Introducing Hierarchical Adversarial Search, a Scalable Search Procedure for Real-Time Strategy Games

Marius Stanescu[1] and **Nicolas A. Barriga**[1] and **Michael Buro**[1]

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 full RTS games. This paper introduces a hierarchical adversarial search framework which implements a different abstraction at each level — from deciding how to win the game at the top of the hierarchy to individual unit orders at the bottom.

1 Introduction

Real-Time Strategy (RTS) games are a genre of video games in which players gather resources, build structures from which different types of units can be constructed or upgraded 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 sub-problems [2].

The best AI systems for RTS games still perform poorly against good human players. Hence, the research community is focusing on developing RTS agents to compete against other RTS agents to improve the state-of-the-art. Such competition has 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, although the research community is starting to tackle the problem of global search in smaller scale RTS games [3, 9, 7]. Existing agents for full RTS games rely on a combination of search and machine learning for specific sub-problems (build order [5], combat [6], strategy selection [10]) and hard-coded expert behaviour.

Even though the structure of most RTS AI systems is complex and comprised of many modules for unit control and strategy selection [12, 11], 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* that can save considerable computing effort by virtue of hierarchical task decomposition. The proposed AI system partitions its forces and resources into groups, each with its own mission to accomplish. Moreover, each group could be further partitioned in a similar fashion to sub-groups, even to the level of individual units.

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 the popular RTS game StarCraft [1], 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 [8]. 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 short time span each agent interacts with only a subset of all other agents.

2 Hierarchical Adversarial Search

In implementing a hierarchical framework we generalize existing search algorithms. Currently, state-of-the-art RTS unit combat AI systems use Alpha-Beta, UCT, or Portfolio Search [6], focusing on only one abstraction level, namely planning combat actions at the unit level. By contrast, our 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.

The basic idea of the 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 expansions, create an armies, defend one bases or destroy enemy bases), 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 achieve 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 computa-

[1] Dept. of Computing Science, University of Alberta, Edmonton, Canada, email: {astanesc, barriga, mburo}@ualberta.ca

tional efficiency of the *hierarchical search* algorithm: 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.

In this initial work we focus on the two lower layers, not implementing the top layer and simply considering destroying and defending all bases as objectives to be accomplished by the middle layer.

2.1 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 [6] to generate moves during each game frame. 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.

2.2 Middle Layer: Plan Selection

The middle layer search explores possible plans and selects the best one for execution. The best plan is found by performing a minimax search, 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 of each unit times its average attack value per frame [6]) of the player units assigned to that task compared to the LTD2 score of the enemy 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 and applying them successively, the other player independently makes his move, and the opposing tasks are paired and combined into a plan consisting of several partial game states. 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. 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. At this point we combine all partial game states into a full game state and recursively continue the search.

Finally, for leaf evaluation, 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). After the middle layer completes the search, we will have a plan to execute: our moves from the principal variation of the minimax tree. 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.

3 Final Remarks

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 (72 vs. 72 units) using SparCraft [4], a StarCraft [1] combat simulator, our Hierarchical Adversarial Search algorithm outperforms adaptations of Alpha-Beta and UCT Search for games with simultaneous and durative moves almost 100% of the time, as well as state-of-the-art Portfolio search about 60% of the time. 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.

In the future, we plan on using another search algorithm at the top layer, as we currently only use the bottom two of the three layers and generate fixed objectives for the middle layer (such as destroy opponent bases). 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

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

[2] Michael Buro, ‘Call for AI research in RTS games’, in *Proceedings of the AAAI-04 Workshop on Challenges in Game AI*, pp. 139–142, (2004).

[3] Michael Chung, Michael Buro, and Jonathan Schaeffer, ‘Monte Carlo planning in RTS games’, in *IEEE Symposium on Computational Intelligence and Games (CIG)*, (2005).

[4] David Churchill. SparCraft: open source StarCraft combat simulation. http://code.google.com/p/sparcraft/, 2013.

[5] David Churchill and Michael Buro, ‘Build order optimization in StarCraft’, in *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, pp. 14–19, (2011).

[6] David Churchill and Michael Buro, ‘Portfolio greedy search and simulation for large-scale combat in StarCraft’, in *IEEE Conference on Computational Intelligence in Games (CIG)*, pp. 1–8. IEEE, (2013).

[7] Santiago Ontañón, ‘The combinatorial multi-armed bandit problem and its application to real-time strategy games’, in *AIIDE*, (2013).

[8] Santiago Ontañón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss, ‘A survey of real-time strategy game AI research and competition in StarCraft’, *TCIAIG*, (2013).

[9] Franisek Sailer, Michael Buro, and Marc Lanctot, ‘Adversarial planning through strategy simulation’, in *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pp. 80–87. IEEE, (2007).

[10] Gabriel Synnaeve, *Bayesian programming and learning for multi-player video games*, Ph.D. dissertation, Université de Grenoble, 2012.

[11] Gabriel Synnaeve and Pierre Bessiere, ‘Special tactics: a Bayesian approach to tactical decision-making’, in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pp. 409–416, (2012).

[12] Samuel Wintermute, Joseph Z. Joseph Xu, and John E. Laird, ‘SORTS: A human-level approach to real-time strategy AI’, in *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, pp. 55–60, (2007).