Fast Heuristic Search for RTS Game Combat Scenarios

David Churchill
University of Alberta, Edmonton, T6G 2E8, Canada (cdavid@cs.ualberta.ca)

Abdallah Saffidine
Universit Paris-Dauphine, 75775 Paris Cedex 16, France (abdallah.saffidine@dauphine.fr)

Michael Buro
University of Alberta, Edmonton, T6G 2E8, Canada (mburo@ualberta.ca)

Abstract

Heuristic search has been very successful in abstract game domains such as Chess and Go. In video games, however, adoption has been slow due to the fact that state and move spaces are much larger, real-time constraints are harsher, and constraints on computational resources are tighter. In this paper we present a fast search method — Alpha-Beta search for durative moves — that can defeat commonly used AI scripts in RTS game combat scenarios of up to 8 vs. 8 units running on a single core in under 5ms per search episode. This performance is achieved by using standard search enhancements such as transposition tables and iterative deepening, and novel usage of combat AI scripts for sorting moves and state evaluation via playouts. We also present evidence that commonly used combat scripts are highly exploitable — opening the door for a promising line of research on opponent combat modelling.

1 Introduction

Automated planning, i.e. finding a sequence of actions leading from a start to a goal state, is a central problem in artificial intelligence research with many applications such as robot navigation and theorem proving. While search-based planning approaches have had a long tradition in the construction of strong AI systems for abstract games like Chess and Go, only in recent years have they been applied to modern video games, such as first-person shooter (FPS) and real-time strategy (RTS) games (Orkin 2006; Churchill and Buro 2011). Generating move sequences automatically has considerable advantages over scripted behavior, as anybody who tried to write a good rule-based Chess program can attest:

- Search naturally adapts to the current situation. By looking ahead it will often find winning variations, where scripted solutions fail due to the enormous decision complexity. For example, consider detecting mate-in-3 situations statically, i.e. without enumerating move sequences.
- Creating search-based AI systems usually requires less expert knowledge and can therefore be implemented faster. Testament to this insight is Monte Carlo tree search, a recently developed sample based search technique that revolutionized computer Go (Coulom 2006).

These advantages come at a cost, namely increased runtime and/or memory requirements. Therefore, it can be challenging to adapt established search algorithms to large scale real-time decision problems, e.g. video games and robotics. In what follows, we will see how the Alpha-Beta search algorithm can be used to solve adversarial real-time planning problems with durative moves. We begin by motivating the application area — small scale combat in RTS games — and discussing relevant literature. We then define our search problem formally, describe several solution concepts, and then present our ABCD algorithm (Alpha-Beta Considering Durations) which we will evaluate experimentally. We conclude by discussing future research directions in this area.

2 Modelling RTS Game Combat

Battle unit management (also called micromanagement) is a core skill of successful human RTS game players and is vital to playing at a professional level. One of the top STARCRAFT players of all time, Jaedong, who is well known for his excellent unit control, said in a recent interview: *"That micro made me different from everyone else in Brood War, and I won a lot of games on that micro alone"*.[1] It has also been proved to be decisive in the previous STARCRAFT AI competitions, with many battles between the top three AI agents being won or lost due to the quality of unit control. In this paper we focus on small-scale battle we call *combat*, in which a small number of units interact in a small map region without obstructions.

In order to perform search for combat scenarios in STARCRAFT, we must construct a system which allows us to efficiently simulate the game itself. The BWAPI programming interface allows for interaction with the STARCRAFT interface, but unfortunately, it can only run the engine at 32 times "normal speed" and does not allow us to create and manipulate local state instances efficiently. As one search may simulate millions of moves, with each move having a duration of at least one simulation frame, it remains for us to construct an abstract model of STARCRAFT combat which is able to efficiently implement moves in a way that does not rely on simulating each in-game frame.

[1] http://www.teamliquid.net/forum/viewmessage.php?topic_id=339200

2.1 Combat Model

To fully simulate RTS game combat, our model is comprised of three main components: states, units, and moves.

State $s = \langle \mathrm{t}, p, m, U_1, U_2 \rangle$

- Current game time t
- Player p who performed move m to generate s
- Sets of units U_i under control of player i

Unit $u = \langle p, \mathrm{hp}, \mathrm{hp}_{\max}, \mathrm{t}_a, \mathrm{t}_m, v, w \rangle$

- Position $p = (x, y)$ in $\mathbb{R}^2$
- Current hit points hp and maximum hit points $\mathrm{hp}_{\max}$
- Time step when unit can next attack t_a, or move t_m
- Maximum unit velocity v
- Weapon properties $w = \langle \mathrm{damage}, \mathrm{cooldown} \rangle$

Move $m = \{a_0, \ldots, a_k\}$ which is a combination of unit actions $a_i = \langle u, \mathrm{type}, \mathrm{target}, t \rangle$, with

- Unit u to perform this action
- The type type of action to be performed:
  Attack unit target
  Move u to position target
  Wait until time t

2.2 Legal Move Generation

Given a state s containing unit u, we generate legal unit actions as follows: if $u.\mathrm{t}_a \leq s.\mathrm{t}$ then u may *attack* any target in its range, if $u.\mathrm{t}_m \leq s.\mathrm{t}$ then u may *move* in any legal direction, if $u.\mathrm{t}_m \leq s.\mathrm{t} < u.\mathrm{t}_a$ then u may *wait* until $u.\mathrm{t}_a$. If both $u.\mathrm{t}_a$ and $u.\mathrm{t}_m$ are $> s.\mathrm{t}$ then a unit is said to have no legal actions. A legal player move is then a set of all combinations of one legal unit action from each unit a player controls.

Unlike strict alternating move games like chess, our model's moves have durations based on individual unit properties, so either player (or both) may be able to move at a given state. We define the player to move next as the one which contains the unit with the minimum time for which it can attack or move.

2.3 Model Limitations

While the mathematical model we propose does not exactly match the combat mechanics of STARCRAFT it captures essential features. Because we don't have access to STARCRAFT's source code, we can only try to infer missing features based on game play observations:

- no spell casting (e.g. immobilization, area effects)
- no hit point or shield regeneration
- no travel time for projectiles
- no unit collisions
- no unit acceleration, deceleration or turning
- no fog of war

Quite a few STARCRAFT AI competition entries are designed with a strong focus on early game play (rushing). For those programs some of the listed limitations, such as single weapons and spell casting, are immaterial because they become important only in later game phases. The utility of adding others, such as dealing with unit collisions and acceleration, will have to be determined once our search technique becomes adopted.

3 Solution Concepts for Combat Games

The combat model defined in Section 2 can naturally be complemented with a termination criterion and utility functions for the players in terminal positions. A position is called *terminal* if all the units of a player have reached $0\mathrm{hp}$, or if a certain time limit (measured in game frames, or unit actions) has been exceeded. Combining the combat model with the termination criterion and utility functions defines a class of games we call *combat games*. In what follows we will assume that combat games are zero-sum games, i.e., utilities for both players add up to a constant across all terminal states. This property together with simultaneous moves and fully observable state variables places combat games in the class of "stacked matrix games". Such games can — in principle — be solved by backward induction starting with terminal states via Nash equilibrium computations for instance by solving linear programs (Saffidine, Finnsson, and Buro 2012). However, Furtak and Buro (2010) showed that deciding which player survives in combat games in which units can't even move is PSPACE-hard in general. This means that playing combat games optimally is computationally hard and that in practice we have to resort to approximations. There are various ways to approximate optimal play in combat games. In the following sub-sections we will discuss a few of them.

3.1 Scripted Behaviors

The simplest approach, and the one most commonly used in video game AI systems, is to define static behaviors via AI scripts. Their main advantage is computation speed, but they often lack foresight, which makes them vulnerable against search-based methods, as we will see in Section 5, where we will evaluate the following simple combat AI scripts:

- The *Random* strategy picks legal moves with uniform probability.
- Using the *Attack-Closest* strategy units will attack the closest opponent unit within weapon's range if it can currently fire. Otherwise, if it is within range of an enemy but is reloading, it will wait in-place until it has reloaded. If it is not in range of any enemy, it will move toward the closest enemy a fixed distance.
- The *Attack-Weakest* strategy is similar to Attack-Closest, except units attack an opponent unit with the lowest hp within range when able.
- Using the *Kiting* strategy is similar to Attack-Closest, except it will move a fixed distance away from the closest enemy when it is unable to fire.

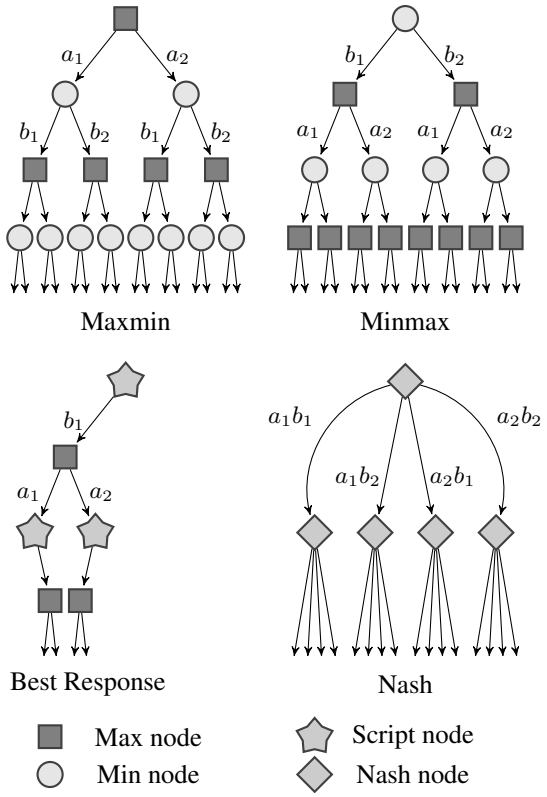

Figure 1: Maxmin, Minmax, Best Response, Nash

The Attack-Closest script was used in second-place entry *UAlbertaBot* in the 2010 AIIDE STARCRAFT AI competition, whereas *Skynet*, the winning entry, used a behavior similar to Kiting.

3.2 Game Theoretic Approximations

As mentioned above, combat games fall into the class of two-player zero-sum simultaneous move games. In this setting, the concepts of optimal play and game values are well defined, and the value $\mathrm{Nash}(G)$ of a game G (in view of the maximizing player *MAX*) can be determined by using backward induction. However, as discussed earlier, this process can be very slow. Kovarsky and Buro (2005) describe how games with simultaneous moves can be sequentialized to make them amenable to fast alpha-beta tree search, trading optimality for speed. The idea is to replace simultaneous move states by two-level subtrees in which players move in turn, maximizing respectively minimizing their utilities (Figure 1: Minmax and Maxmin). The value of the sequentialized games might be different from $\mathrm{Nash}(G)$ and it depends on the order we choose for the players in each state with simultaneous moves: If *MAX* chooses his move first in each such state (Figure 1: Minmax), the value of the resulting game we call the *pure maxmin value* and denote it by $\mathrm{mini}(G)$. Conversely, if *MAX* gets to choose after *MIN*, we call the game's value the *pure minmax value* (denoted $\mathrm{maxi}(G)$). An elementary game theory result is that pure minmax and maxmin values are bounds for the true game value:

Proposition 1. *For stacked matrix games G, we have $\mathrm{mini}(G) \leq \mathrm{Nash}(G) \leq \mathrm{maxi}(G)$, and the inequalities are strict iff the game does not admit optimal pure strategies.*

It is possible that there is no optimal pure strategy in a game with simultaneous moves, as ROCK-PAPER-SCISSORS proves. Less intuitively so, the need for randomized strategies also arises in combat games, even in cases with 2 vs. 2 immobile units (Furtak and Buro (2010)). To mitigate the potential unfairness caused by the Minmax and Maxmin game transformations, (Kovarsky and Buro 2005) propose the Random-Alpha-Beta (RAB) algorithm. RAB is a Monte Carlo algorithm that repeatedly performs Alpha-Beta searches in transformed games where the player-to-move order is randomized in interior simultaneous move nodes. Once time runs out, the move with the highest total score at the root is chosen. (Kovarsky and Buro 2005) shows that RAB can outperform Alpha-Beta search on the Maxmin-transformed tree, using iterative deepening and a simple heuristic evaluation function. In our experiments, we will test the stripped down RAB version we call *RAB'*, which only runs Alpha-Beta once.

Another approach of mitigating unfairness is to alternate the player-to-move order in simultaneous move nodes on the way down the tree. We call this tree transformation *Alt*.

Because RAB' and the Alt transformation just change the player-to-move order, the following result on the value of the best RAB move ($\mathrm{rab}(G)$) and Alt move ($\mathrm{alter}(G)$) are easy to prove by induction on the tree height:

Proposition 2. *For stacked matrix game G, we have*

$$\mathrm{mini}(G) \leq \mathrm{rab}(G), \mathrm{alter}(G) \leq \mathrm{maxi}(G)$$

The proposed approximation methods are much faster than solving games by backward induction. However, the computed moves may be inferior. Section 5 we will see how they perform empirically.

4 Fast Alpha-Beta Search for Combat Games

In the previous section we discussed multiple game transformations that would allow us to find solutions by using backward induction. However, when playing RTS games the real-time constraints are harsh. Often, decisions must be made during a single simulation frame, which can be 50 ms or shorter. Therefore, computing optimal moves is impossible for all but the smallest settings and we need to settle for approximate solutions: we trade optimality for speed and hope that the algorithms we propose defeat the state of the art AI systems for combat games.

The common approach is to declare nodes to be leaf nodes once a certain depth limit is reached. In leaf nodes *MAX*'s utility is then estimated by calling an *evaluation function*, and this value is propagated up the tree like true terminal node utilities.

In the following subsections we will first adapt the Alpha-Beta search algorithm to combat games by handling durative moves explicitly and then present a series of previously known and new evaluation functions.

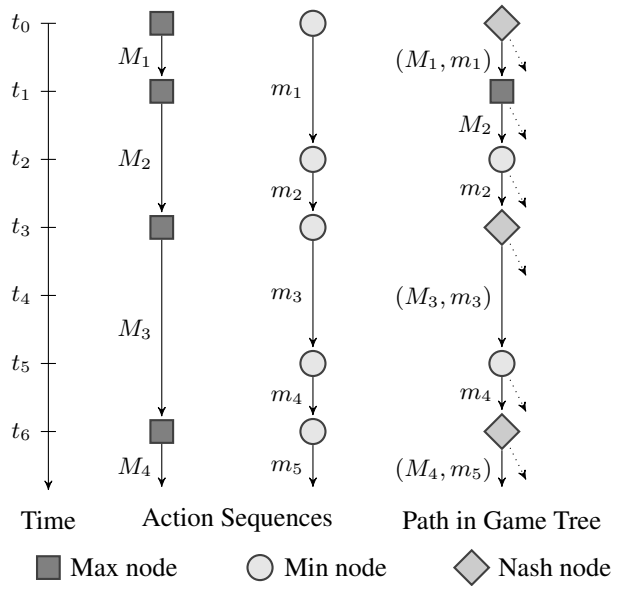

Figure 2: Durations

4.1 Alpha-Beta Search with Durative Moves

Consider Figure 2 which displays a typical path in the sequentialized game tree. Because of the weapon cooldown and the space granularity, battle games exhibit numerous *durative moves*. Indeed, there are many time steps where the only move for a player is just pass, since all the units are currently unable to perform an action. Thus, non-trivial decision points for players do not occur on every frame.

Given a player p in a state s, define the next time where p is next able to do a non-pass move by $\tau(s, p) = \min_{u \in s.U_p}(u.\mathrm{t}_a, u.\mathrm{t}_m)$. Note that for any time step t such that $s.\mathrm{t} < t < \min(\tau(s, \textit{MAX}), \tau(s, \textit{MIN}))$, players cannot perform any move but pass. It is therefore possible to shortcut many trivial decision points between $s.\mathrm{t}$ and $\min(\tau(s, \textit{MAX}), \tau(s, \textit{MIN}))$.

Assume an evaluation function has been picked, and remaining simultaneous choices are sequentialized as suggested above. It is then possible to adapt the Alpha-Beta algorithm to take advantage of durative moves as presented in Algorithm 1

We use the *terminal(s, d)* function to decide when to call the evaluation function. It is parametrized by a maximal depth $d_{\max}$ and a maximal time $t_{\max}$ and return "true" if s is a terminal position or $d \geq d_{\max}$ or $s.\mathrm{t} \geq t_{\max}$.

The third argument to the ABCD algorithm is used to handle the *delayed action effect* mechanism for sequentialized simultaneous moves. If the state does not correspond to a simultaneous decision, m_0 holds a dummy value $\emptyset$. Otherwise, we apply the effects of m_0 after move m is generated because m_0 should not affect the generation of the complementary moves.

Algorithm 1 Alpha-Beta (Considering Durations)

```
1:  procedure ABCD(s, d, m_0, α, β)
2:      if computationTime.elapsed then return timeout
3:      else if terminal(s, d) then return eval(s)
4:      toMove ← s.playerToMove(policy)
5:      while m ← s.nextMove(toMove) do
6:          if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
7:              val ←ABCD(s, d − 1, m, α, β)
8:          else
9:              s' ← copy(s)
10:             if m_0 ≠ ∅ then s'.doMove(m_0)
11:             s'.doMove(m)
12:             v ←ABCD(s', d − 1, ∅, α, β)
13:         if toMove = MAX and (v > α) then α ← v
14:         if toMove = MIN and (v < β) then β ← v
15:         if α ≥ β then break
16:     return toMove = MAX ? α : β
```

4.2 Evaluation Functions

A straight-forward evaluation function for combat games is the hitpoint-total differential, i.e.

$$e(s) = \sum_{u \in U_1} \mathrm{hp}(u) - \sum_{u \in U_2} \mathrm{hp}(u)$$

which, however, doesn't take into account other unit properties, such as damage values and cooldown periods. Kovarsky and Buro (2005) propose an evaluation based on the life-time damage a unit can inflict, which is proportional to its hp times its damage-per-frame ratio:

$$\mathrm{dpf}(u) = \frac{\mathrm{damage}(w(u))}{\mathrm{cooldown}(w(u))}$$

$$\mathrm{LTD}(s) = \sum_{u \in U_1} \mathrm{hp}(u) \cdot \mathrm{dpf}(u) - \sum_{u \in U_2} \mathrm{hp}(u) \cdot \mathrm{dpf}(u)$$

A second related evaluation function Kovarsky and Buro (2005) propose favours uniform hp distributions:

$$\mathrm{LTD2}(s) = \sum_{u \in U_1} \sqrt{\mathrm{hp}(u)} \cdot \mathrm{dpf}(u) - \sum_{u \in U_2} \sqrt{\mathrm{hp}(u)} \cdot \mathrm{dpf}(u)$$

While these evaluation functions are exact for terminal positions, they can be drastically inaccurate for many non-terminal positions. To improve state evaluation by also taking other unit properties such as speed and weapon range into account, we can try to simulate a game and use the outcome as an estimate of the utility of its starting position. This idea is known as *performing a playout* in game tree search and is actually a fundamental part of Monte Carlo Tree Search (MCTS) algorithms which have revolutionized computer GO (Coulom 2006). However, there are differences between the playouts we advocate for combat games and previous work on GO and HEX: the playout policies we use here are deterministic and we are not using MCTS or a best-first search algorithm, but rather depth-first search.

4.3 Move Ordering

It is well-known in the game AI research community that a good move ordering fosters the performance of the Alpha-Beta algorithm. When transposition tables (TTs) and iterative deepening are used, reusing previous search results can improve the move ordering. Suppose a position p needs to be searched at depth d and was already searched at depth d'. If $d \leq d'$, the value of the previous search is sufficiently accurate and there is no need for an additional search on p. Otherwise, a deeper search is needed, but we can explore the previously found best move first and hope for more pruning.

When no TT information is available, we can use scripted strategies to suggest moves. We call this new heuristic *scripted move ordering*. Note that this heuristic could also be used in standard sequential games like CHESS. We believe the reason it has not been investigated closely in those contexts is the lack of high quality scripted strategies.

5 Experiments

We implemented the proposed combat model, the scripted strategies, the new ABCD algorithm, and various tree transformations. We then ran experiments to measure 1) the influence of the suggested search enhancements for determining the best search configuration, and 2) the real-time exploitability of scripted strategies.

Because the scripts we presented in Subsection 3.1 are quite basic, we added a few smarter ones to our set of scripts to test:

- The *Attack-Value* strategy is similar to Attack-Closest, except units attack an opponent unit u with the highest $\mathrm{dpf}(u)/\mathrm{hp}(u)$ value within range when able. This choice leads to optimal play in 1 vs. n scenarios (Furtak and Buro 2010).
- The *NOK-AV* (No-OverKill-Attack-Value) strategy is similar to Attack-Value, except units will not attack an enemy unit which has been assigned lethal damage this round. It will instead choose the next priority target, or wait if one does not exist.
- Using the *Kiting-AV* strategy is similar to Kiting, except it will choose an attack target similar to Attack-Value.

Most scripts we described make decisions on an individual unit basis, with some creating the illusion of unit collaboration (by concentrating fire on closest or weakest or most-valuable units). NOK-AV is the only script in our set that exhibits true collaborative behaviour by sharing information about unit targeting.

Because of time constraints, we were only able to test the following tree transformations: Alt, Alt', and RAB', where Alt' in simultaneous move nodes selects the player that acted last, and RAB' selects the player to move like RAB, but only completes one Alpha-Beta search.

5.1 Setup

The combat scenarios we used for the experiments involved equally sized armies of n versus n units, where n varied from 2 to 8. 1 versus 1 scenarios were omitted due to over 95% of them resulting in draws. Four different army types were constructed to mimic various combat scenarios. These armies were: *Marine* Only, *Marine* + *Zergling*, *Dragoon* + *Zealot*, and *Dragoon* + *Marine*. Armies consisted of all possible combinations of the listed unit type with up to 4 of each, for a maximum army size of 8 units. Each unit in the army was given to player *MAX* at random starting position (x, y) within 256 pixels of the origin, and to player *MIN* at position $(-x, -y)$, which guaranteed symmetric start locations about the origin. Once combat began, units were allowed to move infinitely within the plane. Unit movement was limited to up, down, left, right at 15 pixel increments, which is equal to the smallest attack range of any unit in our tests.

These settings ensured that the Nash value of the starting position was always 0.5. If the battle did not end in one player being eliminated after 500 actions, the simulation was halted and the final state evaluated with LTD. For instance, in a match between a player p_1 and an opponent p_2, we would count the number of wins by p_1, w, and number of draws, d, over n games and compute $r = (w + d/2)/n$. Both players perform equally well in this match if $r \approx 0.5$.

As the 2011 StarCraft AI Competition allowed for 50ms of processing per game logic frame, we gave each search episode a time limit of 5ms. This simulates the real-time nature of RTS combat, while leaving plenty of time (45ms) for other processing which may have been needed for other computations.

Experiments were run single-threaded on an Intel Core i7 2.67 GHz CPU with 24 GB of 1600 MHz DDR3 RAM using the Windows 7 64 bit operating system and Visual C++ 2010. A transposition table of 5 million entries (20 bytes each) was used. Due to the depth-first search nature of the algorithm, very little additional memory is required to facilitate search. Each result table entry is the result of playing 365 games, each with random symmetric starting positions.

5.2 Influence of the Search Settings

To measure the impact of certain search parameters, we perform experiments using two methods of comparison. The first method plays static scripted opponents vs. ABCD with various settings, which are then compared. The second method plays ABCD vs. ABCD with different settings for each player.

We start by studying the influence of the evaluation function selection on the search performance (see Section 4.2). Preliminary experiments revealed that using NOK-AV for the playouts was significantly better than using any of the other scripted strategies. The playout-based evaluation function will therefore always use the NOK-AV script.

We now present the performance of various settings for the search against script-based opponents (Table 1) and search-based opponents (Table 2). In Table 1, the Alt sequentialization is used among the first three settings which allow to compare the leaf evaluations functions LTD, LTD2, and playout-based. The leaf evaluation based on NOK-AV playouts is used for the last three settings which allow to compare the sequentialization alternatives described in Subsection 3.2.

We can see based on the first three settings that doing a

Table 1: ABCD vs. Script - scores for various settings

Opponent	ABCD Search Setting				
	Alt LTD	Alt LTD2	Alt NOK-AV Playout	Alt' NOK-AV Playout	RAB' NOK-AV Playout
Random	0.99	0.98	1.00	1.00	1.00
Kite	0.70	0.79	0.93	0.93	0.92
Kite-AV	0.69	0.81	0.92	0.96	0.92
Closest	0.59	0.85	0.92	0.92	0.93
Weakest	0.41	0.76	0.91	0.91	0.89
AV	0.42	0.76	0.90	0.90	0.91
NOK-AV	0.32	0.64	0.87	0.87	0.82
Average	0.59	0.80	0.92	0.92	0.91

Table 2: Playout-based ABCD performance

Opponent	Alt NOK-AV Playout	Alt' NOK-AV Playout	RAB' NOK-AV Playout
Alt-NOK-AV		0.47	0.46
Alt'-NOK-AV	0.53		0.46
RAB'-NOK-AV	0.54	0.54	
Average	0.54	0.51	0.46

search based on a good playout policy leads to much better performance than doing a search based on a static evaluation function. The search based on the NOK-AV playout strategy is indeed dominating the searches based on LTD and LTD2 against any opponent tested. We can also see based on the last three settings that the Alt and Alt' sequentializations lead to better results than RAB'.

5.3 Estimating the Quality of Scripts

The quality of scripted strategies can be measured in at least two ways: the simplest approach is to run the script against multiple opponents and average the results. To this end, we can use the data presented in Table 1 to conclude that NOK-AV is the best script in our set. Alternatively, we can measure the exploitability of scripted strategies by determining the score a theoretically optimal best-response-strategy would achieve against the script. However, such strategies are hard to compute in general. Looking forward to modelling and exploiting opponents, we would like to approximate best-response strategies quickly, possibly within one game simulation frame. This can be accomplished by replacing one player in ABCD by the script in question and then run ABCD to find approximate best-response moves. The obtained tournament result we call the *real-time exploitability* of the given script. It constitutes a lower bound (in expectation) on the true exploitability and tells us about the risk of being exploited by an adaptive player. Table 3 lists the real-time exploitability of various scripted strategies. Again, the NOK-AV strategy prevails, but the high value suggests that there is room for improvement.

6 Conclusion and Future Work

In this paper we have presented a framework for fast Alpha-Beta search for RTS game combat scenarios of up to 8 vs. 8 units and evaluate it under harsh real-time conditions. Our method is based on an efficient combat game abstraction model that captures important RTS game features, including unit motion, an Alpha-Beta search variant (ABCD) that can deal with durative moves and various tree transformations, and a novel way of using scripted strategies for move ordering and depth-first-search state evaluation via playouts. The experimental results are encouraging. Our search, when using only 5 ms per episode, defeats standard AI scripts as well as more advanced scripts that exhibit kiting behaviour and minimize overkill. The prospect of opponent modelling for exploiting scripted opponents is even greater: the practical exploitability results indicate large win margins best-response ABCD can achieve if the opponent executes any of the tested combat scripts.

Table 3: Real-time exploitability of scripted strategies.

Random	Weakest	Closest	AV	Kiter	Kite-AV	NOK-AV
1.00	0.98	0.98	0.98	0.97	0.97	0.95

The ultimate goal of this line of research is to handle larger combat scenarios with more than 20 units on each side in real-time. The enormous state and move complexity, however, prevents us from applying heuristic search directly, and we therefore will have to find spatial and unit group abstractions that reduce the size of the state space so that heuristic search can produce meaningful results in real-time. Balla and Fern (2009) present initial research in this direction, but their UCT-based solution is rather slow and depends on pre-assigned unit groups.

Our next steps will be to integrate ABCD search into a STARCRAFT AI competition entry to gauge its performance against previous year's participants, to refine our combat model if needed, to add opponent modelling and best-response-ABCD to counter inferred opponent combat policies, and then to tackle more complex combat scenarios.

References

Balla, R.-K., and Fern, A. 2009. UCT for tactical assault planning in real-time strategy games. In Boutilier, C., ed., *IJCAI*, 40–45.

Churchill, D., and Buro, M. 2011. Build order optimization in starcraft. In *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE*.

Coulom, R. 2006. Efficient selectivity and back-up operators in Monte-Carlo tree search. In *Proceedings of the 5th Conference on Computers and Games (CG'2006)*, volume 4630 of *LNCS*, 72–83. Torino, Italy: Springer.

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 the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2010*.

Kovarsky, A., and Buro, M. 2005. Heuristic search applied to abstract combat games. *Advances in Artificial Intelligence* 66–78.

Orkin, J. 2006. Three states and a plan: the AI of FEAR. In *Game Developers Conference*. Citeseer.

Saffidine, A.; Finnsson, H.; and Buro, M. 2012. Alpha-Beta pruning for games with simultaneous moves. In *Proceedings of the Twenty-Sixth Conference on Artificial Intelligence (AAAI-12)*.