# Integrating Search and Scripts for Real-Time Strategy Games: An Empirical Survey

## Abstract

Real-time strategy games are a challenging problem from an AI point of view. Specially, they are particularly hard for tree search algorithms due to their combinatorial branching factors the limited amount of time available to choose actions. As the community grows and many RTS game competitions are held, much work has been done in the direction of integrating hand-authored scripted bots into tree search algorithms, with the goal of making search tractable. In this paper, we survey a collection of representative algorithms that integrate scripts into search or planning algorithms. Then we compare them empirically in the $\mu$RTS environment and examine the trade-offs for designing such algorithms. We also discuss the potential future work in this direction of research and connections to other types of algorithms.

## Introduction

Real-time strategy (RTS) games provide rich and challenging testbed for AI techniques (Buro 2003; Ontanón et al. 2013). With the aid of growing computing power, the combination of deep learning and reinforcement learning has shown great ability in modeling and building game playing agents for complex games, such as AlphaGo (Silver et al. 2016; 2017), OpenAI Five (OpenAI 2018), and AlphaStar (Vinyals et al. 2019). However, human expert knowledge can also play a key role in developing game playing agents for RTS games, partially due to the hardness of exploration in the very large state spaces of these domains, in which deep reinforcement learning or tree search algorithms struggle to perform well when used independently. For example, in the development of AlphaStar, imitation learning from human gameplay was needed to initialize reinforcement learning.

On the other hand, tree search algorithms have a long successful history of building game playing agents since the invention of the minimax algorithm. As complete tree search algorithms fail to scale to complex games like Go, sampling-based algorithms like Monte Carlo Tree Search (MCTS) (Kocsis and Szepesvári 2006; Browne et al. 2012) were proposed. Integrating MCTS with deep learning models, AlphaGo was able to achieve superhuman level of strength in Go. However, RTS games pose an even greater challenge in terms of the size of the state and action space, as well on the limitations on computation time available to make decisions.

In the context of RTS games, in this paper we focus on tree search algorithms, and specifically on techniques to integrate search with human expert knowledge, in the form of hand authored scripts, in order to scale search up to the size of RTS game search spaces. Thanks to the many AI competitions on RTS games, such as AIIDE StartCraft competition (Buro and Churchill 2012) and IEEE-CoG $\mu$RTS competition (Ontañón et al. 2018), many methods for integrating scripts into different AI methods have been developed. As a result, combining tree search methods and scripts has become a common approach to building strong entries in competitions. However, despite the growing body of work, we lack a good understanding of what are the different trade-offs of these approaches. For example, some approaches employ scripts to prune the search space (only allowing tree search to choose between the moves proposed by a collection of scripts), other approaches develop scripts with decision points, where search is used, and yet other approaches use scripts just to bias the search process. We could even generalize the concept of script to consider probabilistic policies, and even include AlphaGo's approach in this framework. In this paper, we compare all of these approaches in equal grounds, to try to start building an understanding of this space of game playing algorithms.

The rest of the paper is structured as follows: we first introduce the background of RTS games, the experimental environment used in this paper, and the application of MCTS in RTS games. Then we survey a collection of different algorithms that integrate scripts into tree search, followed by an empirical evaluation of the algorithms in the $\mu$RTS environment. After that, we discuss the similarities and differences of these algorithms. And we conclude the paper with conclusions and directions for future work.

## Background

RTS is a sub-genre of strategy games where players aiming to defeat their opponents (destroying their army and base) by strategically building an economy (gathering resources and building a base), military power (training units and researching technologies), and controlling those units. From an AI point of view, the main challenges of RTS games with respect to traditional board games like Chess or Go are: (1) the

Copyright © 2019, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

combinatorial growth of the branching factor as a function of the number of units being controlled (Ontanón 2017), (2) limited computation budget between actions due to the real-time nature, (3) they are simultaneous move games (more than one player can issue actions at the same time) with durative actions (actions are not instantaneous), and (4) lack of forward model in most of research environments like Starcraft (necessary to perform lookahead search). Additionally, some RTS games are not fully observable or not deterministic.

Because of these reasons, RTS games have been receiving an increased amount of attention as they are an excellent platform to study the aforementioned problems. Many research environments and tools, such as TorchCraft (Synnaeve et al. 2016), SCIILE (Vinyals et al. 2017), $\mu$RTS (Ontanón 2017), ELF (Tian et al. 2017), and Deep RTS (Andersen, Goodwin, and Granmo 2018) have been developed to promote research in the area. Specifically, in this paper, to stay focused on the problem of interest of this paper, we chose a single evaluation platform, $\mu$RTS, as our experimental domain, as it offers a forward model for game tree search approaches such as minimax or Monte Carlo Tree Search, and many of the algorithms to be compared were already readily available in this platform.

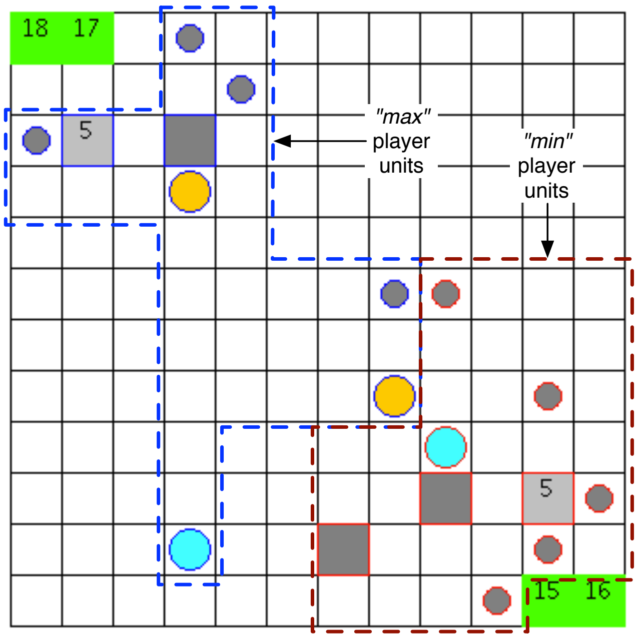

Figure 1: A Screenshot of $\mu$RTS.

## $\mu$RTS

$\mu$RTS[1] is a simplified RTS game designed for AI research. $\mu$RTS provides the essential features that make RTS games challenging from an AI point of view: simultaneous and durative actions, combinatorial branching factors and real-time decision making. The game can be configured to be partially observable and non-deterministic, but those settings are turned off for all the experiments presented in this paper. In the default configuration, $\mu$RTS defines only four unit types and two building types, all of them occupying one tile in the map, and there is only one resource type. Additionally, as required by our experiments, $\mu$RTS allows maps of arbitrary sizes and initial configurations. The default unit types are:

- Base: can train Workers and accumulate resources.
- Barracks: can train attack units.
- Worker: collects resources and construct buildings.
- Light: low power but fast melee unit.
- Heavy: high power but slow melee unit.
- Ranged: long range attack unit.

Additionally, the environment can have walls to block the movement of units. A example screenshot of game is shown in Figure 1. The squared units in green are Minerals with numbers on them indicating the remaining resources. The units with blue outline belong to player 1 (which we will call *max*) and those with red outline belong to player 2 (which we will call *min*). The light grey squared units are Bases with numbers indicating the amount of resources owned by the player, while the darker grey squared units are the Barracks. Movable units have round shapes with grey units being Workers, orange units being Lights, yellow being Heavy units (now shown in the figure) and blue units being Ranged.

[1]`https://github.com/santiontanon/microrts`

## Monte Carlo Tree Search for RTS Games

Monte Carlo Tree Search (MCTS) is a search algorithm designed for domains with large action spaces. The main characteristic of MCTS is that sampling the search tree (rather than systematically exploring it as minimax does). If this sampling is done properly (e.g., following a suitable exploration/exploitation balancing policy), in the limit, the MCTS decision converges to the optimal minimax decision. There are four stages in a standard MCTS procedure, explained as follows:

- Selection: In this stage, the algorithm employs an tree policy to select the most urgent node in the tree that still has children that are not part of the tree. One of the most important design decisions to make when choosing the tree policy is to balance exploration and exploitation.
- Expansion: When a leaf node is selected at the Selection stage, we expand the tree by adding a child node to the selected node.
- Simulation: Then we estimate the value of the newly added node by a simulation policy (also called rollout policy or playout policy).
- Backpropagation: Finally we backpropagate the outcome of the simulation and update the statistics of the nodes visited in this iteration of MCTS.

One of the most popular algorithm in the MCTS family is the Upper Confidence Bound Applied to Trees (UCT) (Kocsis and Szepesvári 2006). UCT models the node selection as a multi-armed bandit problem and uses the UCB1 (Auer, Cesa-Bianchi, and Fischer 2002) formula to balance exploration and exploitation. The UCB1 formula choose the arm the maximizes the following score:

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

where $\overline{X}_j$ is the average reward of arm $j$, $n$ is the total number of trials, and $n_j$ is the number of trials on arm $j$. The first term dictates exploitation, the second term dictates exploration, and $C_p$ is a parameter that can be used to tune the degree of exploration in practice.

UCT achieved many successes in complex domains like computer Go. However, in the domain of RTS games, it is unfeasible to apply UCT, as the UCB1 policy requires exploring each child of a node at least once before it starts balancing exploration/exploitation, and in RTS games, there might be more children of the root node than we have computation budget to explore (Gelly and Wang 2006).

In order to apply MCTS to RTS games, approaches typically either simplify the search space by defining an abstracted action space (Balla and Fern 2009), by using scripts to inform the search (as we survey in this paper), or devise specialized multiarmed bandit strategies that can deal with the combinatorial branching factors. For example, the Naïve Sampling multiarmed bandit strategy models the problem as a combinatorial multi-armed bandit (CMAB), where each unit being controlled in the game is considered as a variable, and the problem is to select the best assignment of values to a set of variables. The MCTS variation that employ Naïve Sampling as the tree policy is called NaïveMCTS (Ontanón 2017), and is used as the base search algorithm in several of the techniques we survey in this paper. In the remaining of this paper, we will focus on surveying those techniques that employ scripts to help MCTS scale up to the complexity of RTS games.

## Combining Script and Search Algorithms

This section briefly describe the main approaches that have been proposed in the literature to integrate scripts with tree search.

### Portfolio Search

The most basic strategy to integrate scripts and search is usually known as *portfolio search*. The key idea is to have a collection of $n$ scripts $S$, where each script is a complete hand-authored agent capable of playing the game.

Portfolio search considers a search space where the only choice is which script to execute. Thus, the most straight forward way of implementing this idea, is to consider all $n \times n$ possibilities of player 1 and player 2 executing each of the scripts, run simulations for each possibility (this results in a payoff matrix, akin to those in standard matrix games like the prisoner's dilemma), and select the script that in average results in the highest reward against all possible scripts by the opponent.

An early instantiation of this strategy for RTS games was MCPlan (Chung, Buro, and Schaeffer 2005).

### Portfolio Greedy Search

Churchill and Buro (Churchill and Buro 2013) introduced an anytime algorithm designed for RTS combat micro management called Portfolio Greedy Search (PGS). PGS combines scripts with local search algorithm to improve upon scripts. PGS has the following characteristics:

- Compared to Portfolio Search, PGS considers unit level scripts (i.e. each script controls a single unit), and thus the search space is significantly larger (assignment of scripts to individual units, rather than to whole players).
- PGS starts by assigning a "default" script to each unit (both friendly and opponent).
- It then tries to improve the scripts assignment of the friednly units one at a time, using hill-climbing, and once it has optimized the friendly scripts, it switches to optimizing the opponent scripts (to find a best response against the scripts assigned to friendly units). It then keeps alternating optimizing friendly and opponent units until it runs out of time.
- Thus, it does not apply tree search but relies on hill-climbing to make search tractable.
- In order to evaluate each script assignment to units, it uses simulations.

### Puppet Search

Barriga et al. (Barriga, Stanescu, and Buro 2015) proposed Puppet Search, which combined scripts with look-ahead search. The basis of the algorithm is a hand-authored non-deterministic script, which completely defines the behavior the player should follow, except for some "choice points". Thus, the search space is defined by the set of choice points exposed by the script. Thus, as the authors described, Puppet Search works like a puppeteer that controls the limbs (choice points) of a puppet (the script).

We can control the branching factor by controlling the number of choice points exposed by the script, since the branching factor now grows with respect to the number of choice points instead of number of units. Puppet search, is thus related to the early idea of adaptive Lisp (ALisp) (Marthi, Russell, and Latham 2005), where a script with choice points was defined and reinforcement learning was used to learn a policy for those choice points.

### Script-Based UCT

Justesen et al. (Justesen et al. 2014) proposed a script-based extension to UCT that searches for sequences of scripts instead of actions. Instead of search in the full action space, script-based UCT only considers the actions proposed by the scripts. The advantage of searching in the space of actions proposed by the scripts is that the branching factor is vastly decreased. Specifically, the branching factor now grows as a function of the number of scripts, rather than the number of possible raw actions units can perform. For example, if we are controlling, say 10 units, and each unit can perform 8 different moves, but we only have 4 scripts. The branching factor goes from $8^{10}$ for vanilla UCT to at most $n^{10}$ for script-based UCTCD (but it can be lower, since if more than one script propose the same action, then the search space is even smaller).

Due to the durative action in RTS games, a variation of UCT algorithm called UCT Considering Durations (UCTCD) is used.

In the implementation used in this paper, we used player-level scripts (i.e. scripts that can play the whole game, like those used in Portfolio Search) to propose actions, and UCTCD is used to select one among those actions at each choice point.

## Action Abstractions for Monte Carlo Tree Search

Search only in the space proposed by scripts can help largely reduce the search space. However, it also inevitably reduces the flexibility and adaptability of the resulting algorithms. Moraes and Lelis proposed Asymmetric Action Abstraction (Moraes and Lelis 2018) and combined it with NaïveMCTS in their later work (Moraes et al. 2018).

The main idea behind action abstractions is to reduce the number of legal moves, by using the scripts, of only a subset of the units; legal moves of the other units remain unchanged. The authors proposed three variations of the algorithm: A1N, A2N, and A3N.

- A1N is the baseline algorithm and it searches only in the unit actions proposed by the set of script. Thus, it is basically the same as script-based UCT, but using NaïveMCTS instead of UCTCD.
- A2N allows NaïveMCTS to search in an asymmetrically-abstracted action space defined by two sets of scripts, one for economy units and one for combat units.
- Finally, A3N, further relaxes the prunning of the search space, and some units are marked as "unrestricted", and the full action space of those is considered during search. A heuristic function is used to determine which units are unrestricted, such as considering the unit closest to the enemy units to be unrestricted, and all the rest to be restricted by scripts.

## Guided Naïve Monte Carlo Tree Search

Guide Naïve Monte Carlo Tree Search (GNS) (Yang and Ontanón 2019) is another work on combining MCTS and scripts. The key characteristic of this approach is that it uses scripts only to guide the exploration of the tree in MCTS, instead of for pruning the search space, as had been done in other approaches discussed in this paper. In this way, GNS preserves the original search space of MCTS unchanged, and under certain conditions, the resulting MCTS algorithm will still converge to the optimal action (the one that regulat MCTS would have found in the limit if scripts were not used).

The authors proposed two variations of the algorithm: First Choice GNS (FC-GNS) and multi-script GNS ($k$-GNS). FC-GNS works by incorporating only one script and force select the script-provided action only when a new node is added to the tree to bias the search. The intuition is that for nodes that are visited only once or few times, we want to use the script to select the action, in order for this selection to be representative (rather than being a random action). $k$-GNS allows the node expansion process to select from a pool of scripts to increase variety. Additionally, with a factor of $\epsilon$, the tree policy is forced to choose from the pool of actions proposed by the scripts when at a non-leaf node. The parameter $\epsilon$ enables tuning of the level of script bias introduced to the search.

## Adversarial Hierarchical-Task Network

Hierarchical-Task Network (HTN) (Erol, Hendler, and Nau 1994) planning is a planning algorithm that creates plans by decomposing tasks into smaller and smaller tasks. Adversarial hierarchical-task network (AHTN) (Ontanón and Buro 2015) combines the minimax game tree search algorithm with the HTN planning. There are two types of tasks in HTN planning: primitive tasks (low level), and non-primitive tasks (high level). And the algorithm iteratively decompose the non-primitive tasks until all leaves of the HTN tree are primitive tasks. AHTN works by creating a search tree that alternates between max and min player, like the minimax tree fashion, and each node keeps track of the game state and the execution state of the HTN plan for max and min players.

We consider AHTN to be a form of integration of scripts and tree search, as the tasks in the HTN definition of the domain can be created in such a way that they resemble scripts with a few choice points (similar to Puppet search).

## Empirical Evaluation

To evaluate the performance of the algorithms described before, we designed a round-robin experiment to test the algorithm against each other in terms of the game play strength. To fairly evaluate those algorithms, we separate the experiments into two tracks (*standard* and *competition*), both of which are run with fixed time budget and in three maps of increasing size. The standard track includes the standard implementation of all the algorithms, without any special fine-tuning of parameters. However, some of these algorithms, such as GNS and A3N have versions that were optimized for the IEEE-CoG $\mu$RTS competition. In order to compare the full potential of these algorithms when optimized, we compare these separately in the competition track. As the available implementation of Puppet Search has only a competition version available, the same version will be evaluated in both tracks.

Specifically, the three map sizes are $8 \times 8$, $32 \times 32$, and $128 \times 112$ [2], and the time budget each player has at each decision cycle is 100 milliseconds. Games are cut-off at 3000, 6000, and 12000 cycle respectively for the three map sizes and recorded as a draw if they reach this length. We chose these three sizes because they represent games of different complexities, and we anticipate that algorithms that perform well in the small map do not necessarily perform well in larger maps, or vice versa. The agents play against each other in a round-robin manner, and 10 and 20 full round-robin iterations are run for standard and competition track respectively. Thus, each agent plays 120 games in each map

[2]Specifically, the maps used are *basesWorkers8x8A*, *basesWorkers32x32A*, and *BroodWar/(2)Benzene.scxA*.

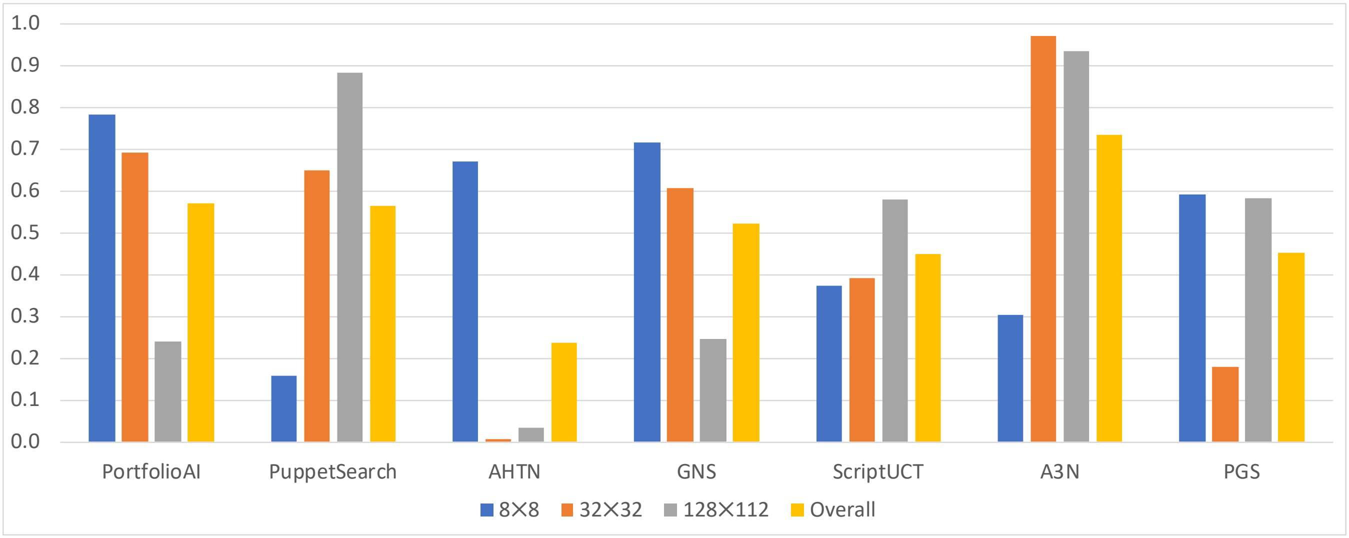

Figure 2: Win rate comparison of the standard track in maps of size 8×8, 32×32, 128×112, and overall.

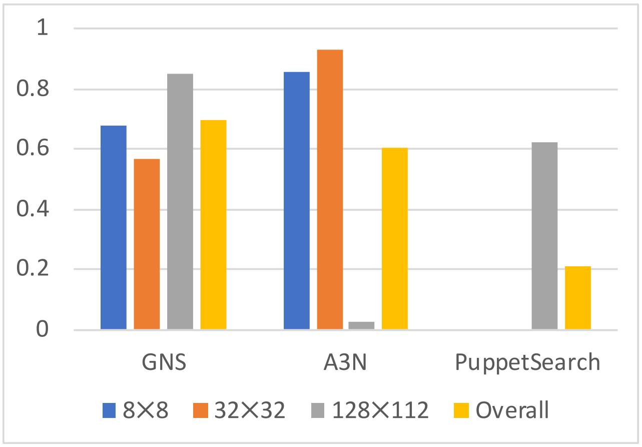

Figure 3: Win rate comparison of the competition track in maps of size 8×8, 32×32, 128×112, and overall.

in the standard track and 80 games in each map in the competition track.

The results are shown in Figure 2 and Figure 2 for standard and competition track respectively. In the 8×8 map, the best performing algorithms are Portfolio search (78.3% win rate), GNS (71.7% win rate), and AHTN (67.1% win rate). In the 32×32 map, the best performing algorithms are A3N (97.1% win rate), Portfolio search (69.2% win rate), and Puppet search (65.0% win rate). And in the 128×112 map, the best performing algorithms are A3N (93.3% win rate), Puppet search (88.3% win rate), and PGS (58.3% win rate). Overall, the best performing algorithm combining all three mas is A3N, with a 73.6% win rate.

Also, we would like to point out that there is one result (PGS for 32×32 maps) that seems to be an outlier. It happens to be that given the start configuration of this map, when PGS plays as player 1 in this map, it builds units in such a way that it blocks its own resource gathering path, thus getting stuck. This is a pure coincidence, and as part of our future work, we will evaluate the algorithms using a larger variety of maps, in order to avoid these experimental artifacts.

We observe that almost all algorithms perform well at one end of the map complexity and not so well at the other end of the map complexity. The main trade-off to make here is the pruning of the action space. For agents that perform well in smaller maps like AHTN and GNS, they preserve more flexibility for the search algorithm to explore. And in 8×8 maps, such exploration is feasible. However, as the rapid growth of the game complexity, such exploration becomes unfeasible and thus we observe the performance drop. On the other hand, for algorithms that prune the search space aggressively, such as A3N and Puppet Search, they do not perform well in the small map due to lack of flexibility, but are able to handle and scale to larger maps with much greater complexity.

For the competition track, we want to point out that the 128×112 is a map size much larger than the largest map used in the IEEE-CoG $\mu$RTS competition ($64 \times 64$). GNS and A3N both use the configuration optimized for the IEEE-CoG $\mu$RTS competition. We observe that for $8 \times 8$ and $32 \times 32$ maps, A3N has the best performance, with win rates of 85.6% and 93.1%. For the 128 map, GNS is the best performing algorithm, with 85.0% win rate. Puppet Search cannot win a single game in $8 \times 8$ and $32 \times 32$ maps, but it has a 62.5% win rate in the $128 \times 112$ map. This is because in smaller maps, the constrainted search space of PuppetSearch is a disadvantage that can be exploited by the other algorithms. Finally, the consistent performance of GNS indicates that it is beneficial for the search algorithm to preserve the full search space in scenario that needs to recover from the mismatch between the configuration and the environment.

Table 1: Comparison Summary on the Algorithms

| | Search | Role of Scripts | Search Space | Optimality | Multi-scripts |
|---|---|---|---|---|---|
| Script. UCT | MCTS | Define unit action space | Assign scripts to units over time | Script space | Yes |
| Puppet S. | MCTS/minimax | Script choice points | Abs. actions space | Script space | No |
| PGS | Local Search | Define unit action space | Assign scripts to units | No guarantee | Yes |
| Portfolio S. | Grid Seearch | Define player action space | Choose a script | Script space | Yes |
| GNS | MCTS | Bias search | Full action space | Yes | Yes |
| A3N | MCTS | Define unit action space for restricted units | Some units restricted to choosing a script | Script space | Yes |
| AHTN | HTN+minimax | Define search space | Defined by the HTN definition | Script space | Yes |

## Discussion

In this section, we discuss the differences and similarities between the algorithms. Then we discuss and relate them with other methods of incorporating human knowledge with search such as machine learning models.

### Comparison of Algorithms

We compare the algorithms in from several aspects, including the type of search algorithm used, the role played by the scripts, the search space, whether the search is optimal in the limit, and whether there is support of multiple scripts. A summary is shown in Table 1.

**Search Algorithm** In script-based UCT, Puppet Search, GNS, and A3N, Monte Carlo Tree Search is applied. Specifically, script-based UCT uses standard UCT, while other UCT enhancements such as FPU and PUCB can potentially be applied. For puppet Search, complete tree search algorithm like minimax can also be applied, but presumably not as good performing as MCTS when number of choice point is large or action space is large. As for A3N and GNS, they also use MCTS. But instead of UCT, NaïveMCTS is employed, which has shown to have better performance than UCT in domains with combinatorial branching factors.

Portfolio greedy search uses hill-climbing style local search to alternatively update the max player and min player. The local search algorithm has no optimality guarantees and could get stuck in loops and local minima, though it achieved good empirical performance in the domain of StarCraft. On the other hand, portfolio search performs a grid search among all the script proposed player actions, which in the long run should find the best one.

AHTN integrates HTN planning into game tree search and it allows creating domain configurable planners that do not have to explore the whole combinatorics of the task, but instead choose from a reduced set of methods, scripts in our case, to achieve each task.

**Role of Scripts and Search Space** Puppet search uses the pre-defined choice points in the script to define the search space. The reduced search space is referred to as action abstractions, and the action abstraction is symmetric because the same script is used to construct the set of possible actions for all units.

In both script-based UCT and A3N, the scripts are used to prune the search space for MCTS. The key difference is that in the original script-based UCT, the same set of scripts are used for all units. Hence the action abstraction in script-based UCT in symmetric. Nonetheless, one can extend script-based UCT to asymmetric action abstraction by use different sets of script for different units. The A3N algorithm extends this idea by allowing unrestricted units, whose action space is not limited by the scripts but span the full space of legal actions.

Similarly, PGS also uses scripts to prune the search space (i.e., to define an action abstraction). The difference is that PGS has unit-level scripts for each type of units. Also, from the point of view of action abstractions, standard PGS has symmetric action abstraction since it uses the same set of scripts for all units. Portfolio search has the simplest search space, which is just choosing which script to use. It performs a pair-wise grid search to find the best script to commit to. AHTN also uses scripts to prune the search space, searching only in the space of actions allowed by the HTN definition.

Unlike the algorithms mentioned above, GNS only uses the script to bias the tree search. Specifically, GNS prioritizes the actions proposed by the scripts and then grows the tree around those actions without pruning the search tree. Thus the search space of GNS is the full action space.

In summary, we can see two general uses of scripts: to prune the search space, or to bias the search process. We also see that scripts can be defined as the unit level (a script controls a single unit) or at the player level (the script controls all the units simultaneously). We call these two “unit actions” and “player actions”.

**Optimality** If an algorithm searches only in the reduced search space defined by the scripts, there is no guarantee that the optimal action(s) are actually present in the search space. Thus, all of the algorithms but GNS necesssarily sacrificy any optimality guarantees. By using MCTS in the pruned action space, script-based UCT, Puppet Search, and A3N only achieve optimality only with respect to this pruned action space. PGS, with the local search scheme, has no optimality guarantees. Finally, GNS preserves the global optimality guarantee of MCTS since the search space is not pruned, and it is easy to show that in the limit, it would converge to the same solution as standard MCTS.

**Multi-script Support** Allowing for integrating more than one script into the search can add variety to the search space and potentially yield more robust behavior. The multi-script support of the surveyed algorithms is described as follows:

- Scrip-based UCT: Multiple scripts are supported. If the script is deterministic, then multiple scripts are mandatory for scrip-based UCT to function.

- Puppet Search: Multiple scripts are not supported. Puppet Search works a single script with one or more choice points to apply search.
- PGS: Multiple unit level scripts for each type of units are mandatory for PGS to function.
- Portfolio Search: Multiple scripts are mandatory.
- GNS: Multiple scripts are supported.
- A3N: Multiple scripts are supported.
- AHTN: Multiple scripts are suported, defined in the form of HTN task decompositions.

**Relation to Machine Learning Approaches** Some approaches, such as AlphaGo (Silver et al. 2016) integrate stochastic policies (trained via supervised and/or reinforcement learning) into tree search. In order to do so, AlphaGo employs a tree policy related to PUCB (Rosin 2011), which can take into account the probability distribution of actions provided by the policy to guide the search, effectively focusing the search. The scripts considered in this paper can be seen as deterministic policies (where one action has probability 1, and the rest have 0). GNS, for example, uses a mixing factor $\epsilon$ that determines how often to follow the script in any iteration other than the first (where the script is always followed). So, in a sense, this $\epsilon$ parameter can be seen as turning a deterministic script into a non-deterministic one, in order to use it in a similar way as AlphaGo does. There are some significant differences, however, such as the fact that GNS is designed for RTS games, where more than one unit is controlled at the same time, and thus PUCB would not be applicable (notice that the search space in RTS games is many orders of magnitude larger than that of Go). An application of the Alphago ideas fo RTS games is the Informed NaïveMCTS algorithm that uses a variation of Naïve Sampling rather than PUCB in order to integrate the machine learned policy into MCTS (Ontanón 2016).

Generalizing our comparison to include methods such as AlphaGo, by considering non-deterministic scripts/policies is part of our future work, since this could shed light on the design space of possible new algorithms.

## Conclusion

In this paper, we survey and compare the recent advances of combining scripts and search algorithms. The community has approached this problem from a variety of the angles, as shown by the large number of existing approaches. The empirical results show that the algorithms differed in performance under maps of different scales. Specifically, algorithms that can search thoroughly and perform well in small maps cannot scale to large maps because of the scale of the search space. Algorithms that aggressively prune the search space and thus perform well in large maps are not so well performing in smaller maps due to the lack of optimality (in small maps, search-based algorithms can find plans that are closer to the optimal policy, and thus optimality matters in them).

We have also seen the relation between these methods and recent systems such as AlphaGo, which incorporate stochastic policies trained via machine learning into tree search. We believe studying the relation between these approaches is an important direction of future work, which can result in significant advances in game play strength. Finally, another key direction for future work is the design of algorithms that can strike a balance and scale up well to large maps, without sacrificing optimality, in order to handle well in smaller maps.

## References

Andersen, P.-A.; Goodwin, M.; and Granmo, O.-C. 2018. Deep rts: a game environment for deep reinforcement learning in real-time strategy games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. IEEE.

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47(2-3):235–256.

Balla, R.-K., and Fern, A. 2009. Uct for tactical assault planning in real-time strategy games. In *Twenty-First International Joint Conference on Artificial Intelligence*.

Barriga, N. A.; Stanescu, M.; and Buro, M. 2015. Puppet search: Enhancing scripted behavior by look-ahead search with applications to real-time strategy games. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1):1–43.

Buro, M., and Churchill, D. 2012. Real-time strategy game competitions. *AI Magazine* 33(3):106–106.

Buro, M. 2003. Real-time strategy games: A new ai research challenge. In *IJCAI*, volume 2003, 1534–1535.

Chung, M.; Buro, M.; and Schaeffer, J. 2005. Monte carlo planning in rts games. In *CIG*. Citeseer.

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

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994. Umcp: A sound and complete procedure for hierarchical task-network planning. In *AIPS*, volume 94, 249–254.

Gelly, S., and Wang, Y. 2006. Exploration exploitation in go: Uct for monte-carlo go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*.

Justesen, N.; Tillman, B.; Togelius, J.; and Risi, S. 2014. Script-and cluster-based uct for starcraft. In *2014 IEEE Conference on Computational Intelligence and Games*, 1–8. IEEE.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*, 282–293. Springer.

Marthi, B.; Russell, S.; and Latham, D. 2005. Writing stratagus-playing agents in concurrent alisp. *Reasoning, Representation, and Learning in Computer Games* 67.

Moraes, R. O., and Lelis, L. H. 2018. Asymmetric action abstractions for multi-unit control in adversarial real-time games. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Moraes, R. O.; Marino, J. R.; Lelis, L. H.; and Nascimento, M. A. 2018. Action abstractions for combinatorial multi-armed bandit tree search. In *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Ontanón, S., and Buro, M. 2015. Adversarial hierarchical-task network planning for complex real-time games. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

Ontanón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games* 5(4):293–311.

Ontañón, S.; Barriga, N. A.; Silva, C. R.; Moraes, R. O.; and Lelis, L. H. 2018. The first microrts artificial intelligence competition. *AI Magazine* 39(1).

Ontanón, S. 2016. Informed monte carlo tree search for real-time strategy games. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. IEEE.

Ontanón, S. 2017. Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research* 58:665–702.

OpenAI. 2018. Openai five. `https://blog.openai.com/openai-five/`.

Rosin, C. D. 2011. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence* 61(3):203–230.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature* 529(7587):484.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of go without human knowledge. *Nature* 550(7676):354.

Synnaeve, G.; Nardelli, N.; Auvolat, A.; Chintala, S.; Lacroix, T.; Lin, Z.; Richoux, F.; and Usunier, N. 2016. Torchcraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*.

Tian, Y.; Gong, Q.; Shang, W.; Wu, Y.; and Zitnick, C. L. 2017. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. *Advances in Neural Information Processing Systems (NIPS)*.

Vinyals, O.; Ewalds, T.; Bartunov, S.; Georgiev, P.; Vezhnevets, A. S.; Yeo, M.; Makhzani, A.; Küttler, H.; Agapiou, J.; Schrittwieser, J.; et al. 2017. Starcraft ii: a new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*.

Vinyals, O.; Babuschkin, I.; Czarnecki, W. M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D. H.; Powell, R.; Ewalds, T.; Georgiev, P.; et al. 2019. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature* 1–5.

Yang, Z., and Ontanón, S. 2019. Guiding monte carlo tree search by scripts in real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, 100–106.