Incorporating Search Algorithms into RTS Game Agents

David Churchill and Michael Buro
University of Alberta
Computing Science Department
Edmonton, Alberta, Canada, T6G 2E8

Abstract

Real-time strategy (RTS) games are known to be one of the most complex game genres for humans to play, as well as one of the most difficult games for computer AI agents to play well. To tackle the task of applying AI to RTS games, recent techniques have focused on a divide-and-conquer approach, splitting the game into strategic components, and developing separate systems to solve each. This trend gives rise to a new problem: how to tie these systems together into a functional real-time strategy game playing agent. In this paper we discuss the architecture of UAlbertaBot, our entry into the 2011/2012 StarCraft AI competitions, and the techniques used to include heuristic search based AI systems for the intelligent automation of both build order planning and unit control for combat scenarios.

Introduction

Traditional games such as Chess and Go have for centuries been regarded as the most strategically difficult games to play at a top level. High-level play involves complex strategic decisions based on knowledge obtained through study and training, combined with online analysis of the pieces on the current board. Top players are able to "look ahead" a dozen or more moves into the future to decide on an action, often under strict time constraints, with clocks for each player ticking away as they think make their decision. Let us now imagine a genre of game in which the playing field is 256 times as large, contains up to several hundred pieces per player, with pieces able to be created or destroyed at any moment. On top of this, players may move any number of pieces simultaneously in real-time, with the only limit being their own dexterity. What we have just described is a real-time strategy (RTS) game, which combines the complex strategic elements of traditional games with the real-time actions of a modern video game.

A relatively new genre, the first RTS games started to appear in the early 1990s with titles such as Dune II, WarCraft, and Command and Conquer. Originally introduced as a single-player war simulation, their popularity exploded as the internet allowed for players to compete against each other in multiplayer scenarios. With the creation of StarCraft in 1998, RTS games had reached a level of strategy unseen in other video game genres. Tournaments such as the World Cyber Games had prize pools large enough to create the first professional players by the year 2000, with prize totals in 2011 totalling in the millions of dollars.

In recent years, the field of AI for RTS games has grown as it has shown to be an excellent test-bed for AI algorithms. Due to the enormous size and complexity of the RTS domain, as well as its harsh real-time processing requirements, RTS AI agents are still quite weak with respect to their human counterparts, when compared to the successes of AI agents in traditional games like chess. For example, the winning entry of both the 2011 AIIDE StarCraft AI Competition and the 2011 CIG StarCraft AI Competition, Skynet, was later trivially defeated by a skilled amateur player in a man-machine showmatch.

Due to the complexity of AI algorithms for RTS games (Furtak and Buro 2010), current state of the art RTS AI agents consist primarily of large rule-based systems (like finite state machines) which implement hard-coded strategies derived from expert knowledge. Recently however some agents have started to incorporate dynamic AI solutions in an to attempt to improve performance in certain areas of their play. As more and more AI solutions are developed, the task of integrating them smoothly into an existing agent can be a difficult one.

In this paper we address the problem of incorporating AI algorithms into RTS game agents which face uncertainty about the durations of actions as they are executed by the game engine for which no source code is available. Moreover, game engine interfaces may limit access to essential state information, such as the last time opponent units fired, which is crucial for planning actions for combat units. In the following sections we first describe the StarCraft programming interface and how to extract relevant game data from it. Then we discuss our RTS game agent's hierarchical AI structure and its novel search-based AI modules for build order planning and unit micro-management, and encountered complications when integrating the search procedures into our StarCraft agent. We conclude the paper by experimental results and remarks on future work in this research area.

StarCraft Programming

In the development of any game-playing agent, two requirements are crucial:

- Access to game specific data to perform strategic analysis both on and offline

- Access to play the game itself

Unlike traditional games like chess, real-time strategy games are much more difficult to simulate, requiring systems that govern all aspects of play such as unit movements and attacking. Because StarCraft is a retail game for which no publicly available source code exists, we have two options: either model an approximation of the game in a system we construct, or construct a programming interface to the game to allow us to play the exact version. Luckily, there are tools available for us that do both.

BWAPI

BWAPI (http://code.google.com/p/bwapi/) is a programming interface written in C++ which allows users to read data and send game commands to a StarCraft: BroodWar game client. BWAPI contains all functionality necessary for the creation of a competitive AI agent. Examples of BWAPI functionality are:

- Perform unit actions, i.e.: `Attack, Move, Build`
- Obtain current data about any visible unit, such as `Position, HP, Mana, isIdle`
- Obtain offline data about any unit type, such as `MaxSpeed, Damage, MaxHP, Size, isFlyer`

Programs written with BWAPI are compiled into a Windows dynamically linked library (dll) which is injected into the StarCraft game process via a third-party program called "ChaosLauncher". BWAPI allows the user to perform any of the above functionality while the game is running, after each logic frame update within the game's software. After each logic frame, BWAPI interrupts the StarCraft process and allows the user to read game data and issue commands, which are stored in a queue to be executed during the game's next logic frame. No further StarCraft game logic or animation updates are allowed until all sequential user code has finished executing, a fact we will be concerned about when writing resource intensive AI systems.

As no StarCraft source code has been released by its producer (Blizzard), BWAPI has been created via a process of reverse engineering, with all functionality arising from the getting and setting of data in the StarCraft process' memory space. Although BWAPI provides much functionality, it currently does not give access to some specific data such as unit attack animation frame durations, which are necessary for optimal unit control (discussed in section 5).

PyICE

To extract additional data, we use PyICE: part of the PyMS (http://www.broodwarai.com/index.php?page=pyms) modding suite, it is a tool which allows the reading and editing of StarCraft IScript bin files, which are used by the game to control various aspects of StarCraft such as unit behaviour, sprite animations, attack damage timings, and sound triggers. By analyzing the game script files we are able to extract all additional information necessary to (in theory) carry out unit commands.

Architecture

Several architecture models exist for the construction of game playing agents. (Wintermute, Xu, and Laird 2007) use the SOAR cognitive architecture to implement an RTS agent for the ORTS RTS game (ORTS 2010). They implement a finite state machine based approach to respond to perceptual data from the game environment. (Jaidee, Muoz-Avila, and Aha 2011), use a case-based reasoning architecture to play Wargus, an open source clone of Warcraft II. Their method selects goals based on 'discrepency' events which differ from expected behaviours. (McCoy and Mateas 2008) designed a Wargus agent using ABL (A Behavioural Language), a language designed for reactive planning. Their system uses multiple manager modules for controlling subtasks of more high-level goals.

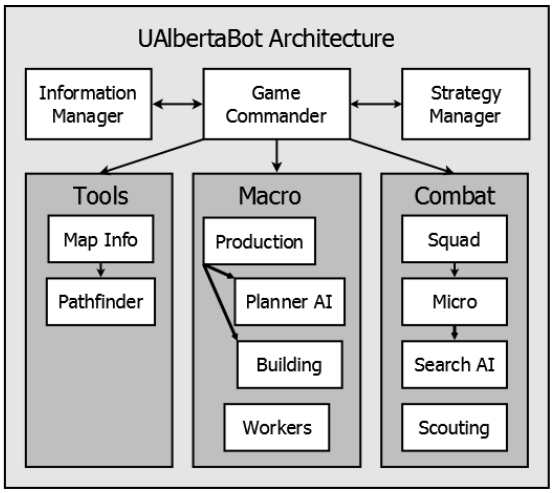

Figure 1: UAlbertaBot Class Diagram

Like many RTS game AI authors before, we have designed UAlbertaBot's architecture in a modular and hierarchical fashion (Figure 1) for two reasons: to retain an intuitive command hierarchy, and to avoid overlap in any computations that need to be performed. Inspired by military command structures, tasks are partitioned among modules by their intuitive strategic meaning (combat, economy, etc.), with vertical communication being performed on a "need to know" basis. High level strategy decisions are made by the game commander by compiling all known information about the current game state. Commands are then given to combat and macro sub-commanders, which in turn give commands to their sub-commanders which are directly in charge of completing the low-level task.

To illustrate how this system works, we will use an example of combat unit micro-management. When the game commander decides to attack a location on the map, it passes this decision onto the combat manager. The combat manager then decides on a squad of attack units to send and moves them toward the location. Once these units enter a given radius of where the battle is to take place, the units are then controlled by the micro manager. The micro manager packages the information about the current combat scenario and then passes that information to a function which will return the actions we should perform for each unit.

The advantage of this architecture is that this function can now be of any level of sophistication, whether it be hand-coded script or a complex search-based AI system. By implementing this design, we can start with a system which

may be full of scripted modules, and gradually replace them by more sophisticated AI systems as we develop them without changing the overall architecture of the agent. For example in 2010, UAlbertaBot used all scripted solutions for each module it contained. For the 2011 version we developed a heuristic search algorithm for build order planning, which simply replaced the existing scripted build order module. We have now done the same thing with unit micro-management by replacing our previously hand-scripted unit actions with our AI based approach.

Resource Allocation

In most cases, the more resources given to an AI agent, the stronger the results that will be returned. This can be problematic for AI systems in RTS games due to the harsh real-time constraints of the game. In the case of BWAPI, the interface will halt the progression of the game until all calculations in the agent's `onFrame()` method have been performed. If a user runs an algorithm for one minute to decide on an action for a given frame, the game will be effectively paused for both players for the full minute. In the 2011 AIIDE StarCraft AI Competition, bots were given 55 ms per game frame to perform all calculations, with game loss penalties for going over this threshold a set number of times. Therefore, it is critical to come up with an efficient scheme for resource allocation when integrating an AI system into an agent that has to act in real-time.

BWAPI allows an agent to gain frame-accurate details about a game state. Therefore, an ideal system would process the game state at every frame and decide on a new set of actions to be carried out for the following frame. This however may not be possible in all cases due to resource budgeting, so one of three situations must arise:

1. Resources (such as processing time) given to an AI system must be reduced to produce an action for the following frame. This will typically result in decreased performance from the AI.
2. The AI system will use more time than is remaining in a single frame, either by staggering the computations across multiple frames, or by performing them in a separate thread. This will yield delayed results from the algorithm because they were given input from a frame at some point in the past.
3. Similar to the previous situation, however one can attempt to estimate the input of a future state to alleviate the side effects of delayed effects. This method then relies on having a good future estimation.

UAlbertaBot uses method 1 and method 2 for its AI systems. In the case of unit micro-management, method 1 is used and the heuristic search algorithm is given the remaining resources for a particular frame to perform its calculations. This is done due to the nature of real-time combat, in which a delayed action may be the difference between life or death. For the build order planning system method 2 is used and calculations are staggered over multiple frames (up to a maximum of 50 frames). The reason for this is that the build order planner may produce a plan which ends up spanning several minutes into the future, so we typically see that the shorter plans produced by allocating more resources far outweigh delaying its execution for a few seconds. In the cases where both AI systems are required to function on the same frame, we simply split the available resources equally among both systems, which can be done either in sequence or in parallel by spawning a separate thread whose duration matches the time remaining in the frame. To accurately determine how much time to allocate to the AI systems we delay their execution until after all other bot tasks have been completed for this frame. To illustrate how this is done, pseudo code for game commander's `onFrame()` method is shown in Algorithm 1.

Algorithm 1 Game Commander onFrame

```
1: procedure ONFRAME()
2:     Clock.start();
3:     Tools.update();
4:     ... Tools.Map.update();
5:     ... Tools.Map.Pathfinder.update();
6:     Macro.update();
7:     ... Macro.WorkerManager.update();
8:     ... Macro.ProductionManager.update();
9:     Combat.update();
10:    ... Combat.ScoutManager.update();
11:    ... Combat.SquadManager.update();
12:    ... Combat.MicroManager.update();
13:    BuildPlannerAI.run(TimeLimit-Clock.elapsed());
14:    MicroAISystem.run(TimeLimit-Clock.elapsed());
```

AI-Specific Implementation Details

When integrating an AI system into an RTS agent, especially those designed to work with retail gaming software such as StarCraft, we may encounter different issues depending on the role of the AI we are implementing. In this section we will discuss some of the implementation details and difficulties associated with the two AI systems we have incorporated into UAlbertaBot.

Build Order Planning

In any RTS game there is an initial phase in which players must gather resources in order to construct training facilities, which in turn construct their armies which will engage in combat. An ordered sequence of actions which produces a given set of goal units is called a *build order*. Due to its exponential time complexity, build order planning requires an approximate AI solution for their construction. We have developed a build order planning using an any-time depth first branch and bound algorithm (Churchill and Buro 2011).

When implementing this system into UAlbertaBot, no real difficulties arose. The system is treated as a black-box function which when given an input set of unit goals, returns a sequence of actions to be executed by the agent. This function simply replaced the previous build order system, which was a rule based finite state machine. Due to the nature of the the build order problem, all actions produced by the algorithm are at a macro scale, such as "construct building" and "gather minerals", with no fine-grain motion or knowledge of the StarCraft engine required.

This is not the case however for actions at a micro scale, where delaying execution by even a few frames may cause disastrous effects.

Table 1: Sequence of events occurring after an attack command has been given in StarCraft. Also listed are the associated BWAPI `unit.isAttacking()` and `unit.isAttackFrame()` return values for the given step.

Attack Sequence	isAttacking	isAttackFrame	Additional Notes
1. Unit is Idle	`False`	`False`	Unit may be idle or performing another command (i.e.: move)
2. Issue Attack Cmd	`False`	`False`	Player gives order to attack a target unit
3. Turn to Face Target	`False`	`False`	May have 0 duration if already facing target
4. Approach Target	`False`	`False`	May have 0 duration if already in range of target
5. Stop Moving	`False`	`False`	Some units require unit to come to complete stop before firing
6. Begin Attack Anim	`True`	`True`	Attack animation starts, damage not yet dealt
7. Anim Until Damage	`True`	`True`	Animation frames until projectile released
8. Mandatory Anim	`True`	`True`	Extra animation frames after damage (may be 0)
9. Optional Anim	`True`	`True`	Other command can be issued to cancel extraneous frames
10. Wait for Reload	`True`	`False`	Unit may be given other commands until it can shoot again
11. Goto Step 3	`False`	`False`	Repeat the attack

Unit Micro-Management

RTS battles are complex adversarial encounters which can be classified as two-player zero-sum simultaneous move games (Churchill and Buro 2011). As in most complex games, no simple scripted strategy performs well in all possible scenarios. We implemented the ABCD (Alpha-Beta search Considering Durations) search algorithm for use in RTS combat settings that is described in (Churchill, Saffidine, and Buro), and integrated it into UAlbertaBot. ABCD search takes action durations into account and approximates game-theoretic solutions of simultaneous move games by sequentializing moves using simple policies such as Max-Min-Max-Min or Min-Max-Min-Max. Applied to RTS game combat scenarios, the paper describes various state evaluation methods ranging from simple static evaluation to simulation-based approaches in which games are played to completion by executing scripted policies. Some details of our implementation are discussed later. For more information about ABCD search we refer the reader to (Churchill, Saffidine, and Buro).

Similar to the build order planning system, our ABCD search based combat AI system acts as a function which takes a given combat scenario as input, and as output produces individual unit actions. Unlike the previous case however, these actions must be carried out with extreme precision in order to guarantee good results. Despite BWAPI's comprehensive interface into the StarCraft game engine, there are still some intuitively simple tasks which require non-trivial effort to implement. Take for example the case of issuing an attack command to a unit in the game. To carry out frame-perfect unit micro-management we will require knowledge of the exact frame in which the unit has fired its weapon and dealt its damage. This is important because StarCraft's game engine will cancel an attack command if another command is given before damage has been dealt, resulting in less damage being done by the unit over time. Currently, there is no functionality in BWAPI which can give us this exact information, so it must be extracted via a combination of reverse-engineered game logic and animation script data obtained via PyICE.

BWAPI gives us access to two separate functions to help determine if a unit is currently attacking: `unit.isAttacking()`, which returns true if the unit is currently firing at a unit with intent to continue firing, and `unit.isAttackFrame()`, which returns true if the unit is current animating with an attack animation frame. Table 1 shows the sequence of events which take place after issuing a unit.attack() command in StarCraft. Steps 1-5 deal with the unit moving into a position and heading at which it can fire, steps 6-9 deal with the actual firing time of the unit, and step 10 is a period of time where the unit is waiting until it can fire again. We can see by this sequence that neither function gives us the exact time when the unit dealt its damage, due to steps 8 and 9, which are steps in which these functions return true, but after damage has already been inflicted. We must therefore use the information extracted from PyICE to determine the frame when damage has been dealt (the end of step 7). For a given unit, we extract the duration of steps 6-9 from PyICE and call this value $atkFrames$.

To determine this timing, we will keep track of the unit after we have given an attack command to make sure no other commands are given before the end of step 7. We record the first frame after the attack command was given for which the unit.isAttackFrame() returns true (the beginning of step 6), and call this value $startAtk$. We then calculate the frame in the future when the unit will have dealt its damage by:

$$damageFrame = startAtk + atkFrames$$

By issuing subsequent commands to the unit only after $damageFrame$ we ensure that no attacks will be interrupted, while allowing the unit to perform other commands between attacks for as long as possible. For example, a Protoss Dragoon unit has an attack cooldown of 23 frames, but an $atkFrames$ value of 7, which means it has 16 frames after firing that it is free to move around before it fires again, which can be useful for strategic attack sequences such as *kiting*, a technique used against units with short range weapons to avoid taking damage by fleeing outside of its weapon range while waiting to reload.

However, despite this effort which should work in theory, in practice the StarCraft engine does not behave in a strict deterministic fashion, and work is still being done to perfect this model so that a higher level of precise combat unit control can be obtained.

Experiments

In order to evaluate the success of our architecture, we will perform experiments that demonstrate two points:

- That the AI systems have been successfully integrated into the bot, and that the resource allocation strategy we

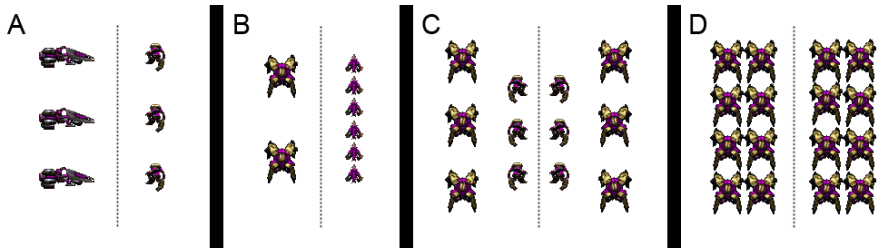

Figure 2: Micro search experiment scenarios. A) 3 ranged Vultures vs. 3 melee Zealot. B) 2 ranged Dragoons vs. 6 fast melee Zerglings. C) 3 Dragoon + 3 Zealots in symmetric formation. D) 8 Dragoons in symmetric two-column formation.

chose did not cause any issues in bot performance

- That the AI system contributed to the success of the agent as a whole, and is an improvement over existing techniques

Build Order Planner Performance

To evaluate our build order planning AI we use the results from the 2011 AIIDE StarCraft AI Competition. UAlbertaBot incorporated the build order planning system described in section 5.1. All building actions were planned by this system, and re-planning was triggered by the following events: at the start of the game, when the current build queue is exhausted, or when a worker unit or building unit is destroyed.

As shown in Figure 1, the build order planner was controlled by the production manager of the bot. When one of the above events was triggered, the production manager would pass a new unit composition goal into the planner, and it would return an ordered list of actions to be carried out by the building manager.

Resource management was implemented as shown in algorithm 1. The game commander's main onFrame() function keeps track of remaining resources as each of the other systems perform their tasks. Remaining time resources for the current frame were then given to the planner, which ran for the given time limit during the current frame. At the end of this frame time limit, the search state was saved, and the computation resumed during the next frame, until an overall time limit was reached. For our experiment, this overall time limit was set to 5 seconds. The bot ran single-threaded on an Intel E8500 CPU with 4 GB of RAM, with less than 1 MB of memory used for the planner itself.

UAlbertaBot came in 2nd place in both the 2011 AIIDE StarCraft AI Competition, as well as the 2011 CIG StarCraft AI Competition. All build orders (several dozen per game) were planned using the AI system in real-time. Over the course of 30 rounds of round robin competition, UAlbertaBot won 79.4% of its 360 matches, with no game losses due to processing time-outs. While these results cannot conclude whether or not the planner contributed to the bot's success, they do confirm that the architecture and resource management strategies were sound. In addition, it was the only build order planning AI present in any competition of 2011, with all other entries using rule-based orderings.

Combat Micro AI Performance

To evaluate our combat search AI system, we implemented a simplified version of UAlbertaBot in BWAPI which only performs combat scenarios. Unfortunately, due to time constraints our micro AI system has not yet been implemented into the full version of UAlbertaBot. For this reason we will only allocate 5 ms per frame to our AI's search algorithm in order to simulate a harsh environment in which the full bot is executing each frame. In 2011, UAlbertaBot's micromanagement system involved a policy of "Attack Weakest Enemy Unit in Range", with an option for game commander to retreat the squad from combat for various strategic purposes. For this experiment no retreat was allowed — combat is performed until one team has been eliminated or a time limit of 5000 frames (208 seconds) is reached.

In this experiment we construct several StarCraft test maps which contain pre-positioned combat scenarios. To evaluate our AI system (Search) we will do a comparison of its performance to that of the micro-management system present in the 2011 version of UAlbertaBot (AttackWeakest). Due to the desire to avoid issues with network latency (necessary to play one combat policy against the other directly) we instead chose to perform combat with both methods vs. the default StarCraft AI, and then compare the obtained scores. The default StarCraft AI's combat policy is not explicitly known, however it is thought to be approximately equal to AttackWeakest, however it does not appear to be deterministic. To test our BWAPI implementation itself we will also run the same experiments inside a simulated model of the StarCraft combat engine as described in (Churchill, Saffidine, and Buro). Games were played against the AttackWeakest scripted policy, which is the closest known to that of the default StarCraft AI.

The scenarios we construct are designed to be realistic early-mid game combat scenarios which could be found in an actual StarCraft game. They have also been designed specifically to showcase a variety of scenarios for which no single scripted combat policy can perform well under all cases, and can be seen in Figure 2. Units for each player are shown separated by a dotted line, with the default AI units placed to the right of this line. Unit positions were fixed to the formations shown at the start of each trial, but units were allowed to freely move about the map if they are instructed to do so. For each method, 200 combat trials were performed in each of the scenarios.

Scenario A is designed such that the quicker, ranged Vulture units start within firing range of the zealots, and must adopt a "run and shoot" strategy (known as kiting) to defeat the slower but stronger melee Zealot units. Scenario B is similar to A. However, two strong ranged Dragoons must also kite a swarm of weaker melee Zerglings to survive. Sce-

Table 2: Results from the micro AI experiment. Shown are scores for Micro Search, AttackWeakest, and Kiter decision policies each versus the built-in StarCraft AI for each scenario. Scores are shown for both the micro simulator (Sim) and the actual BWAPI-based implementation (Game).

	Combat Decision Settings					
	Search (5 ms)		AtkWeakest		Kiter	
	Sim	Game	Sim	Game	Sim	Game
A	1.00	0.81	0	0	1.00	0.99
A'	1.00	0.78	0	0	1.00	0.99
B	1.00	0.65	0	0	1.00	0.94
B'	1.00	0.68	0	0	1.00	0.89
C	1.00	0.95	0.50	0.56	0	0.14
C'	1.00	0.94	0.50	0.61	0	0.09
D	1.00	0.96	0.50	0.58	0	0.11
D'	1.00	0.97	0.50	0.55	0	0.08
Avg	1.00	0.84	0.25	0.29	0.50	0.53

nario C is symmetric, with initial positions allowing the Dragoons to reach the opponent zealots, but not the opponent Dragoons. Scenario D is also symmetric, wherein each unit is within firing range of each other unit. Therefore, a good targeting policy will perform well.

In addition to the scenarios shown, four more scenarios A', B', C', and D' were tested, each having a similar formation to the previously listed scenarios. However, their positions will be perturbed slightly to break their perfect line formations. In the case of C and D, symmetry was maintained for fairness. These experiments were performed on hardware similar to the build-order planning hardware, with 1 MB total memory used for the search routine and 2 MB for the transposition table.

The results from the micro search experiment are presented in Table 2. Shown are scores for a given combat method, which are defined as: $score = wins + draws/2$. We can see from these results that it is possible (through expert knowledge) to design a scripted combat policy (such as Kiter) which will perform well in scenarios where it is beneficial to move out of shorter enemy attack range like in scenarios A/B, but will fail in scenarios where excess movement is detrimental as it imposes an small delay on firing like in scenarios C/D. Scripts such as AttackWeakest perform better than Kiter in scenarios in which its better targeting policy and lack of movement allow for more effect damage output, but fail completely in situations such as A/B where standing still in range of powerful melee enemies spells certain death. By implementing a dynamic AI solution for combat micro problems, we have dramatically improved overall performance over a wide range of scenarios, even while under the extremely small time constraint of 5 ms per frame.

Also of note in these results is the fact that although the scripted strategies are deterministic, the outcome in the actual BWAPI implementation was not always the same for each trial. In a true deterministic and controllable RTS game model (such as our simulator), each of the scripted results should either be all wins, losses, or draws. This surprising result must be due to the nature of the StarCraft engine itself, for which we do not have an exact model. It is known that the StarCraft engine does have a small level of stochastic behaviour both in its unit hit chance mechanism (http://code.google.com/p/bwapi/wiki/StarcraftGuide) and its random starting unit heading direction. It is unknown whether or not the default combat policy contains non-deterministic elements. It also highlights an additional frustration of implementing an RTS game bot in a real-world scenario: that results may not always be exactly repeatable, so robust designs are necessary.

Conclusion and Future Work

In this paper we have described the architecture of our RTS game playing agent and how we integrated various AI components in a hierarchical fashion. The current focus of our research is on adding AI search components for build order planning and unit micro management. While integrating build order planning is straight forward because precise action timing isn't crucial, we have encountered problems when incorporating our combat search module into a StarCraft bot that have yet to be overcome. The challenge is to accurately model complex action timings — many of them undocumented but crucial for combat situations — based on limited information provided by BWAPI, the programming interface to the StarCraft game engine. Stated another way, we need to better align the fidelity of our unit micro-management search with an environment in which exact data is not available. This can possibly be achieved by measuring (average) action times in controlled experiments or by using a two-step search procedure in which a high-level search considers sequences of low-level macro actions that are restricted to those that can be efficiently executed without slowing down attacks and unit motion. The promising experimental results we presented make us feel confident that in time for the 2012 AIIDE StarCraft AI competition our unit micro-management will generally outperform the built-in combat scripts and — hopefully — most of the competitors.

References

Churchill, D., and Buro, M. 2011. Build order optimization in StarCraft. In *Proceedings of AIIDE*.

Churchill, D.; Saffidine, A.; and Buro, M. Fast heuristic search for RTS game combat scenarios. In *Proceedings of AIIDE, (pre-print available at www.cs.ualberta.ca/˜mburo/ps/aiide12-combat.pdf*.

Furtak, T., and Buro, M. 2010. On the complexity of two-player attrition games played on graphs. In Youngblood, G. M., and Bulitko, V., eds., *Proceedings of AIIDE*.

Jaidee, U.; Muoz-Avila, H.; and Aha, D. W. 2011. Case-based learning in goal-driven autonomy agents for real-time strategy combat tasks. In *Proceedings of the ICCBR Workshop on Computer Games*, 43–52.

McCoy, J., and Mateas, M. 2008. An integrated agent for playing real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence. Chicago, Illinois: AAAI Press*, 1313–1318.

ORTS. 2010. ORTS - A Free Software RTS Game Engine. http://www.cs.ualberta.ca/˜mburo/orts/.

Wintermute, S.; Xu, J.; and Laird, J. E. 2007. Sorts: A human-level approach to real-time strategy ai. In *Proceedings of the Third AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2007*.