ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/313561586

Game Tree Search Based on Non-Deterministic Action Scripts in Real-Time Strategy Games

Article *in* IEEE Transactions on Computational Intelligence and AI in Games · June 2017

DOI: 10.1109/TCIAIG.2017.2717902

CITATIONS
0

READS
109

3 authors, including:

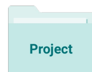
Nicolas A. Barriga
University of Alberta
18 PUBLICATIONS **54** CITATIONS

SEE PROFILE

Marius Stanescu
University of Alberta
14 PUBLICATIONS **32** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



Deep Learning for Real-Time Strategy Games (StarCraft) View project

All content following this page was uploaded by Nicolas A. Barriga on 24 June 2017.

The user has requested enhancement of the downloaded file. All in-text references underlined in blue are added to the original document and are linked to publications on ResearchGate, letting you access and read them immediately.

Game Tree Search Based on Non-Deterministic Action Scripts in Real-Time Strategy Games

Nicolas A. Barriga, Marius Stanescu and Michael Buro

***Abstract*—Significant progress has been made in recent years towards stronger Real-Time Strategy (RTS) game playing agents. Some of the latest approaches have focused on enhancing standard game tree search techniques with a smart sampling of the search space, or on directly reducing this search space. However, experiments have thus far only been performed using small scenarios. We provide experimental results on the performance of these agents on increasingly larger scenarios. Our main contribution is *Puppet Search*, a new adversarial search framework that reduces the search space by using scripts that can expose choice points to a look-ahead search procedure. Selecting a combination of a script and decisions for its choice points represents an abstract move to be applied next. Such moves can be directly executed in the actual game, or in an abstract representation of the game state which can be used by an adversarial tree search algorithm. We tested *Puppet Search* in μRTS, an abstract RTS game popular within the research community, allowing us to directly compare our algorithm against state-of-the-art agents published in the last few years. We show a similar performance to other scripted and search based agents on smaller scenarios, while outperforming them on larger ones.**

***Index Terms*—Real-Time Strategy (RTS) Games, Adversarial Search, Heuristic Search, Monte-Carlo Tree Search**

I. Introduction

In the past 20 years AI systems for abstract games such as Backgammon, Checkers, Chess, Othello, and Go have become much stronger and are now able to defeat even the best human players. Video games introduce a host of new challenges not common to abstract games. Actions in video games can usually be issued simultaneously by all players multiple times each second. Moreover, action effects are often stochastic and delayed, players only have a partial view of the game state, and the size of the playing area, the number of units available and of possible actions at any given time are several orders of magnitude larger than in most abstract games.

The rise of professional *eSports* communities enables us to seriously engage in the development of competitive AI for video games. A game played just by amateurs could look intriguingly difficult at first glance, but top players might be easily defeated by standard game AI techniques. An example of this is Arimaa [1], which was purposely designed to be difficult for computers but easy for human players. It took a decade for game playing programs to defeat the top human players, but no new AI technique was required other than those already in use in Chess and Go programs. And after a decade we still don't know if the computer players are really strong at Arimaa or no human player has a truly deep understanding of the game. In this respect a game with a large professional player base provides a more solid challenge.

One particularly interesting and popular video game class is Real-Time Strategy (RTS) games, which are real-time war simulations in which players instruct units to gather resources, build structures and armies, seek out new resource locations and opponent positions, and destroy all opponent's buildings to win the game. Typical RTS games also feature the so-called "fog of war" (which restricts player vision to vicinities of friendly units), large game maps, possibly hundreds of mobile units that have to be orchestrated, and fast-paced game play allowing players to issue multiple commands to their units per second. Popular RTS games, such as StarCraft and Company of Heroes, constitute a multi billion dollar market and are played competitively by thousands of players.

Despite the similarities between RTS games and abstract strategy games like Chess and Go, there is a big gap in state- and action-space complexity [2] that has to be overcome if we are to successfully apply traditional AI techniques such as game tree search and solving (small) imperfect information games. These challenges, in addition to good human players still outperforming the best AI systems, make RTS games a fruitful AI research target, with applications to many other complex decision domains featuring large action spaces, imperfect information, and simultaneous real-time action execution.

To evaluate the state of the art in RTS game AI, several competitions are organized yearly, such as the ones organized at the AIIDE[1] and CIG conferences, and SSCAI[2]. Even though bot competitions show small incremental improvements every year, strong human players continue to defeat the best bots with ease at the annual man-machine matches organized alongside AIIDE. When analyzing the games, several reasons for this playing strength gap can be identified. Good human players have knowledge about strong game openings and playing preferences of opponents they encountered before. They can also quickly identify and exploit non-optimal opponent behaviour, and — crucially — they are able to generate robust long-term plans, starting with multi-purpose build orders in the opening phase. RTS game AI systems, on the other hand, are still mostly scripted, have only modest opponent modeling abilities, and generally don't seem to be able to adapt to unforeseen circumstances well. In games with small branching

N. Barriga, M. Stanescu and M. Buro are with the Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, T6G 2E8 (e-mail: {barriga|astanesc|mburo}@ualberta.ca)

[1] http://starcraftaicompetition.com
[2] http://sscaitournament.com/

factors a successful approach to overcome these issues is look-ahead search, i.e. simulating the effects of action sequences and choosing those that maximize the agent's utility. In this paper we present and evaluate an approach that mimics this process in video games featuring vast search spaces. Our algorithm uses scripts that expose choice points to the look-ahead search in order to reduce the number of action choices. This work builds on the *Puppet Search* algorithm proposed in [3]. Here we introduce new action scripts and search regimens, along with exploring different game tree search algorithms.

In the following sections we first describe the proposed algorithm in detail, then show experimental results obtained from matches played against other state-of-the-art RTS game agents, and finish the paper with discussing related work, our conclusions and motivating future work in this area.

II. Algorithm Details

Our new search framework is called *Puppet Search*. At its core it is an action abstraction mechanism that, given a non-deterministic strategy, works by constantly selecting action choices that dictate how to continue the game based on look-ahead search results. Non-deterministic strategies are described by scripts that have to be able to handle all aspects of the game and may expose choice points to a search algorithm the user specifies. Such choice points mark locations in the script where alternative actions are to be considered during search, very much like non-deterministic automata that are free to execute any action listed in the transition relation. So, in a sense, *Puppet Search* works like a puppeteer who controls the limbs (choice points) of a set of puppets (scripts).

More formally, we can think of applying the *Puppet Search* idea to a game as 1) creating a new game in which move options are restricted by replacing original move choices with potentially far fewer choice points exposed by a non-deterministic script, and 2) applying an AI technique of our choice to the transformed game to find or approximate optimal moves. The method, such as look-ahead search or finding Nash-equilibria by solving linear programs, will depend on characteristics of the new game, such as being a perfect or imperfect information game, or a zero sum or general sum game.

Because we control the number of choice points in this process, we can tailor the resulting AI systems to meet given search time constraints. For instance, suppose we are interested in creating a fast reactive system for combat in an RTS game. In this case we will allow scripts to expose only a few carefully chosen choice points, if at all, resulting in fast searches that may sometimes miss optimal moves, but generally produce acceptable action sequences quickly. Note that scripts exposing only a few choice points or none do not necessarily produce mediocre actions because script computations can themselves be based on (local) search or other forms of optimization. If more time is available for computing moves, our search can visit more choice points to generate better moves.

The idea of scripts exposing choice points originated from witnessing poor performance of scripted RTS AI systems and realizing that one possible improvement is to let look-ahead search make fundamental decisions based on evaluating the impact of chosen action sequences. Currently, RTS game AI systems still rely on scripted high level strategies [4], which, for example, may contain code that checks whether now is a good time to launch an all-in attack based on some state feature values. However, designing code that can accurately predict the winner of such an assault is challenging, and comparable to deciding whether there is a mate in k moves in Chess using static rules. In terms of code complexity and accuracy it is much more preferable to use look-ahead search to decide the issue, assuming sufficient computational resources are available. Likewise, given that currently high-level strategies are still mostly scripted, letting search decide which script choice point actions to take in complex video games has the potential to improve decision quality considerably while simplifying code complexity.

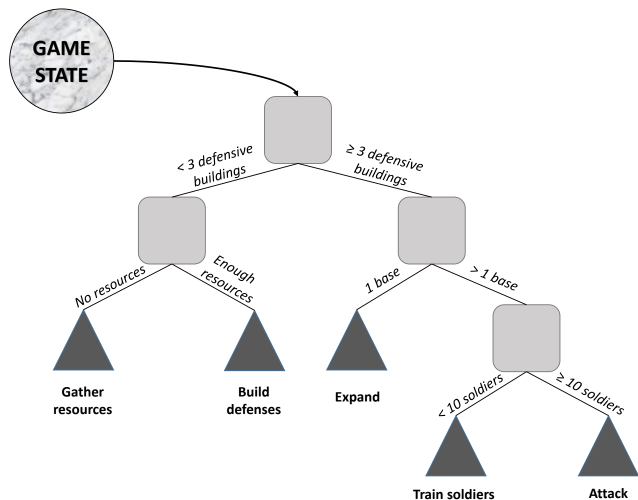


Fig. 1. Decision tree representing script choices.

A. Scripts

For our purposes we define a script as a function that takes a game state and returns a set of actions to be performed now. The method for generating actions is immaterial: it could be a rule based player, hand coded with expert knowledge, a machine learning or search based agent, etc. The only requirement is that the method must be able to generate actions for any legal game state. As an example consider a "rush", which is a common strategy in RTS games that tries to build as many combat units as fast as possible in an effort to destroy the opponent's base before suitable defenses can be built. A wide range of these aggressive attacks are possible. At one extreme end, the fastest attack can be executed using only workers, which usually deal very little damage and barely have any armor. Alternatively, the attack can be delayed until more powerful units become available.

Figure 1 shows a decision tree representing a script that first gathers resources, builds some defensive buildings, expands to a second resource location, creates soldiers and finally attacks the enemy. This decision tree is executed at every game simulation frame to decide what actions to issue next.

B. Choice Points

When writing a script, we must make some potentially hard choices, such as when to expand to create a new base. After training a certain number of workers, or maybe only after most of the current bases' resources are depleted. Regardless of the decision, it will be hardcoded in the script, according to a set of static rules about the state of the game. Discovering predictable patterns in the way the AI acts might be frustrating for all but beginner players. Whether the behavior implemented is sensible or not in the given situation, opponents will quickly learn to exploit it and the game will likely lose some of its replay value in the process. As script writers, we would like to be able to leave some choices open, such as which units to rush with. But the script also needs to deal with any and all possible events happening during the strategy execution. The base might be attacked before it is ready to launch its own attack, or maybe the base is undefended while our infantry units are out looking for the enemy. Should they continue in hope of destroying their base before they raze ours? Or should they come back to defend? What if when we arrive at the enemy's base we realize we don't have the strength to destroy it? Should we push on nonetheless? Some, or all, of these decisions are best left open, so that they can be explored dynamically and the most appropriate choice taken during the game. We call such non-deterministic script parts choice points.

C. Using Non-Deterministic Scripts

Scripts with choice points can transform a given complex game with a large action space into a smaller games to which standard solution methods can be applied, such as learning policies using machine learning (ML) techniques, MiniMax or Monte Carlo Tree Search, or approximating Nash-equilibria in imperfect information games using linear programming or iterative methods. The number of choice points and the available choices are configurable parameters that determine the strength and speed of the resulting AI system. Fewer options will produce faster but more predictable AI systems which are suitable for beginner players, while increasing their number will lead to stronger players, at the expense of increased computational work.

ML-based agents rely on a function that takes the current game state as input, and produces a decision for each choice in the script. The parameters of that function would then be optimized either by supervised learning methods on a set of game traces, or by reinforcement learning via self-play. Once the parameters are learned, the model acts like a static (but possibly stochastic) rule based system. If the system is allowed to keep learning after the game has shipped, then there are no guarantees as to how it will evolve, possibly leading to unwanted behavior. Another approach, look-ahead search, involves executing action sequences and evaluating their outcomes. Both methods can work well. It is possible to have an unbeatable ML player if the features and training data are good enough, as well as a perfect search based player if we explore the full search space. In practice, neither requirement is easy to meet: good representations are hard to design, and time constraints prevent covering the search space in most games. Good practical results are often achieved by combining both approaches [5].

Algorithm 1 Puppet ABCD Search

```
1:  procedure PUPPETABCD(state, height, prevMove, α, β)
2:     player ← PLAYERTOMOVE(state, policy, height)
3:     if heigth == 0 or TERMINAL(state) then
4:        return EVALUATE(state, player)
5:     end if
6:     for move in GETCHOICES(state, player) do
7:        if prevMove == ∅ then
8:           v ← PUPPETABCD(state, h − 1, move, α, β)
9:        else
10:          state′ ←COPY(state)
11:          EXECCHOICES(state′, prevMove, move)
12:          v ← PUPPETABCD(state′, height − 1, ∅, α, β)
13:       end if
14:       if player = MAX and v > α then α ← v
15:       if player = MIN and v < β then β ← v
16:       if α ≥ β then break
17:    end for
18:    return player == MAX ? α : β
19: end procedure
20:
21: #Example call:
22: #state: current game state
23: #depth: maximum search depth, has to be even
24: value = PUPPETABCD(state, depth, ∅, −∞, ∞)
```

D. Game Tree Search

To decide which choice to take, we can execute a script for a given timespan, look at the resulting state, and then backtrack to the original state to try other action choices, taking into account that the opponent also has choices. To keep the implementation as general as possible, we will use no explicit opponent model. We'll assume he uses the same scripts and evaluates states the same way we do.

Algorithm 1 shows a variant of *Puppet Search* which is based on ABCD (Alpha-Beta Considering Durations) search, that itself is an adaptation of alpha-beta search to games with simultaneous and durative actions [6]. To reduce the computational complexity of solving multi-step simultaneous move games, ABCD search implements approximations based on move serialization policies which specify the player which is to move next (line 2) and the opponent thereafter. Commonly used strategies include random, alternating, and alternating in 1-2-2-1 fashion, to even out first or second player advantages.

To fit into the *Puppet Search* framework for our hypothetical simultaneous move game we modified ABCD search so that it considers puppet move sequences — series of script and choice combinations — and takes into account that at any point in time both players execute a puppet move, to perform actions at every frame in the game. The maximum search depth is assumed to be even, which allows both players to select a puppet move to forward the world in line 11.

Moves for the current player are generated in line 6. They contain choice point decisions as well as the player whose move it is. Afterwards, if no move was passed from the previous recursive call (line 7), the current player's move $previousMove$ is passed on to a subsequent PUPPETABCD call at line 8. Otherwise, both players' moves are applied to the state (line 11).

RTS games are usually fast paced. In StarCraft bot competitions, for example rules allow up to 41 milliseconds computing time per simulation frame. However, as actions take some time to complete, it is often the case that the game doesn't really change between one game frame and the next one. This means that an agent's computations can be split among several consecutive frames until an action has to be issued. To take advantage of this split computation, the recursive Algorithm 1 has to be transformed into an iterative one, by manually managing the call stack. The details of this implementation are beyond the scope of this paper, and can be reviewed in our published code at the μRTS repository[3].

Alpha-beta search variants conduct depth-first search, in which the search depth needs to be predefined, and no solution is returned until the algorithm runs to completion. In most adversarial video games, the game proceeds even when no actions are emitted, resulting in a disadvantage if a player is not able to issue actions in a predefined time. Algorithms to play such games have to be *anytime* algorithms, that is, be able to return a valid solution even when interrupted before completion. Iterative deepening has traditionally been used to transform alpha-beta search into an anytime algorithm. It works by calling the search procedure multiple times, increasing the search depth in each call. The time lost due to repeating computations is usually very low, due to the exponential growth of game trees. However, even this minimal loss can be offset by subsequently reusing the information gained in previous iterations for move sorting.

In our implementation we reuse previous information in two ways: a *move cache* and a *hash move*. The *move cache* is a map from a state and a pair of moves (one for each player) to the resulting state. With this cache, we can greatly reduce the time it takes to forward the world, which is the slowest operation in our search algorithm. The *hash move* is a standard move ordering technique, in which the best move from a previous iteration is used first, in order to increase the chances of an earlier beta cut. Algorithm 2 shows an UCT version of *Puppet Search*. The node structure on line 1 contains the game state, the player to move (moves are being serialized as in ABCD), a list of the children nodes already expanded, a list of legal moves (applicable choice point combinations), and a previous move, which can be empty for the first player.

Algorithm 2 Puppet UCTCD Search

```
1:  Structure Node
2:      state
3:      player
4:      children
5:      moves
6:      previousMove
7:  End Structure
8:
9:  procedure PUPPETUCTCD(state)
10:     root ← NODE(state)
11:     while not timeUp do
12:         leaf ← SELECTLEAF(root)
13:         newState ←
14:         SIMULATE(leaf.state, policy, policy, leaf.parent.player,
            leaf.player, EVAL_PLAYOUT_TIME)
15:         e ← EVALUATE(newState, leaf.player)
16:         UPDATE(leaf, e)
17:     end while
18:     return GETBEST(root)
19: end procedure
20:
21: procedure SELECTLEAF(root)
22:     if SIZE(root.children) < SIZE(root.moves) then
23:         move ← NEXT(root.moves)
24:         if root.previousMove == ∅ then
25:             node ← NODE(root,move)
26:             APPEND(root.children, node)
27:             return SELECTLEAF(node)
28:         else
29:             newState ←
30:             SIMULATE(root.state, root.previousMove, move,
                root.parent.player, root.player, STEP_PLAYOUT_TIME)
31:             node ← NODE(newState)
32:             APPEND(root.children, node)
33:             return node
34:         end if
35:     else
36:         node ← UCB1(root.children)
37:         return SELECTLEAF(node)
38:     end if
39: end procedure
```

Procedure PuppetUCTCD in line 9 shows the main loop, which calls the selectLeaf procedure at line 12, which will select a leaf, expand a new node, and return it. Then a playout is performed in line 14 using a randomized policy for both players. The playout is cut short after a predefined number of frames, and an evaluation function is used. Afterwards, a standard UCT update rule is applied. Finally, when the computation time is spent, the best move at the root is selected and returned.

Procedure selectLeaf in line 21 either expands a new sibling of a leaf node, or if there are none, uses a standard UCB1 algorithm to select a node to follow down the tree. When expanding a new node, it always expands two levels, because playouts can only be performed on nodes for which both players have selected moves.

E. State Evaluation

Forwarding the state using different choices is only useful if we can evaluate the merit of the resulting states. We need to decide which of those states is more desirable from

[3]https://github.com/santiontanon/microrts

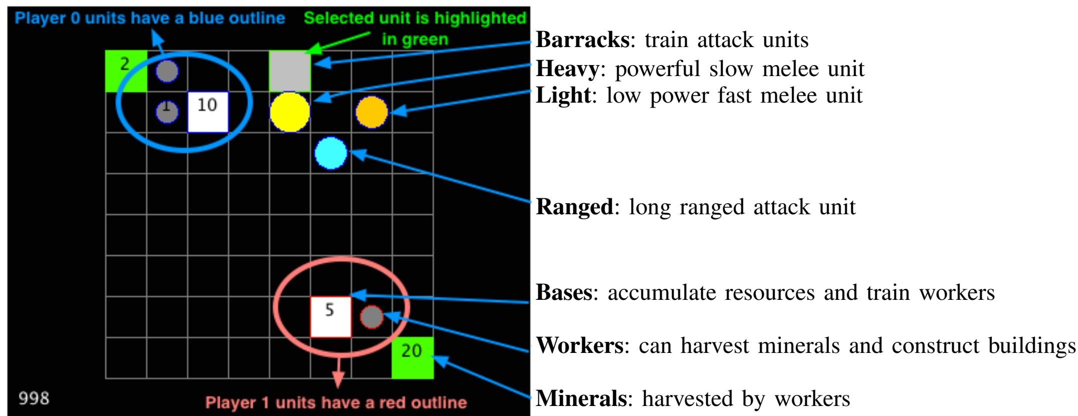


Fig. 2. Screenshot of μRTS, with explanations of the different in-game symbols.

the point of view of the player performing the search. In other words, we need to evaluate those states, assign each a numerical value and use it to compare them. In zero-sum games it is sufficient to consider symmetric evaluation functions $eval(state, player)$ that return positive values for the winning player and negative values for the losing player with $eval(state, p1) = -eval(state, p2)$.

The most common approach to state evaluation in RTS games, and the one we use in our experiments, is to use a linear function that adds a set of values that are each multiplied by a weight. The values usually represent simple features, such as the number of units of each type a player has, with different weights reflecting their estimated worth. Weights can be either hand-tuned or learned from records of past games using logistic regression or similar methods. An example of a popular metric in RTS games is Life-Time Damage, or LTD [7], which tries to estimate the amount of damage a unit could deal to the enemy during its lifetime. Another feature could be the cost of building a unit, which takes advantage of the game balancing already performed by the game designers. Costlier units are highly likely to be more useful, thus the player that has a higher total unit cost has a better chance of winning. [8] describes a state-of-the-art evaluation method based on Lanchester's attrition laws that takes into account combat unit types and their health.

[9] presents a global evaluation function that takes into account not just military features, but economic, spatial and player skill as well. [10] implements a Convolutional Neural Network for game state evaluation that provides a significant increase in accuracy compared to other methods, at the cost of execution speed. How much the trade-off is worth depends on the search algorithm using it.

A somewhat different state evaluation method involves Monte Carlo simulations. Instead of invoking a static function, one could have a pair of fast scripts, either deterministic or randomized, play out the remainder of the game, and assign a positive score to the winning player [6]. The rationale behind this method is that, even if the scripts are not of high quality, as both players are using the same policy, it is likely that whoever wins more simulations is the one that was ahead in the first place. If running a simulation until the end of the game is infeasible, a hybrid method can be used that performs a limited playout for a predetermined amount of frames, and then calls the evaluation function. Evaluation functions are usually more accurate closer to the end of a game, when the game outcome is easier to predict. Therefore, moving the application of the evaluation function to the end of the playout often results in a more accurate assessment of the value of the game state.

F. Long Term Plans

RTS game states tend to change gradually, due to actions taking several frames to execute. To take advantage of this slow rate of change, we assume that the game state doesn't change for a predefined amount of time and try to perform a deeper search than otherwise possible during a single frame. We can then use the generated solution (a series of choices for a script's choice points) to control the game playing agent, while the search produces the next solution. We call this approach having a *standing plan*.

Experiments reported in the following section investigate how advantageous is the trade-off between computation time and recency of the data being used to inform the search.

III. Experiments and Results

The experiments reported below were performed in μRTS[4], an abstract RTS game implemented in Java and designed to test AI techniques. It provides the core features of RTS games, while keeping things as simple as possible. In the basic setup only four unit and two building types are supported (all of them occupying one map tile), and there is only one resource type. μRTS is a 2-player real-time game featuring simultaneous and durative actions, possibly large maps (althouth sizes of 8x8 to 16x16 are most common), and by default all state variables are fully observable. μRTS comes with a few basic scripted players, as well as search-based players that implement several state-of-the-art RTS game search techniques. This makes it an useful tool for benchmarking new algorithms. Figure 2 shows an annotated screenshot.

μRTS comes with four scripted players, each implementing a rush strategy with different unit types. A rush is a simple

[4] https://github.com/santiontanon/microrts

strategy in which long term economic development is sacrificed for a quick army buildup and early assault on the opponent's base. The first *Puppet Search* version we tested includes a single choice point to select among these 4 existing scripts, generating a game tree with constant branching factor 4. We call this version *PuppetSingle*. A slightly more complex script was implemented as well. In addition to the choice point for selecting the unit type to build, *PuppetBasic* has an extra choice point for deciding whether to expand (i.e., build a second base) or not. Because this choice point is only active under certain conditions, the branching factor is 4 or 8, depending on the specific game state. Both ABCD and UCTCD search were used, with and without a standing plan, leading to a total of eight different agents.

The algorithms used as a benchmark are NaïveMCTS and two versions of AHTNs: AHTN-F, the strongest one on small maps, and AHTN-P, the more scalable version. These algorithms are described in detail in section IV.

All experiments were conducted on computers equipped with an Intel Core i7-2600 CPU @ 3.4 GHz and 8 GB of RAM. The operating system was Ubuntu 14.04.5 LTS, and Java JDK 1.8.0 was used.

μRTS was configured in a similar manner to experiments reported in previous papers. A computational budget of 100ms was given to each player between every simulation frame. In [11], [12] games ended in ties after running for 3000 game frames without establishing a winner, i.e. a player eliminating all the other player's units. On bigger maps this produces a large number of ties, so we used different cutoff thresholds according to the map sizes:

Size	8x8	16x16	24x24	64x64	>64x64
Frames	3000	4000	5000	8000	12000

Applying these tie rules produced tie percentages ranging from 1.1% to 3.9% in our experiments.

All algorithms share the same simple evaluation function, a randomized playout policy and a playout length of 100 simulation frames ([11], [12]), which were chosen to easily compare our new results with previously published results. The *Puppet Search* versions that use a standing plan have 5 seconds of planning time between plan switches, which was experimentally determined to produce the strongest agent. The UCTCD *Puppet Search* versions use exploration factor c = 1 for the 8x8 map, 10 for the 16x16 map, and 0.1 for all others, tuned experimentally. All other algorithms maintain their original settings from the papers in which they were introduced.

The following maps were used in the experiments. Starting positions with a single base were chosen because they are by far the most common in commercial RTS games. Unless specified, 24 different starting positions were used for each map.

8x8: an 8x8 map with each player starting with one base and one worker. This map has been used in previous papers ([11], [12]).

16x16: a 16x16 map with each player starting with one base and one worker. This map was previously used in [12].

24x24: a 24x24 map with each player starting with one base and one worker.

BloodBath: a port of a well known 64x64 StarCraft: Brood War map. 12 different starting positions were used.

AIIDE: a port of 8 of the StarCraft: Brood War maps used for the AIIDE competition[5]. 54 different starting positions were used in total.

Figures 3 and 4 show results grouped by algorithm type. The bars on the left of each group (blue) represent the average and maximum win rates of the scripts in a round robin tournament (all versus all). The middle (red) bars show the average and maximum performance of the benchmark algorithms (NaïveMCTS, AHTN-P and AHTN-F). The bars on the right of each group (yellow) show the average and maximum performance of the eight *Puppet Search* versions we implemented.

On small maps the performances of *Puppet Search* and other search based algorithms are similar. The average performance of the scripted agents is fairly low, however, on the smaller maps, there is always one competitive script (*WorkerRush* in 8x8 and 16x16, and *LightRush* in 24x24). On the bigger maps, no agent comes close to *Puppet Search*. The differences between *Puppet Search* versions are shown below in Figures 5 to 7.

Figure 5 shows similar performance for the UCTCD *Puppet Search* versions and the ABCD ones on small maps. On the bigger maps ABCD has a higher winrate. This uneven performance by UCTCD can be a result of MCTS algorithms being designed for games with a larger branching factor. This weakness is masked on the smaller maps because of the larger number of nodes that can be explored due to faster script execution.

Figure 6 shows that *PuppetSingle* clearly outperforms *PuppetBasic* on the smaller maps, while the opposite is true on the bigger ones. This exemplifies the importance of designing choice points carefully. They must be potentially useful, otherwise they are just increasing the branching factor of the search tree without providing any benefit.

Figure 7 shows that investing time to compute a longer term plan, and then using it while computing the next plan, is only useful on big maps. On such maps the playouts are usually slower and the action consequences are delayed. Therefore, deep search is important. In smaller maps, however, acting on recent game state information seems more important, as the actions' effects propagate faster. As expected, computing a plan every time an action needs to be issued instead of reusing a standing plan leads to better performance.

Table I shows win rates for all algorithms on all map sets. *PuppetABCDSingle* consistently outperforms its four constituent scripts in all but the smallest map, in which there is a clearly dominating strategy, and any deviation from it results in lower performance. After watching a few games, it seems clear that this demeanor—the total is more than the sum of its parts—is driven by some emergent patterns. Behaviors than were not included in the scripts begin to appear. Two in particular seem worthy of mention: *Puppet Search* will

[5] `http://www.starcraftaicompetition.com`

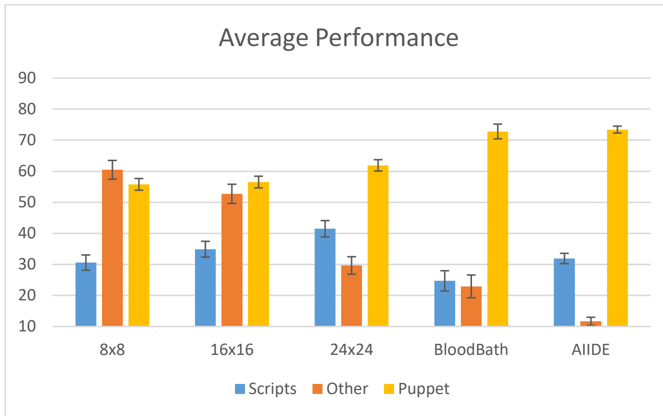


Fig. 3. Average performance of the agents, grouped by algorithm type. Error bars indicate one standard error.

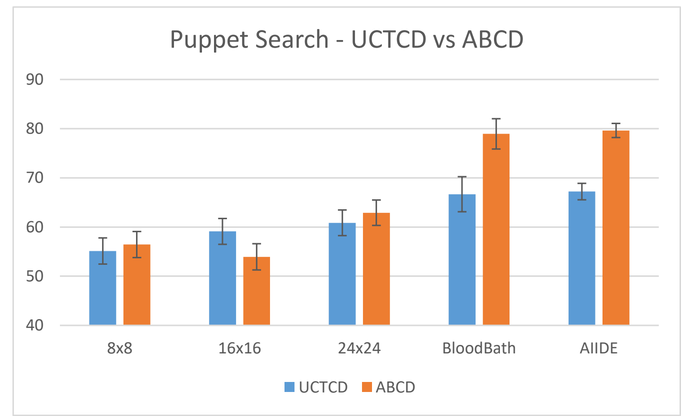


Fig. 5. Average performance of ABCD and UCTCD versions of *Puppet Search*. Error bars show one standard error.

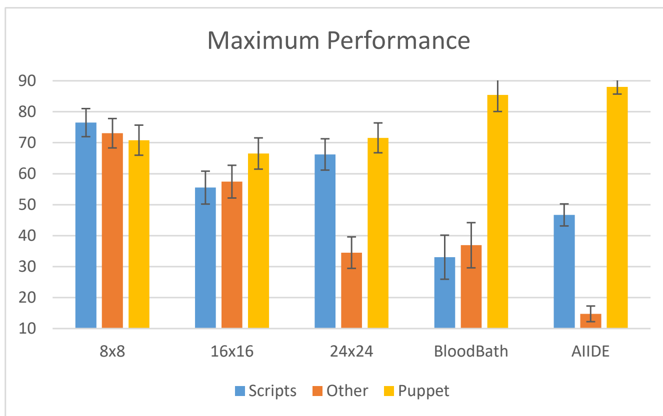


Fig. 4. Maximum performance of each type of agent. Error bars show one standard error.

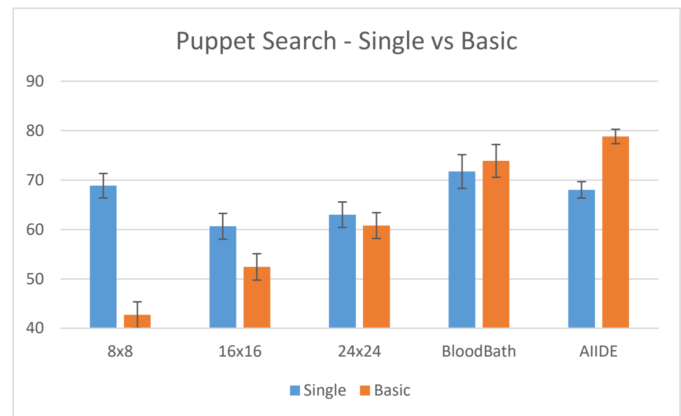


Fig. 6. Average performance of *Puppet Search* versions using a single choice point or the basic script. Error bars show one standard error.

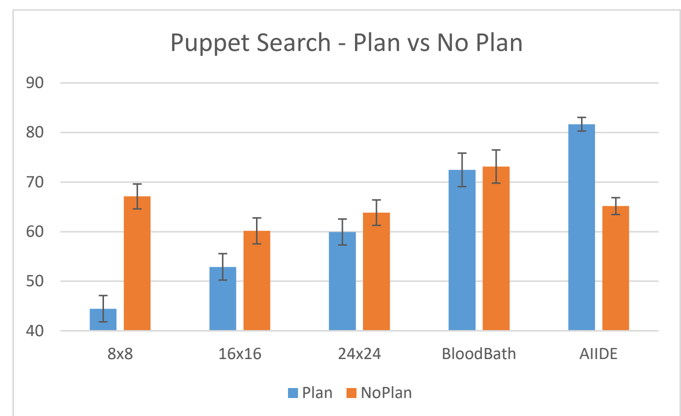


Fig. 7. Average performance of *Puppet Search* using a standing plan or replanning every time. Error bars show one standard error.

often start with a *WorkerRush* and later switch to training stronger units and use its previously built workers for gathering resources, thus bypassing the scripts' hard-coded decision to only use one gathering worker. Another similar behavior is that of switching back and forth between a script that trains ranged units and one that trains stronger melee units. The latter ones protect the weak ranged units, while both engage enemies simultaneously.

Table I also shows that NaïveMCTS cannot scale to the larger maps, and corroborates results in [12] that AHTN-F is stronger than AHTN-P in small maps, but it doesn't scale as well (though still better than NaïveMCTS). Worth noting is that two instances of *Puppet Search*—PuppetABCDSingle and PuppetUCTCDBasicNoPlan— can outperform almost all of the non *Puppet Search* agents in all maps, except for the *WorkerRush* on the 8x8 map.

IV. Related Work

NaïveMCTS [11] uses smart action sampling for dealing with large action spaces. It treats the problem of assigning actions to individual units as a Combinatorial Multi-Armed Bandit (CMAB) problem, that is, a bandit problem with multiple variables. Each variable represents a unit, and the legal actions for each of those units are the values that each variable can take. Each variable is treated separately rather than translated to a regular Multi-Armed Bandit (MAB) (by considering that each possible legal value combination is a different arm) as in UCTCD. Samples are taken assuming that the reward of the combination of the arms is just the sum of the rewards for each arm (which they call the *naive assumption*). Each arm is treated as an independent local

TABLE I
AVERAGE TOURNAMENT WIN RATES GROUPED BY MAP TYPE. 95% CONFIDENCE INTERVALS IN PARENTHESIS.

Player	8x8	16x16	24x24	BloodBath	AIIDE
WorkerRush	76.5($\pm$4.5)	55.5($\pm$5.3)	30.5($\pm$4.9)	30.4($\pm$6.8)	10.5($\pm$1.9)
LightRush	24.0($\pm$4.5)	46.1($\pm$5.3)	66.2($\pm$5.0)	33.0($\pm$7.1)	46.7($\pm$3.5)
RangedRush	12.6($\pm$3.5)	12.2($\pm$3.5)	23.2($\pm$4.5)	15.5($\pm$5.5)	28.8($\pm$3.2)
HeavyRush	9.2($\pm$3.0)	25.7($\pm$4.6)	46.0($\pm$5.3)	19.9($\pm$6.0)	41.7($\pm$3.4)
NaiveMCTS	53.7($\pm$5.2)	57.4($\pm$5.2)	33.9($\pm$4.9)	3.0($\pm$1.8)	8.2($\pm$1.3)
AHTN-P	54.6($\pm$5.2)	53.0($\pm$5.3)	34.5($\pm$5.1)	28.9($\pm$6.7)	14.7($\pm$2.3)
AHTN-F	73.1($\pm$4.7)	47.8($\pm$5.3)	20.5($\pm$4.3)	36.9($\pm$7.1)	12.0($\pm$2.0)
PuppetUCTCDSingle	65.5($\pm$5.0)	61.2($\pm$5.2)	65.5($\pm$5.1)	69.6($\pm$7.0)	78.5($\pm$2.9)
PuppetABCDSingle	68.5($\pm$4.9)	66.1($\pm$5.0)	68.8($\pm$4.9)	79.5($\pm$6.1)	83.0($\pm$2.7)
PuppetUCTCDSingleNoPlan	70.7($\pm$4.7)	66.5($\pm$5.0)	53.1($\pm$5.3)	62.5($\pm$7.3)	44.9($\pm$3.5)
PuppetABCDSingleNoPlan	70.8($\pm$4.7)	48.8($\pm$5.3)	64.6($\pm$5.1)	75.3($\pm$6.4)	65.7($\pm$3.3)
PuppetUCTCDBasic	21.3($\pm$4.4)	43.3($\pm$5.3)	53.3($\pm$5.3)	65.2($\pm$7.2)	77.2($\pm$3.0)
PuppetABCDBasic	22.6($\pm$4.4)	41.1($\pm$5.2)	52.2($\pm$5.3)	75.6($\pm$6.5)	88.0($\pm$2.3)
PuppetUCTCDBasicNoPlan	63.1($\pm$5.1)	65.5($\pm$5.0)	71.6($\pm$4.8)	69.3($\pm$6.9)	68.3($\pm$3.3)
PuppetABCDBasicNoPlan	63.8($\pm$5.0)	59.8($\pm$5.2)	66.1($\pm$5.0)	85.4($\pm$5.3)	81.8($\pm$2.7)

MAB and the individual samples are combined into a global MAB. The algorithm (naïveMCTS) was compared against other algorithms that sample or explore all possible low-level moves, such as ABCD and UCTCD. It outperformed them all, in three μRTS scenarios, with the biggest advantages found on the more complex scenarios.

Adversarial Hierarchical Task Networks (AHTNs) [12] are an alternative approach, that instead of sampling from the full action space, uses scripted actions to reduce the search space. It combines minimax tree search with HTN planning.

The authors implement five different AHTNs: 1) one with only the **Low Level** actions available in the game, which produces a game tree identical to one traversed by minimax search applied to raw low-level actions; 2) **Low Level** actions plus **Pathfinding**; 3) **Portfolio**, in which the main task of the game can be achieved only by three non-primitive tasks that encode three different hard-coded *rush* strategies, thus yielding a game tree with only one choice node at the top; 4) **Flexible**, with non-primitive tasks for harvesting resources, training units of different types, and attacking the enemy; and 5) **Flexible Single Target**, similar to Flexible, but encoded in such a way that all units that are sent to attack are sent to attack the same target, to reduce the branching factor. Experiments are presented in μRTS, a small scale RTS game designed for academic research, against four different scripts: a random script biased towards attacking, and the three scripts used by the Portfolio AHTN. The latter three AHTNs show good results, with some evidence that the Portfolio AHTN is the most likely to scale well to more complex games. The experiments presented in the previous section back this claim, though the performance couldn't match *Puppet Search*'s.

The AHTN approach, particularly the portfolio version, seems to have similar capabilities to *Puppet Search*. Both have the ability to encode strategies as high level tasks with options to be explored by the search procedure. However, *Puppet Search* is much simpler to implement, by having the ability to reuse scripts already present in the game. The full AHTN framework, including the tree search algorithm, is implemented in around 3400 lines of Java code, compared to around 1600 for *Puppet Search*. The AHTN definitions take 559 lines of LISP for AHTN-F and 833 for AHTN-P, while *Puppet Search*'s scripts took 66 lines of Java code for the single choice point one (plus 620 reused from the scripts already provided by μRTS) and 447 lines for the more complex one. The vast performance discrepancy with *PuppetABCDSingle* is due to a key difference between the algorithms. In AHTN, game tree nodes are the possible decompositions of the HTN plans, with leaves where no further decomposition can be performed. In the AHTN-P example, if both players have three choices in a single choice point, the tree has exactly 9 leaves. In *PuppetABCDSingle*, the choices are applied, the game forwarded (Algorithm 1, line 11), and then the choices will be explored again and the tree will continue to grow as long as there is time left.

Hierarchical Adversarial Search [13], [14] is an approach that uses several layers at different abstraction levels, each with a goal and an abstract view of the game state. The top layer of their three layer architecture chooses a set of objectives needed to win the game, the middle layer generates possible plans to accomplish those objectives, and the bottom layer evaluates the resulting plans and executes them at the individual unit level. For search purposes the game is advanced at the lowest level and the resulting states are abstracted back up the hierarchy. The algorithm was tested in SparCraft, a StarCraft simulator that only supports basic combat. The top level objectives, therefore, were restricted to destroying all opponent units while defending their own bases. Though this algorithm is general enough to encompass a full RTS game, only combat-related experiments were conducted.

An algorithm combining state and action abstractions for adversarial search in RTS games was proposed in [15], [16]. It constructs an abstract representation of the game state by decomposing the map into connected regions and grouping units into squads of a single unit type in each of the regions. Actions are restricted to squad movement to a neighboring region, attacking an enemy squad in the same region or staying idle. The approach only deals with movement and combat, but in their experiments it was added as a module to an existing StarCraft bot, so that it could play a full game. However, the only experiments presented were against the built-in AI, a much weaker opponent than current state-of-the-art bots.

Finally, the main difficulty with applying *PuppetSearch*, or

any look-ahead search technique to a commercial RTS game, such as StarCraft: Brood War, is the lack of an accurate forward model or simulator. In contrast with board games, where the forward model is precisely detailed in the rules of the game, and everyone can build its own simulator to use for search purposes, commercial games are usually closed source, which means we don't have access to a simulator, nor can we easily reverse engineer it. An earlier version of *PuppetSearch* [3] has been used in StarCraft, with encouraging results despite to the poor accuracy of the simulator used. Experiments in the current paper were performed in μRTS to better evaluate *PuppetSearch*'s performance without external confounding elements.

V. Conclusions and Future Work

We have introduced a new search framework, *Puppet Search*, that combines scripted behavior and look-ahead search. We presented a basic implementation as an example of using *Puppet Search* in RTS games, with the goal of reducing the search space and make adversarial game tree search feasible. The decision tree structure of the scripts ensures that only the choice combinations that make sense for a particular game state will be explored. This reduces the search effort considerably, and because scripts can play entire games, we can use the previous plan for as long as it takes to produce an updated one.

Our experiments show a similar performance to top scripted and search based agents in small maps, while vastly outperforming them on larger ones. Even a script with a single choice point to choose between different strategies can outperform the other players in most scenarios. Furthermore, on larger maps, *Puppet Search* benefits from the ability to use a standing plan to issue actions, while taking more time to calculate a new plan, resulting in even stronger performance.

From a design point of view *Puppet Search* allows game designers—by using scripts— to keep control over the range of behaviors the AI system can perform, while the adversarial look-ahead search enables it to better evaluate action outcomes, making it a stronger and more believable enemy. Based on promising experimental results on RTS games, we expect this new search framework to perform well in any game for which scripted AI systems can be built.

As for the general idea of *Puppet Search*, we believe it has great potential to improve decision quality in other complex domains as well in which expert knowledge in form of non-deterministic scripts is available.

In the future, we would like to extend this framework to tackle games with partial observability by using state and strategy inference similar to Bayesian models for opening prediction [17] and plan recognition [18], and particle models for state estimation [19].

References

[1] O. Syed and A. Syed, "Arimaa-a new game designed to be difficult for computers," *ICGA JOURNAL*, vol. 26, no. 2, pp. 138–139, 2003.

[2] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A survey of real-time strategy game AI research and competition in StarCraft," *TCIAIG*, vol. 5, no. 4, pp. 293–311, 2013.

[3] N. A. Barriga, M. Stanescu, and M. Buro, "Puppet Search: Enhancing scripted behaviour by look-ahead search with applications to Real-Time Strategy games," in *Eleventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2015.

[4] D. Churchill, M. Preuss, F. Richoux, G. Synnaeve, A. Uriarte, S. Ontanon, and M. Certicky, "StarCraft bots and competitions," *Springer Encyclopedia of Computer Graphics and Games*, 2016.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[6] D. Churchill, A. Saffidine, and M. Buro, "Fast heuristic search for RTS game combat scenarios," in *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE'12, 2012, pp. 112–117.

[7] A. Kovarsky and M. Buro, "Heuristic search applied to abstract combat games," *Advances in Artificial Intelligence*, pp. 66–78, 2005.

[8] M. Stanescu, N. A. Barriga, and M. Buro, "Using Lanchester attrition laws for combat prediction in StarCraft," in *Eleventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2015.

[9] G. Erickson and M. Buro, "Global state evaluation in StarCraft," in *Proceedings of the Tenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. AAAI Press, 2014, pp. 112–118.

[10] M. Stanescu, N. A. Barriga, A. Hess, and M. Buro, "Evaluating real-time strategy game states using convolutional neural networks," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2016.

[11] S. Ontañón, "The combinatorial multi-armed bandit problem and its application to real-time strategy games," in *AIIDE*, 2013.

[12] S. Ontañón and M. Buro, "Adversarial hierarchical-task network planning for complex real-time games," in *Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI)*, 2015, pp. 1652–1658.

[13] M. Stanescu, N. A. Barriga, and M. Buro, "Introducing hierarchical adversarial search, a scalable search procedure for real-time strategy games," in *Proceedings of the Twenty-first European Conference on Artificial Intelligence (ECAI)*, 2014, pp. 1099–1100.

[14] ——, "Hierarchical adversarial search applied to real-time strategy games," in *Proceedings of the Tenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2014, pp. 66–72.

[15] A. Uriarte and S. Ontañón, "Game-tree search over high-level game states in RTS games," in *Proceedings of the Tenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE'14, 2014, pp. 73–79.

[16] ——, "High-level representations for game-tree search in RTS games," in *Artificial Intelligence in Adversarial Real-Time Games Workshop, Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014.

[17] G. Synnaeve and P. Bessiere, "A Bayesian model for opening prediction in RTS games with application to StarCraft," in *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*. IEEE, 2011, pp. 281–288.

[18] G. Synnaeve, P. Bessiere *et al.*, "A Bayesian model for plan recognition in RTS games applied to StarCraft," *Proceedings of AIIDE*, pp. 79–84, 2011.

[19] B. G. Weber, M. Mateas, and A. Jhala, "A particle model for state estimation in real-time strategy games," in *Proceedings of AIIDE*, AAAI Press. Stanford, Palo Alto, California: AAAI Press, 2011, p. 103–108.

Nicolas A. Barriga is a Ph.D. candidate at the University of Alberta, Canada. He earned B.Sc., Engineer and M.Sc. degrees in Informatics Engineering at Universidad Técnica Federico Santa María, Chile. After a few years working as a software engineer for Gemini and ALMA astronomical observatories he came back to graduate school and he is currently working on state and action abstraction mechanisms for RTS games.

Marius Stanescu is a Ph.D. candidate at the University of Alberta, Canada. He completed his M.Sc. in Artificial Intelligence at University of Edinburgh in 2011, and was a researcher at the Center of Nanosciences for Renewable & Alternative Energy Sources of University of Bucharest in 2012. Since 2013, he is helping organize the AIIDE StarCraft Competition. Marius' main areas of research interest are machine learning, AI and RTS games.

Michael Buro is a professor in the computing science department at the University of Alberta in Edmonton, Canada. He received his PhD in 1994 for his work on Logistello - an Othello program that defeated the reigning human World champion 6-0. His current research interests include heuristic search, pathfinding, abstraction, state inference, and opponent modeling applied to video games and card games. In these areas Michael and his students have made numerous contributions, culminating in developing fast geometric pathfinding algorithms and creating the World's best Skat playing program and one of the strongest StarCraft: Brood War bots.

View publication stats