Game-Tree Search over High-Level Game States in RTS Games

Alberto Uriarte and **Santiago Ontañón**
Computer Science Department
Drexel University
{albertouri,santi}@cs.drexel.edu

Abstract

From an AI point of view, Real-Time Strategy (RTS) games are hard because they have enormous state spaces, they are real-time and partially observable. In this paper, we present an approach to deploy game-tree search in RTS games by using game state abstraction. We propose a high-level abstract representation of the game state, that significantly reduces the branching factor when used for game-tree search algorithms. Using this high-level representation, we evaluate versions of alpha-beta search and of Monte Carlo Tree Search (MCTS). We present experiments in the context of StarCraft showing promising results in dealing with the large branching factors present in RTS games.

Introduction

Real-Time Strategy (RTS) games pose a significant challenge for artificial intelligence (AI) mainly due to their enormous state space and branching factor, and because they are real-time and partially observable (Buro 2003). These challenges have hampered the applicability of game-tree search approaches, such as *minimax* or *Monte Carlo Tree Search* (MCTS) to RTS games, and contribute to the fact that proficient humans can still defeat the best AI solutions for RTS games we have nowadays (Ontanón et al. 2013).

In this paper we present an approach to address the large branching factors in RTS games based around the idea of game state abstraction. Specifically, we focus on using game-tree search for controlling the army movements in full-scale RTS games. In other words, how to maneuver all the combat units around the map during a full game and deciding which enemies to attack and when. Solutions to large-scale adversarial domains are important both due to their applicability to entertainment (creation of more challenging RTS games), and to other domains such as cyber security or disaster planning.

Specifically, our approach focuses on translating the low-level game state to an abstracted high-level game state that captures most of the tactical army information. To construct this, we propose to divide the map into a collection of regions, and represent the distribution of friendly and enemy units in these regions. We show how this abstract representation can be used in the context of game-tree search, and analyze the difference in branching factor between using the high-level representation versus a low-level representation of the game. We evaluate this approach into an actual StarCraft playing agent.

The remainder of this paper is organized as follows. First we provide background on game-tree search in RTS games. Then we present our high-level abstraction approach. After that, we review the two game-tree search algorithms that we used in our evaluation: *alpha-beta* and MCTS. Finally, we present an empirical evaluation in StarCraft, a popular RTS game used as a testbed for RTS Game AI.

Background

RTS is a sub-genre of strategy games where players need to build an economy (gathering resources and building a base) and military power (training units and researching technologies) in order to defeat their opponents (destroying their army and base). From a theoretical point of view, the main differences between RTS games and traditional board games such as Chess are: they are *simultaneous move* games (more than one player can issue actions at the same time), they have *durative actions* (actions are not instantaneous), they are *real-time* (each player has a very small amount of time to decide the next move), they are *partially observable* (players can only see the part of the map that has been explored, although in this paper we assume full observability) and they might be *non-deterministic*.

Classical game-tree search algorithms have problems dealing with the large branching factors in RTS games. For example the branching factor in StarCraft can reach numbers between 30^{50} and 30^{200} (Ontanón et al. 2013). To palliate this problem several approaches have been explored. For example, (Chung, Buro, and Schaeffer 2005) applied Monte Carlo planning to an RTS game by simplifying the decision space: assuming that each player can choose only one amongst a finite set of predefined plans. Balla and Fern (2009) applied the UCT algorithm to tactical assault planning in Wargus. To make game-tree search applicable at this level, they perform an abstraction of the game state representation grouping the units in groups but keeping information of each individual unit at the same time, and allowing only two types of actions per group: attack

and merge with another group. *Alpha-beta* has been used in scenarios with *simultaneous moves* (Saffidine, Finnsson, and Buro 2012) and Churchill et al. (Churchill, Saffidine, and Buro 2012) extended it with *durative actions*, being able to handle situations with up to eight versus eight units without using abstraction. An improvement of this work is presented in (Churchill and Buro 2013), where they defined scripts to improve the move-ordering; and experiments with UCT considering durations and a Portfolio Greedy Search; showing good results in larger combat scenarios than before. Ontañón (2013) presented a MCTS algorithm called NaïveMCTS specifically designed for RTS games, and showed it could handle full-game, but in the context of a simple RTS game. Some work has been done also using Genetic Algorithms and High Climbing methods (Liu, Louis, and Nicolescu 2013) or Reinforcement Learning (Jaidee and Muñoz-Avila 2012).

In this paper we build upon some of those ideas to present an approach that combines game state abstraction with scalable game-tree search approaches to handle the army strategy in RTS games.

High-level Abstraction in RTS Games

In order to deploy game-tree search in RTS games, we defined an abstraction mechanism that provides a high-level representation of the game state. In the following subsections, we describe the different elements involved in this abstraction: game state, action space, and mapping from low-level game state representation to high-level representation.

State Representation

Since we focus on controlling combat units, our high-level representation focuses on placement of those units in the game state. Moreover, instead of representing the exact location of groups of units, we use a high-level representation of the map, by dividing it into a set of regions using Perkins algorithm (Perkins 2010), implemented in the BWTA library. This algorithm decomposes the map into a set of regions connected by chokepoints. Let us call R' to the set of regions, and C to the set of chokepoints resulting from this analysis. From this information, we generate a set of regions R, where each region is a set of tiles from the map, and the union of all regions in R is the complete set of walkable tiles in the map. R is defined as follows:

- For each chokepoint $c \in C$, there is an region $r \in R$ containing all the cells whose distance to the center of c is smaller than the radius of the chokepoint (defined by BWTA).
- For each region $r' \in R'$, there is a region $r \in R$ containing all the cells from region r' that are not part of any region generated from a chokepoint.

As a result we have an abstract interpretation of our map as a set of n regions $R = \{r_1, ..., r_n\}$, (similarly to the representation proposed by Synnaeve and Bessière (2012)). In Figure 1 we can see a snapshot of a game state and its abstract representation, where we can appreciate the region decomposition (each region r_i is labeled by its index i), and

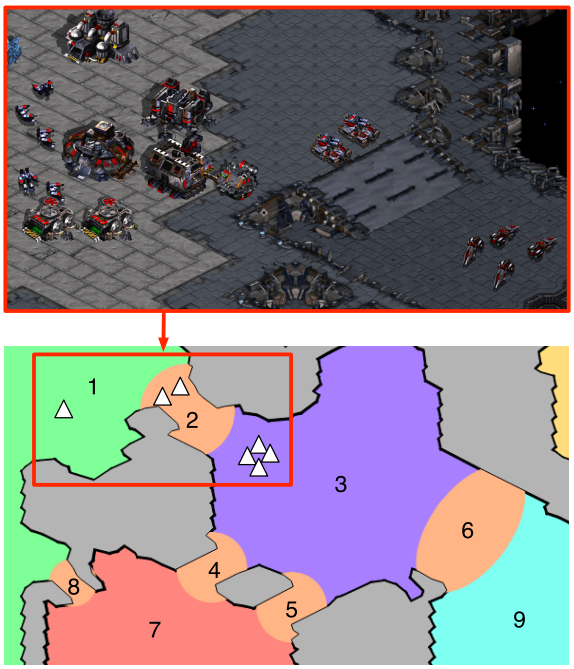

Figure 1: Top image is a snapshot of a StarCraft game. Bottom image illustrates the representation of a game state using our high-level abstraction with the ID of each region.

where non walkable regions in gray. Since the map is invariant through all the game, this process only has to be done once, at the beginning of the game. Moreover, in order to speed up our calculations later on, we precompute a matrix with all the distances between the center points of each pair of regions.

As we stated before, we focus on combat units. Hence, we only include those in the abstract representation, and ignore all the workers and buildings (except the command center buildings). We include the command centers because we are using them as an indicators of where the enemy created bases, since this helps the search algorithm in knowing what regions to attack or to defend in the absence of combat units. Otherwise our army would be "blind" and unable to attack the enemy base or defend our bases unless there are combat units to attack or defend. The remaining units are grouped by unit type and region. Thus, the whole game state is represented as a matrix with one row per unit type and region, and the following columns (illustrated in Table 1):

- *Player*. Which player controls this group.
- *Type*. Type of units in this group.
- *Size*. Number of units forming this group.
- *Region*. Which region is this group in.
- *Order*. Which order is this group currently performing.
- *Target*. If the order requires, the ID of the target region (e.g. towards which region is the group moving).
- *End*. In which game frame is the order estimated to finish.

Table 1: Grouped units in the high-level abstraction.

Player	*Type*	*Size*	*Rgn.*	*Order*	*Target*	*End*
1	Base	1	1	N/A	-	-
1	Tank	2	2	Move	3	230
1	Vulture	4	3	Idle	-	400

Actions

Let us now define the set of actions that a player can perform given a high-level representation of the game state. We define the following set of possible actions for each high-level group: *N/A*, *Move*, *Attack* and *Idle*:

- *N/A*: only for buildings as they cannot perform any action,
- *Move*: move to an adjacent region,
- *Attack*: attack any enemy in the current region, and
- *Idle*: do nothing during 400 frames.

For example, the groups of player 1 in the game state in Table 1 can execute the following actions:

- Base will be stuck in *N/A* for the whole game since buildings cannot perform any other action in our high-level representation (we only consider combat).
- Assuming that the Tanks were not executing any action, they could *Move* to region 3 or 1, or stay *Idle*. They cannot *Attack* since there is no enemy in the same region (2).
- Assuming that the Vultures were not executing any action, they can *Move* to regions 2, 4, 5 or 6, or stay *Idle*.

Thus, since player 1 can issue any combination of those actions to her units, the branching factor of this example is $(1) \times (2+1) \times (4+1) = 15$. An important aspect to analyze is how much does this high-level representation reduce the branching factor with respect to having a low-level representation. To answer that, we analyzed the branching factor in StarCraft game states every 400 frames during a regular game. Figure 2 shows the branching factor from three points of view: *Low Level* is the actual branching factor of StarCraft when all actions of all units are considered in the low-level game state representation, *SparCraft* is a simplified branching factor of the low-level game state only considering combat units (as calculated by the combat simulator SparCraft[1] (Churchill, Saffidine, and Buro 2012)), and *High Level*, which the branching factor using our high-level representation. As expected, the branching factor using the low-level representation of the game state is too large for game-tree search approaches (reaching values of up to 10^{300} in our experiment). Our high-level abstraction is able to keep the branching factor relatively low (reaching a peak of about 10^{10}). The branching factor considered by SparCraft is in between, but still many orders of magnitude larger (reading more than 10^{200}).

Mapping Low-Level and High-Level States

The goal of our high-level abstraction is to use a game-tree search for high-level control of combat units in actual full-scale RTS games like StarCraft. Therefore, in order to employ it, we need to define a mapping between low-level states and high-level states that is to be used while playing the game. Moreover, our search process does not produce the exact micro actions that the combat units need to perform, but high-level actions such as moving from one region to another, or attacking a particular enemy group. Thus, we assume that there is a low-level agent that translates the low-level state to our high-level representation and then is capable of translating the actions defined above (*N/A*, *Move*, *Attack*, *Idle*) into actual low-level micro actions.

[1]https://code.google.com/p/sparcraft/

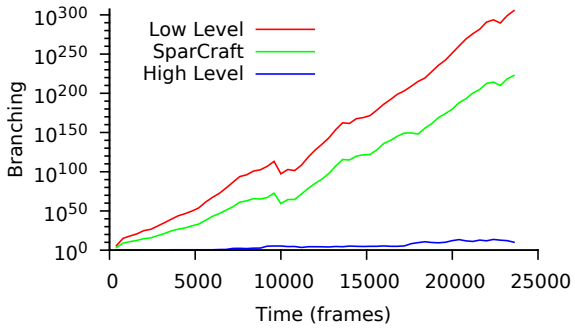

Figure 2: Comparison of the branching factor using our high-level representation with respect to the branching factor using the low-level StarCraft game states and SparCraft abstraction.

Most StarCraft bots are decomposed in several individual agents that perform different tasks in the game, such as scouting, construction, etc. (Ontanón et al. 2013). One of such agents is typically in charge of combat units, and is in charge of controlling a military hierarchy architecture. The high-level game-tree search approach presented in this paper is designed to replace such agent. Figure 3 shows the military hierarchy architecture used in the StarCraft bot used in our experiments (Uriarte and Ontañón 2012), and where our high-level game-tree search is injected (*abstract layer*). As described above, our high-level search process assigns actions to groups of units. Our StarCraft bot uses the intermediate concept of *squads* to control groups of units, which is analogous to the concept of groups in the high-level representation. Unfortunately, generating the set of group in a given low-level game state given the set of squads is not trivial, since, for example, units in a given squad might appear to split in two groups when moving from one region to another, as some units arrive earlier than others, thus breaking squad continuity. Additionally, if two squads with units of the same type arrive to the same region, they would be grouped together into a single large group according to the high-level representation, which might not be desirable.

To address these problems, given a squad q in the low-level, we add each unit $u \in q$ to the high-level group, recording which high-level group the unit is being mapped to. Once all the units in the squad have been added to the high-level state, all the units are reassigned to the high-level group to which most of the units in q had been assigned. During

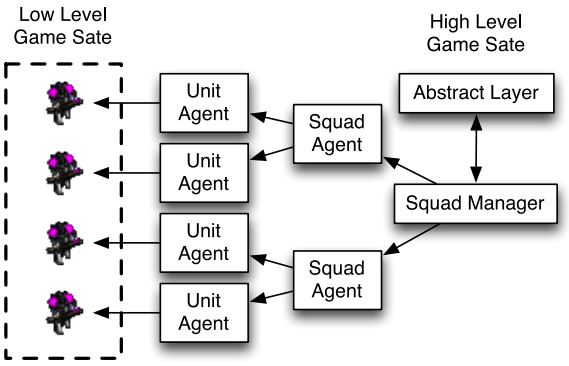

Figure 3: The different layers of abstraction from Low-Level Game State to High-Level Game State.

this process, the *abstract layer* records the set of squads that were mapped to each of the high-level groups (notice that a high-level group can have more than one squad since two different squads in the same region and with the same unit type, will be one unique high-level group).

Once the low-level state has been mapped to a high-level game state, we can use a game-tree search algorithm to get an action for each group (which can be mapped to orders for each squad for execution by the other agents of the StarCraft playing bot). Since RTS games are real-time, we perform a search process periodically (every 400 game frames), and after each iteration, the orders that each different squad is executing are updated if different from the one the squad is currently executing.

High-Level Game-Tree Search

The previous section has described our high-level game state representation, and how to compute the set of actions in a given game state (corresponding to the children states of a given state in the game-tree), how to map low-level states to high-level states and how to map high-level actions to low-level actions. However, two more pieces are required in order to perform game-tree search. The first one is a *state forwarding* function that can roll the game forward using the high-level game state representation, and the second one is the *state evaluation* function. The next subsections explain a basic way to have this; and then we explain two different game-tree search algorithms that we experimented with (ABCD (Churchill, Saffidine, and Buro 2012) and MCTSCD) that can exploit the high-level game state representation presented in this paper.

High-Level State Forwarding

As we already explained, one of the particularities of RTS games is that they have *simultaneous* and *durative* actions. For this reason we need a state forwarding process that can simulate the game for as many frames as needed to reach the next decision point (a game state node where at least one player can execute an action).

Our high-level state forwarding has two components:

- **End frame action prediction**. Given a durative action (*Move*, *Attack* or *Idle*) for a group g, we predict in which game frame the action will be finished. For *Move*, we get the group velocity $g.velocity$ (all the units of the group are the same unit type) and the group location $g.location$, and target region $m.target$. Using the region distance matrix previously computed, we can apply the formula $g.velocity \times distanceMatrix[g.location][m.target]$ to get the expected end frame. For *Attack*, we compute the minimum amount of time to finish the battle. To get this we use the *Damage Per Frame* (DPF) of each group in the region in conflict and the total *Hit Points* (HP) of each group. Then we apply the following equation $timeToKill_{enemy} = enemy_{HP}/self_{DPF}$ to get both times and the minimum is when the battle will end. The *Idle* action always takes 400 frames.
- **Simulation**. When we reach a state where none of the players can make a move, then we identify the action with the smallest *end* time and forward the game time to that moment. At this point we need to update the outcome of the finished actions. For the *Move* action we only need to update the group position with the target position. For the *Attack* action we remove all the units of the defeated group, and we also remove the units that the defeated group can manage to kill before being destroyed. Since this is only an approximation we consider that the survivors are taking damage in a sequential fashion rather than an even spread.

High-Level State Evaluation

A game state evaluation helps us to define how good or bad is a non terminal state (or how promising it is). In our experiments we use a simple game state evaluation where we use the *destroy score* of a unit. This score is defined by StarCraft and it is related to the value of the resources needed to produce that unit. So, given a set of high-level friendly groups F of size n and a set of high-level enemy groups E of size m, we calculate the following reward: $score = \sum_1^n (F_i.size \times F_i.destroyScore) - \sum_1^m (E_j.size \times E_j.destroyScore)$.

High-Level ABCD

To determine whether our high-level abstraction captures enough information of the game state to produce meaningful actions, we experimented with two different game-tree search algorithms. The first one is a variant of *alpha-beta* search capable of dealing with simultaneous moves and durative actions defined by Churchill et al. (Churchill, Saffidine, and Buro 2012) called ABCD (Alpha-Beta Considering Durations). They showed how this algorithm can be applied to handle low-level control for small scale combat situations in RTS games. ABCD does not automatically alternate between players, instead it determines the next player that can do a move based on the duration of the actions that are currently being executed. In game states where both players can issue actions simultaneously, ABCD defines a policy to establish the order of the player to move. In their work they experimented with two policies: RAB (Random-

Algorithm 1 MCTS Considering Durations

```
1: function MCTSSEARCH(s_0)
2:     n_0 ← CREATENODE(s_0,∅)
3:     while withing computational budget do
4:         n_l ← TREEPOLICY(n_0)
5:         △ ← DEFAULTPOLICY(n_l)
6:         BACKUP(n_l, △)
7:     return (BESTCHILD(n_0)).action
8:
9: function CREATENODE(s,n_0)
10:    n.parent ← n_0
11:    n.lastSimult ← n_0.lastSimult
12:    n.player ← PLAYERTOMOVE(s,n.lastSimult)
13:    if BOTHCANMOVE(s) then
14:        n.lastSimult ← n.player
15:    return n
16:
17: function DEFAULTPOLICY(n)
18:    lastSimult ← n.lastSimult
19:    s ← n.s
20:    while withing computational budget do
21:        p ← PLAYERTOMOVE(s,lastSimult)
22:        if BOTHCANMOVE(s) then
23:            lastSimult ← p
24:        simulate game s with a policy and player p
25:    return s.reward
```

Alpha-Beta presented by (Kovarsky and Buro 2005)), that randomizes the player that will play first in each simultaneous node; and *Alt*, which alternates between players. They report better results with the *Alt* policy in their experiments, which is the one we employed in our work.

High-Level MCTS Considering Durations (MCTSCD)

MCTS is a family of algorithms which is better suited with games with large branching factors than the variants of *minimax*. The main concept of MCTS is that the value of a state may be approximated using repeated stochastic simulations from the given state until a leaf node (or a terminal condition). There are two important parts: the *tree policy* is used to balance the *exploration* (look in areas that have not been explored yet) and *exploitation* (look at the most promising areas of the tree); the other part is the *default policy* used to simulate games until a terminal node is reached. The simplest default policy that can be used to run the simulations is selecting uniform random actions for each player.

Research exists on adapting MCTS to games with *simultaneous moves* (Teytaud and Flory 2011; Churchill and Buro 2013; Ontañón 2013). In our approach we modify a standard MCTS algorithm using an ϵ-greedy tree policy to be able to process simultaneous and durative actions, and we call it Monte Carlo Tree Search Considering Durations (MCTSCD). Another recently defined MCTS-variant that considers simultaneous and durative actions is UCTCD (Churchill and Buro 2013). Algorithm 1 shows the pseudocode of the MCTSCD algorithm, where the simultaneous move policy is defined in PLAYERTOMOVE$(state, lastSimultaneousPlayer)$.

Experimental Evaluation

In order to evaluate the performance of our approach, we used the RTS game StarCraft. We incorporated our abstraction layer into a StarCraft bot (Uriarte and Ontañón 2012) and evaluated the performance of using both ABCD and MCTSCD to command our army during a real game. The following subsections present our experimental setup and the results of our experiments.

Experimental Setup

Dealing with partial observability, due the *fog of war* in StarCraft is out of scope in this paper. Therefore we disable the fog of war in order to have perfect information of the game. We also limited the length of a game to avoid situations where our bot is unable to win because it cannot find all the opponent's units (StarCraft ends when all the opponent units are destroyed). In the StarCraft AI competition the average game length is about 21,600 frames (15 minutes), and usually the resources of the initial base are gone after 26,000 frames (18 minutes). Therefore, we decided to limit the games to 20 minutes (28,800 frames). If we reach the timeout we evaluate the current game state using the evaluation function to decide who won the game.

In our experiments, we perform one high-level search every 400 frames, and we pause the game while the search is taking place for experimentation purposes. As part of our future work, we want to explore splitting the search along several game frames, instead of pausing the game.

For ABCD we defined the following limits. First, the maximum depth is limited to 3. Second, the number of children to be considered at each node in the tree is limited to 100,000 nodes (given that the branching factor can grow enormously towards the mid game). And finally, we limited the ABCD execution to 30 seconds, that means that if our search is taking longer than that, after 30 seconds, ABCD will start to close all the open nodes to return the best actions found so far. In the case of MCTSCD we use a ϵ-greedy tree policy with $\epsilon = 0.2$, a random move selection for the *default policy* and an *Alt* policy to decide which player will play first in a simultaneous node. We also limit the depth of the *tree policy* to 10, and run MCTSCD for 2,000 playouts with a length of 7,200 game frames.

We used two different tournament StarCraft maps (Benzene and Destination) for our evaluation and run 40 games on each one of the maps with our bot playing the Terran race against the built-in Terran AI of StarCraft. We also compare the results against a highly optimized scripted version (participating in the StarCraft AI competition in recent years).

Results

Table 2 shows the results we obtained with each algorithm against the built-in AI. The columns labeled as *Avg. Eval* and $\% > 0$ show the average value of the evaluation function at the end of the game, and the percentage of games

Table 2: Results of two game-tree search approaches using our high-level game state representation and a scripted AI.

Algorithm	*Map*	*Avg. Eval*	*% > 0*	Avg. % overwrt.
Scripted	Benzene	35853.33	100.00	-
Scripted	Destination	32796.51	100.00	-
ABCD	Benzene	16020.45	81.82	83.05
ABCD	Destination	18226.32	87.72	88.05
MCTSCD	Benzene	16170.51	89.74	83.74
MCTSCD	Destination	20753.85	84.62	92.25

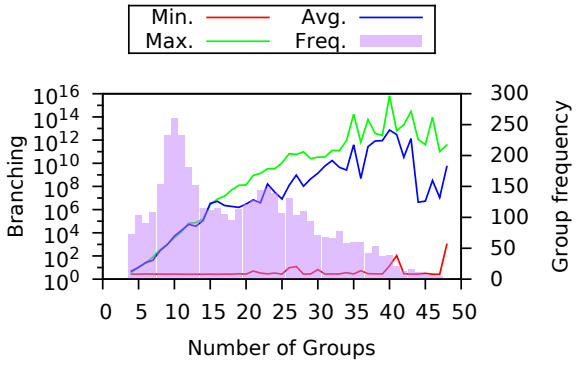

Figure 4: Average branching factor grouped by the number of groups in the high-level state.

where our system was ahead at the end of the game. As we can see, MCTSCD achieves slightly better results than ABCD. *Avg. % overwritten* shows the percentage of times that the result of a search was a different action than the action currently being executed by a group of units. As a reference point we compare the results against a highly optimized scripted version of the bot showing that the scripted version still achieves a higher win ratio. A close examination revealed that both ABCD and MCTSCD sometimes were not able to search deep enough in the tree to realize that they should attack the enemy, when the enemy was far away in the map. Future work will focus on addressing this problem by increasing play out length, and by adding deterministic playouts to the ABCD version as done in (Churchill, Saffidine, and Buro 2012). Moreover, the fact that both ABCD and MCTSCD played better than the built-in AI of StarCraft indicates that our high-level abstraction is useful in transforming the problem of large-scale combat in RTS games to a level of abstraction where game-tree search is feasible and where the resulting actions are meaningful in the game.

To get further insights into the results, Figure 4 shows the average branching factor as a function of the number of groups in a high-level game state. As we can see the branching factor only grows beyond 10^{10} when having a large number of groups (30 or more), which, as shown in the group frequency bars, is infrequent. In StarCraft the maximum number of units at a given time is limited to 400 (200 for each player), therefore the theoretical limit of the number of groups in our high-level representation is 400. But having 400 different type of units all spread over the map is really

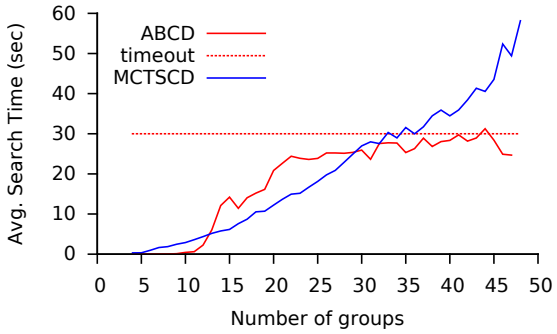

Figure 5: Average search time for each algorithm grouped by the number of groups in the high-level state.

unlikely. Hence, as we can see in Figure 4 the most common number of groups in our high-level representation is around 10 with an average branching factor of 10^4.

We also analyze the amount of time spent doing search. This is a crucial point since we want to be able to execute the search during a RTS game. Figure 5 shows the evolution of time spent per search by each algorithm as the number of groups in the high-level state increases. Notice how ABCD starts to saturate (stopping the search due the defined timeout) when we have more than 30 groups. After having more than 19 groups, our implementation of ABCD starts hitting the limit of 100,000 nodes explored per tree level, and thus time stabilizes. MCTSCD's time increases with the number of groups, since the time it takes the simulator to roll the game state forward increases with the number of groups.

Conclusions

This paper has presented an approach to game-tree search in large-scale RTS combat scenarios based on game-state abstraction. We presented a state representation, a procedure to map low-level states to high-level states and a simulator that can roll the state forward. We also presented a new variant of MCTS that considers durative actions (MCTSCD). Our experimental results indicate that the information captured in our high-level abstraction is at the appropriate level of detail to reduce the branching factor, while still producing meaningful actions.

As part of our future work, we would like to first explore different bandit strategies for MCTS, such as Naïve Sampling that better exploit the limited number of iterations that MCTS can be executed in an RTS game. Additionally, we would like to incorporate additional aspects of the game (such as construction, and resource gathering) into the game-tree search approach, to achieve a RTS-playing system purely based on game-tree search. Finally, we would also like to improve our game simulator to learn during the course of a game, and produce more accurate combat estimations, independently of the RTS game being used.

References

Balla, R.-K., and Fern, A. 2009. UCT for tactical assault planning in real-time strategy games. In *International Joint*

Conference of Artificial Intelligence, IJCAI, 40–45. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Buro, M. 2003. Real-time strategy games: a new AI research challenge. In *Proceedings of IJCAI 2003*, 1534–1535. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

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. 2013. Portfolio Greedy search and Simulation for Large-Scale Combat in StarCraft. In *CIG*. IEEE.

Churchill, D.; Saffidine, A.; and Buro, M. 2012. Fast Heuristic Search for RTS Game Combat Scenarios. In *AIIDE*.

Jaidee, U., and Muñoz-Avila, H. 2012. CLASSQ-L: A Q-Learning Algorithm for Adversarial Real-Time Strategy Games. In *AIIDE*.

Kovarsky, A., and Buro, M. 2005. Heuristic search applied to abstract combat games. In *Advances in Artificial Intelligence*. Springer. 66–78.

Liu, S.; Louis, S. J.; and Nicolescu, M. N. 2013. Comparing heuristic search methods for finding effective group behaviors in RTS game. In *IEEE Congress on Evolutionary Computation*. IEEE.

Ontañón, S. 2013. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *AIIDE 2013*, 58–64.

Ontanón, 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. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)* 5:1–19.

Perkins, L. 2010. Terrain Analysis in Real-Time Strategy Games: An Integrated Approach to Choke Point Detection and Region Decomposition. In *AIIDE*, 168–173.

Saffidine, A.; Finnsson, H.; and Buro, M. 2012. Alpha-Beta Pruning for Games with Simultaneous Moves. In *26th AAAI Conference (AAAI)*. Toronto, Canada: AAAI Press.

Synnaeve, G., and Bessière, P. 2012. A Bayesian Tactician. In *Computer Games Workshop at ECAI*.

Teytaud, O., and Flory, S. 2011. Upper confidence trees with short term partial information. In *Applications of Evolutionary Computation*. Springer. 153–162.

Uriarte, A., and Ontañón, S. 2012. Kiting in RTS Games Using Influence Maps. In *AIIDE*.