Puppet Search: Enhancing Scripted Behavior by Look-Ahead Search with Applications to Real-Time Strategy Games

Nicolas A. Barriga, Marius Stanescu, and Michael Buro
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada, T6G 2E8
{barriga|astanesc|mburo}@ualberta.ca

Abstract

Real-Time Strategy (RTS) games have shown to be very resilient to standard adversarial tree search techniques. Recently, a few approaches to tackle their complexity have emerged that use game state or move abstractions, or both. Unfortunately, the supporting experiments were either limited to simpler RTS environments (μRTS, SparCraft) or lack testing against state-of-the-art game playing agents.

Here, we propose *Puppet Search*, a new adversarial search framework based on 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 a move to be applied next. Such moves can be executed in the actual game, thus letting the script play, or in an abstract representation of the game state which can be used by an adversarial tree search algorithm. Puppet Search returns a principal variation of scripts and choices to be executed by the agent for a given time span.

We implemented the algorithm in a complete StarCraft bot. Experiments show that it matches or outperforms all of the individual scripts that it uses when playing against state-of-the-art bots from the 2014 AIIDE StarCraft competition.

Introduction

Unlike several abstract games such as Chess, Checkers, or Backgammon, for which strong AI systems now exist that play on par with or even defeat the best human players, progress on AI systems for Real-Time Strategy (RTS) video games has been slow (Ontanón et al. 2013). For example, in the man-machine matches following the annual StarCraft AI competitions held at the AIIDE conference, a strong human player was able to defeat the best bots with ease in recent years.

When analysing these games several reasons for this playing strength gap become apparent: to start with, good human players have knowledge about strong openings and playing preferences of opponents they encountered before. They also can 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. Game AI systems, on the other hand, are still mostly scripted, have only modest opponent modelling abilities, and generally don't seem to be able to adapt to unforeseen circumstances well. In games with small branching factors a successful approach to overcome these issues is to use 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 by reducing action choices by means of scripts that expose choice points to the look-ahead search.

Copyright © 2015, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Background

A few attempts have been made recently to use state and action abstraction in conjunction with adversarial search in RTS games. (Stanescu, Barriga, and Buro 2014a; 2014b) propose Hierarchical Adversarial Search in which each search layer works at a different abstraction level, and each is given 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 those 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. Their 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. Though this algorithm is general enough to encompass a full RTS game, only combat-related experiments were conducted.

Another proposal, by (Uriarte and Ontañón 2014a; 2014b), followed a different path, by abstracting the game state, searching at the highest abstraction level, and then translating the results back to the lower level for execution. The state representation uses a decomposition of the map into connected regions, and groups all units of the same type into squads in each region. Moves are restricted to squads, which can move to a neighboring region, attack, or stay idle. This approach, similar to the previous one, only deals with combat, but it was added to an existing StarCraft bot, so that it can play a full game. However, results were only presented for playing against the built-in AI which is much weaker than state-of-the-art bots.

Puppet Search

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 a search or solution technique of our choice to the transformed game, which 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 don't necessarily produce mediocre actions because script computations can themselves be based on (local) search or other forms of optimizations. If more time is available, our search can visit more choice points and generate better moves. Finally, for anytime decision scenarios, one can envision an iterative widening approach that over time increases the number of choice points the scripts expose, thereby improving move quality.

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 (Ontanón et al. 2013), 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 tricky, 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 preferable to launch a search to decide the issue, assuming sufficient computational resources are available. Likewise, letting look-ahead 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.

In the rest of the paper we will use the following terminology when discussing general aspects of *Puppet Search* and its application to RTS games in particular:

Game Move: a move that can be applied directly to the game state. It could be a simple move such as placing a stone in Go, or a combined move like instructing several units to attack somewhere while at the same time ordering the construction of a building and the research of an upgrade in an RTS game like StarCraft.

Script: a function that takes a game state and produces a game move. How the move is produced is irrelevant — it could be rule based, search based, etc. A script can expose choice points, asking the caller of the function to make a decision on each particular choice applicable to the current game state, which we call **puppet move**. For instance, one possible implementation could provide function GETCHOICES(s) which returns a set of puppet moves in game state s that a script offers, together with function EXECCHOICE(s, m) that applies script moves to state s following puppet move choice m.

Applying Puppet Search

Reducing the search space in the fashion described above allows AI systems to evaluate long-term effects of actions, which for instance has been crucial to the success of Chess programs. If we wanted to apply *Puppet Search* to a standard two player, zero sum, perfect information game we can use any existing adversarial tree search algorithm and have it search over sequences of puppet moves. Monte-Carlo Tree Search (MCTS) (Kocsis and Szepesvári 2006) seems particularly well suited, as it does not require us to craft an evaluation function, and scripts already define playout policies. Alpha-Beta search could also be used, granted a suitable evaluation function can be provided for the game.

Puppet Search does not concern itself with either the origin or the inner workings of scripts. E.g., scripts can be produced by hand coding expert knowledge, or via machine learning approaches (Synnaeve, Bessiere, and others 2011; Ontañón et al. 2008). Also, scripts can produce moves in a rule-based fashion or they can be search based (Ontañón 2013; Churchill and Buro 2013).

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 (Churchill, Saffidine, and Buro 2012). 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 3) and the opponent thereafter. Policies they discuss 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 and takes into account that at any point in time both players execute a puppet move. The maximum search depth is assumed to be even, which lets both players select a puppet move to forward the

Algorithm 1 Puppet ABCD Search

```
1: procedure PUPPETABCD(s, h, m_1, α, β)
2:   if h = 0 or TERMINAL(s) then return EVAL(s)
3:   toMove ← PLAYERTOMOVE(s, policy, h)
4:   for m_2 in GETCHOICES(s, toMove) do
5:     if m_1 = ∅ then
6:       v ← PUPPETABCD(s, h − 1, m_2, α, β)
7:     else
8:       s' ←COPY(s)
9:       EXECCHOICES(s', m_1, m_2)
10:      v ← PUPPETABCD(s', h − 1, ∅, α, β)
11:    end if
12:    if toMove = MAX and v > α then α ← v
13:    if toMove = MIN and v < β then β ← v
14:    if α ≥ β then break
15:  end for
16:  return toMove = MAX ? α : β
17: end procedure
```

world in line 9. Moves for the current player are generated in line 4. 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 5), the current player's move m_2 is passed on to a subsequent PUPPETABCD call at line 6. Otherwise, both players' moves are applied to the state (line 9). The exact mechanism of applying a move is domain specific. We will give an example specific to RTS games later (Algorithm 2). The resulting Algorithm 1 is the search routine that will be used in the *Puppet Search* application to a real RTS game which we discuss next.

Puppet Search in RTS Games

This section describes the implementation of *Puppet Search* in a StarCraft game playing agent. StarCraft is a popular RTS game in which players can issue actions to all individual game units under their control — simultaneously — several times per second. Moreover, some actions are not instantaneous — they take some time to complete and their effects are sometimes randomized, and only a partial view of the game state is available to the players. Finally, the size of the playing area, the number of available units, and the number of possible actions at any given time are several orders of magnitude larger than most board games.

Our implementation tackles some of these issues by adapting standard AI algorithms, such as serializing moves in simultaneous move settings, or ignoring some issues altogether, for instance by assuming deterministic action effects and perfect information. Also, due to software limitations such as the lack of access to the engine to forward the world during the search and the unavailability of a suitable simulator, several simplifications had to be made.

Concurrent Actions and Script Combinations

So far we have presented a puppet move as a choice in a single script. In the presence of concurrent actions in RTS games, such as "send unit A there" and "send unit B there", it might be more natural to let a set of scripts deal with units or groups independently — each of them exposing their own choice points as necessary. To fit this into the *Puppet Search* framework, we could combine all such scripts and define a single choice point whose available (concurrent) puppet moves consist of vectors of the individual scripts' puppet moves. For example, a concurrent puppet move could be a pair of low-level puppet moves such as "take option 2 at the choice point of the script dealing with unit group A, and option 3 at the choice point of group B", which could mean "A assaults the main base now and B builds a secondary base when the game reaches frame 7000". The puppet move pair can then be executed for a fixed time period, or as long as it takes the search to produce an updated plan, or as long as no script encounters a choice point for which the move doesn't contain a decision. An example of the latter could be that the enemy built the infrastructure to produce invisible units which we cannot detect with our current technology. If that possibility wasn't encountered by the search, there is no decision made for that choice point.

In another scenario we may have access to multiple scripts that implement distinct full game strategies. For example, we could regard all programs participating in recent StarCraft AI competitions as scripts, and try to combine them into one StarCraft player based on the *Puppet Search* idea, by identifying weak spots in the individual strategies, exposing appropriate choice points, and then adding a choice point at the top that would give the AI system the option to continue executing one of the scripts for a given number of frames or until certain game events happen.

StarCraft Scripts

A common type of strategy in StarCraft is the *rush*: trying to build as many combat units as fast as possible, in an effort to destroy the opponent's base before he has the time to build suitable defences. This kind of strategy usually sacrifices long term economy in exchange for early military power. A range of rushes are possible, from quickly obtaining several low level units, to waiting some time to obtain a handful of high level units. We have implemented four rush strategies that fall in that spectrum: S1) Zealot rush: a fast rush with inexpensive units, S2) Dragoon rush: somewhat slower and stronger, S3) Zealot/Dragoon rush: combines the previous two, and S4) Dark Templar rush: slower, but creating more powerful units. We then combine these four strategies into one script with a single choice point at the top. We expect *Puppet Search* to be able to figure out which unit types are more suitable to assault the enemy's base, and adapt to changes —by switching unit types— in the opponent's defensive force composition during the game. The scripts share the rest of the functionality needed to satisfy the requirement that they must be able to play a full game.

Planning and Execution

During game play, a player receives the state of the game and can issue actions to be applied to that state by the game engine. This happens at every game frame, whether you are a human interacting with StarCraft via the GUI, or a bot receiving a game state object and returning a vector of actions. We call this the execution phase.

The vector of actions to be executed is decided in the planning phase, usually by some form of look-ahead search which requires a mechanism to see the results of applying actions to a state. A video game bot cannot use the game engine because sending a move to it would execute it in the game. Instead, it needs to be able to simulate the game. If a perfect game simulation existed, aligning planning and execution is easy, as it is the case for traditional board games, in which, for instance, the look-ahead search can know the exact outcome of advancing a pawn in a game of Chess. In the following subsections we will discuss problems arising from inaccurate simulations further.

Another important aspect is that in most turn based games planning and execution are interleaved: at each turn, the player to move takes some time to plan, and then executes a single move. That need not be the case in general, however, as the plan returned by the search could consist of several moves. As long as the moves are legal, we can continue executing the same plan. In our case, the plan returned by the planning phase is a collection of decisions in a script that can play a full game. So it can be used for execution as long as it doesn't reach a choice point for which no decision has been selected yet.

State Representation and Move Execution

The StarCraft game state contains every unit's state — position, hit points, energy, shields, current action, etc. — plus some general information about each side such as upgrades, resources, and map view. We will make a copy of all this information into a state data structure of our own which is used in our search procedure.

During the search, applying a puppet move requires forwarding the game state for a certain number of frames during which buildings need to be constructed, resources mined, technology researched, units moved, and combat situations resolved. We use the Build Order Search System (BOSS) (Churchill and Buro 2011) for forwarding the economy and SparCraft (Churchill and Buro 2013) for forwarding battles. BOSS is a library that can simulate all the economic aspects of a StarCraft game — resource gathering, building construction, unit training and upgrades —, while SparCraft is a combat simulator. That still leaves unit movement for which we implemented a simplified version of our bot's logic during the game: during the execution phase, our bot acts according to a set of rules or heuristics to send units to either defend regions under attack or attack enemy regions. We mimic those rules, but as we cannot use the StarCraft engine to issue orders to units and observe the results of those actions, we built a high level simulation in which units are grouped into squads and these squads move from one region to another along the shortest route, towards the region they are ordered to defend or attack. If they encounter an enemy squad along the way, SparCraft is used to resolve the battle. Although this simulation is not an exact duplicate of the bot's in-game logic, it is sufficiently accurate to allow the search to evaluate move outcomes with respect to combat.

Forwarding the world by a variable number of frames generates an uneven tree in which two nodes at the same tree depth are possibly not referring to the same game time

Algorithm 2 Executing Choices and Forwarding the World

```
1:  procedure EXECCHOICES(State s, Move m1, Move m2)
2:    define constant frameLimit = N
3:    b1 ← GETBUILDORDER(s, m1)
4:    b2 ← GETBUILDORDER(s, m2)
5:    while s.currentFrame < frameLimit and b1 ≠ ∅ and
      b2 ≠ ∅ and CHECKCHOICEPOINTS(s) do
6:      b ← POPNEXTORDER(b1, b2)      ▷ get the next order
                                      ▷ that can be executed
7:      t ← TIME(b)                   ▷ time at which order
                                      ▷ b can be executed
8:      FORWARDECONOMY(s, t)          ▷ gather resources,
                                      ▷ finish training units,
                                      ▷ build buildings
9:      BOSSENQUEUEORDER(s, b)
10:     FORWARDSQUADS(s, t)           ▷ movement and combat
11:   end while
12: end procedure
```

(i.e., move number in board games, or game frame in RTS games). Evaluating such nodes has to be done carefully. Unless we use an evaluation function with a global interpretation — such as winning probability or expected game score — iterative deepening by search depth cannot be performed because values of states at very different points in the game could be compared. Therefore, the iterative deepening needs to be performed with respect to game time.

Because simultaneously forwarding the economy, squads, and battles is tricky and computationally expensive, we decided that the actions that take the longest game time would dictate the pace: forwarding the economy, as shown in Algorithm 2. Every time we need to apply a move, the script is run, and it returns an ordered list of buildings to construct (lines 3, 4). In line 8, the state is forwarded until the prerequisites (resources and other buildings) of the next building on the list are met. The building is then added to the BOSS build queue (line 9), so that next time the economy is forwarded, its construction will start. Then, at line 10, squad movement and battles are forwarded for the same number of frames. The scripts will then be consulted to check if they have hit an undecided choice point, in which case the forwarding stops. This cycle continues until either a build list becomes empty, the game ends, or a given frame limit N is reached. For simplicity we show this frame limit as a constant in line 2.

Search

Due to having an imperfect model for forwarding unit movement and combat, we decided against against using MCTS which would heavily rely on it in the playout phase. Instead we opted for using a modified Alpha-Beta search. We only search to even depths which ensures that each player chooses a puppet move and both moves can be applied simultaneously. Iterative deepening is done by game time, not by search depth. In every iteration we increase the depth by N frames. When applying a pair of moves, the world will be

forwarded by N frames until one of the build orders is completed or until a new choice point is reached. The algorithm then continues with making the next recursive call. Due to the three different stopping conditions, a search for X frames into the future can reach nodes at different depths.

State Evaluation

We evaluated a few approaches, such as the destroy score used in (Uriarte and Ontañón 2014b; 2014a) or LTD2 (Kovarsky and Buro 2005; Churchill, Saffidine, and Buro 2012). The first one is a score assigned by StarCraft to each unit based on the resources required to build it, which for our purposes overvalues buildings. The second is LTD2 ("Life-Time-Damage-2"), a measure of the average damage a unit can deal over its lifetime, which doesn't value non-attacking units and buildings at all, and even for combat units, it doesn't account for properties such as range or speed, or special abilities like cloaking. Instead of trying to handcraft an evaluation function that addresses these shortcomings we decided to use a model based on Lanchester's attrition laws presented in (Stanescu, M. and Barriga, N.A. and Buro, M. 2015). It automatically trains an evaluation function for units, tuned to each individual opponent. It still only accounts for combat units, so an obvious next step would be to use a function that evaluates the entire game state as described in (Erickson and Buro 2014).

Hash Tables

Because move execution is costly we use a hash table to store states, indexed by hashing the sequence of moves used to get to that state. It works similarly to a transposition table based on Zobrist hashing (Zobrist 1970), but as the state is too big and costly to hash, we instead hash the sequence of moves leading to the current state from the root. Clearing a big hash table can be costly. So, to avoid clearing it after every search, at the root of the tree the hash value is seeded with a 64-bit random number. This makes it very unlikely that hashes from two different searches match. The table is then indexed by the hash value modulo the table size, and the hash value itself is stored along with the state. This hash table can be queried to see if a certain move was previously applied to a certain state, in which case the successor state can simply be retrieved instead of applying the move again. A standard transposition table is also used, with the same hashing mechanism as the hash table. As we are hashing the moves that lead to a state, rather than the state itself, we don't expect any transpositions to arise with our hashing mechanism, so the table is only used for retrieving best moves for states already seen which likely lead to earlier beta cut-offs when considered first.

Implementation Limitations

StarCraft runs at 24 frames per second, but our search needs more time than the 42[ms] between frames. StarCraft also has a hard limit of 45 seconds of inactive time before forcefully forfeiting the game. To handle this limitations, our current implementation freezes the game for six seconds of thinking time when it needs to get a new puppet move. This happens whenever the build order returned by the previous puppet move is fully constructed, approximately every 1500 to 2500 frames. We found this to be a good balance between the depth of a single search instance, vs. how many searches we can execute during a complete game. Future improvements could be either to spread the search computation across several frames, or to move it to a background thread, which is not trivial due to BWAPI not being thread safe.

Dealing with imperfect information is not in this paper's scope, so in our experiments we disable StarCraft's *Fog of War*. Having the full game state information available to our agent, the game state at the beginning of the search contains both players' units and buildings. To avoid giving our bot an unfair advantage over its opponents, we retain our base bot's scouting behaviour and do not use the extra information except for the initial state of the search.

Both simplifications have been used before. For instance, (Uriarte and Ontañón 2014b) pause games for up to 30 seconds every 400 frames and also disables the *Fog of War*.

Experiments and Results

Experiments were conducted using 12 VirtualBox virtual machines (VMs), each equipped with 2 cores of an Intel Xeon E5420 CPU running at 2.5GHz, and 2GB of RAM. The guest operating system was Microsoft Windows XP SP3. The StarCraft AI Tournament Manager (github.com/davechurchill/StarcraftAITournamentManager) was used to coordinate the matches. The default tournament timeout policy was changed to allow our bot to spend 6 seconds of search time it needs when the build queue runs out about every 2000 frames. As our bot currently contains only Protoss scripts for *Puppet Search* to use, we play against 6 of the top AIIDE 2014 Protoss bots (Ximp, Skynet, UAlbertaBot, Xelnaga, Aiur, and MooseBot) named E1...E6 on the 10 maps used in the 2014 AIIDE StarCraft AI competition. We will compare *Puppet Search*'s performance against E1...E6 with that of 4 versions of our bot playing a fixed script (S1...S4).

Table 1 shows the results of *Puppet Search* compared to the fixed scripts by playing against AIIDE 2014 bots. Our bot has a higher win rate than all individual scripts, except for S1 for which the mean and median performances are not statistically different from Puppet's: the Chi-squared two-tailed P values are 0.45 and 0.39 respectively). The scripts' average is the performance we can expect of a bot that plays one of the four scripts at random. The scripts' maximum value is the performance we can expect of a bot that plays the best script against each opponent. To be able to play this best response, we would need access to the opponents' bots beforehand. This is, of course, unfeasible in a real tournament. The best we could do is select a best response to last year's version of each bot. But this would likely lead to a much less robust implementation that wouldn't be able to respond to opponent's behaviour changes well.

Table 1 also suggests that the search could benefit from having access to more scripts: against all opponents for which at least one of the scripts defeats it more than 50% of the time *Puppet Search* also defeats it more than 50% of the time. In the cases of E2 and E6, even though only 1 script performed well (S4 and S1 respectively), *Puppet*

Table 1: Win rate of individual scripts and *Puppet Search* playing against AIIDE 2014 bots E1 . . . E6. 100 games were played between each pair of bots on 10 different maps.

	Med.	Mean	E1	E2	E3	E4	E5	E6
S1	55	49.7	0	42	47	78	68	63
S2	29.5	31.5	28	31	13	42	56	19
S3	18	19.8	6	20	3	18	54	18
S4	40.5	40.8	1	72	2	84	77	9
Avg.	34.3	35.5	8.8	41.3	16.3	55.5	63.8	27.3
Max	67.5	61.8	28	72	47	84	77	63
P.S.	61	51.8	3	55	42	72	67	72

Table 2: Win rate of *Puppet Search* against individual scripts. 100 games were played between each pair of bots on 10 different maps.

	S1	S2	S3	S4
P.S.	44	77	99	100

Table 3: Probability of winning 50 or more games out of 100 against each opponent, assuming the probabilities in Table 1 are the true winning probabilities of each bot.

	E1	E2	E3	E4	E5	E6
S1	0	.07	.31	$>.99$	$>.99$	$>.99$
P.S.	$<.01$	.86	.07	$>.99$	$>.99$	$>.99$

Search achieves win rates of 55% and 72%, respectively. This last result is better than any of the scripts. On the other hand, in the two instances where no script defeated the opponent (E1 and E3), the search couldn't defeat it either.

Table 2 shows the performance of *Puppet Search* when matched directly against each of the individual scripts it uses. *Puppet Search* defeats three of the fixed scripts by a wide margin, while the difference with S1 is not statistically significant (the Binomial P value is 0.14).

Because the average win rates of our bot and script S1 are not statistically different, Table 3 shows an analysis of the probabilities of winning 50 or more games, out of 100, against each opponent. This shows that using *Puppet Search* is more robust than using a single script, it has a higher probability of winning a series of games against a single random opponent. This is due to S1 successfully exploiting opponents that can't counter its fixed strategy, while using *Puppet Search* produces a bot that can win against a wider variety of opponents, but doesn't exploit the weak opponents as much.

We should bear in mind that the implementation evaluated here has some serious limitations, such as inaccurate squad movement and combat, an evaluation function that only accounts for combat units and searching over only a small number of scripts. Furthermore, all four scripts encode similar strategies, rushes, only changing the unit type used for the rush. Because of these limitations, the average performance doesn't show improvements over the best benchmark script. However, the analysis of the results in Table 3 indicates some of the outcomes we can expect from a more thorough implementation: a robust algorithm that can defeat a wider range of opponents than any fixed strategy can, and one that can take advantage of any script that can counter at least a single opponent strategy.

Table 4: Search speed, forward length and depth reached in 6 seconds. Average over 10 games.

	Nodes/sec	Fwd. length	Depth[frames]
Early game	648.8	1984.2	10000
Midgame	505.9	941.4	6000

Table 4 shows the nodes visited per second, average world forward length and the depth reached by the search in the 6 seconds allotted. Depth is in frames because, as mentioned earlier, we do iterative deepening by frame rather than by tree depth. We use a frame forward limit of 2000 (Algorithm 2, line 2), but as there are several stopping conditions, this number varies. Smaller numbers mean a deeper tree needs to be searched to reach a given frame depth. 500 nodes per second might not seem much, but at 1000 frames of game time forwarded on average for each node, we are running 500k frames per second. Looking 6000 frames ahead means that *Puppet Search* is able to evaluate action effects at least a quarter of the average game length into the future.

To the best of our knowledge, these are the first experiments conducted for high level search algorithms in RTS games against state-of-the-art AI systems. The search system over high level states shown in (Uriarte and Ontañón 2014b) used StarCraft's built-in AI as a benchmark. The hierarchical search system implemented by (Stanescu, Barriga, and Buro 2014a) used SparCraft, a StarCraft simulator limited only to the combat aspects of RTS games.

Conclusions and Future Work

We have introduced a new search framework, *Puppet Search*, that combines scripted behaviour 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. *Puppet Search* builds on recent work on hierarchical decomposition and high-level state representation by adding look-ahead search on top of expert knowledge in the form of non-deterministic scripts. While in our experiments the average performance against all chosen opponents is similar to the best benchmark script, further analysis indicates that *Puppet Search* is more robust, being able to defeat a wider variety of opponents. Despite all the limitations of the implementation used for the experiments, such as imperfect squad movement and combat modelling, incomplete evaluation function, and small variety of scripts, our encouraging initial results suggest that this approach is worth further consideration.

In future work we would like to remove the current limitations of our implementation, starting with the search time limits and the need to access the full map information. Some techniques have been proposed to provide inference capabilities to estimate the enemy's current state from available scouting information (Weber, Mateas, and Jhala 2011;

Synnaeve and Bessiere 2011; Synnaeve, Bessiere, and others 2011; Weber and Mateas 2009). Improving the performance of the current Alpha-Beta search will require a more comprehensive evaluation function. Alternatively, switching to MCTS would need more accurate combat and squad movement simulation to perform playouts. Even with the simple scripts available, *Puppet Search* manages to produce a robust player, showing it is more than just the sum of its components. However, some effort needs to be dedicated to crafting scripts that contain choice points dealing with as wide a range of situations as possible, to provide *Puppet Search* with the building blocks it needs to adapt to any scenario.

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.

References

Churchill, D., and Buro, M. 2011. Build order optimization in StarCraft. *Proceedings of AIIDE* 14–19.

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

Churchill, D.; Saffidine, A.; and Buro, M. 2012. Fast heuristic search for RTS game combat scenarios. In *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*.

Erickson, G., and Buro, M. 2014. Global state evaluation in StarCraft. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *Machine Learning: ECML 2006*. Springer. 282–293.

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

Ontañón, S.; Mishra, K.; Sugandh, N.; and Ram, A. 2008. Learning from demonstration and case-based planning for real-time strategy games. In Prasad, B., ed., *Soft Computing Applications in Industry*, volume 226 of *Studies in Fuzziness and Soft Computing*. Springer Berlin / Heidelberg. 293–310.

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. *TCIAIG* 5(4):293–311.

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

Stanescu, M.; Barriga, N. A.; and Buro, M. 2014a. Hierarchical adversarial search applied to real-time strategy games. In *Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.

Stanescu, M.; Barriga, N. A.; and Buro, M. 2014b. Introducing hierarchical adversarial search, a scalable search procedure for real-time strategy games. In *European conference on Artificial Intelligence*.

Stanescu, M. and Barriga, N.A. and Buro, M. 2015. Using Lanchester attrition laws for combat prediction in StarCraft. In *accepted for presentation at the Eleventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.

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

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

Uriarte, A., and Ontañón, S. 2014a. Game-tree search over high-level game states in RTS games. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Uriarte, A., and Ontañón, S. 2014b. High-level representations for game-tree search in RTS games. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Weber, B. G., and Mateas, M. 2009. A data mining approach to strategy prediction. In *IEEE Symposium on Computational Intelligence and Games (CIG)*.

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

Zobrist, A. L. 1970. A new hashing method with application for game playing. *ICCA journal* 13(2):69–73.