Robust Continuous Build-Order Optimization in StarCraft

David Churchill
Computer Science
Memorial University
St. John's, NL, Canada
dave.churchill@gmail.com

Michael Buro
Computing Science
University of Alberta
Edmonton, AB, Canada
mburo@ualberta.ca

Richard Kelly
Computer Science
Memorial University
St. John's, NL, Canada
richard.kelly@mun.ca

***Abstract*—To solve complex real-world planning problems it is often beneficial to decompose tasks into high-level and low-level components and optimize actions separately. Examples of such modularization include car navigation (a high-level path planning problem) and obstacle avoidance (a lower-level control problem), and decomposing playing policies in modern video games into strategic ("macro") and tactical ("micro") components. In real-time strategy (RTS) video games such as StarCraft, players face decision problems ranging from economic development to maneuvering units in combat situations. A popular strategy employed in building AI agents for complex games like StarCraft is to use this strategy of task decomposition to construct separate AI systems for each of these sub-problems, combining them to form a complete game-playing agent. Existing AI systems for such games often contain build-order planning systems that attempt to minimize makespans for constructing specific sets of units, which are typically decided by hand-coded human expert knowledge rules. Drawbacks of this approach include the human expert effort involved in constructing these rules, as well as a lack of online adaptability to unforeseen circumstances, which can lead to brittle behavior that can be exploited by more advanced opponents. In this paper we introduce a new robust build-order planning system for RTS games that automatically produces build-orders which optimize unit compositions toward strategic game concepts (such as total unit firepower), without the need for specific unit goals. When incorporated into an existing StarCraft AI agent in a real tournament setting, it outperformed the previous state-of-the-art planning system which relied on human expert knowledge rules for deciding unit compositions.**

I. Introduction

Real-Time Strategy (RTS) games have become a popular test-bed for modern state-of-the-art artificial intelligence systems. With their real-time computational constraints, imperfect information, simultaneous moves, and extremely large state and action spaces, they have proven to be an excellent domain for testing AI systems in preparation for real-world scenarios [1]. In the past few years, large companies such as Google DeepMind, Facebook AI Research, and OpenAI have each been developing AI systems for strategic video games, which could lead to more powerful AI solutions for their real-world industry problems [2].

The goal of any RTS game is to defeat the forces of your enemy, and in order to achieve that goal a player must first construct an army with which to fight. These armies must be built by the player by first using worker units to gather resources, then using these resources to construct additional buildings and infrastructure, which can then produce an army of combat units such as soldiers or tanks. The sequence of actions taken to arrive at a given set of goal units is called a *build-order*. Professional human players learn and/or memorize several proven build-order sequences for the initial few minutes of a game, which they then later adapt on the fly based on information obtained about their opponent. Build-order planning and resource gathering make up the economic, or "macro" (as it is known to RTS players) part of the game, which is the most important aspect of overall strategic play. Widely considered the best StarCraft player of all time, Lee "Flash" Young Ho was known for his strong macro play, which many have said was the key to his dominance throughout his career[1]. It is therefore very important that any competitive StarCraft AI agent must have a strong macro decision making systems, for which build-order planning plays a large part.

The earliest solutions for StarCraft AI build-order planning were simple rule-based systems in which human experts assigned priority values to various in-game unit types, which were then constructed in order of priority. The most well-known of these solutions was the BWAPI Standard Add-on Library (BWSAL)[2], which allowed AI programmers to use hard-coded priorities to construct given unit types, however the build-orders it produced were quite slow in comparison to expert humans. Since then, several automated planning systems have been developed which attempt to minimize total build-order construction time (makespan), however they all require that a goal set of units be given to the underlying search algorithm, usually given by human expert knowledge in the form of a rule-based system. This reliance on human-coded rules for unit goals ultimately makes such systems less adaptable to online observations within a game, and come at a significant authorship cost. It is our goal to construct a build-order planning system which relies on as little human-coded information as possible, which can adapt automatically to its environment during a game.

In this paper we introduce a new robust build-order planning system for RTS games that automatically produces build-orders which optimize unit compositions toward strategic

[1]https://liquipedia.net/starcraft/Flash
[2]https://github.com/Fobbah/bwsal

978-1-7281-1884-0/19/$31.00

game concepts (such as total unit firepower) continuously throughout the game, without the need for specific unit goals, with the option for specifying additional constraints if desired. In the following section we will describe background and related work in the area of build-order planning for RTS games. We then give the details of our proposed new robust continuous build-order planning system, and the experiments carried out to compare it to previous state-of-the-art methods. We then finish with a discussion of the results, and ideas for future work in the area.

II. Background and Related Work

In this section we summarize previous work on build-order optimization and describe in more detail the formalism and branch-and-bound algorithms presented in [3] on which our work is based.

A. Prior Work on Build-Order Optimization

Build-order optimization problems are constraint resource allocation problems featuring concurrent actions for which we seek to minimize makespans to achieve given build goals. Initial work on build-order optimization in the context of RTS games focused on modeling them in PDDL so that solutions could be found by off-the-shelf planning software [4], and on studying ordering heuristics in concurrent action execution environments [5]. Build-order optimization problems were first tackled by employing means-end analysis (MEA), followed by a heuristic rescheduling phase to shorten makespans [6]. This method generates satisficing action sequences quickly, but the resulting plans are not necessarily optimal. Their technique was extended in [7] by incorporating best-first search and solving intermediate goals to reduce makespans further. Branquinho and Lopes [8] extended this line of work by combining two new techniques called MeaPop (MEA with partial order planning) and Search and Learning A* (SLA*). Their results improve on the makespans generated by MEA but are too slow to be used in real-time games. To address this problem a branch-and-bound algorithm was introduced in [3] which presents various heuristics and abstractions that reduce the search effort for solving build-order problems in complex games such as STARCRAFT significantly while producing near optimal plans in real-time. Their build-order search system (BOSS) software[3] is freely available and being used in many STARCRAFT AI systems. Because our robust build-order optimization system is based on BOSS and uses its simulator we will describe it in more detail in the next subsection.

In recent years genetic algorithm based approaches have also been successfully applied to build-order optimization problems. Blackford and Lamont [9] present a precise mathematical formalization of the heuristics presented in [3] and solve the resulting multi-objective optimization problem with a genetic algorithm. [10] show how full-game strategies can be evolved including macro- and micro-level decisions. The evolved strategies, however, are fixed, and therefore cannot be adjusted during gameplay. Justesen and Risi [11] solve this problem by applying continual online evolutionary planning. Their method is able to defeat basic scripted STARCRAFT bots, but relies on a complex approximate economic forward model and a handcrafted fitness function.

B. BOSS

The focus of build-order optimization is on actions related to producing units and structures in the shortest possible time. More formally, we want to minimize the makespan of concurrent action plans that transform the current abstracted RTS game state into a given abstracted goal state with certain properties. To make real-time planning feasible, classic build-order planning abstractions disregard the opponent and concentrate on unit counts and resource gathering aggregate statistics rather than low-level game state features such as unit positions and actual resource gatherers' travel times. In a typical STARCRAFT build-order planning scenario we start with 4 workers, sufficient resources (which can be gathered by workers), and a resource depot, and ask the planner to compute the build-order which produces a given goal set of units with the shortest makespan (e.g., a goal of 6 Terran Marines with the intent to rush the opponent's base).

The build-order search system BOSS [3] which we will use in this work contains a simulator which models economic features of STARCRAFT. Its simulator uses resource income abstractions, macro actions, and multiple lower bound makespan heuristics which reduce the search space significantly. In addition, a fast-forwarding approach decreases the branching factor and eliminates the need for addressing the subset action selection problem which arises in concurrent action planning problems. Using these enhancements the depth-first branch-and-bound planner included in BOSS is capable of producing plans in real-time which are comparable to professional STARCRAFT players. BOSS is an open source software project and part of the UAlbertaBot StarCraft AI agent [12]. For more details we refer the reader to the original paper.

C. AlphaStar

Recently, Google DeepMind have had success in applying Deep Reinforcement Learning to the full game of StarCraft II without the need for the same technique of task decomposition which has so far been used in the RTS AI research community. The agent they created is named AlphaStar[4], and plays the game of StarCraft II at a professional (but not world champion) human level on a single small 2-player map, with a single race combination (Protoss vs. Protoss). While full details have not been released about the exact methods used, they have claimed that their research team consisted of more than 30 Google DeepMind employees, which when coupled with the extraordinary amount of computation required to train their agent yields a lower bound cost of several million dollars to accomplish this task. While these results are impressive, the

[3]https://github.com/davechurchill/ualbertabot

[4]https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/

excessive costs and computation time mean that video game companies cannot yet use these methods for developing AI systems for retail games. In order for an AI method to be useful in current commercial RTS video games they must be faster and cheaper to develop than AlphaStar, and ideally run in real-time - properties held by the new method we propose.

III. Robust Continuous Build-Order Optimization

Previous work on build-order optimization has focused on minimizing the makespan of concurrent action plans that achieve given unit counts. This is useful in situations for which goals are known, such as scripted high-level AI systems, because achieving unit production goals faster usually correlates with better performance in competitive real-time domains. However, formulating and testing high-level strategies including subgoals is a non-trivial task which requires considerable domain knowledge and time. So, an interesting research question is whether heuristic search can be used in a more direct and abstract fashion that requires less domain knowledge and is less dependent on subgoal engineering.

In the domain of RTS games, or more generally military-style confrontations, firepower advantage is highly correlated with winning. This suggests that it may be possible to construct AI systems for such domains by applying heuristic search to abstract concepts such as firepower maximization. Concretely, instead of achieving specific unit-count goals (such as producing 6 tanks) we may be able to design an optimizer that finds concurrent action sequences which maximize firepower directly. When doing this continually while taking scouting information about the enemy into account, there is a good chance that such an optimizer can discover ways of increasing firepower which have escaped human AI script designers or subgoal authors.

For this approach to work we need to concretely specify what we mean by "firepower", and how to optimize it. The following subsections address these tasks.

A. Army Value Functions

For our initial feasibility study we chose STARCRAFT as our application domain, due to its long-established competitiveness in both human and AI tournaments. In this popular RTS game unit firepower and abilities have been shown to be well correlated (balanced) to the unit's overall resource cost, meaning that the amount of resources spent on units is a good game-state independent proxy for the ability to deal damage to enemy units. In the experiments reported later we use the army value (AV), i.e., the resource aggregate spent on military units (non-worker units which can both move and attack), as the firepower measure to be maximized. This is a coarse approximation which disregards weapon types (e.g. anti-air/anti-ground) and close-range and area effects, which determine the actual unit effectiveness against a set of enemy units. Constructing more robust and dynamic AV functions is left to future work.

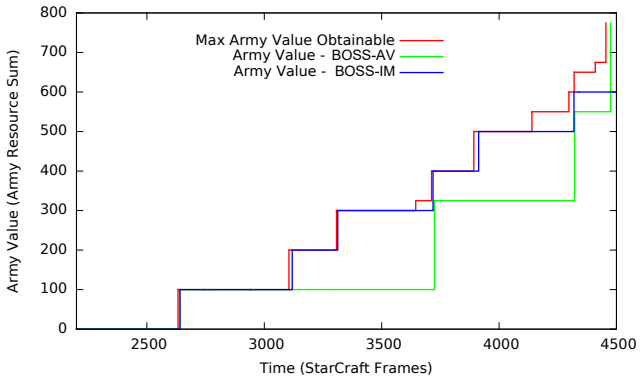

Fig. 1. Shown are three lines which demonstrate the results of army value maximization build-order search, up to a maximim of 4500 STARCRAFT game frames. The red line is the maximum possible army value obtainable by any build-order at a given time. The green line is the army value at any given time for the single build-order which maximizes the army value at time 4500 (BOSS-AV with $T = 4500$). The blue line is the army value for the single build-order which maximizes the area under the army value curve (BOSS-IM with $T = 4500$).

B. Army Value Optimization

The AV functions we just described are one-sided heuristic evaluation functions that allow us to use efficient single-agent search to optimize actions sequences (build-orders) for individual players. To be useful in 2-player games requires that maximizing AV correlates with maximizing the AV difference between both players, which itself is often correlated with winning. At least during the opening phase of RTS games before player units interact, the above assumption is warranted. For the single-agent AV optimization to also work well in later game stages, AVs should reflect relative strength against opponent units the player is aware of.

Assuming that single-agent AV maximization is useful, what does an optimal action sequence look like when planning to maximize AV for a set number of time steps T in the future? If T is long we expect optimal build-order plans to create a large army by initially producing many worker units to increase income, then build factories, and only in the final stages to produce many attack units in parallel. The green line in Figure 1 depicts a graph of AV over time for a build-order which maximizes AV at $T = 4500$ BOSS simulation frames (184 seconds in STARCRAFT game time) in which the AV more than doubles during the last 200 frames. The red line shows the maximum possible AV obtainable by any build-order at any time, calculated by running BOSS-AV for each time step T. It is apparent that using the first (green) build-order computed by the BOSS-AV algorithm (which maximises AV at $T = 4500$) can be easily countered by optimizing for $T = 3500$ and attacking at that time, since it will have an AV of 300 vs. 100 and have a large military advantage. In general, for each fixed time T there may be an earlier time for the opponent to optimize AV for, attack, and win, making BOSS-AV highly exploitable.

To overcome this exploitability, we could define a 2-player

Algorithm 1 BOSS Recursive DFS (Generalized)

Require: Initial State I, Frame T, TimeLimit L
Require: BuildOrder $B \leftarrow$ EmptyStack()
Require: BuildOrder $Best \leftarrow$ EmptyStack()

```
1:  procedure BOSS-RDFS(State S)
2:      if S_t > T or TimeElapsed ≥ L then
3:          return                          ▷ search termination criteria
4:      if Eval(B, I) > Eval(Best, I) then
5:          Best ← B
6:      for Action a ∈ S.legalActions do
7:          S' ← S.doAction(a)
8:          B.push(a)
9:          BOSS-RDFS(S')
10:         B.pop()
```

Algorithm 2 BOSS-AV Max Army Value Evaluation

```
1: procedure EVAL-AV(Initial State S, BuildOrder B)
2:     for Action a ∈ B do
3:         S ← S.doAction(a)
4:     return ArmyValue(S)
```

Algorithm 3 BOSS-IM Army Integral Evaluation

```
1:  procedure EVAL-IM(Initial State S, BuildOrder B)
2:      IntegralValue I ← 0
3:      PreviousActionTime T ← S_t
4:      PreviousArmyValue V ←ArmyValue(S)
5:      for Action a ∈ B do
6:          S ← S.doAction(a)
7:          δ ← S_t − T
8:          I ← I + δ × V
9:          V ←ArmyValue(S)
10:         T ← S_t
11:     return I
```

AV game in which two players pick their respective optimization time T, optimize their AV accordingly, and engage in battle at the earliest time, and compute a Nash equilibrium strategy for both players. However, the dynamics of a full-fledged RTS game are not that simple, and we may not have time to compute a Nash equilibrium in real-time. Instead, we concentrate on more robust build-orders that can be found with single-agent search. Our idea is to instead find a build-order with smaller exploitation potential by minimizing the area between its AV curve and the red line whose AV values (by definition) are maximal. This is equivalent to maximizing the area under the AV curve. We call this integral maximization algorithm BOSS-IM. The blue line in Figure 1 depicts the BOSS-IM AV curve for a build-order which maximizes the area under its AV curve up to $T = 4500$. This BOSS-IM build-order is more robust than the BOSS-AV solution for $T = 4500$, since it obtains a max AV very near that of BOSS-AV at the final T, while also constructing army units much earlier in the game, leaving it less vulnerable to early attack. There may also be other exploitation metrics (such as the maximum distance between both curves) that we could try to optimize, but we will leave studying those again to future work and just use the more robust BOSS-IM for our experiments in Section IV.

C. Search Algorithm & Evaluation Functions

To implement both BOSS-AV and BOSS-IM, the same high-level search algorithm can be used, with calls to different evaluation functions to maximize for AV or IM. This generic search algorithm can be implemented in a number of different ways, and is shown in Algorithm 1 as a recursive depth-first search (DFS) function. On line 4 of this algorithm, an evaluation function is called in order to determine the value which is to be maximized. Line 7 of the algorithm constructs a new state S', which is the result of issuing action a at the input state S. The action performed is then pushed onto the build-order stack B, and the function is recursively called. After a recursive level unrolls, the action is popped from the stack, which results in the current build-order being stored in variable B at all times, the best of which is stored in variable $Best$ which is returned after the search episode has terminated.

To implement BOSS-AV or BOSS-IM, this search algorithm calls the specific evaluation functions Eval-AV or Eval-IM, shown in Algorithms 2 and 3 respectively. Eval-AV accepts as input an initial game state S and a build-order B, performs all actions of the build-order on the state, and returns the AV computed at the final state. Eval-IM computes the area under the AV curve by computing the time deltas between issuing each action, multiplying by the previous state's AV, and summing over all actions.

D. RTS AI Agent Architecture

The algorithms presented in this section could be integrated into an RTS AI agent in a number of ways, either as a mid-level module which can replace an existing build-order planning system, or as a more high-level architecture for the rest of a bot to be based upon. One of the main advantages of BOSS-IM is that it frees RTS AI bot authors from spending time on authoring specific unit goals or strategic army compositions by producing units which automatically suit the current state of the game. As a mid-level module replacement, BOSS-IM could take the place of existing build-order planning systems such as the previously mentioned BWSAL, or more complex search-based planning systems like the one present in current StarCraft AI agents such as UAlbertaBot (explained further in Section IV-A). As a high-level architecture, an RTS AI agent could be made which instead of using predetermined strategies that dictate unit compositions, uses a system like BOSS-IM to produce units it feels are best for a given situation, and then act appropriately with those units. For example, if the algorithm produced more worker units it would use them to expand economically, or if it produced more military units it would decide to attack.

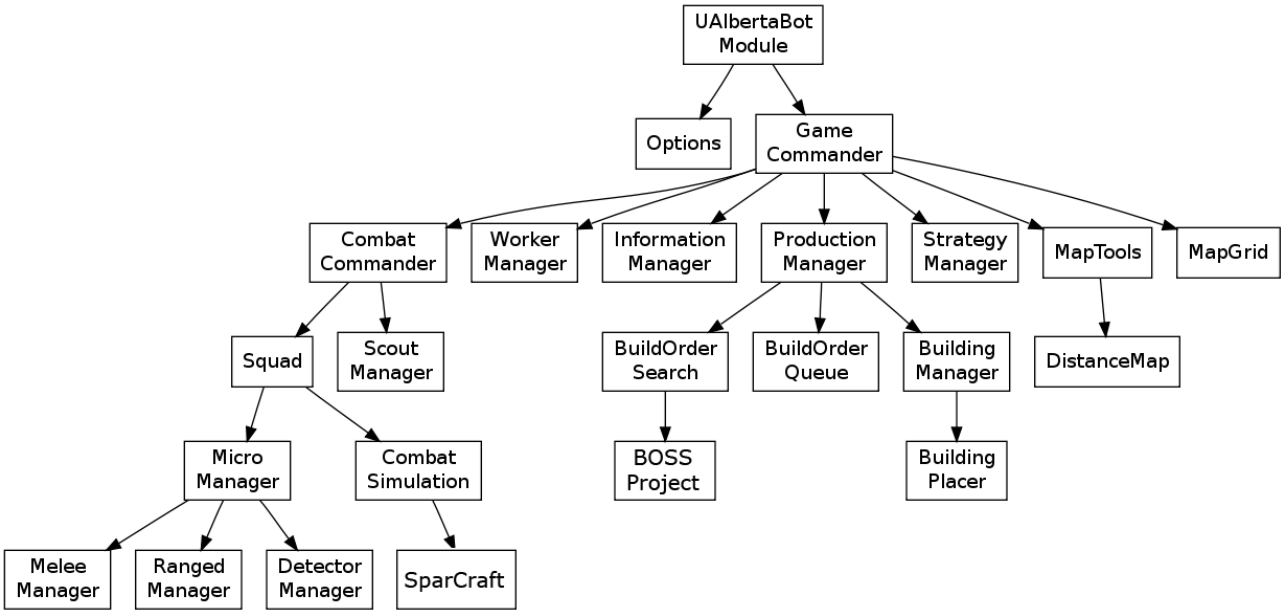

Fig. 2. A class diagram of UAlbertaBot, showing its modular design which facilitates AI task decomposition. We can easily substitute BOSS-IM or BOSS-UG within the UAlbertaBot framework above by replacing the "BOSS Project" module without modifying any other bot behavior, which allows us to isolate the performance difference of the build-order planning system on its own.

IV. Experiments

In order to evaluate the effectiveness of BOSS-IM, two experiments were performed in which it integrated into an existing StarCraft AI agent and played in real-time in a competitive setting, comparing its performance to the previous state-of-the-art method of BOSS-UG. The StarCraft AI agent used for these experiments is UAlbertaBot (UAB)[5], a competitive bot which has participated in all major StarCraft AI tournaments, has been the architectural basis for many current top-performing bots, and won the 2013 AIIDE StarCraft AI Competition. UAlbertaBot uses a modular hierarchical architecture specifically designed for easy modification of specific game play behaviors such as build-order planning, resource gathering, or combat, making it an ideal way to test the effectiveness of BOSS-IM to BOSS-UG.

A. UAlbertaBot Integration

In both experiments we used the 2017 AIIDE StarCraft AI Competition version of UAlbertaBot, with some minor changes to suit the experiments. During the 2017 competition, UAlbertaBot played Random race, meaning that each time a game started it would be randomly assigned one of the 3 StarCraft Races: Protoss, Terran, or Zerg. For each race, it had a number of strategies which were chosen based on several factors such as: opponent race, opponent name, and the map that was being played. In order to produce more reliable results for this paper and eliminate the hard-coded strategy selection mechanisms present in the bot for the 2017 competition, we removed the Random element of the race selection by limiting UAlbertaBot to only play the Protoss race, and limited it to only playing its strongest strategy: the Zealot rush. This 2017 version of the bot used the BOSS-UG build-order search system, with unit goal compositions decided by its StrategyManager module, consisting of a dozen or so human crafted hard-coded rules based on several factors such as: time of game, number of expansions, enemy army unit compositions, etc. New instances of the BOSS-UG search are triggered in the bot based on several possible in-game events such as: the current build-order queue being empty, one of its worker or building units being destroyed, or seeing our enemy build invisible units. The BOSS-UG search instance was given a time limit of 3 seconds, which was interleaved over multiple frames of computation during the game, with the resulting build-order being carried out by the bot as soon as the search episode terminated. UAlbertaBot implements the returned build-order by constructing each successive action as soon as possible in-game.

Combat in UAlbertaBot is handled in a generic manner that does not rely on any hard-coded attack timings or event triggers. As soon as any military unit is constructed, it is immediately sent to attack the opponent's base. Each second, UAlbertaBot uses its SparCraft combat simulation module [13] to simulate the outcome of its currently attacking units vs. scouted opponent units, and either retreats to regroup if it predicts it will lose the battle, or continues attacking if it believes it will win. This system eliminates the need for human-coded rules for engaging or disengaging the enemy, relying entirely on simulation, making it an ideal fit for use with BOSS-IM. Full details about all systems used in

[5]https://github.com/davechurchill/ualbertabot

UAlbertaBot can be found on GitHub[6].

These described modifications to UAlbertaBot will be referred to as UAB-BOSS-UG in the experimental results, and is the baseline version we will compare against.

To test BOSS-IM we created UAB-BOSS-IM from UAB-BOSS-UG by removing the BOSS-UG build-order search and replacing it entirely with BOSS-IM. Every other system of UAlbertaBot remained exactly the same, including combat, unit positioning, building placement, build-order search instance triggers, base expansion logic, scouting, resource gathering, etc. The only difference being that when an event triggers a new build-order search episode, BOSS-IM search was used in place of the original BOSS-UG search. Since BOSS-IM does not require specific unit goals like BOSS-UG, the unit goal decision logic system in UAlbertaBot went unused in UAB-BOSS-IM.

B. Search Parameters and Hardware

A search look-ahead limit of t=3000 in-game frames (125 human-play seconds) was used for UAB-BOSS-IM, with a search instance time-out of 3000ms (same as UAB-BOSS-UG), staggered across multiple frames in the same way as BOSS-UG to prevent any single frame from going over the competition-enforced rule of 55ms of computation per frame. UAB-BOSS-IM was also limited to producing armies consisting of Protoss Zealot and Dragoon units, which was the same constraint placed on UAB-BOSS-UG by the human-coded rules for selecting unit goals.

All experiments were performed using several identical computers running games in parallel, each running Windows 10 Professional edition with an Intel i7-7700k processor and 16gb of RAM. UAlbertaBot (including all search algorithms) runs in a single thread, so only one core of each processor was being used at any given time. Experiment 2 made extensive use of the AIIDE StarCraft AI Competition Tournament Manager Software[7].

C. Experiment 1: Retail StarCraft AI

The first experiment tested how UAB-BOSS-IM performed against the built-in retail StarCraft AI agents in comparison to UAB-BOSS-UG. For this experiment, each version of the bot played 50 games against each of the 3 races of the built-in StarCraft AI on each of the 10 maps used in the 2017 StarCraft AI Competition, for a total of 1500 games each. As the retail AI for StarCraft is known to be quite weak in comparison to modern competitive StarCraft AI agents, it is expected that both versions of the bot should win close to 100% of the games played. After 1500 games each, UAB-BOSS-UG won 1494 games, while UAB-BOSS-IM won 1488 games. Upon inspection of game replays, the extra few losses of UAB-BOSS-IM occurred to the retail AI's Protoss Dark Templar strategy, in which it makes invisible units to attack its opponent. Since UAB-BOSS-UG's unit goal-based search included human-coded rules to make invisibility detecting units upon seeing the opponent construct such invisible units, it did not lose to this strategy, unlike UAB-BOSS-IM which had no logic for making invisibility detection. So while this experiment showed that UAB-BOSS-IM was not significantly weaker than UAB-BOSS-UG against the retail AI, it did show one of its current possible weaknesses in that it did not tailor its build-orders to specific enemy strategies such as making invisible units. This problem can be rectified in the future by including invisibility detection as part of the heuristic evaluation that BOSS-IM attempts to maximize. However, such modifications may be seen as going counter to its initial purpose of avoiding the inclusion of such human-coded rules.

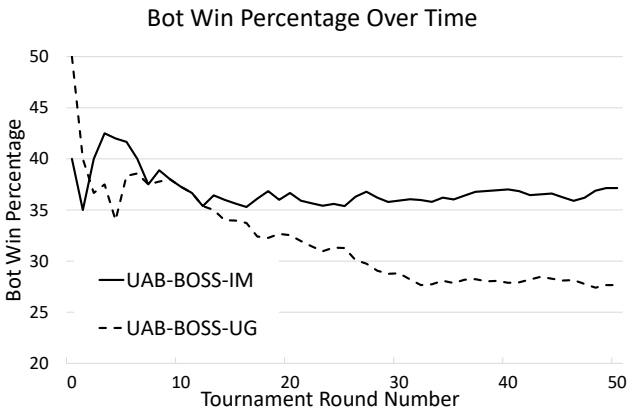

Fig. 3. Shown are results for the tournament run as part of Experiment 2, with 50 total rounds of round-robin play. All games for both versions were played against the same opponents - the top 10 ranked bots from the 2017 AIIDE StarCraft AI Competition.

D. Experiment 2: AIIDE StarCraft AI Competition

The second experiment tested both versions of UAlbertaBot in a real tournament setting, playing them against each of the top 10 ranked bots from the 2017 AIIDE StarCraft AI Competition for 50 rounds of round-robin play (5 rounds against each bot on each of the 10 tournament maps). These top 10 bots were chosen for this experiment for two main reasons. First, running all 28 bots would take an exceedingly long time, with the 2017 competition taking nearly 3 full weeks to run to completion. Second, there was a significant gap in win rate between the 10th and 11th place bots Steamhammer and AILien, so we felt it was a good round number for a cut-off point. The exact conditions and rules of the 2017 competition were used[8], using the exact same hardware that the competition was played on, with the only difference being that the 2017 version of UAlbertaBot was replaced with both UAB-BOSS-UG, and then with UAB-BOSS-IM. This experiment was performed in order to compare the performance of BOSS-IM versus BOSS-UG under strict real-time computational constraints against the current state-of-the-art in StarCraft AI agents. At the time of running this experiment, these were the 10 strongest publicly known StarCraft AI bots in the world, and since this

[6]https://github.com/davechurchill/ualbertabot/wiki

[7]https://github.com/davechurchill/StarcraftAITournamentManager

[8]http://www.starcraftaicompetition.com/rules.shtml

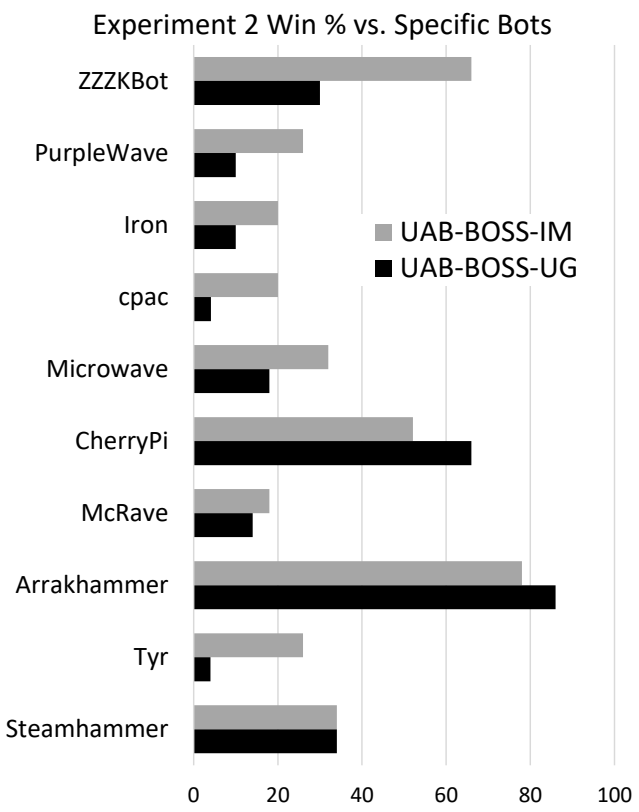

Fig. 4. Shown are win percentages for both versions of UAlbertaBot against each specific opponent in Experiment 2. Bots are listed in order of their placement in the 2017 AIIDE StarCraft AI Competition, from 1st place on top to 10th place on bottom. UAB-BOSS-IM performs better than UAB-BOSS-UG against these state-of-the-art opponents.

experiment only one bot has been developed which has been stronger, SAIDA[9] bot, which was created in 2018 by Samsung and won the 2018 AIIDE StarCraft AI Competition.

Results for experiment 2 can be seen in Figure 3, showing the win rate over time for each of UAB-BOSS-IM and UAB-BOSS-UG. Results show that that UAB-BOSS-UG obtains an overall win percentage of 27.66%, while UAB-BOSS-IM obtains a win percentage of 37.19%, for a total win percentage gain of 9.53%. This improvement is quite significant in a tournament setting. For example in the 2017 AIIDE competition, 4th place and 10th place were separated by less than 7% win percentage. Win rate decreased slowly over time for both versions due to several opponents implementing strategy learning over time in their bots, which tends to increase their win rate slightly vs. bots (like ours) that did not implement any learning. Note, however, that this decrease over time is less pronounced for UAB-BOSS-IM, perhaps due to the more robust build-orders being less exploitable by learning opponents.

Another interesting result can be observed in the more detailed results in Figure 4, which shows the win percentages for UAB-BOSS-IM and UAB-BOSS-UG against each other bot in the competition. This figure lists each enemy bot in the order they placed in the official 2017 AIIDE StarCraft AI Competition from 1st place on top to 10th place on bottom, which (intuitively) should correlate highly with overall bot strength. From this figure we can see that UAB-BOSS-IM outperforms UAB-BOSS-UG even more against higher ranked bots. For example, if we calculate the win percentages against only the top 5 ranked bots from the 2017 AIIDE StarCraft AI Competition, the win rate for UAB-BOSS-UG is 14.42%, while the win rate for UAB-BOSS-IM is 32.80%, for a gain of 18.38% win rate.

It should be noted that even though UAB-BOSS-UG contained human-coded unit goals, as well as specific strategic responses to given scenarios (such as building invisibility detectors when noticing invisible units), UAB-BOSS-IM still outperformed it without these particular expert knowledge / scripted responses. We suspect that the new BOSS-IM search system can react more dynamically to these stronger opponents than BOSS-UG which relies on its less flexible human-coded rules for its unit goals, which can fall especially easy prey to bots which learn over time.

V. Conclusions and Future Work

In this paper we have introduced two build-order optimization methods — BOSS-AV and BOSS-IM — that find action sequences maximizing the army value and the army value integral, respectively, for a given game time duration T. BOSS-AV is motivated by the observation that maximizing army values can be conceptually easier than minimizing makespans for achieving certain unit count goals because it may not be evident how to select such concrete goals depending on the current RTS game state in the first place. Action plans computed by BOSS-AV, however, may be prone to opponents attacking earlier than time T because such plans tend to delay producing army units. To mitigate this exploitability, while still using single-agent optimization, we proposed BOSS-IM which maximizes the integral of the army value curve up until time T. Our experimental results indicate that STARCRAFT bots using the newly proposed BOSS-IM search can perform better against current world-champion StarCraft AI agents than bots using the previous state-of-the-art unit-count goal-driven BOSS-UG method, when everything else remains equal. Not only does BOSS-IM perform significantly better, but it eliminates the previous problem of deciding on army compositions, by deciding which unit types to build automatically.

These promising initial results encourage us to explore multiple possible enhancements of BOSS-IM search in future work. For instance, refining the notion of army value to differentiate anti-air and anti-ground weaponry may be useful, as well as adjusting the army value definition depending on scouting information to bias search results towards countering enemy units. It may also be possible to combine search and learning techniques by automatically learning effective army value functions from human game replay files in order to completely remove the need for any human knowledge.

[9]https://github.com/TeamSAIDA/SAIDA

REFERENCES

[1] S. Ontanón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A survey of real-time strategy game AI research and competition in StarCraft," *TCIAIG*, 2013. [Online]. Available: http://webdocs.cs.ualberta.ca/~cdavid/pdf/starcraft_survey.pdf

[2] O. Vinyals *et al.*, "StarCraft II: A new challenge for reinforcement learning," 2017. [Online]. Available: https://arxiv.org/abs/1708.04782

[3] D. Churchill and M. Buro, "Build order optimization in StarCraft," in *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, 2011, pp. 14–19.

[4] M. Buro and A. Kovarsky, "Concurrent action selection with shared fluents," in *AAAI Vancouver, Canada*, 2007.

[5] A. Kovarsky and M. Buro, "A first look at build-order optimization in real-time strategy games," in *Proceedings of the GameOn Conference*, 2006, pp. 18–22.

[6] H. Chan, A. Fern, S. Ray, N. Wilson, and C. Ventura, "Online planning for resource production in real-time strategy games," in *Proceedings of the International Conference on Automated Planning and Scheduling, Providence, Rhode Island*, 2007.

[7] ——, "Extending online planning for resource production in real-time strategy games with search," *ICAPS Workshop on Planning in Games*, 2007.

[8] A. A. Branquinho and C. R. Lopes, "Planning for resource production in real-time strategy games based on partial order planning, search and learning," in *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on*. IEEE, 2010, pp. 4205–4211.

[9] J. Blackford and G. Lamont, "The real-time strategy game multi-objective build order problem," in *AIIDE*, 2014.

[10] P. Garćıa-Sánchez, A. Tonda, A. M. Mora, G. Squillero, and J. J. Merelo, "Towards automatic StarCraft strategy generation using genetic programming," in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, Aug 2015, pp. 284–291.

[11] N. Justesen and S. Risi, "Continual online evolutionary planning for in-game build order adaptation in StarCraft," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '17. New York, NY, USA: ACM, 2017, pp. 187–194. [Online]. Available: http://doi.acm.org/10.1145/3071178.3071210

[12] D. Churchill, "UAlbertaBot," https://github.com/davechurchill/ualbertabot/, 2016. [Online]. Available: https://github.com/davechurchill/ualbertabot/

[13] D. Churchill and M. Buro, "Portfolio greedy search and simulation for large-scale combat in StarCraft," in *IEEE Conference on Computational Intelligence in Games (CIG)*. IEEE, 2013, pp. 1–8.