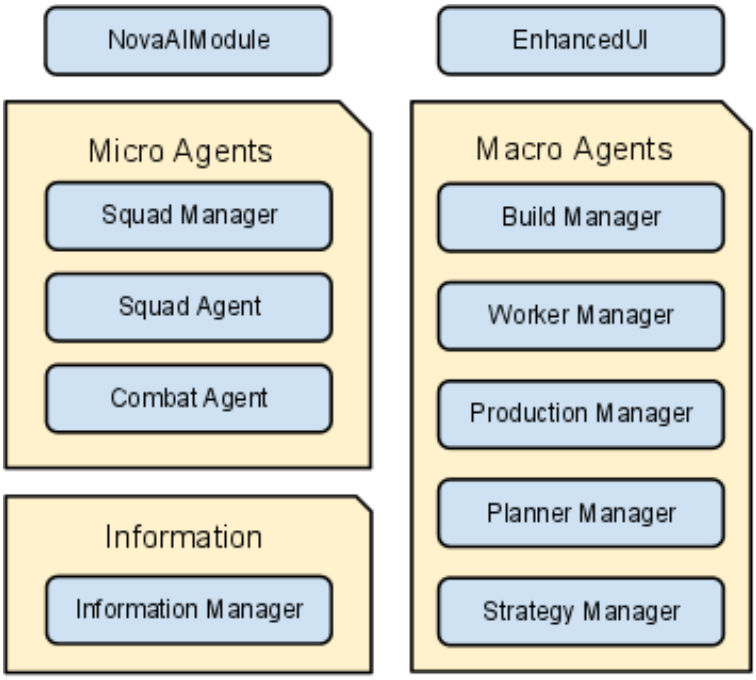

Figure 4.3: Nova modules overview

Generic modules

- **NovaAIModule**: Has the main loop of our bot and receives all the callbacks from BWAPI.
- **EnhancedUI**: Draws on the screen a lot of useful information on the screen to make more easy the debugging tasks during the bot execution.

Information retrieval

- **Information Manager**: Stores all the relevant information about the game (map, enemies, resources, threats, Ě) and accepts requests from the desired actions of the agents.

Micro Agents

- **Squad Manager**: Controls all the military squads of the bot, giving orders and receiving requests.
- **Squad Agent**: Controls all the units in the squad with different behaviours for each state.
- **Combat Agent**: Controls the behaviour of a military unit to optimize its abilities.

Macro Agents

- **Build Manager**: Executes the build order list and finds the best place to construct.
- **Worker Manager**: Controls all the workers to keep gathering and to do special tasks like building or scouting.

- **Production Manager**: Takes care of training new units or researching new technologies.
- **Planner Manager**: Monitors the army composition and balance the production order.
- **Strategy Manager**: Taking in count the type of enemy (Opponent Modelling), decides the build order and base army composition.

4.3 Working memory

In order to store all the facts of the game, we used the paradigm of a working memory. As we explained in Section 3.5.1, this is a shared memory where all agents can access to read the state of the world and write useful information detected by agent's sensors.

In the following subsections we will explore the most important sensors for our domain. We will explain how the agents gather information and how important it is in order to take further decisions.

4.3.1 Terrain Analysis

We showed the RTS concept of "fog of war" or unexplored terrain, but in a Starcraft game both players may have a previous knowledge of the map and they may know how to move through the map. Human players study the different maps before playing a game, in order to choose the best strategy. In other words, they do not need to explore the terrain to know it.

This is a different concept from Chess, which is always played in an 8×8 board. In a RTS we can play in many different maps with a variety of sizes, furthermore, in Chess, the pieces (units) have no inherited advantages based on where they are, aside from the tactical and strategic value of holding territory. On the other hand, in Starcraft, it is possible to have a higher ground, so units can acquire terrain tactical advantatge.

For all of these reasons, it is important to perform terrain analysis before starting the game.

As we saw in section 3.1 there are several algorithms to analyze maps. Our main goal is collecting the regions and choke points of the map. And for our concrete domain, we need to detect where the resources are, to identify the best locations to place a base.

To do this analysis, we used the open source library BWTA (Brood War Terran Analysis). This add-on uses Voronoi diagrams to identify the regions of the map and it is able to store the analysis in a XML file in order to avoid re-analyzing the whole map again when the same map is played repeatedly. In Figure 4.4 we can observe the analysis result of a typical Starcraft map. On the left hand we have an screenshot of a Starcraft map, and on the right hand we can see the same map divided by regions, represented by different colors, and the choke points of each region after the analysis.

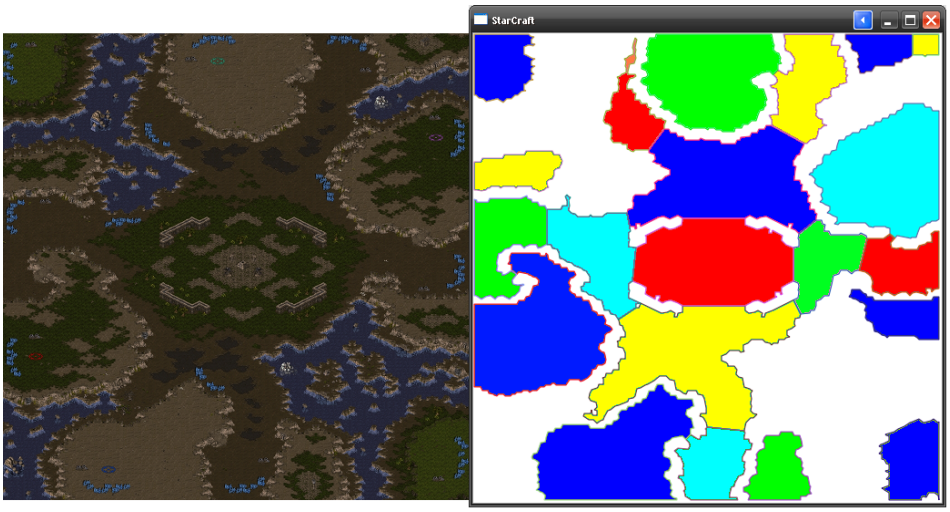

Figure 4.4: Terrain analysis of a Starcraft's map

With this information we are able to determinate the best locations to expand our base, make predictions about where our opponent will expand and identify which are the most important bottlenecks of the map. It can also be useful to improve pathfinding.

4.3.2 Opponent modelling

Another problem bots has to deal with in RTS games is imperfect information. For example, in Chess both players know what their opponent is doing and they can make predictions about what their opponent is planning. However, in Starcraft the player is only aware of the visible terrain, so it is impossible to know what an opponent is doing at every moment. This makes planning harder.

To solve this issue, opponents need to be scouted in order to get an idea about their plans. There are three different races in Starcraft (Terran, Zerg and Protoss), each one of them with different abilities, which allow for radically different scouting strategies. Therefore, the proper way to manage the scouting task depends on the specific race. In our case we decided to develop a Terran bot, therefore we have two options:

- **Sending a unit to scout**. This is a common practice to gather information about the enemy. The most extended is sending a worker to look for the enemy base early in the game. This has two functions, first we want to discover the starting location of the opponent, second we want to try to predict the initial build order and possible strategies that our opponent will take.
- **Using the scanner sweep ability**. Building a Comsat Station, Terrans have the ability to scan any map location and reveal a little area during 10 seconds. With this ability we can monitor all the enemy expansions. Firstly we have to scan all the empty base locations to detect the moment when an enemy expands. Secondly we have to monitor all enemy bases to predict technology

development and army composition. Another thing we have to keep in mind is reserving some energy to use this ability on cloaked enemy units, since the scanner sweep can reveal hidden units.

With the proper scouting abilities, Nova can gather sufficient information to make a opening prediction. In Starcraft context, we define an opening as the initial build order (the first buildings to be constructed at the beginning of the game), which is similar to the the idea of Chess openings.

It is quite usual to build a model to predict the opponent's opening. The usual way to do this is using a learning algorithm on Starcraft's replays (game logs)[13][14]. Although this has two problems, first we need to define opening labels and assign it to replays, we can find documentation about the most usual openings, but these openings are basic and there are a lot of variations and unusual openings; and second in a replay we have complete information, so we have to simulate incomplete information if we want to obtain an accurate model that can be used later during a real game.

In fact, the reason to predict opponent's opening is to plan the best counter strategy. Although this can be dangerous if we trust our prediction too much and the opponent shows us a fake opening.

In conclusion, due the complexity of building a good opening model and the questionable usefulness of this model, we decided for a simpler approach, consisting of detecting only the extreme openings that indicate the opponent is using a rush strategy. This kind of strategies are very aggressive and often are all-in. They try to attack as soon as possible but if we manage to hold back the attack, our enemy will be defenseless.

The last important piece of information Nova gather about an opponent is their army composition and their threat. To do this, Nova saves every seen enemy unit during the game until that unit is destroyed. As we said this has two purposes, first we want to know the air/ground damage per second of the enemy army, and second, the damage per second on every tile to build a threat map.

We calculated the DPS of air and ground damage for both players using the next formula:

$$DPS = \sum_{i=1}^{unitsSize} (weaponDamage_i \times \frac{24 frames}{weaponCooldown_i})$$

And to know the hit points of an army, we used the next formula:

$$HP = \sum_{i=1}^{unitsSize} (shiled_i + hitPoints_i)$$

We also extracted other indicators from this information to take decisions like the "time to kill". For instance, if we want to know how much time we need to kill all air enemy units we can use the next formula:

$$timeToKillEnemyAir = \frac{enemyAirHP}{ourAirDPS}$$

Now with all of this information we can take decisions at different levels, for example, we can decide the best unit to produce or, in case the squad has no options to win, make it retreat. On the other hand, the threat map is useful to detect the most dangerous areas or to find a save spot to retreat to. Through

this threat map we can also simulate the use of potential fields giving a virtual range to a melee unit. For example, we used this technique to create a potential field on Zealots (a melee Protoss basic unit) in order to repel Vultures and keep them in a safe attack range. In Figure 4.5 we can see the debug information of our threat map, each tile with a number is the value of "ground threat", in other words, the ground DPS on that tile. We also can observe the Vulture flee movement to the cyan dot.

Figure 4.5: Starcraft screenshot with threat map information

Table 4.1 summarize the information gathering and the usefulness of this information.

Gathering task	**Usefulness**
Initial scout with a unit	Enemy start location Detect rush strategy
Scanner sweep scanning	Detect target locations
Enemy air/ground DPS	Decide to build anti-air units
Threat map	Pathfinding to save location Avoid dangerous regions

Table 4.1: Gathering tasks and usefulness

4.4 Micro management

Micro management is the ability to control individual, normally combat, units. The agents responsible of this task are SquadManager, SquadAgent and CombatAgent.

As we explained before, these three agents follow a military organization, Figure 4.6 illustrates this hierarchy. The SquadManager can create and delete new SquadAgent instances and assign CombatAgent instances into the squad, also it can take important decisions like selecting target locations, when and where a squad has to retreat, assigning enemy units to a squad or deciding when merge two squads together.

The SquadAgent controls the behaviour of the squad with a simple finite state machine with states like Attacking, Moving or Merging.

The CombatAgent controls one combat unit, each unit has its abilities and properties and they must to be used it in an optimum way.

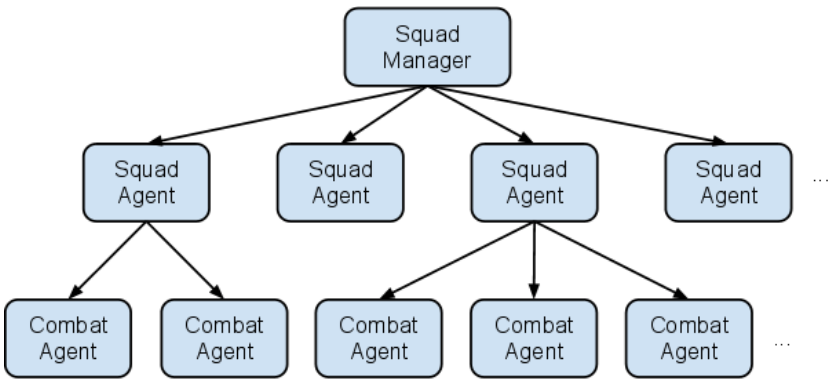

Figure 4.6: Micro agents hierarchy

Figure 4.7 represents an instance of these agents. Green members are Combat Agents, blue ones are the Squad Agents, and the red one is the Squad Manager.

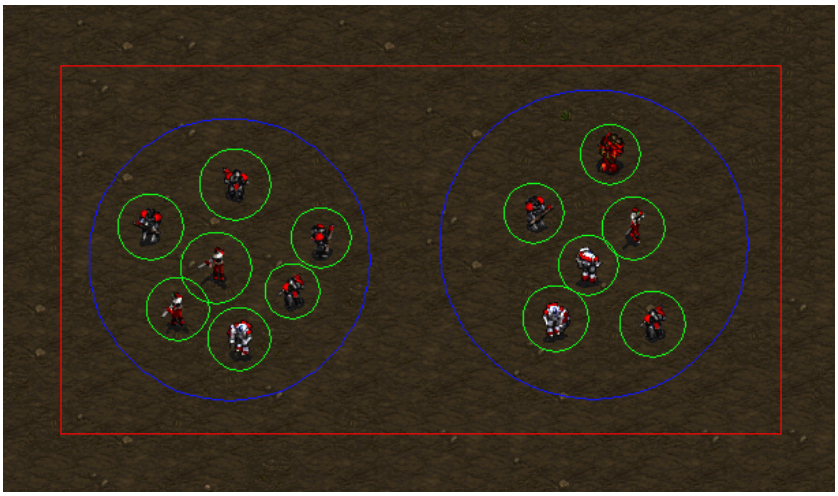

Figure 4.7: Example of micro agents instances

4.4.1 Squad Agent

We designed 4 states into our FSM: Idle, Get Position, Attack and Merge Squads. We have represented the state transitions in Figure 4.8.

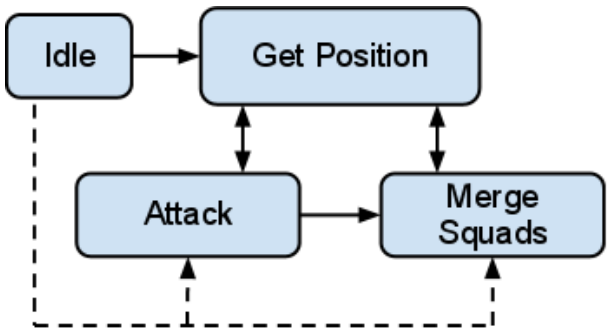

Figure 4.8: FSM of the Squad Manager

Idle is the initial state of a squad. We defined a minimum squad size, when a squad achieves this size, the Squad Manager assigns a target location to it and the squad state changes to "Get Position". In some conditions an Idle squad can be selected to attack a nearby enemy or to merge with another squad.

In **Get Position** the squad is trying to move to the target location fixed by the Squad Manager. In this state we designed a group behaviour in order to move the squad in a compact way. To this purpose, we used specific algorithms to cohesion the squad. The most basic of these algorithms is for a unit to wait for more units to catch up if there are less than a minimum number in range. A more elaborated

one, is to calculate the centroid and the spread of the units of the squad. Then we have to calculate a maximum and minimum spread depending on unit sizes. If the current spread is bigger than the maximum spread we do a cohesion movement and if the current spread is less than the minimum spread we can keep going towards the target location. Figure 4.9 illustrates the different spreads calculated and the centroid of the squad.

Figure 4.9: Spread debugging

If a squad reaches the target location, a new target is requested to the Squad Manager.

If an enemy is assigned to a squad, then the state changes to **Attack**. In this state the Sqaud Agent evaluates the enemy force and if it cannot win it performs a retreat request to the Squad Manager. In case it is decided to proceed with the fight, we delegate each unit control to the Combat Agent. In the scenario of having to retreat, the Squad Manager will assign another squad to merge with. Then we will change the state to Merge Squads. In the other hand if we killed all the enemies we will go to Get Position state.

In the **Merge Squads** state, we have to take care of moving towards the other squad assigned to merge.

4.4.2 Combat Agent

The Combat Agent controls a military unit during a combat. It has a common part to select the best target, and it specific methods for each kind of unit, in order to take advantage of their special abilities.

Target selection

Selecting the right target is essential to maximize the effectiveness of our attack, this can mean the difference between win or losing a combat. The Combat Agent assigns a score to each target, the higher the number the more priority a target has. To compute this score we decomposed the information in three parameters: aggro, tactical threat and distance to target.

$$Score = Aggro \times AggroWeight + Tactical \times TacticalWeight - Distance \times DistanceWeight$$

We also assigned a weight to each parameter in order to control the relative importance of each of them.

The *Distance* is the pixel distance to the target.

The *Tactical* threat is a fixed value depending on the abilities or type of unit. See Table 4.2 for details.

Tactical condition	**Value**
Terran Medic	100
Terran SCV reparing	400
Worker	250
Detector (Science Vessel, Obersver)	100
Carriers of other units	400
Protoss Pylon or Shield Battery	50
Zerg Queen	100
Buildings that can produce units	200
Resource Depots	100
self flyer unit vs enemy ground weapon	2500
self air weapong vs enemy flyer unit	2500

Table 4.2: Tactical values

The *Aggro* is the result of the DPS that would be inflicted to the target by the attacking unit (DPSToTarget) divided by the time to kill the target. We computed this using the following formulas and algorithm:

$$Aggro = \frac{DPSToUnit}{timeToKillTarget}$$

$$timeToKillTarget = \frac{targetHitPoints}{DPSToTarget}$$

Algorithm 1 $DPSToTarget(unit, target)$

Require: An attacking $unit$ and a $target$.

Ensure: A DPS value to target.

```
DPS = 0
if ( targetIsFlyer AND unitAirWeaponDamage > 0) OR ( !targetIsFlyer AND
unitGroundWeaponDamage > 0) then
  DPS = unitWeaponDamage*(24/unitAirWeaponCooldown);
  if unitWeaponDamageType == explosive then
    if targetSize == small then
      DPS = DPS*0.5
    else
      if targetSize == medium then
        DPS = DPS*0.75
      end if
    end if
  end if
  if unitWeaponDamageType == concussive then
    if targetSize == large then
      DPS = DPS*0.25
    else
      if targetSize == medium then
        DPS = DPS*0.5
      end if
    end if
  end if
end if
return DPS
```

We adjusted the weights to make the magnitudes quite equivalent since we have different concepts. After some experimentation the final weights for our computation are:

$$AggroWeight = 10000; TacticalWeight = 1; DistanceWeight = 0.1$$

Now we have the best target to attack, but we have different units with different abilities, so we have to control each one in a proper way.

Special micro management

We defined six special controllers for specific units:

- **Marine micro management**. If the stim pack[1] is researched, Marines can use it if they have enough hit points.
- **Biological micro management**. The biological units (Marine, Firebat and Ghost) will go close to a Medic if there is any in the squad and the unit's HP is low (under half of the maximum hit points).
- **Vulture micro management**. Vultures are a faster unit and they can increase their speed researching an upgrade. For this reason, Vultures can keep an enemy unit at a certain distance while dealing damage at the same time, specially effective against small melee units. To do this tactical maneuver we emulated potential fields through the threat map. Doing this, Vultures can win against a large group of Zealots without taking any damage. The algorithm consists in using the Patrol command near the enemy, which will trigger an attack, and retreating to a save location as soon as possible. We used the Patrol command because has a shorter shooting delay than Attack, and using the Patrol command in a 15 degree angle from the target, Vulture can avoid turning around and losing retreat time. In Figure 4.10 the process of (1) retreat, (2) patrol and (3) retreat again can be seen.

Figure 4.10: Vulture micro orders

[1]Using the Stim Pack abilitity, Marine and Firebat units double their rate of fire and increas their movement speed for a limited period of time in exchange of 10 hit points.

Researching the Spider Mines, Vultures gain the ability to lay mines. These mines can be used to protect the tanks lines or being more aggressive surrounding medium size enemy units. In our case, we modified the previous algorithm to lay mines instead to patrol against medium size units, but with the constraint of not laying mines too near between them.

- **Wraith micro management**. Wraiths have two basics moves. First, they will retreat to a safer location while their weapon is in cooldown. To find this spot we used a spiral algorithm on the threat map. Second, their cloaking ability. A Wraith will cloak if there is an enemy with air weapons in attack range and no enemy detector is present, since a detector makes this ability useless.
- **Tank micro management**. Tanks have the ability to change into siege mode. In this mode, tanks have the best attack range in the game, but they are vulnerable while changing modes since they cannot attack or move. Therefore, it is important to unsiege tanks at the right time. To achieve this, we will check the following conditions to decide if a tank should switch to siege mode:
  - We do not have any enemy in attack range.
  - The last change to siege mode time is greater than 4×24 frames.
- **Ghost micro management**. Like Wraiths, Ghosts have the ability to cloak, so we will use this ability if there are any enemy military unit near. And we will also use the Lockdown ability to neutralize any mechanical enemy unit. Before using it we have to check if the unit is not already Lockeddown or there are no Lockdown bullets for that unit.
- **Science Vessel micro management**. Science Vessels are the detector unit of the Terran. They also have special abilities but in this first version of our bot we have focused on their natural ability to detect cloaked units. Therefore, if any of our units detects an enemy cloaking we will produce Science Vessels. The Science Vessel will stay near the unit most closest to the target location and at the same time it will be away from anti-air threats.

4.5 Macro management

Macro management refers to the general economical aspects, or in other words, the way resources are gathered and spent. This includes actions like constructing buildings, conducting research and producing units. The goal here is to keep increasing the production capability while having the resources to support it. Specifically, Macro management in Nova is divided into two main tasks: resource gathering and resource allocation, where resource allocation consists in building constructions, researching technologies and producing units.

4.5.1 Gathering resources

The first important task in Starcraft is obtaining a sufficient income rate of resources. There are two types of resources: minerals and gas; optimizing this gathering task is key in the game. In the case of gas, the optimum number of workers per refinery is well known: three. If more workers are assigned to extracting gas the fourth worker will have to wait until the third finish, so the income rate will not increase. In the other hand, in the case of minerals the answer is not clear. It depends on the distance between the Command Center and the expectations of building a new Command Center transferring workers from one base to another. There are a few studies about this topic[15][16], but not a final conclusion. In our case, we decided to use the formula 2 *workers* $\times$ *mineral field*, since this is the minimal optimal value for all the existing studies. We established the optimal number of workers for gathering, but this is not the only task that workers must do. They have to do tasks like building, defending or scouting besides gathering.

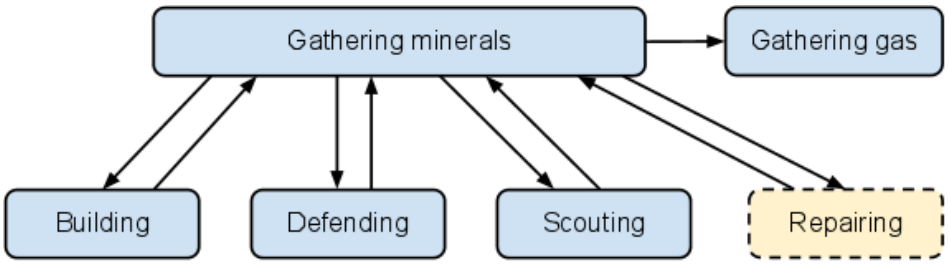

Figure 4.11: Workers' FSM

The Worker Manager controls all the SCV (Terran workers) through the FSM shown in Figure 4.11. Observing the FSM we can see that the initial state is "Gathering minerals" and only workers in this state can be requested to do other tasks. When a SCV completes any other task, it will return to "Gathering minerals" state, except for "Gathering gas" since this is a final task. State description:

- **Gathering minerals**. To enter in this state, the Worker Manager must select the mineral field with lowest workers assigned. Once inside the state, we have to take care that the field does not have any idle worker to keep up the incoming rate.
- **Gathering gas**. Very similar to "Gathering minerals", workers must be extracting gas at all times.
- **Building**. When an agent wants to start a building, the closest worker to the building placement is selected and changed to "Building" state. If the building placement is visible, the order of building is direct, else we first have to move the worker to the destination place so that the target position becomes visible before building can start.
- **Defending**. A worker enters in this state when our base is under attack (or under scouting) and we do not have any military unit (combat agent). Since workers can also attack, they can be used

as a defensive force. We will return to the gathering state if we kill the enemy or the enemy goes outside of our region, this is to prevent following the unit across all the map.

- **Scouting**. At the beginning of the game we will send workers to explore the multiple initial locations of our opponent. When the enemy base is located, the worker will move around the base in order to gather information about the initial build order of the opponent.
- **Repairing**. This is a special state because we have not defined it in our FSM. This is it because it is a very tiny temporal state. When a building is damaged and there are no enemies near, the closest SCV will repair the building. After finishing the repair, the worker will be idle, but since it is on "Gathering minerals" state it will resume gathering his mineral supply.

4.5.2 Spending resources

Gathering resources ensures an income rate, the resource spending task takes care of controlling the outcome rate. We have three main different ways to spend resources: building, researching technologies and producing units.

Building

The initial build order will condition which units we can produce and also the technologies to research, so this order is very important. We cannot build the buildings we want at any time since they have dependencies between them. Figure 4.12 shows that dependencies for Terran buildings.

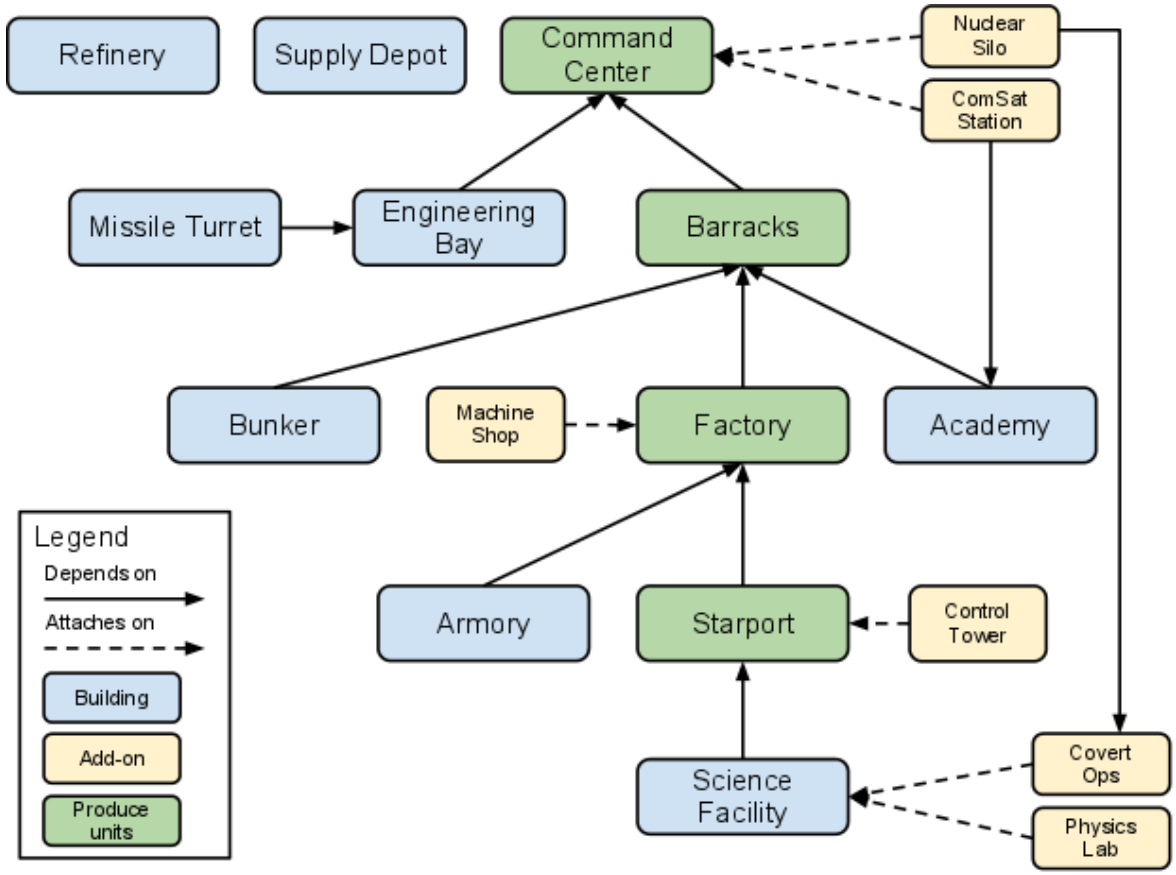

Figure 4.12: Terran's technology tree

These dependencies are controlled by the Information Manager through the Blackboard. Each time an agent requests a building to the Information Manager, it checks all the dependencies and sends the request to the Build Manager. The Build Manager stores these requests in a FIFO list. This FIFO has two exceptions to prevent blocking states:

- If we have the resources to build the top two buildings of the list, but the first one cannot be built for whatever reason, we will issue the order to build the second one.
- There is the possibility to request critical buildings, in those cases the request will skip to the top of the list.

The Build Manager also decides the buildings' placement. To do this we defined a two dimensional matrix, with the same size of the tiles of the map, where we marked each point as buildable or not buildable. First of all, we marked all the natural buildable tiles, then we blocked any tile with a static element like a geyser. Each time a building is placed, we block the tiles occupied by the building. We also reserve the right column and bottom row of any production building to be sure units can be spawned.

Figure 4.13 shows the debug information of the build map, green dots represent buildable tiles, red ones represent blocked tiles.

Figure 4.13: Build map information

To find the best build location, we designed an algorithm that, given a seed position, searches outwards in a spiral for a non blocked location on the build map. A pseduo code can be seen in Algorithm 2.

Algorithm 2 $getBuildLocation(seedPosition)$

Require: A seed location x,y.

Ensure: A build location x,y.

```
actualPosition = seedPosition
segmentLenght = 1
while segmentLenght < mapWidth do
  if ( buildMap[buildingSize] == buildable AND (buildMap[bottomBuilding] == buildable
  OR buildMap[topBuilding] == buildable) AND (buildMap[leftBuilding] == buildable
  OR buildMap[rightBuilding] == buildable) ) then
    return actualPosition
  else
    actualPosition = nextPositionInTheSpiral
  end if
end while
```

With this algorithm we prevent any posibility of new buildings creating walls that block paths.

Some managers have been designed with auto build features. For instance, the Build Manager checks on every frame if we need a supply depot to avoid blocking unit production.

Producing units

One of the keys in Starcraft is keeping the unit production constant, and this is the task of the Production Manager, which takes the control of all buildings that can produce units and keeps them busy. In Starcraft, unit training orders can be queued, but doing so spends resources that are locked until the training really starts, so we decided not to use the queue functionality.

The Production Manager also has the ability of auto build more production buildings. If all buildings of same type are training units and we have an excess of resources, in other words, we have resources to train another unit, build a supply depot and build the production building, then the Production Manager will request a new building by expressing its desires into the Blackboard.

The decision of what kind of unit we must train is taken by the Planner Manager. The Planner Manager balances the unit production according to our plans. For example, we can set a desired army composition of 60% marines and 40% medics and the Planner Manager will get the army rate most closest as possible to these numbers.

Researching technologies

Some buildings can research new technologies in order to improve some units. Each time the Strategy Manager sends a research request to the Product Manager, the latter puts unit training on hold until the research is in progress.

This behavior is because we prioritized to build over to research and to research over to train. But with the exception of workers since more workers means more resources, and thus training of workers is never blocked.

4.5.3 Strategy

Finally, the Strategy Manager decides what strategy the other managers must follow in order to win the game. The best strategy to use depends on the race against which Nova is playing. There are some reactive strategies that are common for all enemies' race, but we designed a specific FSM encoding the strategies to use against each race.

Reactive strategies

- **Cloaked enemy detected**. If a cloaked enemy is detected we will proceed to use the scanner sweep, or to build a ComSat Station if we do not have any, in order to reveal the area where the

enemy is cloaked. In parallel, we will start working towards the requirements to train Science Vessels.

- **Gas steal**. Sometimes our enemy will steal our gas at an early stage of the game. In those cases we will try to kill the worker and the building as soon as possible.
- **Defend base from scouts**. If an enemy scout is in our base we will try to kill it as soon as possible to hide our build order.
- **Scan empty bases**. Once we can use the scanner sweep ability, we will track all empty bases to search for any enemy expansion.
- **Need anti air units**. We defined a threshold to the enemy air threat. If the time to kill all enemy air units is bigger than 5 seconds we will keep training Goliaths (good anti-air units). Also, at first enemy air unit detected we will start to build missile turrets in our bases (anti-air defenses).

FSM strategies

- **Versus Terran**. Against Terran we will try to have a good production of tanks with siege mode researched. Thanks to initial scouting, we will detect if our opponent is performing a rush strategy, in which case we will produce Vultures to resist any marine rush. Figure 4.14 illustrates the states against a Terran opponent.

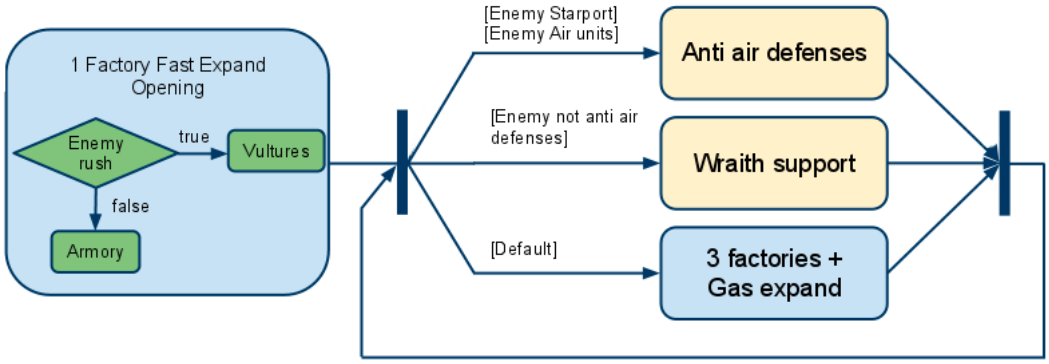

Figure 4.14: FSM against Terrans

- **Versus Protoss**. The Protoss strategy has three phases. First we will try to produce Vultures to manage any Zealot rush with a kiting technique or lay mines against Dragoons (100% Vultures). The second phase consists in changing progressively to a tank-centric production, from a 90% Vultures - 10% Tanks, to finally have a 25% Vultures and 75% Tanks army on the final state. Figure 4.15 shows the different states versus Protoss.

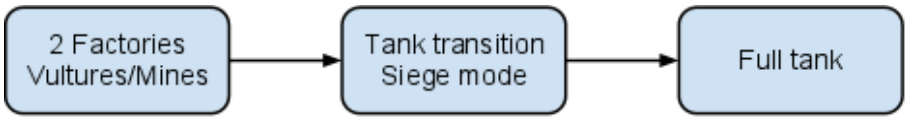

Figure 4.15: FSM against Protoss

- **Versus Zerg**. Our Zerg strategy is quite simple since it only has one state. We designed a standard "1 Barracks Fast Expansion" opening, so our army will be based on marines (80%) and medics (20%) with stim pack researched and some upgrades for our infantry. Some Zergs use a rush strategy consisting in sending a bunch of zerglings in an early attack. In those situations we will order to our workers to attack the rush.

4.6 Debug

Debugging a real time agent system is a hard task. To help us, we designed several tools for doing an online debugging and an offline debugging. The EnhancedUI provides us useful visual information during the game. Each manager can print relevant information on the screen. In Figure 4.16 different blocks of information are show, which allow quick access to any issue during execution.

Figure 4.16: Starcraft screenshot with macro debugging

Figure 4.17: Starcraft screenshot with micro debugging

In Figure 4.17 we can observe the debug info for each squad agent: the number of units in the squad, the number of assigned enemies, the state of the squad and the target location of the squad.

In some cases we need more accurate info about what is happening, for those cases we designed a persistent log file system to be able to analyze any problem in a more detailed way.

Chapter 5

Experimental Evaluation

5.1 Individual Agent Evaluation

In this section we present our experimental results of applying the different techniques on each agent individually. Some of the techniques have several parameters that need to be set. Due to lack of time, we were not able to deploy a machine learning system to automatically set the value of such parameters, so we tested different settings manually trying to converge towards optimum results.

5.1.1 Experimental Procedure

Experimental steps. For all micro experiments we used a simple balanced map without any terrain advantage (see Figure 5.1) and we ran several games against the built-in AI in a predefined situation like five vs five marines combat. For macro experiments we used maps from ICCup an International Cyber Cup where players from whole world compete on-line against each others. We tested Nova against the built-in AI and against other bots from AIIDE 2010 Starcraft Competition.

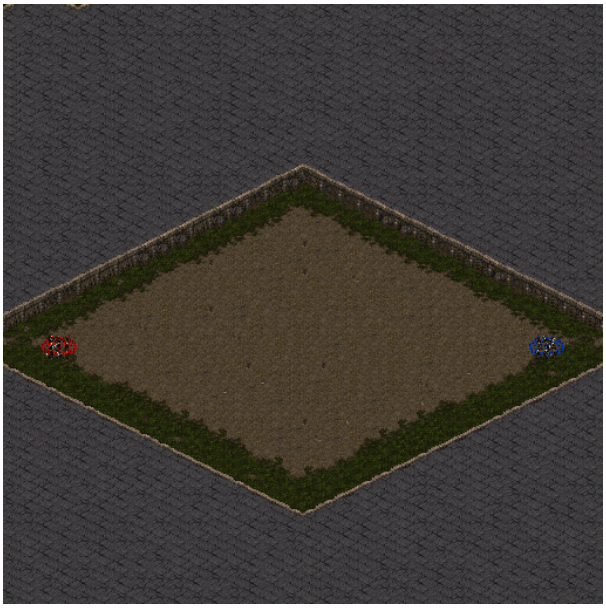

Figure 5.1: Starcraft map for our tests

Experimental platform. We ran all our experiments on a Windows Vista 32-bits, with Intel Core 2 Duo E8400 processor having a speed of 3 Ghz and 2GB of RAM.

5.1.2 Experimental Results

Pathfinding issues

Even if we did not implement any pathfinding technique, in our experiments we detected some issues with the built-in pathfinding. The issues appear when an indestructible building (defined in the map) is walling a path to a target location. The built-in algorithm only checks that building if it is on an unit's sight range, causing a blocking state where the unit is recalculating the best path to a location every time the building becomes visible or hidden. This problem is very common when a worker is trying to build a new base. In those cases we solve the problem defining a timer for a task, if the time is bigger than a threshold, we request another build location. In Video 1 we can see the mentioned issue with the time trigger solution:

Video 1: `http://www.youtube.com/watch?v=nt2ZSDue9kM`

Effects of improving best target selection

We wanted to prove that in equal army forces, a better unit control can make the difference between winning or losing a game. To test the impact of our implementation of a better target selection, we ran several games in balanced custom maps. Our tests also showed us that tactical movement is very

important too, but with our improved target selection we can manage to win in all battles against built-in AI. Video 2 shows how an army of marines and medics commanded by our bot can win the same army commanded by built-in AI:

Video 2: `http://www.youtube.com/watch?v=CFnNbttUleU`

Avoiding damage

To test our bot micro capabilities, we designed a scenario to handle combats of a ranged unit against a melee army. In our first iteration we tested a combat between one Vulture versus one Zealot without Potential Fields and Nova lost a 100% of the games. In a second iteration we applied Potential fields achieving a 100% win rate. In our third iteration we tested the combat of 1 Vulture versus six Zealots and we won half of the times due the poor target selection. In the final iteration we improved the target selection and won 100% of the times Our results show that applying potential fields to avoid damage is a powerful technique. Video 3 shows our iterative experiments.

Video 3: `http://www.youtube.com/watch?v=6QcEBFcMaBw`

Adaptive Strategy

n order to evaluate our capacity to adapt the current strategy, we tested our bot Nova in three different common scenarios.

Gas steal is a strategy consisting in building a refinery in the enemy geyser that makes a development delay or a blocking state for a bot without adaptive rules. In our experiments, Nova shows a right reactive planning against this strategy attacking the enemy's refinery and building its own refinery after destroying the enemy one.

Stop an opponent rush. Some aggressive strategies need to be detected as soon as possible in order to build a counter. In our tests against a typical BBS[1] opening, we observed a good transition between a normal state strategy training Tanks to a counter BBS strategy prioritizing Vultures to defeat Marines. Doing this, Nova changed his win ratio from 0% to 100% against a BBS opening. This shows the importance of a good initial scouting and a good adaptation to different opponent strategies.

Enemy air force all-in. Another example of Nova's adaptive strategy is the capacity to do a good opponent modelling to detect the enemy air threat. A quick detection of this threat, gives Nova time to change its unit production preferences and build a good army composition to counter the enemy air force. To test this capability we evaluated Nova against the AI bot Chronos (a participant in the 2010 Starcraft Game AI competition); we observed that without air-threat detection, Nova lost a 100% of the games, while including this feature, the picture changed completely, and Nova managed to win 100% of the times.

[1] A BBS opening consists in buildign two Barracks before a Supply Depot and doing a massive training of Marines.

Video 4 shows the highlights of our tests with Nova dealing the strategies described.

Video 4: `http://www.youtube.com/watch?v=2_GnbNocZ8g`

5.2 Global Nova Performance

In this section we present our experimental results of Nova's performance in a full game against the Starcraft built-in AI and other bots. To test Nova against other bots we participated on the following tournaments:

- The 2nd Annual AIIDE Starcraft AI Competition
- CIG 2011 Starcraft RTS AI Competition

On time to writing this master thesis the CIG Competition is still on progress, so we have not been able to include the results.

5.2.1 Experimental Procedure

Experimental steps. The map for each game in the competition was selected randomly between 10 maps (see Table 5.1 for details). And each bot played 30 games to each other to make the results statistically significant.

Map Name	Players	URL
Benzene	2	`http://www.teamliquid.net/tlpd/korean/maps/407_Benzene`
Heartbreak Ridge	2	`http://www.teamliquid.net/tlpd/korean/maps/199_Heartbreak_Ridge`
Destination	2	`http://www.teamliquid.net/tlpd/korean/maps/188_Destination`
Aztec	3	`http://www.teamliquid.net/tlpd/korean/maps/402_Aztec`
Tau Cross	3	`http://www.teamliquid.net/tlpd/korean/maps/3_Tau_Cross`
Empire of the Sun	4	`http://www.teamliquid.net/tlpd/korean/maps/406_Empire_of_the_Sun`
Andromeda	4	`http://www.teamliquid.net/tlpd/korean/maps/175_Andromeda`
Circuit Breaker	4	`http://www.teamliquid.net/tlpd/korean/maps/404_Circuit_Breaker`
Fortress	4	`http://www.teamliquid.net/tlpd/korean/maps/390_Fortress`
Python	4	`http://www.teamliquid.net/tlpd/korean/maps/147_Python`

Table 5.1: Maps of the AIIDE Starcraft AI Competition

Experimental platform. The tournament ran all the games on a Windows XP, with Intel Core 2 Duo E8500 processor having a speed of 3,16 Ghz and 4GB of RAM.

5.2.2 Experimental Results

Evaluation Against Starcraft's Built-in AI

On AIIDE competition we tested Nova performance against built-in AI. Figure 5.2 shows the results after 250 games against each race, where we can observe an excellent performance against Protoss and Terran but a lot of crashes against Zerg. This is because we have some bugs in our Squad Manager, when handling squad merge in some special conditions. As we mentioned before, debugging a real-time system is hard and, due the lack of time, we were not able to fix all bugs. So, as we can observe, the main reason for losing games against Zerg is Nova's crashes.

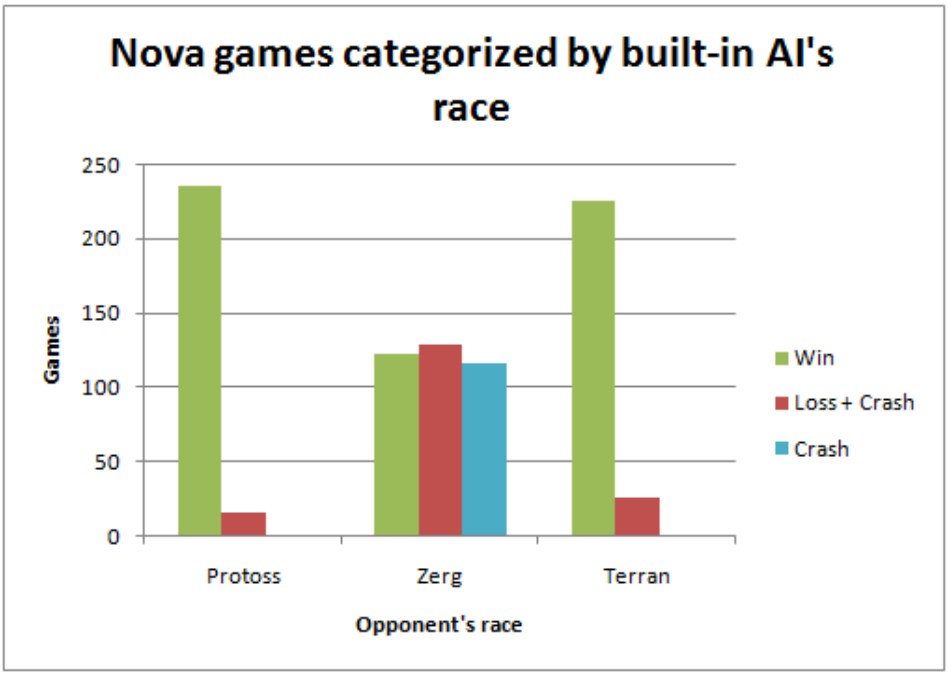

Figure 5.2: Nova games categorized by built-in AI's race

Figure 5.3 shows the win rate in % against each race.

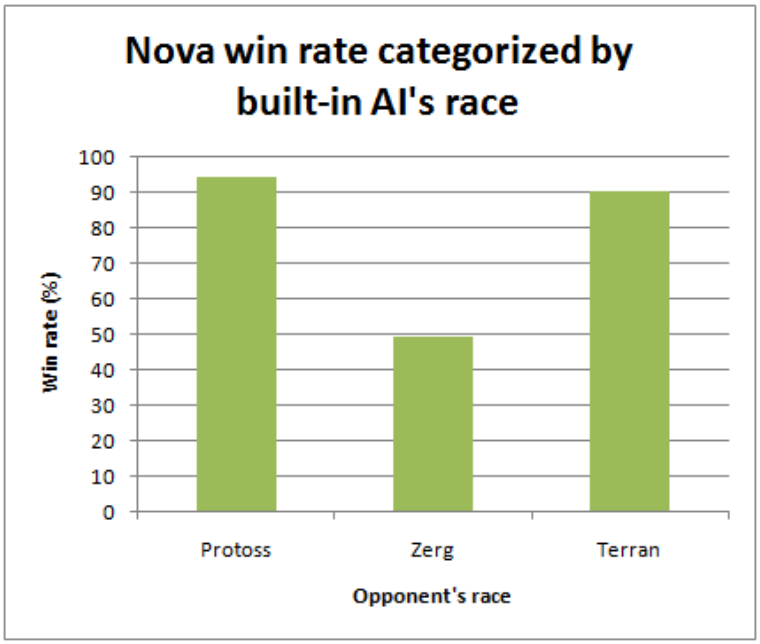

Figure 5.3: Nova win rate categorized by built-in AI's race

Results of the 2011 Starcraft AI Competition

Since Starcraft built-in AI always follows the same strategy, it is quite easy to design a winner strategy. The other bots in a tournament represent a better challenge to evaluate Nova performance. Thirteen different bots from different affiliations participated on AIIDE competition: six Protoss (UAlbertaBot, BroodwarBotQ, EISBot, Aiur, SPAR and Skynet), four Terran (Cromulent, Quorum, Undermind and Nova) and three Zerg (bigbrother, BTHAI and ItayUndermind).

After a total of 2.340 games, 360 for each bot, Skynet proclaimed as winner. Figure 5.4 represents the final classification with the win rate of each bot. Orange bars represent Protoss bots, red means Zerg bots and Blue for Terran bots. Nova's final position was 8th, since our most tested strategy was against Terran and most of the participants were Protoss, this is not bad result at all.

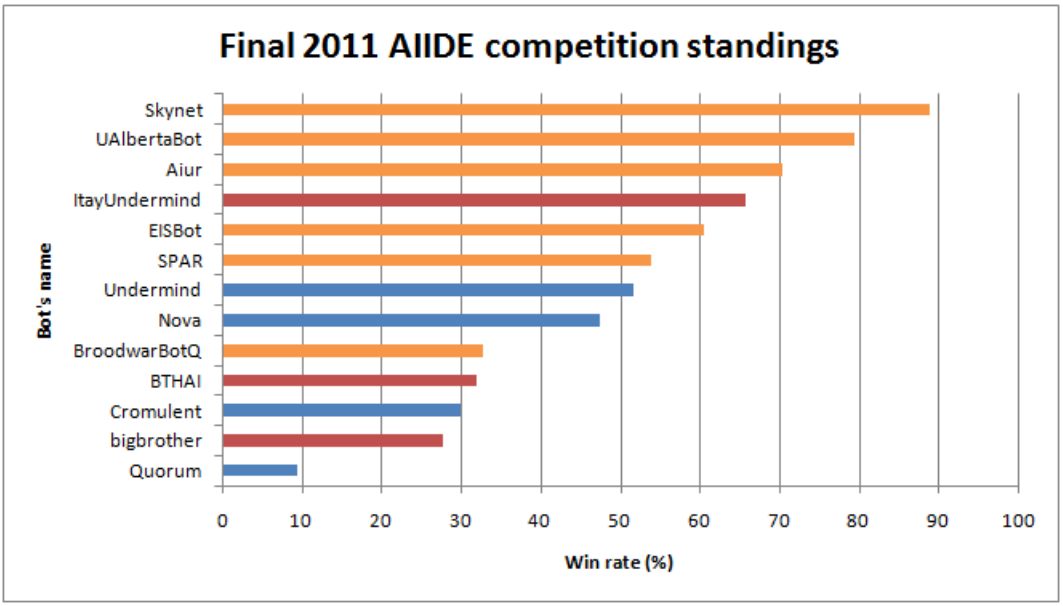

Figure 5.4: Final 2011 AIIDE competition standings

Next, we will analyze on detail all the statistical information to extract more conclusions.

One thing that can influence a lot the win rate is the robustness of the bot. During the competition each time a bot crashes was counted as a loss, also they defined a timeout rule by which, if a bot slowed down games by repeatedly exceeding 55ms on frame execution will lose the game. Figure 5.5 shows the crashes and timeouts of each bot. Notice there is only a little correlation between 'win rate' and 'crash+tiemout', meaning that a considerable crash rate like UAlbetaBot's, did not impede a good classification result.

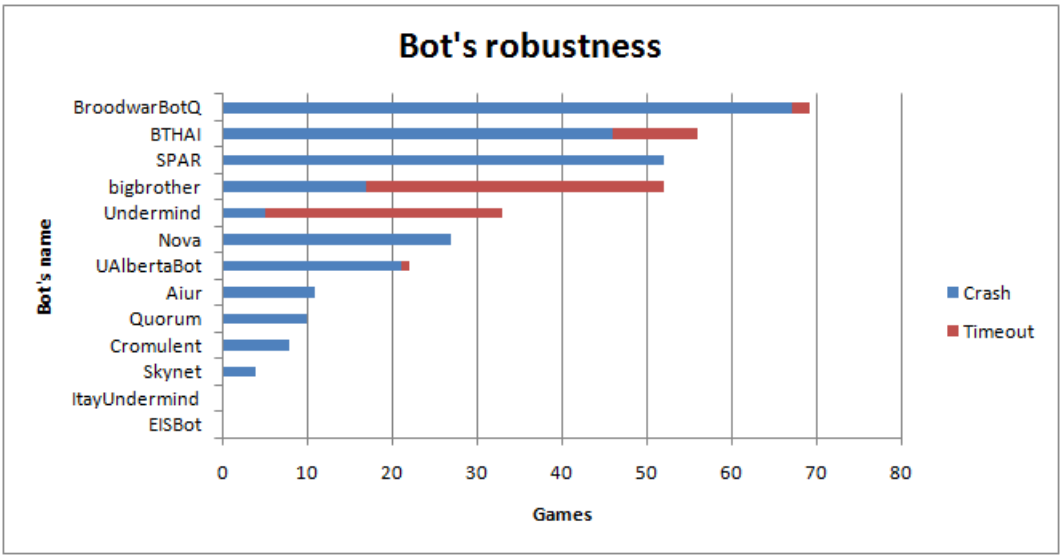

Figure 5.5: Bot's robustness

In Figure 5.6 we can see the individual results of Nova against each bot, where we can observe that most of Nova crashes happened against Zerg bots (BTHAI and bigbrother) and all the losses against these bots were only by Nova's crashes. Meanwhile, our best results were against Terran bots (Undermind, Quorum and Cromulent); and we had a good performance against the second placed UAlbertaBot, which used a Zealot rush strategy which our Vulture's kiting did a good job against.

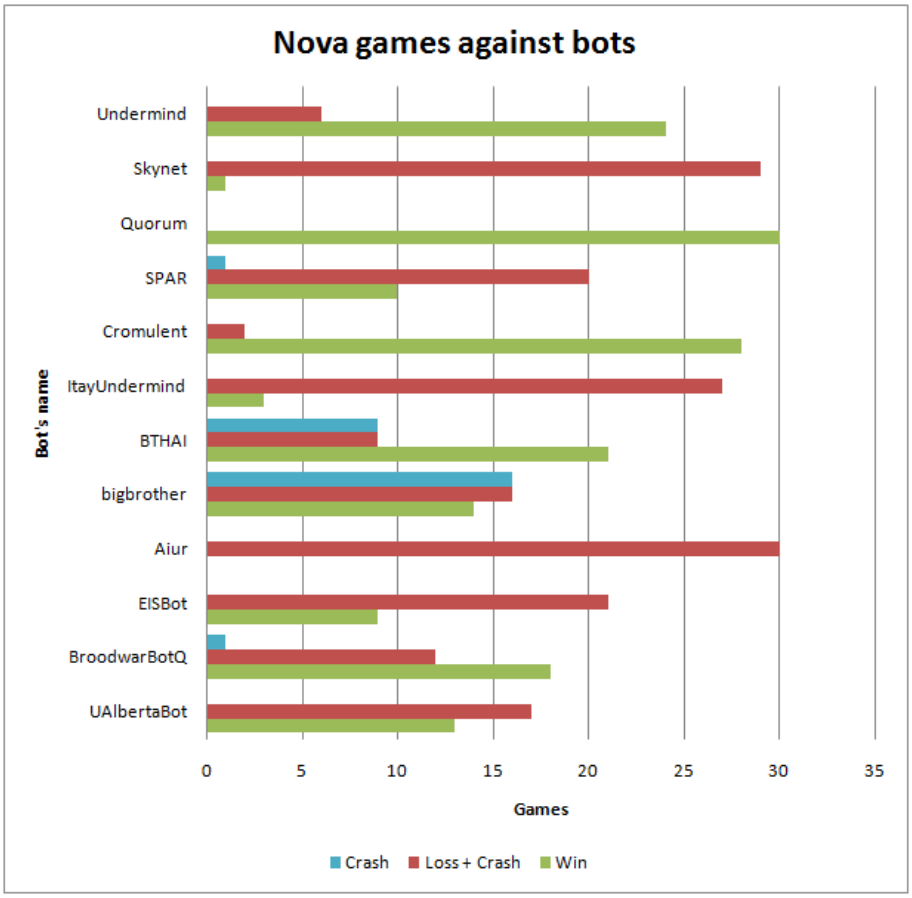

Figure 5.6: Nova games against bots

Figures 5.7 and 5.8 shows the results categorized by bot's race where we can observe the great results against Terran bots. In fact, Nova was the best Terran Vs Terran bot of the competition with a win rate of 91%. We also can see the buggy performance against Zergs and the poor strategy developed against Protoss.

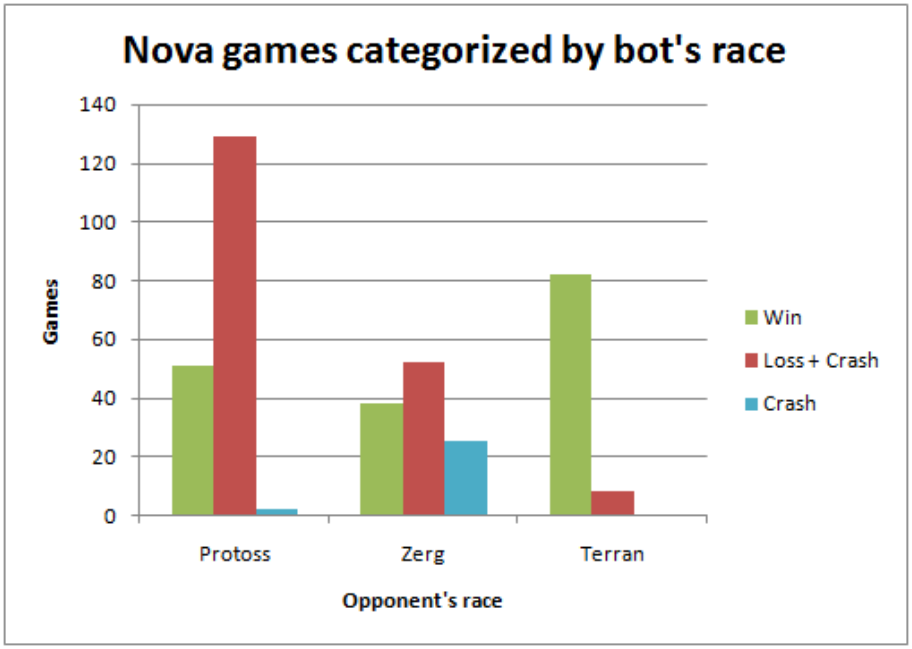

Figure 5.7: Nova games categorized by bot's race

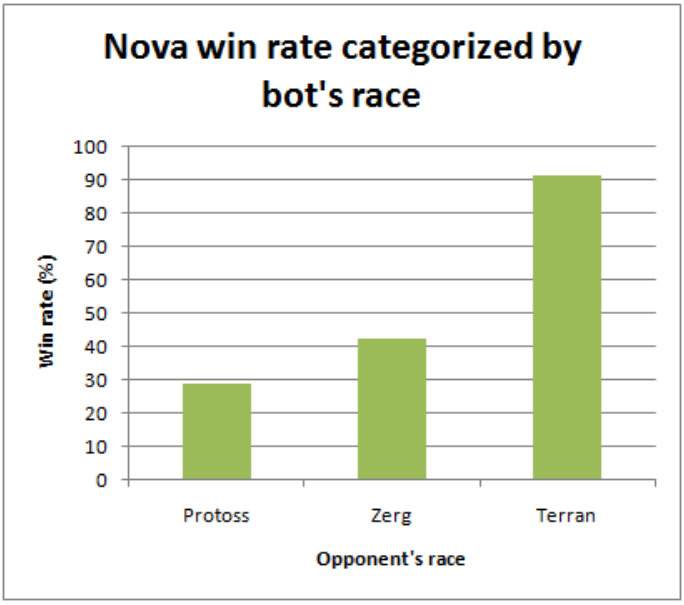

Figure 5.8: Nova win rate catgeroized by bot's race

Chapter 6

Conclusions and Future Work

Designing, implementing and debugging an AI for a RTS game is a complex task. AI bots have to deal with a lot of individual problems that can ruin the good job done by the individual agents. Mircromanagement tasks need a constant evaluation of the domain in real-time while using the static information like terrain analysis to take any possible advantage in any situation. On the other hand macromanagment tasks have to deal with a non-deterministic domain, forcing the agent to do predictions under uncertainty and executing the plans that have the maximum chance of success. In this context, designing a system capable of reactive planning, or replanning is a must if we want the strategy to be adaptive to unexpected events.

Our decision to build a multi agent system with a Working Memory and a Blackboard has given us good results on this real-time environment, but our crashes indicate that we need better tools for debugging these kind of systems. Potential Fields raise as an effective tool for tactical decisions, but they can be improved applying learning algorithms for better parameter tuning and they can be improved with terrain information. We also observed that the unit control task can improve a lot the bot's performance and trying to model all enemies behaviours is a hard task. Finally, using FSM as our main strategy handler makes Nova easy to predict and less effective against undefined situations.

Nova has demonstrated to be a solid and quite robust real-time multi agent system capable of performing the tasks required to play a RTS game. It is an improvement over the built-in AI of Starcraft and can be a great opponent against Terran players.

For all of this, Nova is a solid starting point for a more sophisticated RTS AI. There is a lot of room for improvements like:

- Implementing its own pathfinding and path planning systems due the issues of the built-in ones.
- A better terrain analysis to detect invincible buildings as walls, and classify regions by their height.
- Coordinate squads to achieve joint goals.
- Exploit tactical locations to take advantage in combats.

- Improve the opponent modelling to make better predictions about build orders and next movements.
- Test other techniques for planning like GOAP (Goal-oriented action planning).
- Design a learning system to emerge new AI behaviours and/or strategies.
- Better exploiting of all the special abilities and units of the game. Nova only knows how to control 8 of the 13 Terran's units. This can lead to new strategies.
- Use squad formations to flank the enemy and test more squad movement behaviours.
- Make a study of what is the best number of workers and the best way to do expansion transitions.

Bibliography

[1] M. Buro, "Real-time strategy gaines: A new ai research challenge," *International Joint Conference on Artificial Intelligence*, pp. 1534–1535, 2003.

[2] IGN, "Pc retroview: Dune ii," http://uk.pc.ign.com/articles/082/082093p1.html, 2000, [Online; accessed 4-September-2011].

[3] "Starcraft in south korea," http://www.onlinegraphicdesignschools.org/starcraft-in-south-korea, [Online; accessed 4-September-2011].

[4] L. Perkins, "Terrain analysis in real-time strategy games : An integrated approach to choke point detection and region decomposition," *Artificial Intelligence*, pp. 168–173, 2010.

[5] D. Demyen and M. Buro, "Efficient triangulation-based pathfinding," *Proceedings of the 21st national conference on Artificial intelligence - Volume 1*, pp. 942–947, 2006.

[6] C. W. Reynolds, "Steering behaviors for autonomous characters," *Proceedings of Game Developers Conference 1999*, pp. 763–782, 1999.

[7] H. Danielsiek, R. Stuer, A. Thom, N. Beume, B. Naujoks, and M. Preuss, "Intelligent moving of groups in real-time strategy games," *2008 IEEE Symposium On Computational Intelligence and Games*, pp. 71–78, 2008.

[8] J. Van Den Berg and D. Manocha, "Reciprocal velocity obstacles for real-time multi-agent navigation," *2008 IEEE International Conference on Robotics and Automation*, pp. 1928–1935, 2008.

[9] "Starcraft 2 - beta - 350+ zergling swarm - 1080p - path finding demo," http://www.youtube.com/watch?v=RfNrgymu41w, [Online; accessed 4-September-2011].

[10] "Supreme commander 2's flowfield pathfinding system," http://www.youtube.com/watch?v=iHuFCnYnP9A, [Online; accessed 4-September-2011].

[11] L. Liu and L. Li, "Regional cooperative multi-agent q-learning based on potential field," pp. 535–539, 2008.

[12] J. Hagelb, "A multiagent potential field-based bot for real-time strategy games," *International Journal of Computer Games Technology*, vol. 2009, no. 1, pp. 90–11.

[13] B. G. Weber and M. Mateas, "A data mining approach to strategy prediction," *Proceedings of the 5th international conference on Computational Intelligence and Games*, pp. 140–147, 2009.

[14] G. Synnaeve and P. Bessiere, "A bayesian model for opening prediction in rts games with application to starcraft," 2011.

[15] "Worker saturation," http://www.teamliquid.net/forum/viewmessage.php?topic_id=83287, [Online; accessed 4-September-2011].

[16] "Ideal mining thoughts," http://www.teamliquid.net/forum/viewmessage.php?topic_id=89939, [Online; accessed 4-September-2011].