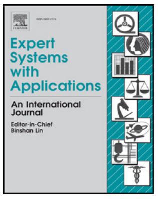
Expert Systems With Applications 71 (2017) 192–205

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa



Playing real-time strategy games by imitating human players' micromanagement skills based on spatial analysis

In-Seok Oh, Hochul Cho, Kyung-Joong Kim*

Department of Computer Science and Engineering, Sejong University, Seoul, South Korea

A R T I C L E   I N F O

Article history:
Received 26 April 2016
Revised 27 October 2016
Accepted 19 November 2016
Available online 21 November 2016

Keywords:
Imitation
Case-based reasoning
Influence map
Potential field
Real-time strategy
Micromanagement

A B S T R A C T

Unlike the situation with board games, artificial intelligence (AI) for real-time strategy (RTS) games usually suffers from numerous possible future outcomes because the state of the game is continuously changing in real time. Furthermore, AI is also required to be able to handle the increased complexity within a small amount of time. This constraint makes it difficult to build AI for RTS games with current state-of-the art intelligence techniques. A human player, on the other hand, is proficient in dealing with this level of complexity, making him a better game player than AI bots. Human players are especially good at controlling many units at the same time. It is hard to explain the micro-level control skills needed with only a few rules programmed into the bots. The design of micromanagement skills is one of the most difficult parts in the StarCraft AI design because it must be able to handle different combinations of units, army size, and unit placement. The unit control skills can have a big effect on the final outcome of a full game in professional player matches. For StarCraft AI competitions, they employed a relatively simple scripted AI to implement the unit control strategy. However, it is difficult to generate cooperative behavior using the simple AI strategies. Although there has been a few research done on micromanagement skills, it is still a challenging problem to design human-like high-level control skills. In this paper, we proposed the use of imitation learning based on human replays and influence map representation. In this approach, we extracted huge numbers of cases from the replays of experts and used them to determine the actions of units in the current game case. This was done without using any hand-coded rules. Because this approach is data-driven, it was essential to minimize the case search times. To support fast and accurate matching, we chose to use influence maps and data hashing. They allowed the imitation system to respond within a small amount time (one frame, 0.042 s). With a very large number of cases (up to 500,000 cases), we showed that it is possible to respond competitively in real-time, with a high winning percentage in micromanagement scenarios.

1. Introduction

In real-time strategy (RTS) game combat, players need to be able to handle multiple units simultaneously at a micro-level. It is like juggling several balls at once, and players have to divide their attention into several on-going sub tasks. Also, the players only have a limited view of their opponent's territory. This means a player must integrate multiple sources of information of varying uncertainty to make quick decisions. In addition, it is difficult to control combat units because a player must quickly consider several things at once including the arrangement of their units, the current state of war, and the uncertain strategy and unit location of the enemy.

In micromanagement, it is necessary to control individual units to achieve goals. Because RTS games have many units, it is difficult to control all of them effectively. Micro-level control includes navigation (path planning) and action (command assignment) for each unit. The actions include "attack," "move," and "stop." In games with professional players, micro-level control skills are among the most important factors determining who wins the championship. In combat, human players minimize damage to their own units and maximize elimination of opponents' units.

The design of micromanagement skills for RTS games is known to be a challenging problem (Buro, 2003). Computers have been very successful in solving some turn-based discrete games with perfect information such as chess, checkers, and Othello (Schaeffer, 2009). However, the search space for RTS games is continuous because they are real-time, making it difficult to find an optimal action based on possible future outcomes. Unlike board games, it is not feasible to construct opening and endgame databases. In the

* Corresponding author.
E-mail addresses: ohinsuk@naver.com (I.-S. Oh), chc2212@naver.com (H. Cho), kimkj@sejong.ac.kr (K.-J. Kim).

http://dx.doi.org/10.1016/j.eswa.2016.11.026
0957-4174/© 2016 Elsevier Ltd. All rights reserved.

combat stage of an RTS game, the computer has a very limited capability to understand the game situation, and the best action sequence is highly dependent on several factors.

RTS game replays are easily downloadable from popular gaming portals, which have 300,000 replays[1] especially for StarCraft.[2] A replay contains all the information required to reconstruct the original game, as it saves all the actions of the gamers with their game states. Like transcripts in Go, the replays have been widely used to study professional players' skills or simply enjoy the games by watching them. Following the development of major interest in StarCraft games, these game replays have been shared and distributed by gamers, fans, and spectators. Considerable research has been conducted in an effort to exploit replay databases for RTS games. For example, Cho et al. used replay files to build a better prediction strategy model than the expert-designed rules (Cho, Kim, & cho, 2013). Hostetler et al. built a probabilistic model to infer the opponent's hidden information based on observable data (Hostetler, Dereszynski, Dietterich, & Fern, 2012). Although these replays are useful for testing the predictions of the various machine-learning approaches, they are not widely used for micromanagement.

In this paper, we report on a test of a data-driven approach for the imitation learning of micromanagement skills. In terms of imitation learning, we used an influence map representing the real-time strategy game. The replayed game states of the scenes were stored as individual cases and aggregated into a case library. As a replay has the potential to serve as the basis of an immense number of possible cases, we then retrieved the case that was most similar to the state of the current game from the Case library, and used the selected case for imitation. Finally, the AI agent imitated the movement of units in the "best" case for next time frame. Several design choices are needed to allow the imitation to be flexible and to respond in real time. Because the game state of each scene involves different conditions, a direct comparison is not feasible. Instead, we use a unified representation of each game state when comparing the game states of two scenes. Then, hashing techniques with tolerance were used to reduce search time. Finally, a unit-by-unit mapping was conducted, allowing the imitation to be carried out.

The strong point of our model is that because it is based on Full Game Replays instead of on Combat Replay Data, it is able undertake the combat part of a bot without additional adjustments. Furthermore, the proposed Case-based reasoning method based on spatial analysis can be applied to a number of expert systems besides the StarCraft domain, making the model appropriate to be applied in a number of situations where the best way to solve problems are unclear. In addition, when applying techniques such as reinforcement learning in a combat platform, the efficiency of the opposing agent is an important factor. Our proposed model can be used to build opponent AIs which can in turn be used for learning in combat platforms.

This paper is organized as follows: Section 2 introduces StarCraft and case-based reasoning as background information. Section 3 summarizes related work on unit micromanagement in RTS and the use of imitation learning for games. Section 4 describes the proposed method in detail, Section 5 explains the experimental results obtained using the competition platform, Section 6 deals with the pros and cons of our proposed method. Section 7 discusses the conclusions drawn from this study and proposes possible future projects.

[1] www.bwreplays.com (338,649 replays are available, May 2015).

[2] A military science fiction RTS game released by Blizzard in 1998. It has been one of the most popular RTS games with professional gamers, broadcast channels, and an enormous number of replays.

2. Background

2.1. StarCraft and AI competition

StarCraft: Brood War is a popular RTS game released by Blizzard Entertainment. In the game, a player needs to choose a race: Terran, Protoss, or Zerg. Although they are well balanced, their units have different versatility, flexibility, manufacturing processes, strengths, resistance, and cost. In the game, players collect resources (minerals and gas) to create buildings, produce units, and upgrade both of these resources. Because the players have only a limited view of their opponents' territory, it is essential to be vigilant about the locations of opponents. The term "fog of war" refers to the lack of information in this regard. Human expert players divide decision-making tasks into micro- and macro-levels. At the micro-level, players control units individually, whereas, at the macro-level, they produce units and expand territories.

Although RTS AI has attracted a lot of research, the number of RTS games with open interfaces that allow AI development was very limited. This hinders the development of new AI techniques for popular commercial RTS games. The introduction of the BWAPI (Brood War API)[3] for StarCraft games changed the situation significantly, enabling international competitions for RTS AI (BWAPI, 2016). Since 2010, several international conferences have hosted special events (RTS game AI competitions with the BWAPI) (Ontañón et al., 2013). The AI program for StarCraft AI competitions must be able to respond within 55 ms. For human matches, there are no restrictions on the response time. However, proficient players usually execute more than 200 actions per minute (APM). This means that a player selects an action every 300 ms. Although most of the submissions in the early competitions were designed by tight hand-crafted rules, the events have progressed research in computational intelligence and AI for RTS games.

The best StarCraft AI bots from the competitions are still not as good as professional human players. Weber et al. reported that their EISBot achieved an average win rate of 32% against human opponents on the international cyber cup tournament ladder (Weber, Mateas, & Jhala, 2011). Because the AI competitions only consider games between two AI programs, the entries usually ignore human-level skills not yet implemented in the current state-of-the-art submissions. For example, early stage scouting behavior and build-order adaptation are very important skills in human player games, but they are relatively under-developed in the AI competitions (Park, Cho, Lee, & Kim, 2012). Similarly, the bots assume that the unit controls of the other programs may not be as sophisticated as a human.

2.2. Case-based reasoning

Case-based reasoning (CBR) (Kolodner, 2014) is an approach to solving problems that involves drawing on solutions based on similar previous problems. Furthermore, CBR includes revising past cases to solve new problems. Given a novel situation, a CBR system retrieves the most similar case from a library of previous cases and applies its solution to the problem. After analyzing the result, the selected case is revised and returned to the library. Generally, the process of CBR is divided into four stages: retrieval, re-use, revise, and retain. Each process has the following functions:

- Retrieval: Given a problem, search for the most similar case from past cases. In this stage, it is important to index the most similar case quickly and accurately.

[3] https://github.com/bwapi/bwapi, The Brood War Application Programming Interface, a free and open source C++ framework to interact with the popular RTS game StarCraft.

Table 1
Summary of unit micromanagement research (StarCraft and WarCraft).

Type	Reference	Unit control		Evaluation			
		Action	Navigation	Setting (*N* vs. *N*)	Number of unit types	Opponent	Map
Design	Weber et al. (2011)	Hand authoring		Testing as an integrated AI			
	Young et al. (2012)	Reactive planning	A* search	Testing as an integrated AI			
	Chruchill et al. (2012)	Alpha-beta search (Search constraint 5 ms)		2~8	1, 2	Scripted Bots	
	Churchill and Buro (2013)	UCT search (Search constraint 40 ms)		8, 16, 32, 50	1, 2	Scripted Bots	
	Nguyen et al. (2013)	Script	Potential flow	12, 24, 36, 48	2	Built-in AI Scripted Bots SkyNet	No obstacle or building
	Uriarte and Ontañón (2012)	Script	Influence maps	1 vs. 4 4 vs. 6	2	Built-in AI	An open square map
Learning	Ontañón (2012)	Maps	4 vs. 6				
	Shantia, Begue, and Wiering (2011)	Reinforcement learning for neural network		3, 6	1	Handcrafted AI	
	Gabriel, Negru, and Zaharie (2012)	Neural network with evolution		12	2	EISBot Overmind	2010 AIIDE competition
	Parra and Garrido (2013)	Expert-designed Bayesian networks (CPT trained from replays)		2 vs. 3	1		13 × 13 tiles, diamond-shaped arena, with fog of war
	Zhen and Watson (2013)	Neural network with evolution		12	2		2010 AIIDE competition
	Szczepanski and Aamodt (2009)	Case-based reasoning (expert involvement + about 25 cases)				Computer human players	Custom map
	Proposed method	Imitation learning with influence maps		8, 12, 16, 36, 37	1, 2	FreSCBot UAlbertaBot Skynet, Ximp Megabot, CruzBot, MooseBot Built-in AI	2010 AIIDE competition Python

*Setting (*N* vs. *N*) controls the number of units for each player. For example, if *N* is 8, each player starts the combat with 8 units.
*In StarCraft, each race has units that are used primarily for attack. For example, zealot and dragoon are key units in combat. The number of unit types is 1 if the AI player uses only one type of unit (e.g., zealot only or dragoon only). If the value is 2, it means that the AI player configures his group with different units (e.g., zealot + dragoon). Usually, a combination of two different units is stronger than single-unit troops. For example, the zealot unit is strong in short-distance combat, but the dragoon unit has very long-range weapons to attack units at a distance.
*Scripted bots are designed based on simple micromanagement skills (Attack-Closest, Attack-Weakest, Kiting, and so on). The details of each strategy can be found in (Chruchill et al., 2012).
*Built-in AI was included in the StarCraft: Brood War game.
*SkyNet, EISBot, Overmind, FreSCBot, Ximp, UAlbertaBot, MooseBot, CruzBot and MegaBot are names of participants in the StarCraft AI competition.

- Re-use: Apply the solution from the most similar case to the present problem.
- Revise: If it fails to solve the problem, part of the solution is revised.
- Retain: If the case solves the problem with the revised solution, save it in the case library.

The performance of CBR is affected by the scope of cases and the appropriate revision of cases for new problems. Thus, it is desirable that the system include many cases, their solutions, and proper retrieval and revision methods.

3. Related works

3.1. Unit micromanagement in RTS games

Imitation of human micromanagement skills is especially difficult. For humans, these constitute very sophisticated skills, developed from participation in many games. They require the consideration of both an individual unit's movement (navigation and action selection) as well as its cooperation during attack and defense. This means an immense number of possible cases. For example, each case may have a different number of units, with differing unit types, positions and health power, as well as a varying opponents' state. It is not trivial to build a model to generalize effectively on such diverse cases. Although it is possible to mine a limited number of common management patterns from human replays, it is hard to implement these rules into a program that generalizes the behaviors for new matches.

There has been considerable research on the design of micromanagement skills for AI bots in RTS games (Table 1). It is possible to formulate the problem as one involving task allocation in multi-agent systems (Rogers & Skabar, 2014). Successful entries in the StarCraft AI competition have used the Attack-Closest (UAlbertabot) and the Kiting (Skynet) (Chruchill, Saffidine, & Buro, 2012) strategies. They are both simple combat AI scripts. Attack-Closest involves attacking the closest opponent unit and Kiting is similar to Attack-Closest except it involves moving away when it is impossible to fire. Uriarte and Ontañón (2012) showed that a small number of units can win against a larger number of units by kiting control. They designed kiting control with influence maps and showed that one range unit killed four melee units.

The research on unit micromanagement can be categorized into two groups:

- Design: In this approach, micromanagement skills are implemented in a designed system. For example, a manual design uses micromanagement skills implemented based on expert domain knowledge. Simply put, a unit attacks the closest enemy unit within attack range. Additionally, control of the units can be formulated as a tree search, such chess or go. Although the complexity is greater than that for traditional board games, it can be approached using several advanced techniques for tree searches (e.g., Monte-Carlo Tree Search, Move ordering). Another design method is Potential Field (PF). The concept of PF derives from robot path planning and has been used to control units in RTS games (Hagelback & Johansson, 2008). In Preuss

et al. (2010), an influence map and flocking path findings were used in GLEST, an open-source RTS game.
- Learning: It is desirable to artificially learn unit control without human intervention. There have been several attempts at implementing the concept of unit control learning using artificial neural networks and Bayesian networks. Each model controls the behaviors of individual units.

In the Weber et al. (2011) and Young, Smith, Atkinson, Poyner, and Chothia (2012) studies, they used domain knowledge to design micromanagement skills. This required a domain expert to design the system. Because these systems were tested as an integrated system, it is difficult to determine the relative superiority of the micromanagement module. Churchill et al. (Chruchill et al., 2012; Churchill & Buro, 2013) used a special-purpose combat simulator developed to simulate thousands of future events. Advanced search techniques are promising approaches to micromanagement, but it is not easy to use an algorithm to determine the values of numerous variables (e.g., unit positions, commands, and targets) for bots. Instead, they can reach a simple decision for a group of units (retreat or attack). Nguyen, Nguyen, and Thawonmas (2013) designed their micromanagement system using potential flow. It defines an artificial flow caused by territory, buildings, and units and simulates the movement of units using idealized flow equations. To use the algorithm, it is necessary to define the type of flow model for each object and to identify its parameters.

Because micromanagement is a sophisticated skill, it is not easy to define the domain knowledge held by human experts. The learning approach involves learning the control behavior of units based on past experience (rewards or replays). Although several studies have been conducted from this perspective, they have relied on limited data or a limited number of test cases. In some cases, they still used expert domain knowledge to design the models. In Szczepanski and Aamodt (2009), the case library had only 25 cases, and these were compiled by human experts. In Parra and Garrido (2013), the topology of a Bayesian network was determined by experts, and only the parameters of the model were trained using replays. The test scenario was relatively small for use with AI bots.

We propose to use thousands of frames extracted from human replays and imitate the actions and navigations of the most similar cases. We further propose to extensively use the replays of human players. Although replays have been used partially to tune the parameters of Bayesian networks (Parra & Garrido, 2013), their use has been limited. Instead, these approaches have leaned toward the adoption of “reinforcement-style” learning by playing against each other or against scripted bots (built-in AI). As with the search approach, it is important to be able to meet real-time constraints with our method. In the search, the time bottleneck comes from expanding the search tree and simulating the outcomes based on the battle. However, in our approach, the primary concern is minimizing the time needed for the case-by-case comparison.

To date, there have only been a few studies on imitation learning for RTS games. This is still at an early stage of development, because of the complexity of imitation and the limited use of human replays. Gemine, Safadi, Fonteneau, and Ernst (2012) trained 58 neural networks from artificially generated training data to imitate the production strategy in the game StarCraft II. They collected data from computers players’ games and tested their performance against the built-in AI. Parra et al. used a Bayesian network to control units in a “2 units versus 3 units” scenario (Parra & Garrido, 2013). The network structure was designed by experts, and the parameters of the model were learned from replays. Compared with our work, the 2-units-versus-3-units combat was a relatively small setting.

3.2. *Case-based reasoning for game AI*

The CBR approach has several potential advantages for designing game AI. Because CBR uses case data from previous plays, it does not require the design of manual behavior. This also means that although researchers do not have domain-specific knowledge about the game, they can design behaviors for game AI. CBR is a kind of lazy learning algorithm that delays generalization until a query is made. It is easy to adapt to the new problems by changing the case library. However, it requires considerable space to store the cases, suffers from noise/error in case handling, and is slow to evaluate cases.

Gillespie, Karneeb, Lee-Urban, and Muñoz-Avila (2010) introduced a stochastic policy for CBR in a first-person shooting game. The main contribution was to introduce a general framework to provide probabilistic case representation. Romdhane and Lamontagne (2008) combined CBR with reinforcement learning to evaluate the weight of patterns for TETRIS. Compared with RTS games, TETRIS can be played using a relatively small number of cases ($\sim$5000 cases). Watson, Rubin, and Robertson (2012) proposed CBR specifically designed to play only heads-up poker. It can dramatically reduce the number of betting patterns that occur.

In the RTS game domain, CBR has been used to generate the behaviors of units and strategies. Ontañón et al. used CBR to generate behaviors in WARGUS (Ontañón, Mishra, Sugandh, & Ram, 2007), a WarCraft II mod. In the study, they created cases from expert demonstrations and annotations. The expert annotated the trace to determine goals to pursue and their associated behaviors. Using these high-level cases, they could generate the goal and detailed behaviors for the current situation. Ontañón et al. attempted to learn plans, represented by Petri-Net, from the demonstration of expert players (Ontañón et al., 2009). The system was tested on S2, a simplified WarCraft II. Wender and Watson (2014) proposed a hybrid reinforcement learning (RL) and CBR approach for controlling combat units in StarCraft. The resulting hybrid RL and CBR agent performed well when compared with an RL-only agent and the built-in AI. In their tests, Scenario A involved a weak combat unit against six slower, stronger enemy units, and Scenario B used three agent units and eight opponents. Compared with our work, this work used relatively simple testing conditions and did not exploit replays to learn the micromanagement skills.

In a simulated soccer league, the CBR has been used to imitate the plays of other agents. In the league, the agent must send an action before the end of the time cycle (30 ms). Although a robot soccer system requires real-time processing, there are several distinct differences with real-time strategy games. The two systems have different definitions of vision, numbers of units and types, sources of the case library, case structures, and response-time requirements. Floyd, Davoust, and Esfandiari (2008) proposed an algorithm to enable CBR to be performed by individual imitative agents. In their study, they applied a case base size threshold ($\sim$3000 cases per cycle) to send an action in time. Thus, their work attempted to process CBR in real-time with a threshold. Our study differs from this work in its use of an influence map, application to real-time strategy games, and representation of cases. Davoust, Floyd, and Esfandiari (2008) proposed an image-like representation based on histograms of objects with customizable granularity for simulated robot soccer. They showed that their representation was highly efficient for scene comparison. Floyd, Esfandiari, and Lam (2008) tested different preprocessing techniques for robotic soccer agents. They reported that a histogram representation significantly outperformed the raw representation. Although the histogram-based representation can be useful for hastening the imitation comparison, we adopted an influence map, known to be useful for RTS research, as a basic representation for a spatially aware system.

Table 2
An example of a scene extracted from a replay. It is possible to extract one scene per eight frames from a replay.

ID	Owner's base (o'clock)	Type	Position	Health power	Command	Moving direction
0	12	Dragoon	(113,1134)	180	Stop	Right
1	12	Dragoon	(113,1218)	180	Move	Right-Up
2	12	Zealot	(147,1217)	160	Attack	Left-Up
3	6	Dragoon	(60, 875)	180	Stop	Left-Up
4	6	Zealot	(80, 758)	160	Move	Right-Up
5	6	Dragoon	(200,1025)	180	Attack	Down
…		…	…	…	…	…
10	6	Zealot	(200,1218)	160	Stop	Left-Up

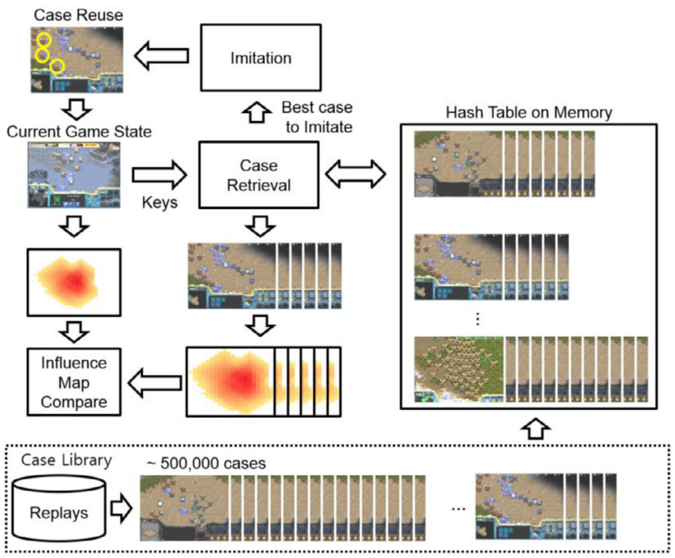


Fig. 1. Influence map-based case search with in-memory hash table.

4. Imitation-based unit micromanagement

Imitation searches seek the most similar situations among replays and then mimic the actions taken by the human players. This process can be divided into offline and online processing. During offline processing, the agent makes a case library by storing the pre-processed data of existing replay files. In online processing, the agent searches for the case in the case library that is most similar to the current game state. An influence map representation was adopted to analyze the influence of units spatially and to allow high-speed case comparison. Additionally, case hashing reduced the time it took to find the appropriate moments to imitate.

The extracted case is then applied to the current game, modifying the action of the units based on that case. After finding the best case, it is necessary to imitate the actions/navigation of the units considering the difference between the current and the reference frame. The best case may not be exactly the same as the game state of the current scene. For example, there may be different numbers of units, types, and positions. It is important to assign each unit in the game state of the current scene to a unit in the associated best case. A policy must be in place to handle the event of a unit that is unmatched. This adjustment is essential to allow the imitation of slightly different situations. Fig. 1 illustrates the proposed imitation process.

4.1. Extraction of scenes from replays for the case library

Replays can be downloaded from well-known replay repositories or game portal sites. Because a single replay file contains approximately 14,400 frames (10 min/replay), it is possible to construct a large case library using only a small number of replays. It is possible to extract all the game states of units, buildings, and scores for all players at each time frame from the replay. In this work, we used only unit-related data sampled every eight frames. Because sequential game states of a scene are likely to be similar with small differences, we use subsets of game states by sampling every eight frames. One scene is created as a vector of all units' attribute data from each frame sampled. The attribute data includes type, x-y coordinates, health, commands, and the moving direction of each unit.

The scene is extracted using a replay analyzer.[4] Table 2 shows an example of a scene extracted from StarCraft replays. In the example, there are 10 units on the map. The player with the 12 o'clock base has three units (two dragoons and one zealot), and the other player with the 6 o'clock base has eight units. In detail, the dragoon unit owned by the top player is located at (113, 1134), left-bottom, with full-health power. The dragoon's initial health power is 180. The player assigned a "Stop" command to the unit, and the moving direction was "right."

A scene is regarded as one case in this study. A case consists of three components, and the case structure is defined as follows.

Game State $G = \{i, o\}$

- i: Unit's unique identifier used to track the unit during game play.
- o: Owner's base (o'clock) identified by the location of owner's base.

Unit State $U = \{t, p, h\}$

- t: Unit type. In StarCraft, each race has about eight ground unit types and five or six air unit types. The type should be the same as one of them.
- p: Unit's position: x-y coordinates of each unit on the map. Usually, the StarCraft map size is 4096×4096 pixels. Each value ranges from 0 to 4095.
- h: Unit's health power. Initially, the value is set at the maximum, which is different on different unit types and decreases when the unit is damaged.

Command $C = \{c, d\}$

- c: Command. The current command assigned to the unit ("stop," "move," or "attack.") Although there is a "patrol" command, it is ignored because it is rarely used.
- d: Direction of the unit. It is one of eight directions (up, down, left, right, right-up, left-up, right-down, and left-down).

4.2. Similarity measure for case matching

In case retrieval, it is necessary to calculate the similarity between two cases (scenes). Although a table-based representation

[4] We did not use the BWChart or LMRB replay analyzer software. Instead, we extracted the information from replays using BWAPI (Brood War API).

```
Input: Scene($t$) // A scene at time frame $t$
Output: IM$_1$~ IM$_n$ // Influence Map of $n$ Players
SIZE: The width/height of IM

Initialize(); // set all IM values to zero

For the $i^{th}$ Player {
 For($x$ = 1; $x$<=SIZE; $x$++) {
 For($y$ = 1; $y$<=SIZE; $y$++) {
  For units owned by the $i^{th}$ Player in Scene($t$) {
  ($unitX$, $unitY$) = unit.Position();
  alpha = 1 – (|$x$-$unitX$|+|$y$-$unitY$|)×0.1;
  IM$_i$[x][y] += $\frac{unit.HealthPower()}{unit.MaxHealthPower()}$×alpha;
  }
}}}
```

Fig. 2. A pseudo-code to calculate IMs from a scene.

can be used in game state comparison, there are several reasons to devise a different form of representation. Because the game state of each scene contains different numbers and types of units, it is difficult to define a unified comparison formula. Each row of the table only contains information about an individual unit and does not include any additional high-level features (e.g., cooperation of units, spatial distribution of units etc.). In this work, we propose using an influence map to represent the unit distribution.

Influence maps (IMs) have been used to represent the numerical influence of each position in RTS games (Miles & Louis, 2006; Miles, Quiroz, Leigh, & Louis, 2007). We use IMs to spatially analyze the influence of units and their surrounding terrain. In our model, IMs help represent higher dimensional unit information, similar to a human's viewpoint, facilitating effective imitation. Also, IMs solve the technical difficulties of comparing frames. For example, between-unit comparison can be difficult. This makes it difficult to carry out most imitations through a direct unit-to-unit comparison. However, by using IMs, the influence each unit has is marked on the map and covers a larger area than the unit itself. This helps to enable more efficient searches for the most similar case by requiring less data. Also, IMs can enhance the performance of a bot through the use of spatial reasoning, which can have a big effect on the results of combat.

Each scene can be transformed into an IM. When memory space is sufficient, it is possible to create a 4096×4096 IM. However, we reduced the resolution to 64×64 to save memory. A total of 16 K bytes ($64 \times 64 \times 4$ bytes) is required for one IM. If the game is played by N players, each scene would be transformed into N IMs. Each IM is created using the units owned by the player. For example, if the game were a match between the top (12 o'clock base) and the bottom (6 o'clock base) players, there would be two IMs for each scene: one from the top player's units, and the other from the bottom player's troop.

Fig. 2 shows a pseudo-code of the influence map calculation algorithm used in this paper. The input of the algorithm is a scene at time frame t, and the outcome is n influence maps created for each player. In this IM process, let ($unitX$, $unitY$) be the actual position of the unit (transformed into 64×64 resolution). If the unit is close

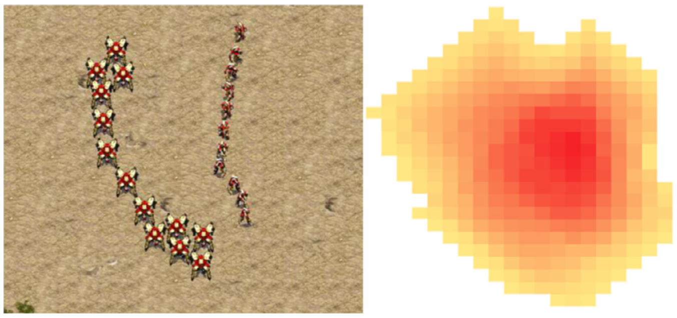


Fig. 3. StarCraft unit formation (Left) and corresponding influence map (Right). Yellow represents area of lowest influence of the units and red marks the highest influence. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

to a specific location, the unit has a direct impact on it. In addition, the IM value is proportional to the unit's health. Fig. 3 shows an example of an IM made using the calculation. The figure on the left shows the configuration of units (dragoons and zealots), and the figure on the right shows its corresponding IM. The strongest point of influence is clearly apparent.

The similarity of two scenes is calculated using the influence maps. If we define IM_{A1} as the first player's IM from Scene A, the similarity would be defined as follows. It is based on the sum of the Euclidean distances between corresponding IMs. As in Fig. 2, IMs were created through the influence values of units. In order to find influence maps from the data base that corresponded to the influence map of the present situation, first we filtered the data by the average position of units, and the number of units contained in the influence map. Then we compared the Similarity (Eq. (1)) between our present influence map and the influence maps in the data base, and selected the map with the highest similarity to the present map. Thus, we selected corresponding IMs to our present IMs based on the value of their positions.

$$\text{Similarity} = \frac{1}{\sum_{i=1}^{n} \sum_{x=1}^{SIZE} \sum_{y=1}^{SIZE} (IM_{Ai}[x][y] - IM_{Bi}[x][y])^2 + 1} \quad (1)$$

If Scene A were the current game state, and Scene B were from the case library, the IM calculation from Scene B would consider only unit types seen on Scene A.

4.3. Case retrieval

The goal of this step was to find the case with the greatest similarity to the game state of the current scene from the case library. Because the AI needs to respond in a very short amount of time, this step needs to be processed in real-time. When the AI program begins the game, the cases in the case library (in raw game event format) are fully loaded into main memory. Although all of the data are in memory, it is not realistic to compare the game state of the current scene with all of the cases in real-time. Instead, we propose using a case hashing algorithm to reduce the case search space significantly. The hash function maps the keys to the bucket of scenes. The selected cases are converted into influence maps and then compared with the IM of the current game state.

The goal of hashing is to select a subset of cases from the case library loaded in memory. This step is essential in order to meet the real-time constraint. In the hashing, we select cases that satisfy conditions derived from the current state of the game. In this work, we use multiple keys in the database search.

- ***The number of units for each type***: A key is defined for each unit type. For example, if the player currently has zealots and

Table 3
Sixteen levels of hashing tolerance (initially, tolerance was set at the 3rd level for all keys). The '± 1' means that the system accepts cases whose key distance is between -1 and $+1$.

Level	Number of zealots	Number of dragoons	Average x position	Average y position
0	Exact matching	Exact matching	Exact matching	Exact matching
1	± 1	± 1	± 1	± 1
2	± 2	± 2	± 2	± 2
3 (default)	± 3	± 3	± 3	± 3
…	…	…	…	…
15	± 15	± 15	± 15	± 15

dragoon units, the number of keys is two. The key value for zealot unit type is set as the number of zealot units on the current game scene.

- ***The center position of units***: It is calculated as the average x and y position of all units. Each key is the average of the x position and the y position.

For example, if the number of zealot and dragoon units is two and three, respectively, and the center position for each group is (20, 30), the key values are 2, 3, 20, and 30.

In the event that the number of cases exceeds the MaxScene, it returns only the maximum number of cases and the rest are ignored. In this study, we set the MaxScene as a large number, and our system retrieved not only cases that exactly matched the keys but also cases with tolerable differences. In fact, there is little chance that cases with exact matches would be ignored. The algorithm continuously adjusts the tolerance of case retrieval during game playing. For example, if the tolerance were low, it would extract only cases whose keys are very similar to or exactly match the current keys. However, if tolerance were high, cases with tolerable differences would also be considered. Our observation is that the number of cases retrieved varies over the game. For example, it can return enough cases whose keys are so close that no tolerance is necessary. However, if it returns too few cases and is likely to miss the most similar influence map, it would be necessary to allow tolerance. The hashing tolerance has 16 different levels (Table 3[5,6]).

Fig. 4 illustrates an algorithm that searches the case database for the case to imitate. The input of the algorithm is the current scene, and the output is the most similar scene found in the case library. The first step is to get the key values from the current scene (4th line). The next step is to get a bucket of scenes using the key values and hashing tolerance (6th–8th lines). If the number of keys were M, it would return M buckets of scenes. In the case retrieval, the intersection of the all the scenes from the buckets is used. It is defined as *Scenes* (9th line). The most similar scene is found among the *Scenes* using the similarity measure based on the influence maps (12th–14th lines). The final step involves adjusting the hashing tolerance of the key with the minimum (if too small) or maximum (if too many) number of scenes for the next search (15th–25th lines). Finally, it returns the *Scene* (similar).

4.4. Case reuse

Imitation is the final step in determining the actions and movement of the units for the current case. The case search returns a single case for imitation. The raw game events for the selected case can then be extracted from main memory. During imitation, the most difficult problem to deal with is that the two cases (current and similar scenes) could have different numbers and types of units, as well as varying positions for those units.

```
1   Input: Scene(current)
2   Output: Scene(similar) from case library
3
4   Keys_{1~m} = Hash_Function(Scene(current));
5
6   For $i^{th}$ key {
7    $Scenes_i$ = Bucket_Search(key_i, tolerance_i);
8   }
9   $Scenes = \bigcap_{i=1}^{m} Scenes_i$;
10
11  // Find the most Similar Scene
12  Scene(similar)
13  $= \mathrm{argmax}_{1\sim Scenes}\, Similarity(Scenes(i), Scene(Current))$
14
15  // Adjust Hashing Tolerance Level
16  If(Scenes < $C_1 \times$ MaxScene) {
17   min = $\mathrm{argmin}_i\, Scenes_i$
18       $tolerance_{min}$++;
19  }
20  If(Scenes > $C_2 \times$ MaxScene) {
21       max = $\mathrm{argmax}_i\, Scenes_i$
22       $tolerance_{max}$--;
23  }
24
25  Return Scene(similar)
```

Fig. 4. A pseudo-code for case retrieval ($C_1 = 0.3$, $C_2 = 0.7$).

To solve this, the imitation process (copying actions from a scene similar to the current scene) is conducted for each unit type separately. Only units of the same type are imitated.[7] As a result, there is no chance that a zealot unit will copy the actions of a dragoon unit. When there are more than two units of the same type, the closest one is assigned. Then, the unit follows the action ("attack," "move," or "stop") and moving direction of the matched unit by copying the original movement vector. When the distance (Manhattan distance > 4) between the unit and the matched unit is too far, the action copy is not allowed. When there is no unit of the same type, the unit is ordered to the location with maximum influence. In professional players' micromanagement, they usually do not care about "whom to attack" and use the default "near-by attack." In contrast, they are extremely careful in positioning units. Following the human players' style, we used the "near-by" attack option.

Fig. 5 shows an example of cases merged for unit matching. The size of map is 12×12. The blue (A, B, C, and D) cells represent units that are of the same type in the current game case, and the red (F, G, H, and I) cells are from the imitation case. A cell is located in the same position as one red cell. This means that A is exactly matched with the unit on the red cell and follows its action. This unit on the red cell is then deleted from the candidate list because it has been matched once. The yellow cells show the distance (Manhattan distance) between the units (in this work, a maximum expansion of four cells is allowed). C is assigned to F

[5] Zealot is a basic melee unit of Protoss which is one of the races of StarCraft
[6] Dragoon is a basic range unit of Protoss which is one of the races of StarCraft

[7] We also tested a version that allowed unit matching regardless of unit type. The performance was similar or a bit worse because of some unmatching between units of different types.

Table 4
Comparison of the AI Bots used in the experiment (P: Protoss, T: Terran, and Z: Zerg races, rankings indicate the results each bot received from the StarCraft AI Competition).

Competition	Name	Race	Ranking
–	Built-in AI from StarCraft	P, T, Z	–
AIIDE 2010 Micromanagement Track	FreSCBot	P, Z	1st
AIIDE 2014 StarCraft Competition	Ximp	P	2nd
	Skynet	P	5th
	UAlbertaBot	P	7th
	MooseBot	P	9th
	CruzBot	P	15th
CIG 2016 StarCraft Competition	MEGABot	P	Final Stage 7th

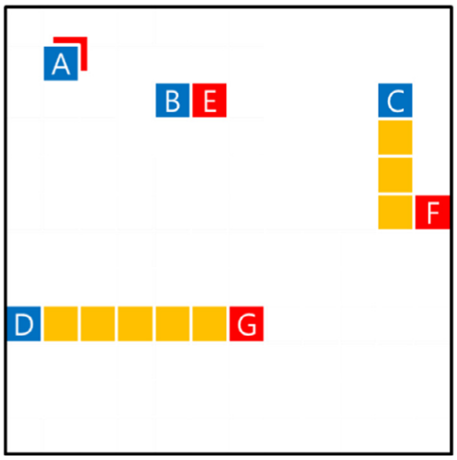


Fig. 5. An example of two cases merged (current game and imitation cases). The blue cells are the location of units in the current game, and the red cells are imitation cases. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

because the distance is four in the horizontal and vertical directions. However, D is not assigned to G because the distance is six, which is larger than four. There is no unit that can match D, so D follows the action of the closest unit (A) in the current game.

5. Experimental results and analysis

In this study, we conducted two different experiments using micromanagement tasks. Our bots only used imitation to play the games, without the use of explicitly hand-coded rules. We used a machine with an Intel Core i7-4790 CPU at 3.60 GHz running on Windows 7 Professional Edition with 32GB of DDR3 RAM. Our program runs in single-thread mode. The parameter *MaxScene* was set to 1000. Experimental metrics are defined as follows.

- Unmatched unit ratio (%): Average percentage of units not matched to units from the case library.
- Average tolerance levels: During the game play, the AI bots continuously change the tolerance for case retrieval. It averages the tolerance levels during a game.

Table 4 summarizes the AI bots used in our experiments for comparison purposes. They include entries from both the AIIDE 2010 micromanagement competition and the 2014 AIIDE "full game" AI competitions. The 2010 competition was used because it was the last micromanagement AI competition and was specialized for the skills. In the competition, all entries were asked to prepare to play both the Zerg and Protoss races. Additionally, they were designed to play both 9 versus 9 and 12 versus 12 matches. In the 2014 competition, they were designed for a full game match and usually prepared many skills: scouting, build orders, combat, and resource management.

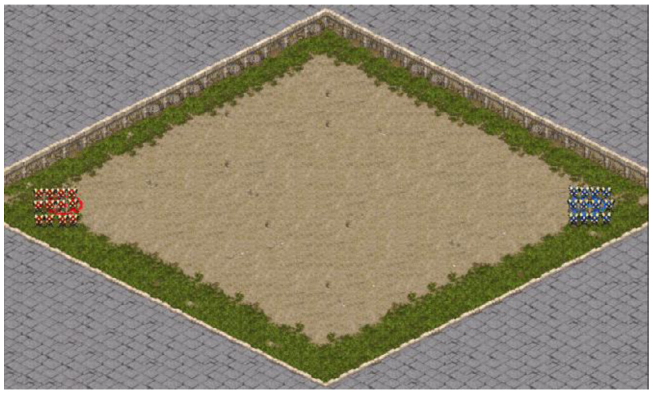


Fig. 6. A map used in the 2010 AIIDE micromanagement competition.

In the first experiment, we adopted the map from the 2010 AIIDE (Artificial Intelligence and Interactive Digital Entertainment) StarCraft micromanagement AI competition. Because the map has not been widely used by human players, we used two expert-level human players to create the replays for the map. The fog of war was activated when we extracted the raw game events from the replays and during the experimental games between the two AI bots. In the test, only the 2010 competition entries were used because full-game entries did not work well on the specialized maps.

The micromanagement competition for AIs has not been held since 2010. The researchers had no choice but to use results from the most recent 2010 competition as entries for our experiment.

In the second experiment, we adopted the famous "python" map widely used by human players. The replays were downloaded from an online replay sharing portal site. Although the replays used in the first experiments only had combat cases, the replays in the second experiment contained the "full" game played by two players. This means that there are cases for scouting, combat, production, construction, etc. In this work, we did not apply an additional algorithm to categorize the unrelated cases from the case library. Instead, all of the cases from the full game replays were loaded into main memory.

We conducted statistical significance tests (one sample and two sample student's t-tests) for bot comparisons in the first and second experiments. In the first experiment, the degrees of freedom are defined as ($N - 1$). In the second experiments, the degrees of freedom are defined as ($N - N - 2$). If the t-value is larger than the value in t-table, it is statistically significant.

5.1. Imitation with combat-oriented replays

In 2010, AIIDE StarCraft AI competition had a special track focusing on micromanagement. It simplified the map settings allowing the two AI bots to focus on micromanagement skills. Fig. 6 shows the map used in the specialized competition. It is a diamond-shaped map. Although the map is useful for testing mi-

Table 5
One sample t-test of the imitation player against AI bots in the micromanagement competition map (10 games × 10 runs).

Opponent	12 vs. 12 (dragoon only)		36 vs. 36 (dragoon only)		36 vs. 36 (zealot and dragoon)	
	T value	Win rates (%)	T value	Win rates (%)	T value	Win rates (%)
FreSCBot	6.5	76 ± 12.65	X	100 ± 0.00	N/A*	N/A*
Built-in AI	10.2	85 ± 10.8	49.0	99 ± 3.16	27.0	95 ± 5.27

*N/A means the bots were not designed to play the combination of units.
* Degrees of freedom = 9, p value = 0.05, t-table value = 2.262, if the t value is larger than the value of 2.262, it is statistically significant.

Table 6
Performance of the imitation system against the built-in AI (average of 200 games = 40 games × 5 runs) in 36 vs. 36 (zealot & dragoon) mode.

Number of cases	Response time (sec)	Unmatched unit ratio (%)	Avg. tolerance level	Win rates (%)
100	0.008	75.4%	13.5	89 ± 13.7
500	0.012	38.8%	13.5	88 ± 14.8
2000	0.016	24.9%	9.9	98 ± 2.9
8000	0.020	25.1%	5.9	94.5 ± 4.0
33,539	0.019	24.1%	3.1	95 ± 3.2

cromanagement skills, it is not easy to get human player replays for the map. In this work, we had two expert human players create the replays. One of the players was a professional StarCraft II gamer and the other had a long experience of game playing.

In total, 80 replays were collected from the expert matches. They included 20 games each for 12 vs. 12 dragoons, 24 vs. 24 dragoons, 36 vs. 36 dragoons, and 36 vs. 36 dragoons/zealots. In the dragoon and zealot combination, there were 50% dragoons and 50% zealots. In the map, fog of war was enabled so the replay extraction took this into consideration. This means that the enemy's unit information was not stored in the raw game event table if it was invisible to the player. In this manner, it was possible to test the performance of the imitation learning with fog of war. The size of the raw game event was approximately 0.5GB and the total number of cases (scenes) was 33,539. Because the map is specialized for combat, all of the cases were related to combat.

Table 5 indicates a sample of t-test result of games played between the proposed imitation bot and the winner bot of the 2010 StarCraft Competition. For all of the matches, fog of war was enforced. The imitation bot performed better in the large-scale combat (36 vs. 36) than in the small-scale combat (12 vs. 12). It achieved on average a high winning percentage in the zealot-dragoon combination tests as well as in the dragoon only tests.

Despite the fog of war, the imitation was able to perform well on the micromanagement tasks. Table 6 summarizes the performance of the imitation system based on the number of different cases used. The winning percentage decreased slightly when fewer scenes were used (e.g., 100). It was possible to achieve a winning percentage of approximately 95% if the number of cases was 2000 or more. However, the standard deviation (14.8%) was a bit high with only 500 scenes. Overall the results showed that the imitation player was able to achieve the optimal performance using only 2000 cases (just 6%) of the total 33,539 cases.

In the test with only 100 cases, most of the units (75.4%) were unmatched during imitation. However, the unmatched unit ratio was less than 25% when 2000 or more cases were used. The average response time was 0.019 s for 33,539 cases. This was faster than professional human players (~0.3 s) and met the requirement for the StarCraft AI competition (~0.042 s).

5.2. Imitation with full game replays

In the second experiment, "full game" replays were downloaded from a replay-sharing portal, where 300,000 StarCraft replay files are available to the public (http://bwreplays.com). "Python" is one of the most widely used maps in StarCraft. We downloaded 216 replays played on the "Python" map for Protoss vs. Protoss matches. The replays satisfied the following conditions:

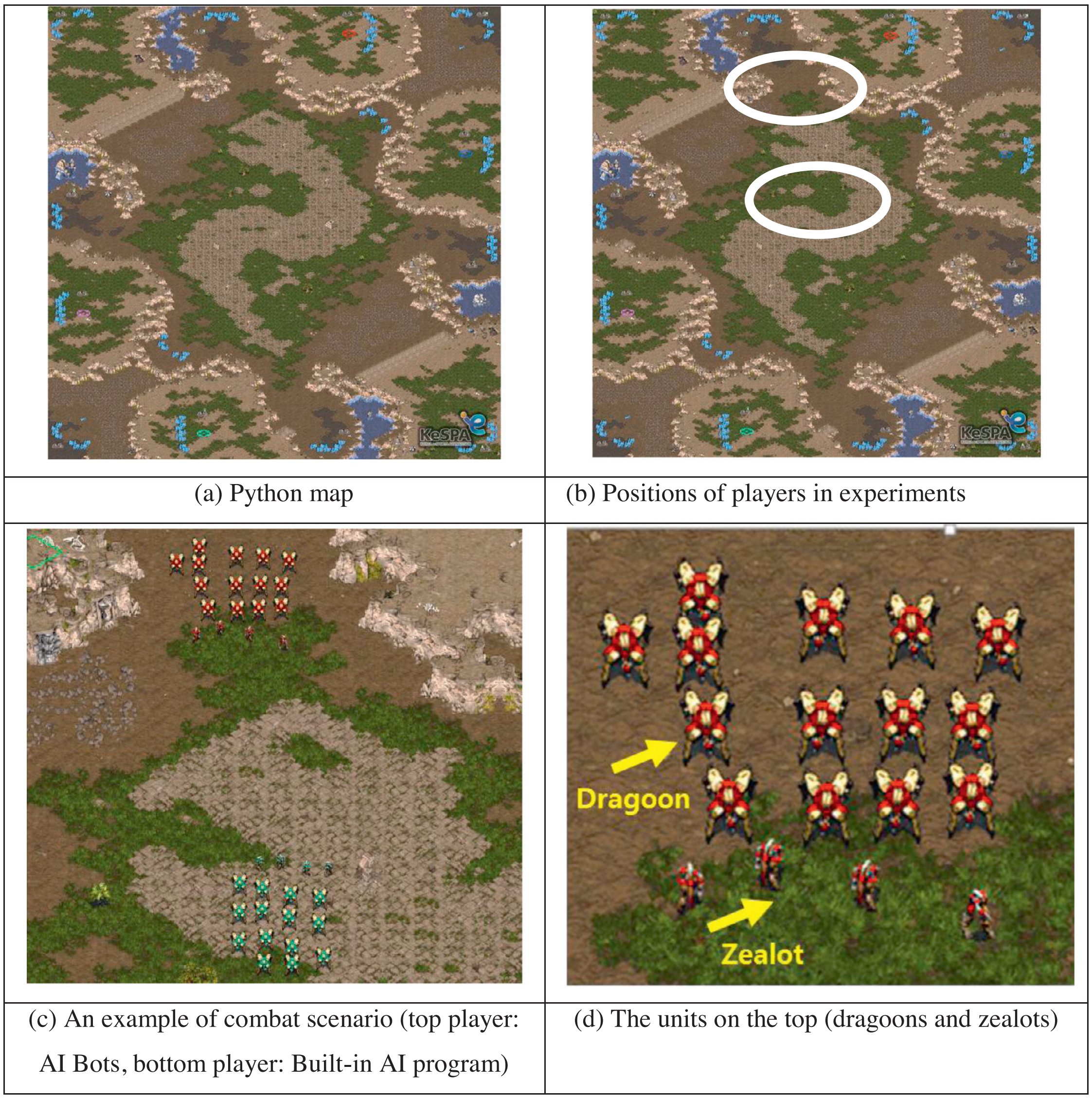
- Game type: one vs. one full-game
- Race: Protoss vs. Protoss
- Map: Python
- APM (actions per minute): More than 170 APM for both players
- Player location: Top vs. Bottom

To consider the proficiency of human players, we chose replays with more than 170 APM for both players because APM indirectly reflects a player's proficiency in the game. Using the BWAPI, the raw game events were extracted from the replays. Fog of war was not considered during extraction. Each case (scene) was sampled every eight frames. The size of the raw game events was 3.7GB and the number of cases was 0.5 million (500,000).

Fig. 7 shows a screenshot of the "Python" map. In our testing, each AI played against the built-in AI program. When we tested the AI bots, their units were located at the entrance of the base on the top, and the built-in AI was located in the center of the map. The built-in AI (center) attacked the AI bots (top) in order to eliminate the fundamental building (Nexus) behind the entrance. Because the entrance at the top was narrow and the center area was wide, it was important to control the units efficiently to win the combat. For the test, three different combinations of units were defined: small-scale, middle-scale and large-scale combat. During the combat, it was necessary that the zealots fight in close quarters with the enemy's units while the dragoons fought from a distance.

- Small-scale combat: 7 dragoons and 1 zealot
- Middle-scale combat: 14 dragoons and 4 zealots
- Large-scale combat: 21 dragoons and 6 zealots

Table 7 shows the two student's sample t-tests of the AI bots against the built in AI. For the three settings (small-, middle-, and large-scale), the AI bots played 100 games (10 games × 10 runs). If the t-value of each game is higher than the t-table value of 2.010, the imitation bot can be said to have outperformed its competing AI. The table shows that the imitation bot (IM (1000)) outperformed the other AI bots in every case regardless of the size of the combat, except in one case (IM 1000 verses Ximp where t-value of 1.0 is lower than the t-table value of 2.101 in large-scale). In micromanagement, it is important for the player to position units efficiently to maximize the power of attack. For example, players will



(a) Python map

(b) Positions of players in experiments

(c) An example of combat scenario (top player: AI Bots, bottom player: Built-in AI program)

(d) The units on the top (dragoons and zealots)

Fig. 7. "Python" map (a) and an example of combat scenario to test the micromanagement skills (c).

Table 7
Two sample t-tests of the imitation player against AI bots in the micromanagement competition map (10 games × 10 runs).

Entries	Small-scale		Middle-scale		Large-scale	
	T value	Win rates (%)	T value	Win rates (%)	T value	Win rates (%)
IM (1000)	N/A	95 ± 7.07	N/A	95 ± 5.27	N/A	72 ± 18.14
Ximp	7.6	55 ± 15.09	12.1	27 ± 17.03	1.0	65 ± 14.34
Skynet	9.9	50 ± 12.47	13.0	24 ± 16.47	2.6	53 ± 14.18
UalbertaBot	10.1	39 ± 15.95	32.1	6 ± 7.00	12.6	0 ± 0.00
MooseBot	6.5	60 ± 15.63	14.0	48 ± 9.19	4.0	40 ± 17.64
CruzBot	42.5	0 ± 0.00	5.7	58 ± 19.89	11.5	2 ± 6.32
MegaBot	7.8	44 ± 19.55	12.0	28 ± 16.87	4.9	31 ± 19.12

* IM 1000 is the standard for conducting the sample tests. Therefore, the t value of IM (1000) cannot be defined, as it cannot compare against itself.
* Degrees of freedom = 18, p value = 0.05, t-table value = 2.101, if the t value is larger than the value of 2.101, it is statistically significant.
* IM(x) means that the imitation learning sets the *MaxScene* as x.

form concave or convex curves, spread units, or ball up in order to maximize attack. However, AI bots tend to focus solely on whom to attack (closest or weakest) because positioning at a human level is highly difficult. Our observation shows that our systems were able to learn to position units in a way similar to human players, which was the main reason for its outperformance.

AI bots for full game mode require many different skills: scouting, micromanagement, resource management, and tactical decision making. When the bots play a full game, they usually use all of them to defeat opponents. In full game mode, the AI bots may be better than the built-in AI. However, our experiments were designed to test only the micromanagement skills in combat sit-

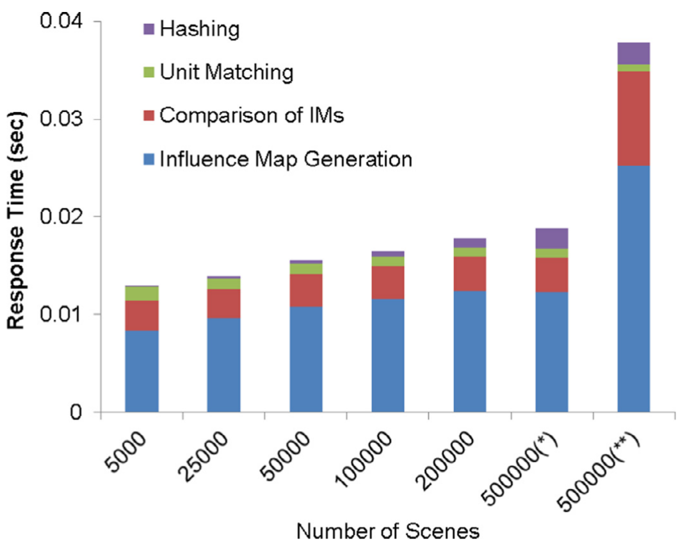


Fig. 8. Response times of the imitation AI on different numbers of cases (*$MaxScene = 1000$ and **$MaxScene = 2000$) in the Python Map with large-scale combat scenarios (average of 200 games: 40 games × 5 runs).

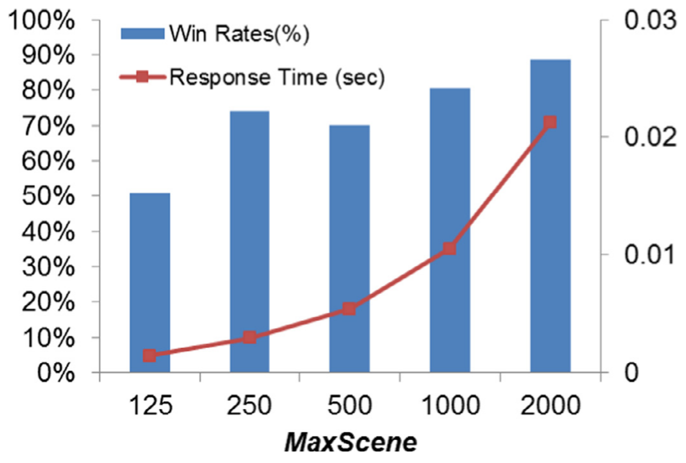


Fig. 9. The win ratio and response time with different *MaxScene* values (average of 200 games) in the Python Map with large-scale combat scenarios and the imitation system using 500,000 cases.

uations. It showed that some AI bots have relatively underdeveloped micromanagement skills. Additionally, they attempted to finish games as early as possible using predefined strong build orders and to avoid very complex combat situations in long-run games. This is the reason that some AIs performed worse than the built-in AI in the micromanagement tests.

Additionally, because the imitation was based on cases from full games, it contained cases for scouting, combat, production, and construction. Although there was no rule to filter out the cases unrelated to combat before the imitation, it was still possible to achieve a high winning percentage, thus supporting the idea that influence map-based comparisons are successful at finding similar combat-related cases. Although it shows the possibility of robust performance, it is still necessary to reduce the error/noise: one approach is to use only replays from expert players' games. Recently, we studied the reliability of the imitation system (Oh & Kim, 2015).

5.3. Response time

The response time of the imitation learning was analyzed to determine if the resulting AI can meet the real-time constraint. The response time was analyzed for the second experiment because it used more cases than the first. Fig. 8 shows the response time of the imitation AI. The average response time of the AI was measured using different numbers of cases. When the imitation system used only 5000 cases, response times were within 0.01 s. The influence map generation was one of the critical time-consuming functions. The time required to generate IMs increased slightly based on the number of scenes because of the additional scenes that have to be passed through the hash keys. When *MaxScene* is adjusted to 2000, the time for IM generation doubled. The second greatest time consumer was the influence map comparison between the current game state and the cases from the bucket. The hashing takes a very small amount of time. The time burden of the unit-matching task was relatively small.

In 500,000 cases, the response time was approximately 0.19 s when *MaxScene* was 1000. Even for larger numbers of cases, the response time could be maintained within the single frame (0.042 seconds). This is because the system limits the number of influence maps generated and compared with the *MaxScene* parameter. Also, the hashing significantly reduced the case search time. However, response time doubled when *MaxScene* was set to 2000 and the average response time was slightly under one frame.

5.4. Sensitivity to parameter selection

To load the influence maps, the machine needs to have additional main memory. In the second experiment, the raw game data required 3.7GB of memory. In our study, we needed to calculate two types of influence maps (top vs. bottom players) per scene, and each IM required 16,384 bytes (64 × 64 × 4 bytes). For 500,000 cases, it required ~15GB. In total, the program used 18.7GB of memory. Although this is not small, the cost of memory has been decreasing, making it affordable for desktop machines. Because of the memory issue, we did not pre-calculate the influence map of scenes.

Table 8 summarizes the statistics of the imitation AI for the different numbers of cases used. The unmatched unit ratio means the average percentage of units not matched to the units from the case library. For the test with 5000 cases, the unmatched unit ratio was roughly 34%. However, it was only around 9% for 500,000 cases. If the tolerance level is high, it means that the case search allows more tolerance to increase the number of cases for comparison. From the case library, approximately half of the *MaxScene* cases were selected for comparison with the current game case. The winning percentage against the built-in AI was relatively low with a smaller number of cases such as 5000.

The winning percentage was in the range of 81–97% between 25,000 and 200,000 cases. With a smaller number of cases (5000–100,000 cases), the standard deviation for the winning percentage was high (2.2%–28%). However, the imitation using 200,000 cases achieved a high winning percentage (97%) with a small standard deviation (3%). With 500,000 cases, it was important to set an appropriate *MaxScene* to obtain a good performance. The imitation using 500,000 cases (*MaxScene* was doubled) achieved a high winning percentage (93.5%) with a small standard deviation (3.4%). Fig. 9 shows the winning percentage and the response time of the imitation system with 500,000 cases for different *MaxScene* values.

5.5. Testing on unseen maps

In this test, we created the case library from the replays of the Python map and tested it on a different map ("Lost Temple") not used in training (Fig. 10). The experiment was conducted in a large-scale combat setting. Table 9 shows the results of combat. Compared with the previous results, the unmatched unit ra-

Table 8
Sensitivity to the number of cases used (average of 200 games (40 games × 5 runs)) (**MaxScene* = 1000 and ***MaxScene* = 2000) in Python Map with large-scale combat scenario.

Number of cases	Unmatched unit ratio (%)	Avg. tolerance level	Win rates (%)
5000	33.7	6.8	72.5 ± 28.1
25,000	18.8	5.3	97.5 ± 2.2
50,000	17.6	4.3	81 ± 17.4
100,000	12.9	3.8	91 ± 7.0
200,000	8.8	3.2	97 ± 2.9
500,000(*)	11.1	2.5	81 ± 7.5
500,000(**)	8.8	3.0	93.5 ± 3.4

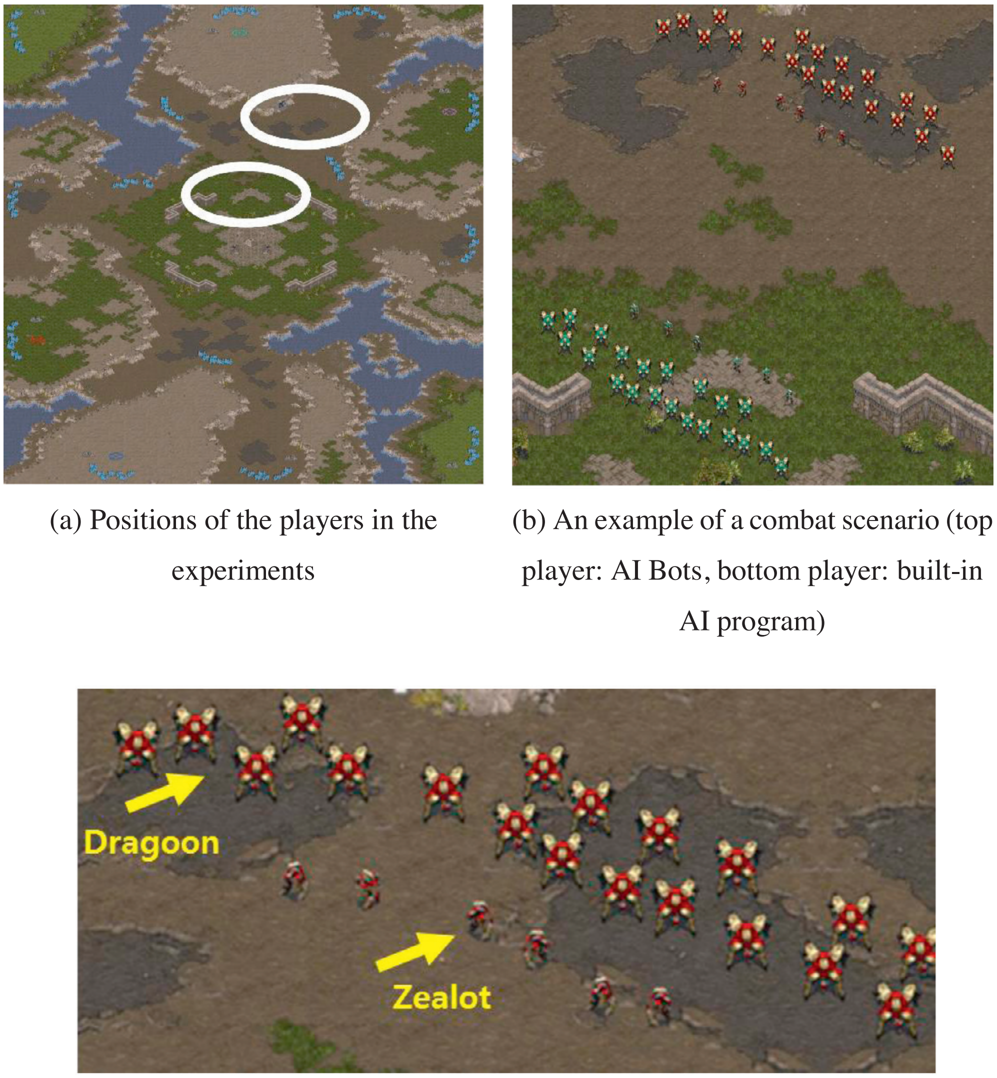

(a) Positions of the players in the experiments
(b) An example of a combat scenario (top player: AI Bots, bottom player: built-in AI program)
Dragoon
Zealot
(c) Unit configuration of zealots and dragoons

Fig. 10. "Lost Temple" map (a) and an example of a combat scenario to test the micromanagement skills (b).

Table 9
Application of case libraries trained on the "Python" map to combat with the "Lost Temple" map using large-scale combat scenarios (average of 200 games (40 games × 5 runs)).

Number of cases	Unmatched unit ratio (%)	Avg. tolerance level	Win rates (%)
5000	50.8%	7.5	91 ± 5.1
25,000	36.3%	5.3	66 ± 32.2
50,000	26.7%	4.9	94 ± 2.5
100,000	23.2%	4.2	93.5 ± 2.5
200,000	21.6%	3.6	93.5 ± 4.9
500,000 (*)	20.7%	2.9	94 ± 3.7
500,000 (**)	18.9%	3.5	93 ± 1.9

tio and average tolerance level were increased slightly. However, it still shows ~90% win rates. This result shows the potential of our approach with previously unseen maps.

6. Discussion

We found out through statistical analysis that the proposed method produced better performances compared to bots that does not use this method. However, as the bots that participated in the 2014 and 2016 competitions were not designed specifically for this experiment, they pose some penalties to the measurement of our method's efficiency. We endeavored to imitate the environments actually faced by bots in the AI competition as closely as possible through using full game replays. Even so, direct comparison between the bots retains some difficulties the competition bots were designed to handle other modules besides combat, while our model has been designed specifically for combat. Thus, rather than automatically comparing the winning rates of our bots against those of existing AIs, it would be more useful to consider the strengths and weaknesses of our method as a whole to determine its usefulness. The strengths and weaknesses of our model are as listed below:

Strengths

- The model can automatically imitate human behavior from replay data, even when players cannot explain reasoning/pattern behind their own behavior.
- The model can collect human behavior-related data from most games and systems such as actual robots, through learning replays and observing human behaviors respectively.
- Despite the constraints of RTS games, the model can locate similar situations in large-scale data in a short amount of time, and can perform well in combat.
- Through combining with other types of learning (e.g. reinforcement learning), the model can learn even complex behaviors over time. The Alphago (Silver et al., 2016), for example, was able to learn complex behaviors through employing imitation learning in the first stage, and reinforcement learning in the second stage.

Weaknesses

- In case of imitation learning, the model imitates human data, therefore there is a limit to outperforming human behavior. Reinforcement learning can be used in combination with imitation learning, to overcome this limitation.
- Current approaches can reduce reaction rates when using large memory and fast CPU, but there may be problems with slow response speed in low specification systems.
- Results may differ depending on the quality of the replay data used for imitation. For example, if there are many players in the data displaying wrong behavior, there is a possibility that the model will learn these behaviors. This situation must be adjusted (Oh & Kim, 2015).
- If a similar scene does not exist, it may be necessary to add some heuristics to make strategic decisions in such situations. This problem can be solved by securing large amounts of data.

7. Conclusions and future works

In this paper, we proposed to imitate human micromanagement skills from a massive number of game cases found in replay files. It is a promising approach because it is easy to obtain the required replays, given that they are shared by gamers. For StarCraft, it is possible to download about 300,000 replays from gaming portals. Even though they are available to the public, the quality of the replays is very high because they contain replays from professional player matches. They record the sophisticated unit controls made by professional players in different game configurations (map, race, starting position and so on). For game companies, they are able to collect massive numbers of replays by recording the games played online through the game servers. Also, they can create in-house replays by recruiting expert players to play games (similar to capturing a professional actor's motion behavior for game character design).

We demonstrated that it is possible to reduce the response time significantly by calculating the IMs offline and loading them into memory during play. This requires a large amount of main memory (18.7GB for 500,000 cases). However, the cost of memory is no longer expensive, so it is possible to run the imitation system using common desktop machines. In addition, the use of a SSD can complement the memory-oriented computing by speeding up the access to the data storage. For example, the pre-calculated IMs can be stored on the SSD and a portion of them will be loaded into memory in real-time, speeding up the case search. Also, research into reducing IM generation time by using GPU may be required because CPU capacity may not be enough to process more unit types and data in RTS games. These approaches offer potential in opening the way to exploit the use of millions of game cases acquired from human replays with a view to imitating human-like and human-level game playing.

Acknowledgments

This work was supported by a National Research Foundation of Korea (NRF) grant, funded by the Korean government (MSIP) (2013 R1A2A2A01016589). This paper has been significantly extended from the two-page demonstration paper published in the IEEE International Conference on Computational Intelligence and Games 2014 (Oh, Cho, & Kim, 2014). We have released the execute files of our system at http://cilab.sejong.ac.kr/home/doku.php?id=public:resources:imitation.

Supplementary materials

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.eswa.2016.11.026.

References

Buro, M. (2003). Real-time strategy games: A new AI research challenge. In *Proceedings of the international joint conference on AI* (pp. 1534–1535).
BWAPI. *An API for Interacting with StarCraft: Brood War (2016).* http://code.google.com/p/bwapi/.
Cho, H.-C., Kim, K.-J., & Cho, S.-B. (2013). Replay-based strategy prediction and build order adaptation for StarCraft AI bots. In *IEEE conference on computational intelligence in games* (pp. 1–7).
Chruchill, D., Saffidine, A., & Buro, M. (2012). Fast heuristic search for RTS game combat scenarios. In *Proceedings of the AAAI conference on artificial intelligence and interactive digital entertainment* (pp. 112–117).
Churchill, D., & Buro, M. (2013). Portfolio greedy search and simulation for large-scale combat in StarCraft. In *IEEE conference on computational intelligence and games* (pp. 1–8).
Davoust, A., Floyd, M. W., & Esfandiari, B. (2008). Use of fuzzy histograms to model the spatial distribution of objects in case-based reasoning. *Lecture Notes in Computer Science, 5032*, 72–83.
Floyd, M. W., Davoust, A., & Esfandiari, B. (2008). Considerations for real-time spatially-aware case-based reasoning: A case study in robotic soccer imitation. In *9th European conference on case-based reasoning* (pp. 195–209).
Floyd, M. W., Esfandiari, B., & Lam, K. (2008). A case-based reasoning approach to imitating RoboCup players. In *proceedings of FLAIRS-2008, Florida AI research symposium* (pp. 251–256).
Gabriel, I., Negru, V., & Zaharie, D. (2012). Neuroevolution based multi-agent system for micromanagement in real-time strategy games. In *Proceedings of the fifth balkan conference in informatics* (pp. 32–39).
Gemine, Q., Safadi, F., Fonteneau, R., & Ernst, D. (2012). Imitative learning for real-time strategy games. In *Proceedings of IEEE international conference on computational intelligence and games* (pp. 424–429).
Gillespie, K., Karneeb, J., Lee-Urban, S., & Muñoz-Avila, H. (2010). Imitating inscrutable enemies: Learning from stochastic policy observation, retrieval

and reuse. In *18th international conference on case-based reasoning (ICCBR)* (pp. 126–140).

Hagelback, J., & Johansson, S. J. (2008). Dealing with fog of war in a real time strategy game environment. In *IEEE symposium on computational intelligence and games* (pp. 55–62).

Hostetler, J., Dereszynski, E. W., Dietterich, T. G., & Fern, A. (2012). Inferring strategies from limited reconnaissance in real-time strategy games. *Uncertainty in Artificial Intelligence*, 367–376.

Kolodner, J. (2014). *Case-based reasoning*. Morgan Kaufmann.

Miles, C., & Louis, S. J. (2006). Towards the co-evolution of influence map tree based strategy game players. In *IEEE computational intelligence and AI in games* (pp. 75–82).

Miles, C., Quiroz, J., Leigh, R., & Louis, S. J. (2007). Co-evolving influence map tree based strategy game players. In *IEEE computational intelligence and AI in games* (pp. 88–95).

Nguyen, T., Nguyen, K., & Thawonmas, R. (2013). Potential flow for unit positioning during combat in StarCraft. In *IEEE 2nd global conference on consumer electronics* (pp. 10–11).

Oh, I.-S., Cho, H.-C., & Kim, K.-J. (2014). Imitation learning for combat system in RTS games with application to StarCraft. In *IEEE conference on computational intelligence and games* (pp. 1–2).

Oh, I.-S., & Kim, K.-J. (2015). Testing Reliability of Replay-based Imitation for StarCraft. In *IEEE conference on computational intelligence and games* (pp. 536–537).

Ontañón, S., Mishra, K., Sugandh, N., & Ram, A. (2007). Case-based planning and execution for real-time strategy games. In *proceedings of the international conference on case-based reasoning (ICCBR)* (pp. 164–178).

Ontañón, S., Bonnette, K., Mahindrakar, P., Gomez-Martın, M. A., Long, K., & Radhakrishnan, J. (2009). Learning from human demonstrations for real-time case-based planning. In *The IJCAI-09 workshop on learning structural knowledge from observations* (pp. 293–310).

Ontañón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., & Preuss, M. (2013). A survey of real-time strategy game AI research and competition in StarCraft. In *IEEE transactions on computational intelligence and AI in games: 5* (pp. 293–311).

Park, H.-S., Cho, H.-C., Lee, K.-Y., & Kim, K.-J. (2012). Prediction of early stage opponent strategy for StarCraft AI using scouting and machine learning. In *Workshop at SIGGRAPH ASIA (Computer Gaming Track)* (pp. 7–12).

Parra, R., & Garrido, L. (2013). Bayesian networks for micromanagement decision imitation in the RTS game StarCraft. *Lecture Notes in Computer Science, 7630*, 433–443.

Preuss, M., Beume, N., Danielsiek, H., Hein, T., Naujoks, B., & Piatkowski, N. (2010). Towards intelligent team composition and maneuvering in real-time strategy games. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG), 2*(2), 82–98.

Rogers, K. D., & Skabar, A. A. (2014). A micromanagement task allocation system for real-time strategy games. *IEEE Transactions on Computational Intelligence and AI in Games*, *6*(1), 67–77.

Romdhane, H., & Lamontagne, L. (2008). Reinforcement of local pattern cases for playing Tetris. In *21st international florida artificial intelligence research society conference* (pp. 263–268).

Schaeffer, J. (2009). *One Jump Ahead* Springer.

Shantia, A., Begue, E., & Wiering, M. (2011). Connectionist reinforcement learning for intelligent unit micromanagement in StarCraft. In *International joint conference on neural networks* (pp. 1794–1801).

Szczepanski, T., & Aamodt, A. (2009). Case-based reasoning for improved micromanagement in real-time strategy games. In *Proceedings of the workshop on case-based reasoning for computer games, ICCBR* (pp. 139–148).

Uriarte, A., & Ontañón, S. (2012). Kiting in RTS games using influence maps. In *proc. AI adversarial real-time games workshop at AIIDE* (pp. 31–36).

Watson, I., Rubin, J., & Robertson, G. (2012). SARTRE: A case-based poker web app. In *Proceedings of the 8th australasian conference on interactive entertainment: playing the system* (p. 23).

Weber, B. G., Mateas, M., & Jhala, A. (2011). Building human-level AI for real-time strategy games. In *Proceedings of the AAAI fall symposium on advances in cognitive systems* (pp. 329–336).

Wender, S., & Watson, I. (2014). Integrating case-based reasoning with reinforcement learning for real-time strategy game micromanagement. In *13th Pacific rim international conference on artificial intelligence* (pp. 64–76).

Young, J., Smith, F., Atkinson, C., Poyner, K., & Chothia, T. (2012). SCAIL: An integrated StarCraft AI system. In *IEEE conference on computational intelligence and games* (pp. 438–445).

Zhen, J. S., & Watson, I. (2013). Neuroevolution for micromanagement in the real–time strategy game StarCraft: Brood war. *Lecture Notes in Artificial Intelligence, 8272*, 259–270.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., & Van Den Driessche, G. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature, 529*(7587), 484–489.