Heuristic Search Applied to Abstract Combat Games

Alexander Kovarsky[1] and Michael Buro[2]

University of Alberta, Edmonton, Alberta, Canada
{kovarsky [1],mburo [2]}@cs.ualberta.ca

Abstract. Creating strong AI forces in military war simulations or RTS video games poses many challenges including partially observable states, a possibly large number of agents and actions, and simultaneous concurrent move execution. In this paper we consider a tactical sub-problem that needs to be addressed on the way to strong computer generated forces: abstract combat games in which a small number of inhomogeneous units battle with each other in simultaneous move rounds until all members of one group are eliminated. We present and test several adversarial heuristic search algorithms that are able to compute reasonable actions in those scenarios using short time controls. Tournament results indicate that a new algorithm for simultaneous move games which we call "randomized alpha-beta search" (RAB) can be used effectively in the abstract combat application we consider. In this application it outperforms the other algorithms we implemented. We also show that RAB's performance is correlated with the degree of simultaneous move interdependence present in the game.

1 Introduction

Abstract combat games — also known as combat attrition scenarios in military literature — have long been a focus of military research [5]. In the area of computer generated forces these models can be used to predict the outcome of simulated battles and to compute actions that would for instance maximize the inflicted damage or unit survival probability. In order to simplify the problem, states are usually abstracted. For instance, terrain is often represented as collection of convex cells — typically squares or hexagons — and objects as vectors that describe attack values, health or so-called hit-points, position, size, maximum and current speed, heading, and sight-range, etc.

The purpose of this research is to design and study fast heuristic decision procedures for abstract combat games that belong to the class of two-player simultaneous move games with otherwise perfect information. Such algorithms have applications in popular real-time strategy (RTS) video games — such as Warcraft (http://www.blizzard.com) — which essentially are real-time battle simulations. For instance, incorporating abstract combat algorithms in graphical user interfaces relieves human players from laborious unit micro-management and lets them focus on more strategic decisions. In extreme cases where hundreds of fighting units have to be managed or several separate battles are fought, issuing fire commands to units manually in timely fashion may not even be possible. When delegating local fights to AI modules the tactical performance could even improve because programs may select targets and concentrate fire faster and more accurate than any human. Increasing the playing strength of allied or opponent

computer players in RTS games to make game playing more challenging is another immediate application from which military combat simulators can benefit, too.

The central idea of this research is to go beyond established analytical methods — that model warfare globally with differential equations [11] — by studying adversarial heuristic search techniques in this domain. In general terms, such algorithms conduct look-ahead searches in (abstract) two-player state spaces by repeatedly generating successor states, deciding where to proceed, evaluating states, and propagating those values in the constructed search graph. Minimax-search and its enhancements, for example, fall into this category. Often, deep search can compensate for less than perfect domain knowledge. A good example is chess, where PC programs now have reached World-champion-level performance by combining deep search with relatively simple (fast) evaluation functions. Using the chess example again, it is a non-trivial task to write an evaluation function that can statically detect and assess capture sequences. On the other hand, look-ahead search that makes use of simple material features can find and evaluate capture sequences quickly. Here, too, the hope is that look-ahead search can overcome the need for accurate evaluation models — which in general are hard to find — and that search even under real-time constraints produces high-quality actions.

Video games have been the focus of several AI research projects in recent years. E.g. in [4] evolutionary algorithms are used to enable AI characters to develop novel behaviours in an RTS-like combat situations. In [8] complex artificial characters for a custom designed adventure game are developed based on the Unreal Tournament game engine and the SOAR AI architecture. Stochastic search has been studied in this area, too (e.g. [9]). However, to our knowledge, no previous research has applied randomized heuristic search to two player games with simultaneous moves.

The remainder of the paper is organized as follows: first we define the class of combat games we consider. Next, a number of search algorithms are presented for these games ranging from an optimal solution based on solving linear programs to a Monte Carlo algorithm. We then describe a new algorithm for simultaneous move games. Finally, we present and discuss experimental results and conclude the paper with suggestions for future research.

2 Abstract Combat Games

In abstract combat games two players are in control of teams of units which attack each other in simultaneous move rounds. In each turn, both players give orders to their units which then are executed simultaneously. Games start with two groups of units and end after a sequence of rounds when one group is eliminated. We make the following simplifying assumptions:

- there are no hidden state variables
- all units have the ability to attack any opponent unit at any time
- units are static objects, they cannot move

In the games we consider here units have the following numerical properties:

h : Hitpoints — the defensive strength of a given unit
a : Attack value (const.) — the amount of damage a unit can inflict
c : Cool-down period (const.) — how long to wait before next attack

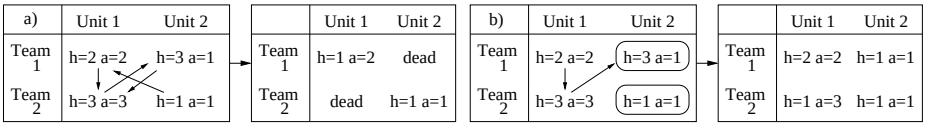

Fig. 1. State transitions in abstract combat games. Arrows indicate attacks, boxes indicate defensive stance. a) non-defensive scenario. b) defensive scenario. When a unit defends, its attack value is temporarily added to its hitpoints. This decreases the damage the unit receives in case it is attacked. If a defending unit is not attacked, its hitpoint value remains unchanged.

Each unit has one weapon with a specific attack value and cool-down period. If the weapon has cooled down, i.e. the weapon was last fired at least $c + 1$ rounds ago, it can fire again. Before attacking, units have to select their target. In the next turn they can shoot only at that target. If a unit wants to attack another target it needs to aim. Aiming costs one time step. When a unit gets attacked its hitpoint value is decreased by the attackers attack value. If a unit's hitpoint value drops below 1, it is considered destroyed and removed from the game. A typical state transition is shown in Fig. 1a).

In the basic set-up despite actions being executed simultaneously we theorize that the success of a player's action does not strongly depend on what the opponent chooses to do. I.e., announcing a move hardly hurts. This is because damage is inflicted regardless. To study the performance of the heuristic search algorithms we implemented in more Rock-Paper-Scissor-like scenarios, where announcing moves is foolish, we add a defensive action and call games with this additional move option "defensive". The defensive action enables the unit to use its attack value for defending rather than for attacking. Specifically, if a unit decides to defend instead of attacking, a certain proportion (possibly > 1) of its attack value is temporarily added to its hitpoints. When this unit is attacked the attacker will only cause damage to the unit if its attack value is bigger than the unit's proportion of attack value used for defence. If a defending unit is not attacked its hitpoints remain unchanged (i.e. not increased). In addition, a defending unit can also cause damage to the attacker. There is a certain benefit to take a defensive stance. But there is also a degree of risk because a unit taking a defensive action might not be attacked and is therefore risking to waste its turn. A defensive game scenario is shown in Fig. 1b).

3 Heuristic Search in Abstract Combat Games

The major challenges in the abstract combat games we just described are large branching factors, limited decision time, and simultaneous move execution. In the simplest case — simultaneous move zero-sum — games consist of one start state and $n \times m$ successor states which are reached after executing single simultaneous action pairs. Here, player A has n actions to choose from and player B has m. The values of the terminal states are given in form of a $n \times m$ payoff matrix. Optimal (mixed) strategies for these games always exist [12] and can be computed by solving a pair of associated linear programs. Abstract combat games can be viewed as multi-step matrix games and

as such can be solved by dynamic programming [3]. Unfortunately, this technique has no practical relevance because state spaces — even for small scenarios — are huge.

An alternative approach is to trade solution quality for speed by considering heuristic search algorithms. In the remainder of this section we first present the evaluation function used in all heuristic algorithms we implemented. Then we will describe all algorithms in turn.

Evaluation Functions for Abstract Combat Games. The heuristic algorithms we describe below rely on an evaluation function that measures the goodness of a state in view of player 1 or 2. The following function estimates the differential of the total lifetime damage two groups of units can inflict:

$$\sum_{i=1}^{n_1} \frac{h_i^{(1)} \times a_i^{(1)}}{c_i^{(1)} + 1} - \sum_{i=1}^{n_2} \frac{h_i^{(2)} \times a_i^{(2)}}{c_i^{(2)} + 1}, \tag{1}$$

where unit attribute superscripts indicate the unit owner and n_1, n_2 denote the number of units for player 1 and 2, respectively. Here, hitpoints are used as estimator of the life expectancy of a unit, while attack value over cool-down plus 1 represents the average damage a unit will deal during one time step. Evaluation function (1) is monotone in the h and a values, takes cool-down into account, and can be computed quickly. We used it for a while in our experiments before a serious weakness became apparent: its inability to differentiate between hitpoint distributions. In general, it is more beneficial for a player to have units with a uniform hitpoint distribution than having some units with low hitpoint values and others with high hitpoint values. This is because units with low hitpoints values are much closer to elimination. For a fixed average attack value, evaluation function (1) only considers the sum of hitpoints. Thus, values for widely varying hitpoint distributions — everything else being equal — could be the same. The following function fixes this problem at the expense of introducing non-linearity and departing from modeling the lifetime damage:

$$\sum_{i=1}^{n_1} \frac{\sqrt{h_i^{(1)}} \times a_i^{(1)}}{c_i^{(1)} + 1} - \sum_{i=1}^{n_2} \frac{\sqrt{h_i^{(2)}} \times a_i^{(2)}}{c_i^{(2)} + 1} \tag{2}$$

By applying the square root to hitpoints the evaluation function implicitly prefers more uniform hitpoint distributions. Function (2) was used in the experiments reported in section 5 after initial tournaments indicated that it performs better than function (1). We have not tried to optimize the evaluation function further — for instance by introducing parameters and optimizing them, because the focus of this work is search rather than evaluation function construction, which is a research topic by itself.

Linear Programming (LP). We have seen that solving abstract combat games by means of dynamic programming and linear programs is impractical. One idea to compute approximate move distributions in this setting is to stop at a certain search depth and to apply a heuristic evaluation function to the end states reached at that depth. Our LP player searches at depth 1. I.e. it generates all moves for both players, considers all possible move pairs, evaluates the reached states using the evaluation function (2), creates a payoff matrix from these values, and solves the linear program associated with

the player to move (see e.g. [3]). It then draws a move with respect to the computed optimal move distribution. We limit the search depth to one because depth 2 computations take too much time in our RTS game setting. This figure may change in future experiments when using better linear program solvers or faster hardware.

Alpha-Beta Search (AB). Minimax search and its enhancements have been proven effective in perfect information games where two players alternate turns under moderate real-time constraints (e.g. chess). The minimax search procedure traverses a search DAG in depth-first order until a certain depth is reached. It then evaluates the resulting position and propagates values to the parent node according to the minimax rule, i.e. maximizing scores in MAX nodes and minimizing scores in MIN nodes. The search continues until the value of the root node has been established, in which case the move leading to the best result is chosen. Alpha-beta pruning [7] is an enhancement of the the minimax algorithm which can reduce the search time exponentially in the depth while still computing the correct minimax values and moves. It is therefore tempting to apply alpha-beta search to abstract combat games, even though the algorithm originally was designed for alternating move games. The way we approximate simultaneous moves is by postponing the execution of the first player's action until the second player has chosen a move. This implicitly gives away the first player's choice, but there is hope that this approximation can still lead to strong performance because of deeper searches compares with other methods. There are many ways to improve the performance of alpha-beta search. For this application we implemented iterative deepening and sorting moves based on their 1-ply evaluations.

Monte Carlo Sampling (MC). Monte Carlo methods solve problems by executing a large number of possibly biased random actions and examining the numerical results such actions generate. The method is used for finding solutions to problems that are too complex to solve analytically. In game AI research, Monte Carlo approaches have been successfully applied to Bridge and recently to the game of Go [2]. In our approach we play out a game until one player is eliminated. For each of the main player's (i.e. the player who performs the simulation) moves at the root, a series of simulations is performed given the available resources. After the runs, the average scores and standard deviations for each move at the top are computed. The move with the "best" average score and standard deviation combination is selected by the player to be executed. Score calculation details are presented in Section 5. In each turn the MC player randomly selects one of the the available actions and then executes it. The run continues in this fashion with both players executing their randomly selected moves, until one of the players is eliminated. A score is calculated for the position, propagated to the top node, and then recorded as one of the values for the selected move.

Move Selection. One way to combat large branching factors in search is to forward-prune subtrees. In some games (e.g. backgammon [6]), limited-depth searches can produce a subsets of moves that likely contain the best moves available. We use move selection for all our search-based methods (i.e. AB, MC, and RAB (described in the next section). Before the start of each of the proposed methods at each level in the tree we perform a complete depth one search for each of the successors. After that, the top N successors are sorted in decreasing order of their scores. Then the search will concentrate only on those top N successors. Because of the decreased branching factor

the search can go much deeper. For such a move sorting scheme to be successful, it is important to have an accurate state evaluation function.

Delayed Move Execution. Delayed move execution is required when using algorithms for alternating move games, such as alpha-beta, in simultaneous move games. In alternating move games, the first player will change the game state by executing a move. After that the second player will change the state by executing his move. Such a sequence of events does not represent the situation accurately, because the state of the game should have remained the same when the second player decides on its move. Therefore, in our search algorithms we delay move executions until both players have committed to an action.

4 Randomized Alpha-Beta Search (RAB)

Our goal when designing the RAB algorithm was to overcome the major disadvantage of the basic alpha-beta algorithm of not addressing possible move dependencies in simultaneous move games. Alpha-beta is a search algorithm for alternating move perfect information games in which all actions are observable. Therefore, players have the advantage of seeing the previous opponent action, which when alpha-beta search is applied to simultaneous move games results in over- or underestimating the value of positions. To soften the effect of advance knowledge of opponent's moves we propose an algorithm based on alpha-beta search which randomizes the color to move in certain nodes. Thus, in some nodes in the tree player one will have the advantage of knowing player two's move, while in other nodes the situation will be reversed.

The algorithm is quite simple: at even depths the color to move is randomized and at odd depths the color to move is changed. The only exception occurs at the root of the tree where the first move always belongs to the player performing the search and its moves are always followed by opponent moves. A sample RAB tree is shown in Fig. 2.

Our hope is that with RAB the advantage of the second player to move will be reduced, because in the whole search tree both players will have equal chances of knowing the opponent's moves. In RTS games, however, algorithms need to perform under tight time constraints. RAB, as opposed to AB is non-deterministic, because every run of the algorithm is likely to produce different results. Being a sampling-based approach, it requires multiple runs to be performed for every move that is available to the player executing the search at the root. For every such move, the best scores will be recorded for every run. Then, for each move the average score and the standard deviation are calculated. Multiple runs are required for drawing valid conclusions about the quality of

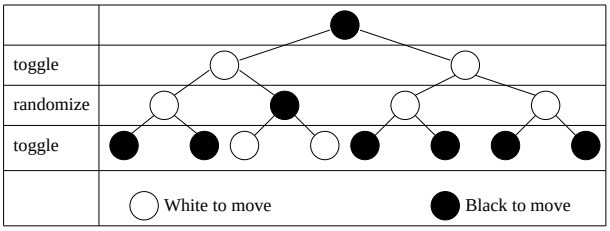

Fig. 2. Sample RAB tree. Black moves first at the root, followed by White. At the next level, the player to move is randomly selected. This player's move is followed by a move of the opponent.

moves. The move that is chosen for execution is determined by taking into account the average scores and standard deviations for each considered move. The move with the best combination of average score and standard deviation is then executed. Section 5 gives details on score calculation. We think that the score average combined with the standard deviation for each move simulates the effect of simultaneous move execution better than a single regular alpha-beta run. However, the main concern with RAB is the number of runs that it will require to find a good move. Also, if there is very little advantage of knowing the opponent's moves in a given game, it is possible that alpha-beta search can find a good move or possibly the best move in just a single run. It is also possible that it is more worthwhile to invest the extra resources into deeper searches than on repeated searches. Fig. 3 shows pseudo-code of the RAB search algorithm.

RAB Implementation and Score Calculation Details. The RAB algorithm is implemented using iterative deepening [10] which is often utilized in environments with real-time constraints. It performs multiple searches starting with the lowest depth and increases the search depth at every successive iteration. The rationale behind the technique is that lower depth searches take only a fraction of the time the next higher depth search will take and thus not much time will be wasted. The main benefit is that at any

```
// compute scores for all moves
void TopLevelRAB(State state, vector<double> &moveScores, int depth) {
  vector<Move> moves;
  moveScores.clear();
  GenerateMoves(state, moves);
  for i = 1..moves.size() { // evaluate all generated moves once
    newState = makeMove(state, moves[i]);
    score = RAB(newState, -infinity, infinity, depth, 0);
    moveScores.append(score);
  }
}

// recursive randomized tree search; uses the negamax variant of alpha-beta
void RAB(State state, int alpha, int beta, int depth, int randGenerate) {
  if (terminalNode(state) || depth == 0) return evaluate(state);
  if (randGenerate) toMove = random() & 1; // pick 0 or 1 randomly
  else toMove = opponent(state); // toggle color to move
  randGenerate = 1 - randGenerate; // toggle flag
  if (parentPlayerToMove(state) == currentPlayerToMove(state)) {
    alpha = -beta; beta = -alpha;
  }
  SetToMove(state, toMove);
  maxScore = -infinity;
  GenerateMoves(state, moves);
  for i = 1..moves.size() {
    newState = MakeMove(state, moves[i]);
    // nega-scout alpha-beta variant
    value = - RAB(newState, -beta, -alpha, depth-1, randGenerate);
    if (value > maxSscore) maxSscore = value;
    if (maxSscore > alpha) alpha = maxSscore;
    if (maxSscore >= beta) break;
  }
  if (parentPlayerToMove(state) == currentPlayerToMove(state)) return -score;
  else return score;
}
```

Fig. 3. RAB Pseudo-code

time a reasonable solution is available which makes the search algorithm fit for real-time applications. At each search depth RAB needs to complete several iterations in order to gather meaningful statistics. The higher the search depth the more iterations RAB needs to complete at that depth, because the deeper the search the higher the variability or standard deviation of the results.

5 Experiments

A tournament environment was set up for performing the experiments and for gathering statistics on the results. A tournament game is a match between two players who battle with each other until one player is eliminated. At each state both players implement their respective algorithms to find their best move. Then both moves are executed simultaneously, the state of the game is updated, and the game continues until one or both players are eliminated (refer to Section 3 for an example). Given two players A and B, a win for A(B) occurs when A(B) has unit(s) remaining while B(A) does not. A draw occurs when both players have no units remaining. Each experiment consists of 200 games. To make the summary of experiments easier to understand and analyze, the experimental results are presented in terms of the *win ratio*:

$$(\#\text{wins} + 0.5 \cdot \#\text{draws}) \,/\, (\#\text{wins} + \#\text{losses} + \#\text{draws})$$

The number of wins, losses, draws, as well as the average scores achieved in each run is recorded. To minimize the variance, symmetric starting positions are chosen. The units in each team are generated randomly within predefined boundaries. There are three types of units: tanks, marines, and artillery. Each type has the ranges of hitpoints, attack values and cool-down periods as shown in Table 1.

The node count limits that are used in the experiments were selected in order to produce acceptable real-time performance on the machines used for the experiments. Specifically, the experiments are run on Athlon MP/XP 2400+ to 2500+ processors with 512–1024 MBs of memory. For non-defensive experiments the node limit for one move in a game for each player when set at 200k nodes results in average game durations of $\approx$4.5 seconds, consisting of $\approx$5-6 moves for each player. For defensive experiments, when the node limit is set at 300k nodes, each game lasts on average 9 seconds, consisting of $\approx$6-7 moves for each player.

To calculate the score in both RAB and MC the setting of (average score – 1 $\times$ standard deviation) is used. This setting was determined in a preliminary set of experiments and will be used in all of our experiments. For all experiments the square root evaluation function (2) is used as it performed best in a preliminary experiment. The empirically values for N — the number of considered moves in the move selection function — were determined in a series of experiments as well. We chose $N = 10$ for 3 vs. 3 non-defensive and $N = 20$ for defensive scenarios. In 4 vs. 4 non-defensive and defensive scenarios N is set to 40 and 60, respectively.

The two types of starting positions examined in all experiments are the 3 versus 3 units and 4 versus 4 units. In the 3 vs. 3 case teams consist of two marines and one tank. In the 4 vs. 4 case, teams consist of one artillery unit, two marines, and one tank.

Performance of all Methods. The results of the 3 vs. 3 experiments shown in Table 2 and Table 3 suggest that the RAB and AB players are the best performers in two typical

Table 1. Hitpoint, attack value, and cool-down period ranges used in the experiments.

Attribute	Tank	Marine	Artillery
Hitpoints	60..90	30..40	20..30
Attack value	30..45	15..25	40..60
Cool-down period	1	0	2

Table 2. Cumulative win ratios for each algorithm obtained by a round-robin tournament for non-defensive and defensive 3 vs. 3 scenarios.

Algorithm	Cumulative Win % non-defensive	Cumulative Win % defensive
RAB	73%	75%
AB	68%	68%
MC	64%	53%
LP	43%	53%
RAND	2%	2%

Table 3. Round-robin tournament results. Reported are win percentages in view of the player named in the left-hand column for the non-defensive and defensive 3 vs. 3 scenarios.

Players	RAB	AB	MC	LP	RAND
RAB	—	52%,60%	56%,72%	84%,67%	99%,99%
AB	48%,40%	—	53%,67%	75%,66%	97%,99%
MC	44%,28%	47%,33%	—	64%,52%	99%,98%
LP	16%,33%	25%,34%	36%,48%	—	96%,97%
RAND	1%,1%	3%,1%	1%,2%	4%,3%	—

scenarios, with MC coming third. The LP player's performance is not very close to that of the best methods, because of its limited search depth. In defensive scenarios LP's performance improves significantly, but still is not on par with that of either RAB or AB. Therefore, the remaining experiments will concentrate only on RAB and AB.

Varying RAB Nodes vs. Constant AB. This experiment, shown in Fig. 4(A,B), is designed to determine how providing RAB more resources changes its performance. RAB is playing against the AB algorithm whose maximal node count is held constant, while the number of nodes assigned to RAB is varied. For 3 vs. 3 defensive and non-defensive scenarios AB is assigned 50k and 100k nodes, respectively. For both 4 vs. 4 defensive and non-defensive scenarios AB is assigned 200k. The general trend is that as the number of RAB's node budget increases, the quality of RAB's moves increases. The increase is more gradual in the case of defensive scenarios, as compared to the non-defensive ones. In non-defensive scenarios there is less interdependence than in the defensive scenarios, therefore reaching higher depths has more effect on the quality of the resultant solution.

Strict Constraints Experiment. The results in Fig. 4(C,D) for both defensive and non-defensive scenarios show that RAB performs better than AB across most of the settings. The most surprising finding is that given a very limited number of nodes for both defensive and non-defensive scenarios RAB outperforms AB. This shows that even though AB can reach greater depth than RAB given the same node limit, investing into randomization and extra runs rather than into deeper searches pays off very early for the RAB algorithm. Another general trend observed is the gradual reduction of RAB's improvement over AB. The results show that as the number of nodes increased for both algorithms, RAB reaches a ceiling in its winning percentage over AB.

Degree of Move Interdependence. The independent variable in the experiment shown in Fig. 4(E,F) is the degree of dependence of a given scenario. This variable can be easily adjusted in our domain starting with a setting with no defensive actions, and finishing with the setting where there is a very high potential reward for selecting defensive actions. Specifically, in a scenario with no reward for the defensive actions units are not motivated to execute such actions since the defensive actions do not benefit them. The exact settings for labels in Fig. 4(E,F) are as follows: "no defence": no defensive action used; "small defence": 0.5 $\times$ attack value is used for defence and 0.1 to hit back at the attacker; "medium Defence: 1.0 $\times$ attack value for defence and 0.2 to hit back at the

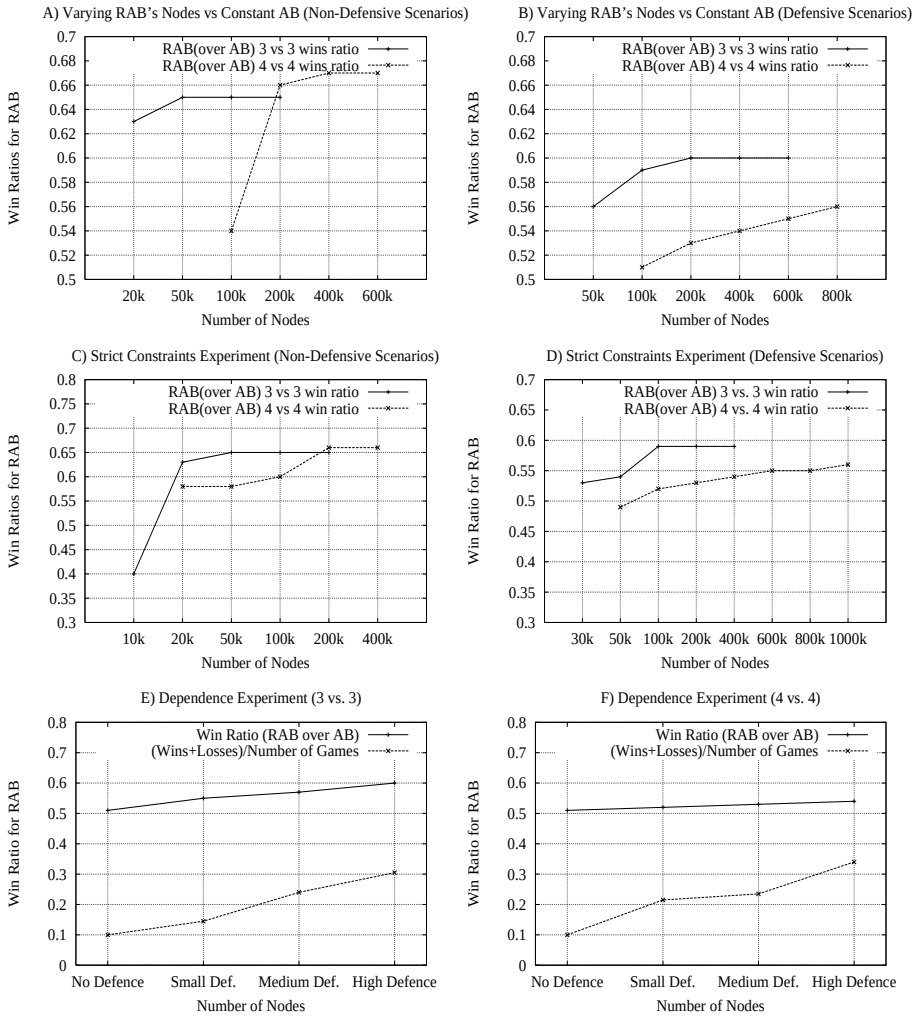

Fig. 4. Tournament Results: (A,B), (C,D), (E,F)

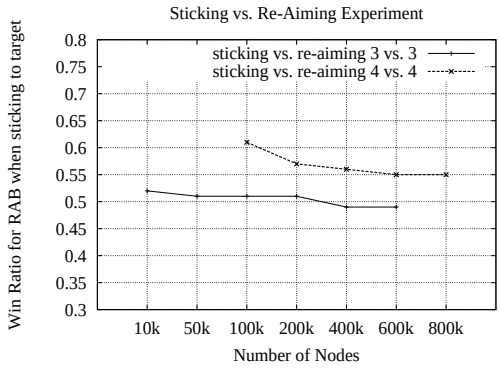

Fig. 5. Sticking to target vs. re-aiming results.

attacker; "high Defence": 1.3 $\times$ attack value for defence and 0.3 to hit back. The results for both 3 vs. 3 and 4 vs. 4 situations show that as the reward for a defensive action increases so do the win ratios for RAB over AB. This result underpins our initial hypothesis that in highly interdependent scenarios the RAB will perform better. Another correlation that can be observed in both graphs is between the win ratio of RAB and the number of wins and losses as a percentage of the number of simulations. This is not surprising, since as the move interdependence increases the success of actions increasingly depends on what the opponent will choose to do. Therefore, in a highly defensive scenario there is no single move that guarantees at least a draw for a player. The opponent can counteract most moves taken by the player leading to a higher standard deviation of the results.

Sticking to Target Improvement. One of the constraints that can significantly reduce the branching factor is not allowing re-aiming. It means that if a unit has picked a target it should keep shooting (stick) at that target. That is, from the time the unit has picked a target until the target elimination, that unit has only one action available to it. We would like to see whether not allowing units to re-aim could lead to a better real-time performance. The results in Fig. 5 show a small advantage when units implement the sticking to the target policy. Because of the reduced branching factor when no re-aiming is allowed it is more advantageous to stick to the target when the number of nodes is small. As the node budget increases, the performance of the method that does not stick to its target slowly increases to over 50% in the 3 vs. 3 case. We can conclude that sticking to a target is especially useful when there are strict real-time constraints, however when the node limit is increased the performance of the no-sticking algorithm improves.

6 Conclusion and Future Work

One of the goals of our research was to examine whether search-based methods can be used effectively in real-time domains with simultaneous move execution. We have

shown that heuristic adversarial search can be successful in small-scale abstract combat games with real-time constraints. Moreover, experimental evidence suggests that non-deterministic search methods perform better than traditional minimax algorithms in games with higher simultaneous move dependence. This result shows that search depth is not a crucial feature when designing algorithms for simultaneous move games, as opposed to alternating move perfect information games, where search depth strongly correlates with the quality of the found solution.

The best overall performer in the field of algorithms we have considered was a naive implementation of RAB which is based on the idea of combining deep alpha-beta searches with sampling to robustly compute actions in case of simultaneous move dependencies. This result is very promising and encourages us to look at RAB improvements that make better use of gathered sample data and would allow the algorithm to return better estimates of the current state value. Scalability to larger problem instances should also be addressed in order to make RAB truly suitable for handling close-combat scenarios in RTS games. Another interesting line of future research could address more theoretical aspects of generalized abstract combat games such as their computational complexity and how to define the degree of simultaneous move interdependence rigorously. Our next step will be to incorporate our methods into an ORTS [1] client, either as part of a stand-alone RTS game AI or as a helper module in the graphical user interface to alleviate the burden of manually micro-managing units in RTS game combat.

Acknowledgments

Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. ORTS — a free software RTS game engine: http://www.cs.ualberta.ca/~mburo/orts.
2. B. Bouzy. Associating domain-dependent knowledge and Monte Carlo approaches within a go program. In *Proc. of the Joint Conf. on Information Sciences*, pages 505–508, Cary, 2003.
3. M. Buro. Solving the Oshi-Zumo game. In *Proceedings of the 10. Advances in Computer Games Conference*, pages 361–366, Graz, 2003.
4. B. Fogel, T. Hays, and D. Johnson. A platform for evolving characters in competitive games. In *Proceedings of CEC2004*, pages 1420–1426, 2004.
5. R. Gozel. Firepower score attrition algorithms in highly aggregated combat models. *RAND*, pages 47–60, 2000.
6. T. Hauk, M. Buro, and J. Schaeffer. *-minimax performance in backgammon. In *Proceedings of the Computers and Games Conference*, 2004.
7. D. Knuth and R. Moore. An analysis of alphabeta pruning. *Artif. Intell.*, 6(4):293–326, 1975.
8. J. E. Laird and et al. A test bed for developing intelligent synthetic characters. In *Proceedings of Spring Symposium on Artificial Intelligence and Interactive Entertainment, AAAI*, 2002.
9. W. Ruml. Incomplet tree search using adaptive probing. In *Proceedings of the International Joint Conference on AI*, pages 235–241, 2001.
10. D. Slate and L. Atkin. Chess 4.5. *Springer-Verlag*, 1977.
11. J. Taylor. Lanchester models of warfare. In *Operations Res. Soc. Vol 1+2*, Arlington, 1983.
12. J. von Neumann. Zur Theorie der Gesellschaftsspiele. *Math.Ann.100*, pages 295–320, 1928.