Adversarial Hierarchical-Task Network Planning for Complex Real-Time Games

Santiago Ontañón[1] and **Michael Buro**[2]
[1] Drexel University, Philadelphia, USA
santi@cs.drexel.edu
[2] University of Alberta, Edmonton, Canada
mburo@ualberta.ca

Abstract

Real-time strategy (RTS) games are hard from an AI point of view because they have enormous state spaces, combinatorial branching factors, allow simultaneous and durative actions, and players have very little time to choose actions. For these reasons, standard game tree search methods such as alpha-beta search or Monte Carlo Tree Search (MCTS) are not sufficient by themselves to handle these games. This paper presents an alternative approach called Adversarial Hierarchical Task Network (AHTN) planning that combines ideas from game tree search with HTN planning. We present the basic algorithm, relate it to existing adversarial hierarchical planning methods, and present new extensions for simultaneous and durative actions to handle RTS games. We also present empirical results for the μRTS game, comparing it to other state of the art search algorithms for RTS games.

Introduction

Real-Time Strategy (RTS) games are popular video games that are particularly challenging for AI research [Buro, 2003; Ontanón *et al.*, 2013]. Previous work in this area [Ontañón, 2013] has shown that standard game tree search methods such as alpha-beta search [Knuth and Moore, 1975], or Monte Carlo Tree Search (MCTS) [Browne *et al.*, 2012] are not enough by themselves to play proficiently and to challenge human supremacy in these games. In this paper we present a new approach to search in RTS games called adversarial hierarchical-task network (AHTN) planning that integrates concepts of HTN planning [Erol *et al.*, 1994] into game tree search to address one of the key open challenges in RTS games: their enormous branching factor. We evaluate our approach in the μRTS game, a minimalistic RTS game, that has been used in the past to evaluate new approaches to RTS games [Ontañón, 2013; Shleyfman *et al.*, 2014].

RTS games are complex because they have large state spaces and are real-time. In this context "real-time" means that: 1) RTS games typically execute 10 to 50 simulation cycles per second, leaving just a fraction of a second to decide the next move, 2) players do not take turns (like in Chess), but instead can issue *simultaneous actions* at the same time, 3) *concurrent actions* are allowed (i.e., players can issue actions in parallel to as many game objects as they want), and 4) actions are *durative*. In addition, some RTS games are partially observable and non-deterministic. In this paper we concentrate on perfect information RTS games.

Durative actions and simultaneous moves have been addressed in the past in the context of RTS games [Churchill *et al.*, 2012; Kovarsky and Buro, 2005; Saffidine *et al.*, 2012], but the large branching factor caused by independently acting objects remains a big challenge to be addressed. Recently studied methods, such as *Combinatorial Multi-Armed Bandits* [Ontañón, 2013; Shleyfman *et al.*, 2014] have started to tackle this problem but are still not sufficient to handle large-scale RTS games such as StarCraft. To have a sense of scale, the worst case branching factor of StarCraft has been estimated to be on the order of 10^{50} or higher [Ontanón *et al.*, 2013], which is staggering if we compare it with the relatively small branching factors of games like Chess (about 36) and Go (about 180 on average).

The approach presented in this paper directly addresses this problem by integrating HTN planning into game tree search. HTN planning allows us to create "domain configurable" planners that do not have to explore the whole combinatorics of the task at hand, but instead choose from a reduced set of methods to achieve each task. Our new AHTN approach represents a general hierarchical game tree search algorithm for adversarial games that can be configured for specific RTS games by providing a domain definition similar to standard HTN planners. We present the basic AHTN algorithm and then extend it to support durative, simultaneous, and concurrent actions to apply it to RTS games.

In the remainder of this paper, after discussing related work, we first introduce HTN planning. Then we present our AHTN algorithm, followed by extensions to apply it to RTS games, and an experimental evaluation in the μRTS game.

Related Work

The work presented in this paper is related to HTN-based planning systems for adversarial domains such as TIGNUM 2 [Smith *et al.*, 1998] for the game of Contract Bridge. TIGNUM 2 allowed for adversarial games by defining the tasks in HTN planning in such a way that they refer to moves of each player and then using a standard HTN planning algorithm. TIGNUM 2 also handled partial observability, which

we do not address in this paper. In the game of Go, adversarial HTN planners also have been proposed [Meijer and Koppelaar, 2001; Willmott *et al.*, 1999]. Meijer and Koppelaar used a standard HTN planner in which one player plans individually, and then passes the plan to the other player, who plans and may force the first player to backtrack (similarly to [Jonsson and Rovatsos, 2011]). This process is iterated until plans are found to achieve the goals of both players or until the search space is exhausted. An approach to hierarchical adversarial search in RTS games was presented in [Stanescu *et al.*, 2014] based on performing game tree search at two separate levels of abstraction instead of using HTN planning. Compared to these methods, our main contributions are: 1) a general adversarial HTN planning algorithm that is a direct application of HTN planning to game tree search (and thus can take advantage of standard optimizations such as alpha-beta pruning), and 2) extensions for durative and simultaneous actions to handle RTS games.

HTN Planning

HTN planning [Erol *et al.*, 1994] is a planning approach that creates plans by decomposing tasks into smaller and smaller tasks. In this context a *task* t is a symbolic representation of an activity in the world [Hogg *et al.*, 2010]. There are two types of tasks: *primitive tasks* correspond to actions that an agent can directly execute in the world. *Non-primitive tasks* represent goals that an agent might want to achieve and require finding a plan to achieve them before the agent can execute them.

For the purposes of this section, we will assume plans are totally-ordered [Ghallab *et al.*, 2004]. This assumption is lifted later in the paper. Under this assumption, a *method* $m = (t, C, w)$ is a tuple that contains a non-primitive task t, a set of preconditions C, and a sequence of tasks $w = [t_1, ..., t_n]$. Intuitively, a method m represents a way in which we can decompose t into a set of subtasks w if the set of preconditions C is satisfied.

A *hierarchical task network* (HTN) N is a tree, in which nodes are tasks (*task-nodes*) or methods (*method-nodes*). Each non-primitive task can only have one child, which must be a method. A method $m = (t, C, w)$ has one child for each of the tasks in w. Primitive tasks cannot have children. We write $leaves(N)$ to denote the set of leaves of HTN N. We say that a task-node t is *fully decomposed* when all the leaves in the sub-HTN rooted in t are primitive tasks.

Given a domain in which the world can be in one state $s \in \mathcal{S}$, and the agent can select actions from a set $\mathcal{A}$, we assume the existence of *state transition function* $\gamma : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S} \cup \{\bot\}$ that defines the effect of actions in the world. If an action a is not applicable in s, we define $\gamma(s, a) = \bot$. Given state $s \in \mathcal{S}$ and a fully decomposed HTN N, we can use γ to determine the effect of executing N on s. To do this, we just need to execute each of the leaves in N in their respective sequential order. Given HTN N and a non-primitive task $t \in leaves(N)$, we say that a method $m = (t, C, w)$ is *applicable* to t given N (we write $m \in applicable(N, t)$) if all the leaves in N that precede t are fully decomposed, and if all preconditions in C are satisfied in the state that results from executing all tasks in N that precede t. HTN N is *valid* if all methods in N are applicable to their parent tasks. The HTN that results from adding an applicable method m as the child of a non-primitive task t in HTN N is called a *decomposition* of N, and is denoted as $decomposition(N, t, m)$.

Algorithm 1 HTNPlanning(s_0, N_0)

```
1: loop
2:    if fullyDecomposed(N_0) ∧ γ(s_0, N_0) ≠ ⊥ then
3:       return N_0
4:    end if
5:    pick a non-primitive task t ∈ leaves(N_0)
6:    if applicable(N_0, t) = ∅ then
7:       Backtrack
8:    end if
9:    pick m ∈ applicable(N_0, t)
10:   N_0 = decomposition(N_0, t, m)
11: end loop
```

An *HTN planning problem* $P = (\mathcal{S}, \mathcal{A}, \gamma, \mathcal{T}, \mathcal{M}, s_0, N_0)$ is a tuple, where $\mathcal{S}$ is the set of possible states of the world, $\mathcal{A}$ is the finite set of actions that can be executed by the agent, γ is the transition function that defines the effects of each action in the world, $\mathcal{T}$ is a finite set of tasks, $\mathcal{M}$ is a finite set of methods, $s_0 \in \mathcal{S}$ is the initial state of the world, N_0 is an initial HTN that is used to encode the goals of the agent. The purpose of HTN planning is to find a valid full decomposition of the initial HTN N_0 given the initial state s_0.

Algorithm 1 shows a standard HTN planning algorithm, which works as follows: lines 2-4 just check whether the current HTN is fully decomposed, and if so return the solution. Line 5 selects a non-primitive task from the HTN that doesn't have a method yet, lines 6 to 9 select a method for the task, and line 10 adds the method to the HTN. The algorithm loops until the HTN is fully decomposed. If at any iteration M_t is empty, the algorithm backtracks (lines 7-9).

Adversarial HTN Planning

This section presents an adversarial HTN planning algorithm that we call AHTN. AHTN combines the minimax game tree search algorithm with the HTN planning Algorithm 1. The algorithm presented in this section assumes a turn-based, perfect information, deterministic zero-sum game and purely sequential methods (i.e., tasks are subdivided into a list of subtasks to be executed sequentially) for clarity of presentation. We relax these assumptions in the next section.

AHTN assumes that there are two players, max and min, and it searches a game tree (with maximum depth d) consisting of *min* and *max* nodes in a similar fashion as minimax game tree search, but with several key differences, which we describe below. Each node n in the game tree is annotated with a tuple (s, N_+, N_-, t_+, t_-), where s is the current state of the world, N_+ and N_- are the HTNs representing plans for player max and min, respectively, t_+ and t_- (where t_+ is a task in N_+ and t_- is a task in N_-) represent *execution pointers* that keep track of which parts of the HTN have already been executed. In the root of the game tree, $t_+ = \bot$ and $t_- = \bot$ indicate that no actions have been executed yet.

Algorithm 2 AHTNMax$(s, N_+, N_-, t_+, t_-, d)$

1: **if** $terminal(s) \vee d \leq 0$ **then**
2: **return** $(N_+, N_-, e(s))$
3: **end if**
4: **if** $nextAction(N_+, t_+) \neq \perp$ **then**
5: $t = nextAction(N_+, t_+)$
6: **return** AHTNMin$(\gamma(s,t), N_+, N_-, t, t_-, d-1)$
7: **end if**
8: $N_+^* = \perp, N_-^* = \perp, v^* = -\infty$
9: $\mathcal{N} = decompositions_+(s, N_+, N_-, t_+, t_-)$
10: **for all** $N \in \mathcal{N}$ **do**
11: $(N'_+, N'_-, v') =$ AHTNMax(s, N, N_-, t_+, t_-, d)
12: **if** $v' > v^*$ **then**
13: $N_+^* = N'_+, N_-^* = N'_-, v^* = v'$
14: **end if**
15: **end for**
16: **return** (N_+^*, N_-^*, v^*)

$nextAction(N, t)$ is a function that given HTN N and execution pointer t returns the next primitive task to be executed in N after t. If $t = \perp$, it returns the first primitive task to be executed in N. If N is still not fully decomposed, and no such primitive task yet exists in N, then $nextAction(N, t) = \perp$.

We say that a *max* node $n = (s, N_+, N_-, t_+, t_-)$ is *consistent* if the primitive actions that are already in N_+ and N_- can be executed given state s and transition function γ. Formally:

- $nextAction(N_+, t_+) = \perp$, or
- $s' = \gamma(s, nextAction(N_+, t_+)) \neq \perp$ and *min* node $n' = (s', N_+, N_-, nextAction(N_+, t_+), t_-)$ is consistent.

The definition of *min* node consistency is analogous.

For max node $n = (s, N_+, N_-, t_+, t_-)$, we let $decompositions_+(s, N_+, N_-, t_+, t_-)$ denote the set of its valid decompositions that add only one new method to N_+: $\{decomposition(N_+, t, m) \mid m \in applicable(N_+, t)\}$, where t to the first non-primitive task of N_+ (according to execution order). $decompositions_-$ is defined analogously.

Finally, we assume an evaluation function e that when applied to $s \in \mathcal{S}$ returns player $max's$ payoff in s if s is terminal or an approximation thereof if s is non-terminal.

Using these definitions, Algorithm 2 shows the AHTN algorithm for *max* nodes, function ATHFNMin is analogous. Algorithm AHTN returns a triple (N_+, N_-, v) with the best plans found for both players, and the result of the evaluation function e in the terminal node reached by executing these plans. Intuitively: **Lines 1–3** determine whether a terminal node or the maximum search depth d has been reached, in which case the current plans and evaluation are returned. **Lines 4–7** determine whether the next action to execute for player max is already in the current plan N_+, and in such case, execute it and yield to player min. **Line 8** initializes some variables to compute the best plan for player max. **Line 9** computes all possible decompositions that player max has for its current plan. **Lines 10–15** determine the decomposition resulting in the highest evaluation.

A key difference between the AHTN algorithm and standard minimax search is that recursive calls do not always alternate between *max* and *min*. Notice that the recursive call in line 14 is still to AHTNMax and not to AHTNMin, because the plan of player max might still not be sufficiently decomposed to produce an action. The algorithm only switches from *max* nodes to *min* nodes when the current plan can produce the next action (line 6). Lastly, alpha-beta pruning [Knuth and Moore, 1975] can be easily added to the algorithm to decrease the number of explored nodes. The experimental results reported in this paper use alpha-beta pruning.

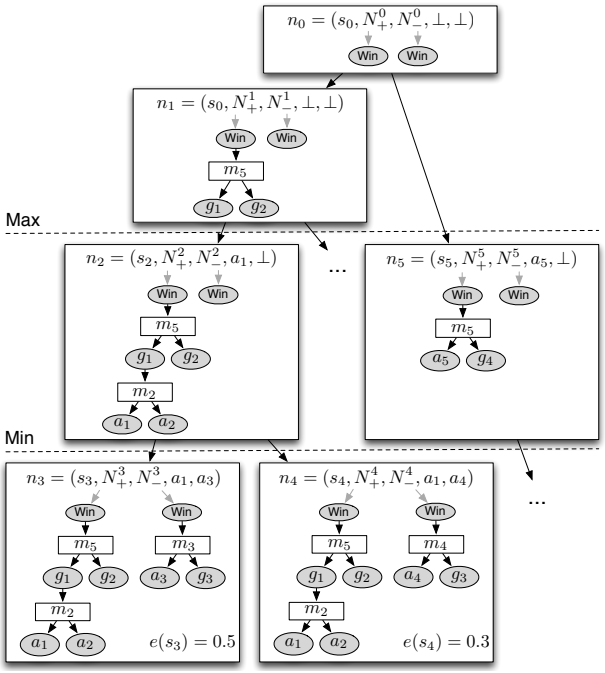

Figure 1: Illustration of a tree generated by the AHTN algorithm for depth 2.

Figure 1 illustrates the tree generated by Algorithm 2 for depth $d = 2$. HTNs for each player in each of the game tree nodes are shown. For example, we can see that in the root node n_0, the HTNs for both *max* and *min* contain a single non-primitive task *Win* that needs to be decomposed. There are two decompositions that *max* can apply to its HTN, resulting in nodes n_1 and n_5. The decomposition leading to n_1 does not yet result in any primitive action, and, thus, n_1 is still a *max* node. Once player *max* has decomposed the HTN to the point where the first action can be generated (nodes n_2 and n_5, it is *min*'s turn to decompose his HTN. Finally, once *min* can generate its action, the maximum depth is reached (nodes n_3 and n_4) and evaluation function e is applied to the corresponding game states to determine the value of the leaves. Moreover, the game state in nodes n_0 and n_1 is the same (s_0) because no action was generated by *max* as the result of the expansion that led from n_0 to n_1. This example motivates the fact that the number of leaves of the tree explored by AHTN is bounded above by the product of the number of possible HTN decompositions available to each player, which is expected to be much smaller than the number of raw sequences of actions.

Algorithm 3 AHTNCD(s, N_+, N_-, t_+, t_-, d)

1: $s' = simulateTillNextChoicePoint(s)$.
2: **if** $terminal(s') \vee d \leq 0$ **then**
3: **return** $(N_+, N_-, e(s'))$
4: **end if**
5: **if** $canIssueActions(+, s')$ **then**
6: **return** AHTNMaxCD($s', N_+, N_-, t_+, t_-, d$)
7: **end if**
8: **return** AHTNMinCD($s', N_+, N_-, t_+, t_-, d$)

Algorithm 4 AHTNMaxCD(s, N_+, N_-, t_+, t_-, d)

1: **if** $nextAction(N_+, t_+) \neq \perp$ **then**
2: $t = nextAction(N_+, t_+)$
3: **return** AHTNCD($\gamma(s, t), N_+, N_-, t, t_-, d-1$)
4: **end if**
5: $N^*_+ = \perp, N^*_- = \perp, v^* = -\infty$
6: $\mathcal{N} = decompositions_+(s, N_+, N_-, t_+, t_-)$
7: **for all** $N \in \mathcal{N}$ **do**
8: $(N'_+, N'_-, v') =$ AHTNCD(s, N, N_-, t_+, t_-, d)
9: **if** $v' > v^*$ **then**
10: $N^*_+ = N'_+, N^*_- = N'_-, v^* = v'$
11: **end if**
12: **end for**
13: **return** (N^*_+, N^*_-, v^*)

Problem Definitions for AHTN

Assuming the HTN planning problem definition used for the execution of AHTN is incomplete, and at a given point in the search, one player, say *min*, does not have any available decomposition (i.e., $\mathcal{N} = \emptyset$ in line 9 of the algorithm), the tree will be pruned, and even if *max*'s plan would have led to a very good state, this will not be taken into account during the search because of the incompleteness of the problem definition being used. For this reason, in order for AHTN to function as expected, the problem definition must satisfy the condition that for any game state $s \in S$ and for any task $t \in \mathcal{T}$, there must always be at least one method $m \in \mathcal{M}$ that is applicable to t in state s. In practice, this is easily achieved in RTS games, by having an extra method for each task defined in the following way: the precondition that is exactly the negation of all the preconditions of all the other methods for defined for t; the decomposition performs a *wait* action (for a fixed number of game cycles), which is always executable in RTS games, followed by a recursive call to t.

Extensions

Algorithm 2 assumes a turn-based game. To apply the framework to RTS games, we need the following extensions.

Durative and Simultaneous Actions

To allow simultaneous actions, we employ the same approach proposed in the ABCD algorithm (alpha-beta considering durations) [Churchill *et al.*, 2012]. Specifically, by checking at the beginning of the algorithm whether any of the two players is ready to issue an action ($canIssueActions(p, s)$, where p is a player) in the current game state. If only one player is ready, then everything proceeds normally. If both players are ready to execute an action, then one is picked (e.g., *max*) and the algorithm proceeds normally. Notice that this approach is a simplification to avoid having to generate the complete matrix game that results from all possible action pairs players can execute.

To allow durative actions, the first lines of the algorithm need to be modified in the following way: when both players are currently executing actions the game simply needs to be simulated until a point when one of the two player actions finish and at least one player needs to issue another action (*simulateTillNextChoicePoint* function). The AHTN algorithm resulting from adding durative and simultaneous action support is shown in Algorithms 3 and 4. As a result of this change, if a player is executing a long-lasting action, the other player might execute multiple short actions in parallel. Notice that because of durative and simultaneous actions, the *depth* of the tree (measured in number of actions issued) might now be different than the number of game cycles, since there might be cycles when all units are already executing actions.

Concurrent Actions

In RTS games, players control multiple units to which they can issue actions in parallel, i.e., while one unit is executing an action, the player can issue actions to the other units. In order to allow for concurrent actions, the algorithm needs to be modified as follows: First, we need to extend the HTN formalism to allow for concurrent method decompositions by adding the possibility of parallel decompositions $w = \{t_1, ..., t_n\}$ for methods $m = (t, C, w)$. Second, the transition function γ needs to be redefined as $\gamma : \mathcal{S} \times 2^{\mathcal{A}} \rightarrow \mathcal{S} \cup \{\perp\}$, so that sets of actions can be issued simultaneously. Third, the execution pointers t_+ and t_- must now represent sets of actions rather than individual actions, and the $nextAction$ method needs to be modified to accept and return sets of actions, because now more than one action might be next to execute concurrently. This has to be reflected in lines 5 and 6 of Algorithm 2, and lines 2 and 3 of Algorithm 4. The rest of the algorithm remains unchanged.

Finally, we note that other (non-adversarial) approaches to HTN planning with durative and concurrent actions exist [Goldman, 2006; Ghallab *et al.*, 2004].

Variables

The HTN formalization used in Algorithm 2 is rather simplistic. In practice, a richer representation allowing the use of variables in the definition of tasks and methods is useful. In such representations, when decomposing a task t with a method m, all different variable instantiations of m would be returned as different decompositions in line 9 of the algorithm. The rest of the algorithm remains unchanged.

Experimental Results

We evaluated the performance of AHTN using the free-software μRTS (https://github.com/santiontanon/microrts), which has been used by several researchers to validate new algorithms for RTS games [Ontañón, 2013; Shleyfman *et al.*, 2014]. Figure 2 shows a screenshot of a μRTS game, in which two players (blue and red) compete to destroy each

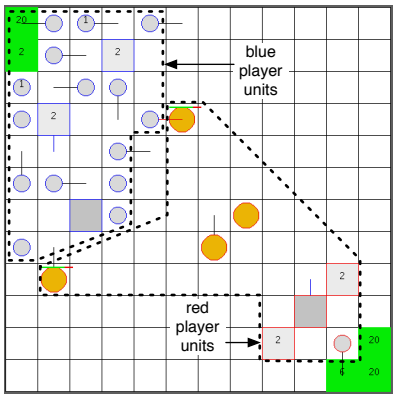

Figure 2: A screenshot of a μRTS game state. Green squares are resources that can be harvested, small grey circles are workers, larger yellow circles are melee units, and grey squares are bases or barracks.

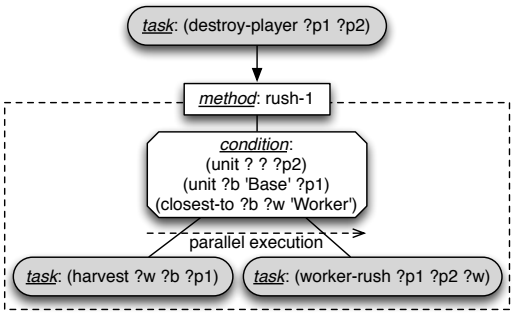

Figure 3: A sample method from the AHTN-F definition stating that a way to destroy player $p2$, (the (unit ? ? ?$p2$) condition checks that player $p2$ has at least one unit) if we have a base and a worker, is to 1) "harvest" resources, and 2) perform a "worker rush" with the remaining units.

other's units. μRTS is a minimalistic yet complex real-time game environment that captures several defining features of full-fledged RTS video games: durative and simultaneous actions, large branching factors, resource allocation, and real-time combat. μRTS games are fully observable and deterministic. To test AHTN's performance, we crafted five different AHTN definitions for the μRTS domain which work as follows:

Low Level (AHTN-LL) contains just the low-level primitive actions available in the game without any further non-primitive tasks. The actions units can execute in μRTS are: move (in any of the 4 cardinal directions), attack (any enemy unit within range), some units (bases, workers, and barracks) can produce new units (in any of the 4 cardinal directions), harvest minerals, return minerals to a base, or stay idle. Thus, game trees traversed by AHTN-LL are identical to those searched by standard minimax search applied to raw low-level actions. This definition contains a total of 11 methods for 3 different tasks. **Low Level with Pathfinding (AHTN-LLPF)** is similar to Low Level, but instead of moving step by step, the move action is defined as taking a target as parameter (the A* algorithm is used to find a shortest path). This definition contains a total of 12 methods for 4 different tasks. **Portfolio (AHTN-P),** in which the main task of the game can be achieved only by three non-primitive tasks (three rushes with three different unit types in the game) that encode three different hard-coded strategies to play the game. Thus, the game tree only has one *max/min* choice layer at the top, followed by scripted action sequences. This definition contains a total of 76 methods for 28 different tasks. **Flexible (AHTN-F)** which is a more elaborate AHTN for μRTS, with non-primitive tasks for harvesting resources, training units of different types, and attacking the enemy. An example method from this definition is shown in Figure 3. This definition contains 49 methods for 19 different tasks. **Flexible Single Target (AHTN-FST)** which is similar to Flexible, but encoded in such a way that all units that are sent to attack are sent to attack the same target. This drastically reduces the branching factor because Flexible considers all the combinatorics of sending each unit to attack every possible target. This definition contains a total of 51 methods for 21 different tasks.

Our AHTN framework is flexible enough to generalize standard minimax game tree search (AHTN-LL) and portfolio search (AHTN-P), in which the only decision is which strategy out of a predefined set of strategies to deploy for each player. To evaluate AHTN, we compared it against the following collection of players:

RandomBiased is a random player with action probability biased toward attacking, if possible. **LightRush** is a hardcoded rush strategy based on producing *light* units and sending them to attack immediately. **RangedRush** is identical to LightRush, except for producing *ranged* units. **WorkerRush** is a hardcoded rush strategy that constantly sends workers to attack. **ABCD** (Alpha-Beta Considering Durations, [Churchill *et al.*, 2012]) is a version of alpha-beta search designed to handle domains with simultaneous and durative actions that uses a playout policy in the leaves of the tree to evaluate the states (in our version we did not use scripted move ordering, as in the original ABCD). **MonteCarlo** is standard Monte Carlo search which performs as many random playouts as possible for each of the actions available, and returns the one achieving the highest average evaluation. **UCT** is a standard implementation of UCT [Kocsis and Szepesvri, 2006], extended with the same modifications as ABCD to handle domains with simultaneous and durative actions. **NaiveMCTS** is a Monte Carlo Tree Search algorithm specifically designed to games with large branching factors, such as RTS games [Ontañón, 2013].

We gave every player a budget of 200 playouts per game frame to decide on actions (but players can split the search process over multiple frames; for example, if the game state does not change during 10 game frames before a player needs to issue an action, then such player has had time to perform 2000 playouts to decide for such action). All playouts were limited to 100 game frames using RandomBiased as playout policy, after which an evaluation function is used to determine the value of the final state reached in the playout. ABCD's playout policy was chosen to be WorkerRush because in our experiments, ABCD worked better with deterministic play-

Table 1: Average score for each player.

Player	Total	Maps M1	M2	M3	Player	Total	Maps M1	M2	M3
RndBiased	.20	.19	.17	.23	ABCD	.36	.46	.38	.24
LightRush	.52	.35	.61	.61	MonteCarlo	.52	.68	.51	.39
RangedRush	.43	.28	.47	.53	UCT	.37	.53	.35	.24
WorkerRush	.64	.69	.60	.73	NaiveMCTS	.56	.59	.63	.47
AHTN-LL	.11	.12	.08	.14	AHTN-F	**.90**	**.87**	**.90**	**.92**
AHTN-LLPF	.32	.33	.27	.35	AHTN-FST	.79	.80	.77	.81
AHTN-P	.77	.71	.75	.84					

Table 2: Game tree statistics for AHTN players: average value and the maximum (in parenthesis)

	d	Tree Leaves	Lookahead
LL	1.71 (6)	3065.66 (12020)	13.71 (40)
LLPF	2.44 (28)	3053.03 (9415)	37.20 (1001)
P	23.03 (123)	6.12 (8)	183.29 (1111)
F	4.58 (13)	1254.89 (11140)	48.50 (210)
FST	5.83 (20)	474.82 (2355)	58.52 (200)

out policies (since it does a single playout in each leaf of the tree). For the players that require an evaluation function we use a function that — inspired by LTD2 [Churchill and Buro, 2013] — computes the sum of the cost of each friendly unit multiplied by the square root of their hitpoints, minus the corresponding sum for the enemy units.

The implementation of the AHTN algorithm that we used employs alpha-beta search, and, in order to improve the evaluation of game states, it performs one single playout at each terminal leave, before applying the evaluation function (exactly in the same way as ABCD does).

To evaluate each algorithm we played a round-robin tournament, in which each algorithm played 20 games (with various starting positions) against each other in each of 3 different maps, making a total of 60 games per pair of algorithms ($13 \times 13 \times 3 \times 20 = 10,140$ games in total). To compute a score, we reward wining with 1 point and tying with 0.5 points. If a game reaches 3000 game cycles, it is considered a tie. The three maps we used for our experimentation are: M1 (8×8 tiles), M2 (12×12 tiles), and M3 (16×16 tiles). All players start with one base and one worker.

Table 1 shows the results we obtained. The "Total" column lists the average score obtained by each player, showing that the best scoring players are all AHTN players. Specifically, the best performing player is AHTN-F (0.90), followed by AHTN-FST, and AHTN-P. AHTN outperforms state of the art MCTS algorithms such as UCT, or NaiveMCTS. This is remarkable, because NaiveMCTS uses specialized Monte Carlo search designed for RTS games. The map specific scores indicate that the performance of non-AHTN methods (ABCD, MonteCarlo, UCT, and NaiveMCTS) deteriorates as the size of the map grows. This is expected, because more units are produced in games played on larger maps which increases the branching factor and thus reduces the search depth. For example, ABCD achieves a score of 0.46 on the 8×8 map, but only 0.24 on the 16×16 map. Some of the scripted methods seem to perform better on larger maps (e.g., LightRush), but this is in fact caused by ABCD, MonteCarlo, UCT and NaiveMCTS under-performing on larger maps.

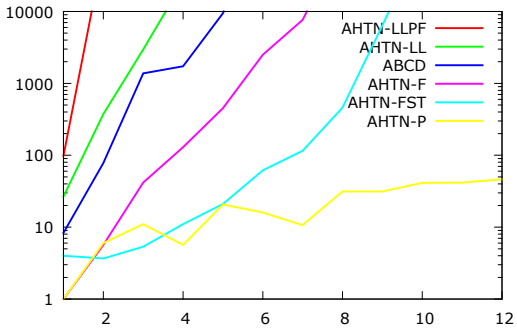

Figure 4: Time required (vertical axis, in milliseconds) to reach a certain depth for the algorithms listed in Table 2.

To gain more insight into the performance of AHTN, we analyzed the game trees the various AHTN players generate when given a fixed time budget. Table 2 shows the average depth (d) the players using iterative deepening had time to reach, the number of tree leaves, and the *lookahead* which is the average difference measured in game time between the game state at the root of the tree and the game state at the leaves of the tree. Notice that depth is measured in number of actions issued, and lookahead in number of game cycles, which might not be the same, since there are durative actions. To generate these statistics we made each of the players in Table 2 play one game against all other players on all maps giving them 100 ms of time per game cycle.

Finally, Figure 4 shows the average time that the different AHTN players and ABCD require to reach a certain search depth (defined as number of primitive actions produced in sequence). AHTN-P is the method whose time grows slowest because there is only one decision point at the beginning. The time required by AHTN-LL and AHTN-LLPF grows the fastest (AHTN-LL and ABCD explore the full game search space). AHTN-LLPF is slower than AHTN-LL because the actions considered by AHTN-LLPF are higher-level than those considered by AHTN-LL (although the branching factor can also be higher). Also, we can see that even if the search space explored by AHTN-LL is equivalent to that explored by ABCD, ABCD is faster because it is a specialized algorithm, whereas AHTN-LL pays the extra computational cost of the AHTN planning machinery. In summary, AHTN can be "configured" via different HTN domain definitions that represent different tradeoffs between search-space size, completeness, and computation time.

Conclusions and Future Work

Large branching factors caused by independent action selection have held back search-based AI systems for complex adversarial real-time decision domains, such as real-time strategy video games. The AHTN algorithm we introduced in this paper addresses this problem by combining the idea of hierarchical task decomposition with minimax search. The positive results we obtained when comparing AHTN with numerous state-of-the-art methods in a minimalistic RTS game leads us to believe that AHTN has the potential to tackle more complex decision problems involving concurrent actions and

long-range economic planning, subject to the availability of effective task decompositions.

There are multiple directions future research can expand on AHTN's ability to significantly decrease branching factors and to look ahead much farther than traditional adversarial search methods that optimize raw action sequences. For instance, existing scripted players for popular RTS games such as StarCraft can be combined using AHTN to improve their playing strength. Also, with thousands of RTS game replays available, automatically extracting HTNs that mimic human expert play at various strategic levels seems possible and could enable AHTN-based AI systems to defeat human experts who excel at long-range planning.

References

[Browne *et al.*, 2012] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

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

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

[Churchill *et al.*, 2012] D. Churchill, A. Saffidine, and M. Buro. Fast heuristic search for RTS game combat scenarios. In *Proceedings of AIIDE*. The AAAI Press, 2012.

[Erol *et al.*, 1994] Kutluhan Erol, James A Hendler, and Dana S Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *AIPS*, volume 94, pages 249–254, 1994.

[Ghallab *et al.*, 2004] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: theory & practice*. Elsevier, 2004.

[Goldman, 2006] Robert P Goldman. Durative planning in htns. In *ICAPS*, pages 382–385, 2006.

[Hogg *et al.*, 2010] Chad Hogg, Ugur Kuter, and Hector Muñoz-Avila. Learning methods to generate good plans: Integrating HTN learning and reinforcement learning. In *Proceedings of AAAI*, 2010.

[Jonsson and Rovatsos, 2011] Anders Jonsson and Michael Rovatsos. Scaling up multiagent planning: A best-response approach. In *Proceedings of ICAPS*, 2011.

[Knuth and Moore, 1975] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[Kocsis and Szepesvri, 2006] Levente Kocsis and Csaba Szepesvri. Bandit based Monte Carlo planning. In *Proceedings of ECML 2006*, pages 282–293. Springer, 2006.

[Kovarsky and Buro, 2005] Alexander Kovarsky and Michael Buro. Heuristic search applied to abstract combat games. In *Canadian Conference on AI*, pages 66–78, 2005.

[Meijer and Koppelaar, 2001] AB Meijer and H Koppelaar. Pursuing abstract goals in the game of Go. *Belgium-Netherlands Conference on AI (BNAIC)*, pages 415–422, 2001.

[Ontanón *et al.*, 2013] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. 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, 2013.

[Ontañón, 2013] Santiago Ontañón. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Proceedings of AIIDE*, 2013.

[Saffidine *et al.*, 2012] Abdallah Saffidine, Hilmar Finnsson, and Michael Buro. Alpha-beta pruning for games with simultaneous moves. In *Proceedings of AAAI*, Toronto, Canada, 2012. AAAI Press.

[Shleyfman *et al.*, 2014] Alexander Shleyfman, Antonín Komenda, and Carmel Domshlak. On combinatorial actions and CMABs with linear side information. In *Proceedings of ECML 2014*. Springer, 2014.

[Smith *et al.*, 1998] Stephen J Smith, Dana Nau, and Tom Throop. Computer Bridge: A big win for AI planning. *AI Magazine*, 19(2):93, 1998.

[Stanescu *et al.*, 2014] Marius Stanescu, Nicolas A Barriga, and Michael Buro. Hierarchical adversarial search applied to real-time strategy games. In *Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2014.

[Willmott *et al.*, 1999] Steven Willmott, Julian Richardson, Alan Bundy, and John Levine. An adversarial planning approach to Go. In *Computers and Games Conference*, pages 93–112. Springer, 1999.