High-Level Representations for Game-Tree Search 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 explore an approach to deploy game-tree search in RTS games by using game state abstraction, and explore the effect of using different abstractions over the game state. Different abstractions capture different parts of the game state, and result in different branching factors when used for game-tree search algorithms. We evaluate the different representations using Monte Carlo Tree Search in the context of StarCraft.

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; Robertson and Watson 2014).

In order to asses the applicability of game-tree search approaches we explore the possibility of using abstraction of the game state to reduce the complexity. Specifically, this paper presents an evaluation of four different game state abstractions. We show their effects on gameplay strength and their impact on the resulting branching factor.

We build upon our work on game-tree search over abstract game representations (Uriarte and Ontañón 2014), which used an abstraction based on dividing the terrain in regions using the BroodWar Terrain Analysis (BWTA), and grouped the units by type and region. In this paper, we explore different space partitions and unit groupings, measuring different attributes to evaluate the performance.

The remainder of this paper is organized as follows. First we provide background on game-tree search in RTS games. Then we present different high-level abstraction approaches to simplify the complexity and we review the Monte Carlo Tree Search algorithm that we used in our evaluation (MCTSCD). Finally, we present an empirical evaluation using StarCraft, a popular RTS game used as a testbed for RTS Game AI, where we evaluate the performance of a bot (an AI agent) and the accuracy of the simulator defined in the MCTSCD.

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) and they are *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 at any given time only one amongst a finite set of predefined plans. Balla and Fern (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 (Ontañón 2013) presented a MCTS algorithm called NaïveMCTS specifically designed for RTS games, and showed it could handle full-game, but small scale RTS game scenarios. 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).

High-level Abstraction in RTS Games

The key idea of the different state abstractions explored in this paper is to first simplify the game map by dividing it in a set of regions. Specifically, as in our previous work (Uriarte and Ontañón 2014), we used Perkins algorithm (Perkins 2010), implemented in the BWTA library, for this purpose. Since the map is invariant through all the game we only need to compute this once. With this region decomposition now the combat units (and the main bases) are grouped by unit type and region. For each group we capture the following information: *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* (the ID of the target region) and *End* (In which game frame is the order estimated to finish).

Based on this idea, we propose four different abstractions:

- **A-RC**: This is our baseline abstraction, and corresponds to the one used in our previous work (Uriarte and Ontañón 2014). Similar to the abstraction proposed by Synnaeve (Synnaeve and Bessière 2012), in addition to the regions returned by Perkins' algorithm, we add one additional region for each chokepoint in the map (with center at the center of the chokepoint, and a circular area of the same diameter as the chokepoint). We only consider military units in this abstraction, although for the specific case of StarCraft, we also add the main bases (Terran Command Centers, etc.), since it is necessary for the AI to know where to send units to attack.
- **A-RCB**: Same as A-RC, but we also add all the buildings in the game.
- **A-R**: Like A-RC, but without having additional regions for chokepoints. In this way we have a simpler high-level map representation. To evaluate the impact of these simplification we computed the number of nodes, the average connectivity and the diameter of the generated graph are shown in Table 1.
- **A-RB**: Like A-R, but also adding all the buildings in the game.

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.

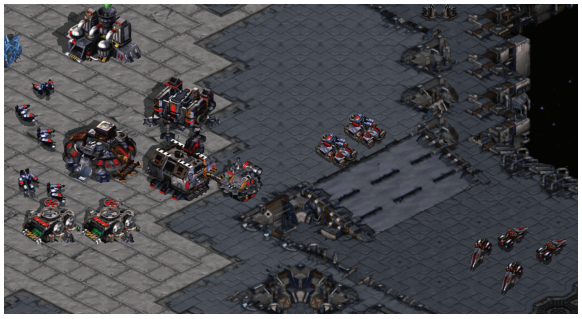

Figure 1: Snapshot of a StarCraft game.

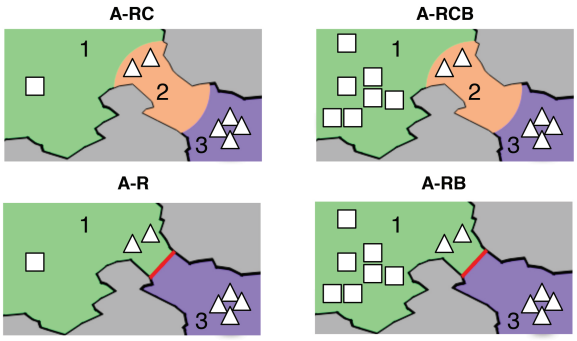

Figure 2: Representation of a game state using different high-level abstraction with the ID of each region. Triangles are military units. Squares are buildings.

Figure 1 shows a portion of a real game state of a StarCraft game. And in Figure 2 we graphically illustrate the different high-level abstractions defined previously using the game state from Figure 1. The actual internal representation of the high-level game state is simply a matrix with one row per unit type $\times$ region, where in each row we store the number of units of that type, and the action they are currently executing. Table 1 shows, for a few StarCraft maps, the number of regions in which each map is divided, the average connectivity of each region, and the diameter of the resulting graph.

High-Level Game-Tree Search

To evaluate the proposed high-level game state abstraction, we decided to use the game-tree search algorithm MCTSCD (Uriarte and Ontañón 2014). MCTSCD is a variant of *Monte Carlo Tree Search* algorithm that can handle simultaneous moves and durative actions (features present in all RTS games). To be able to perform any MCTS algorithm we need to define two components. 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. We used the ones defined by MCTSCD authors, where:

- The *state forwarding* first tries to predict in which game frame the action of each group will be finished. To do this we use the group velocity and the distance between re-

Table 1: Statistics of different StarCraft maps and map abstractions.

Map	*Abs.*	*Size*	Avg. Connect.	*Diam.*
(2)Benzene	RC	59	1.492	8
(2)Benzene	R	37	1.189	16
(2)Destination	RC	65	1.538	8
(2)Destination	R	40	1.250	16
(2)Heartbreak Ridge	RC	68	1.471	9
(2)Heartbreak Ridge	R	43	1.163	18
(3)Aztec	RC	70	1.371	6
(3)Aztec	R	46	1.043	12
(3)Tau Cross	RC	58	1.379	6
(3)Tau Cross	R	38	1.053	12
(4)Python	RC	33	1.212	4
(4)Python	R	23	0.870	8
(4)Fortress	RC	29	1.103	4
(4)Fortress	R	21	0.762	8
(4)Empire of the Sun	RC	105	1.448	9
(4)Empire of the Sun	R	67	1.134	18
(4)Andromeda	RC	81	1.284	10
(4)Andromeda	R	55	0.945	20
(4)Circuit Breaker	RC	93	1.462	6
(4)Circuit Breaker	R	59	1.153	12

gions to predict movements. And the *Damage Per Frame* (DPF) of each group in the region in conflict to predict the output of a combat. Then we identify the action with the smallest *end time* and forward the game time to that moment. Notice that we do not implement any kind of merge operation, i.e. if two groups of the same unit type meet in the same region, we do not merge both groups. We consider that since both groups have different "timing" one of them will perform actions faster than the other.

- To compute the *state evaluation* we use the *destroy score* of a 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: $score = \sum_0^n (F_i.size \times killScore) - \sum_0^m (E_j.size \times killScore)$, where the *killScore* is a score that StarCraft internally assigns to each unit.

Experimental Evaluation

In order to compare the performance of the different abstractions, we used the RTS game StarCraft. We incorporated our abstraction layer into a StarCraft bot (Uriarte and Ontañón 2012) and evaluated the performance using 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, and is part of our future work. 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 call MCTSCD to perform high-level search once every 400 frames (16,6 seconds of real gameplay). Taking into account that the minimum training time for a unit is 300 frames, this gives a confidence margin to reevaluate our decision with the new units in the game state. For experimentation purposes, we pause the game while the search is taking place. As part of our future work, we want to explore splitting the search along several game frames, instead of pausing the game.

For MCTSCD we use a ϵ-greedy tree policy with $\epsilon = 0.2$, a random move selection for the *default policy* and an *Alt* policy (Churchill, Saffidine, and Buro 2012) 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 1,000 playouts with a length of 2,880 game frames (120 seconds of real gameplay). We experimented with 4 different high-level abstract representations as we explained in previous sections, leading to the following configurations: MCTSCD-RC, MCTSCD-RCB, MCTSCD-R and MCTSCD-RB.

We used the Benzene StarCraft map for our evaluation and run 40 games with our bot playing the Terran race against the built-in Terran AI of StarCraft. We compare the results against a highly optimized scripted version (which has participated in the StarCraft AI competition).

Moreover, we performed two sets of experiments (explained in the following two sections). In the first, we evaluate the performance of our bot, when using each of the four abstract representations. In the second, we evaluated how accurate are the simulations used internally by the search algorithm (to roll the state forward) using each of the four abstract representations (i.e., which of the representations result in a game tree that is a more accurate representation of the actual game?).

Table 2: Results of MCTSCD using different high-level game state representations and a scripted AI.

Algorithm	*Avg. Kill Score* Δ	$\% > 0$
Scripted	9171.25	100.00
MCTSCD-RC	2562.50	97.50
MCTSCD-RCB	2183.75	87.50
MCTSCD-R	4796.25	97.50
MCTSCD-RB	5051.25	97.50

Bot Performance Evaluation

Table 2 shows the results we obtained with each configuration. The column labeled as *Avg. Kill Score* Δ shows

Table 3: Similarity between the predicted game state and the current game state.

Algorithm	*Similarity*
MCTSCD-RC	0.573152
MCTSCD-RCB	0.768658
MCTSCD-R	0.688179
MCTSCD-RB	0.818043

the average value of the difference on the kill score of each player at the end of the game. The kill score is a score that StarCraft maintains, based on how many enemy units each player manages to kill during the game, and on specific scores assigned to each unit. The column labeled as $\% > 0$ shows the percentage of games where the Kill Score Δ was positive (i.e., our bot achieved a higher kill score than the opponent). 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. The results reveal two important facts. First, although the win ratio adding chokepoints or not is similar (MCTSCD-RC vs MCTSCD-R), without including the chokepoints (MCTSCD-R) we achieve a better average kill score Δ, meaning that we were winning with a better margin. The second fact is the poor performance when we consider all the buildings (MCTSCD-RCB) which it has the worst win ratio. But if we look at the MCTSCD-RB, it is better than the other high-level abstraction. Our hypothesis is that including chokepoints in the game state creates confusion that is carried over when buildings are included.

Simulation Accuracy Evaluation

In order to gain more insight into the experiments, in this section we evaluate the accuracy of the simulator being used in the roll-out step of the MCTSCD. To evaluate this we defined a similarity measure between game states, based on the Jaccard similarity coefficient. We use $HLGM_t$ to denote the similarity between the actual high-level state at time t, with the high-level state at time t that resulted from simulation, given the actions selected by MCTSCD to reach time t. More formally we use the method $Simulate(GameState, Orders, Frames)$ where $GameState$ is a high-level game state, orders is the orders to execute each unit, and frames the amount of frames to forward the high-level game state. So, we execute the method with the following arguments:

$$HLGMSim = Simulate(HLGM_{x-400}, Orders, 400)$$

Then we can use Equation 1 to compute the Jaccard index.

$$J(HLGM, HLGMSim) = \frac{|HLGM \cap HLGMSim|}{|HLGM \cup HLGMSim|} \quad (1)$$

This Jaccard index helps us to see how accurate is our simulator, in Table 3 we can observe how the map abstraction with region and chokepoints (MCTSCD-RC) has the worst

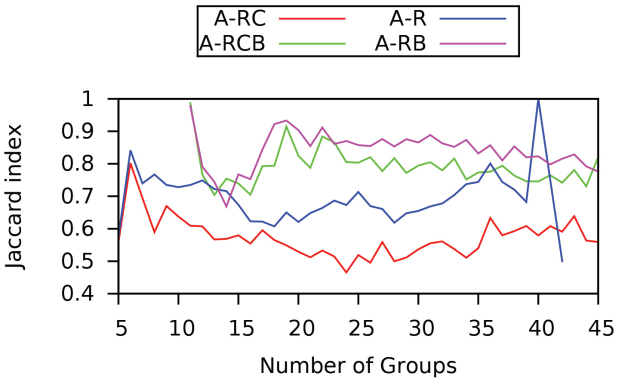

Figure 3: Average Jaccard index grouped by the number of groups in the high-level state.

similarity, in other words, the simulator makes less accurate predictions of unit positions. As expected, we get the bests results when we add all the building in our high-level representation (MCTSCD-RCB and MCTSCD-RB) since buildings are easy to predict due their lack of movement. But the remarkable part is that the simpler map abstraction only considering regions (MCTSCD-R) has better predictions than the baseline (MCTSCD-RC). That also explains why we get better average evaluation scores with this simpler abstraction.

To get further insights into the similarity we analyzed the Jaccard index by group size. As we can see on Figure 3, adding the chokepoints to our map abstraction deteriorates the accuracy of the prediction. The reader has to keep in mind that this similarity coefficient is only computed with the prediction after 400 frames when in the MCTSCD we simulate until reach 2,880 frames. Therefore the error between the actual game state and the predicted game state at the end of a playout of length 2,880 could be even larger.

Conclusions

This paper has presented our experiments on different ways to reduce the search complexity using abstractions of the game state. We also present a methodology to evaluate the accuracy of the simulator inside MCTS for RTS games. Our experimental results indicate that it is better to keep the abstraction simple in order to get better predictions (and therefore better performance of our agent). So, the map abstraction chokepoints are not needed to capture all the needed detail for our high-level abstraction, while we can afford the inclusion of all the buildings for a better search.

As part of our future work, we would like to improve the game tree search algorithm (for example, exploring different bandit strategies for MCTS or to be able to deal with partial observability). Additionally, we would like to continue exploring abstractions and their tradeoffs. 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*.

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.

Robertson, G., and Watson, I. 2014. A Review of Real-Time Strategy Game AI. *AI Magazine*.

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*.

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

Uriarte, A., and Ontañón, S. 2014. Game-Tree Search over High-Level Game States in RTS Games. In *AIIDE*.