# Portfolio Greedy Search and Simulation for Large-Scale Combat in StarCraft

David Churchill and Michael Buro
University of Alberta, Edmonton, T6G 2E8, Canada
Email: dave.churchill@gmail.com,mburo@ualberta.ca

***Abstract*—Real-time strategy video games have proven to be a very challenging area for applications of artificial intelligence research. With their vast state and action spaces and real-time constraints, existing AI solutions have been shown to be too slow, or only able to be applied to small problem sets, while human players still dominate RTS AI systems. This paper makes three contributions to advancing the state of AI for popular commercial RTS game combat, which can consist of battles of dozens of units. First, we present an efficient system for modelling abstract RTS combat called SparCraft, which can perform millions of unit actions per second and visualize them. We then present a modification of the UCT algorithm capable of performing search in games with simultaneous and durative actions. Finally, a novel greedy search algorithm called Portfolio Greedy Search is presented which uses hill climbing and accurate playout-based evaluations to efficiently search even the largest combat scenarios. We demonstrate that Portfolio Greedy Search outperforms state of the art Alpha-Beta and UCT search methods for large StarCraft combat scenarios of up to 50 vs. 50 units under real-time search constraints of 40 ms per search episode.**

## I. Introduction and Background

Multi-agent planning is an important sub-field of Artificial Intelligence research with many real-world applications that deal with agent cooperation, such as robotic search and rescue missions and unit coordination on a battlefield. Finding optimal actions for collections of agents if computationally intractable for all but the simplest planning tasks. For instance, finding shortest solutions in the generalized sliding tile puzzle — which can be regarded as a restricted multi-agent path-planning problem — is NP-hard [1]. When constructing planning systems involving multiple agents acting in the world, we therefore have to resort to approximations which hopefully generate useful results, even under real-time constraints.

Video games are fruitful application areas for multi-agent planning research. Often fast paced and featuring numerous game objects that can act independently, video games pose challenging research questions, such as:

1) How to quickly navigate groups of units on large maps, while staying in formation?
2) How to deal with imperfect information, such as unknown opponent locations?
3) How to cooperate well with team members?
4) How to detect and exploit opponent weaknesses?
5) How to coordinate attacks involving dozens of units?

Most planning tasks in video games are time critical — any delay can be costly. Consider, for instance, a combat unit targeting system. Finding most effective target sequences is PSPACE-hard [2]. So, if we actually took the time to solve non-trivial instances of this problem optimally while playing, we could see ourselves outplayed by a much faster scripted solution that employs simple strategies such as "attack the weakest opponent unit in weapon's range". It is therefore important to balance plan quality and planning time, and to strive for systems that are able to interleave planning with plan execution.

In recent years, developing AI systems for video games has gained attention in the AI research community due to the challenging problems these games pose, combined with the fact that human players still outperform AI systems in this application area, as demonstrated — for example — at the AIIDE 2012 workshop on Artificial Intelligence in Adversarial Real-Time Games [3], where a strong, but not quite world-championship level StarCraft player defeated the best StarCraft AI systems without any problem. StarCraft by Blizzard Entertainment is a popular real-time strategy game in which players try to defeat opponents by gathering resources, producing fighting units, and destroying their buildings on large playing fields in real-time.

With the advent of the BroodWar API (BWAPI, [4]) it is now possible to construct AI systems for StarCraft that directly communicate with the game engine and play against each other. Several StarCraft AI competitions are being held each year with a dozen entries or more [4]. While the best AI players are still rule-based systems, the number of competition entries utilizing more sophisticated AI techniques such as Bayesian inference and heuristic search is rising.

RTS game strategy can loosely be divided into macro and micro, which deal with high-level aspects such as economic development and technology advancement, and low-level aspects such obstacle avoidance and unit targeting in combat. When developing AI systems for complex video games it is tempting to use algorithms that have been shown well suited for adversarial abstract games such as Chess and Go, namely Alpha-Beta and UCT search. However, the astronomical search spaces render such an approach fruitless, unless one is willing to consider state or action abstractions, or one considers small sub-problems, such as small-scale battles.

In this article, we focus on the problem of finding good action sequences for a group of units engaged in combat with opponent units, where individual unit's actions consist of either targeting units within weapon range or moving around. A combat game is a series of simultaneous move rounds, in which units lose so-called hit points when being attacked and are removed when their hit point counts drop below 1. The player who first destroys all opponent's units wins. Playing such combat sub-games well is crucial for winning RTS games in confrontations between players with similar macro skills. One of the best StarCraft players of all time, Jaedong, attributed

his success to his unit control, saying "That micro made me different from everyone else in Brood War, and I won a lot of games on that micro alone". [5]

As shown in [2], [6], simple combat games, in which units can't move, are multi-round zero sum matrix games, which therefore — in principle — can be solved bottom up by linear programming. However, deciding whether there exists a pure winning strategy for simple combat games has been shown to be PSPACE-hard and in EXPTIME [2]. It is an open problem, whether this decision problem is PSPACE-complete. Solving large game instances under tight real-time constraints is therefore infeasible, and one needs to resort to approximations. Prior work on such approximations, including those for the full combat game featuring unit motion is discussed in Section III.

In this paper we focus on the combat game with unit motion. We present a new algorithm that performs much better than state-of-the-art Alpha-Beta and UCT based players in settings relevant to playing RTS games with dozens of units involved in combat. In what follows we first present the combat model we will be using in detail and describe basic scripted policies, state evaluations, and existing approximation algorithms. Then we present a modification of UCT tree search for use in RTS combat scenarios. Next, we introduce portfolio greedy search — a local search approach designed to overcome the enormous branching factor in multi-unit combat scenarios — which we then empirically compare with state-of-the-art search techniques. We conclude by discussing future work.

## II. RTS Game Combat Model

Before attempting to construct AI agents for combat in StarCraft, we must first construct a system which efficiently simulates the game itself. One way of performing StarCraft combat is through the BWAPI programming interface, which allows for interaction with the actual StarCraft game interface. Unfortunately, this method has a maximum speed of approximately 800 game frames per second, and does not allow us to control and manipulate local state instances directly. As one search episode may perform tens of thousands of moves with a real-time constraint of 40 ms, with each move having a duration of at least one simulation frame, we must construct an abstract model of StarCraft combat which efficiently implements moves in a way that does not rely on simulating each in-game frame.

### A. Combat Model: SparCraft

To simulate StarCraft game combat, we have created an open source simulation package called SparCraft. Since the source code for StarCraft is unavailable, this system is our best approximation as to how the game functions. Units can be given attack, move, and wait commands. All unit properties such as hit points, cool-down period, speed, size, armor, and weapon types are modelled exactly from StarCraft with the exception of acceleration, with all units having constant speed while moving. All upgrades and research are modelled. However, spell casters and units that contain other units (reavers, carriers, bunkers, transports) are not yet implemented. SparCraft does not yet implement unit collisions (to increase simulation speed) or fog of war, either.

For complete details of the system, its functionality and its limitations please consult [7]. SparCraft is comprised of three main data components and two main logic functions:

**State** $s = \langle \mathrm{t}, U_1, U_2 \rangle$
- Current game time $t$
- Sets of units $U_i$ under control of player $i$

**Unit** $u = \langle p, \mathrm{hp}, \mathrm{t}_a, \mathrm{t}_m, \mathrm{type} \rangle$
- Position $p = \langle x, y \rangle$ in $\mathbb{R}^2$
- Current hit points $\mathrm{hp}$
- Time step when unit can next attack $t_a$, or move $t_m$
- StarCraft unit $\mathrm{type}$, defining all static unit properties such as damage, maximum $\mathrm{hp}$, armor, speed, etc

**Move** $m = \langle a_1, \ldots, a_k \rangle$ 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 $\mathrm{type}$ of action to be performed: *Attack* unit $\mathrm{target}$, *Move* $u$ to position $\mathrm{target}$, or *Wait* until time $t$

**Player** function $p$ $[m = p(s, U)]$
- Input state $s$ and units $U$ under player's control
- Performs Move decision logic
- Returns move $m$ generated by $p$

**Game** function $g$ $[r = g(s, p_1, p_2)]$
- Initial state $s$ and players $p_1, p_2$
- Performs game simulation logic
- Returns game result $r$ (win, lose or draw)

Within this framework, once an AI agent is constructed for performing combat decisions it can be implemented in the system as a Player function.

## III. AI Systems for RTS Combat Games

We can categorize AI players for RTS combat games as one of two types: static scripted players, and more reactive and complex players based on search or learning techniques. The simplest and most common approach thus far are scripted players, which are currently the only solution implemented for retail RTS game AI, and RTS AI competition bots. We define a *script* $c$ in this context as a function $a = c(s, u)$ which given input state $s$ and unit $u$, performs a series of static scripted rules (similar to a finite state machine) to produce an action $a$ which is to be performed by $u$ in the next time step. A scripted player can then be thought of as implementing a sequence of scripts $\langle c_1, c_2, ..., c_n \rangle$ where $n$ is the number of units controlled by the player and $a_i = c_i(s, u_i)$ determines the action to be performed by unit $u_i$. Scripts $c_i$ may all be the same script function or be some combination of different scripts.

Some examples of commonly implemented scripts are:

- The *Attack-Closest* script in which 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. This script is (probably) used by the retail AI in StarCraft: BroodWar.
- The *Kiter* script is similar to Attack-Closest, except it will move a fixed distance away from the closest enemy when it is unable to fire.

- The *Attack-Value* strategy is similar to Attack-Closest, except units attack an opponent unit with the highest damage per frame / $\mathrm{hp}$ value within range when able. This choice leads to optimal play in 1 vs. $n$ scenarios [2].
- The *No-OverKill-Attack-Value* (NOK-AV) 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.

RTS games such as WarCraft 3, StarCraft, and StarCraft 2 appear to control their units via a single scripted player similar to Attack-Weakest. In the 2012 StarCraft AI Competition, the top 3 finishing AI bots: Skynet, Aiur, and UAlbertaBot each used script players for combat, with each unit type being guided by its own script, i.e.: DragoonBehavioursScript, MarineBehavioursScript, which each were small finite state machines consisting of scripts similar to Attack-Weakest and Kiter.

In [8] a combination of scripted actions and influence maps was used to perform kiting behaviour for small scale combat. Initially designing their system for use in 1 vs. 1 combat, it scaled well up combat of 4 fast kiting vulture units vs 6 slower scripted zealot units. However, their system was designed specifically for kiting scenarios, so it is not applicable to general large-scale RTS combat.

### A. Existing Search Techniques

As scripted player policies in RTS are highly exploitable [5] and unable to adapt to different situations, we are interested in more dynamic solutions which are more robust to changing opponents and combat settings. Two-player RTS combat can be classified as a two-player zero-sum simultaneous move game [5], and as such it is guaranteed to have a Nash equilibrium. Unfortunately, due to the PSPACE-hardness of finding pure winning strategies, approximations have to be found via other techniques. One method for doing this in classical games is minimax tree search, with the two most popular tree search algorithms being Alpha-Beta search and UCT search.

In [6], a method for simulating simultaneous moves for two player tree search was introduced, which instead of deciding on a move for each player simultaneously carries out two sequential moves, with action effects being delayed until after the second move. Delayed action effects are necessary to ensure that no player gains an advantage by being the first player chosen to act. For example, if two units are low on health and are able to shoot and kill each other at the same time, in a sequential move setting the first unit to act would kill the other without any retaliation. By delaying action effects until the second player has chosen a move in these sequential move pairs, then applying all effects at once, fairness is assured. In [5] a modification of Alpha-Beta search called Alpha-Beta Considering Durations (ABCD) was introduced which applied this technique for dealing with simultaneous move games, and performed very well against scripted solutions for small-scale combat scenarios. A version of UCT was applied to RTS combat in [9], however it was a simplified version of Wargus, an open-source WarCraft 2 clone in which only units of the same type (footmen) were considered. Due to the more complex nature of StarCraft we must make modifications to UCT which are similar to those of ABCD in order to use it for general RTS combat. This algorithm, called UCT Considering Durations (UCTCD) will be presentation in Section IV.

### B. State Evaluation and Playouts

As with many game search applications, state spaces are often too large to search completely, so heuristics must be employed to evaluate non-terminal states. In traditional games such as Checkers or Chess these heuristic functions often depend on expertly crafted formula based on intuitive notions such as game positioning, or material counts. In [6], several suggestions were made for formula-based evaluations for RTS combat scenarios. The best one described in the paper was the LTD2 evaluation formula which combines the sum of the square root of hit points remaining of each unit times their maximum damage rate:

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

It was shown in [5] that this formula is beaten by evaluation methods using deterministic script-based game playouts. By performing a playout using the same scripted policy for both players, we can estimate which player has an advantage at a given state. Intuitively, if both players continue playing from a given state with the same policy and one wins, it probably had a strategic advantage at the initial state.

## IV. UCT Considering Durations

Implementations of UCT tree search for traditional games like Chess or Go assume alternating moves where actions have identical durations. To apply UCT to simultaneous move games with durative actions we must modify UCT in several aspects. Algorithm 1 shows this modified algorithm, UCT Considering Durations, which consists of four main procedures. The first, UCTSearch takes as input an initial state $s$ and the player we are maximizing $p$ and as output returns the move to be performed by the player in the next time step. It chooses this move in the normal UCT fashion of repeated traversals through the game tree via the Traverse procedure. Traverse selects which child node to recurse through via the SelectNode procedure, and at unvisited or terminal nodes applies a playout evaluation of the resulting state. When Traverse visits a node for the second time it generates the children of that node, storing the parent of the node as well the move which generated it. As mentioned previously, we simulate performing simultaneous moves in tree search via two sequential moves, one for each player. To enable this, nodes in the search are labelled with one of three types. Type FIRST signifies a node which is the "first" such sequential move, with SECOND being the second move. Finally, a SOLO node signifies only one player is able to act. UpdateState only applies moves to states which are not of type FIRST, unless they are leaf nodes in the tree (which must be evaluated). Whenever a node of type SECOND is encountered, the move from its parent is applied at the same time as its own move, simulating simultaneous actions.

## V. Portfolio Greedy Search Algorithm

We introduce a new any-time greedy search algorithm for RTS combat micro called Portfolio Greedy Search. Search algorithms such as Alpha-Beta and UCT attempt to search as

**Algorithm 1** UCT Considering Durations

```
1: procedure UCTCD(State s)
2:     root ← new Node
3:     for i ← 1 to maxTraversals do
4:         Traverse(root, Clone(s))
5:         if timeElapsed > timeLimit then break
6:     return most visited move at root
7:
8: procedure TRAVERSE(Node n, State s)
9:     if n.visits = 0 then
10:        UpdateState(n, s, true)
11:        score ← s.eval()
12:    else
13:        UpdateState(n, s, false)
14:        if n.isTerminal() then
15:            score ← s.eval()
16:        else
17:            if !n.hasChildren() then
18:                generateChildren(s, n)
19:            score ← Traverse(SelectNode(n), s)
20:    n.visits++
21:    n.updateTotalScore(score)        ▷ w.r.t. player to move
22:    return score
23:
24: procedure SELECTNODE(Node n)
25:    bestScore ← −∞
26:    for child c in n.getChildren() do
27:        if c.visits = 0 then return c
28:        score ← c.totalScore / c.visits +
29:                K · sqrt(log(n.visits)/c.visits)
30:        if score > bestScore then
31:            bestScore ← score
32:            bestNode ← c
33:    return bestNode
34:
35: procedure UPDATESTATE(Node n, State s, bool leaf)
36:    if (n.type ≠ FIRST) or leaf then
37:        if n.type = SECOND then
38:            s.makeMove(n.parent.move)
39:        s.makeMove(n.move)
```

many actions as possible from a given state in order to cover a large portion of the search space. They then recursively search child nodes deeper into the tree in order to determine which actions at the root will yield beneficial future states. Move-ordering schemes such as those discussed in Subsecion VI-D can be implemented to reduce the branching factor, but they are still quite large. For RTS combat scenarios, the number of actions possible from any state is the combination of all possible actions by each unit, which is approximately $L^U$ where $L$ is the average number of legal moves per unit, and $U$ is the number of units which can act. Also an issue for traditional search techniques is inaccurate evaluations for non-terminal nodes, which has improved with the introduction of scripted playouts, but still suffers from the fact that these playouts apply a single script policy to every unit in the state. Portfolio Greedy Search deals with these issues in several ways:

- It reduces the number of actions searched for each unit by limiting them to actions produced by a set of scripts called a *portfolio*
- Instead of searching an exponential number of combinations of unit actions, it instead applies a hill-climbing technique to reduce this to a linear amount
- It does not perform any recursive tree search, but instead relies on accurate heuristic evaluations at the root node
- It improves the quality of heuristic evaluation by performing playouts with individually chosen unit-script assignments, rather than assuming all units follow the same policies during the playout.

### A. Algorithm

Portfolio Greedy Search takes as input an initial RTS combat state, a set of scripts to be searched called a *portfolio*, and two integer values $I$ and $R$. $I$ is the number of improvement iterations we will perform, and $R$ is the number of *responses* we will perform. As output it produces a player move, similar to the output of Alpha-Beta or UCT. The algorithm can be broken down into three main procedures:

- The main procedure PortfolioGreedySearch sets up the initial players and performs the main loops for improving the player policies. Players are initially *seeded* by the GetSeedPlayer procedure that returns an initial player which can then be improved upon via the hill-climbing Improve procedure. After we have improved our player, we can then improve our enemy by the same method, and re-improve our player based on the now stronger opponent. This process is repeated as many times as desired and the resulting player policy is returned.
- The GetSeedPlayer procedure can be seen on line 14. This procedure produces an initial policy to be implemented by all units the player controls. To do this, it iterates over all scripts in our portfolio, setting each unit's policy to the current script, and then perform a playout with each iteration. We then set our player's initial seed policy to the best performing script found via this process.
- The Improve procedure is the most important part of the Portfolio Greedy Search algorithm. Instead of searching an exponentially large combination of all possible unit actions, it instead uses a hill-climbing procedure to search over each script in our portfolio exactly once for each unit. At each iteration it performs a playout using the individual unit-script assignments, the result is recorded, and after each script has been applied to a unit, that unit's script is set to the best one found so far during the process.

## VI. EXPERIMENTS

Two main sets of experiments were carried out to compare the performance of ABCD, UCTCD, and the new Portfolio Greedy Search algorithms. The first set of experiments play ABCD vs. UCT, in order to show the comparative strength of the two baseline search algorithms. The second set of experiments then play ABCD and UCT vs. the proposed Portfolio Greedy Search algorithm to see how it performs against the current state of the art.

**Algorithm 2** Portfolio Greedy Search

```
Portfolio P                                   ▷ Script Portfolio
Integer I                                     ▷ Improvement Iterations
Integer R                                     ▷ Self/Enemy Improvement Responses
Script D                                      ▷ Default Script

procedure PortfolioGreedySearch(State s, Player p)
    Script enemy[s.numUnits(opponent(p))].fill(D)
    Script self[] ← GetSeedPlayer(s, p, enemy)
    enemy ← GetSeedPlayer(s, opponent(p), self)
    self = Improve(s, p, self, enemy)
    for r = 1 to R do
        enemy = Improve(s, opponent(p), enemy, self)
        self = Improve(s, p, self, enemy)
    return generateMoves(self)

procedure GetSeedPlayer(State s, Player p, Script e[])
    Script self[s.numUnits(p)]
    bestValue ← −∞
    Script bestScript ← ∅
    for Script c in P do
        self.fill(c)
        value ← Playout(s, p, self, e)
        if value > bestValue then
            bestValue ← value
            bestScript ← c
    self.fill(bestScript)
    return self

procedure Improve(State s, Player p, Script self[],
                                              Script e[])
    for i = 1 to I do
        for u = 1 to self.length do
            if timeElapsed > timeLimit then return
            bestValue ← −∞
            Script bestScript ← ∅
            for Script c in P do
                self[u] ← c
                value ← Playout(s, p, self, e)
                if value > bestValue then
                    bestValue ← value
                    bestScript ← c
            self[u] ← bestScript
    return self
```

### A. Combat Scenario Setup

Each experiment consists of a series of combat scenarios in which each player controls an identical group of $n$ StarCraft units. To show how each algorithm performs in large combat scenarios, each experiment was repeated for values of $n$ equal to 8, 16, 32, and 50, 50 being roughly the size of the largest battles seen in a typical game of StarCraft. Further, two different geometric configurations of the initial unit states were used:

- *Symmetric* states, in which units for each player are placed randomly symmetric about the midpoint $m$ of the battlefield. For each unit in position $m + (x, y)$ for player 1, player 2 receives the same unit at position $m+(-x, -y)$. This ensures a fair initial starting position, but one which would not typically be seen in an RTS combat setting.
- *Separated* states were designed to more closely resemble an actual RTS combat scenario. A midpoint $m$ for the battlefield is chosen, and then each player's force is generated randomly symmetric to the midpoint, and then translated a fixed distance $d$ to the left or right. For example, a unit for player 1 generates a random $(x, y)$ position and is placed at location $m + (x - d, y)$ with player 2's identical unit being placed at position $m + (-x + d, -y)$. Distance $d$ was chosen so that it is larger than the largest attack radius of any unit, so that both groups of unit are separated before attacking begins, simulating two opposing forces clashing on a battlefield. Each separated state is generated twice, with each force appearing once on the left and once on the right, for fairness.

For both symmetric and separated states, random positions $(x, y)$ were generated with bounds of $x, y \in [-128, 128]$ pixels. This kept a decent spacing of starting units, while mimicking the tight formation of a typical group of units in a combat scenario. The battlefield itself was an enclosed arena with width 1280 pixels and height 720 pixels, with midpoint position $m = (640, 360)$. Units were free to move anywhere within the arena, but could not move through the "walls" at the outer edges. An enclosed arena was used to ensure that each battle eventually terminated, as an infinite plane resulted in many cases of one player simply running away from a fight indefinitely.

Although movement in SparCraft can be performed in any direction, for our experiments we limit movement to only allow fixed length movements up, down, left, or right. This abstraction is necessary to reduce the search space for each algorithm. Although this abstraction may seem quite coarse, by setting a small movement length of 8 pixels the movement of units in the simulator appears quite similar to the actual game of StarCraft.

For each set of experiments, 5 different different configurations of starting unit types were also used to simulate various RTS army compositions with both melee and ranged units of different strengths. Also, early game units were used as they are by far the most commonly seen units in StarCraft combat. The following were used as starting unit type counts for each player for each battle of size $n$ units:

- $n$ Protoss Dragoons (Strong Ranged)
- $n$ Zerg Zerglings (Weak Melee)
- $n/2$ Protoss Dragoons (Strong Ranged) with $n/2$ Protoss Zealots (Strong Melee)
- $n/2$ Protoss Dragoons (Strong Ranged) with $n/2$ Terran Marines (Weak Ranged)
- $n/2$ Terran Marines (Weak Ranged) with $n/2$ Zerg Zerglings (Weak Melee)

100 randomly generated battles were carried out for each of the 5 starting unit configurations, giving 500 total battles for each separated state and for each symmetric state experiment for each tested value of $n$ starting units.

### B. Environment and Search Settings

All experiments were performed on an Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz running Windows 7 Professional Edition, with all algorithms running single-threaded. A total of 12 GB DDR3 1600MHz RAM was available, however the maximum amount of RAM consumed by any process monitored at less than 14 MB, which was used to store both the UCT search tree and the Alpha-Beta transposition table. Experiments were programmed in C++ and compiled using Visual Studio 2012.

### C. Search Algorithm Parameters

Each search algorithm was given a 40 ms time limit per search episode to return a move at a given state. This time limit was chosen to mimic real-time performance in StarCraft, which runs at 24 fps (42 ms per frame). Alpha-Beta and UCT search algorithms were given an upper limit of 20 children per search node. Due to the exponential number of possible actions at each search state, having no upper bound on the number of children at a node would often produce searches which did not leave the root node of a tree, which produced very bad results. In practice we found that imposing a child limit, when combined with clever move-ordering (next section) produce best results.

- Alpha-Beta search:
  - Time Limit: 40 ms
  - Max Children: 20
  - Evaluation: NOK-AV vs. NOK-AV Playout
  - Transposition Table Size: 100000 (13.2 MB)
- UCT search:
  - Time Limit: 40 ms
  - Max Children: 20
  - Evaluation: NOK-AV vs. NOK-AV Playout
  - Final Move Selection: Most Visited
  - Exploration Constant: 1.6
  - Child Generation: One-at-leaf
  - Tree Size: No Limit (6 MB largest seen in 40 ms)
- Portfolio Greedy search:
  - Time Limit: 40 ms
  - Improvement Iterations $I$: 1
  - Response Iterations $R$: 0
  - Initial Enemy Script: NOK-AV
  - Evaluation: Improved Playout
  - Portfolio Used: (NOK-AV, Kiter)

Of note is the choice of low settings for $I = 1$ and $R = 0$. These were chosen for two reasons: first, to show the performance of the base settings for Portfolio Greedy Search, and also because higher settings do not yet run within 40 ms.

### D. Move Ordering

It is well known that with game tree search algorithms such as Alpha-Beta or UCT, a good move-ordering scheme can greatly improve performance. If better moves are searched first, Alpha-Beta can produce better cuts and search deeper, while if UCT searches better nodes first, it will spend less time exploring less valuable moves. With a child limit imposed on our search, we must ensure that the moves we search are useful, and we do this in several ways. At each search node, both Alpha-Beta and UCT first search the moves generated by our NOK-AV and Kiter scripts. These moves are then followed by moves containing Attack actions, then by moves containing Movement actions. Movement actions are explored in random order for fairness. In the case of Alpha-Beta we also consider moves which have been stored in the transposition table.

### E. Opponent Modelling

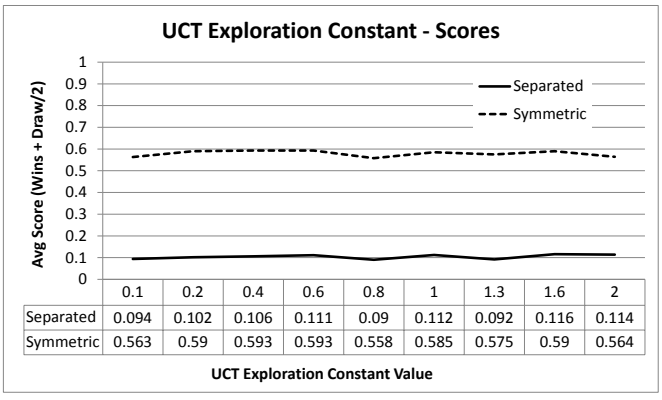
Experiments involving Alpha-Beta (AB) and UCTCD (UCT) were conducted with two opponent-modelling parameter settings: either all opponent actions were searched in the game tree, or opponent actions were fixed to that of the NOK-AV script. By fixing the enemy actions, we are effectively calculating a "best response" to that script (opponent modelling) in an attempt to exploit it. This was shown to give a substantial performance against scripted opponents in [5], and so we tested to see if it would have any effect against Portfolio Greedy Search, which searches over scripted moves. If these best response searches are found to increase results against Portfolio Greedy Search, then this will be a significant weakness in the algorithm.

## VII. Results

Before results were run, parameter optimization was performed on the exploration constant K of the UCT algorithm (Algorithm 1, line 29) to ensure good performance in our experiments. The results from this optimization can be seen in Fig. 1, which determined that the choice of constant did not highly affect results in either the symmetric or separated state experiments against Alpha-Beta. We chose a value of 1.6, which was the value with the highest result sum from both experiments.

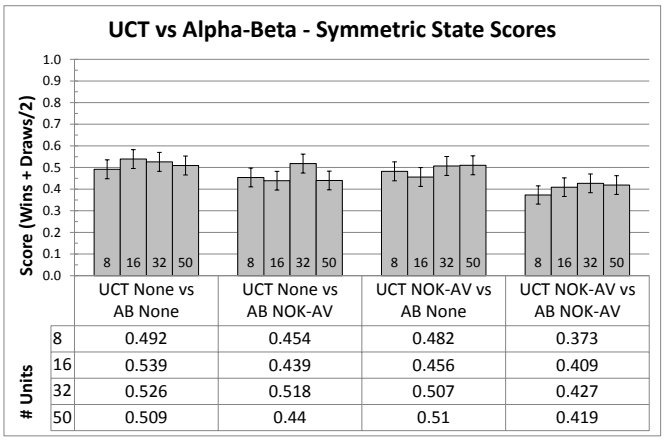
### A. Search vs. Script

Experiments were performed with Alpha-Beta, UCT, and Portfolio Greedy Search against each script type listed in III, with all 3 search techniques achieving a win rate of 100% against scripted players for all battle sizes.




| | 0.1 | 0.2 | 0.4 | 0.6 | 0.8 | 1 | 1.3 | 1.6 | 2 |
|---|---|---|---|---|---|---|---|---|---|
| Separated | 0.094 | 0.102 | 0.106 | 0.111 | 0.09 | 0.112 | 0.092 | 0.116 | 0.114 |
| Symmetric | 0.563 | 0.59 | 0.593 | 0.593 | 0.558 | 0.585 | 0.575 | 0.59 | 0.564 |

Fig. 1: Average scores for various settings of UCT exploration constant $K$. Experiments were performed vs. Portfolio Greedy Search with 8, 16, 32, and 50 starting units for both separated and symmetric states. $K = 1.6$ was chosen for the paper's main experiments.



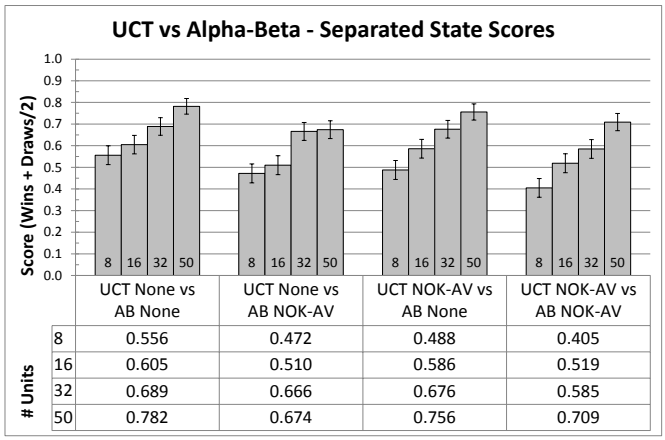
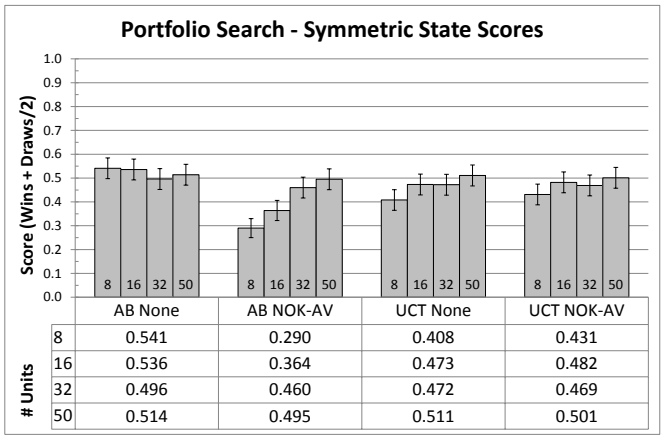




Fig. 2: Results of Alpha-Beta vs. UCT for Symmetric States (left) and Separated States (right). Both algorithms have two configurations, one without opponent modelling labelled "None", and with modelling against script NOK-AV. Results are shown for combat scenarios of $n$ vs. $n$ units, where $n = 8, 16, 32, 50$. 500 combat scenarios were played out for each configuration. 95% confidence error bars are shown for each experiment.



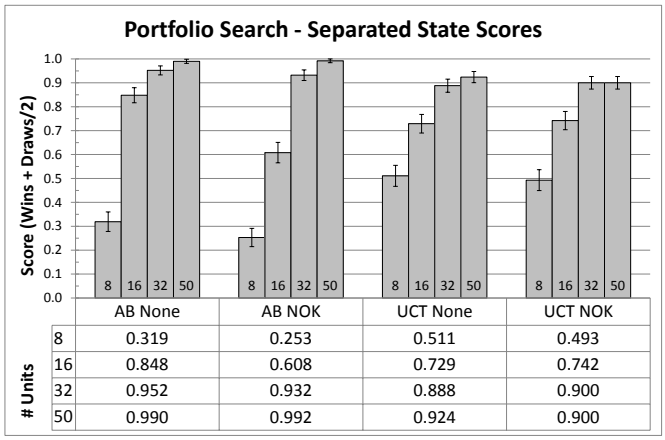




Fig. 3: Results of Portfolio Greedy Search vs. Alpha-Beta and UCT for Symmetric States (left) and Separated States (right). Both algorithms have two configurations, one without opponent modelling labelled "None", and with modelling against script NOK-AV. Results are shown for combat scenarios of $n$ vs. $n$ units, where $n = 8, 16, 32, 50$. 500 combat scenarios were played out for each configuration. 95% confidence error bars are shown for each experiment.

### *B. UCT vs. Alpha-Beta*

The results from the UCT vs. Alpha-Beta experiment can be seen in Fig. 2. Immediately one notices the dramatic difference in the result between symmetric state and separated state types. Experiments performed in symmetric states tend to show equal performance between both algorithms, except for the case where both UCT and Alpha-Beta are configured to compute a best response to the NOK-AV script. Experiments on separated states (the more realistic of the two types) show that for small battles, both methods perform equally well, but UCT outperforms Alpha-Beta as the battles grow larger.

A possible explanation for the difference in results between the two state types is intuitive: in symmetric states, units are usually within firing range of many other units, and since there is a small reload-speed penalty for moving (as is present in StarCraft), the problem reduces almost entirely to a unit-targeting problem. By almost completely eliminating the need for clever movement, neither search algorithm can gain an advantage over the other through search. For separated states, there is much more room for clever tactics such as kiting, retreating when at low health, group formations, etc. Since both search algorithms are given identical action spaces to search, this shows that the UCT algorithm is better suited for larger RTS combat scenarios than Alpha-Beta.

### *C. Portfolio Greedy Search*

Results from the Portfolio Greedy Search algorithm can be seen in Fig. 3. As in the previous experiment, the results for symmetric states are fairly even, with the exception versus the Alpha-Beta algorithm which computes a best response to NOK-AV. Because NOK-AV is one of the two scripts in the portfolio and symmetric states tend to favour no movement, NOK-AV will be the script chosen by the greedy search the majority of the time. As shown in [5], this type of best response computation can be quite powerful in exploiting scripted behaviours. However, these results also show that UCT does far worse than Alpha-Beta at performing this exploitation.

The separated state results show that the portfolio greedy search algorithm easily defeats Alpha-Beta and UCT for larger state sizes. While performance is weak for 8 vs. 8 units, as

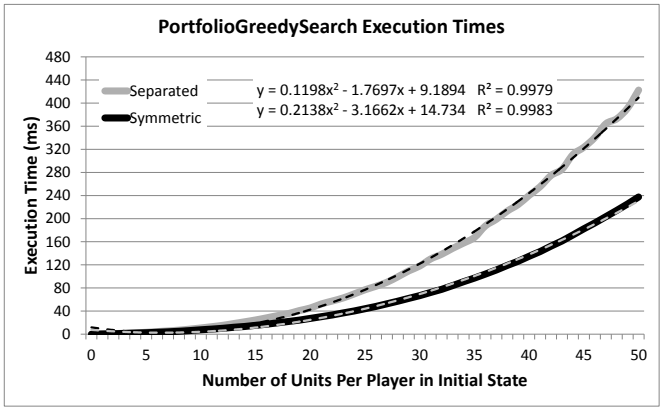



Fig. 4: Graph showing average execution times of complete Portfolio Greedy Search search episodes with respect to the number of units in the combat scenario when no time limit is specified. Execution times are extracted from the first move from the initial symmetric or separated states. Sample standard deviations for symmetric state running times for different unit numbers are: 10 units: 2.3 ms, 25 units: 9.0 ms, 50 units: 55.5 ms, and for separated states: 10 units: 2.2 ms, 25 units: 19.7 ms, 50 units: 111.5 ms.

combat scenarios increase in size it dominates the traditional search algorithms, winning nearly all battles against Alpha-Beta and more than 90% of battles against UCT.

Fig. 4 shows average execution times of complete Portfolio Greedy Search search episodes with respect to the number of units in a separated state scenario, if no time limit had been specified. This graph illustrates the quick running time of the Portfolio Greedy Search algorithm with respect to traditional tree search methods which would require vast computational resources to fully search large scenarios. We can see that the time limit of 40 ms was only reached when performing searches on states with more than $2 \times 25$ units. Of note is the quadratic running time with respect to the number of units in the scenario, which one would expect to be linear due to nature of the algorithm. This is explained by the use of playouts for state evaluations whose running times are themselves linear with respect to the number of units in a scenario, due to the need for an action to be calculated for each unit. Execution times were recorded only for the first move of symmetric and separated states in order to illustrate their differences, which exist due to the underlying scripts in the portfolio. Since the scripts are optimized to choose attack actions before move actions, they encounter their worst-case running time on initially separated states in which no attack options are found, forcing all move options to be explored. However, once both opposing forces of a separated state engage in battle, their values approach that of symmetric states (on average for the duration of the battle).

## VIII. Conclusion and Future Work

In this paper we have presented a modified version of UCT for handling games with simultaneous and durative actions, as well as a new greedy search algorithm for RTS combat: Portfolio Greedy Search. We have implemented and shown experimental results comparing Alpha-Beta, UCT, Portfolio Greedy Search, for use in RTS game combat scenarios. We have shown that UCT outperforms Alpha-Beta in battle scenarios with realistic unit positions (separated states) as battle sizes get larger. We have also shown that the new Portfolio Greedy Search algorithm outperforms both Alpha-Beta and UCT for medium to large size separated state battle scenarios, winning over 90% of battles with more than 32 units.

Several improvements can be made to the Portfolio Greedy Search algorithm which can improve both its speed and results. Using portfolio $P$, Portfolio Greedy Search performs $|P|$ playouts per unit per search. These playouts could be trivially parallelised, allowing a linear speed-up in running time with respect to $|P|$. In our case, using a two script portfolio, this would yield a 100% speed increase in the algorithm.

To improve performance of Portfolio Greedy Search, extra decision points (iterating over scripts for each unit) could be created in the search tree to improve the accuracy of the evaluation. Unlike tree search methods, Portfolio Greedy Search only optimizes decision at the root node before performing its playout evaluation. By implementing a scheme in which extra search is performed after a certain number of moves have been performed in the playout, it could improve performance. We can then imagine a hybrid tree search algorithm in which Portfolio Greedy Search is the method used by a minimax type algorithm to choose which moves to play at a given node in the tree. Portfolio Greedy Search would need to be significantly faster in order to be used within another tree search algorithm, but it is not outside the realm of possibility.

It is our intention to use Portfolio Greedy Search for combat decision making in a future version of UAlbertaBot, our entry to the StarCraft AI Competition. By examining the results in Fig. 3, we can see that while Portfolio Greedy Search performs quite well for larger combat scenarios, it is beaten by Alpha-Beta for smaller scenarios. We can now envision a hybrid AI agent which dynamically chooses which search method to use based on size of the combat scenario presented. Because each algorithm has its strengths and weaknesses, creating an agent which is able to capitalize on all of the strengths with none of the weaknesses seems like the most intelligent choice for future competitions.

## References

[1] D. Ratner and M. K. Warmuth, "Finding a shortest solution for the n $\times$ n extension of the 15-puzzle is intractable," in *AAAI*, T. Kehler, Ed. Morgan Kaufmann, 1986, pp. 168–172.

[2] T. Furtak and M. Buro, "On the complexity of two-player attrition games played on graphs," in *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2010*, G. M. Youngblood and V. Bulitko, Eds., Stanford, California, USA, Oct. 2010.

[3] AIIDE 2012, "Workshop on artificial intelligence in adversarial real-time games," https://skatgame.net/mburo/aiide12ws/.

[4] BWAPI, "Broodwar API," http://code.google.com/p/bwapi/.

[5] D. Churchill, A. Saffidine, and M. Buro, "Fast heuristic search for RTS game combat scenarios," in *AIIDE*, M. Riedl and G. Sukthankar, Eds. The AAAI Press, 2012, pp. 112–117.

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

[7] D. Churchill, "SparCraft: open source StarCraft combat simulation," http://code.google.com/p/sparcraft/.

[8] A. Uriarte and S. Ontañón, "Kiting in RTS games using influence maps," in *AIIDE Workshop on Workshop on Artificial Intelligence in Adversarial Real-Time Games*, 2012, pp. 31–36.

[9] R.-K. Balla and A. Fern, "UCT for tactical assault planning in real-time strategy games," in *IJCAI*, 2009, pp. 40–45.