Using Case-Based Reasoning to Improve Real-Time Strategy Game AI

Daniel Barton

May 8, 2014

Abstract

In recent years using real-time strategy (RTS) games as a test bed for Artificial Intelligence techniques has become increasingly popular. RTS games allow an environment that is complex enough to present the AI with a sufficiently challenging situation. These games also provide an easy way to evaluate the effectiveness of the technique. In this paper, I will describe how I built on the work of Aha et al., who in [2] used a custom Case-Based Reasoner to govern an AI in Wargus, an open source RTS game, that learns to play better as it plays more games. I replicated their experiment with a different Case-Based Reasoner. I created and worked with BARC, a Case-Based Reasoner written in C++ for this specific application. BARC is based on the description of CAT in [2], but shares no source code.
1 Introduction

Real-Time Strategy (RTS) is a genre of video games in which the player controls a group of units in order to achieve an objective, typically to conquer all of the other players. The units a player controls can range from buildings to military units. Each of the units may act towards goals independently of one another, but some units may require other units to perform actions before they can achieve their goal. Sometimes a particular type of unit cannot be created until a certain building has been constructed. All of these intricacies make the problem of creating an intelligent AI difficult.

In addition to the complexities of the game, three development aspects have made creating an AI difficult as well. First, the AI development stage generally has a smaller time frame than other pieces of the game. Second, level design is completed independently of creating the AI even though the AI must work within the environment created by the level designers. Finally, as the game progresses through development, many aspects of the game will be changed. New pieces will be added, and other pieces will be removed. This could mean that an AI developed near the beginning of the project will be playing a different game by the time the release version is ready [9, p.1].

The development constraints and complexity issues have led to many otherwise good RTS games having poor AIs which distract the user from the game. Some in the academic world have begun applying machine learning techniques to create more intelligent RTS AIs. In this environment, the development constraints are not a concern, and the researchers can focus on the AI itself. In this paper, I will focus on the use of Case-Based Reasoning (CBR), an AI technique based in psychology, in which the AI uses previous solutions to solve new problems to improve RTS game AI, specifically in the game Wargus. Wargus takes place in a medieval fantasy world of Azeroth where humans and orcs are at war. The player takes control of the humans or orcs, and in each game fights against an AI controlling the other team. Each player constructs buildings, researches upgrades, and trains units to gather resources and attack their opponent. A player wins by destroying all opposing buildings and units. For my purposes, a player will also win if they have the highest score (as calculated by Wargus) at the end of 20 minutes of gameplay. A player’s score is increased by destroying enemy units. Destroying stronger units will raise a player’s score more than destroying weaker units. The time limit will keep games from going on for hours, reducing testing time significantly, while allowing for enough time to demonstrate the effectiveness of the AI.

The rest of this paper will be organized as follows. First, I will discuss the historical background of the problem, previous work in the area, my contribution, and the motivation of the problem. Next, I will describe the problem and fully define CBR. After this, I will discuss the implementation details of this project, and then the methodology for testing the BARC AI. Finally, I will detail opportunities for future directions.
1.1 Historical Background

RTS games have been around for decades, but due to the aforementioned problems, the AI in the games have used the same strategies. One of the most commonly used strategies is allowing the AI to cheat by obtaining resources faster, making units cost less, or making units train faster [9, p.1]. This approach frustrates players and makes the AI very predictable, lessening the entertainment value of the game. As a result many players prefer to play against other humans instead of the AI[5, p.36].

1.2 Previous Work

Much of the interest in RTS AI is a result of Buro and Furtak’s work in 2003, where they outlined research possibilities in AI in RTS games. They also gave an update on their ORTS (Open RTS) project, an environment for running RTS AI experiments [4, p.5].

That same year Michael Fagan and Pádraig Cunningham published a paper on using Case-Based Reasoning with the arcade game Space Invaders to predict the user’s moves [6, p.1]. This was one of the first works on using CBR in video game AI.

In 2004 Danny Cheng and Ruck Thawonmas created an RTS AI with case-based plan recognition to minimize the predictability of the AI. Their goal was to get the AI to predict the user’s plan [6].

Marc Ponsen submitted his Masters Thesis in 2004 as well, which covered improving game AI with evolutionary learning. Ponsen’s thesis, while not using CBR, dealt extensively with the Stratagus framework. He edited the AI so that it could adapt to situations on the fly [8].

In 2005, David Aha, Matthew Molineaux, and Marc Ponsen published a paper in which they applied CBR to an RTS AI in Stratagus. My work extends their research. They created a custom CBR (called CAT) to play Wargus, which is the basis for BARC. Many of the tactics in each game state for CAT\(^1\) were generated using a genetic algorithm from Ponsen’s Masters Thesis [8], and some were created by students of the authors. They wrote software to connect the AI to Wargus, and to automate games between CAT and other AIs. They were able to train their AI to win more than 80% of its games consistently [2, p.11]. They proved it was possible to successfully apply CBR techniques to RTS games to govern the actions of the AI.

In 2006, Ponsen returned to his thesis topic in working with Pieter Spronck, Ida Sprinkhuizen-Kuyper and Eric Postma on using adaptive AI techniques on a commercial game. This paper explored the topic of using academic AI techniques on a full scale commercial game [11].

The next year Sánchez et al. published a paper on implementing CBR with jCOLIBRI on turn-based strategy games. Their work demonstrates how the jCOLIBRI system may be integrated into an AI for a strategy game [9].

\(^1\)states and tactics will be discussed later
Also in 2007, a team out of Georgia Institute of Technology published a paper on transfer learning and CBR/RL (Reinforced Learning) systems in RTS games. This group outlined many important representations of cases which will apply to my work. Their work involved creating a hierarchical CBR structure with the AI to deal with higher level decisions, and then gave instructions to the lower level operations [10].

In 2008 Robin Baumgarten, Simon Colton, and Mark Morris published a paper on combining AI techniques into a single bot. The bot combined CBR with several other features to create a hybrid AI [3].

My contribution is a proof-of-concept of the work of Aha et al. in 2005. I implemented an AI for Wargus from scratch, along with constructing the Case-Based Reasoner (BARC) from scratch. The Case-Based Reasoner was constructed based on the description of CAT in [2].

1.3 Motivation

When working on RTS game AI for academia, the motivation is rarely to create an AI which will be more entertaining to play against. Rather, the RTS game environment gives an interesting platform to test AI techniques, as it involves multiple complex problems. Some of these problems are [4, p.2-4]:

- **Adversarial real-time planning.** The environment in an RTS game is constantly changing, and the AI must be able to execute a high level strategy in the changing environment.

- **Decision making under uncertainty.** In an RTS game, there are many uncertainties the AI will face. Enemy movements are not known, and the map may be unexplored, meaning the AI will have to make decisions on what to do without complete information on the problem.

- **Opponent modeling, learning.** This involves identifying weaknesses in the opponent’s strategy and exploiting them in order to beat the opponent.

- **Spatial and temporal reasoning.** The AI will need to be able to wisely plan out a group of buildings with limited space, as well as choose the best times and places to attack.

- **Resource management.** One of the largest pieces of an RTS game is economy management. The AI must be able to decide what is the best use of its resources at any given time. This includes when it is time to upgrade a building versus buying a new one, when it is time to invest in military units versus civilian units, and careful allocation when resources are running low.

- **Collaboration.** Many AI’s do not work well together when placed on a team against a set of humans. They act independently instead of working together against the human.
• **Pathfinding.** Pathfinding is one of the most obvious aspects of AI, as the user will also see this directly when his/her units move. A pathfinding algorithm that leads units into walls or corners can leave a user frustrated and breaks any illusion of intelligence in the game.

These are all problems which also exist outside the scope of RTS games. Many of them are related to the robotics field, and many have military applications [4, p.4]. Thus the motivation for working on RTS game AI is less of one to create more entertainment value, although this certainly plays a role in some of the research, but more of one to find and test new methods to solve the aforementioned problems. RTS games provide a complex enough environment to adequately test AI techniques, and give clear results of experiments, as either the AI won or the AI lost.

## 2 Problem Definition

My work addressed four of the problems mentioned in the list above: adversarial real-time planning, decision making under uncertainty, spatial and temporal reasoning, and resource management. These can all be thought of as higher level decisions made by the AI. My work did not deal with lower level AI, or individual unit AI (e.g., troop pathfinding, worker reaction to being attacked, etc). One of the most challenging problems with addressing the decision making of the AI is that the decision space, or number of possible actions, grows exponentially with the number of workers and troops. The decision space can be defined with the following equation [2, p.5]:

\[
O(2^W \times (A \times P) + 2^T \times (D + S) + B \times (R + C))
\]

Where:
- \(W\) = current number of workers
- \(A\) = assignments workers can perform
- \(P\) = average number of workplaces
- \(T\) = number of troops (fighters plus workers)
- \(D\) = average number of directions that a unit can move
- \(S\) = choice of troop’s stance
- \(B\) = number of buildings
- \(R\) = average choice of research objectives at a building
- \(C\) = average choice of units to create at a building

After a short period of gameplay, the amount of moves grows too large to consider all possible options. One of the first issues of developing a high level AI is limiting the decision space. This will be accomplished by splitting the game into states, as described in [2]. The game states will be divided based on what buildings are currently built. Since what buildings can come next, what units can be trained, and what can
be researched all depends on the current buildings, splitting the game into states based on this sufficiently limits the decision space.

The state lattice is shown in Figure 1. Each state has a set of tactics to choose from to execute while in that state. Each tactic will lead to a different subsequent state. The labels on the transitions between states are what tactic takes the game from the current state to the attached next state. A transition is completed when the transition building is constructed. Note that some buildings such as farms, towers, and several others do not cause state transitions.

When the decision space is limited, the problem becomes choosing the best tactic given the current state. The best tactic will be defined as the one that will increase the likelihood of the AI attaining victory the most. Case-Based Reasoning will be used to determine which tactic is the best at the time by pulling from a knowledge base of other situations faced by the AI.

In solving the problem of which solution is best the AI would also solve the four problems mentioned above. It would solve adversarial real-time planning by changing the AI’s plan depending on the changing environment because as the game situation changes the AI may choose a different solution. Decision making under uncertainty is addressed by having the AI select the best solution without knowing everything about the world and the opponent’s next example. Selecting the best solution also solves spatial and temporal reasoning by learning what are the best times to attack. Although resource management would not be directly addressed in that the selected tactic would not be dependent on the current economic situation, the AI would have to gather and spend resources just as a human player would. Thus, the important problem is, “What is the best decision for the AI to make at this time to put itself in a position to win the game?”

2.1 Case-Based Reasoning

Case-Based Reasoning is a machine learning technique partially based on psychological studies of how humans make decisions [1, p.2]. The idea behind CBR is that when making decisions, we remember previous life experiences, and then reapply solutions that worked in previous situations. If it is a situation that has not yet been encountered, we often recall similar past experiences, and then adjust the solutions to those in order to make them better fit the current situation. For example, if a doctor sees a patient who has similar symptoms to a patient that had been treated a month ago, the doctor can reapply the technique that worked last time to this situation. Or if a mechanic gets a car that has a similar problem to a truck he recently repaired, he can reuse his previous solution. The fix may not be able to be directly applied because of the differences between the vehicles, forcing the mechanic to adjust the previous solution in order to fix the car.

Similar to the doctor and the mechanic, the Case-Based Reasoner takes a current situation, compares it to a knowledge base of previous cases, decides which to apply
Figure 1: The state lattice that will be used to limit the decision space. Labels on state transitions mark the tactic that is employed at the origin state to transition into the destination state. Building abbreviations are: Th=Townhall, Ba=Barracks, Lm=Lumbermill, Bs=Blacksmith, Kp=Keep, St=Stables, Ca=Castles, Ap=Airport, Mt=Magetower, Tm=Temple.

and adjusts it, and then adds the new case to the knowledge base. This process can be summed up in four steps: retrieve, reuse, revise, retain [1, p.8]. These will be described in further detail in the following sections.
2.1.1 Retrieve

The retrieval step involves getting the cases which are relevant to the given problem. This typically involves two steps: retrieving good cases, and then selecting a subset of the best cases [7, p.18]. In the first step, the reasoner selects previous cases which match closely to the current situation. In the second step, the reasoner takes the matches and selects one or more to be the best cases to move forward with [7, p.19].

2.1.2 Reuse

In the reuse step, the reasoner creates a preliminary solution by taking the case or cases that were selected in the retrieve step and selecting all or part of their solutions. It then combines these to create the ballpark solution [7, p.20].

2.1.3 Revise

In the revise step, the reasoner takes the proposed solution from the reuse step and adapts it to fit the current situation. This is also typically done in two steps: deciding what to adapt and applying the adaptation [7, p.21].

2.1.4 Retain

In the retain step, the reasoner saves the case into the knowledge base. The reasoner also saves the effectiveness of the solution, generally a measure of success for the given problem the reasoner is solving, along with the case so that it will know in the future if the solution worked. Some reasoners also save information on how a failed solution may be fixed, or why a successful solution worked. [7, p.23].

3 Implementation

The project goals are (in order of completion):

1. Create a custom CBR (BARC) to be used with Wargus.
2. Integrate the CBR with the Wargus AI.
3. Test the CBR AI against other AIs and human opponents and track its progress.

In order to complete these goals, the implementation will require a game engine to work on top of, and a Case-Based Reasoner to handle decision making.
3.1 Game Engine

The game engine is the framework upon which the RTS game is built. The engine handles many of the essential pieces of the game, such as graphics, collision detection, unit movement, resources, and unit integration. The option to not use a game engine and build everything from the ground up was briefly considered, but the amount of time that would be required to accomplish that is far beyond the scope of this project, and would likely not be completed in time. Many engines were explored, but only three were seriously considered because they were the three that were most robust and best documented. These three are GameMaker Pro, Spring RTS, and Stratagus.

3.1.1 GameMaker Pro

GameMaker Pro is a commercial game engine developed by YoYo Games™. It is one of the few engines that was considered that is not an open source project. GameMaker was very user friendly, but too limiting in its scripting for the purposes of this project. Most of the scripting was event driven, and was in a custom language that would not interface well with outside tools.

3.1.2 Spring RTS

Spring RTS is an open source engine which has several open source games released with it to be edited. When using Spring, the user can select which map, AI, and Spring-compatible game to play. This made it very tempting, as testing a custom AI against other AIs would be very easy. Unfortunately, much of the Spring documentation was outdated, and the code appeared to be in the process of transitioning to an object oriented approach, but significant portions of it was still running the non-updated code, leading to a confusing network of poorly maintained and documented code.

3.1.3 Stratagus

Stratagus is an open source engine that was used in several of the papers. It was extremely easy to set up and get running. The Stratagus based game that will be used is Wargus, based on Blizzard’s Warcraft II: Tides of Darkness. The AI scripting is done in Lua in Stratagus. The engine itself is written in C++, which interfaces easily with the Lua scripts. Stratagus’s documentation was up to date and fairly extensive, particularly with the Lua scripting portions. Stratagus also has an active developer base, with forums and walk-throughs.

3.2 Case-Based Reasoner

The Case-Based Reasoner is the framework which will implement the Case-Based Reasoning. It will handle saving the case-base, the retrieving of appropriate cases, and updating cases. The CBR described in [2], and the one my project aims to replicate, is
slightly different than the traditional CBR technique described in Section 2.1. This is due to the immense decision space of the RTS game discussed in Section 2. A typical CBR in this context would look at the game state, and select the next move which would most increase the chance of winning based on previous experiences. But the number of moves the AI would have to select from would grow at an exponential rate throughout the game, making this approach not feasible. Thus, the CBR for the AI splits the game into the states described in Figure 1 to limit the decision space, and then selects a tactic from that state. The differences between traditional Case-Based Reasoning techniques and the requirements for the Wargus CBR meant that many of the freely available CBR frameworks online would not be viable options. Therefore, I decided to create my own CBR framework, named BARC (Barton CBR). BARC was originally written in Java and was going to link to Wargus using LuaJava, but because LuaJava required that the Java program start the Lua program when I needed Lua to start the Java program, I decided to rewrite BARC in C++ and use Lua calls to start it. The majority of the equations and descriptions in this section are based off of CAT in [2].

In BARC, each state has its own set of associated tactics. Cases in this context map a game description (see Table 1) upon entering a new state to a tactic based on how the tactics have performed in similar situations before. The tactics are never edited, and once a case has been selected for a state the tactic is executed until the next state is entered. Each case is defined by four items:

\[ C = \langle \text{State, Tactic, Description, Performance} \rangle \]

Where:
- **State** = Current Building State (From the state lattice described above)
- **Tactic** = Selected Tactic
- **Description** = 8-tuple of information describing the game situation (described in Table 1)
- **Performance** = Value between 0-1 describing how well the case performed (described in Section 4.3)

A game of Wargus with BARC controlling the AI begins with the Lua AI script, which in turn starts up the BARC executable files. This initializes BARC by reading in the case base from a .barc file (see Appendix B for the .barc file definition), allowing the cases and tactics in their associated game states to be remembered between runs. BARC runs the initial tactic, which leads to the creation of a Town Hall and a Barracks. Upon completion of these buildings, the AI begins a state transition. On a state transition, game information is sent from Wargus to BARC and is saved in the case for performance evaluation at the end of the game. This is started by a call to a C++ function from the end of the Lua AI tactic loop (see Appendix A for more details on AI implementation in Stratagus). Then, a new case is selected or created, and the Lua tactic script for the case is selected to be used. Once the tactic has been
selected, Wargus will run the selected AI script, which will have another call to BARC at the end of the loop to continue the cycle. This will continue until the end of the game, when BARC will go through the states it used and update the performance of the selected cases. When the game ends, BARC writes out the number of cases and the scores of itself and its opponent to a CSV file. This file will be used to analyze BARCs performance. The details of the individual steps are listed below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>∆Kills_{i−1}</td>
<td>Number of opponent combat and worker units killed in the last state minus own killed in last state</td>
</tr>
<tr>
<td>∆Razings_{i−1}</td>
<td>Number of opponent buildings razed (destroyed) in last state minus own razed in last state</td>
</tr>
<tr>
<td>Buildings_o</td>
<td>Number of opponent buildings ever created</td>
</tr>
<tr>
<td>CombatUnits_o</td>
<td>Number of opponent combat units ever created</td>
</tr>
<tr>
<td>Workers_o</td>
<td>Number of opponent workers ever created</td>
</tr>
<tr>
<td>Buildings_{p,i}</td>
<td>Number of own buildings currently existing</td>
</tr>
<tr>
<td>CombatUnits_{p,i}</td>
<td>Number of own combat units currently existing</td>
</tr>
<tr>
<td>Workers_{p,i}</td>
<td>Number of own workers currently existing</td>
</tr>
</tbody>
</table>

Table 1: Values in a game description, taken from [2]

3.2.1 BARC Retrieval

Upon entering a state, BARC selects which case to implement using one of two methods. First, it checks to see if all of the tactics at the current state have been used a certain number of times, referred to as the exploration threshold. This ensures that each tactic is used several times and that all options are explored. If there are multiple tactics which have not been used enough times, one of the tactics is randomly selected and used. For now, the exploration threshold has been set at three, which is large enough to ensure each tactic gets tested multiple times, but small enough that the training period (when the CBR is in exploration and not selecting the best performing case) is not prohibitively long.

If all of the tactics have met their exploration threshold (that is, the CBR is no longer in its training period), BARC selects a subset of the available cases using a k-nearest neighbors algorithm. The similarity function between a case \( C \) and a game description \( D \) is as follows:

\[
Sim(C, D) = (C_{\text{performance}} / \text{dist}(C_{\text{description}}, D)) - \text{dist}(C_{\text{description}}, D)
\]

In the equation, dist() is the distance among the 8 values. This takes into account both the similarity to the game description, along with giving weight to the performance of the cases. Once the set of neighbors has been determined, the tactic of the case with the highest performance is selected to be used. If the highest case has a performance of less than .5, BARC selects the least used tactic and employ it.

Throughout the Retrieval stage, if a case already exists with the exact game description and the chosen tactic, then the existing case is used. Otherwise, a new
case is created with the given game description and selected tactic. Whether the case is selected or created, its performance is not be updated until the end of the game.

### 3.2.2 BARC Reuse

Since the tactics are not edited, BARC does not adapt the cases to the current situation. A small amount of adaptation takes place (building placement, etc) when the game engine executes the instructions given by the tactic.

### 3.2.3 BARC Revision

At the end of the game, the performance value of a case is revised. Since a case is only used once in a given game, it is alright to not update the performance until the end of the game because it will not need to be referenced until a subsequent run of the game. The performance value is the only piece of the case which is revised, as again the tactic is never edited.

The performance equation used for BARC is the same as the one described in [2]. The performance of a case $C$ from state $b$ is measured by both its local (during execution in $b$) and global (from beginning of execution in $b$ until the end of the game) impact on game score. The equation used is:

$$C_{\text{performance}} = \frac{\sum_{i=1}^{n} C_{\text{performance},i}}{n}$$

$$C_{\text{performance},i} = \frac{1}{2}(\Delta \text{Score}_i + \Delta \text{Score}_{i,b})$$

$$\Delta \text{Score}_i = \left(\text{Score}_{i,p} - \text{Score}_{i,p,b}\right)/\left((\text{Score}_{i,p} - \text{Score}_{i,p,b}) + (\text{Score}_{i,o} - \text{Score}_{i,o,b})\right)$$

$$\Delta \text{Score}_{i,b} = \left(\text{Score}_{i,p,b+1} - \text{Score}_{i,p,b}\right)/\left((\text{Score}_{i,p,b+1} - \text{Score}_{i,p,b}) + (\text{Score}_{i,o,b+1} - \text{Score}_{i,o,b})\right)$$

Where:

- $n =$ number of games $C$ was selected
- $\text{Score}_{i,p} =$ player’s WARGUS score at the end of $i^{th}$ game in which $C$ was used
- $\text{Score}_{i,p,b} =$ player’s score before $C’$s tactic is used in game $i$
- $\text{Score}_{i,p,b+1} =$ player’s score after $C’$s tactic executes and next state begins
- $\text{Score}_{i,o} =$ opponent’s score at end of $i^{th}$ game in which $C$ was used
- $\text{Score}_{i,o,b} =$ opponent’s score before $C’$s tactic is used in game $i$
- $\text{Score}_{i,o,b+1} =$ opponent’s score after $C’$s tactic executes and next state begins

### 3.2.4 BARC Retention

At the end of a game, BARC saves out the current knowledge base to a .barc file so that the knowledge base will persist through runs. On initialization, BARC reads in the information from the .barc file to populate the knowledge base. Cases are always saved to the knowledge base, and once saved they are never deleted from the knowledge base.
4 Methodology

BARC was tested with human participants playing against the AI in one on one matches. The matches were won when one player destroyed all of the units of the other player, or at the end of 20 minutes. If the game reaches the time limit, the player with the higher score wins.

Using human versus BARC for the testing poses several problems in evaluating BARC. First, many of the people who took part in the study were inexperienced players of not only Wargus, but of the RTS genre in general. This led the AI to have some early wins that would not have happened if it were playing a built-in AI. Another major problem when trying to decide if BARC is learning to play better is that humans learn to play better as well. If BARC were playing against another AI, the opponent would be consistent each time, making it easy to see if BARC was getting closer to winning and learning how to beat the opponent. Human opponents will vary their strategy each time, making it harder to gauge results because BARC could be improving because it is learning to play better, or it may be improving because the human is making poor decisions. Humans playing better as time goes on can also make BARC look as though it is getting worse with time. As more games are played, the new human players will learn and improve, possibly beating BARC in later rounds even if BARC is making better decisions. Finally, humans require a slight handicap when facing BARC. BARC can issue commands in a fraction of the time it takes a person to issue a command, so delays had to be implemented at the beginning of each state transition. If BARC were playing another AI, no delays would need to be added.

Despite the advantages of testing BARC against another AI opponent as opposed to a human opponent, BARC was only tested against human players. This was because the game crashed when Wargus was set up as AI versus AI.

After having multiple people play several games, it was suspected that BARC was not properly saving cases into its case base because BARC was choosing the same tactics each time after exploration. After viewing the .barc file containing the case base, the suspicion was confirmed. Multiple cases across many states were corrupted and were using tactics which were not available in the current state. The cases were also not saving their associated game descriptions or their performances properly. This went unnoticed for a fair amount of time because it did not appear to be a problem during the exploration phase, and was not rechecked until a few runs after the exploration phase had ended. With the cases not saving properly, BARC was unable to learn how to play better. At this point, there was not enough time to fix BARC and run more tests to get data.
5 Conclusion

I created a CBR system to improve Wargus AI based on the description of CAT given in Aha et al.’s work in “Learning To Win” [2]. The implementation of this proof-of-concept was described in detail, along with previous work in RTS AI and an overview of Case-Based Reasoning. The issues with the current version of BARC were discussed and along with the shortcomings of the testing methodology. After implementing the fixes listed below, BARC could be a fully functioning CBR AI system for Wargus. It could then be tested with the new methods discussed in Section 5.1.2.

5.1 Future Work

There are two main areas for future work on BARC. The first is BARC improvements. These are improvements which can be made to the BARC system or to the Stratagus engine. The second is testing and analysis. These are ways in which BARC can be run and how the data can be analyzed.

5.1.1 BARC Improvements

1. Fix BARC Cases - BARC’s inability to successfully save cases for future reference is currently the largest problem with it. Without this fixed, BARC cannot be successful.

2. Diversify Tactics - After several games of Wargus, it occurred to me that the tactics mostly do the same things at the same times. They all attack at the same levels in the state lattice, they all build extra buildings at the same time, and they all research upgrades at similar times. This may not give BARC enough diversity to choose from when deciding what tactic is best. In order to fully exploit Case-Based Reasoning, BARC needs more diverse options instead of just choosing the build order. Some of the options on a level should attack, and others should not. Some should research upgrades, some should not. This point leads directly into the next improvement.

3. Add More Tactics - Having a larger tactic base will allow for more diverse tactics. Having multiple options of how to get from one state to the next instead of just one will be a significant improvement.

4. Add AI Versus AI - Being able to test BARC versus another AI will give it a foe which will do the same thing each time. This will make it easier to evaluate if BARC learns, as we can simply look at how BARC performs against it over time. Further benefits of using AI versus AI over human versus AI for testing are discussed in Section 4.
5.1.2 Testing and Analysis

Once BARC has been fixed, there are several tests which can be run with it. First, it should train against other AIs. Training against AIs will give a good sense of whether it improves or not, for the reasons stated above. Keeping track of not only wins and losses, but also margins of wins and losses (how many points the AI wins/loses by) during games would give further information as to how well the AI is learning. A further measurement to keep track of is the total number of cases at each run. This will show what the ideal number of cases is, what the relationship between number of cases and win percentage is, and if there is a point where the AI ‘overtrains’, that is if there is a point where the AI begins to lose more often.

After training with AIs, BARC should be tested against human players. These would preferably be experienced Warcraft II players, or at least experienced RTS game players. Again, during these trials scores and cases should be monitored to evaluate performance.
6 Bibliography

References


Appendices

A AI in Stratagus

The AI in the Stratagus engine is controlled by Lua files in the scripts directory of the game running on the Stratagus engine. In this case, I used the Wargus AI scripts. The Wargus scripts control the AI via loops which iterate through lists of functions to execute. Upon startup, Wargus creates a table of 15 loop indexes, one for each of the possible players in the game (there is a maximum of 15 players in a game). These indexes all start at one, and represent the location in the AI function loop of the given AI player. For example, if the value at index $i$ is 5, then player $i$ will execute the fifth element of the function list next.

The function lists are defined in Lua AI files. Some of the files which come with the Wargus engine include “Air_Attack” and “Land_Attack”. In the Lua AI files, the function lists are formatted as such:

```lua
local loop_funcs = {
  function () return AiNeed(AiCityCenter()) end,
}
```

A function then is created that will call on AiLoop with the function list and the index array. The AiLoop function will iterate through the function list, updating the index at each iteration. The function is then registered at the end of the file using the “DefineAi” function, which takes the function and associates it with a name, such as “wc2-air-attack”. This is shown in the following listing. Once the AI has been registered, it can be used in a game.

```lua
function CustomAi()
  // stratagus.gameData.AIState.index is defined in ai.lua
  AiLoop(loop_funcs, stratagus.gameData.AIState.index)
end
```

// params: name, races it can be used for (* for both), // class ("passive", "land-attack", or other), script DefineAi("wc2-custom-ai", ",", "wc2-custom-ai", CustomAi)

To use the AI in a game, the map must be edited to tell the game engine to use the specific AI. The map-name.sms.gz file in the Wargus/maps directory defines which AI’s should be used for each player on the map. In order to change the map default AI, change the AI name in the SetAiType calls. This is shown in the following code snippet from the beginning of a map file:
// the first parameter is the player number
SetStartView(0, 80, 30)
SetPlayerData(0, "Resources", "gold", 2000)
SetPlayerData(0, "Resources", "wood", 1500)
SetPlayerData(0, "Resources", "oil", 1000)
SetPlayerData(0, "RaceName", "orc")
SetAiType(0, "wc2-custom-ai")

With the AI defined in the Lua file and selected in the map file, the given AI will be used by the computer player the next time the game starts on the changed map. When the game starts, Stratagus will call on the AI function registered with "DefineAi", which will start iterating through the function list via the "AiLoop" function. The Stratagus engine will continue to call on the registered function every few cycles to keep the AI running. Each time the "AiLoop" function will be called with the updated loop index, ensuring that the AI is moving through its function list. When it reaches the end of the list, the AI will terminate, ending the game. This can be avoided by making the last line of the function list reset the index to one, causing it to start over on the next call. With this adjustment, the custom AI loop from above would be the following:

```lua
local loop_funcs = {
    function () return AiNeed(AiCityCenter()) end,
    function () stratagus.gameData.AIState.index[1+AiPlayer()] = 1; return false end,
}
```
The .barc file extension is used by BARC to store the case-base between runs. When storing the case base, BARC only keeps track of the game states and their associated cases. The file extension format is defined below. The comments are not a part of the file convention, they are only for convenience. ‘%d’ denotes an integer, while ‘%f’ denotes a double:

```
s // denotes start of state definition
%d %d %d // next_tactics
%d %d %d // times_used
c // denotes start of case (associated with above)
%d // state
%d // tactic
%d %d %d %d %d %d %d %d // game description
%f // performance
%f %f .... %f // previous performances
c // next case in case list
.... // definition of case same as above
s // next state
```
C Further Details of BARC Tactics

This section will give further details of the BARC tactics, specifically when upgrades occur and when certain extra buildings are constructed.

<table>
<thead>
<tr>
<th>Tactic</th>
<th>Upgrades</th>
<th>Extra Buildings</th>
<th>Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>m1</td>
<td>Ba</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>w1, a1</td>
<td>Ba</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>w1, a1</td>
<td>Ba</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>es</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>m1</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>m1, es</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>m1, a1</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>m1, es</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>w1, a1</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>11</td>
<td>es</td>
<td>Bs, Th</td>
<td>No</td>
</tr>
<tr>
<td>12</td>
<td>w1, a1</td>
<td>Bs, Th</td>
<td>No</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>Th</td>
<td>No</td>
</tr>
<tr>
<td>14</td>
<td>m1, es</td>
<td>Bs, Th</td>
<td>No</td>
</tr>
<tr>
<td>15</td>
<td>m1, es</td>
<td>Bs, Th</td>
<td>No</td>
</tr>
<tr>
<td>16</td>
<td>w1, a1</td>
<td>Th</td>
<td>No</td>
</tr>
<tr>
<td>17</td>
<td>c1, es3</td>
<td>Ba</td>
<td>Yes</td>
</tr>
<tr>
<td>18</td>
<td>w1, a1, c1, es3</td>
<td>Ba, Bs</td>
<td>Yes</td>
</tr>
<tr>
<td>19</td>
<td>m1, es, c1, es3</td>
<td>Ba</td>
<td>Yes</td>
</tr>
<tr>
<td>20</td>
<td>w2, a2, c2, es2, es1</td>
<td>Lm, Gtx3</td>
<td>Yes</td>
</tr>
<tr>
<td>21</td>
<td>m2</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>m2, ms1</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>m2, ms1</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>m2, cm, cms1</td>
<td>Ba</td>
<td>Yes</td>
</tr>
<tr>
<td>25</td>
<td>ms1, ms2, ms3</td>
<td>Ba</td>
<td>Yes</td>
</tr>
<tr>
<td>26</td>
<td>cm, cms1, ms2, ms3</td>
<td>Ba</td>
<td>Yes</td>
</tr>
<tr>
<td>27</td>
<td>ms2, ms3, cm, cms1, cms2</td>
<td>Ba</td>
<td>No</td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>Ba</td>
<td>Yes</td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>Ba</td>
<td>Yes</td>
</tr>
<tr>
<td>30</td>
<td>cms2, ms1, ms2, ms3</td>
<td>Ba</td>
<td>Yes</td>
</tr>
<tr>
<td>31</td>
<td>cm, cms1, cms2, ms4, ms5</td>
<td>Ba, Gtx5</td>
<td>No</td>
</tr>
<tr>
<td>32</td>
<td>cms2, ms1, ms2, ms3, ms4, ms5</td>
<td>Ba, Gtx5</td>
<td>No</td>
</tr>
<tr>
<td>33</td>
<td>ms4, ms5</td>
<td>Ba, Gtx5</td>
<td>No</td>
</tr>
<tr>
<td>34</td>
<td></td>
<td>Ba, Apx3, Gt</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2: Extra information about the tactics. Upgrade key: m=missile, w=weapons, a=armor, es=elite shooter, c=catapult, ms=mage spell, cm=cavalry mage, cms=cavalry mage spell. Building key: Ba=Barracks, Bs=Blacksmith, Lm=LumberMill, Gt=Guard Tower, ,Ap = Airport, Th=Town Hall.
D Modifications to Wargus Source Code

This section will detail the modifications made to the Stratagus and Wargus source code. The first section gives a general overview of what was done in each file, and the second section lists where the source code can be obtained.

D.1 Itemized Changes

- "CMakeList.txt"
  - add ai_barccpp to the ai files to compile
- "src/ai/ai_barccpp"
  - new file added to the source to contain the BARC code
- "src/ai/ai_building.cpp"
  - Fix the issue of the AI not building LumberMills or Harbors.
- "src/ai/ai_local.h"
  - Add in prototype for function to register BARC functions in Lua
- "src/ai/ai_resource.cpp"
  - Fix the issue of AI workers not sharing a mine with each other.
- "src/ai/script_ai.cpp"
  - register BARC functions with Lua
- "src/stratagus/script.cpp"
  - uncomment the luaopen_package, luaopen_io, luaopen_os lines in the init_lua function
- "src/ui/mainscr.cpp"
  - Change label for Timer to fix segmentation fault in DrawTimer()
- "src/unit/unit.cpp"
  - Comment out assertion at 1876 to fix and game crashes
- "Wargus/maps/multi/(2)mysterious-dragon-isle.sms.gz"
  - Change map to use BARC AI. Add in Timer creation and start functions
- "Wargus/scripts/ai.lua"
  - Additional indexes added for each state, AiLog function added

- "Wargus/scripts/ai/barc_init.lua"
  - new file added to handle all of the scripting for BARC.

- "Wargus/scripts/stratagus_lua"
  - Turn off fog of war, increase game speed, reveal map

- "Wargus/scripts/wc2.lua"
  - Call end_game function upon finishing a match

D.2 Source Code

- The Stratagus source code is freely available at: “https://launchpad.net/stratagus/+download”

- The Wargus source code is freely available at: “https://launchpad.net/wargus/+download”
  In order to install Wargus, you will need a Warcraft II: Tides of Darkness game CD to get the resource (audio and visual) files for the game.


  In order to run Wargus with my AI on a Linux machine from these binaries, do the following:

  1. Install Ruby and Lua5.1 on your system. This can be done with the following commands:

     ```
     sudo apt-get install ruby
     sudo apt-get install lua5.1
     ```

  2. Download libpng from this address:
     “http://sourceforge.net/projects/libpng/files/libpng16/1.6.10/libpng-1.6.10.tar.gz/download”
3. Extract libpng. Then install it using these three commands:

./configure
make check
make install

4. Open up a terminal to the folder into which you extracted the code. Run the following command:

lua run.lua

Note that if you wish to test the BARC AI, you must select ‘Single Player Game’ from the main screen, and select ‘Start Game’ with the defaults (namely that the map is (2)mysterious-dragon-isle). Any other map will run the map default AI instead of BARC.