Stratified Strategy Selection for Unit Control in Real-Time Strategy Games

Levi H. S. Lelis
Departamento de Informática, Universidade Federal de Viçosa, Brazil
levi.lelis@ufv.br

Abstract

In this paper we introduce Stratified Strategy Selection (SSS), a novel search algorithm for micromanaging units in real-time strategy (RTS) games. SSS uses a type system to partition the player's units into types and assumes that units of the same type must follow the same strategy. SSS searches in the state space induced by the type system to select, from a pool of options, a strategy for each unit. Empirical results on a simulator of an RTS game shows that SSS employing either fixed or adaptive type systems is able to substantially outperform state-of-the-art search-based algorithms in combat scenarios with up to 100 units.

1 Introduction

Real-time strategy (RTS) games represent a major challenge to Artificial Intelligence (AI). In contrast with traditional games such as Chess and Go, players act simultaneously, actions are durative, and the branching factor can be very large—for some RTS games it can be as large as 10^{50} [Ontañón *et al.*, 2013]. Although some RTS games are partially observable and non-deterministic, we assume deterministic and perfect information RTS games in this paper.

In an RTS game, the player controls a set of units to gather the resources needed to train and evolve an army. Eventually, the player's army is used to battle the opponent's. One of the challenges playing agents face in an RTS match is the control of dozens of combat units during battles. Due to the real-time constraints, agents have little time to plan before acting—planning time is on the order of milliseconds.

Churchill and Buro [2013] launched a line of research for dealing with large-scale RTS combat scenarios. Namely, they assume the existence of a pool of scripts Σ. A script $\bar{\sigma}$ is a function mapping a game state s and a unit u to a legal action m for u at s. In every game state s in which a unit u is ready to perform an action, a search algorithm selects a script $\bar{\sigma}$ from Σ for u. We call strategy selection algorithms the methods that search for a script assignment to every unit a player controls. Since a script maps a unit to an action, by assigning a script to a unit, strategy selection algorithms effectively decide which action each unit will perform. Churchill and Buro's strategy selection algorithm, Portfolio Greedy Search (PGS), substantially outperforms search algorithms such as Alpha-Beta [Knuth and Moore, 1975] and UCT [Kocsis and Szepesvári, 2006] in combat scenarios with dozens of units.

The strategy selection scheme allows search algorithms to concentrate their efforts in parts of the game tree that are deemed as promising by the scripts. That is, instead of considering all legal actions for unit u at game state s, a search algorithm considers only the actions returned by scripts in Σ. However, even then, depending on the number of units and scripts considered, search algorithms often evaluate only a small fraction of all possible actions within the time limit.

The main contribution of this paper is a search algorithm that uses a partition of the player's units, which we call a type system, to further reduce the number of possible unit-action assignments considered during search. This reduction is achieved by assigning the same script to units of the same type. For example, all wounded units (type) must move away from the battle (strategy encoded in a script). We call our algorithm Stratified Strategy Selection (SSS).

Another contribution of this paper is a meta-reasoning approach to automatically choose SSS's type system. Our approach chooses, from a pool of options, the type system to be used based on an estimate of SSS's running time. We call SSS+ this meta-reasoning-based variant of SSS.

Finally, with SSS and SSS+, we test the hypothesis that one is able to find strong strategies by searching the space of script assignments in which all units of the same type must be assigned the same script. Our experiments are run in SparCraft, a simplified combat simulation environment of Blizzard's StarCraft [Churchill and Buro, 2013]. Our results show that SSS and SSS+ are able to substantially outperform state-of-the-art approaches in combats with up to 100 units. We also show empirically SSS+'s advantage over SSS.

As the first of their kind, SSS and SSS+ represent new research directions, which include the development of novel type systems and algorithms for searching in type-induced state spaces for controlling units in RTS combats.

2 Related Work

Wang *et al.* [2016] also presents a strategy selection algorithm to control RTS combat units. However, instead of performing a hill-climbing search as is done by PGS, Wang *et al.* use a genetic algorithm for searching in the space of script as-

signments. Wang *et al.* showed that their genetic algorithm, Portfolio Online Evolution (POE), outperforms PGS.

The work of Justesen *et al.* [2014] also deals with the script assignment problem. Their algorithm is similar to ours because it also uses a partition of the units to guide its search. However, their method differs from ours in fundamental ways. SSS's partitions are defined by a type system, while Justesen *et al.* use a clustering algorithm to define their partitions. Our type systems use a diverse set of game-state features, while Justesen *et al.* use only position-based features. Also, one of our methods uses a meta-reasoning approach for selecting the type system from a pool of options. These differences result in a major performance improvement. Wang *et al.* [2016] showed that Justesen *et al.*'s approach is only competitive with PGS and is substantially outperformed by POE. We show in this paper that our stratified approaches substantially outperform both PGS and POE.

Before the invention of strategy selection approaches, state-of-the-art search-based algorithms included Monte-Carlo methods [Chung *et al.*, 2005; Balla and Fern, 2009; Ontañón, 2013] and Alpha-Beta variants [Churchill *et al.*, 2012]. Note that some of these methods are more general than strategy selection approaches (*e.g.*, [Ontañón and Buro, 2015]) in the sense that they can be used to control a playing agent throughout a complete RTS game. By contrast, the algorithms we consider in this paper are specialized for combat scenarios. Sailer *et al.* [2007], Preuss *et al.* [2013], and Tavares *et al.* [2016] present methods for selecting strategies for players in RTS games. By contrast, our strategy selection approaches assign strategies to units through scripts.

Another line of research uses learning to derive combat strategies. Search algorithms require an efficient forward model of the game to plan before acting. By contrast, learning approaches do not necessarily require an efficient forward model of the game as they learn from past experiences. Notable examples of this line include the work by Usunier *et al.* [2016] and Liu *et al.* [2016]. Likely due to the use of an efficient forward model, strategy selection algorithms tend to scale more easily to large combat scenarios than learning-based methods—while the former can effectively handle battles with more than 100 units, the latter are usually tested on battles with around 50 units.

Others have used type systems as part of algorithms for estimating the running time of search methods [Chen, 1992; Korf *et al.*, 2001; Zahavi *et al.*, 2010; Lelis *et al.*, 2013b; 2013a]. Type systems have also been used to enhance single-agent search algorithms [Lelis *et al.*, 2013c; Xie *et al.*, 2014; Betzalel *et al.*, 2015] and as part of algorithms for predicting the optimal cost of search problems [Lelis *et al.*, 2016]. By contrast, we use type systems as a means of enhancing search algorithms for RTS combat scenarios.

3 RTS Combat Scenarios

Combat scenarios that arise in RTS games, which we also call **matches**, can be described as finite zero-sum two-player games with simultaneous and durative moves. These matches can be defined by a tuple $(\mathcal{N}, \mathcal{S}, s_{init}, \mathcal{A}, \mathcal{R}, \mathcal{T})$, where,

- $\mathcal{N} = \{i, -i\}$ is the set of **players** (we assume to be controlling i and $-i$ to be our opponent);
- $\mathcal{S} = \mathcal{D} \cup \mathcal{F}$ is the set of **states**, where $\mathcal{D}$ denotes the set of **non-terminal states** and $\mathcal{F}$ the set of **terminal states**. Every state $s \in \mathcal{S}$ includes the joint set of **units** $\mathcal{U} = \mathcal{U}_i \cup \mathcal{U}_{-i}$, for players i and $-i$, respectively;
- $s_{init} \in \mathcal{D}$ is the start state of the match;
- $\mathcal{A} = \mathcal{A}_i \times \mathcal{A}_{-i}$ is the set of joint **player actions**. Each player action a is denoted by a tuple of n **unit actions** $(m_1, \cdots, m_n)$, where $m_k \in a$ is the action of the k-th **ready unit** of player i. A unit u is not ready at a given state s if u is currently performing an action. Whenever clear from the context, we will write *action* instead of *player action* and *unit action*;
- $\mathcal{R} : \mathcal{F} \times \mathcal{N} \to \mathbb{R}$ is a **utility function** with $\mathcal{R}(s, i) = -\mathcal{R}(s, -i)$, for any $s \in \mathcal{F}$;
- The **transition function** $\mathcal{T} : \mathcal{S} \times \mathcal{A}_i \times \mathcal{A}_{-i} \to \mathcal{S}$ determines the sucessor state given a state s and the set of joint player actions taken at s.

We denote the set of unit actions as $\mathcal{M}$, which includes actions for moving up, left, right and down, waiting, and attacking an enemy unit. Also, every unit $u \in \mathcal{U}$ has x and y **coordinates**, given by $x(u)$ and $y(u)$, **attack range** $r(u)$, **attack damage** $d(u)$, **unit action duration**, which depends on the unit and action being performed, **current hit points** $hp(u)$, **maximum hit points** $hp_m(u)$, and **weapon cool-down time** (*i.e.*, the time the unit has to wait before repeating an attack action). We assume Euclidean distance whenever referring to the distance between units.

A **decision point** of player i is a state s in which i has ready units and i has to decide on which action to take. A **strategy** $\sigma_i : \mathcal{S} \to \mathcal{A}_i$ for player i maps a state s to a player action a. One way of deriving good strategies for RTS games is by simplifying the problem's state space through the script assignment scheme introduced by Churchill and Buro [2013].

4 Scripts

A **script** $\bar{\sigma} : \mathcal{S} \times \mathcal{U} \to \mathcal{M}$ is a function mapping a state s and a unit u in s to an action m for u. A script $\bar{\sigma}$ allows one to define a strategy σ by applying $\bar{\sigma}$ to every ready unit in the state. We write $\bar{\sigma}$ instead of $\bar{\sigma}(s, u)$ whenever s and u are clear from the context. At every decision point s, strategy selection algorithms assign a script $\bar{\sigma}$ from Σ to every ready unit u in s. Unit u then performs the action returned by $\bar{\sigma}(s, u)$.

The success of strategy selection algorithms depends on the quality of the set of scripts Σ available. We use two scripts introduced by Churchill and Buro [2013] which they named No-Overkill-Attack-Value (NOKAV) and Kiter. NOKAV assigns actions to units so that the units do not cause more damage than the required to reduce an enemy's unit hp to zero. Kiter allows a unit to attack and then move away from the target; see Churchill and Buro [2013] for details.

In addition to NOKAV and Kiter, we use a third script called Cluster.[1] Cluster computes the average x and y coordinates, denoted $(\bar{x}, \bar{y})$, of all ally units $\mathcal{U}_i$. Then, it assigns

[1]The Cluster script was invented by David Churchill and is available in the SparCraft codebase [Churchill and Buro, 2013].

Algorithm 1 Stratified Strategy Selection

Require: scripts Σ, default script $\bar{\sigma}_d$, time limit e, evaluation function Ψ, type system T for the set of units $\mathcal{U}_i$.
Ensure: action a for player i, boolean c indicating if the algorithm finished a complete iteration over all types in T

```
1: b_i ← {σ̄_d, ··· , σ̄_d} // vector with |U_i| elements
2: b_{-i} ← {σ̄_d, ··· , σ̄_d} // vector with |U_{-i}| elements
3: c ← false
4: while time elapsed is not larger than e do
5:    for each t ∈ T do
6:       for each σ̄ ∈ Σ do
7:          b'_i ← b_i with the script of all
            units of type t replaced by σ̄
8:          if Ψ(b'_i, b_{-i}) > Ψ(b_i, b_{-i}) then
9:             b_i ← b'_i
10:         if time elapsed is larger than e then
11:            return action computed from b_i and boolean c
12:   c ← true // iterated over all types
13: return action computed from b_i and boolean c
```

to unit u an action that will take it closer to $(\bar{x}, \bar{y})$; Cluster assigns the wait move if the unit is already at $(\bar{x}, \bar{y})$.

The Cluster script allows us to expose a weakness in PGS's evaluation function while dealing with non-offensive scripts. In contrast with NOKAV and Kiter, which include attack actions in their rules, Cluster never assigns an attack action to a unit. We explain why PGS's evaluation function is unable to properly evaluate non-offensive strategies in Section 5.2. Also, we show empirically that the effective use of the Cluster script altogether with NOKAV and Kiter can result in strong gameplay strategies.

5 Stratified Strategy Selection (SSS)

SSS uses a partition of units, which we call a **type system**, and assigns the same script to units of the same type. For example, all units with low hp-value (type) move away from the battle so that they can survive longer (strategy encoded in a script). A type system is defined as follows.

Definition 1 (Type System) *Let $\mathcal{U}_i$ be the set of player i's units. $T = \{t_1, \ldots, t_k\}$ is a type system for $\mathcal{U}_i$ if it is a partitioning of $\mathcal{U}_i$. If $u \in \mathcal{U}_i$ and $t \in T$ with $u \in t$, we write $T(u) = t$.*

We describe our type systems in Section 5.1.

Algorithm 1 shows the pseudocode of SSS, which is invoked for deciding player i's action in their decision points throughout the game. SSS performs a hill-climbing search similar to PGS's search (see Churchill and Buro [2013] for details). However, in contrast with PGS, SSS searches in the space of script assignments induced by a type system.

SSS receives as input a set of scripts Σ, a default script $\bar{\sigma}_d \in \Sigma$, a time limit e, an evaluation function Ψ, and a type system for the units $\mathcal{U}_i$ at state s. SSS returns a player action for i (*i.e.*, a vector containing one unit action for each ready unit in s) and a boolean value c indicating whether SSS was able to complete one iteration over the set of types in T. The boolean value c is used by SSS+, as explained in Section 5.3.

SSS starts by initializing vectors b_i and b_{-i} with the default script $\bar{\sigma}_d$; b_i and b_{-i} store one script for each ready unit controlled by players i and $-i$, respectively (in the pseudocode we assume all units in $\mathcal{U}_i$ and $\mathcal{U}_{-i}$ to be ready). We also assume an arbitrary ordering of the units in $\mathcal{U}_i$ and $\mathcal{U}_{-i}$ so that the script stored in the first position of b_i (resp. b_{-i}), denoted $b_i[0]$ (resp. $b_{-i}[0]$), corresponds to the script of $\mathcal{U}_i$'s (resp. $\mathcal{U}_{-i}$) first unit. Vector b_i unambiguously define a player action for i. The player action is computed from b_i by using $b_i[0]$ to compute the unit action of the first unit in $\mathcal{U}_i$, $b_i[1]$ to compute the unit action of the second unit and so on.

SSS performs a greedy search to iteratively improve the script assignment of b_i (lines 4–9). Namely, SSS evaluates all possible assignments of scripts from Σ to units of a given type t while the script of units with types other than t are fixed. SSS keeps in b_i the assignment with largest Ψ-value encountered during search (lines 8 and 9).

The evaluation function Ψ receives b_i and b_{-i} as input and estimates the end-game utility assuming player i has taken the action computed from b_i and player $-i$ the action computed from b_{-i}. We discuss different evaluation functions in Section 5.2. SSS returns the player action computed from b_i once it reaches the time limit e (lines 11 and 13).

Note that vector b_{-i} does not change during search and SSS is effectively searching for a best response to b_{-i}. In theory, players following strategies derived by SSS can be exploited as $-i$ might be able to anticipate that i is computing a best response to b_{-i} and thus compute a best response to i's action. However, due to the problem's size and real-time constraints, it is currently impractical to guess a player's strategy and compute a best response to it. Also, in theory, even if SSS had unlimited time to compute a strategy, an opponent that considers all legal actions during search might be able to exploit player i, as SSS is constrained to the unit actions returned by the scripts. In practice, however, as Churchill and Buro [2013] have shown, one is only able to derive good strategies while accounting for all legal actions in scenarios with very few combat units.

5.1 Type Systems

Since SSS assigns the same script to all units of a given type, it is important that the type system captures important strategic information regarding the game state at hand.

Intuitively, the units' attack range (r), damage (d), and hit points (hp) encode important strategic information. For example, the Kiter script tends to be more effective if employed by units with a large attack range, as it allows the unit to attack and retreat without being hit by the enemy. Similarly, by assigning a less offensive script to units with lower hp-values one might be able to preserve the units longer in the match. We define the canonical type system T_c as follows.

$$T_{c,l}(u) = \big(r(u), d(u), hp(u,l)\big) \, ,$$

where l is an integer used for mapping u's hp-value into "levels",

$$hp(u,l) = \left\lfloor \frac{hp(u)}{\lfloor \frac{hp_m(u)}{l} \rfloor} \right\rfloor .$$

We do not use similar mappings for r and d because they are fixed (and thus less diverse) throughout the match.

Also, we define $T_{c,0}(u) = \big(r(u), d(u)\big)$.

Example 1 *Let u_1, u_2, and u_3 be three units with $r = 20$, $d = 5$, $hp_m = 60$, and with hp-values of 10, 15, and 30, respectively. For $l = 3$, $T_{c,3}(u_1) = T_{c,3}(u_2) = \big(20, 5, 0\big)$, but u_3 has a different type as $T_{c,3}(u_3) = \big(20, 5, 1\big)$.*

The parameter l of T_c allows one to create type systems with different **granularities**. By increasing the value of l one creates larger type systems, thus allowing SSS to find "finer strategies" at the cost of increasing the running time for completing one iteration of SSS's while loop (see Algorithm 1). The granularity of type systems is defined as follows.

Definition 2 (Type Systems Granularity) *Let T_1, T_2 be type systems for $\mathcal{U}_i$. Also, let $|T_1|$ and $|T_2|$ be the maximum number of types T_1 and T_2 might have during a match. T_1 is finer than T_2, denoted $T_1 \preceq T_2$, if $|T_1| \geq |T_2|$.*

For example, since the maximum number of different types $T_{c,3}$ might have is larger than the maximum number of different types $T_{c,2}$ might have in a given match, we say that $T_{c,3}$ is finer than $T_{c,2}$, or $T_{c,3} \preceq T_{c,2}$.

5.2 Evaluation Functions

In this section we discuss different evaluation functions Ψ. The first evaluation we discuss is LTD2, a function introduced by Kovarsky and Buro [2005] that accounts for the unit's hp and how much damage each unit can cause per frame of the game (dpf). The LTD2 of a state s with units $\mathcal{U} = \mathcal{U}_i \cup \mathcal{U}_{-i}$ is written as follows,

$$\sum_{u \in \mathcal{U}_i} \sqrt{hp(u)} \cdot dpf(u) - \sum_{u \in \mathcal{U}_{-i}} \sqrt{hp(u)} \cdot dpf(u) \, .$$

Churchill *et al.* [2012] introduced a playout-based evaluation that outperforms LTD2. Instead of evaluating the state s directly, their scheme simulates the game from s to a terminal state s_f while following a fixed set of scripts for player i, b_i, and for player $-i$, b_{-i} (all player actions in the simulation are computed according to b_i and b_{-i}); the LTD2-value of s_f is computed and returned as an estimated end-game value of s. PGS uses Churchill *et al.*'s evaluation function. The drawback of this evaluation scheme is that it incorrectly evaluates actions computed from vectors b_i containing non-offensive scripts such as Cluster. This is because the evaluation function assumes that the units will follow the scripts defined in b_i in every decision point to the end of the match. This means that one assumes that units following the Cluster script will not attack for the rest of the match, which is unreasonable.

A secondary contribution of this paper is the use of a different evaluation function with PGS. Namely, we use a playout-based evaluation in which the first joint player action in the simulation is computed from b_i and b_{-i} and all the others, to the end of the simulation, are computed according to the NOKAV script. We call PGS with this evaluation function PGS+. SSS and SSS+ also use PGS+'s evaluation scheme. We note that POE's evaluation function is similar to PGS+'s; see Wang *et al.* [2016] for details.

5.3 SSS with Adaptive Type Systems (SSS+)

Depending on the number and on the diversity of units (*e.g.*, units with different attack ranges) present in the match, SSS might be unable to iterate through all types in T before reaching time limit e. This is because a large diversity of units result in more types being considered during search. Also, the running time of Ψ tends to increase as one increases the number of units in the match as the simulation takes longer.

If SSS is unable to iterate through all types, then it returns moves computed from the default script $\bar{\sigma}_d$ for units u whose type $T(u)$ was not accounted for during search. The assignment of $\bar{\sigma}_d$ might lead to a poor overall strategy as there could be better scripts that were not verified by the algorithm due to the lack of time. Aiming at preventing SSS from not iterating at least once over all types, we developed a meta-reasoning system to adjust the granularity of the type system used. This adjustment occurs in between searches and is based on the estimated running time of a SSS iteration. We call the resulting algorithm SSS+.

Instead of receiving one type system T, SSS+ receives a set of type systems $\mathcal{Y} = \{T_1, T_2, \cdots, T_N\}$ with $T_1 \preceq T_2 \preceq \cdots \preceq T_N$. SSS+ starts with T_1, the finest type system in $\mathcal{Y}$. If the search returns false while using T_1 (see variable c in Algorithm 1), SSS+ replaces T_1 by T_2 for its next search. If the search returns true while using T_2 and the meta-reasoning system estimates that SSS+ will be able to complete an iteration of the algorithm with T_1, then it replaces T_2 by T_1.

In general, let T_j be the type system in $\mathcal{Y}$ used in SSS+'s last search. SSS+'s meta-reasoning defines the type system T to be used in its next search as follows.

$$T = \begin{cases} T_{j+1}, & \text{if condition } C1 \text{ is satisfied} \, , \\ T_{j-1}, & \text{if condition } C2 \text{ is satisfied} \, , \\ T_j, & \text{otherwise} \, . \end{cases}$$

Here, conditions C1 and C2 are defined as,

$$C1 = \neg c \wedge (j < N) \, ,$$
$$C2 = c \wedge (j > 1) \wedge (\tilde{t} \times \widetilde{|T_{j-1}|} \times |\Sigma| \leq e) \, .$$

Here, $\tilde{t}$ is an estimate of Ψ's running time and $\widetilde{|T_{j-1}|}$ is an estimate of the number of types in T_{j-1}. As mentioned before, Ψ's running time varies during the match as the number of units in the game state changes. Thus, $\tilde{t}$ is estimated as the average running time of the multiple Ψ runs performed in SSS+'s last search. Whenever the meta-reasoning approach changes the type system employed from T_{j-1} to a coarser type system T_j we store in memory the number of types of T_{j-1} encountered during search, which we use as $\widetilde{|T_{j-1}|}$.

Condition C1 is satisfied when SSS+ is unable to iterate through all types ($\neg c$) and the current type system being employed is not the coarsest in $\mathcal{Y}$ ($j < N$). Condition C2 is satisfied when SSS+ is able to iterate through all types (c), the current type system being employed is not the finest in $\mathcal{Y}$ ($j > 1$), and the meta-reasoning approach estimates that SSS+ will be able to iterate through all types if using a type system that is finer than the last used ($\tilde{t} \times \widetilde{|T_{j-1}|} \times |\Sigma| \leq e$).

(Zea 4)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.04	0.08	0.06	0.06
POE	0.96	-	0.44	0.39	0.32
PGS+	0.92	0.56	-	0.41	0.42
SSS	0.94	0.61	0.59	-	0.50
SSS+	0.94	0.68	0.58	0.50	-

(Zea 2, Dra 2)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.24	0.24	0.23	0.23
POE	0.76	-	0.33	0.37	0.28
PGS+	0.76	0.67	-	0.44	0.44
SSS	0.77	0.63	0.56	-	0.50
SSS+	0.77	0.72	0.56	0.50	-

(Zea 2, Dra 2, Ling 2)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.44	0.26	0.22	0.22
POE	0.56	-	0.21	0.24	0.14
PGS+	0.74	0.79	-	0.41	0.41
SSS	0.78	0.76	0.59	-	0.50
SSS+	0.78	0.86	0.59	0.50	-

(Zea 2, Dra 2, Ling 2, Mar 2)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.41	0.15	0.25	0.25
POE	0.59	-	0.06	0.14	0.12
PGS+	0.85	0.94	-	0.57	0.57
SSS	0.75	0.86	0.43	-	0.50
SSS+	0.75	0.88	0.43	0.50	-

(Zea 16)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.01	0.03	0.00	0.00
POE	0.99	-	0.29	0.04	0.04
PGS+	0.97	0.71	-	0.12	0.12
SSS	1.00	0.96	0.88	-	0.50
SSS+	1.00	0.96	0.88	0.50	-

(Zea 8, Dra 8)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.07	0.28	0.09	0.09
POE	0.93	-	0.44	0.08	0.07
PGS+	0.72	0.56	-	0.19	0.19
SSS	0.91	0.92	0.81	-	0.49
SSS+	0.91	0.93	0.81	0.51	-

(Zea 6, Dra 6, Ling 6)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.18	0.13	0.04	0.02
POE	0.82	-	0.12	0.03	0.03
PGS+	0.87	0.88	-	0.18	0.18
SSS	0.96	0.97	0.82	-	0.50
SSS+	0.98	0.97	0.82	0.50	-

(Zea 4, Dra 4, Ling 4, Mar 4)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.30	0.19	0.18	0.17
POE	0.70	-	0.10	0.16	0.13
PGS+	0.81	0.90	-	0.33	0.26
SSS	0.81	0.84	0.67	-	0.46
SSS+	0.83	0.87	0.74	0.54	-

(Zea 40)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.00	0.00	0.00	0.00
POE	1.00	-	0.02	0.00	0.00
PGS+	1.00	0.98	-	0.12	0.14
SSS	1.00	1.00	0.88	-	0.50
SSS+	1.00	1.00	0.86	0.50	-

(Zea 20, Dra 20)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.00	0.08	0.01	0.01
POE	1.00	-	0.47	0.07	0.05
PGS+	0.92	0.53	-	0.30	0.23
SSS	0.99	0.93	0.70	-	0.39
SSS+	0.99	0.95	0.77	0.61	-

(Zea 14, Dra 14, Ling 14)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.01	0.01	0.00	0.00
POE	0.99	-	0.12	0.02	0.01
PGS+	0.99	0.88	-	0.09	0.06
SSS	1.00	0.98	0.92	-	0.37
SSS+	1.00	0.99	0.94	0.63	-

(Zea 10, Dra 10, Ling 10, Mar 10)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.05	0.11	0.09	0.05
POE	0.95	-	0.18	0.26	0.09
PGS+	0.89	0.82	-	0.26	0.04
SSS	0.91	0.74	0.74	-	0.25
SSS+	0.95	0.91	0.96	0.75	-

(Zea 56)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.01	0.00	0.00	0.00
POE	0.99	-	0.00	0.00	0.00
PGS+	1.00	1.00	-	0.15	0.14
SSS	1.00	1.00	0.84	-	0.52
SSS+	1.00	1.00	0.86	0.48	-

(Zea 28, Dra 28)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.00	0.03	0.01	0.01
POE	1.00	-	0.32	0.03	0.04
PGS+	0.97	0.68	-	0.32	0.36
SSS	0.99	0.97	0.68	-	0.54
SSS+	0.99	0.96	0.64	0.46	-

(Zea 18, Dra 18, Ling 18)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.01	0.01	0.00	0.00
POE	0.99	-	0.08	0.01	0.01
PGS+	0.99	0.92	-	0.10	0.06
SSS	1.00	0.99	0.90	-	0.42
SSS+	1.00	0.99	0.94	0.58	-

(Zea 14, Dra 14, Ling 14, Mar 14)	PGS	POE	PGS+	SSS	SSS+
PGS	-	0.02	0.03	0.10	0.01
POE	0.98	-	0.10	0.54	0.04
PGS+	0.97	0.90	-	0.84	0.05
SSS	0.90	**0.46**	**0.16**	-	0.07
SSS+	0.99	**0.96**	**0.95**	**0.93**	-

Table 1: Winning rate of the row player against the column player for various combat scenarios. The numbers are rounded to two decimal places. Orange-colored cells show scenarios in which the row player won more than 50% of the 1,000 matches tested; black-colored cells show the results in which SSS and SSS+ differ the most.

6 Empirical Results

We use as our testbed a simulation environment of Blizzard's StarCraft called SparCraft.[2] The unit properties such as hit points, cool-down time, and damage are exactly the same as the original game. However, SparCraft does not implement fog of war, collisions, and unit acceleration (all units move at constant speed) [Churchill and Buro, 2013]. We use SparCraft because, in contrast with the original game, it offers an efficient forward model of the game. All experiments are run on 2.66 GHz machines.

6.1 Combat Scenarios

We experiment with units with different hp, d, and r-values. We use $\uparrow$ to denote large and $\downarrow$ to denote small hp and d-values. Also, we call u a melee unit if u's attack range equals zero ($r = 0$), and we call u a ranged unit if u is able to attack from far ($r > 0$). Namely, we use the following unit kinds: Zealots (Zea, $\uparrow hp$, $\uparrow d$, melee), Dragoons (Dra, $\uparrow hp$, $\uparrow d$, ranged), Zerglings (Ling, $\downarrow hp$, $\downarrow d$, melee), Marines (Mar, $\downarrow hp$, $\downarrow d$, ranged). We consider the combat scenarios where each player controls units of the following kinds: (i) Zea; (ii) Zea and Dra; (iii) Zea, Dra, and Ling; and (iv) Zea, Dra, Ling, and Mar. We experiment with matches with as few as 4 units and as many as 56 units on each side. The largest number of units controlled by a player in a typical StarCraft combat is around 50 [Churchill and Buro, 2013].

The units are placed in a walled arena with no obstacles of size 1280 $\times$ 780 pixels; the largest unit is approximately 40 $\times$ 50 pixels large. The walls ensure finite matches by preventing units from indefinitely moving away from enemies. For each combat scenario we generate 1,000 start states as explained by Churchill and Buro [2013]. Namely, player i's units are placed at a random coordinate to the right of the center of the arena, which we define to be the coordinate $(0, 0)$. Player $-i$'s units are placed at a symmetric position to the left of $(0, 0)$. The coordinates are chosen so that the units are at a distance in between 0 and 128 pixels from $(0, 0)$. Then, to ensure that no unit starts within the attack range of an enemy unit, we add 220 pixels to the x-coordinate of player i's units, and subtract 220 pixels from the x-coordinate of player $-i$'s units, thus increasing the distance between enemy units by 440 pixels. We present the winning rate over the 1,000 matches for each combat scenario and algorithm tested.

[2] github.com/davechurchill/ualbertabot/tree/master/SparCraft

6.2 Configurations of Algorithms Tested

We test PGS [Churchill and Buro, 2013], PGS+ (PGS with our Ψ), POE [Wang *et al.*, 2016], and SSS and SSS+. We also experimented with ABCD [Churchill *et al.*, 2012] and UCTCD [Churchill and Buro, 2013], but both algorithms performed much worse than the others and we omit their results for clarity. All algorithms use the same set $\Sigma = \{$NOKAV, Kiter, Cluster$\}$ of scripts, and have a time limit of 40 milliseconds for each decision point.

PGS and PGS+: Instead of limiting PGS and PGS+ to a fixed number of iterations (the number of times they try to improve their script assignments), as Churchill and Buro [2013] did, we let PGS and PGS+ improve the assignments while there is time available. Also, as suggested by Churchill and Buro, we set PGS's and PGS+'s improvement response parameter to 0; see Churchill and Buro [2013] for details.

POE: We have implemented POE in SparCraft's codebase and tested several configurations of its parameters, including the values suggested by Wang *et al.* [2016]. The best configuration we found is the following: population size of 36,

with the 6 fittest individuals being selected for generating 5 offsprings each. Also, POE uses a playout-based evaluation function that is limited to 25 decision points; see Wang *et al.* [2016] for details.

SSS and SSS+: For SSS we use $T_{c,3}$ and for SSS+ we use $\mathcal{Y} = \{T_{c,3}, T_{c,2}, T_{c,1}, T_{c,0}, T_{RGD}\}$, where T_{RGD} is a type system that assigns the same type to two units if they are either both melee or both ranged. Clearly, $T_{c,3} \preceq T_{c,2} \preceq T_{c,1} \preceq T_{c,0} \preceq T_{RGD}$. We also tested SSS with $T_{c,2}$, $T_{c,1}$, $T_{c,0}$, and T_{RGD} (results are omitted for lack of space); the best overall results for SSS were obtained with $T_{c,3}$.

6.3 Discussion

Table 1 shows the winning rate of the row player against the column player for various combat scenarios. For example, for the combat scenario in which each player controls 8 Zealots and 8 Dragoons (Zea 8, Dra 8) SSS wins 92% of the matches against POE. The orange-colored cells in Table 1 show scenarios in which the row player won more than 50% of the 1,000 matches tested; black-colored cells show the results in which SSS and SSS+ differ the most.

POE substantially outperforms PGS in all scenarios tested. Note, however, that PGS+ outperforms both PGS and POE. This is because the evaluation function used with PGS assumes that the script chosen at a given decision point will be used throughout the match. As a result, PGS does not select the non-offensive Cluster script. By contrast, our results suggest that PGS+ is able to effectively select non-offensive strategies and gain strategic advantage over its opponents.

SSS and SSS+'s winning rates are smaller for scenarios with fewer units, *e.g.*, (Zea 4), (Zea 2, Dra 2), (Zea 2, Dra 2, Ling 2), and (Zea 2, Dra 2, Ling 2, Mar 2), but almost always above 50%, with the exception of (Zea 2, Dra 2, Ling 2, Mar 2), where PGS+ won 57% of the matches against both SSS and SSS+. PGS+ is able to outperform SSS and SSS+ because it might be able to evaluate all possible script assignments in scenarios with fewer units, while SSS and SSS+ consider only a subset of the assignments considered by PGS+. Nevertheless, SSS and SSS+ have winning rates usually above 80% in scenarios with more units. Our results show that one is able to find stronger strategies than competing schemes by searching in the space induced by a type system. This suggests that the type system effectively prunes a potentially large and unpromising portion of the game tree by reducing the number of script assignments considered.

SSS and SSS+ present similar results for matches with one, two, and three kinds of units. However, major differences are observed between the two algorithms in scenarios with four kinds of units; these differences are highlighted by the dark cells in Table 1. In such scenarios, due to the large number of types and units, SSS might be unable to iterate through all types within the time limit. As a result, most units follow the default NOKAV script. As the match progresses and units are removed from the game state, the running time of Ψ reduces, which allows SSS to iterate through all types within the time limit. However, by then, the enemy has likely gained an irreversible strategic advantage over SSS.

Figure 1 shows the running time of Ψ throughout the first 150 decision points of player i in a representative (Zea 14, Dra 14, Ling 14, Mar 14) match. The background color shows the type system being used at a given decision point. Here, the darkest color indicates the use of the coarsest T_{RGD} type system, while the lightest color indicates the use of the finest $T_{c,3}$. SSS+ quickly switches to the coarsest T_{RGD} type system (depicted by the dark region on the left-hand side of the figure). As the match progresses and the running time of Ψ decreases, the coarsest type system is replaced by finer ones, allowing SSS+ to search in a richer space of possibilities, until it eventually converges to the $T_{c,3}$ type system.

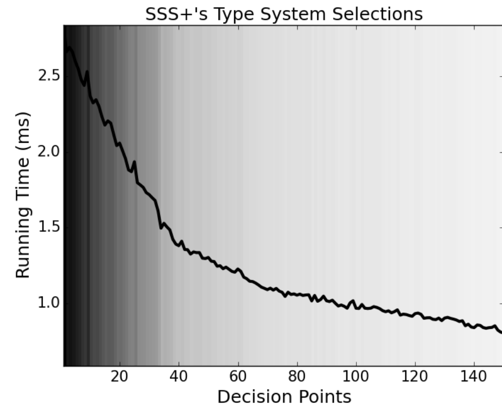

Figure 1: Ψ's running time throughout a match. Darker colors in the background mean the use of coarser type systems.

SSS+'s automatic type system selection results in major improvements in the (Zea 14, Dra 14, Ling 14, Mar 14) configuration: SSS+ wins 96% and 95% of its matches against POE and PGS+, respectively, while SSS wins only 46% and 16% of the matches against the same opponents. Moreover, in this scenario, SSS+ wins 93% of the matches against SSS.

7 Conclusions and Future Works

In this paper we introduced SSS, a search algorithm for RTS combat scenarios that is guided by a type system. We also introduced SSS+, a SSS variant that uses a meta-reasoning approach to automatically choose its type system. Experiments on SparCraft showed that both SSS and SSS+ are able to substantially outperform state-of-the-art algorithms. Also, our results showed that in combat scenarios with a large and diversified number of units SSS+ can outperform SSS.

As future research, we intend to test SSS and SSS+ in games with unit acceleration and collision. Also, Churchill and Buro [2015] showed that a variant of PGS is used in a comercial card game. SSS could also be used to enhance the game's AI so that it might challenge the more skilled players.

Acknowledgements

The author gratefully thank Rubens de O. Moraes Filho, Rob Holte, Anderson Tavares and the anonymous referees of this paper for great suggestions, and David Churchill, Chen Wang and Julian Togelius for kindly answering questions about SparCraft, PGS, and POE. Financial support for this research was provided by Brazil's CAPES CsF, FAPEMIG, and CNPq.

References

[Balla and Fern, 2009] R-K. Balla and A. Fern. Uct for tactical assault planning in real-time strategy games. In *International Joint Conference on Artificial Intelligence*, pages 40–45, 2009.

[Betzalel *et al.*, 2015] O. Betzalel, A. Felner, and S. E. Shimony. Type system based rational lazy IDA. In *Symposium on Combinatorial Search*, pages 151–155, 2015.

[Chen, 1992] P.-C. Chen. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, 21:295–315, 1992.

[Chung *et al.*, 2005] M. Chung, M. Buro, and J. Schaeffer. Monte Carlo planning in RTS games. In *IEEE Symposium on Computational Intelligence and Games*, 2005.

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

[Churchill and Buro, 2015] D. Churchill and M. Buro. Hierarchical portfolio search: Prismata's robust AI architecture for games with large search spaces. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 16–22, 2015.

[Churchill *et al.*, 2012] D. Churchill, A. Saffidine, and M. Buro. Fast heuristic search for RTS game combat scenarios. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2012.

[Justesen *et al.*, 2014] N. Justesen, B. Tillman, J. Togelius, and S. Risi. Script- and cluster-based UCT for StarCraft. In *IEEE Conference on Computational Intelligence and Games*, pages 1–8, 2014.

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

[Kocsis and Szepesvári, 2006] L. Kocsis and C. Szepesvári. Bandit based monte-Carlo planning. In *European Conference on Machine Learning*, pages 282–293. Springer-Verlag, 2006.

[Korf *et al.*, 2001] R. E. Korf, M. Reid, and S. Edelkamp. Time complexity of Iterative-Deepening-A*. *Artificial Intelligence*, 129(1-2):199–218, 2001.

[Kovarsky and Buro, 2005] A. Kovarsky and M. Buro. Heuristic search applied to abstract combat games. In *Advances in Artificial Intelligence: Conference of the Canadian Society for Computational Studies of Intelligence*, pages 66–78. Springer, 2005.

[Lelis *et al.*, 2013a] L. H. S. Lelis, L. Otten, and R. Dechter. Predicting the size of depth-first branch and bound search trees. In *International Joint Conference on Artificial Intelligence*, pages 594 – 600, 2013.

[Lelis *et al.*, 2013b] L. H. S. Lelis, S. Zilles, and R. C. Holte. Predicting the Size of IDA*'s Search Tree. *Artificial Intelligence*, pages 53–76, 2013.

[Lelis *et al.*, 2013c] L. H. S. Lelis, S. Zilles, and R. C. Holte. Stratified Tree Search: a novel suboptimal heuristic search algorithm. In *Conference on Autonomous Agents and Multiagent Systems*, pages 555–562, 2013.

[Lelis *et al.*, 2016] L. H. S. Lelis, R. Stern, S. Zilles, A. Felner, and R. C. Holte. Predicting optimal solution costs with bidirectional stratified sampling in regular search spaces. *Artificial Intelligence*, pages 51–73, 2016.

[Liu *et al.*, 2016] S. Liu, S. J. Louis, and C. A. Ballinger. Evolving effective microbehaviors in real-time strategy games. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(4):351–362, 2016.

[Ontañón and Buro, 2015] S. Ontañón and M. Buro. Adversarial hierarchical-task network planning for complex real-time games. In *International Joint Conference on Artificial Intelligence*, pages 1652–1658, 2015.

[Ontañón *et al.*, 2013] 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*, 5(4):293–311, 2013.

[Ontañón, 2013] S. Ontañón. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 58–64, 2013.

[Preuss *et al.*, 2013] M. Preuss, D. Kozakowski, J. Hagelbck, and H. Trautmann. Reactive strategy choice in starcraft by means of fuzzy control. In *Conference on Computational Intelligence in Games*, pages 1–8. IEEE, 2013.

[Sailer *et al.*, 2007] F. Sailer, M. Buro, and M. Lanctot. Adversarial planning through strategy simulation. In *IEEE Symposium on Computational Intelligence and Games*, pages 80–87, 2007.

[Tavares *et al.*, 2016] A. Tavares, H. Azpúrua, A. Santos, and L. Chaimowicz. Rock, paper, starcraft: Strategy selection in real-time strategy games. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 93–99, 2016.

[Usunier *et al.*, 2016] N. Usunier, G. Synnaeve, Z. Lin, and S. Chintala. Episodic exploration for deep deterministic policies: An application to StarCraft micromanagement tasks. *CoRR*, abs/1609.02993, 2016.

[Wang *et al.*, 2016] C. Wang, P. Chen, Y. Li, C. Holmgård, and J. Togelius. Portfolio online evolution in StarCraft. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 114–120, 2016.

[Xie *et al.*, 2014] F. Xie, M. Müller, R. Holte, and T. Imai. Type-based exploration with multiple search queues for satisficing planning. In *AAAI Conference on Artificial Intelligence*, pages 2395–2402, 2014.

[Zahavi *et al.*, 2010] U. Zahavi, A. Felner, N. Burch, and R. C. Holte. Predicting the performance of IDA* using conditional distributions. *Journal of Artificial Intelligence Research*, 37:41–83, 2010.