MATEC Web of Conferences **68**, 18001 (2016) DOI: 10.1051/matecconf/20166818001
ICIEA 2016

Evaluating the Improvements of *StarCraft* Gameplay in the ABL Agent EISBot by Implementing Dynamic Specificities through an External Component

Jared Okun

Byram Hills High School, USA

Abstract. I studied how artificial intelligence (AI) adapts to failure. I worked with AI framework ABL, which uses a behavior tree. The tree begins by creating behaviors to accomplish an AI goal. Behaviors are chosen based on priorities. However, these priorities are not effective in all situations, and cannot change in ABL, which prevents the AI to learn from failure. For my study, I gave ABL the ability to change its priorities. This essentially allows an ABL AI to adapt from its mistakes. This ability was then tested for effectiveness using the ABL AI EISBot (an AI that plays videogames) and the game of *StarCraft*. The new ability caused EISBot to improve significantly showing that the ability to change priorities improves an AI's performance.

1 Introduction

A main goal of Artificial Intelligence (AI), and intelligent agents more specifically, is to mimic high level decision making such as that of a human in computer systems [1]. Games are useful test beds for assessing our AI approaches because the actions of AI players (agent), and their results are observable in real-time. My study used Real-Time Strategy (RTS) games, where the player acts as an off-field general for an army, usually with the goal of destroying the opponent. The genre of RTS games is of particular interest because it requires an agent to exhibit multi- scale reasoning and concurrent planning, which requires learning and recognizing an opponent's plan from their observed actions, along with spatial and temporal reasoning [1]. Expressive Intelligence Studio Bot (EISBot) is an example of such an agent [1]. EISBot plays the RTS game *StarCraft: Brood War* and attempts to emulate human-level reasoning with its unique decision-making process [1].

EISBot's decision-making process is implemented in A Behavioral Language (ABL) using an Active Behavior Tree (ABT). The ABT contains all the goals that EISBot is pursuing. An example of an ABT can be seen in Figure 3 in Section 4.2. Examples of these goals could be to collect a specific amount of resources or train a certain number of units. To attain these goals, open leaf nodes are used, which are behaviors available from a hand-authored behavior library within EISBot that are selected to be enacted by EISBot based on a priority that is determined by the behavior's specificity to complete these goals. These specificities exist within EISBot as integer values. The one with the greatest specificity has the highest priority of being chosen. If there is a tie, the behavior is chosen at random from the behaviors with the highest specificity. These behaviors are comprised of a list of steps that could be mental actions (e.g., mathematical operations), physical actions (e.g., moving units), or actions that create goals which produce more open leaf nodes to accomplish the created goals. Ultimately, the chosen leaf node is expanded into a set of steps that will be added to the ABT in the form of *children* to that leaf node. In other words, the steps of the behavior are children created by the *parent leaf node*. If one of the children steps fails, the leaf node it was expanded from is aborted and the leaf node with the next highest specificity is chosen.

However, there is a problem with this method. The priorities are not changed after a failure has occurred; hence, the node with the next highest priority may contain similar actions that caused the failure of the previously selected node, thus causing the leaf node to likely fail again. Obviously, this is detrimental to winning the game. While EISBot spends time by continuing to try to accomplish the desired goal, the opponent may use this time to set up and strengthen their army. This was demonstrated in the 2011 game of EISBot versus a well-known professional player that was documented in a video recording on YouTube [2]. EISBot attacked the human player with dragoons, which are offensive units in *StarCraft*, and in response, the human player produced hydralisks and zerglings, which are units that counter dragoons. After again attacking the human player with dragoons, EISBot's army was destroyed by the human player's hydralisks. However, EISBot still continued to primarily produce dragoons that attacked the human player and failed, giving the human player the time to produce even more hydralisks and zerglings. Ultimately, EISBot lost the game to the human player's army of hydralisks and zerglings. In other words, EISBot lost that game because it did not respond to the human player creating units that countered EISBot's army. Importantly, if EISBot had the ability to "learn" from its failure, then it might have chosen to cease building dragoons and instead produce units to counter the human player's army. Repeating useless behavior is counterproductive to EISBot's success. The cause of this problem is that the priorities of EISBot's nodes remain stagnant throughout the game. These priorities do not change even after a failure has occurred. Unlike EISBot, humans will normally learn from mistakes and try not to repeat what caused the mistake. To address this problem, I proposed a solution: I modified EISBot's source code to automatically reduce the priority of leaf nodes based on their similarity to the one that failed. These revisions will be based on the steps that the nodes and failed leaf nodes both contain. These changing priorities are known as *dynamic specificities* or *dynamic priorities*.

In my study, EISBot was edited to allow the use of dynamic specificities. This new agent is called *EISBot Dynamic Specificities* (EISBotDS). I tested EISBotDS against EISBot in the game *StarCraft: Brood War* by playing them both against the pre-existing StarCraft agent in a pre-made scenario and comparing their number of wins over time. This is the first study on adding dynamic specificities to ABL, as ABL does not support them. The implementation of dynamic specificities into EISBot will future agents created with ABL to use dynamic specificities, which should increase their performance.

2 Hypotheses

H_0: EISBot and EISBotDS will record the same total wins in the scenario
H_1: EISBotDS will record more total wins in the scenario than EISBot

3 Objectives

1. Implement dynamic specificities into ABL so EISBot can use dynamic specificities.
2. Create a controlled scenario that both agents will play to determine whether implementing specificities has a positive effect on EISBotDS's ability to play *StarCraft.*
3. Test EISBotDS against EISBot with the scenario and compare total wins.

4 Methods

I created this study under the tutelage of my mentor, Dr. David W. Aha of the Naval Research Laboratory's Navy Center for Applied Research in Artificial Intelligence in Washington, D.C. He introduced me to Dr. Ben Weber's agent, EISBot, which plays *StarCraft: Brood War*. After reading Weber's dissertation on the subject and watching it play for a period of a few months, I began to focus on its ABL implementation. I decided I wanted to begin a project where I would improve upon EISBot's method for responding to failing behaviors. For this study, I created an implementation of dynamic specificities and the test scenario with the help of Michael Leece and Dr. Mark (Mak) Roberts, who are collaborators with Dr. Aha. I also ran the experiment and performed the statistical tests reported below.

4.1 StarCraft

StarCraft and its expansion, *StarCraft: Brood War*, are futuristic RTS games developed by Blizzard Entertainment in 1998. Unlike many video games, *StarCraft: Brood War* is played at a competitive level due to its fast, deep, and unforgiving gameplay [1]. This game is used in AI research because gameplay requires many aspects that are needed in human-level AI, and actions in game are observable and can be repeated.

In a standard game of *StarCraft*, the goal is to destroy all of your opponents' structures. Each player has the choice of three races at the start of the game. These races are the *Terran, Zerg,* and *Protoss*. Each race has its own unique units, structures, and mechanics. For example, the *Zerg* and *Protoss* have limited areas where they can construct structures while the *Terran* can make structures on any level ground as long as there is room. Mastering any race in this game requires a lot of practice for a human player. The races are considered to be balanced as all races are viable competitively.

In a game of *StarCraft*, a player performs multiple tasks. These include producing an army to counter an opponent, micromanaging each unit in the army to make sure they are doing their optimal work, expanding their race's tech tree to make the best units and upgrades, and managing their resources to be able to produce units and upgrades. Since this game is also multiplayer, the player needs to keep track of his opponent. If the player does not know what his opponent is doing, he will be unable to prepare and effectively counter the opponent's strategy. Figure 1 shows a screenshot of StarCraft's user interface.

StarCraft and *StarCraft: Brood War* have a built-in AI that players can choose to play against. The disadvantage of this AI is that it is inherently random. That is, in the same repeated scenario, it could choose different units to create or attack in different ways.

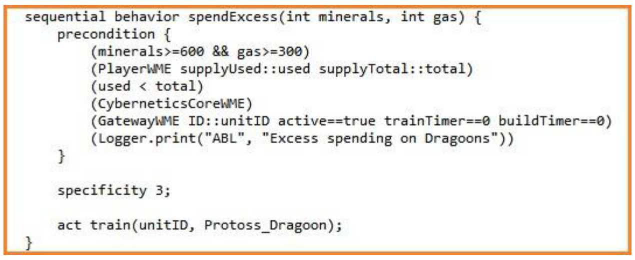
4.2 ABL

A Behavior Language (ABL) is a behavioral language created by Michael Mateas and Andrew Stern in 2004 based on the language HAP used in the Oz Project in order to make life-like agents [3]. One of ABL's most notable advantages is that it allows the construction of multi-scale AI because the agents it creates can pursue multiple goals simultaneously and each pursuit of a goal can communicate with any other pursuit.

Bots made with ABL can be given a set of goals to complete. These goals are then pursued by selecting a behavior from a hand-authored library. In other words, each goal can be achieved by selecting a sequence of actions that was created by the agent's developer.

Each behavior is itself a list of actions that the agent performs. These actions can be (1) physical actions that cause the agent to interact with the medium in which it is acting upon, (2) mental actions that cause the agent to perform mathematical operations or perform external functions, (3) actions that create goals and subgoals, or (4) actions that change elements in the working memory. Each behavior can have specificity, which are priorities that determine what order the behaviors will be run in if they fail. In other words, if a behavior fails, the agent will check the current specificity of the behavior that failed and then choose the next behavior that has specificity less than the current one. If the specificities of multiple behaviors are the same, the behavior that is run among them is randomly selected. Behaviors may also be given preconditions that have to be met in order for the behavior to run in the first place. Behaviors can also have parameters that serve a similar function to the parameters of methods in the programming language, Java. This means that the behaviors can be given variables to be used during execution. Behaviors can also be sequential or parallel. Sequential behaviors run their actions one after another while parallel behaviors run their actions simultaneously. A behavior that was used in this study is displayed in Figure 2.

Figure 1. A screenshot taken in *StarCraft: Brood War*. In the top right corner is the player's resources. The bottom left corner is the minimap. The center screen shows the blue *Protoss* player under attack by a teal *Protoss* player.



```
sequential behavior spendExcess(int minerals, int gas) {
    precondition {
        (minerals>=600 && gas>=300)
        (PlayerWME supplyUsed::used supplyTotal::total)
        (used < total)
        (CyberneticsCoreWME)
        (GatewayWME ID::unitID active==true trainTimer==0 buildTimer==0)
        (Logger.print("ABL", "Excess spending on Dragoons"))
    }

    specificity 3;

    act train(unitID, Protoss_Dragoon);
}
```

Figure 2. An example of the spendExcess behavior in EISBot's behavior library. This behavior is sequential and contains a precondition, a specificity, and a physical action. [4].

All active behaviors are stored in the Active Behavior Tree (ABT). During each execution cycle, goals that are not being pursued are added to a set. For each goal, a behavior is chosen to be executed based on the specificity and preconditions of that behavior. This behavior then adds its actions to the tree as children of the behavior. An example of an ABT is shown in Figure 3.

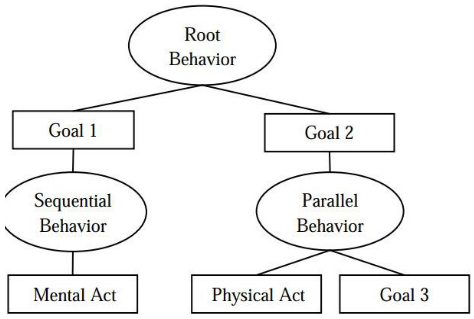


Figure 3. An abstract example of an ABT. The behavior 'Root Behavior' was a parallel behavior with two goals. The behavior 'Sequential Behavior' that was chosen to pursue 'Goal 1' executes a mental action. The behavior 'Parallel Behavior' that was chosen to pursue 'Goal 2' executes a physical action and creates another goal.

One of ABL's advantages for use in game AI is the scheduling and plan monitoring of the AI can be handled by the language itself without the need for external software [1]. This is most relevant for physical actions as some require multiple frames in the game to run. These physical actions will be set to executing, and any actions after that in a sequential behavior will be unable to run ABL has its own mechanism for handling failures while the code is running. If a behavior fails for any reason, the behavior and its actions are then removed from the tree. The goal that the failed behavior was pursuing will once again be available to be pursued. The behavior chosen to achieve this goal cannot be one that had been previously chosen and failed. However, the next behavior chosen can be similar to the one that failed.

4.3 EISBot

EISBot (Expressive Intelligence Studio Bot) is a AI software system that was developed by Dr. Ben Weber while he was with the Expressive Intelligence Studio at the University of California, Santa Cruz, and it plays the game, *StarCraft: Brood War*. EISBot was developed using a combination of Java and ABL. EISBot interfaces with *StarCraft* through the use of JNIBWAPI, Control Process, and ProxyBot [1].

Table 1. This lists the purpose of some of the managers in EISBot. I used behaviors from the Strategy manager in this study.

Manager	Purpose
Strategy	Responsible for choosing strategy and the timing to execute actions
Tactics	Responsible for combat and micromanagement of units
Worker	Responsible for resource income, workers, and expansion
Construction	Manages requests for construction
Supply	Manages the supply resource. This resource is required to build more units in *StarCraft: Brood War*
Build Order	Manages the order in which structures are built, units are trained, and upgrades are researched

EISBot is composed of multiple managers that are associated with different aspects of play in *StarCraft* [1]. Its managers include: Strategy manager, Tactics manager, Worker manager, Construction manager, Supply manager, and Build Order manager [4-5]. The purpose of each relevant manager is listed in Table 1. The managers are all interfaced with the various components and sensors that are related to the work they perform. Each of these interfaces also determines how working memory elements are edited by each manager.

Working memory elements are the primary source of communication of the various managers. Working memory elements can also act as a blackboard for the managers and external components to post information about the current game such as the opponent's predicted strategy [1]. The external planner of EISBot also uses working memory. The planner adds to working memory, which causes actions and behaviors to activate in the ABT.

4.4 Implementation

In order to add dynamic specificities to EISBot, I had to bypass ABL's normal specificities because they are static. I did this by creating a text file called "buildSpecificities" that contained the specificities and a Java class called "SpecificityEditor" that had methods to interact with the ABL code. This study specifically worked with the spendExcess behaviors from the Strategy manager in EISBot.

SpecificityEditor contains local variables that corresponded with the specificities of the relevant spendExcess behaviors. The file buildSpecificities also contains values that corresponded with the specificities of the spendExcess behaviors. Upon a new instance of SpecificityEditor being created, the variables in SpecificityEditor would be set to the values in buildSpecificities. The current behavior that EISBotDS was executed is assigned to a variable, currentBuild, in SpecificityEditor. I created a method that rewrites the buildSpecificities file with new specificities was also added to SpecificityEditor. I also added a static SpecificityEditor object that refers to the SpecificityEditor used during each run of EISBotDS.

Implementing SpecificityEditor with ABL required making the specificities of the spendExcess behaviors equal. This allowed EISBotDS to choose between these behaviors "randomly." To prevent the behaviors from being chosen randomly, each behavior was given an extra precondition. This precondition used the getters from SpecificityEditor to emulate the function of specificities. In other words, this precondition requires the behavior that is chosen to have the highest specificities among the other specificities that can be chosen. A mental action that sets the value of currentBuild in SpecificityEditor to correspond with the behavior being executed was also added to each of these behaviors.

The specificities are currently set to change at the end of each game that EISBotDS plays. EISBot already has a function that returns the victor of the game. Using the value that this returns, SpecificityEditor rewrites buildSpecificities with new values for the specificities. If the game was won, the last behavior chosen would have its specificity increase by one. If the game was lost, the last behavior chosen would have its specificity decrease by one. I chose to change the last behavior as EISBotDS will use only one behavior in each scenario if that behavior has the highest specificity. If there is a tie, EISBotDS will be choosing its behaviors randomly and therefore the specificity it chooses to lower will also be random. EISBotDS was not able to change its specificities in the middle of the scenario.

5 Experimental Setup and Results

For this study, EISBot was integrated with the ability to use dynamic specificity for its spendExcess behaviors. In other words, the only action that changes between EISBot and EISBotDS is the units that they are building. In order to perceive if this change affects EISBot, I tested EISBot and EISBotDS by playing them on a scenario 100 times. This scenario required the agents to defend the base against a wave of enemy units controlled by *StarCraft*'s built-in AI. Since EISBot plays as the *Protoss* and by extension EISBotDS plays as the *Protoss*, both of the agents began with two structures used in the spendExcess behaviors to build units, the various other structures required to advance the tech tree to build all the available units and upgrade them, and sufficient resources. The agents were given two minutes to build and upgrade the units before the enemy wave arrived at their base. The enemy wave would always consist of the same units. The way that the enemy wave attacks is partially random because the built-in *StarCraft* AI is controlling it. The enemy wave could spread out their units, which would cause the unit that EISBot uses to be less effective. On the other hand, the enemy wave could also focus on attacking structures instead of units, which would generally help the agent it is playing against. If the current agent can destroy all enemy units in a wave, it wins. However, like the game, if the current agent loses all its structures, they lose. Because the scenario does not start like a normal match would begin, the Construction manager, the Worker manager, the Supply manager, and the Build Order manager were disabled in both agents.

Table 2. The total wins that EISBot and EISBotDS had at the end of the 100 scenarios

	EISBot	EISBotDS
Total Wins	41	88

For each run of the scenario, the total wins up to that game were collected. The data can be seen in Table 2. EISBotDS had a statistically greater proportion ($p<0.0001$) of wins than EISBot using a two proportion z-test. The win counts of both agents were normalized to the current game number. The results of normalization can be seen in Figures 4 and 5.

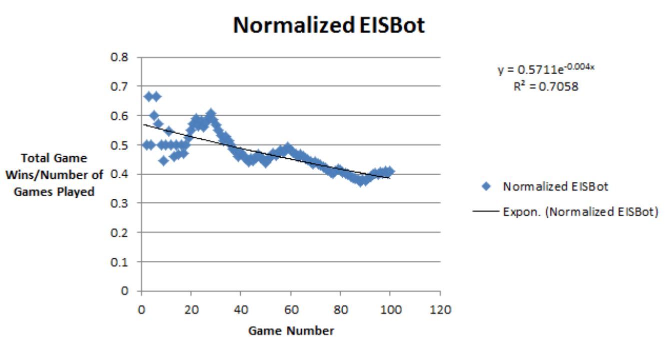


Figure 4. A graph showing the normalized wins of EISBot. The first point of the graph at (1,0) was removed which produced an exponential regression with an R^2 above 0.7.

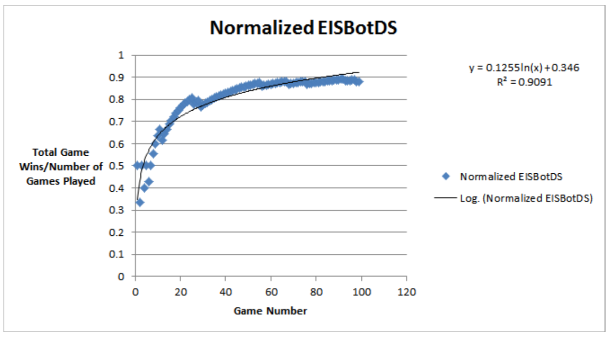


Figure 5. A graph showing the normalized wins of EISBotDS. The first point of the graph at (1,1) was removed which produced a logarithmic regression with an R^2 above 0.9.

6 Discussion

I rejected H_0. The changes made to EISBot to allow it to use dynamic specificities have shown, for the scenario used in Section 5, a significant (p<0.0001) improvement in its gameplay in terms of wins. The results support H_1. Based on both agents' proportion of wins to the 100 games played, it is reasonable to state that EISBotDS is better than EISBot at *StarCraft*, versus the built-in game AI, in scenarios that are similar to the given scenario. From this experiment, there is reason to believe that dynamic specificities will help EISBot succeed at other scenarios with different behaviors chosen. As can be seen in Figure 5, EISBotDS was able to improve over time as its wins to games played ratio approached one over 100 games; in contrast, EISBot's performance decreased, as shown in Figure 4. Because adding dynamic specificities to EISBotDS caused it to perform better at *StarCraft* than EISBot, other creations using the ABL framework may also benefit from the addition of dynamic specificities. This shift to using dynamic specificities could possibly result in a change of the ABL language or the creation of a new language that uses a similar framework to ABL but also has the ability to use dynamic specificities.

This study has implications for ABL agents. All behaviors that were created for these agents had to use static specificities. However, with the use of external components like SpecificityEditor, ABL agents can now use dynamic specificities. As can be seen by the results of the study, giving an ABL agent the ability to use dynamic specificities could cause it to be more effective in the scenarios it is being run in. For example, an external component like SpecificityEditor could be used for an ABL agent that controls a drone in a flight simulation. If the agent is running a behavior with static specificities that resulted in a crash, the agent will continue to crash in subsequent runs. If the same agent was implemented with dynamic specificities, it would run a different behavior to avoid the crash.

This study was limited in that it tested only one set of behaviors with one scenario. Another limitation is that the game AI of *StarCraft* is inherently random and does not act like a human player. This limitation can be fixed by using another external agent like EISBot as the player that controls the units.

7 Conclusion

All of the objectives of this study were met. *This study is the first of its kind to develop an external component to ABL to allow it to use dynamic specificities*. EISBotDS was significantly improved for the given scenario and therefore I accepted H_1. This experiment shows that adding dynamic specificities to ABL can give it the ability to perform better at complex tasks.

In future work, different behaviors will be tested in more scenarios. An example of another behavior that can be tested is the one that chooses how units attack, which would be found in the Tactics manager. The SpecificityEditor class was also specialized for this experiment. The implementation of SpecificityEditor needs to be simpler with more behaviors instead of having to add more methods and variables. The option might also present itself when dynamic specificities can be directly integrated into the ABL framework so an external component is not needed. Another area of future research is modifying EISBotDS to update its specificities in the middle of the game. This would improve EISBotDS's ability to play longer games where similar situations are occurring multiple times. Currently, EISBotDS can only update specificities at the end of the game based on wins. It would be preferable to change specificities in the middle of game so it could change its strategy in the middle of the game. This would require a new metric to determine how to change the specificities instead of wins and where to implement this change in the code.

This research validated the use of dynamic specificities in ABL. These dynamic specificities have applications in other AI that use the ABL framework. The SpecificityEditor class that was developed in this study can also be used in future ABL agents. ABL can be used for more than just games. AI is used for language identification, autocorrect, drones, robots, self-driving cars, searching, and various other applications. ABL with dynamic specificities can be the language that is used to create AI for all of these and should be considered to be used in future AI research.

Acknowledgements

This project would not have been possible without the overarching guidance of my mentor, Dr. David Aha. I would like to thank Michael Leece and Dr. Mark (Mak) Roberts, who helped me when I was working on the scenario and SpecificityEditor. I would also like to thank the science research teachers Mrs. Stephanie Greenwald and Mr. David Keith for pointing me down the right path for my project. I would also like to thank my family and friends for supporting me through this difficult endeavor.

References

1. B. Weber. Integrating learning in a multi-scale agent. *Doctoral dissertation: Department of Computer Science, University of California, Santa Cruz, CA.* (2012)
2. B. Weber [UCSCbweber]. EISBot vs Dennis Fong (Thresh). Retrieved from https://www.youtube.com/watch?v=txGPuN7PXAY &list=UUB0UyZ9THzSjcXaCkGCQLMg&index=14 (2011)
3. M. Mateas & A. Stern. A Beh. Lang. Joi. Act. and Beh. Id. *Like-like Characters: Tools, Affective Functions, and Applications* 135 (2004)
4. B. Weber. EISBot [computer software]. Available from https://github.com/bgweber/eisbot (2012)
5. B. Weber, M. Mateas, A. Jhala. Building Human-Level AI for Real-Time Strategy Games. *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems* (2011)