Reactive Strategy Choice in StarCraft by Means of Fuzzy Control

Mike Preuss
Comp. Intelligence Group
TU Dortmund
mike.preuss@tu-dortmund.de

Daniel Kozakowski
Pirhana Bytes, Essen
daniel.kozakowski@tu-dortmund.de

Johan Hagelbäck
School of Computing
Blekinge Institute of Technology
jhg@bth.se

Heike Trautmann
Information Systems and Statistics
Münster University
trautmann@wi.uni-muenster.de

Abstract—Current StarCraft bots are not very flexible in their strategy choice, most of them just follow a manually optimized one, usually a rush. We suggest a method of augmenting existing bots via Fuzzy Control in order to make them react on the current game situation. According to the available information, the best matching of a pool of strategies is chosen. While the method is very general and can be applied easily to many bots, we implement it for the existing BTHAI bot and show experimentally how the modifications affects its gameplay, and how it is improved compared to the original version.

I. INTRODUCTION

Realtime Strategy (RTS) games are not only still a very popular computer game genre, but also a very interesting testbed for AI techniques as setting up a good bot (in terms of playing level, not to speak of believability which in case of RTS is not very conflicting) is still very challenging. Regular competitions held in the last years (at AIIDE conference 2010, 2011, and 20121, at CIG conference in 2011 and 20122) have shown that there is some improvement, but we are still far from being human-competitive. Notably, a lot of the currently published work on RTS games deals with the StarCraft game that was published by Blizzard in 1998. Next to the impact of the competitions, this is most likely due to three facts:

- StarCraft has been immensely popular in the e-sports leagues (now replaced by StarCraft II which employs very similar gameplay mechanisms), with considerable media coverage. Even young casual players know about the game.
- The game is still sold, at a very low price.
- Most importantly, a very powerful programming interface exists with the BWAPI3 that uses the original human interface as backdoor to inject bot code.

One of the reasons why even average human players can easily beat the existing bots is that these cannot adapt their strategies to the course of the game. While most current bots just go with one hard-coded strategy, others as AIUR4 choose from a set of strategies at the beginning, but then stay with it throughout a game. Newest developments enable to guess the opponent’s strategy by means of scouting information and a large database of recorded games, as done by Synnaeve and Bessiere [1] on base of the work of Weber and Matteas [2]. However, this is only targeting at setting up the own building order and currently not applied to the middle game. Very few attempts exist to change the strategy within a game. A case-based reasoning approach has been presented by Mishra, Ontaño, and Ram [3], and Young and Hawes [4] provide a mechanism for detecting a suitable priority profile dynamically throughout a game, thereby changing the chosen strategy. This system is fuelled by a genetic algorithm. To our knowledge, no fuzzy based system has been suggested yet to establish dynamic strategy shifting, although a fuzzy controller approach (see e.g. Passino and Yurkovich[5]) appears very well suited: it is the heart of fuzzy logic to express expert knowledge first in common language rules and then to translate these rules into a computable reasoning system. Fuzzy logic thus should be the ideal tool to implement strategy changes in a way a human would use them, and could be used to improve the playing level of a bot as well as its believability.

Consequently, in this work we show how to establish a strategy selection fuzzy controller within an existing bot, BTHAI [6], but concentrating only on the ability to play well. We chose this bot as it possesses a very modular design and is easy to enhance with new functionality, and its code is available and well documented. The fuzzy-enhanced BTHAI is experimentally compared to its original version used at the CIG 2012 StarCraft competition, the default StarCraft AI, and another existing bot (AIUR) in order to check which strategy changes actually occur, and how they pay off. We start with describing the BTHAI architecture first and then explain the Fuzzy controller and its integration into BTHAI. Then follow the experimental analysis and some conclusions.

II. BTHAI ARCHITECTURE

Besides the obvious task of playing StarCraft reasonably well, the main goal of the BTHAI is to provide an architecture that can be easily used and modified. The flexible and modular design makes it ideal for implementing new logic, in this case the fuzzy controller. The architecture of the bot is shown in Figure 1. It is divided into three modules; Managers,
CombatManagers and Agents. Each module is described in detail below.

A. Agents

Each unit and building in the game is handled by an agent. The agents implementation uses an inheritance system where, for each newly constructed in-game unit or building, the most specific agent implementation found is used. A difference is that most units, due to having special abilities, have their own agent implementation while buildings often have similar functionality (doing upgrades or constructing units) that is handled from the general StructureAgent.

Special abilities have been implemented for most units. Examples are Terran Marines who can enter bunkers and use Stim Packs, Siege Tanks enter siege mode when there are ground targets within range and Science Vessels use EMP Shockwave on shielded Protoss units. Worker units have very different tasks than other mobile units and therefore have their own agent, WorkerAgent. Workers shall gather resources (minerals and gas), construct buildings and repair damaged buildings (Terran only).

B. Managers

A manager is a global agent that handles some higher level tasks. The BuildPlanner agent contains a build order list of what buildings to construct and in which order. It uses the ResourceManager agent to check if there are enough resources (minerals and gas) available to construct the next building in the list. Build order lists are read from text files during the startup of the bot. These contain ordered lists of the buildings to be constructed, using the BWAPI name of the building types.

The UpgradesPlanner agent works in a similar fashion. It contains lists of unit/building upgrades and technology improvements. When the requirements for the next upgrade or tech is fulfilled (they usually require that a specific building has been constructed) and there are enough resources available, the upgrade is built or the technology is researched. There are three priority levels for upgrades/techs, and all entries in a higher priority list must be handled before starting the upgrades/research in a lower priority list. The upgrades/techs lists are also read from text files at the startup of the bot.

The AgentManager agent is a container class for all agents in the game. It contains a list of active agents, adds new agents when a unit or building has been created, and removes agents when a unit or building has been destroyed. It also contains a lot of decision support methods, for example calculating the number of enemy units within a specific area or how many own units of a specific unit type that have been created.

The ExplorationManager agent is responsible for all tasks involved in exploring the game world. It controls a number of scout units, and decides where each unit should go next.

C. CombatManagers

For combat tasks the bot uses a command hierarchy in three levels (Commander, Squad and UnitAgent). The Commander agent is responsible for decisions at the highest level. It is a global agent that can be accessed by all agents in the system (similar to non-combat managers). Some tasks of the Commander are:

- Launch attacks against the enemy.

Fig. 1. BHTAI architecture, consisting of the three main modules Managers, Combat managers and Agents
• Find a good spot to attack an enemy base at.
• Decide which squads, if any, shall assist buildings or units under attack.
• Decide good spots for defending own base(s).
• If playing Terran, assign workers to repair damaged buildings or important mechanical units.

The Commander is implemented as a rule based system. It is in charge of one or more squads. Each squad is in turn responsible for a number of units attached to the squad. The Squad agent handles tasks such as moving the units in the squad to a position, defending a position or attacking a target.

A squad can consist of almost any number and combination of units. The squad agent is responsible for coordinating its units to attack in an efficient manner. This involves moving the squad as a group and not let the units spread out too much, put long range units in the rear when attacking, defending weak but powerful units such as siege tanks, and more. Squads setup are read from text file at the startup of the bot.

III. Fuzzy Controlled Strategy Choice

Fuzzy logic is employed to model and compute with statements that are formulated in usual human language, which are usually difficult to translate to crisp values or standard crisp sets. Unlike the latter, a fuzzy set can contain elements to a certain degree. E.g., the fuzzy term old of the linguistic variable age of humans is not translated to a sharp range (as age > 60), but can be expressed via a (in this example) linear membership function that may start with membership degree 0 at the age of 40 and reaches degree 1 at the age of 80. A human being at age 60 would then be interpreted to be 50% old but could at the same time be 10% young and 40% middle-aged. In the following, these memberships are used as input values for fuzzy rules which can not only fire or not fire, but fire to a certain extent. The outputs of the rules are then combined to one fuzzy set on which at last a defuzzification scheme is applied to generate a crisp output value.

It gets immediately clear that this scheme is well suited to handle expert knowledge that comes in human language. As an example we take the statement ‘if the opponent possesses many flying units and I have few anti-air units, then I build many anti-air units’ (note that it would also be reasonable to build many flying units, but this would be another strategy). If we translate this into an algorithmic description, finding exact values for few or many will be difficult. Fuzzy logic can assist us here as any concrete number (e.g. 5) can be ‘a lot of few’ and ‘a bit of many’ at the same time, and the transition between the two is made smoother than it would be for a standard (crisp) rule-based system.

In our case, we want to rate the usefulness of a number of predefined strategies according to the current game state, employing our own bot’s status information and the available

Fig. 2. Two in-game shots with different strategies selected, templar attack on the left, and ground attack on the right side. Note the list of compared strategies and their defuzzified evaluation values (usefulness) on the right of each shot.

Fig. 3. Example of a strategy file, here: Templar attack
information about the opponent. After defuzzification, each strategy is assigned a crisp output value ranging from 0 to 100. The strategy with the highest value is then taken and injected into the BTHAI by means of loading the appropriate files for build order, squad setup, and upgrade. Thus, no change is done to the overall behavior of the BTHAI, but, as time goes by, we provide it with different units and buildings according to the chosen strategy.

Next to the ability of the fuzzy controller to detect the best matching strategy, the achieved improvement over the default BTHAI configuration therefore heavily depends on the setup of these 3 files that have to be created for every single strategy. We rather see our current implementation as a demonstration of how the system works and what it can do and do not claim that the strategy choices we made are optimal in any sense. Besides, there is currently no mechanism to prevent that strategies are changed often between two almost equally good alternatives. Our experimental results will show that this does not happen too often, but it is definitively a weakness that will have to be addressed in future. Note however, that a repair mechanism for units that does not match with the new squad setup has been implemented, these are moved into an offensive optional squad in order to support the next attack.

The current strategy set is built only for playing the Protoss race and consists of the following 5 alternatives: Templar attack, rush defense, ground attack, anti-air, and air attack. In order to enable the modified BTHAI to play Terran or Zerg factions, one would have to create appropriate strategy files (Figure 3 defines the templar attack strategy as an example, it is explained in more detail later on) together with their specific build order, squad setup, and upgrade files. However, as the whole configuration is done via text files, it is rather simple to modify or extend the system.

Before describing how the fuzzy reasoning actually works, we need to define linguistic variables that are employed for expressing the current game situation. Table I provides an overview over all considered variables. In order to assign (crisp) values to these variables, the current game situation is evaluated, taking only the incomplete amount of information into account that is also available to players. Army and defensive strength is measured by summing up the destroy scores of units and/or buildings. This is the amount of points granted to a player for destroying a certain unit/building, ranging from 50 for a Zergling to 2400 for a battle cruiser.

The value for InvisibleThreat is the difference of the own detector values to the value sum of all known invisible enemy units and buildings that are necessary to produce invisible units. The DetectorThreat is computed by summing up the value of the known enemy detectors that are able to see invisible units. OpponentRace is a static value that is set to 1000 if the opponent faction is Protoss, 2000 if it is Zerg, and 3000, if it is Terran (the values themselves are not important, the numbers only need to be different).

The rules for each of the 5 alternative strategies are provided to the controller by means of a strategy file that looks similar to the one of the Templar attack given in Figure 3. These employ an EBNF-grammar, with one block for each variable. Note that we have exactly 3 rules for each variable, one for every fuzzy term (Low, Medium, and High) of the usefulness output variable. This is because we employ Combs method [7], [8] in order to prevent the combinatorial explosion of rules as described in [9]. Combs method works by partitioning rules that employ combinations of fuzzy terms into simpler rules that each possess only one fuzzy term. It
employs the following result from (crisp) logic (easily shown by truth table):

\[(A \land B) \Rightarrow C \Leftrightarrow (A \Rightarrow C) \lor (B \Rightarrow C)\]

Under the assumption that rules with the same consequent are going to be combined with the or operator (this is a design decision), we can partition the rule if A and B then C into the two rules if A then C and if B then C.

With only one fuzzy term in the condition of a rule, the number of possible rules is of course very limited. Although it is not possible to transfer every existing fuzzy rule base into Combs format, but if the decision to use it is already known when setting up the rule base, it is not really a handicap.

We will now use the Templar attack strategy as an example in order to explain how the crisp input values are matched to fuzzy terms, how the rules are established and evaluated, how their outputs are combined into one fuzzy set and how this set is defuzzified to obtain a crisp output value (usefulness as assessment of how well a strategy matches with the current game situation).

By Templar attack we mean a Dark Templar rush. The strategy depends on the 2 linguistic variables GameTime and GasRichness. This makes sense because (Dark) Templars are invisible units that can have a very high impact as long as no detector units/buildings are available to the opponent. If the game lasts longer, they more and more use their advantage. Additionally, building Templars uses up a lot of vesepen gas so that a low amount of vesepen gas prohibits creating more units. One could also have added the DetectorThreat variable, but at the one hand we have no detector information when the game starts and we will see in the following that the strategy choice in this case works quite well without an additional variable.

As for all input variables, the fuzzy terms and their fuzzy sets are modeled after the same shape, which is given in Figure 4, left. The strategy file therefore has to provide numerical values for the 5 defining points LowerBound, LowerTurning, Center, UpperTurning, and UpperBound. The output variable usefulness always consists of the three fuzzy terms Low, Medium, and High, with the associated fuzzy sets given in Figure 4, right.

The <inference> blocks of each rule provide the usefulness level (given as 0, 1, 2 for Low, Medium, and High, respectively) for the fuzzy terms of the condition, which leads to the following 6 rules in this case:

- if GameTime_Low then Usefulness_High
- if GameTime_Medium then Usefulness_High
- if GameTime_High then Usefulness_Low
- if GasRichness_Low then Usefulness_Medium
- if GasRichness_Medium then Usefulness_High
- if GasRichness_High then Usefulness_High

Let us assume that the computation of the input variables resulted in 500 (seconds) for the GameTime variable and 1200 (units of vesepen gas) for the variable GasRichness. The degree of membership (DOM) for each fuzzy term that is employed in the above rules can then be computed as follows:

\[ DOM_{GameTime\_Low}(500) = 0.25 \]
\[ DOM_{GameTime\_Medium}(500) = 0.75 \]
\[ DOM_{GameTime\_High}(500) = 0.0 \]
\[ DOM_{GasRichness\_Low}(1200) = 0.0 \]
\[ DOM_{GasRichness\_Medium}(1200) = 0.53 \]
\[ DOM_{GasRichness\_High}(1200) = 0.47 \]

Putting the computed DOM values into the rules leads to
the DOM values for the output variable:

\[
\text{GameTime\_Low} = 0.25 \Rightarrow \text{Usefulness\_High} = 0.25 \\
\text{GameTime\_Medium} = 0.75 \Rightarrow \text{Usefulness\_High} = 0.75 \\
\text{GameTime\_High} = 0.0 \Rightarrow \text{Usefulness\_Low} = 0.0 \\
\text{GasRichness\_Low} = 0.0 \Rightarrow \text{Usefulness\_Medium} = 0.0 \\
\text{GasRichness\_Medium} = 0.53 \Rightarrow \text{Usefulness\_High} = 0.53 \\
\text{GasRichness\_High} = 0.47 \Rightarrow \text{Usefulness\_High} = 0.47
\]

As we have defined above that different results for the same output variable are aggregated via the fuzzy or operator (this just means to compute their maximum) in order to use Combs method, we can derive the final values for the three output variable fuzzy terms:

\[
\text{Usefulness\_Low} = \max\{0.0\} = 0.0 \\
\text{Usefulness\_Medium} = \max\{0.0\} = 0.0 \\
\text{Usefulness\_High} = \max\{0.25, 0.75, 0.53, 0.47\} = 0.75
\]

In order to obtain a crisp output value, a defuzzification method has to be applied. There are several common techniques to do that, with the center of gravity computation being the most well-known. [9] gives an overview for utilization in games. As we are in a real-time situation, we choose a more simple alternative that may slightly compromise accuracy but can be computed very fast, namely the average of maxima. This method employs the representative value of each fuzzy set, which is the middle of the values for which the degree of membership is 1. The representative values for the fuzzy sets of the output variable are 12.5, 50, and 87.5, for Low, Medium, and High, respectively (see Figure 4, right). The general formula for computing the average of maxima just sums up the products of the representative values and the appropriate output values for each output variable fuzzy terms (3 in this case, see usefulness computation above) and divides that by the sum of all output values (note that computing the center of gravity would be reasonably more complex):

\[
\text{crisp output} = \frac{\sum \text{representative value} \times \text{term output value}}{\sum \text{term output value}}
\]

Applying this to the concrete output values derived above results in the crisp usefulness value of 87.5 (87.5 is also the maximum, 12.5 would be the minimum due to the chosen defuzzification method).

\[
\text{crisp output} = \frac{12.5 \times 0.0 + 50 \times 0.0 + 87.5 \times 0.75}{0.0 + 0.0 + 0.75} = 87.5
\]

This is done in turn for all defined strategies, and the strategy with the highest usefulness value (if higher than the value for the currently chosen strategy) is injected into the BTHAI. In the following, we will term the BTHAI version with integrated fuzzy controller FABTHAI for fuzzy adaptive BTHAI in order to differentiate it from the original BTHAI when describing the experimental results. For debug purposes, the current FABTHAI implementation outputs the computed usefulness values for all strategies on the right side of the screen, as can be seen in Figure 2. A strategy with a usefulness of 87.5 is very likely to be chosen. Usually, the computed values range between 50 and 80.

IV. EXPERIMENTAL ANALYSIS

With the following experiments, we want to discuss two fundamental questions:

- Does the fuzzy adaptive BTHAI (FABTHAI) really adapt the chosen strategies to the current game situation, e.g. with respect to the opponent and the map?
- Does the FABTHAI play better than the original BTHAI?

The chosen experimental setup is rather explorative than accurate, at this stage we would like to get the big picture...
instead of making definitive statements. We test the FABTHAI against BTHAI, AIUR (CIG 2012 competition version), itself, and the default Protoss AI, with different motivations:

- **FABTHAI vs. BTHAI**: FABTHAI (always playing Protoss) is run against BTHAI (Terran) and BTHAI (Protoss), in order to see if it shows differences in the strategy choice process. Here we can also assess if the FABTHAI has improved compared to the BTHAI. However, one must be very careful with such statements as the FABTHAI could specifically be optimized to beat BTHAI but otherwise play weaker.

- **FABTHAI vs. AIUR**: As AIUR is a well-established bot that is considered just a bit worse than the best bots (placed 3rd of 10 in the CIG 2012 StarCraft competition), it may be a good candidate to test against. Comparison data for AIUR vs. BTHAI is available from that competition, too\(^5\). BTHAI was able to win about 20% of its games against AIUR.

- **FABTHAI vs. FABTHAI**: By testing against itself, we want to see how consistent the strategy choice process is, and if strategy choices are different from games against BTHAI and AIUR.

- **FABTHAI vs. default Protoss AI**: We try to answer 2 questions, namely if the FABTHAI is able to consistently beat the default AI, and how large the influence of the chosen maps is. Next to the Fading Realm map, we also play on the Athena-2 and Legacy maps from the CIG 2012 StarCraft competition.

Next to the binary win or loss information, we record the game time, the number of strategy changes, and the percentage each of the 5 alternative strategies was employed with during one game. If not stated otherwise, all match groups are run 10 times on the Fading Realm map that comes with the StarCraft Broodwar installation. It is clear that with only 10 repeats, we cannot expect hypothesis tests to become statistically significant too often, thus the following results should be taken with care. However, we can report some interesting trends.

Figure 5 shows the differences in the number of strategy changes within one game. Interestingly, the FABTHAI reacts very differently on a BTHAI playing Terran instead of playing Protoss, the number of strategy changes is even significantly different for a Wilcoxon rank-sum test. It seems that FABTHAI also performs better against BTHAI playing Protoss (100% wins). Against AIUR, even fewer strategy changes were tried than against BTHAI/Terran, and the overall performance of FABTHAI against AIUR seems to be comparable to the one of BTHAI against AIUR. In columns 4 and 5, we see the two FABTHAI bots battling each other, and they seem to behave largely similar. Regarding strategy changes, a FABTHAI opponent seems to appear more alike a BTHAI/Terran opponent than a BTHAI/Protoss opponent from the viewpoint of a FABTHAI. The last 3 columns against the Protoss default AI show that the map indeed has an influence.

\(^5\)http://ls11-www.cs.uni-dortmund.de/rs-competition/starcraft.cig2012
on the number of strategy choices, and also on the performance of the FABTHAI. In particular, the Athena-2 map contains several narrow passes (and also plateaus not accessible by ground units), where FABTHAI has its difficulties as it cannot handle air transport any better than BTHAI (however, this is true for most current bots).

Concerning the actually chosen strategies, Figures 6 to 8 present a parallel coordinate based overview of all games, sorted according to the columns used in Figure 5. Next to the fraction of time the 5 strategies have been employed, the first three columns show the wins, the overall game time (rescaled in seconds and the number of strategy changes (rescaled). For the games against BTHAI/Terran, BTHAI/Protoss, and AIUR, the Templar attack strategy is clearly dominating, whether for the games of FABTHAI against itself (Figure 7), the ground attack strategy is chosen much more often, which clearly demonstrates a very different behavior. If FABTHAI plays against the default Protoss AI (Figure 8), the ground attack strategy is even more pronounced. Whether in the two first figures, the air and anti-air strategies were very rarely employed, this is different here: anti-air is chosen much more often, especially for the Athena-2 map. This makes sense because the map contains plateaus and the default AI therefore builds air transport and combat units to reach these. Summarizing, we can state that Templar attack and ground attack are by far the most important strategies, and air attack is to some extent important, depending on the map. The air strategy is of minor importance, and the rush defense is never chosen.

V. CONCLUSIONS
We have presented a fuzzy adaptive BTHAI version that is able to switch between prepared strategies during the game according to the current situation. The experimental analysis shows that this switching is indeed adaptive and the employed strategies seem to be reasonable with respect to the current conditions. The direct encounter between FABTHAI and BTHAI is clearly dominated by the FABTHAI, however, FABTHAI it is not able to reach the playing strength of one of the currently best rated bots, here represented by the AIUR bot. Although the number of games played and the number of maps employed does not allow for statistically hard statements, the trends seem clear: The fuzzy adaptive strategy choice works well and provides an improvement over the static strategy choice of the original BTHAI.

As the system is configured by text files, it is easily extendible, and the numerical values of the conditions could also be optimized by an external tool, e.g., an Evolutionary Algorithm. We do not claim that the 5 currently provided strategies are in some sense optimal, it was our task to show that adaptive strategy choice can be implemented by means of fuzzy control and provides an interesting strategy reasoning add-on that may also be used in other RTS bots.

REFERENCES