Hierarchical Adversarial Search Applied to Real-Time Strategy Games

Marius Stanescu and **Nicolas A. Barriga** and **Michael Buro**
Deptartment of Computing Science, University of Alberta
Edmonton, Canada, T6G 2E8
Email: {astanesc, barriga, mburo}@ualberta.ca

Abstract

Real-Time Strategy (RTS) video games have proven to be a very challenging application area for artificial intelligence research. Existing AI solutions are limited by vast state and action spaces and real-time constraints. Most implementations efficiently tackle various tactical or strategic sub-problems, but there is no single algorithm fast enough to be successfully applied to big problem sets (such as a complete instance of the StarCraft RTS game). This paper presents a hierarchical adversarial search framework which more closely models the human way of thinking — much like the chain of command employed by the military. Each level implements a different abstraction — from deciding how to win the game at the top of the hierarchy to individual unit orders at the bottom. We apply a 3-layer version of our model to SparCraft — a StarCraft combat simulator — and show that it outperforms state of the art algorithms such as Alpha-Beta, UCT, and Portfolio Search in large combat scenarios featuring multiple bases and up to 72 mobile units per player under real-time constraints of 40 ms per search episode.

Introduction

Real-Time Strategy (RTS) games are a genre of video games in which players gather resources, build structures from which different types of troops can be trained or upgraded, recruit armies, and command them in battle against opponent armies. RTS games are an interesting domain for Artificial Intelligence (AI) research because they represent well-defined complex adversarial decision problems and can be divided into many interesting and computationally hard subproblems (Buro 2004).

The best AI systems for RTS games still perform poorly against good human players (Buro and Churchill 2012). Hence, the research community is focusing on developing RTS agents to compete against other RTS agents to improve the state-of-the-art. For the purpose of experimentation, the RTS game StarCraft: Brood War (Blizzard Entertainment 1998) has become popular because it is considered well balanced, has a large online community of players, and has an open-source interface — BWAPI, (Heinermann 2013) — which allows researchers to write programs to play the full game. Several StarCraft AI competitions are organized every year (Buro and Churchill 2012). Such contests have sparked increased interest in RTS game AI research and many promising agent frameworks and algorithms have already emerged. However, no unified search approach has yet been developed for a full RTS game such as StarCraft, although the research community is starting to tackle the problem of global search in smaller scale RTS games (Chung, Buro, and Schaeffer 2005; Sailer, Buro, and Lanctot 2007; Ontañón 2013). Existing StarCraft agents rely on a combination of search and machine learning for specific subproblems (build order (Churchill and Buro 2011), combat (Churchill and Buro 2013), strategy selection (Synnaeve 2012)) and hard-coded expert behaviour.

Motivation and Objectives

Even though the structure of most RTS AI systems is complex and comprised of many modules for unit control and strategy selection (Wintermute, Joseph Xu, and Laird 2007; Churchill and Buro 2012; Synnaeve and Bessiere 2012), none comes close to human abstraction, planning, and reasoning abilities. These independent modules implement different AI mechanisms which often interact with each other in a limited fashion.

We propose a unified perspective by defining a *multi-level abstraction framework* which more closely models the human way of thinking — much like the chain of command employed by the military. In real life a top military commander does not concern himself with the movements of individual soldiers, and it is not efficient for an RTS AI to do that, either. A hierarchical structure can save considerable computing effort by virtue of hierarchical task decomposition. The proposed AI system partitions its forces and resources to a number of entities (we may call commanders) — each with its own mission to accomplish. Moreover, each commander could further delegate tasks in a similar fashion to sub-commanders, groups of units or even individual units. More similar to the human abstraction mechanism, this flexible approach has a great potential of improving AI strength.

One way of implementing such a layered framework is to have each layer playing the game at its own abstraction level. This means we would have to come up with both the abstractions and a new game representation for each level. Both are difficult to design and it is hard to prove that they actually model the real game properly. Another way, which we introduce in this paper, is to partition the game state and to assign objectives to individual partition elements, such that at the lowest level we try to accomplish specific goals by searching smaller state spaces, and at higher levels we search over possible partitions and objective assignments. Our approach comes with an additional benefit: the main issue that arises when approximating optimal actions in complex multi-agent decision domains is the combinatorial explosion of state and action spaces, which renders classic exhaustive search infeasible. For instance, in StarCraft, players can control up to 200 mobile units which are located on maps comprised of up to 256×256 tiles, possibly leading to more than $10^{1,926}$ states and 10^{120} available actions (Ontañón et al. 2013). The hierarchical framework we propose has the advantage of reducing the search complexity by considering only a meaningful subset of the possible interactions between game objects. While optimal game play may not be achieved by this kind of abstraction, search-based game playing algorithms can definitely be more efficient if they take into account that *in a limited time frame each agent interacts with only a small number of other agents* (Lisŷ et al. 2010).

Background and Related Work

In implementing a hierarchical framework we extend state of the art RTS unit combat AI systems that use Alpha-Beta, UCT, or Portfolio Search (Churchill and Buro 2012; Churchill, Saffidine, and Buro 2012; Churchill and Buro 2013) and focus only on one abstraction level, namely planning combat actions at the unit level. By contrast, our proposed search algorithm considers multiple levels of abstractions. For example, first level entities try to partition the forces into several second level entities – each with its own objective.

Similar approaches have been proposed by (Mock 2002) and (Rogers and Skabar 2014). The latter implements a framework to bridge the gap between strategy and individual unit control. It conceptualizes RTS group level micromanagement as a multi-agent task allocation problem which can in principle be integrated into any layered RTS AI structure. However, while these papers only deal with the bottom level, we target several abstraction levels at the same time. Our goal is to interleave the optimization of direct unit control with strategic reasoning. Next, we look at competing algorithms designed for other problem domains whose large complexity render complete search impractical.

Multi-Agent Planning

A similar problem has been considered in (Lisŷ et al. 2010). The authors substitute global search considering all agents with multiple searches over agent subsets, and then combine the results to form a global solution. The resulting method is called Agent Subset Adversarial Search (ASAS). It was proved to run in polynomial time in the number of agents as long as the size of the subsets is limited. ASAS does not require any domain knowledge, comparing favourably to other procedural approaches such as *hierarchical task networks* (HTNs), and greatly improves the search efficiency at the expense of a small decrease of solution quality. It works best in domains in which a relatively small number of agents can interact at any given time. Because we aim our framework specifically towards RTS games, avoiding using domain knowledge would be wasteful. Hence, we introduce a procedural component in the algorithm, in the form of objectives that each subset tries to accomplish. This will allow our algorithm to make abstract decisions at a strategic level, which should further improve the search efficiency compared to the ASAS method.

Goal Driven Techniques

Heuristic search algorithms such as Alpha-Beta search and A* choose actions by looking ahead, heuristically evaluating states, and propagating results. By contrast, HTNs implement a goal-driven method that decomposes goals into a series of sub-goals and tries to achieve these. For choosing a move, goals are expanded into sub-goals at a lower level of abstraction and eventually into concrete actions in the environment. In (Smith, Nau, and Throop 1998), the authors successfully deploy HTNs to play Contract Bridge. Using only a small number of operators proves sufficient for describing relevant plays (finessing, ruffing, cross-ruffing, etc.). HTNs have also been successfully applied to Go (Willmott et al. 2001). The advantage is that Go knowledge (e.g., in books) is usually expressed in a form appropriate for encoding goal decompositions by using a rich vocabulary for expressing reasons for making moves. However, HTNs generally require significant effort for encoding strategies as goal decompositions, and in some domains such as RTS games this task can be very difficult. For example, (Menif, Guettier, and Cazenave 2013) uses HTNs to build a hierarchical architecture for Infantry Serious Gaming. Only a small set of very simple actions are supported, such as monitoring and patrolling tasks, and we can already see that the planning domain would become quite convoluted with the addition of more complex behaviour.

Goal Based Game Tree Search

Goal based game tree search (GB-GTS) (Lisŷ et al. 2009) is an algorithm specifically designed for tackling the scalability issues of game tree search. It uses procedural knowledge about how individual players tend to achieve their goals in the domain, and employs this information to limit the search to the part of the game tree that is consistent with the players' goals. However, there are some limitations: goals are only abandoned if they are finished, policy which doesn't allow replacing the current plan with a potentially better one. Also, goals are assigned on unit-basis level and more units following the same goal require more computational effort than if they were grouped together under the same objective. This is more similar to a bottom-up approach, the goals in GB-GTS serving as building blocks for more complex behaviour. It contrasts with HTN-based approaches where the whole strategies are encoded using decompositions from the highest levels of abstraction to the lower ones.

Hierarchical Adversarial Search

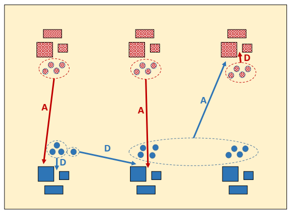
The basic idea of our algorithm we call *Hierarchical Adversarial Search* is to decompose the decision making process into three layers: the first layer chooses a set of **objectives** that need to be accomplished to win the game (such as build an expansion, create an army, defend one of our bases or destroy an enemy base), the second layer generates action sequences to accomplish these objectives, and the third layer's task is to *execute* a plan. Here, executing means generating "real world" moves for individual units based on a given plan. Games can be played by applying these moves in succession. We can also use them to roll the world forward during look-ahead search which is crucial to evaluating our plans. An objective paired with a group of units to accomplish it is called a **task**. A pair of tasks (one for each player) constitutes a **partial game state**. We consider partial game states independent of each other. Hence, we ignore any interactions between units assigned to different partial game states. This assumption is the reason for the computational efficiency of the *hierarchical search* algorithm, because searching one state containing N units is much more expensive than searching M states with N/M units each. Lastly, a **plan** consists of disjoint partial game states, which combined represent the whole game state.

Application Domain and Motivating Example

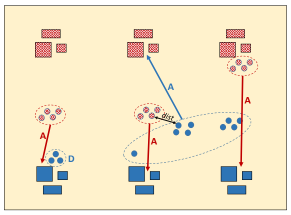
Our implementation is based on SparCraft (Churchill 2013), which is an abstract combat simulator for StarCraft that is used in the current *UAlbertaBot* StarCraft AI to estimate the chance of winning fights before engaging the enemy. Since simulation must be faster than real-time, SparCraft abstracts away game details such as building new structures, gathering resources or training units. Because of this limitation we consider simpler base-defense scenarios in this paper instead of playing full RTS games, and we focus on the lower two levels of the search: constructing and executing plans, while fixing the objective to destroying or defending bases. With extra implementation effort these scenarios could be set up in StarCraft as well, likely leading to similar results.

Fig. 1(a) shows a sample plan with three partial game states. The blue (solid, bottom) player chose three units to defend the left base (task 1), one unit to defend the middle base (task 2) and eight units to attack the enemy base to the right (task 3). Analogously, the red (striped, top) player chose two attacking tasks (tasks 1 and 2) and one defending task (task 3), assigning four units to each. In this example the first partial game state contains the first task of each player, the second partial game state the second task, and the last partial game state the third task. After a while the red units of the middle (second) partial game state get within attack range of the blue units assigned to the third partial game state and re-planning is triggered. One possible resulting scenario is shown in Fig. 1(b):

- the blue player still chooses to defend with three units against four on the left side in the first partial game state
- he switches his nine units to attack the middle enemy base, and is opposed by four enemy units in the second partial game state
- the blue player chooses to do nothing about the right base, so the third partial game state will only contain four enemy units attacking (blue player has an idle objective).



(a) Initial plan



(b) Re-planning

Figure 1: An example of two consecutive plans

Bottom Layer: Plan Evaluation and Execution

The bottom layer serves two purposes: in the hierarchical search phase (1) it finds lowest-level actions sequences and rolls the world forward executing them, and in the plan execution phase (2) it will generate moves in the actual game. In both phases we first create a new sub-game that contains only the units in the specific partial game state. Then, in phase (1) we use fast players with scripted behaviour specialized for each objective to play the game, as the numbers of playouts will be big. During (1) we are not concerned with returning moves, but only with evaluating the plans generated by the middle layer. During phase (2) we use either Alpha-Beta or Portfolio search (Churchill and Buro 2013) to generate moves during each game frame, as shown in Fig. 2. The choice between Alpha-Beta or Portfolio search depends on the number of units in the partial game state. Portfolio search is currently the strongest algorithm for states with large numbers of units, while Alpha-Beta search is the strongest for small unit counts. These processes are repeated in both phases until either a predefined time period has passed, or any task in the partial game state has been completed or is impossible to accomplish.

Middle Layer: Plan Selection

The middle layer search, shown in Fig. 3, explores possible plans and selects the best one for execution. The best plan is found by performing a minimax search (described in Algorithm 1), where each move consists of selecting a number of tasks that will accomplish some of the original objectives. As the number of moves (i.e., task combinations) can be big, we control the branching factor with a parameter X: we only consider X heuristically best moves for each player. With good move ordering we expect the impact in plan quality to be small. We sort the moves according to the average completion probability of the tasks. For each task, this probability is computed using the LTD2 ("Life-Time Damage 2") score (the sum of the square root of hit points remaining of each unit times its average attack value per frame (Churchill and Buro 2013)) of the player units assigned to that task compared to the LTD2 score of the enemy

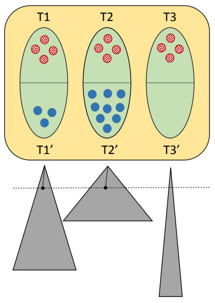


Figure 2: Alpha-Beta search on partial game states.

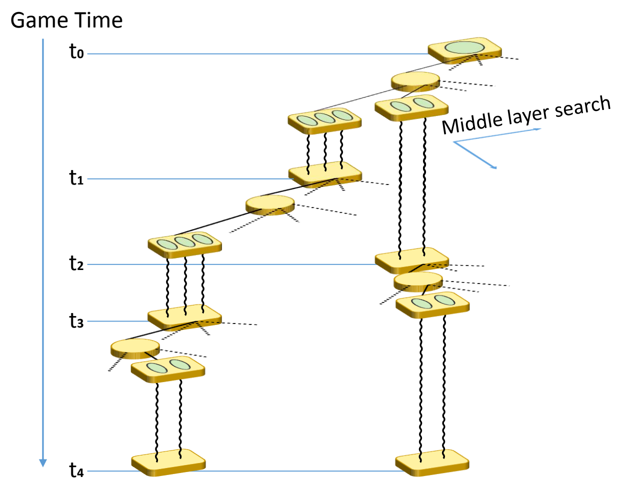


Figure 3: Middle layer search: the middle layer performs minimax search on abstract moves (rectangles depict max nodes, circles depict min nodes). The bottom layer executes scripted playouts to advance the game. Given more time, scripts can be replaced by search.

units closest to the task. We assume that enemy units influence only the task closest to them.

One of the players makes a move by generating some tasks (line 7) and applying them successively (line 9), the other player independently makes his move (line 14), and the opposing tasks are paired and combined into a plan consisting of several partial game states (line 15). We call a plan **consistent** if the partial game states don't interfere with each other, i.e., units from different partial game states can't attack each other (Fig. 1(b)). If a plan is not *consistent* we skip it and generate the next plan to avoid returning a plan that would trigger an immediate re-plan. Otherwise, we do a playout using the scripted players for each partial game state and roll the world forward, until either one of the tasks is completed or impossible to accomplish or we reach a predefined game time (line 17). Fig. 3 shows two playouts at the same search depth, finishing at different times t_1 and t_2.

At this point we combine all partial game states into a full game state (line 18) and recursively continue the search (line 19). The maximum search depth is specified in game time, not search plies. Hence, some branches in which the playouts are cut short because of impossible or completed objectives might trigger more middle layer searches than others, like the leftmost branch in Fig. 3.

Finally, for leaf evaluation (line 5), we check the completion level of the top layer objectives (for example, how many enemy buildings we destroyed if we had a destroy base objective). Node evaluations can only be done at even depths, as we need both players to select a set of tasks before splitting the game into several partial game states.

After the middle layer completes the search, we will have a plan to execute: our moves from the principal variation of the minimax tree (line 22). At every game frame we update the partial game states in the plan with the unit data from the actual game, and we check if the plan is still valid. A plan might become invalid because one of its objectives is completed or impossible, or because units from two different partial game states are within attack range of each other. If the plan is invalid, re-planning is triggered. Otherwise it will be executed by the bottom layer using Alpha-Beta or Portfolio search for each partial game state, and a re-plan will be triggered when it is completed. To avoid re-planning very often, we do not proactively check if the enemy follows the principal variaton but follow the more lazy approach of re-planning if his actions interfere with our assumption of what his tasks are.

We did not implement the top layer and we simply consider destroy and defend all bases as objectives to be accomplished by the middle layer, but we discuss more alternatives in the last section, as future work.

Algorithm 1 Hierarchical Adversarial Search (2 layers)

```
 1: procedure PlanSearch(Depth $d$, State $s$, Task $oppTask$, Player $p$)
 2:     $best \leftarrow -\infty$;
 3:     if Empty($oppTask$) then            ▷ First player gen. tasks
 4:         if EndSearch() then
 5:             return Evaluate($s$)          ▷ w.r.t. objectives
 6:         else
 7:             $tasks \leftarrow$ GenTasks($s$, $p$)
 8:             for Task $t$ in $tasks$ do
 9:                 $val \leftarrow$ - PlanSearch($d$+1,$s$,$t$,opp($p$))
10:                 if $val > best$ then
11:                     $best \leftarrow val$;
12:             return $best$
13:     else                                 ▷ Second player gen. tasks
14:         $tasks \leftarrow$ GenTasks($s$,$p$)
15:         $plans \leftarrow$ GenPlans($s$,$enemyTask$,$tasks$,$p$)
16:         for Plan $plan$ in $plans$ do
17:             Playout($plan$)
18:             Merge($s$)          ▷ Merge partial game states
19:             $val \leftarrow$ - PlanSearch($d+1$,$s$,<>,opp($p$))
20:             if $value > best$ then
21:                 $best \leftarrow val$;
22:                 UpdatePrincipalVariation()
23:         return $best$
```

Implementation Details

After receiving a number of objectives that should be completed from the top layer, the middle layer search has to choose a subset of objectives to be accomplished next. To generate a move we need to create a task for each objective in this subset: each task consists of some units assigned to accomplish a particular objective. Such assignments can be generated in various ways, for instance by spatial clustering. Our approach is to use a greedy bidding process similar to the one described in (Rogers and Skabar 2014) that assigns units to tasks dependent on proximity and task success evaluations. The first step is to construct a matrix with bids for each of our units on each possible objective. Bids take into account the distance to the target and an estimate of the damage the unit will inflict and receive from enemy units close to the target. Let O be the total number of objectives, and $N \in \{1, \cdots, O\}$ a smaller number of objectives we want to carry out in this plan. We consider all permutations of N objectives out of

O possible objectives (for a total of $O!/(O-N)!$). For example, if $O = 4$ and $N = 2$, we consider combinations $(1,2),(2,1),(1,3),(3,1),\ldots,(3,4),(4,3)$. The difference between (x,y) and (y,x) is objective priority: we assign units to accomplish objective x or y first.

For each of these combinations and each objective, we iterate assigning the unit with the highest bid to it until we think that the objective can be accomplished with a certain probability (we use 0.8 in the experiments) and then continue to the next objective. This probability is estimated using a sigmoid function and the LTD2 scores of the Hierarchical Adversarial Search player's units assigned to complete that objective versus the LTD2 score of the enemy units closest to the objective. The moves will be sorted according to the average completion probability of all the O tasks (the assigned N as well as the $O-N$ we ignored). Consider the case in which one of the left-out objectives is a defend base objective. Consequently, the middle layer did not assign any units for this particular objective. In one hypothetical case there are no enemy units close to that particular base, so we will likely be able to accomplish it nonetheless. In another case, that particular base is under attack, so there will be a very low probability of saving it as there are no defender units assigned for this task. If the middle layer sorts the moves only using the probabilities for the N tasks to which it has assigned units, it won't be able to differentiate between the previous two cases and will treat them as equally desirable. Hence, we need to consider the completion probability of all tasks in the move score, even those to which the middle layer did not assign units.

After both players choose a set of tasks, we need to match these tasks to construct partial game states. Matching is done based on distances between different tasks (currently defined as the minimum distance between two units assigned to those tasks). Because partial game states are considered independent, we assign tasks that are close to each other to the same partial game state. For example, "destroy base A" for player 1 can be matched with "defend base A" for player 2 if the attacking units are close to base A. If the units ordered to destroy base A for player 1 are close to the units that have to destroy base B for player 2, then we have to match them because the involved units will probably get into a fight in the near future. However, we now also need to add the units from "defend base A" and "defend base B" to the same partial game state (if they exist) to avoid conflicts. After matching, if there are any unmatched tasks, we add dummy idle tasks for the opponent.

While performing minimax search, some of the plans generated by task matching might still be illegal, i.e., units in one partial game state are too close to enemy units from another partial game state. In that case we skip the faulty plan and generate the next one.

In our current implementation, when re-planning is needed, we take the time needed to search for a new plan. In an actual bot playing in a tournament, we would have a current plan that is being executed, and a background thread searching for a new plan. Whenever the current plan is deemed illegal, we would switch to the latest plan found. If that plan is illegal, or if the search for a new plan hasn't finished, we would fall back to another algorithm, such as Alpha-Beta, UCT, or Portfolio Search.

For plan execution, the time available at each frame is divided between different Alpha-Beta searches for each partial game state. The allotted time is proportional to the number of units in each, which is highly correlated to the branching factor of the search. As the searches usually require exponential effort rather than linear, more complex time division rules should be investigated in the future.

It is worth noting that both plan execution and evaluation are easily parallelizable, because each partial game state is independent of each other. Plan search can be parallelized with any minimax parallelization technique such as ABDADA (Weill 1996). Because at each node in the plan search we only explore a fixed number of children and ignore the others, the search might also be amenable to random sampling search methods, such as Monte Carlo tree search.

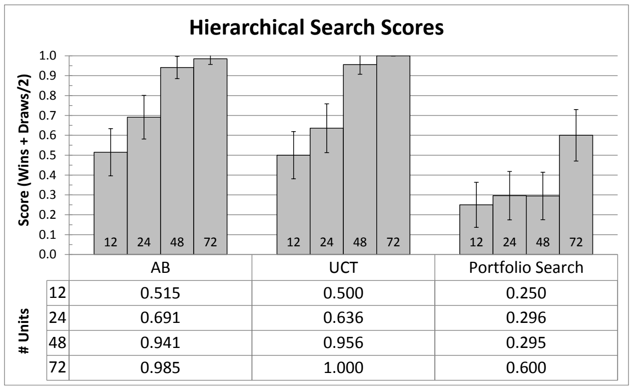
Empirical Evaluation

Two sets of experiments were carried out to evaluate the performance of the new Hierarchical Adversarial Search algorithm. In the first set we let the Hierarchical Adversarial Search agent which uses Portfolio Search for plan execution play against the Alpha-Beta, UCT, and Portfolio Search agents presented in (Churchill and Buro 2013). All agents have the same amount of time available to generate a move. In the second set of experiments Hierarchical Adversarial Search using Alpha-Beta Search for plan execution plays against an Alpha-Beta Search agent whose allocated time varies.

Similar to (Churchill and Buro 2013), each experiment consists of a series of combat scenarios in which each player controls an identical group of StarCraft units and three bases. Each base consists of four (on half of the maps) or five (on the other half) structures: some are buildings that cannot attack but can be attacked (which are the objectives in our game), and some are static defences that can attack (e.g., photon cannons in StarCraft). Both players play the Protoss race and the initial unit/building placement is symmetric to ensure fairness.

The battlefields are empty StarCraft maps of medium size (128×72 tiles) without obstacles or plateaus because none of the tested algorithms has access to a pathfinding module. Experiments are done on four different maps, with three bases totalling 12 to 15 buildings for each player. The set-up is similar to Fig. 1(a), though on some maps the position of the middle base for the two players is swapped. To investigate the performance of the different algorithms we vary the number of units — 12, 24, 48, or 72 per player, equally split between the three bases.

For the first experiment we generate 60 battles for each unit configuration. For the second experiment we use only one map with 72 units per player, and play 10 games for each Alpha-Beta time limit. Each algorithm was given a 40 ms time limit per search episode to return a move. This time limit was chosen to comply to real-time performance restrictions in StarCraft, which runs at 24 frames per second (42 ms per frame). The versions of Alpha-Beta, UCT, and Portfolio Search (depending on parameters such as maximum limit of children per search node, transposition table size, exploration constant or evaluation) are identical to those used in



# Units	AB	UCT	Portfolio Search
12	0.515	0.500	0.250
24	0.691	0.636	0.296
48	0.941	0.956	0.295
72	0.985	1.000	0.600

Figure 4: Results of Hierarchical Adversarial Search against Alpha-Beta, UCT and Portfolio Search for combat scenarios with n vs. n units ($n = 12, 24, 48$ and 72). 60 scenarios were played allowing 40 ms for each move. 95% confidence intervals are shown for each experiment.

(Churchill and Buro 2013). It is important to note that the Alpha-Beta Search agent uses a fall-back strategy: in cases when it can't complete even a 1-ply search it will execute scripted moves. In our experiments we impose a time limit of 3000 frames for a game to finish, at which time we decide the winner using the LTD2 score. All experiments were performed on an Intel(R) Core2 Duo P8400 CPU 2.26GHz with 4 GB RAM running Fedora 20, using single-thread implementations. The software was implemented in C++ and compiled with g++ 4.8.2 using -O3 optimization.

Results

In the first set of experiments, we compare Hierarchical Adversarial Search using Portfolio Search for partial game states against Alpha-Beta, UCT and Portfolio Search. Fig. 4 shows the win rate of the hierarchical player against the benchmark agents, grouped by number of units controlled by each player. Hierarchical Adversarial Search has similar performances against Alpha-Beta and UCT algorithms. It is roughly equal in strength when playing games with the smallest number of units, but its performance grows when increasing the number of units. From a tie at 12 vs. 12 units, our algorithm exceeds 90% winning ratio for 48 vs. 48 units and reaches 98% for the largest scenario. As expected, performing tree searches on the smaller game states is more efficient than searching the full game state, as the number of units grows. Against Portfolio Search, our method does worse on the smaller scenarios. Hierarchical Adversarial Search overcomes Portfolio Search only in the 72 vs. 72 games, which are too complex for Portfolio to execute the full search in the allocated 40 ms.

In the second set of experiments, we compare Hierarchical Adversarial Search using Alpha-Beta Search for partial game states against Alpha-Beta Search. Hierarchical Adversarial Search computing time is limited to 40 ms and Alpha-Beta Search agent's time was varied using 20 ms steps to find at which point the advantage gained by partitioning the game state can be matched by more computation time. Fig. 5 shows the win ratio of Alpha-Beta in scenarios with 72 units for each player. Hierarchical Adversarial Search still wins

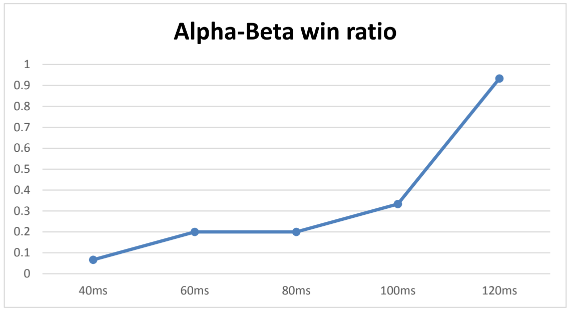


Figure 5: Alpha-Beta Search strength variation when increasing computation time, playing against Hierarchical Adversarial Search using 40 ms per move and Alpha-Beta Search for plan execution. 10 experiments using 72 units on each side were performed for each configuration.

when Alpha-Beta Search has double the search time (80 ms), but loses when tripling Alpha-Beta's time. These results suggest that our Hierarchical Abstract Search implementation can be improved by complete enumeration of tasks in the middle layer and/or conducting searches in the bottom layer once unit counts drop below a certain threshold.

Conclusions and Future Work

In this paper we have presented a framework that attempts to employ search methods for different abstraction levels in a real-time strategy game. In large scale combat experiments using SparCraft, our Hierarchical Adversarial Search algorithm outperforms adaptations of Alpha-Beta and UCT Search for games with simultaneous and durative moves, as well as state-of-the-art Portfolio search. The major improvement over Portfolio Search is that Hierarchical Adversarial Search can potentially encompass most areas of RTS game playing, such as building expansions and training armies, as well as combat.

As for future work, the first area of enhancements we envision for our work is adding pathfinding to Alpha-Beta, UCT, and Portfolio Search to test Hierarchical Adversarial Search on real StarCraft maps. We also plan to improve the task bidding process by using a machine learning approach similar to the one described in (Stanescu et al. 2013) for predicting battle outcomes. It would still be fast enough but possibly much more accurate than using LTD2 scores, and could also integrate a form of opponent modelling into our algorithm. Finally, we currently only use two of the three layers and generate fixed objectives for the middle layer (such as destroy opponent bases) instead of using another search algorithm at the top layer. If we extend SparCraft to model StarCraft's economy, allowing the agents to gather resources, construct buildings, and train new units, the top layer decisions become more complex and we may have to increase the number of abstraction layers to find plans that strike a balance between strengthening the economy, building an army, and finally engaging in combat.

References

Blizzard Entertainment. 1998. StarCraft: Brood War. http://us.blizzard.com/en-us/games/sc/.

Buro, M., and Churchill, D. 2012. Real-time strategy game competitions. *AI Magazine* 33(3):106–108.

Buro, M. 2004. Call for AI research in RTS games. In *Proceedings of the AAAI-04 Workshop on Challenges in Game AI*, 139–142.

Chung, M.; Buro, M.; and Schaeffer, J. 2005. Monte Carlo planning in RTS games. In *IEEE Symposium on Computational Intelligence and Games (CIG)*.

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

Churchill, D., and Buro, M. 2012. Incorporating search algorithms into RTS game agents. In *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*.

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

Churchill, D.; Saffidine, A.; and Buro, M. 2012. Fast heuristic search for RTS game combat scenarios. In *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*.

Churchill, D. 2013. SparCraft: open source StarCraft combat simulation. http://code.google.com/p/sparcraft/.

Heinermann, A. 2013. Brood War API. http://code.google.com/p/bwapi/.

Lisỳ, V.; Bošanskỳ, B.; Jakob, M.; and Pechoucek, M. 2009. Adversarial search with procedural knowledge heuristic. In *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*.

Lisỳ, V.; Bošanskỳ, B.; Vaculín, R.; and Pechoucek, M. 2010. Agent subset adversarial search for complex non-cooperative domains. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, 211–218. IEEE.

Menif, A.; Guettier, C.; and Cazenave, T. 2013. Planning and execution control architecture for infantry serious gaming. In *Planning in Games Workshop*, 31.

Mock, K. J. 2002. Hierarchical heuristic search techniques for empire-based games. In *Proceedings of the International Conference on Artificial Intelligence (IC-AI)*, 643–648.

Ontañón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A survey of real-time strategy game AI research and competition in StarCraft. *TCIAIG*.

Ontañón, S. 2013. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *AIIDE*.

Rogers, K., and Skabar, A. 2014. A micromanagement task allocation system for real-time strategy games. *Computational Intelligence and AI in Games, IEEE Transactions on*.

Sailer, F.; Buro, M.; and Lanctot, M. 2007. Adversarial planning through strategy simulation. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, 80–87. IEEE.

Smith, S. J.; Nau, D.; and Throop, T. 1998. Computer Bridge: A big win for AI planning. *AI magazine* 19(2):93.

Stanescu, M.; Hernandez, S. P.; Erickson, G.; Greiner, R.; and Buro, M. 2013. Predicting army combat outcomes in StarCraft. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Synnaeve, G., and Bessiere, P. 2012. Special tactics: a Bayesian approach to tactical decision-making. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, 409–416.

Synnaeve, G. 2012. *Bayesian programming and learning for multi-player video games*. Ph.D. Dissertation, Université de Grenoble.

Weill, J.-C. 1996. The ABDADA distributed minimax search algorithm. In *Proceedings of the 1996 ACM 24th annual conference on Computer science*, 131–138. ACM.

Willmott, S.; Richardson, J.; Bundy, A.; and Levine, J. 2001. Applying adversarial planning techniques to Go. *Theoretical Computer Science* 252(1):45–82.

Wintermute, S.; Joseph Xu, J. Z.; and Laird, J. E. 2007. SORTS: A human-level approach to real-time strategy AI. In *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, 55–60.