Script- and Cluster-based UCT for StarCraft

Niels Justesen
IT University of Copenhagen
Copenhagen, Denmark
noju@itu.dk

Bálint Tillman
IT University of Copenhagen
Copenhagen, Denmark
btil@itu.dk

Julian Togelius
IT University of Copenhagen
Copenhagen, Denmark
julian.togelius@gmail.com

Sebastian Risi
IT University of Copenhagen
Copenhagen, Denmark
sebr@itu.dk

***Abstract*—Monte Carlo methods have recently shown promise in real-time strategy (RTS) games, which are challenging because of their fast pace with simultaneous moves and massive branching factors. This paper presents two extensions to the Monte Carlo method *UCT Considering Durations* (UCTCD) for finding optimal sequences of actions for units engaged in combat in the RTS game StarCraft. The first extension is a *script-based approach* inspired by *Portfolio Greedy Search* and searches for sequences of scripts instead of actions. The second extension is a *cluster-based approach* as it assigns scripts to clusters of units based on their type and position. The presented results demonstrate that both the script-based and cluster-based UCTCD extensions outperform the original UCTCD with a winning percentage of 100% for battles with 32 units or more. Additionally, unit clustering is shown to give some improvement in large scenarios while it is less effective in small combats. The algorithms were tested in our StarCraft combat simulator called *JarCraft*, a complete Java translation of the original C++ package SparCraft, made in hopes of making this research area more accessible.**

I. Introduction

Controlling units in real-time strategy (RTS) games is a challenging problem in AI research as these games usually are characterized by massive branching factors, simultaneous moves, partial observability, open-endedness and a short amount of time to decide what moves to perform [1], [2]. The RTS game StarCraft[1] is the most popular test bed for AI research in this genre [1]. The challenge in StarCraft combats is to find the optimal sequence of actions for a group of units engaged in combat. However, the original AI in StarCraft as well as state of the art bots from StarCraft AI competitions appear to control units with simple scripts alone without implementing any learning techniques or search methods.

The UCT (*Upper Confidence bound applied to Trees*) is a popular tree-search algorithm in the Monte Carlo Tree-Search family. The algorithm is used for finding optimal decisions by using random sampling. Churchill and Buro [3] have successfully applied a variation of the UCT algorithm called *UCT Considering Durations* (UCTCD) to combats in StarCraft. The branching factor of the UCTCD is however extremely large as each unit under control can choose from multiple possible actions. The UCTCD was shown to be beaten by a greedy search algorithm, called *Portfolio Greedy Search* [3], which searches for sequences of scripts instead of individual unit actions. The final sequence of scripts is used to assign actions to units. In this way, the search space can be decreased significantly as long as the number of possible scripts is kept low.

This paper introduces two UCTCD extensions that are aimed to decrease the branching factor, potentially allowing improved control of larger groups of units in StarCraft. First, a novel *script based extension to UCTCD* is introduced. The script-based extension searches for sequences of scripts instead of unit actions in a similar way as the Portfolio Greedy Search [3]. Next, the script-based UCTCD is extended further with a *cluster-based approach*. The idea behind the cluster-based approach is that it is likely to be more efficient to assign actions (or scripts) to groups of units instead of individual units. The insight behind the clustering is that units of the same type and similar position in combat should likely execute similar actions and can therefore be grouped together.

The main conclusion is that a script-based approach can be applied successfully to StarCraft and outperforms the standard UCTCD algorithm. Additionally, the presented results show that clustering can give small improvements in large scenarios while being less effective in small combats. In addition to these two extensions we translated an existing StarCraft combat simulator called SparCraft [4] into the Java programming language to make this research area more accessible.

II. Background

A. StarCraft

StarCraft is a Real Time Strategy (RTS) video game released by Blizzard Entertainment in 1998 and has sold millions of copies worldwide. In StarCraft each player controls one of three different races; Terran, Protoss or Zerg, each with their own strengths and weaknesses. The goal of the game is to eliminate the enemy base and in order to do so each player must gather resources, build buildings and produce units.

RTS games such as StarCraft offer challenging AI control problems, as they are usually characterized by uncertainty, massive branching factors, simultaneous moves and open-endedness [2]. In StarCraft each player takes turn simultaneously, can perform actions in each frame of the game (StarCraft has 24 frames per second) and the game is partially observable [1]. Units in StarCraft have an attack and movement cooldown, which prevents them from taking an action for a certain amount of time after they have moved or attacked. Due to cooldown, actions can not be assigned to every unit in each frame and in some frames it is not possible to assign

actions at all. Units that can be assigned actions in a frame will be referred to as being *ready*.

In StarCraft each player can have up to 200 units in a game including workers that gather resources. Additionally, some combat units are so powerful that they count as several units. Churchill and Buro [3] estimate that the largest armies in a typical game of StarCraft consist of roughly 50 units.

B. AI scripts for StarCraft

The simplest way to control units in StarCraft is script-based without implementing any search or learning techniques. A script normally iterates over every unit under control and based on its overall strategy analyzes nearby enemy units, decides which unit to attack and determines the unit's movement direction. Both retail RTS game's AI and bots in AI competitions use scripted unit behaviors [3]. Following [3], two scripts named *No-Overkill-Attack-Value* (NOK-AV) and *Kiter* are employed by the introduced search algorithms and also serve as the baseline controllers in this paper. These two scripts complement each other well, with NOK-AV implementing a more aggressive behavior and Kiter following a more defensive strategy:

- The *No-OverKill-Attack-Value* script (NOK-AV) [3] assigns commands to attack the enemy unit in range with the highest $\frac{dpf(u)}{hp(u)}$, where $dpf(u)$ is the damage per frame the unit is able to deal and $hp(u)$ is the health points of the unit. NOK-AV additionally makes sure that units do not attack an enemy unit, which already will be dealt lethal damage in the current turn. If enemy units are not in range NOK-AV will command units to move toward the closest enemy unit.
- The *Kiter* script [3] is more defensive as it will command units to move away from enemy units if they are unable to attack in the given frame. If the units are able to attack they will attack the closest unit.

C. MCTS and UCT

Monte Carlo Tree-Search (MCTS) is a method for finding optimal decisions by using random sampling [5]. The algorithm builds up a tree where each node represents a game state. For each possible action in a particular game state a node can be expanded, wherein the new child nodes represent the resulting states. MCTS has the following four steps that are executed sequentially until its time budget is reached:

1) *Selection:* A tree policy is applied to the tree to recursively select the most urgent child node until an expandable and non-terminal node is reached.
2) *Expansion:* The selected node is expanded either fully or partially.
3) *Playout*: A game is played from the selected node to a terminal state using the default policy. Playouts are also called rollouts or simulations.
4) *Backpropagation:* The outcome of the playout is back-propagated to the root node where each node has its value and visit count updated.

The default policy can be a random playout, in which each player simply performs random actions. However, better results can often be achieved by adding domain knowledge [2]. The most popular algorithm in the MCTS family is the *Upper Confidence bounds applied to Trees* (UCT) algorithm [6] which employs the UCB1 formula [7] to balance the search between exploitation and exploration.

$$\mathrm{UCB1} = \overline{X}_j + C_p\sqrt{\frac{2\ln n}{n_j}},$$

where n is the visit count of the current node, n_j is the visit count of the child j, C_p is a constant determining the amount of exploration versus exploitation and $\overline{X}_j$ is the normalized value of child j [7]. In the selection step the child with the highest UCB1 value is the most urgent and will be selected over its siblings.

D. UCT for StarCraft

Controlling units in StarCraft is a multi-agent problem, wherein each player can assign several commands each frame. Commands will be denoted as *actions* and are e.g. $<$*Move unit* a *to position* $p>$ or $<$*Unit* a *attack unit* $b>$. In this context, the term *move* denote a vector of actions. In StarCraft players do not take turns but can make moves simultaneously, which is not supported by the original UCT algorithm. Additionally, the cooldown restriction introduces more challenging *durative* unit actions. In other words, only some units can be assigned actions during a frame while other units can be assigned actions in the next frame.

Churchill and Buro [3] introduced a variation of the UCT algorithm which handles simultaneous and durative actions. The algorithm is called UCT Considering Durations (UCTCD) and was able to beat the NOK-AV script in 100% of their tested scenarios. UCTCD employs *move ordering*, which affects the order of moves to be generated by the search in the expansion phase. First, it assigns a move, which is generated by the NOK-AV script and then a move generated by the Kiter script. When we refer to *a move generated by a script* we mean the sequence of actions which the script would have produced. The two script moves are followed by moves generated by selecting random unit actions. This is however not completely true as the possible actions are ordered so the generated moves will contain more attack actions. Because the NOK-AV and Kiter scripts generate the first two moves, it draws on the advantages of using scripts. It is however only the first two moves that are generated by scripts, while the rest are combinations of random actions.

However, a problem with UCTCD is that it does not explore the search space very well for larger combats. If a player has ten ready units in a frame and each unit can move in eight directions the number of possible moves is 8^{10}. The number is even higher when also considering attack actions for ranged units. An approach to reduce the branching factor was introduced with the MCPlan algorithm by Chung, Buro and Schaeffer [8]. MCPlan can use an arbitrary level of abstraction

such as scripts instead of actions and is thus similar to what we propose in this paper. MCPlan has however only been applied to Capture the Flag scenarios and it does not implement the MCTS method. Thus it differs from UCTCD as it simply runs a single simulation for each generated move and selects the best of these. In this paper we will investigate *an exclusively script-based variation* of UCTCD that is able to assign a specific script to some units while assigning different ones to other units. In the original UCTCD the script generated moves can only assign a script that will be collectively employed by all units.

E. Portfolio Greedy Search (PGS)

A novel hill climbing greedy search algorithm called Portfolio Greedy Search (PGS) was also introduced by Churchill and Buro [3]. This algorithm also performs rollouts to evaluate moves but does not build a search tree and is not part of the MCTS family. PGS is given a game state s, a set of scripts (the portfolio) P, a player to optimize p and a default script D. The algorithm returns a move similar to the UCTCD. PGS can be summarized with the following sequence:

1) *Generate seeds:* Create one vector of scripts of length l for each script in P and fill it with the respective script where l is equal to the number units controlled by p in s. Perform one rollout for each vector of scripts where the enemy is using D. Select the best script as the seed s_p. Compute enemy seed s_e in the same way but against s_p.
2) *Improve self*: For each script in s_p swap the current script with each script in P sequentially and do a rollout. Pick the script that increased the value the most for p. During rollouts use s_p and s_e as scripts for units.
3) *Improve enemy*: Improve s_e similar to step 2 but to minimize the value for p. If more time is available go to step 2 else return s_p.

PGS was shown to beat UCTCD in combat sizes of 16 units and more and has the advantage of applying script-based behaviors to a search. While this paper applies the clustering approach to the UCTCD algorithm it could also be applied to PGS methods.

F. Clustering

A cluster is a set of similar objects that are dissimilar to objects in other clusters [9]. In this paper we introduce an extension to the UCTCD algorithm that implements unit clustering as assigning actions to groups instead of individual units. The hope is that such a cluster-based system should perform well in large combats. Balla and Fern applied UCT with groups of units to Wargus, an open source WarCraft 2 Clone, where each group were able to perform two types of high level actions [10]. The two types of actions were Attack(e) where e is an enemy group to attack and Join(G) where G is a friendly group to join. This method groups units in the beginning of the scenario and only changes them with a Join(G) action. We are more interested in exploring continuous clustering of units.

A popular clustering algorithm is K-Means, which is often used for data analysis to find k clusters in a data set. The algorithm initially selects k random mean points and assigns all data points to the closest of the mean points. The algorithm then iteratively calculates the new mean point of each cluster and reassigns data points to the closest. When the mean points no longer change the clustering is done.

Hierarchical clustering approaches [11] on the other hand build a hierarchy of clusters in the form of a tree with a specific height assigned to each node. When the tree is built any number of clusters (lower or equal to the number of objects) can be extracted by cutting the tree at a certain height. While hierarchical clustering is more versatile, K-Means has better performance with large data sets [9]. In this paper we test the performance of a hierarchical clustering method called UPGMA (Unweighted Pair Group Method using arithmetic Averages) [12] and K-Means to determine their applicability to unit clustering in StarCraft.

III. JARCRAFT

JarCraft is an open-source StarCraft combat simulator we implemented in Java. The project is a faithful translation of the original C++ project called SparCraft, which is written and managed by Churchill [3] [4] but has some minor implementation differences such as the choice of data structures. SparCraft can be imported into existing StarCraft AIs using the BWAPI, a programming interface to StarCraft BroodWar [13]. Thus it can be used by various search algorithms to improve the action selection in combats. By building on JNIBWAPI, a JNI interface to BWAPI [14], this is also possible with an AI written in JAVA using JarCraft.

The motivation for creating JarCraft was to have a StarCraft combat simulator for the Java environment. Our intention is to keep the code-base close to SparCraft to enable students and researchers to choose freely between these tools depending on their programming language preferences.

Similar to SparCraft, it is also possible in JarCraft to setup test scenarios and observe the behaviors of different algorithms with a graphical interface, which includes statistics of the current state of the game. However, certain game features are not implemented in JarCraft or are simplified, enabling the simulator to perform efficiently. JarCraft does not include fog-of-war, highlands, unit collisions and some special abilities. Additionally, units can only move in four directions, while StarCraft supports diagonal moves as well. Experiments in this paper compare algorithms in combats scenarios in JarCraft and not in the actual StarCraft game.

IV. APPROACH

This section first introduces the script-based UCTCD and shows how it can be applied to StarCraft. Section IV-B then focuses on the cluster-based UCTCD extension, detailing how efficient unit clustering can be realized.

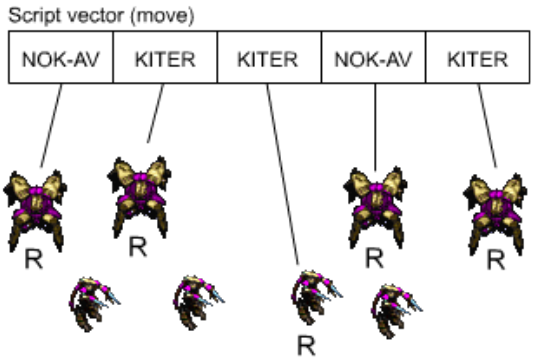

Fig. 1: The script-based UCTCD assigns scripts to ready units only. Ready units are marked with 'R'.

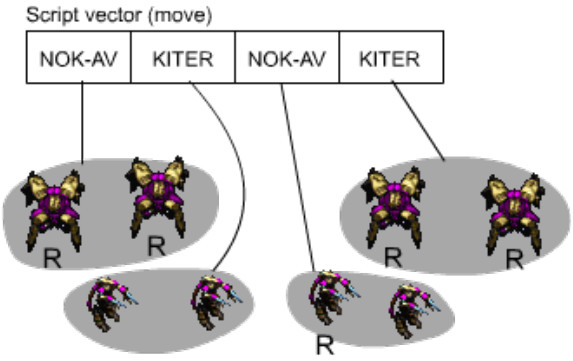

Fig. 2: The cluster-based UCTCD assigns scripts to clusters of units. Ready units are marked with 'R'. Notice how units in a cluster share the same script. In this example scripts are also assigned to clusters that contain non-ready units. In this paper we compare clustering with and without discarding clusters that do not contain ready units.

A. Script-based UCTCD

Group nodes can be employed in UCT algorithms for games with high branching factors and have shown to be successful in Go as well [15]. A group node represents a group of similar moves and thus decreases the branching factor. The branching factor in real-time multi-agent environments, such as in StarCraft, can however be much larger than in Go because of the many possible combinations of actions. Instead of searching for vectors of actions we have altered the UCTCD to search for vectors of scripts similar to the Portfolio Greedy Search. Each script is then used to generate the action of the unit it is pointing to.

The idea of assigning scripts to units is motivated by the advantages of similar approaches in Portfolio Greedy Search and in the move ordering of UCTCD. Move ordering can also be applied to our script-based UCTCD by first generating script vectors only containing one type of script for each unit followed by random combinations of scripts. For example, if we are to generate four moves with the script-based UCTCD for five units using the scripts {NOK-AV, Kiter} with move ordering it could result in the following sequences:

1) {NOK-AV, NOK-AV, NOK-AV, NOK-AV, NOK-AV}
2) {Kiter, Kiter, Kiter, Kiter, Kiter}
3) {NOK-AV, Kiter, NOK-AV, NOK-AV, Kiter}
4) {Kiter, Kiter, Kiter, NOK-AV, NOK-AV}

The script-based UCTCD differs significantly from the original UCTCD as it exclusively searches for sequences of scripts while the UCTCD searches for sequences of actions and only incorporates scripts briefly in the move ordering. A translation function was implemented that, given a gamestate and a set of scripts with one script for each unit, returns a set of actions. This translation function is used when updating the gamestate during the tree traversals. The script-based UCTCD can significantly reduce the branching factor if only a few scripts are employed. If a player has 10 controllable units in a frame and the script-based UCTCD has n scripts the number of possible moves is n^{10} compared to the branching factor of UCTCD which is 8^{10} if only movement actions are considered.

B. Cluster-based UCTCD

A cluster-based variation of the UCTCD algorithm is implemented that first clusters all units into groups. The algorithm first tries to find vectors of scripts similar to the script-based UCTCD but assigns one script to each cluster that will act collectively using the assigned script. Unit clustering must be computationally efficient as a high computation time for clustering will leave less time for searching.

However, clustering the player's units can result in more clusters than the number of ready units in a frame, thereby increasing the search tree branching factor. For example, notice how clusters with non-ready units are assigned actions in Figure 2. Therefore this paper compares clustering with and without discarding clusters that do not contain any ready units.

A potential strategy to make clustering more efficient is to ensure that clusters do not contain units of different types, e.g. clusters will not contain both Protoss Zealots and Protoss Dragoons. As different types of units play different roles in combats this may also lead to better UCTCD generated moves. K-Means and UPGMA can meet this requirement by adding a high value to the distance between two units of different type.

Another issue to consider when choosing between different clustering algorithms is the way the number of clusters is determined. K-Means requires a predetermined number of clusters, while the tree produced by UPGMA can be analyzed further to determine how many splits to perform. We have implemented a dynamic variation of K-Means that determines the number of clusters based on a maximum allowed distance within each cluster mean (see Algorithm 1). The algorithm first splits units by type and then recursively splits each group in two using K-Means until the maximum distance to the mean is less or equal to a specified distance d.

Enemy units can also be clustered when they are assigned moves during the search, but best results were achieved by considering the enemy units as one cluster controlled by the

same script.

Algorithm 1 Dynamic K-Means clustering

```
1: procedure CLUSTER(Unit[] U, Integer d)
2:     Unit[][] K = ∅                          ▷ Clusters to keep
3:     Unit[][] C = SplitByType(U)
4:     for Unit[] c in C do
5:         K.addAll(Split(c, d))
6:     return K
7:
8: procedure SPLIT(Unit[] c, Integer d)
9:     Unit[][] K = ∅
10:    if maxDistanceToMean(c) > d then
11:        Unit[][] S = KMeans(c, 2)           ▷ Split cluster in 2
12:        K.addAll(Split(S[0], d))
13:        K.addAll(Split(S[1], d))
14:    else
15:        K.add(c)
16:    return K
```

V. Experiments

A. Test scenarios

The algorithms are evaluated and compared based on a test scenario created in JarCraft with a map size of 25×20 tiles with a tile size of 32 pixels. The scenario takes one parameter n determining the number of units on each side. Each player in the scenario controls $\frac{n}{2}$ Protoss Zealots (close combat unit) and $\frac{n}{2}$ Protoss Dragoons (ranged combat unit). A realistic battle setup is achieved by first lining up the units vertically by unit type and then changing their position slightly in each direction (randomly choosen between -100 and 100 pixels). Algorithms are tested in this scenario with $n = 4, 8, 16, 32, 48, 64, 80, 96, 112, 128, 144$.

B. Configurations and setup

All experiments are performed on an Intel(R) Core(TM) i7-3517U CPU @ 1.90GHz running Windows 8.1 with 8 GB of DDR3 1600MHz RAM available. All implemented algorithms are single threaded.

- Configurations for all UCT algorithms
  - Time limit: 40 ms
  - Max. children: 20
  - Evaluation: NOK-AV vs. NOK-AV playout
  - Final move selection: Most valuable
  - Exploration constant: 1.6
- UCTCD
  - Child generation: All-at-leaf
- Script-based UCTCD
  - Child generation: One-at-leaf
  - Scripts used: {NOK-AV, Kiter}
- Cluster-based UCTCD (Ready)
  - Child generation: One-at-leaf
  - Scripts used: {NOK-AV, Kiter}
  - Cluster max-distance-to-mean: 30 pixels
  - Opponent clustering: No
  - Units to cluster: Only ready units
- Cluster-based UCTCD (All)
  - Child generation: One-at-leaf
  - Scripts used: {NOK-AV, Kiter}
  - Cluster max-distance-to-mean: 30 pixels
  - Opponent clustering: No
  - Units to cluster: All

In the *All-at-leaf* child generation all children of a leaf node are expanded during the expansion phase, while in *One-at-leaf* only one child is expanded per visit. Following Churchill and Buro [3], the maximum number of children allowed per node was set to 20. The maximum distance to the cluster mean is set to 30 pixels. While this value may seem low it is important to note that simulations in JarCraft currently do not support unit collision, therefore units often stand on top of each other. In games with collision detection this value will likely need to be increased.

C. RandomScript

An implementation of a simple controller, which randomly assigns the NOK-AV script or the Kiter script to units, is implemented as a baseline controller to compare against the script-based UCTCD and cluster-based UCTCD.

VI. Results

This section first presents results on the performance of the different clustering methods, tested under the real-time constraints of StarCraft. Next, the sequence lengths of the different algorithms are investigated and finally each of the algorithms are tested against each other in the described test scenario.

A. Clustering performance

The running times for UPGMA, K-Means and the dynamic K-Means are determined in the test scenario (Figure 4). UPGMA and K-Means are set to find eight clusters, which only affects the running time of K-Means.

When analyzing the different running times, an important factor is the time budget available in each frame, which is 41.6 ms (24 frames per second). While the running time for UPGMA is increasing drastically when the number of units increased, it is still reasonably fast for the usual army sizes in StarCraft. For army sizes of around 140 UPGMA begins to be unsuitable for time requirements of StarCraft as it will leave too little time for the search. As we are interested in a scalable solution UPGMA was not tested with our cluster-based UCTCDs. K-Means is however much better at handling large armies and is able to cluster hundreds of units within a few milliseconds. The dynamic K-Means and K-Means have almost identical running times.

The move length l is the number of units or clusters, which the search algorithm needs to assign actions or scripts to. Table I shows that the cluster-based UCTCD significantly

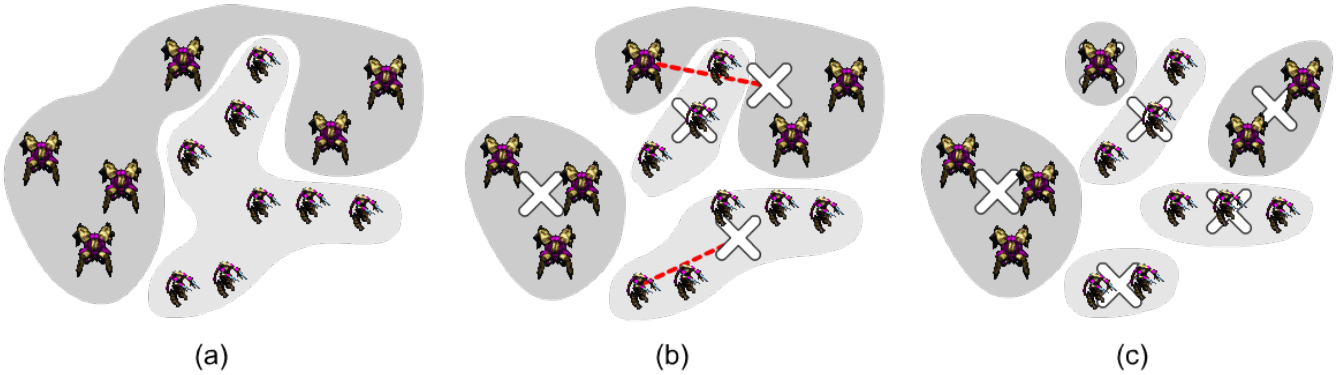

Fig. 3: Dynamic K-Means Implementation. (a) First, units are split into clusters by type, (b) then clusters which have units too far from the cluster mean, shown by the red striped line, are identified and (c) split in two using K-Means until all units are close enough to the cluster mean. The white crosses mark the cluster means.

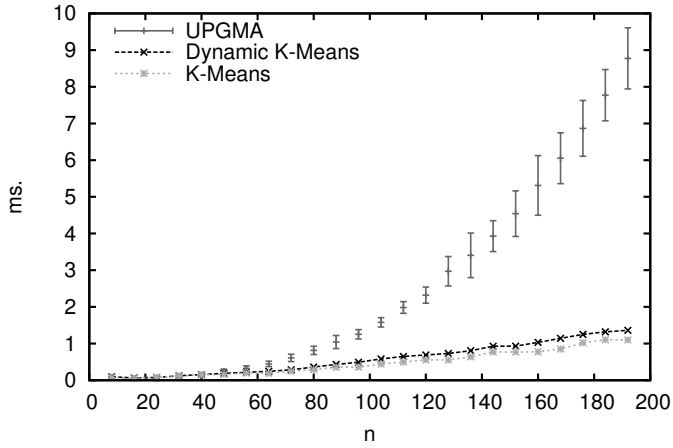

Fig. 4: The average running times for UPGMA, K-Means and the dynamic K-Means with n units ($\frac{n}{2}$ Protoss Zealots and $\frac{n}{2}$ Protoss Dragoons) in 100 runs with a random setup (described in Section V-A). Error bars (only shown for UPGMA) show standard deviation.

	$n = 16$		$n = 64$		$n = 144$	
UCTCD	l	Dev	l	Dev	l	Dev
Script-based	2.24	1.99	5.04	6.75	7.6	13.21
CB (Ready)	1.98	1.22	3.43	3.29	6.18	6.65
CB (All)	6.17	1.70	11.56	6.25	17.40	10.24

TABLE I: Shows move lengths l for script- and cluster-based UCTCDs during a complete game against NOK-AV with n units with the standard deviation. CB is short for Cluster-based.

reduces the average l if only ready units are clustered but is increased if all units are clustered. Furthermore, the high standard deviation for the script-based UCTCD indicates that the number of ready units in some frames is very high which increases the branching factor in the search in these frames. The results also show that l is higher when clustering all units compared to not applying clustering. The reason for this difference is the fact that while some frames only have a few number of ready units, the cluster-based UCTCD assigns actions to all clusters regardless of this property (e.i. the number of clusters is sometimes larger than the total number of ready units). As both the script-based and cluster-based UCTCDs only use two scripts the branching factor is 2^l.

B. Comparison with NOK-AV

The UCTCD algorithm was able to win 100% of the games in all combat sizes against the NOK-AV script (Figure 5). Similar results were reported by Churchill and Buro [3]. However, the script and cluster-based UCTCDs performed worse in very small combats, loosing a few games. In games with $n \geq 32$ all the algorithms won 100% of the time. The RandomScript was also tested but only won 5.2% of the games against NOK-AV and was not tested further.

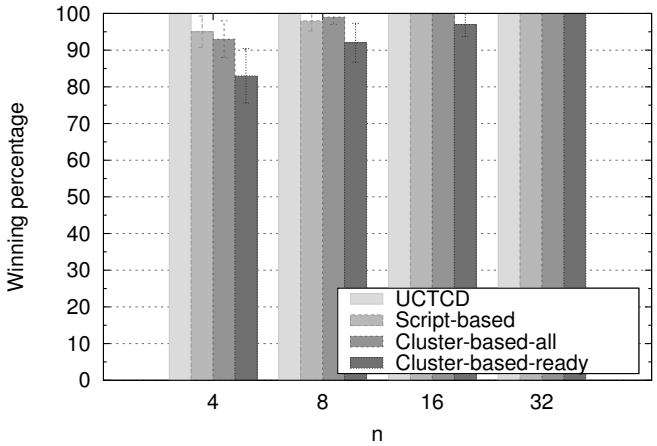

Fig. 5: The winning percentages of the different algorithms in 100 games for each combat size against the **NOK-AV script** where n is the number of units on each side. Error bars show 95% confidence intervals for each experiment.

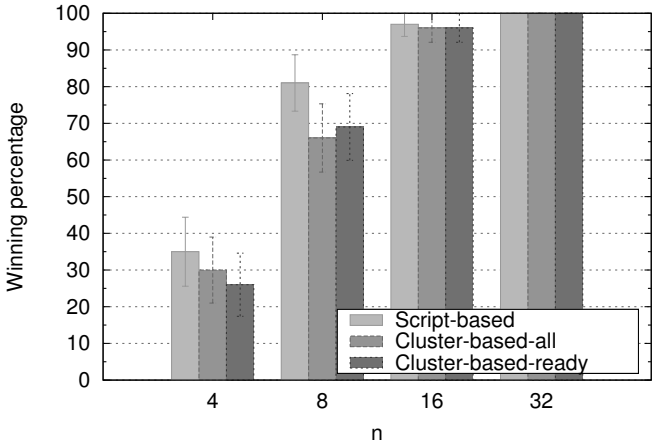

Fig. 6: The winning percentages of the script- and cluster-based UCTCDs in 100 games for each combat size against **UCTCD** where n is the number of units on each side. Error bars show 95% confidence intervals for each experiment.

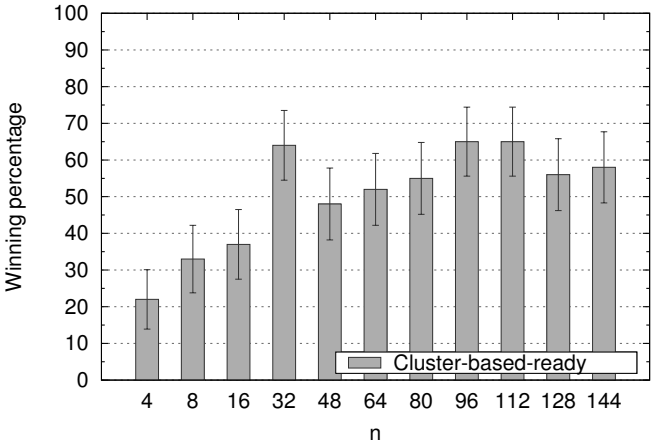

Fig. 7: The winning percentages of the **cluster-based UCTCD (Ready)** in 100 games for each combat size against the **script-based UCTCD** where n is the number of units on each side. Error bars show 95% confidence intervals for each experiment.

C. Comparison with UCTCD

The script- and cluster-based UCTCDs were compared against the original UCTCD and won 100% of the games where $n \geq 32$ (Figure 6). However, UCTCD showed the best performance in very small combats of only four units. It is also noticeable that the cluster-based UCTCDs perform worse than the script-based UCTCD in small combats.

D. Cluster-based UCTCD vs. script-based UCTCD

The cluster-based UCTCDs were compared against the script-based UCTCD to investigate the effect of using unit clustering (Figure 7 and 8). The results do not show a clear overall winner but the script-based UCTCD wins more games if $n \leq 16$ and the cluster-based UCTCD wins more games if $n \geq 64$. Additionally, the results indicate that the cluster-based UCTCD that clusters all units wins more than the cluster-based UCTCD that clusters ready units only.

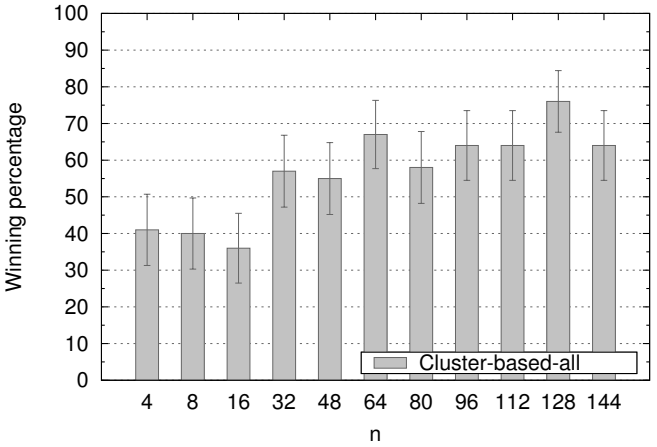

Fig. 8: The winning percentage of the **cluster-based UCTCD (All)** in 100 games against the **script-based UCTCD** where n is the number of units on each side. Error bars show 95% confidence intervals for each experiment.

VII. Discussion and Future Work

JarCraft was used as a test bed for the implemented algorithms and was able to run and visualize abstract StarCraft combats with hundreds of units per side. Additionally, it is able to support rollouts for the UCTCD implementations. JarCraft still requires further work and our intention is to keep working on JarCraft to make this research area more accessible. The source code can be accessed at [16].

The UCTCD algorithm by Churchill and Buro [3] was implemented in JarCraft and was able to beat the NOK-AV script in 100% of the tested scenarios. The implementation is identical to the original C++ implementation [4] besides differences in the final move selection. The original configuration is to select the move of the most visited node [3]. While this configuration was tried in initial experiments, it resulted in all child nodes from the root having a visit count of one if the number of iterations is below 20, therefore not allowing the determination of the best next move. By selecting the node with the highest value, the search also performs well with more than 100 units. As this change was made to all the UCTCDs, including the script- and cluster-based extensions, they are still compared fairly. Depending on the number of units in the game, the algorithms were usually able to make between five and 100 iterations during the 40 ms.

The introduced script-based UCTCD was able to beat the UCTCD in every combat with 32 units or more, while the UCTCD was the best algorithm with four units. In fact with a low amount of possible moves it is likely optimal to search the moves directly instead of combinations of scripts. While, some strategies are simply not considered when applying a script-based approach, the chances for UCTCD to generate good moves decreases when the number of units increases. Churchill

and Buro [3] showed that using scripts in Portfolio Greedy Search instead of actions allows the algorithm to significantly decrease the size of the search space. The results presented in this paper demonstrate that this method can successfully be applied to the UCT algorithm as well. In the future it will be interesting to also apply these script-based approaches to other games with very large branching factors. The algorithm seems to be especially applicable in games with enormous numbers of possible actions but with a set of known strategies.

While another initial goal of the presented investigation was to also compare the script-based UCTCD with the Portfolio Greedy Search, a complete PGS implementation that includes the optimizations of the original C++ implementation was not achieved to run with JarCraft. Thus a fair comparison could not be made and remains an important goal for future work.

The presented results show that K-Means is well suited to cluster units within the time requirements of StarCraft and can also be adjusted to dynamically split a large number of units into clusters. Additionally, tests with another popular hierarchical clustering method called UPGMA, revealed that it also meets the time requirements of StarCraft (e.g. clustering up to 100 units in less than 2 ms). The clustering time does however rise drastically as the number of units increase and is thus not scalable.

An important insight is that the cluster-based UCTCD does not perform well in small combats. Analyzing the clustering during the game revealed that with few units the algorithm often groups all units in one cluster, thereby making clever maneuvering harder. The clustering method could possibly be improved to have a lower limit on the number of clusters. The cluster-based UCTCD improves in larger combats and is able to beat UCTCD in combats with 32 or more units. The presented results show that UCTCD with unit clustering can be applied successfully to decrease the branching factor if only ready units are considered. One key experiment presented in this paper was testing the cluster-based UCTCD against the script-based UCTCD. It is clear that the script-based UCTCD is the best choice for small combats while the cluster-based UCTCD wins slightly more games in larger combats. Surprisingly, the cluster-based UCTCD performed best by clustering all units instead of ready units only.

Further exploring clustering for UCT is an important next step. Also applying it to simpler domains could allow us to better understand its impact on the search and how it can and should be configured to achieve best results.

VIII. Conclusion

In this paper we presented two extensions to the UCT Considering Durations (UCTCD) algorithms and applied them to unit control in StarCraft using the StarCraft combat simulator JarCraft (a Java translation of the original C++ package SparCraft). The first extension is script-based approach, as it searches for sequences of scripts instead of unit actions. The second extension is cluster-based as it also searches for sequences of scripts but this time assigns them to clusters of units instead of individual units. A K-Means clustering approach was shown to be able to efficiently cluster a large set of units and to automatically determine the number of clusters in an army. Both the script- and cluster-based extensions were able to beat the standard UCTCD in 100% of the tested scenarios with 32 units or more. We believe that script-based UCT can be applied successfully to other games with massive branching factors as well. The cluster-based extension was tested against the script-based extension and won only 20-40% of the games in scenarios with fewer than 16 units while the cluster-based extension is better in combats of 64 or more units. We suggest that further investigation of the behavior of cluster-based UCT is needed to understand its impact on the search.

References

[1] 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," *IEEE Transactions on Computational Intelligence and AI in games (2013)*, pp. 293–311, 2013.

[2] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012.

[3] D. Churchill and M. Buro, "Portfolio greedy search and simulation for large-scale combat in starcraft," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE, 2013, pp. 1–8.

[4] D. Churchill, "SparCraft," https://code.google.com/p/sparcraft/, 2013, *Online – accessed 10-December-2013*.

[5] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *Computers and games*. Springer, 2007, pp. 72–83.

[6] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*. Springer, 2006, pp. 282–293.

[7] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[8] M. Chung, M. Buro, and J. Schaeffer, "Monte carlo planning in rts games," in *CIG*, 2005.

[9] O. A. Abbas, "Comparisons between data clustering algorithms." *International Arab Journal of Information Technology (IAJIT)*, vol. 5, no. 3, pp. 320–325, 2008.

[10] R.-K. Balla and A. Fern, "Uct for tactical assault planning in real-time strategy games." in *IJCAI*, 2009, pp. 40–45.

[11] R. Xu and I. Wunsch, D., "Survey of clustering algorithms," *Neural Networks, IEEE Transactions on*, vol. 16, no. 3, pp. 645–678, May 2005.

[12] Y. Loewenstein, E. Portugaly, M. Fromer, and M. Linial, "Efficient algorithms for accurate hierarchical clustering of huge datasets: tackling the entire protein space," *Bioinformatics*, vol. 24, no. 13, pp. i41–i49, 2008.

[13] various, "BWAPI," http://code.google.com/p/bwapi/, 2013, *Online – accessed 10-December-2013*.

[14] ——, "JNIBWAPI," http://code.google.com/p/jnibwapi/, 2013, *Online – accessed 10-December-2013*.

[15] J.-T. Saito, M. H. M. Winands, J. W. H. M. Uiterwijk, and H. J. van den Herik, "Grouping nodes for monte-carlo tree search," in *Computer Games Workshop*, 2007, pp. 276–283.

[16] B. Tillman, "JarCraft," https://github.com/tbalint/JarCraft, 2014.