A Review of Real-Time Strategy Game AI

Glen Robertson and Ian Watson
University of Auckland
New Zealand
{glen, ian}@cs.auckland.ac.nz

Abstract

This literature review covers AI techniques used for real-time strategy video games, focusing specifically on StarCraft. It finds that the main areas of current academic research are in tactical and strategic decision-making, plan recognition, and learning, and it outlines the research contributions in each of these areas. The paper then contrasts the use of game AI in academia and industry, finding the academic research heavily focused on creating game-winning agents, while the industry aims to maximise player enjoyment. It finds the industry adoption of academic research is low because it is either inapplicable or too time-consuming and risky to implement in a new game, which highlights an area for potential investigation: bridging the gap between academia and industry. Finally, the areas of spatial reasoning, multi-scale AI, and cooperation are found to require future work, and standardised evaluation methods are proposed to produce comparable results between studies.

1 Introduction

Games are an ideal domain for exploring the capabilities of Artificial Intelligence (AI) within a constrained environment and a fixed set of rules, where problem-solving techniques can be developed and evaluated before being applied to more complex real-world problems (Schaeffer 2001). AI has notably been applied to board games, such as chess, Scrabble, and backgammon, creating competition which has sped the development of many heuristic-based search techniques (Schaeffer 2001). Over the past decade, there has been increasing interest in research based on video game AI, which was initiated by Laird and van Lent (2001) in their call for the use of video games as a testbed for AI research. They saw video games as a potential area for iterative advancement in increasingly sophisticated scenarios, eventually leading to the development of human-level AI. Buro (2003) later called for increased research in Real-Time Strategy (RTS) games as they provide a sandbox for exploring various complex challenges that are central to game AI and many other problems.

Video games are an attractive alternative to robotics for AI research because they increasingly provide a complex and realistic environment for simulation, with few of the messy properties (and cost) of real-world equipment (Buro 2004; Laird and van Lent 2001). They also present a number of challenges which set them apart from the simpler board games that AI has famously been applied to in the past. Video games often have real-time constraints which prevent players from thinking extensively about each action, randomness which prevents players from completely planning future events, and hidden information which prevents players from knowing exactly what the other players are doing. Similar to many board games, competitive video games usually require adversarial reasoning to react according to other players' actions (Laird and van Lent 2001; Mehta et al. 2009; Weber, Mateas, and Jhala 2010).

1.1 RTS Games

This paper is focused on Real-Time Strategy (RTS) games, which are essentially simplified military simulations. In an RTS game, a player indirectly controls many units and structures by issuing orders from an overhead perspective (figure 1) in "real-time" in order to gather resources, build an infrastructure and an army, and destroy the opposing player's forces. The real-time aspect comes from the fact that players do not take turns, but instead may perform as many actions as they are physically able to make, while the game simulation runs at a constant frame rate (24 frames per second in StarCraft) to approximate a continuous flow of time. Some notable RTS games include Dune II, Total Annihilation, and the Warcraft, Command & Conquer, Age of Empires, and StarCraft series'.

Generally, each match in an RTS game involves two players starting with a few units and/or structures in different locations on a two-dimensional terrain (map). Nearby resources can be gathered in order to produce additional units and structures and purchase upgrades, thus gaining access to more advanced in-game technology (units, structures, and upgrades). Additional resources and strategically important points are spread around the map, forcing players to spread out their units and buildings in order to attack or defend these positions. Visibility is usually limited to a small area around player-owned units, limiting information and forcing players to conduct reconnaissance in order to respond effectively to their opponents. In most RTS games, a match ends when one player (or team) destroys all buildings belonging to the opponent player (or team), although often a player will forfeit earlier when they see they cannot win.

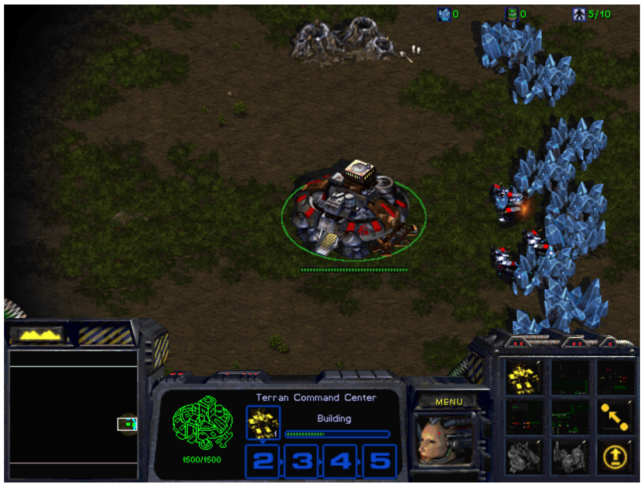

Figure 1: A typical match start in an RTS game. Worker units have been sent to gather resources (right) and return them to the central building. Resources (recorded top right) are being spent building an additional worker (bottom centre). Dark fog (left) blocks visibility away from player units.

RTS games have a variety of military units, used by the players to wage war, as well as units and structures to aid in resource collection, unit production, and upgrades. During a match, players must balance the development of their economy, infrastructure, and upgrades with the production of military units, so they have enough units to successfully attack and defend in the present and enough resources and upgrades to succeed later. They must also decide which units and structures to produce and which technologies to advance throughout the game in order to have access to the right composition of units at the right times. This long-term high-level planning and decision-making, often called "macromanagement", is referred to in this paper as **strategic decision-making**. In addition to strategic decision-making, players must carefully control their units in order to maximise their effectiveness on the battlefield. Groups of units can be manoeuvred into advantageous positions on the map to surround or escape the enemy, and individual units can be controlled to attack a weak enemy unit or avoid an incoming attack. This short term control and decision-making with individual units, often called "micromanagement", and medium-term planning with groups of units, often called "tactics", is referred to collectively in this paper as **tactical decision-making**.

In addition to the general video game challenges mentioned above, RTS games involve long-term goals and usually require multiple levels of abstraction and reasoning. They have a vast space of actions and game states, with durative actions, a huge branching factor, and actions which can have long-term effects throughout the course of a match (Buro and Churchill 2012; Buro and Furtak 2004; Mehta et al. 2009; Ontañón 2012; Tozour 2002; Weber, Mateas, and Jhala 2010). Even compared with Go, which is currently an active area of AI research, RTS games present a huge increase in complexity – at least an order of magnitude increase in the number of possible game states, actions to choose from, actions per game, and actions per minute (using standard rules) (Buro 2004; Schaeffer 2001; Synnaeve and Bessière 2011b). The state space is so large that traditional heuristic-based search techniques, which have proven effective in a range of board games (Schaeffer 2001), have so far been unable to solve all but the most restricted subproblems of RTS AI. Due to their complexity and challenges, RTS games are probably the best current environment in which to pursue Laird & van Lent's vision of game AI as a stepping stone toward human-level AI. It is a particularly interesting area for AI research because even the best agents are outmatched by experienced humans (Huang 2011; Synnaeve and Bessière 2011a; Weber, Mateas, and Jhala 2010), due to the human abilities to abstract, reason, learn, plan, and recognise plans (Buro 2004; Buro and Churchill 2012).

1.2 StarCraft

This paper primarily examines AI research within a subtopic of RTS games: the RTS game StarCraft[1] (figure 2). StarCraft is a canonical RTS game, like chess is to board games, with a huge player base and numerous professional competitions. The game has three different but very well balanced teams, or "races", allowing for varied strategies and tactics without any dominant strategy, and requires both strategic and tactical decision-making roughly equally (Synnaeve and Bessière 2011b). These features give StarCraft an advantage over other RTS titles which are used for AI research, such as Wargus[2] and ORTS[3].

StarCraft was chosen due to its increasing popularity for use in RTS game AI research, driven by the Brood War Application Programming Interface (BWAPI)[4] and the AIIDE[5] and CIG[6] StarCraft AI Competitions. BWAPI provides an interface to programmatically interact with StarCraft, allowing external code to query the game state and execute actions as if they were a player in a match. The competitions pit StarCraft AI agents (or "bots") against each other in full games of StarCraft to determine the best bots and improvements each year (Buro and Churchill 2012). Initially these competitions also involved simplified challenges based on subtasks in the game, such as controlling a given army to defeat an opponent with an equal army, but more recent competitions have used only complete matches. For more detail on StarCraft competitions and bots, see (Ontañón et al. In press).

In order to develop AI for StarCraft, researchers have tried many different techniques, as outlined in table 1. A com-

[1]Blizzard Entertainment: StarCraft: `blizzard.com/games/sc/`

[2]Wargus: `wargus.sourceforge.net`

[3]Open RTS: `skatgame.net/mburo/orts`

[4]Brood War API: `code.google.com/p/bwapi`

[5]AIIDE StarCraft AI Competition: `www.starcraftaicompetition.com`

[6]CIG StarCraft AI Competition: `ls11-www.cs.uni-dortmund.de/rts-competition`

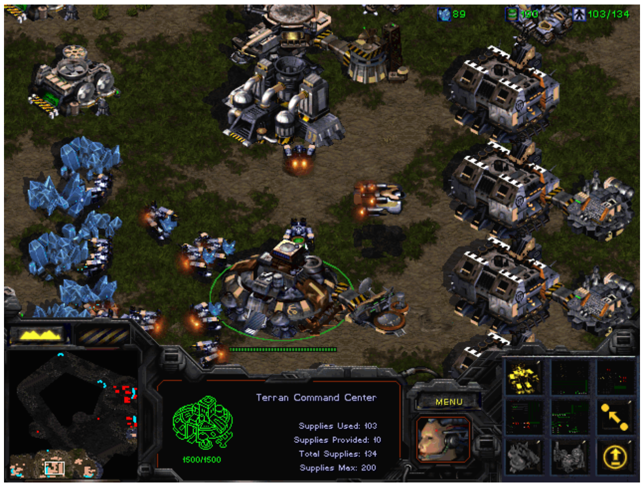

Figure 2: Part of a player's base in StarCraft. The white rectangle on the minimap (bottom left) is the area visible on screen. The minimap shows area that is unexplored (black), explored but not visible (dark), and visible (light). It also shows the player's forces (lighter dots) and last-seen enemy buildings (darker dots).

munity has formed around the game as a research platform, enabling people to build on each other's work and avoid repeating the necessary groundwork before an AI system can be implemented. This work includes a terrain analysis module (Perkins 2010), well-documented source code for a complete, modular bot (Churchill and Buro 2012), and preprocessed data sets assembled from thousands of professional games (Synnaeve and Bessière 2012). StarCraft has a lasting popularity among professional and amateur players, including a large professional gaming scene in South Korea, with international competitions awarding millions of dollars in prizes every year (Churchill and Buro 2011). This popularity means that there are a large number of high-quality game logs (replays) available on the internet which can be used for data mining, and there are many players of all skill levels to test against (Buro and Churchill 2012; Synnaeve and Bessière 2011b; Weber, Mateas, and Jhala 2011a).

Tactical Decision-Making	Strategic Decision-Making & Plan Recognition
Reinforcement Learning	Case-Based Planning
Game-Tree Search	Hierarchical Planning
Bayesian models	Behavior Trees
Case-Based Reasoning	Goal-Driven Autonomy
Neural Networks	State Space Planning
	Evolutionary Algorithms
	Cognitive Architectures
	Deductive Reasoning
	Probabilistic Reasoning
	Case-Based Reasoning

Table 1: AI Techniques Used for StarCraft

This paper presents a review of the literature on RTS AI with an emphasis on StarCraft. It includes particular research based on other RTS games in the case that significant literature based on StarCraft is not (yet) available in that area. The paper begins by outlining the different AI techniques used, grouped by the area in which they are primarily applied. These areas are tactical decision-making, strategic decision-making, plan recognition, and learning. This is followed by a comparison of the way game AI is used in academia and the game industry, which outlines the differences in goals and discusses the low adoption of academic research in the industry. Finally, some areas are identified in which there does not seem to be sufficient research on topics that are well-suited to study in the context of RTS game AI. This last section also calls for standardisation of the evaluation methods used in StarCraft AI research in order to make comparison possible between papers.

2 Tactical Decision-Making

Tactical and micromanagement decisions – controlling individual units or groups of units over a short period of time – often make use of a different technique from the AI making strategic decisions. These tactical decisions can follow a relatively simple metric, such as attempting to maximise the amount of enemy firepower which can be removed from the playing field in the shortest time (Davis 1999). In the video game industry, it is common for simple techniques, such as finite state machines, to be used to make these decisions (Buckland 2005). However, even in these small-scale decisions, many factors can be considered to attempt to make the best decisions possible, particularly when using units with varied abilities (figure 3), but the problem space is not nearly as large as that of the full game, making feasible exploratory approaches to learning domain knowledge (Weber and Mateas 2009). There appears to be less research interest in this aspect of RTS game AI than in the area of large-scale, long-term strategic decision making and learning.

Figure 3: A battle in StarCraft – intense micromanagement is required to maximise the effectiveness of individual units, especially "spellcaster" units like the Protoss Arbiter

2.1 Reinforcement Learning

Reinforcement Learning (RL) is an area of machine learning in which an agent must learn, by trial and error, optimal actions to take in particular situations order to maximise an overall reward value (Sutton and Barto 1998). Through many iterations of weakly supervised learning, RL can discover new solutions which are better than previously known solutions. It is relatively simple to apply to a new domain, as it requires only a description of the situation and possible actions, and a reward metric (Manslow 2004). However, in a domain as complex as an RTS game – even just for tactical decision-making – RL often requires clever state abstraction mechanisms in order to learn effectively. This technique is not commonly used for large-scale strategic decision-making, but is often applied to tactical decision-making in RTS games, likely due to the huge problem space and delayed reward inherent in strategic decisions, which make RL difficult.

RL has been applied to StarCraft by Shantia, Begue, and Wiering (2011), where Sarsa, an algorithm for solving RL problems, is used to learn to control units in small skirmishes. They made use of artificial neural networks to learn the expected reward for attacking or fleeing with a particular unit in a given state (figure 4), and chose the action with the highest expected reward when in-game. The system learned to beat the inbuilt StarCraft AI scripting on average in only small three-unit skirmishes, with none of the variations learning to beat the inbuilt scripting on average in six-unit skirmishes (Shantia, Begue, and Wiering 2011).

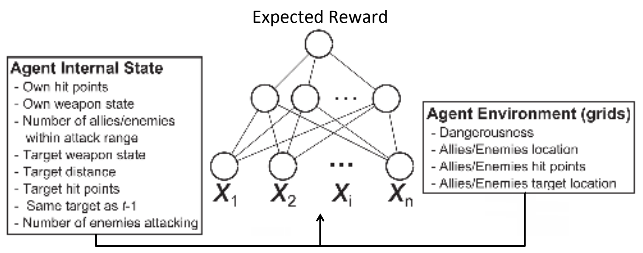
Expected Reward
Agent Internal State
- Own hit points
- Own weapon state
- Number of allies/enemies within attack range
- Target weapon state
- Target distance
- Target hit points
- Same target as t-1
- Number of enemies attacking
X_1 X_2 X_i X_n
Agent Environment (grids)
- Dangerousness
- Allies/Enemies location
- Allies/Enemies hit points
- Allies/Enemies target location

Figure 4: Game state information fed into a neural network to produce an expected reward value for a particular action. Adapted from Shantia, Begue, and Wiering (2011)

RL techniques have also been applied to other RTS games. Sharma et al. (2007) and Molineaux, Aha, and Moore (2008) combine Case-Based Reasoning (CBR) and RL for learning tactical-level unit control in MadRTS[7] (for a description of CBR see section 4.4). Sharma et al. (2007) was able to increase the learning speed of the RL agent by beginning learning in a simple situation and then gradually increasing the complexity of the situation. The resulting performance of the agent was the same or better than an agent trained in the complex situation directly. Their system stores its knowledge in cases which pertain to situations it has encountered before, as in CBR. However, each case stores the expected utility for every possible action in that situation as well as the contribution of that case to a reward value, allowing the system to learn desirable actions and situations. It remains to be seen how well it would work in a more complex domain. Molineaux, Aha, and Moore (2008) describe a system for RL with non-discrete actions. Their system retrieves similar cases from past experience and estimates the result of applying each case's actions to the current state. It then uses a separate case base to estimate the value of each estimated resulting state, and extrapolates around, or interpolates between, the actions to choose one which is estimated to provide the maximum value state. This technique results in a significant increase in performance when compared with one using discrete actions (Molineaux, Aha, and Moore 2008).

Human critique is added to RL by Judah et al. (2010) in order to learn tactical decision-making for controlling a small group of units in combat in Wargus. By interleaving sessions of autonomous state space exploration and human critique of the agent's actions, the system was able to learn a better policy in a fraction of the training iterations compared with using RL alone. However, slightly better overall results were achieved using human critique only to train the agent, possibly due to humans giving better feedback when they can see an immediate result (Judah et al. 2010).

Marthi et al. (2005) argues that it is preferable to decrease the apparent complexity of RTS games and potentially increase the effectiveness of RL or other techniques by decomposing the game into a hierarchy of interacting parts. Using this method, instead of coordinating a group of units by learning the correct combination of unit actions, each unit can be controlled individually with a higher-level group control affecting each individual's decision. Similar hierarchical decomposition appears in many RTS AI approaches because it reduces complexity from a combinatorial combination of possibilities – in this case, possible actions for each unit – down to a multiplicative combination.

2.2 Game-Tree Search

Search-based techniques have so far been unable to deal with the complexity of the long-term strategic aspects of RTS games, but they have been successfully applied to smaller-scale or abstracted versions of RTS combat. To apply these search methods, a simulator is usually required to allow the AI system to evaluate the results of actions very rapidly in order to explore the game tree.

Sailer, Buro, and Lanctot (2007) take a game theoretic approach by searching for the Nash equilibrium strategy among a set of known strategies in a simplified RTS. Their simplified RTS retains just the tactics aspect of RTS games by concentrating on unit group movements, so it does not require long-term planning for building infrastructure and also excludes micromanagement for controlling individual units. They use a simulation to compare the expected outcome from using each of the strategies against their opponent, for each of the strategies their opponent could be using (which is drawn from the same set), and select the Nash-optimal strategy. The simulation can avoid simulating every time-step, skipping instead to just the states in which something "interesting" happens, such as a player making a decision, or units

[7]Mad Doc Software. Website no longer available.

coming into firing range of opponents. Through this combination of abstraction, state skipping, and needing to examine only the possible moves prescribed by a pair of known strategies at a time, it is usually possible to search all the way to an end-game state very rapidly, which in turn means a simple evaluation function can be used. The resulting Nash player was able to defeat each of the scripted strategies, as long as the set included a viable counter-strategy for each strategy, and it also produced better results than the max-min and min-max players (Sailer, Buro, and Lanctot 2007).

Search-based techniques are particularly difficult to use in StarCraft because of the closed-source nature of the game and inability to arbitrarily manipulate the game state. This means that the precise mechanics of the game rules are unclear, and the game cannot be easily set up to run from a particular state to be used as a simulator. Furthermore, the game must carry out expensive calculations such as unit vision and collisions, and cannot be forced to skip ahead to just the "interesting" states, making it too slow for the purpose of search (Churchill, Saffidine, and Buro 2012). In order to overcome these problems, Churchill, Saffidine, and Buro (2012) created a simulator called "SparCraft"[8] which models StarCraft and approximates the rules, but allows the state to be arbitrarily manipulated and unnecessary expensive calculations to be ignored (including skipping uninteresting states). Using this simulator and a modified version of alpha-beta search, which takes into consideration actions of differing duration, they could find effective moves for a given configuration of units. Search time was limited to approximate real-time conditions, so the moves found were not optimal. This search allowed them to win an average of 92% of randomised balanced scenarios against all of the standard scripted strategies they tested against within their simulator (Churchill, Saffidine, and Buro 2012).

Despite working very well in simulation, the results do not translate perfectly back to the actual game of StarCraft, due to simplifications such as the lack of unit collisions and acceleration, affecting the outcome (Churchill and Buro 2012; Churchill, Saffidine, and Buro 2012). The system was able to win only 84% of scenarios against the built in StarCraft AI despite the simulation predicting 100%, faring the worst in scenarios which were set up to require hit-and-run behavior (Churchill and Buro 2012). The main limitation of this system is that due to the combinatorial explosion of possible actions and states as the number of units increases, the number of possible actions in StarCraft, and a time constraint of 5ms per game frame, the search will only allow up to eight units per side in a two player battle before it is too slow. On the other hand, better results may be achieved through opponent modelling, because the search can incorporate known opponent actions instead of searching through all possible opponent actions. When this was tested on the scripted strategies with a perfect model of each opponent (the scripts themselves), the search was able to achieve at least a 95% win rate against each of the scripts in simulation (Churchill, Saffidine, and Buro 2012).

[8]SparCraft: `code.google.com/p/sparcraft/`

Monte Carlo Planning Monte Carlo planning has received significant attention recently in the field of computer Go, but seems to be almost absent from RTS AI, and (to the authors' knowledge) completely untested in the domain of StarCraft. It involves sampling the decision space using randomly-generated plans in order to find out which plans tend to lead to more successful outcomes. It may be very suitable for RTS games because it can deal with uncertainty, randomness, large decision spaces, and opponent actions through its sampling mechanism. Monte Carlo planning has likely not yet been applied to StarCraft due to the unavailability of an effective simulator, as was the case with the search methods above, as well as the complexity of the domain. However, it has been applied to some very restricted versions of RTS games. Although both of the examples seen here are considering tactical- and unit-level decisions, given a suitable abstraction and simulation, MCTS may also be effective at strategic level decision-making in a domain as complex as StarCraft.

Chung, Buro, and Schaeffer (2005) created a capture-the-flag game in which each player needed to control a group of units to navigate through obstacles to the opposite side of a map and retrieve the opponent's flag. They created a generalised Monte Carlo planning framework and then applied it to their game, producing positive results. Unfortunately, they lacked a strong scripted opponent to test against, and their system was also very reliant on heuristic evaluations of intermediate states in order to make planning decisions. Later, Balla and Fern (2009) applied the more recent technique of Upper Confidence Bounds applied to Trees (UCT) to a simplified Wargus scenario. A major benefit of their approach is that it does not require a heuristic evaluation function for intermediate states, and instead plays a game randomly out to a terminal state in order to evaluate a plan. The system was evaluated by playing against a range of scripts and a human player in a scenario involving multiple friendly and enemy groups of the basic footman unit placed around an empty map. In these experiments, the UCT system made decisions at the tactical level for moving groups of units while micromanagement was controlled by the inbuilt Wargus AI, and the UCT evaluated terminal states based on either unit hit points remaining or time taken. The system was able to win all of the scenarios, unlike any of the scripts, and to overall outperform all of the other scripts and the human player on the particular metric (either hit points or time) that it was using.

2.3 Other Techniques

Various other AI techniques have been applied to tactical decision-making in StarCraft. Synnaeve and Bessière (2011b) combines unit objectives, opportunities, and threats using a Bayesian model to decide which direction to move units in a battle. The model treats each of its sensory inputs as part of a probability equation which can be solved, given data (potentially learned through RL) about the distributions of the inputs with respect to the direction moved, to find the probability that a unit should move in each possible direction. The best direction can be selected, or the direction probabilities can be sampled over to avoid having two units choose to move into the same location. Their Bayesian model is paired with a hierarchical finite state machine to choose different sets of behavior for when units are engaging or avoiding enemy forces, or scouting. The bot produced was very effective against the built-in StarCraft AI as well as its own ablated versions (Synnaeve and Bessière 2011b).

CBR, although usually used for strategic reasoning in RTS AI (see section 4.4), has also been applied to tactical decision-making in Warcraft III[9], a game which has a greater focus on micromanagement than StarCraft (Szczepański and Aamodt 2009). CBR generally selects the most similar case for reuse, but Szczepański and Aamodt (2009) added a conditional check to each case so that it could be selected only when its action was able to be executed. They also added reactionary cases which would be executed as soon as certain conditions were met. The resulting agent was able to beat the built in AI of Warcraft III in a micromanagement battle using only a small number of cases, and was able to assist human players by micromanaging battles to let the human focus on higher-level strategy.

Neuroevolution is a technique that uses an evolutionary algorithm to create or train an artificial neural network. Gabriel, Negru, and Zaharie (2012) use a neuroevolution approach called rtNEAT to evolve both the topology and connection weights of neural networks for individual unit control in StarCraft. In their approach, each unit has its own neural network that receives input from environmental sources (such as nearby units or obstacles) and hand-defined abstractions (such as the number, type, and "quality" of nearby units), and outputs whether to attack, retreat, or move left or right. During a game, the performance of the units is evaluated using a hand-crafted fitness function and poorly-performing unit agents are replaced by combinations of the best-performing agents. It is tested in very simple scenarios of 12 versus 12 units in a square arena, where all units on each side are either a hand-to-hand or ranged type unit. In these situations, it learns to beat the built in StarCraft AI and some other bots. However, it remains unclear how well it would cope with more units or mixes of different unit types (Gabriel, Negru, and Zaharie 2012).

3 Strategic Decision-Making

In order to create a system which can make intelligent actions at a strategic level in an RTS game, many researchers have created planning systems. These systems are capable of determining sequences of actions to be taken in a particular situation in order to achieve specified goals. It is a challenging problem because of the incomplete information available – "fog of war" obscures areas of the battlefield that are out of sight of friendly units – as well as the huge state and action spaces and many simultaneous non-hierarchical goals. With planning systems, researchers hope to enable AI to play at a human-like level, while simultaneously reducing the development effort required when compared with the scripting commonly used in industry. The main techniques used for planning systems are Case-Based Planning (CBP), Goal-Driven Autonomy (GDA) and Hierarchical Planning.

A basic strategic decision-making system was produced in-house for the commercial RTS game Kohan II: Kings of War[10] (Dill 2006). It assigned resources – construction, research, and upkeep capacities – to goals, attempting to maximise the total priority of the goals which could be satisfied. The priorities were set by a large number of hand-tuned values, which could be swapped for a different set to give the AI different personalities (Dill 2006). Each priority value was modified based on relevant factors of the current situation, a goal commitment value (to prevent flip-flopping once a goal has been selected) and a random value (to reduce predictability). It was found that this not only created a fun, challenging opponent, but also made the AI easier to update for changes in game design throughout the development process (Dill 2006).

3.1 Case-Based Planning

CBP is a planning technique which finds similar past situations from which to draw potential solutions to the current situation. In the case of a CBP system, the solutions found are a set of potential plans or sub-plans which are likely to be effective in the current situation. CBP systems can exhibit poor reactivity at the strategic level and excessive reactivity at the action level, not reacting to high-level changes in situation until a low-level action fails, or discarding an entire plan because a single action failed (Palma et al. 2011).

One of the first applications of CBP to RTS games was by Aha, Molineaux, and Ponsen (2005), who created a system which extended the "dynamic scripting" concept of Ponsen et al. (2005) to select tactics and strategy based on the current situation. Using this technique, their system was able to play against a non-static opponent instead of requiring additional training each time the opponent changed. They reduced the complexity of the state and action spaces by abstracting states into a state lattice of possible orders in which buildings are constructed in a game (build orders) combined with a small set of features, and abstracting actions into a set of tactics generated for each state. This allowed their system to improve its estimate of the performance of each tactic in each situation over multiple games, and eventually learn to consistently beat all of the tested opponent scripts (Aha, Molineaux, and Ponsen 2005).

Ontañón et al. (2007) use the ideas of behaviors, goals, and alive-conditions from A Behavior Language (ABL, introduced by Mateas and Stern (2002)) combined with the ideas from earlier CBP systems to form a case-based system for playing Wargus. The cases are learned from human-annotated game logs, with each case detailing the goals a human was attempting to achieve with particular sequences of actions in a particular state. These cases can then be adapted and applied in-game to attempt to change the game state. By reasoning about a tree of goals and sub-goals to be completed, cases can be selected and linked together into plan to satisfy the overall goal of winning the game (figure 5).

[9]Blizzard Entertainment: Warcraft III:
`blizzard.com/games/war3/`

[10]TimeGate Studios: Kohan II Kings of War:
`www.timegate.com/games/kohan-2-kings-of-war`

During the execution of a plan, it may be modified in order to adapt for unforeseen events or compensate for a failure to achieve a goal.

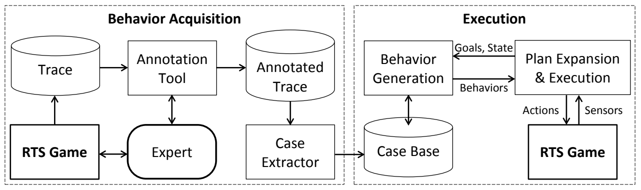

Figure 5: A case-based planning approach: using cases of actions extracted from annotated game logs to form plans which satisfy goals in Wargus. Adapted from Ontañón et al. (2007)

Mishra, Ontañón, and Ram (2008) extend the work of Ontañón et al. (2007) by adding a decision tree model to provide faster and more effective case retrieval. The decision tree is used to predict a high-level "situation", which determines the attributes and attribute weights to use for case selection. This helps by skipping unnecessary attribute calculations and comparisons, and emphasising important attributes. The decision tree and weightings are learned from game logs which have been human-annotated to show the high-level situation at each point throughout the games. This annotation increased the development effort required for the AI system but successfully provided better and faster case retrieval than the original system (Mishra, Ontañón, and Ram 2008).

More recent work using CBP tends to focus on the learning aspects of the system instead of the planning aspects. As such, it is discussed further in section 4.

A different approach is taken by Cadena and Garrido (2011), who combine the ideas of CBR with those of fuzzy sets, allowing the reasoner to abstract state information by grouping continuous feature values. This allows them to vastly simplify the state space, and it may be a closer representation of human thinking, but could potentially result in the loss of important information. For strategic decision-making, their system uses regular cases made up of exact unit and building counts, and selects a plan made up of five high-level actions, such as creating units or buildings. But for tactical reasoning (micromanagement is not explored), their system maintains independent fuzzy state descriptions and carries out independent CBR for each region of the map, thus avoiding reasoning about the map as a whole at the tactical level. Each region's state includes a linguistic fuzzy representation of its area (for example, small, medium, big), choke points, military presence, combat intensity, lost units, and amounts of each friendly and enemy unit type (for example, none, few, many). After building the case base from just one replay of a human playing against the inbuilt AI, the system was able to win around 60% of games (and tie in about 15%) against the AI on the same map. However, it is unclear how well the system would fare at the task of playing against different races (unique playable teams) and strategies, or playing on different maps.

3.2 Hierarchical Planning

By breaking up a problem hierarchically, planning systems are able to deal with parts of the situation separately at different levels of abstraction, reducing the complexity of the problem, but creating a potential new issue in coordination between the different levels (Marthi et al. 2005; Weber et al. 2010). A hierarchical plan maps well to the hierarchy of goals and sub-goals typical in RTS games, from the highest level goals such as winning the game, to the lowest level goals which map directly to in-game actions. Some researchers formalise this hierarchy into the well-defined structure of a Hierarchical Task Network (HTN), which contains tasks, their ordering, and methods for achieving them. High-level, complex tasks in an HTN may be decomposed into a sequence of simpler tasks, which themselves can be decomposed until each task represents a concrete action (Muñoz-Avila and Aha 2004).

HTNs have been used for strategic decision-making in RTS games, but not for StarCraft. Muñoz-Avila and Aha (2004) focus on the explanations that an HTN planner is able to provide to a human querying its behavior, or the reasons underlying certain events, in the context of an RTS game. Laagland (2008) implements and tests an agent capable of playing an open source RTS called Spring[11] using a hand-crafted HTN. The HTN allows the agent to react dynamically to problems, such as rebuilding a building that is lost or gathering additional resources of a particular type when needed, unlike the built in scripted AI. Using a balanced strategy, the HTN agent usually beats the built in AI in Spring, largely due to better resource management. Efforts to learn HTNs, such as Nejati, Langley, and Konik (2006), have been pursued in much simpler domains, but but never directly used in the field of RTS AI. This area may hold promise in the future for reducing the work required to build HTNs.

An alternative means of hierarchical planning was used by Weber et al. (2010). They use an active behavior tree in A Behavior Language, which has parallel, sequential and conditional behaviors and goals in a tree structure (figure 6) very similar to a behavior tree (see section 3.3). However, in this model, the tree is expanded during execution by selecting behaviors (randomly, or based on conditions or priority) to satisfy goals, and different behaviors can communicate indirectly by reading or writing information on a "shared whiteboard". Hierarchical planning is often combined as part of other methods, such as how Ontañón et al. (2007) use a hierarchical CBP system to reason about goals and plans at different levels.

3.3 Behavior Trees

Behavior trees are hierarchies of decision and action nodes which are commonly used by programmers and designers in the games industry in order to define "behaviors" (effectively a partial plan) for agents (Palma et al. 2011). They have become popular because, unlike scripts, they can be created and edited using visual tools, making them much more accessible and understandable to non-programmers

[11]Spring RTS: `springrts.com`

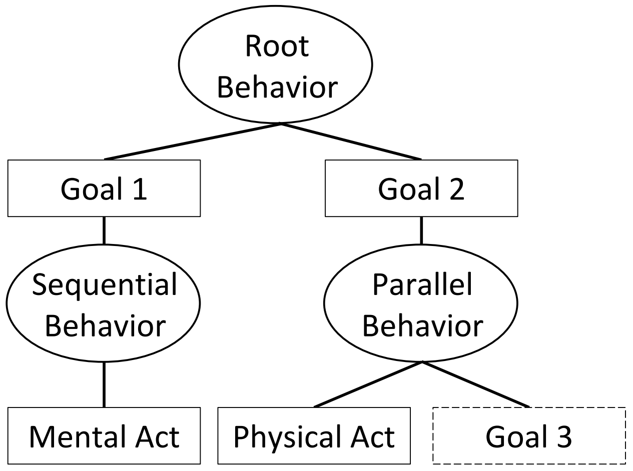

Figure 6: A simple active behavior tree used for hierarchical planning, showing mental acts (calculation or processing), physical acts (in-game actions), and an unexpanded goal. Adapted from Weber et al. (2010)

(Palma et al. 2011). Additionally, their hierarchical structure encourages reuse as a tree defining a specific behavior can be attached to another tree in multiple positions, or can be customised incrementally by adding nodes (Palma et al. 2011). Because behavior trees are hierarchical, they can cover a wide range of behavior, from very low-level actions to strategic-level decisions. Palma et al. (2011) uses behavior trees to enable direct control of a case-based planner's behavior. With their system, machine learning can be used to create complex and robust behavior through the planner, while allowing game designers to change specific parts of the behavior by substituting a behavior tree instead of an action or a whole plan. This means they can define custom behavior for specific scenarios, fix incorrectly learned behavior, or tweak the learned behavior as needed.

3.4 Goal-Driven Autonomy

GDA is a model in which "an agent reasons about its goals, identifies when they need to be updated, and changes or adds to them as needed for subsequent planning and execution" (Molineaux, Klenk, and Aha 2010). This addresses the high- and low-level reactivity problem experienced by CBP by actively reasoning about and reacting to why a goal is succeeding or failing.

Weber, Mateas, and Jhala (2010) describe a GDA system for StarCraft using A Behavior Language, which is able to form plans with expectations about the outcome. If an unexpected situation or event occurs, the system can record it as a discrepancy, generate an explanation for why it occurred, and form new goals to revise the plan, allowing the system to react appropriately to unforeseen events (figure 7). It is also capable of simultaneously reasoning about multiple goals at differing granularity. It was initially unable to learn goals, expectations, or strategies, so this knowledge had to be input and updated manually, but later improvements allowed these to be learned from demonstration (discussed further in section 4.6) (Weber, Mateas, and Jhala 2012). This system was used in the Artificial Intelligence and Interactive Digital Entertainment (AIIDE) StarCraft AI competition entry EISBot and was also evaluated by playing against human players on a competitive StarCraft ladder called International Cyber Cup (ICCup)[12], where players are ranked based on their performance – it attained a ranking indicating it was better than 48% of the competitive players (Weber, Mateas, and Jhala 2010; Weber et al. 2010).

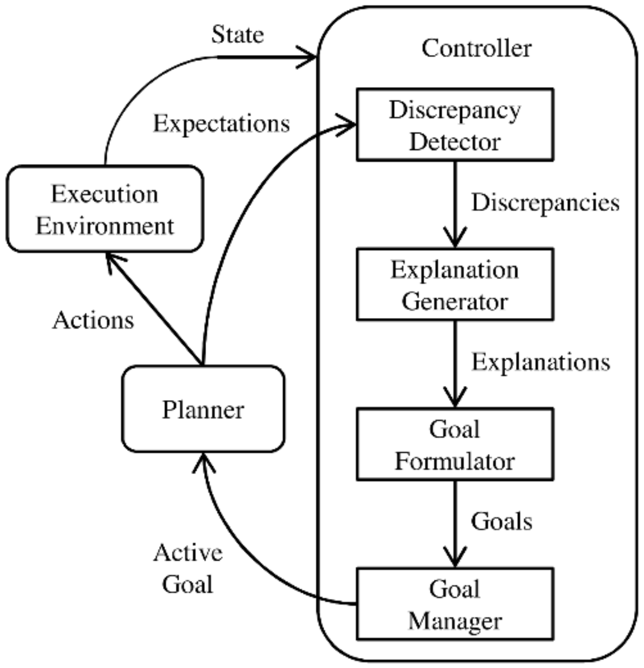

Figure 7: GDA conceptual model: a planner produces actions and expectations from goals, and unexpected outcomes result in additional goals being produced (Weber, Mateas, and Jhala 2012)

Jaidee, Muñoz-Avila, and Aha (2011) integrate CBR and RL to make a learning version of GDA, allowing their system to improve its goals and domain knowledge over time. This means that less work is required from human experts to specify possible goals, states, and other domain knowledge because missing knowledge can be learned automatically. Similarly, if the underlying domain changes, the learning system is able to adapt to the changes automatically. However, when applied to a simple domain, the system was unable to beat the performance of a non-learning GDA agent (Jaidee, Muñoz-Avila, and Aha 2011).

3.5 State Space Planning

Automated planning and scheduling is a branch of classic AI research from which heuristic state space planning techniques have been adapted for planning in RTS game AI. In these problems, an agent is given a start and goal state, and a set of actions which have preconditions and effects. The

[12]International Cyber Cup: `www.iccup.com`

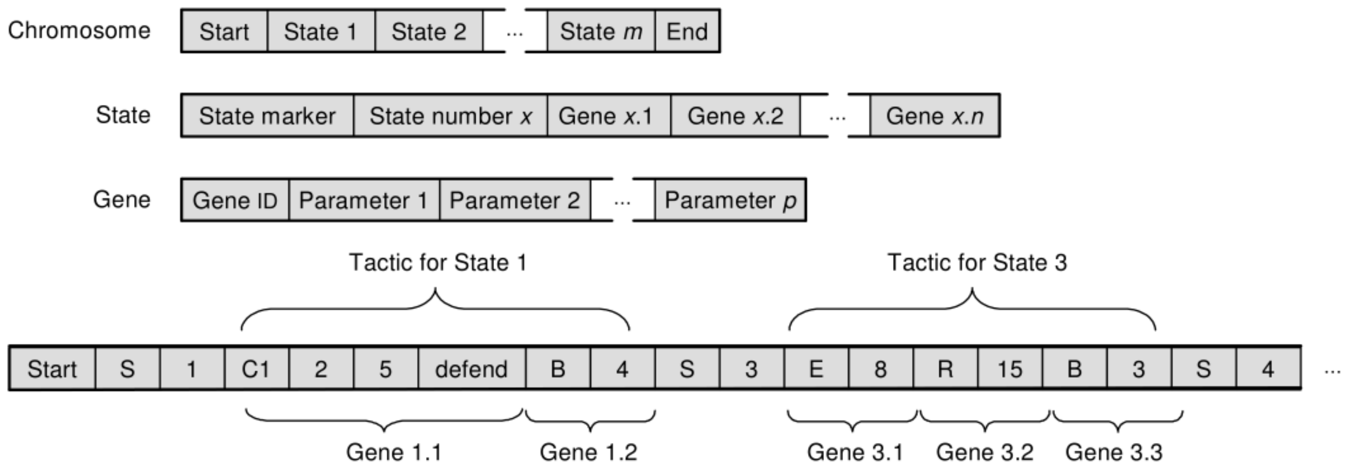

Figure 8: Design of a chromosome for evolving RTS game AI strategies (Ponsen et al. 2005)

agent must then find a sequence of actions to achieve the goal from the starting state. Existing RTS applications add complexity to the basic problem by dealing with durative and parallel actions, integer-valued state variables, and tight time constraints.

Automated planning ideas have already been applied successfully to commercial First-Person Shooter (FPS) games within an architecture called Goal-Oriented Action Planning (GOAP). GOAP allows agents to automatically select the most appropriate actions for their current situation in order to satisfy a set of goals, ideally resulting in more varied, complex, and interesting behavior, while keeping code more reusable and maintainable (Orkin 2004). However, GOAP requires a large amount of domain engineering to implement, and is limited because it maps states to goals instead of to actions, so the planner cannot tell if achieving goals is going to plan, failing, or has failed (Orkin 2004; Weber, Mateas, and Jhala 2010). Furthermore, Champandard (2011) states that GOAP has now turned out to be a dead-end, as academia and industry have moved away from GOAP in favour of hierarchical planners to achieve better performance and code maintainability.

However, Chan et al. (2007) and Churchill and Buro (2011) use an automated planning-based approach similar to GOAP to plan build orders in RTS games. Unlike GOAP, they are able to focus on a single goal: finding a plan to build a desired set of units and buildings in a minimum duration (makespan). The RTS domain is simplified by abstracting resource collection to an income rate per worker, assuming building placement and unit movement takes a constant amount of time, and completely ignoring opponents. Ignoring opponents is fairly reasonable for the beginning of a game, as there is generally little opponent interaction, and doing so means the planner does not have to deal with uncertainty and external influences on the state. Both of these methods still require expert knowledge to provide a goal state for them to pursue.

The earlier work by Chan et al. (2007) uses a combination of means-ends analysis and heuristic scheduling in Wargus. Means-ends analysis produces a plan with a minimal number of actions required to achieve the goal, but this plan usually has a poor makespan because it doesn't consider concurrent actions or actions which produce greater resources. A heuristic scheduler then reorganises actions in the plan to start each action as soon as possible, adding concurrency and reducing the makespan. To consider producing additional resources, the same process is repeated with an extra goal for producing more of a resource (for each resource) at the beginning of the plan, and the plan with the shortest makespan is used. The resulting plans, though non-optimal, were found to be similar in length to plans executed by an expert player, and vastly better than plans generated by state-of-the-art general purpose planners (Chan et al. 2007).

Churchill and Buro (2011) improve upon the earlier work by using a branch-and-bound depth-first search to find optimal build orders within an abstracted simulation of StarCraft. In addition to the simplifications mentioned above, they avoid simulating individual time steps by allowing any action which will eventually complete without further player interaction, and jumping directly to the point at which each action completes for the next decision node. Even so, other smaller optimisations were needed to speed up the planning process enough to use in-game. The search used either the gathering time or the build time required to reach the goal (whichever was longer) as the lower bound, and a random path to the goal as the upper bound (Churchill and Buro 2011). The system was evaluated against professional build orders seen in replays, using the set of units and buildings owned by the player at a particular time as the goal state. Due to the computational cost of planning later in the game, planning was restricted to 120 seconds ahead, with replanning every 30 seconds. This produced shorter or equal-length plans to the human players at the start of a game, and similar-length plans on average (with a larger variance) later in the game. It remains to be seen how well this method would perform for later stages of the game, as only the first 500 seconds were evaluated and searching took significantly longer in the latter half. However, this appears to be an effective way to produce near-optimal build orders for at least the early to middle game of StarCraft (Churchill and Buro 2011).

3.6 Evolutionary Algorithms

Evolutionary algorithms search for an effective solution to a problem by evaluating different potential solutions and combining or randomising components of high-fitness potential solutions to find new, better solutions. This approach is used infrequently in the RTS Game AI field, but it has been effectively applied to the sub-problem of tactical decision-making in StarCraft (see section 2.3) and learning strategic knowledge in similar RTS titles.

Although evolutionary algorithms have not yet been applied to strategic decision-making in StarCraft, they have been applied to its sequel, StarCraft II[13]. The Evolution Chamber[14] software uses the technique to optimise partially-defined build orders. Given a target set of units, buildings, and upgrades to be produced by certain times in the match, the software searches for the fastest or least resource-intensive way of reaching these targets. Although there have not been any academic publications regarding this software, it gained attention by producing an unusual and highly effective plan in the early days of StarCraft II.

Ponsen et al. (2005) use evolutionary algorithms to generate strategies in a game of Wargus. To generate the strategies, the evolutionary algorithm combines and mutates sequences of tactical and strategic-level actions in the game to form scripts (figure 8) which defeat a set of human-made and previously-evolved scripts. The fitness of each potential script is evaluated by playing it against the predefined scripts and using the resulting in-game military score combined with a time factor which favours quick wins or slow losses. Tactics are extracted as sequences of actions from the best scripts, and are finally used in a "dynamic script" that chooses particular tactics to use in a given state, based on its experience of their effectiveness – a form of RL. The resulting dynamic scripts are able to consistently beat most of the static scripts they were tested against after learning for approximately fifteen games against that opponent, but were unable to consistently beat some scripts after more than one hundred games (Ponsen et al. 2005; Ponsen et al. 2006). A drawback of this method is that the effectiveness values learned for the dynamic scripts assumes that the opponent is static and would not adapt well to a dynamic opponent (Aha, Molineaux, and Ponsen 2005).

3.7 Cognitive Architectures

An alternative method for approaching strategic-level RTS game AI is to model a reasoning mechanism on how humans are thought to operate. This could potentially lead towards greater understanding of how humans reason and allow us to create more human-like AI. This approach has been applied to StarCraft as part of a project using the Soar cognitive architecture, which adapts the BWAPI interface to communicate with a Soar agent (Turner 2012). It makes use of Soar's Spatial Visual System to deal with reconnaissance activities and pathfinding, and Soar's Working Memory to hold perceived and reasoned state information. However, it is currently limited to playing a partial game of StarCraft, using only the basic Barracks and Marine units for combat, and using hard-coded locations for building placement (Turner 2012).

A similar approach was taken by Wintermute, Xu, and Laird (2007) but it applied Soar to ORTS instead of StarCraft. They were able to interface the Soar cognitive architecture to ORTS by reducing the complexity of the problem using the concepts of grouping and attention for abstraction. These concepts are based on human perception, allowing the underlying Soar agent to receive information as a human would, post-perception – in terms of aggregated and filtered information. The agent could view entire armies of units as a single entity, but could change the focus of its attention, allowing it to perceive individual units in one location at a time, or groups of units over a wide area (figure 9). This allowed the agent to control a simple strategic-level RTS battle situation without being overwhelmed by the large number of units Wintermute, Xu, and Laird (2007). However, due to the limitations of Soar, the agent could pursue only one goal at a time, which would be very limiting in StarCraft and most complete RTS games.

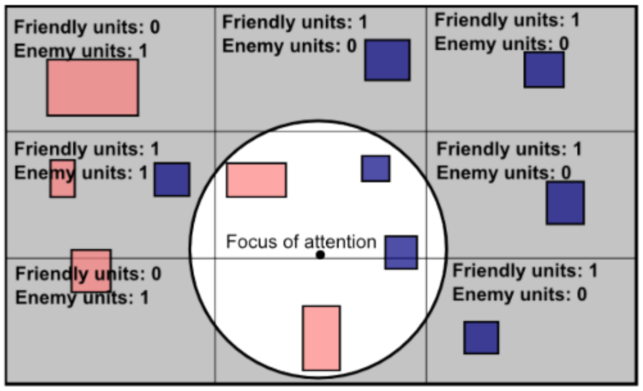

Figure 9: Attention limits the information the agent receives by hiding or abstracting objects further from the agent's area of focus (Wintermute, Xu, and Laird 2007)

3.8 Spatial Reasoning

RTS AI agents have to be able to reason about the positions and actions of often large numbers of hidden objects, many with different properties, moving over time, controlled by an opponent in a dynamic environment (Weber, Mateas, and Jhala 2011b; Wintermute, Xu, and Laird 2007). Despite the complexity of the problem, humans can reason about this information very quickly and accurately, often predicting and intercepting the location of an enemy attack or escape based on very little information, or using terrain features and the arrangement of their own units and buildings to their advantage. This makes RTS a highly suitable domain for spatial

[13]Blizzard Entertainment: StarCraft II: `blizzard.com/games/sc2/`

[14]Evolution Chamber: `code.google.com/p/evolutionchamber/`

reasoning research in a controlled environment (Buro 2004; Weber, Mateas, and Jhala 2011a; Wintermute, Xu, and Laird 2007).

Even the analysis of the terrain in RTS games, ignoring units and buildings, is a non-trivial task. In order to play effectively, players need to be able to know which regions of the terrain are connected to other regions, and where and how these regions connect. The connections between regions are as important as the regions themselves, because they offer defensive positions through which an army must move to get into or out of the region (choke points). Perkins (2010) describes the implementation and testing of the Brood War Terrain Analyzer, which has become a very common library for creating StarCraft bots capable of reasoning about their terrain. The library creates and prunes a Voronoi diagram using information about the walkable tiles of the map, identifies nodes as regions or choke points, then merges adjacent regions according to thresholds which were determined by trial and error to produce the desired results. The choke point nodes are converted into lines which separate the regions, resulting in a set of region polygons connected by choke points (figure 10). When compared against the choke points identified by humans, it had a 0–17% false negative rate, and a 4–55% false positive rate, and took up to 43 seconds to analyse the map, so there is still definite room for improvement (Perkins 2010).

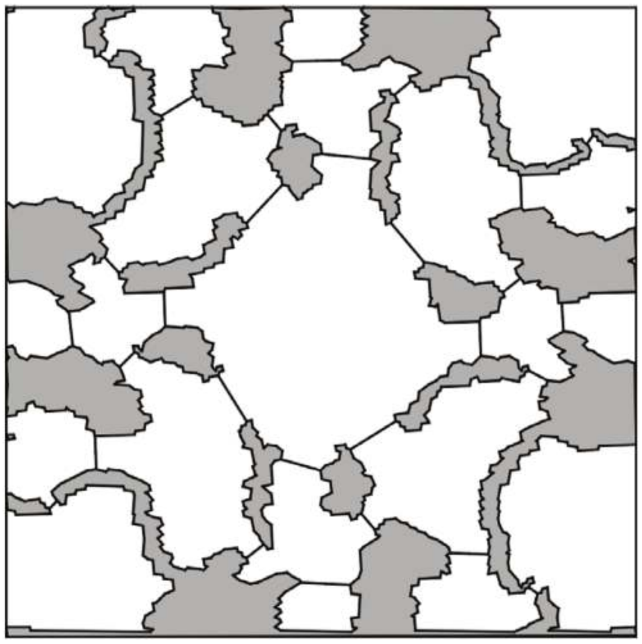

Figure 10: Terrain after analysis, showing impassable areas in grey and choke points as lines between white areas (Perkins 2010)

Once a player is capable of simple reasoning about the terrain, it is possible to begin reasoning about the movement of units over this terrain. A particularly useful spatial reasoning ability in RTS games is to be able to predict the location of enemy units while they are not visible to a player. Weber, Mateas, and Jhala (2011b) use a particle model for predicting enemy unit positions in StarCraft, based on the unit's trajectory and nearby choke points at the time it was seen. A single particle was used for each unit instead of a particle cloud because it is not possible to visually distinguish between two units of the same type, so it would be difficult to update the cloud if a unit was lost then re-sighted (Weber, Mateas, and Jhala 2011b). In order to account for the differences between the unit types in StarCraft, they divided the types into broad classes and learned a movement model for each class from professional replays on a variety of maps. The model allowed their bot to predict, with decreasing confidence over time, the subsequent locations of enemy units after sighting them, resulting in an increased win rate against other bots (Weber, Mateas, and Jhala 2011b).

The bulk of spatial reasoning research in StarCraft and other RTS games is based on Potential Fields (PFs), and to a lesser extent, influence maps. Each of these techniques help to aggregate and abstract spatial information by summing the effect of individual points of information into a field over an area, allowing decisions to be made based on the computed field strength at particular positions. They were first applied to RTS games by Hagelbäck and Johansson (2008), before which they were used for robot navigation. Kabanza et al. (2010) uses an influence map to evaluate the potential threats and opportunities of an enemy force in an effort to predict the opponent's strategy, and Uriarte and Ontañón (2012) uses one to evaluate threats and obstacles in order to control the movement of units performing a hit-and-run behavior known as kiting. (Baumgarten, Colton, and Morris 2009) uses a few different influence maps for synchronising attacks by groups of units, moving and grouping units, and choosing targets to attack. Weber and Ontañón (2010) uses PFs to aid a CBP system by taking the field strengths of many different fields at a particular position, so that the position is represented as a vector of field strengths, and can be easily compared to others stored in the case base. Synnaeve and Bessière (2011b) claims that their Bayesian model for unit movement subsumes PFs, as each unit is controlled by Bayesian sensory inputs that are capable of representing threats and opportunities in different directions relative to the unit. However, their system still needs to use damage maps in order to summarise this information for use by the sensory inputs (Synnaeve and Bessière 2011b).

PFs were used extensively in the "Overmind" StarCraft bot, for both offensive and defensive unit behavior (Huang 2011). The bot used the fields to represent opportunities and threats represented by known enemy units, using information about unit statistics so that the system could estimate how beneficial and how costly it would be to attack each target. This allowed attacking units to treat the fields as attractive and repulsive forces for movement, resulting in them automatically congregating on high-value targets and avoiding defences. Additionally, the PFs were combined with temporal reasoning components, allowing the bot to consider the time cost of reaching a faraway target, and the possible movement of enemy units around the map, based on their speed and visibility. The resulting threat map was used for threat-aware pathfinding, which routed units around more threatening regions of the map by giving movement in threatened areas a higher path cost. The major difficulty they experienced in using PFs so much was in tuning the strengths of the fields, requiring them to train the agent in small battle scenarios in order to find appropriate values (Huang 2011). To the authors' knowledge, this is the most sophisticated spatial reasoning that has been applied to playing StarCraft.

4 Plan Recognition and Learning

A major area of research in the RTS game AI literature involves learning effective strategic-level gameplay. By using an AI system capable of learning strategies, researchers aim to make computer opponents more challenging, dynamic, and human-like, while making them easier to create (Hsieh and Sun 2008). StarCraft is a very complex domain to learn from, so it may provide insights into learning to solve real-world problems. Some researchers have focused on the sub-problem of determining an opponent's strategy, which is particularly difficult in RTS games due to incomplete information about the opponent's actions, hidden by the "fog of war" (Kabanza et al. 2010). Most plan recognition makes use of an existing plan library to match against when attempting to recognise a strategy, but some methods allow for plan recognition without any predefined plans (Cheng and Thawonmas 2004; Synnaeve and Bessière 2011a). Often, data is extracted from the widely available replays files of expert human players, so a dataset was created in order to reduce repeated work (Synnaeve and Bessière 2012). This section divides the plan recognition and learning methods into deductive, abductive, probabilistic, and case-based techniques. Within each technique, plan recognition can be either intended – plans are denoted for the learner and there is often interaction between the expert and the learner – or keyhole – plans are indirectly observed and there is no two-way interaction between the expert and the learner.

4.1 Deductive

Deductive plan recognition identifies a plan by comparing the situation with hypotheses of expected behavior for various known plans. By observing particular behavior a deduction can be made about the plan being undertaken, even if complete knowledge is not available. The system described by Kabanza et al. (2010) performs intended deductive plan recognition in StarCraft by matching observations of its opponent against all known strategies which could have produced the situation. It then simulates the possible plans to determine expected future actions of its opponent, judging the probability of plans based on new observations and discarding plans which do not match (figure 11). The method used requires significant human effort to describe all possible plans in a decision tree type structure (Kabanza et al. 2010).

The decision tree machine learning method used by Weber and Mateas (2009) is another example of intended deductive plan recognition. Using training data of building construction orders and timings which have been extracted from a large selection of StarCraft replay files, it creates a

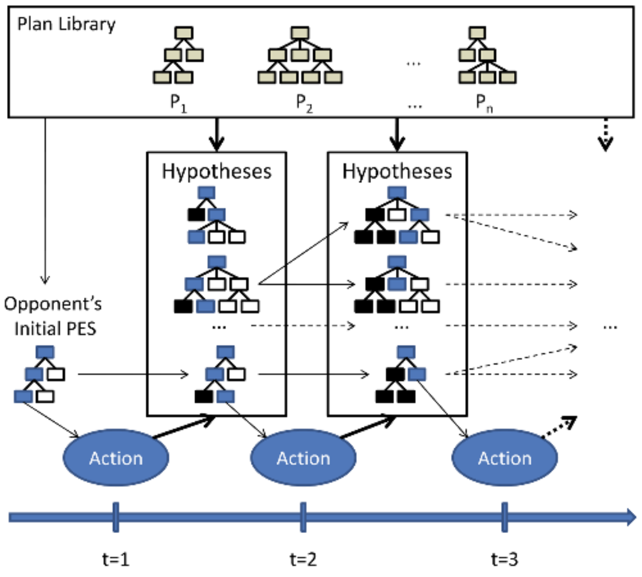

Figure 11: New observations update an opponent's possible Plan Execution Statuses to determine which plans are potentially being followed (Kabanza et al. 2010)

decision tree to predict which mid-game strategy is being demonstrated. The replays are automatically given their correct classification through a rule set based upon the build order. The learning process was also carried out with a nearest neighbour algorithm and a non-nested generalised exemplars algorithm. The resulting models were then able to predict the build order from incomplete information, with the nearest neighbour algorithm being most robust to incomplete information (Weber and Mateas 2009).

4.2 Abductive

Abductive plan recognition identifies plans by making assumptions about the situation which are sufficient to explain the observations. The GDA system described by Weber, Mateas, and Jhala (2010) is an example of intended abductive plan recognition in StarCraft, where expectations are formed about the result of actions, and unexpected events are accounted for as "discrepancies". The planner handles discrepancies by choosing from a set of predefined "explanations" which give possible reasons for discrepancies and create new goals to compensate for the change in assumed situation. This system required substantial domain engineering in order to define all of the possible goals, expectations, and explanations necessary for a domain as complex as StarCraft.

Later work added the ability for the GDA system to learn domain knowledge for StarCraft by analysing replays offline (Weber, Mateas, and Jhala 2012). In this modified system, a case library of sequential game states was built from the replays, with each case representing the player and opponent states as numerical feature vectors. Then case-based goal formulation was used to produce goals at run-time. The system forms predictions of the opponent's future state (referred to as explanations in this paper) by finding a similar opponent state to the current opponent state in the case library, looking at the future of the similar state to find the difference in the feature vectors over a set period of time, and then applying this difference to the current opponent state to produce an expected opponent state. In a similar manner, it produces a goal state by finding the expected future player state, using the predicted opponent state instead of the current state in order to find appropriate reactions to the opponent. Expectations are also formed from the case library, using changes in the opponent state to make predictions about when new types of units will be produced. When an expectation is not met (within a certain tolerance for error), a discrepancy is created, triggering the system to formulate a new goal. The resulting system appeared to show better results in testing than the previous ones, but further testing is needed to determine how effectively it adapts to unexpected situations (Weber, Mateas, and Jhala 2012).

4.3 Probabilistic

Probabilistic plan recognition makes use of statistics and expected probabilities to determine the most likely future outcome of a given situation. Synnaeve and Bessière (2011a), Dereszynski et al. (2011), and (Hostetler et al. 2012) carry out keyhole probabilistic plan recognition in StarCraft by examining build orders from professional replays, without any prior knowledge of StarCraft build orders. This means they should require minimal work to adapt to changes in the game or to apply to a new situation, because they can learn directly from replays without any human input. The models learned can then be used to predict unobserved parts of the opponent's current state, or the future strategic direction of a player, given their current and past situations. Alternatively, they can be used to recognise an unusual strategy being used in a game. The two approaches differ in the probabilistic techniques that are used, the scope in which they are applied, and the resulting predictive capabilities of the systems.

Dereszynski et al. (2011) use hidden Markov models to model the player as progressing through a series of states, each of which has probabilities for producing each unit and building type, and probabilities for which state will be transitioned to next. The model is applied to one of the sides in just one of the six possible race match-ups, and to only the first seven minutes of gameplay, because strategies are less dependant on the opponent at the start of the game. State transitions happen every 30 seconds, so the timing of predicted future events can be easily found, but it is too coarse to capture the more frequent events, such as building new worker units. Without any prior information, it is able to learn a state transition graph which closely resembles the commonly-used opening build orders (figure 12), but a thorough analysis and evaluation of its predictive power is not provided (Dereszynski et al. 2011).

Hostetler et al. (2012) extends previous work by Dereszynski et al. (2011) using a dynamic Bayesian network model for identifying strategies in StarCraft. This model explicitly takes into account the reconnaissance effort made by the player – measured by the proportion of the opponent's main bases that has been seen – in order to determine whether a unit or building was not seen because it was not present, or because little effort was made to find it. This means that failing to find a unit can actually be very informative, provided enough effort was made. The model is also more precise than prior work, predicting exact counts and production of each unit and building type each 30 second time period, instead of just presence or absence. Production of units and buildings each time period is dependent on the current state, based on a hidden Markov model as in Dereszynski et al. (2011). Again, the model was trained and applied to one side in one race match-up, and results are shown for just the first seven minutes of gameplay. For predicting unit quantities, it outperforms a baseline predictor, which simply predicts the average for the given time period, but only after reconnaissance has begun. This highlights a limitation of the model: it cannot differentiate easily between sequential time periods with similar observations, and therefore has difficulty making accurate predictions for during and after such periods. This happens because the similar periods are modelled as a single state which has a high probability of transitioning to the same state in the next period. For predicting technology structures, the model seems to generally outperform the baseline, and in both prediction tasks it successfully incorporates negative information to infer the absence of units (Hostetler et al. 2012).

Synnaeve and Bessière (2011a) carries out a similar process using a Bayesian model instead of a hidden Markov model. When given a set of thousands of replays, the Bayesian model learns the probabilities of each observed set of buildings existing at one second intervals throughout the game. These timings for each building set are modelled as normal distributions, such that few or widely spread observations will produce a large standard deviation, indicating uncertainty (Synnaeve and Bessière 2011a). Given a (partial) set of observations and a game time, the model can be queried for the probabilities of each possible building set being present at that time. Alternatively, given a sequence of times, the model can be queried for the most probable building sets over time, which can be used as a build order for the agent itself (Synnaeve and Bessière 2011a). The model was evaluated and shown to be robust to missing information, producing a building set with a little over one building wrong, on average, when 80% of the observations were randomly removed. Without missing observations and allowing for one building wrong, it was able to predict almost four buildings into the future, on average (Synnaeve and Bessière 2011a).

4.4 Case-Based

Plan recognition may also be carried out using Case-Based Reasoning (CBR) as a basis. CBR works by storing cases which represent specific knowledge of a problem and solution, and comparing new problems to past cases in order to adapt and reuse past solutions (Aamodt and Plaza 1994). It is commonly used for learning strategic play in RTS games because it can capture complex, incomplete situational knowledge gained from specific experiences to attempt to generalise about a very large problem space, without the need to

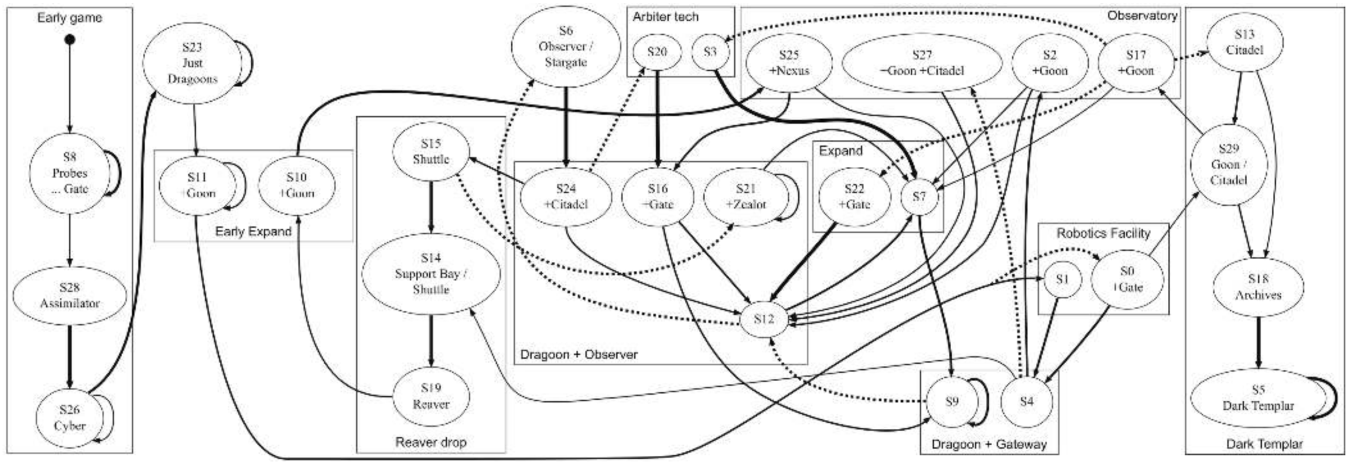

Figure 12: State transition graph learned in Dereszynski et al. (2011), showing transitions with probability at least 0.25 as solid edges, and higher-probability transitions with thicker edges. Dotted edges are low-probability transitions shown to make all nodes reachable. Labels in each state are likely units to be produced, while labels outside states are a human analysis of the strategy exhibited. (Dereszynski et al. 2011)

transform the data (Aamodt and Plaza 1994; Floyd and Esfandiari 2009; Sánchez-Pelegrín, Gómez-Martín, and Díaz-Agudo 2005).

Hsieh and Sun (2008) use CBR to perform keyhole recognition of build orders in StarCraft by analysing replays of professional players, similar to Synnaeve and Bessière (2011a) above. Hsieh and Sun (2008) use the resulting case base to predict the performance of a build order by counting wins and losses seen in the professional replays, which allows the system to predict which build order is likely to be more successful in particular situations.

In RTS games, CBR is often not only used to recognise plans, but as part of a more general method for learning actions and the situations in which they should be applied. An area of growing interest for researchers involves learning to play RTS games from a demonstration of correct behavior. These learning from demonstration techniques often use CBR and CBP, but they are discussed in their own section below.

Although much of the recent work using CBR for RTS games learns from demonstration, Baumgarten, Colton, and Morris (2009) use CBR directly without observing human play. Their system uses a set of metrics to measure performance, in order to learn to play the strategy game DEFCON[15] through an iterative process similar to RL. The system uses cases of past games played to simultaneously learn which strategic moves it should make as well as which moves its opponent is likely to make. It abstracts lower-level information about unit and structure positions by using influence maps for threats and opportunities in an area and by grouping units into fleets and meta-fleets. In order for it to make generalisations about the cases it has stored, it groups the cases similar to its current situation using a decision tree algorithm, splitting the cases into more or less successful games based on game score and hand-picked metrics. A path through the resulting decision tree is then used as a plan that is expected to result in a high-scoring game. Attribute values not specified by the selected plan are chosen at random, so the system tries different moves until an effective move is found. In this way, it can discover new plans from an initially empty case base.

4.5 Learning by Observation

For a domain as complex as RTS games, gathering and maintaining expert knowledge or learning it through trial and error can be a very difficult task, but games can provide simple access to (some of) this information through replays or traces. Most RTS games automatically create traces, recording the events within a game and the actions taken by the players throughout the game. By analysing the traces, a system can learn from the human demonstration of correct behavior, instead of requiring programmers to manually specify its behavior. This learning solely by observing the expert's external behavior and environment is usually called Learning by Observation, but is also known as Apprenticeship Learning, Imitation Learning, Behavioral Cloning, Programming by Demonstration, and even Learning from Demonstration (Ontañón, Montana, and Gonzalez 2011). These learning methods are analogous to the way humans are thought to accelerate learning through observing an expert and emulating their actions (Mehta et al. 2009).

Although the concept can be applied to other areas, learning by observation (as well as learning from demonstration, discussed in the next section) is particularly applicable for CBR systems. It can reduce or remove the need for a CBR system designer to extract knowledge from experts or think of potential cases and record them manually (Hsieh and Sun 2008; Mehta et al. 2009). The replays can be transformed into cases for a CBR system by examining the actions players take in response to situations and events, or to complete

[15]Introversion Software: DEFCON: www.introversion.co.uk/defcon

certain predefined tasks.

In order to test the effectiveness of different techniques for Learning by Observation, Floyd and Esfandiari (2009) compared CBR, decision trees, support vector machines, and naïve Bayes classifiers for a task based on RoboCup robot soccer[16]. In this task, classifiers were given the perceptions and actions of a set of RoboCup players, and were required to imitate their behavior. There was particular difficulty in transforming the observations into a form usable by most of the the classifiers, as the robots had an incomplete view of the field, so there could be very few or many objects observed at a given time (Floyd and Esfandiari 2009). All of the classifiers besides k-nearest neighbour – the classifier commonly used for CBR – required single-valued features or fixed-size feature vectors, so the missing values were filled with a placeholder item in those classifiers in order to mimic the assumptions of k-nearest neighbour. Classification accuracy was measured using the f-measure, and results showed that the CBR approach outperformed all of the other learning mechanisms (Floyd and Esfandiari 2009). These challenges and results may explain why almost all research in learning by observation and learning from demonstration in the complex domain of RTS games uses CBR as a basis.

Bakkes, Spronck, and van den Herik (2011) describes a case-based learning by observation system which is customised to playing Spring RTS games at a strategic level (figure 13), while the tactical decision-making is handled by a script. In addition to regular CBR, with cases extracted from replays, they record a fitness value with each state, so the system can intentionally select suboptimal strategies when it is winning in order to make the game more even (and hopefully more fun to play). This requires a good fitness metric for the value of a state, which is difficult to create for an RTS. In order to play effectively, the system uses hand-tuned feature weights on a chosen set of features, and chooses actions which are known to be effective against its expected opponent. The opponent strategy model is found by comparing observed features of the opponent to those of opponents in its case base, which are linked to the games where they were encountered. In order to make case retrieval efficient for accessing online, the case base is clustered and indexed with a fitness metric while offline. After playing a game, the system can add the replay to its case base in order to improve its knowledge of the game and opponent. A system capable of controlled adaptation to its opponent like this could constitute an interesting AI player in a commercial game (Bakkes, Spronck, and van den Herik 2011).

Learning by observation also makes it possible to create a domain-independent system which can simply learn to associate sets of perceptions and actions, without knowing anything about their underlying meaning (Floyd and Esfandiari 2010; Floyd and Esfandiari 2011a). However, without domain knowledge to guide decisions, learning the correct actions to take in a given situation is very difficult. To compensate, the system must process and analyse observed cases, using techniques like automated feature weighting and case clustering in order to express the relevant knowledge.

[16]RoboCup: `www.robocup.org`

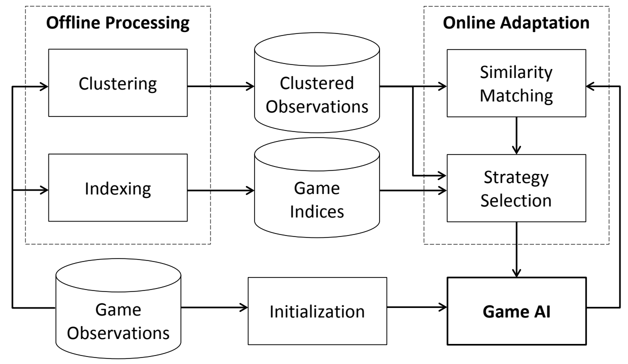

Figure 13: Learning by observation applied to an RTS: offline processing generalises observations, initialisation chooses an effective strategy, and online adaptation ensures cases are appropriate in the current situation. Adapted from Bakkes, Spronck, and van den Herik (2011)

Floyd and Esfandiari (2011a) claim their system is capable of handling complex domains with partial information and non-determinism, and show it to be somewhat effective at learning to play robot soccer and Tetris, but it has not yet been applied to a domain as complex as StarCraft. Their system has more recently been extended to be able to compare perceptions based on the entire sequence of perceptions – effectively a trace – so that it is not limited to purely reactive behavior (Floyd and Esfandiari 2011b). In the modified model, each perceived state contains a link to the previous state, so that when searching for similar states to the current state, the system can incrementally consider additional past states to narrow down a set of candidates. By also considering the similarity of actions contained in the candidate cases, the system can stop comparing past states when all of the candidate cases suggested a similar action, thereby minimising wasted processing time. In an evaluation where the correct action was dependent on previous actions, the updated system produced a better result than the original, but it is still unable to imitate an agent whose actions are based on a hidden internal state (Floyd and Esfandiari 2011b).

4.6 Learning from Demonstration

Instead of learning purely from observing the traces of interaction of a player with a game, the traces may be annotated with extra information – often about the player's internal reasoning or intentions – making the demonstrations easier to learn from, and providing more control over the particular behaviors learned. Naturally, adding annotations by hand makes the demonstrations more time-consuming to author, but some techniques have been developed to automate this process. This method of learning from constructed examples is known as Learning from Demonstration.

Given some knowledge about the actions and tasks (things that we may want to complete) in a game, there are a variety of different methods which can be used to extract cases from a trace for use in Learning by Observation or Learning from Demonstration systems. Ontañón (2012) provides an overview of several different case acquisition techniques, from the most basic reactive and monolithic learning approaches, to more complex dependency graph learning and timespan analysis techniques. Reactive learning selects a single action in response to the current situation, while monolithic sequential learning selects an entire game plan; the first has issues with preconditions and the sequence of actions, whereas the second has issues managing failures in its long-term plan (Ontañón 2012). Hierarchical sequential learning attempts to find a middle ground by learning which actions result in the completion of particular tasks, and which tasks' actions are subsets of other tasks' actions, making them subtasks. That way, ordering is retained, but when a plan fails it must only choose a new plan for its current task, instead of for the whole game (Ontañón 2012).

Sequential learning strategies can alternatively use dependency graph learning, which uses known preconditions and postconditions, and observed ordering of actions, to find a partial ordering of actions instead of using the total-ordered sequence exactly as observed. However, these approaches to determining subtasks and dependencies produce more dependencies than really exist, because independent actions or tasks which coincidentally occur at a similar time will be considered dependent (Ontañón 2012). The surplus dependencies can be reduced using timespan analysis, which removes dependencies where the duration of the action indicates that the second action started before the first one finished. In an experimental evaluation against static AI, it was found that the dependency graph and timespan analysis improved the results of each strategy they were applied to, with the best results being produced by both techniques applied to the monolithic learning strategy (Ontañón 2012).

Mehta et al. (2009) describe a CBR and planning system which is able to learn to play the game Wargus from human-annotated replays of the game (figure 14). By annotating each replay with the goals which the player was trying to achieve at the time, the system can group sequences of actions into behaviors to achieve specific goals, and learn a hierarchy of goals and their possible orderings. The learned behaviors are stored in a "behavior base" which can be used by the planner to achieve goals while playing the game. This results in a system which requires less expert programmer input to develop a game AI because it may be trained to carry out goals and behavior (Mehta et al. 2009).

The system described by Weber and Ontañón (2010) analyses StarCraft replays to determine the goals being pursued by the player with each action. Using an expert-defined ontology of goals, the system learns which sequences of actions lead to goals being achieved, and in which situations these actions occurred. Thus, it can automatically annotate replays with the goals being undertaken at each point, and convert this knowledge into a case base which is usable in a case-based planning system. The case-based planning system produced was able to play games of StarCraft by retrieving and adapting relevant cases, but was unable to beat the inbuilt scripted StarCraft AI. Weber and Ontañón (2010) suggest that the system's capability could be improved using more domain knowledge for comparing state features and identifying goals, which would make it more specific

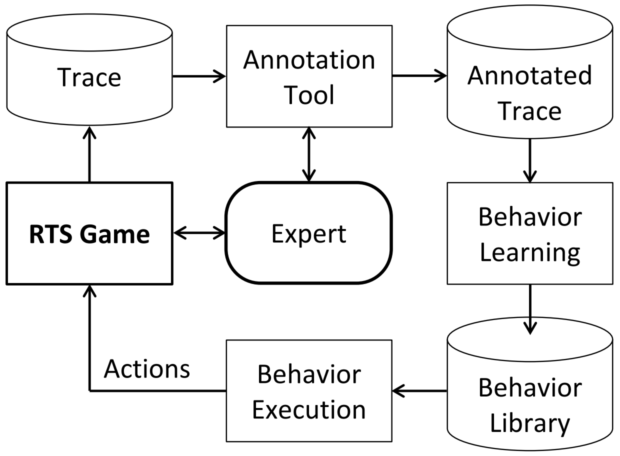

Figure 14: General architecture for a learning by demonstration system. Adapted from Mehta et al. (2009)

to StarCraft but less generally applicable.

An alternative to analysing traces is to gather the cases in real-time as the game is being played and the correct behavior is being demonstrated – known as online learning. This method has been used to train particular desired behaviors in robots learning robot soccer, so that humans could guide the learning process and apply more training if necessary (Grollman and Jenkins 2007). The training of particular desired behaviors in this way meant that fewer training examples could be covered, so while the robot could learn individual behaviors quickly, it required being set into explicit states for each behavior (Grollman and Jenkins 2007). To the authors' knowledge, such an approach has not been attempted in RTS games.

5 Open Research Areas

As well as the areas covered above, most of which are actively being researched, there are some areas which are applicable to RTS AI but seem to have been given little attention. The first of these areas are found by examining the use of game AI in industry and how it differs from academic AI. The next area – multi-scale AI – has had a few contributions but have yet to be thoroughly examined, while the third – cooperation – is all but absent from the literature. Each of these three areas raises problems that are challenging for AI agents, and yet almost trivial a human player. The final section notes the inconsistency in evaluation methods between various papers in the field, and calls for a standardised evaluation method to be put into practice.

5.1 Game AI in Industry

Despite the active research in the RTS AI field, there seems to be a large divide between the academic research, which uses new, complex AI techniques, and the games industry, which usually uses older and much simpler approaches. By examining the differences in academic and industry use of AI, we see new opportunities for research which benefit both groups.

Many papers reason that RTS AI research will be useful for new RTS game development by reducing the work involved in creating AI opponents, or by allowing game developers to create better AI opponents (Baekkelund 2006; Dill 2006; Mehta et al. 2009; Ontañón 2012; Ponsen et al. 2005; Tozour 2002; Woodcock 2002). For example, the RTS game DEFCON was given enhanced, learning AI through collaboration with the Imperial College of London (discussed in section 4.4) (Baumgarten, Colton, and Morris 2009). Similarly, Kohan II: Kings of War was produced with flexible AI through a dynamic goal selection mechanism based on complex priority calculations (discussed in section 3) (Dill 2006). More recently, the currently in-development RTS game Planetary Annihilation[17] is using flow fields for effective unit pathfinding with large numbers of units, and neural networks for controlling squads of units (Robbins 2013).

In practice, however, there is very low rate of industry adoption of academic game AI research. It is typical for industry game producers to manually specify and encode the exact behavior of their agents instead of using learning or reasoning techniques (Mehta et al. 2009; Tozour 2002; Woodcock 2002). Older techniques such as scripting, finite state machines, decision trees, and rule-based systems are still the most commonly used (Ontañón 2012; Robbins 2013; Tozour 2002; Woodcock 2002) – for example, the built-in AI of StarCraft uses a static script which chooses randomly among a small set of predetermined behaviors (Huang 2011). These techniques result in game AI which often has predictable, inflexible behavior, is subject to repeatable exploitation by humans, and doesn't learn or adapt to unforeseen situations or events (Dill 2006; Huang 2011; Ontañón 2012; Woodcock 2002).

There are two main reasons for this lack of adoption of academic AI techniques. Firstly, there is a notable difference in goals between academia and industry. Most academic work focuses on trying to create rational, optimal, agents that reason, learn, and react, while the industry aims to create challenging but defeatable opponents that are fun to play against, usually through entirely predefined behavior (Baumgarten, Colton, and Morris 2009; Davis 1999; Lidén 2004; Ontañón 2012; Tozour 2002). The two aims are linked, as players find a game more fun when it is reasonably challenging (Dicken 2011a; Hagelbäck and Johansson 2009), but this difference in goals results in very different behavior from the agents. An agent aiming to play an optimal strategy – especially if it is the same optimal strategy every game – is unlikely to make a desirable RTS opponent, because humans enjoy finding and taking advantage of opportunities and opponent mistakes (Schwab 2013). An optimal agent is also trying to win at all costs, while the industry really wants game AI that is aiming to lose the game, but in a more human-like way (Davis 1999; Schwab 2013). Making AI that acts more human-like and intelligent – even just in specific circumstances through scripted behaviors – is important in the industry as it is expected to make a game more fun and interesting for the players (Lidén 2004; Scott 2002; Woodcock 2002).

[17]Uber Entertainment: Planetary Annihilation: www.uberent.com/pa

The second major reason for the lack of adoption is that there is little demand from the games industry for new AI techniques. Industry game developers do not view their current techniques as an obstacle to making game AI that is challenging and fun to play against, and note that it is difficult to evaluate the potential of new, untested techniques (Robbins 2013; Schwab 2013; Woodcock 2002). Industry RTS games often allow AI opponents to cheat in order to make them more challenging, or emphasise playing against human opponents instead of AI (Davis 1999; Laird and van Lent 2001; Synnaeve and Bessière 2011a). Additionally, game development projects are usually under severe time and resource constraints, so trying new AI techniques is both costly and risky (Buro 2004; Robbins 2013; Tozour 2002). In contrast, the existing techniques are seen as predictable, reliable, and easy to test and debug (Dill 2006; Baekkelund 2006; Schwab 2013; Tozour 2002; Woodcock 2002). Academic AI techniques are also seen as difficult to customise, tune, or tweak in order to perform important custom scripted tasks, which scripted AI is already naturally suited to doing (Robbins 2013; Schwab 2013).

Some new avenues of research come to light considering the use of game AI in industry. Most importantly, creating AI that is more human-like, which may also make it more fun to play against. This task could be approached by making an RTS AI that is capable of more difficult human interactions. Compared to AI, human players are good at working together with allies, using surprises, deception, distractions and coordinated attacks, planning effective strategies, and changing strategies to become less predictable (Scott 2002). Players that are able to do at least some of these things appear to be intelligent and are more fun for human players to play against (Scott 2002). In addition, being predictable and exploitable in the same fashion over multiple games means that human players do not get to find and exploit new mistakes, removing a source of enjoyment from the game. AI can even make mistakes and still appear intelligent as long as the mistake appears plausible in the context of the game – the sort of mistakes which a human would make (Lidén 2004).

An alternative way to create AI that is more human-like is to replicate human play-styles and skills. Enabling an AI to replicate particular strategies – for example a heavily defensive "turtle" strategy or heavily offensive "rush" strategy – would give the AI more personality and allow players to practice against particular strategies (Schwab 2013). This concept has been used in industry AI before (Dill 2006) but may be difficult to integrate into more complex AI techniques. A system capable of learning from a human player – using a technique such as Learning from Demonstration (see section 4.6), likely using offline optimisation – could allow all or part of the AI to be trained instead of programmed (Floyd and Esfandiari 2010; Mehta et al. 2009). Such a system could potentially copy human skills – like unit micromanagement or building placement – in order to keep up with changes in how humans play a game over time, which makes it an area of particular interest to the industry (Schwab 2013).

Evaluating whether an RTS AI is human-like is potentially an issue. For FPS games, there is an AI competition, BotPrize[18], for creating the most human-like bots (AI players), where the bots are judged on whether they appear to be a human playing the game – a form of Turing Test (Dicken 2011b). This test has finally been passed in 2012, with two bots judged more likely to be humans than bots for the first time. Appearing human-like in an RTS would be an even greater challenge than in an FPS, as there are more ways for the player to act and react to every situation, and many actions are much more visible than the very fast-paced transient actions of an FPS. However, being human-like is not currently a focus of any StarCraft AI research, to the authors' knowledge, although it has been explored to a very small extent in the context of some other RTS games. It is also not a category in any of the current StarCraft AI competitions. The reason for this could be the increased difficulty of creating a human level agent for RTS games compared with FPS games, however, it may simply be due to an absence of goals in this area of game AI research. A Turing Test similar to BotPrize could be designed for StarCraft bots by making humans play in matches and then decide whether their opponent was a human or a bot. It could be implemented fairly easily on a competitive ladder like ICCup by simply allowing a human to join a match and asking them to judge the humanness of their opponent during the match. Alternatively, the replay facility in StarCraft could be used to record matches between bots and humans of different skill levels, and other humans could be given the replays to judge the humanness of each player. Due to the popularity of StarCraft, expert participants and judges should be relatively easy to find.

A secondary avenue of research is in creating RTS AI that is more accessible or useful outside of academia. This can partially be addressed by simply considering and reporting how often the AI can be relied upon to behave as expected, how performant the system is, and how easily the system can be tested and debugged. However, explicit research into these areas could yield improvements that would benefit both academia and industry. More work could also be done to investigate how to make complex RTS AI systems easier to tweak and customise, to produce specific behavior while still retaining learning or reasoning capabilities. Industry feedback indicates it is not worthwhile to adapt individual academic AI techniques in order to apply them to individual games, but it may become worthwhile if techniques could be reused for multiple games in a reliable fashion. A generalised RTS AI middleware could allow greater industry adoption – games could be more easily linked to the middleware and then tested with multiple academic techniques – as well as a wider evaluation of academic techniques over multiple games. Research would be required in order to find effective abstractions for such a complex and varied genre of games, and to show the viability of this approach.

[18]BotPrize: `botprize.org`

5.2 Multi-Scale AI

Due to the complexity of RTS games, current bots require multiple abstractions and reasoning mechanisms working in concert in order to play effectively (Churchill and Buro 2012; Weber et al. 2010; Weber, Mateas, and Jhala 2011a). In particular, most bots have separate ways of handling tactical and strategic level decision-making, as well as separately managing resources, construction, and reconnaissance. Each of these modules faces an aspect of an interrelated problem, where actions taken will have long-term strategic trade-offs affecting the whole game, so they cannot simply divide the problem into isolated or hierarchical problems. A straightforward hierarchy of command – like in a real-world military – is difficult in an RTS because the decisions of the top-level commander will depend on, and affect, multiple subproblems, requiring an understanding of each one as well as how they interact. For example, throughout the game, resources could be spent on improving the resource generation, training units for an army, or constructing new base infrastructure, with each option controlled by a different module which cannot assess the others' situations. Notably, humans seem to be able to deal with these problems very well through a combination of on- and off-line, reactive, deliberative and predictive reasoning.

Weber et al. (2010) defines the term "multi-scale AI problems" to refer to these challenges, characterised by concurrent and coordinated goal pursuit across multiple abstractions. They go on to describe several different approaches they are using to integrate parts of their bot. First is a working memory or "shared blackboard" concept for indirect communication between their modules, where each module publishes its current beliefs for the others to read. Next, they allow for goals and plans generated by their planning and reasoning modules to be inserted into their central reactive planning system, to be pursued in parallel with current goals and plans. Finally, they suggest a method for altered behavior activation, so that modules can modify the preconditions for defined behaviors, allowing them to activate and deactivate behaviors based on the situation.

A simpler approach may be effective for at least some parts of an RTS bot. Synnaeve and Bessière (2011b) use a higher-level tactical command, such as scout, hold position, flock, or fight, as one of the inputs to their micromanagement controller. Similarly, Churchill and Buro (2012) use a hierarchical structure for unit control, with an overall game commander – the module which knows about the high-level game state and makes strategic decisions – giving commands to a macro commander and a combat commander, each of which give commands to their sub-commanders. Commanders further down the hierarchy are increasingly focused on a particular task, but have less information about the overall game state, so therefore must rely on their parents to make them act appropriately in the bigger picture. This is relatively effective because the control of units is more hierarchically arranged than other aspects of an RTS. Such a system allows the low-level controllers to incorporate information from their parent in the hierarchy, but they are unable to react and coordinate with other low-level controllers directly in order to perform cooperative actions (Synnaeve and Bessière 2011b). Most papers on StarCraft AI skirt this issue by focusing on one aspect of the AI only, as can be seen in how this review paper is divided into tactical and strategic decision-making sections.

5.3 Cooperation

Cooperation is an essential ability in many situations, but RTS games present a particular complex environment in which the rules and overall goal are fixed, and there is a limited ability to communicate with your cooperative partner(s). It would also be very helpful in commercial games, as good cooperative players could be used for coaching or team games. In team games humans often team up to help each other with coordinated actions throughout the game, like attacking and defending, even without actively communicating. Conversely AI players in most RTS games (including StarCraft) will act seemingly independently of their teammates. A possible beginning direction for this research could be to examine some techniques developed for opponent modelling and reuse them for modelling an ally, thus giving insight into how the player should act to coordinate with the ally. Alternatively, approaches to teamwork and coordination used in other domains, such as RoboCup (Kitano et al. 1998) may be appropriate to be adapted or extended for use in the RTS domain.

Despite collaboration being highlighted as a challenging AI research problem in Buro (2003), to the authors' knowledge just one research publication focusing on collaborative behavior exists in the domain of StarCraft (and RTS games in general). Magnusson and Balsasubramaniyan (2012) modified an existing StarCraft bot to allow both communication of the bot's intentions and in-game human control of the bot's behavior. It was tested in a small experiment in which a player is allied with the bot, with or without the communication and control elements, against two other bots. The players rated the communicating bots as more fun to play with than the non-communicating bots, and more experienced players preferred to be able to control the bot while novice players preferred a non-controllable bot. Much more research is required to investigate collaboration between humans and bots, as well as collaboration between bots only.

5.4 Standardised Evaluation

Despite games being a domain that is inherently suited to evaluating the effectiveness of the players and measuring performance, it is difficult to make fair comparisons between the results of most literature in the StarCraft AI field. Almost every paper has a different method for evaluating their results, and many of these experiments are of poor quality. Evaluation is further complicated by the diversity of applications, as many of the systems developed are not suited to playing entire games of StarCraft, but are suited to a specific sub-problem. Such a research community, made up of isolated studies which are not mutually comparable, was recognised as problematic by Aha and Molineaux (2004). Their Testbed for Integrating and Evaluating Learning Techniques (TIELT), which aimed to standardise the learning environment for evaluation, attempted to address the problem but unfortunately never became very widely used.

Partial systems – those that are unable to play a full game of StarCraft – are often evaluated using a custom metric, which makes comparison between such systems nearly impossible. A potential solution for this would be to select a common set of parts which could plug in to partial systems and allow them to function as a complete system for testing. This may be possible by compartmentalising parts of an open-source AI used in a StarCraft AI competition, such as UAlbertaBot (Churchill and Buro 2012), which is designed to be modular, or using an add-on library such as the BWAPI Standard Add-on Library (BWSAL)[19]. Alternatively, a set of common tests could be made for partial systems to be run against. Such tests could examine common sub-problems of an AI system, such as tactical decision-making, planning, and plan recognition, as separate suites of tests. Even without these tests in place, new systems should at least be evaluated against representative related systems in order to show that they represent a non-trivial improvement.

Results published about complete systems are similarly difficult to compare against one another due to their varied methods of evaluation. Some of the only comparable results come from systems demonstrated against the inbuilt StarCraft AI, despite the fact that the inbuilt AI is a simple scripted strategy which average human players can easily defeat (Weber, Mateas, and Jhala 2010). Complete systems are more effectively tested in StarCraft AI competitions, but these are run infrequently, making quick evaluation difficult. An alternative method of evaluation is to automatically test the bots against other bots in a ladder tournament, such as in the StarCraft Brood War Ladder for BWAPI Bots[20]. In order to create a consistent benchmark of bot strength, a suite of tests could be formed from the top three bots from each of the AIIDE StarCraft competitions on a selected set of tournament maps. This would provide enough variety to give a general indication of bot strength, and it would allow for results to be compared between papers and over different years. An alternative to testing bots against other bots is testing them in matches against humans, such as how Weber, Mateas, and Jhala (2010) tested their bot in the ICCup.

Finally, it may be useful to have a standard evaluation method for goals other than finding the AI best at winning the game. For example, the game industry would be more interested in determining the AI which is most fun to play against, or the most human-like. A possible evaluation for these alternate objectives was discussed in section 5.1.

6 Conclusion

This paper has reviewed the literature on artificial intelligence for real-time strategy games focusing on StarCraft. It found significant research focus on tactical decision-making, strategic decision-making, plan recognition and strategy learning. Three main areas were identified where future research could have a large positive impact. Firstly creating RTS AI that is more human-like would be an in-

[19]BWAPI Standard Add-on Library: `code.google.com/p/bwsal`

[20]StarCraft Brood War Ladder for BWAPI Bots: `bots-stats.krasi0.com`

teresting challenge and may help to bridge the gap between academia and industry. The other two research areas discussed were noted to be lacking in research contributions, despite being highly appropriate for Real-Time Strategy game research: multi-scale AI, and cooperation. Finally, the paper finished with a call for increased rigour and ideally standardisation of evaluation methods, so that different techniques can be compared on even ground. Overall the RTS AI field is small but very active, with the StarCraft agents showing continual improvement each year, as well as gradually becoming more based upon machine learning, learning from demonstration, and reasoning, instead of using scripted or fixed behaviors.

Acronyms

AI Artificial Intelligence

BWAPI Brood War Application Programming Interface

CBP Case-Based Planning

CBR Case-Based Reasoning

FPS First-Person Shooter

GDA Goal-Driven Autonomy

GOAP Goal-Oriented Action Planning

HTN Hierarchical Task Network

ICCup International Cyber Cup

PF Potential Field

RL Reinforcement Learning

RTS Real-Time Strategy

References

Aamodt, A., and Plaza, E. 1994. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications* 7(1):39–59.

Aha, D. W., and Molineaux, M. 2004. Integrating learning in interactive gaming simulators. In *Proceedings of the AAAI Workshop on Challenges in Game AI*.

Aha, D.; Molineaux, M.; and Ponsen, M. 2005. Learning to win: Case-based plan selection in a real-time strategy game. In Muñoz-Ávila, H., and Ricci, F., eds., *Case-Based Reasoning. Research and Development*, volume 3620 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 5–20.

Baekkelund, C. 2006. Academic AI research and relations with the games industry. In Rabin, S., ed., *AI Game Programming Wisdom*, volume 3. Boston, MA: Charles River Media. 77–88.

Bakkes, S.; Spronck, P.; and van den Herik, J. 2011. A CBR-inspired approach to rapid and reliable adaption of video game ai. In *Proceedings of the Workshop on Case-Based Reasoning for Computer Games at the International Conference on Case-Based Reasoning (ICCBR)*, 17–26.

Balla, R., and Fern, A. 2009. UCT for tactical assault planning in real-time strategy games. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 40–45.

Baumgarten, R.; Colton, S.; and Morris, M. 2009. Combining AI methods for learning bots in a real-time strategy game. *International Journal of Computer Games Technology* 2009:10.

Buckland, M. 2005. *Programming Game AI by Example*. Wordware Publishing, Inc.

Buro, M., and Churchill, D. 2012. Real-time strategy game competitions. *AI Magazine* 33(3):106–108.

Buro, M., and Furtak, T. M. 2004. RTS games and real-time AI research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference*, 63–70. Citeseer.

Buro, M. 2003. Real-time strategy games: a new AI research challenge. In *Proceedings of the IJCAI*, 1534–1535. Citeseer.

Buro, M. 2004. Call for AI research in RTS games. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, 139–142.

Cadena, P., and Garrido, L. 2011. Fuzzy case-based reasoning for managing strategic and tactical reasoning in StarCraft. In Batyrshin, I., and Sidorov, G., eds., *Advances in Artificial Intelligence*, volume 7094 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 113–124.

Champandard, A. J. 2011. This year in game AI: Analysis, trends from 2010 and predictions for 2011. `http://aigamedev.com/open/editorial/2010-retrospective/`. Retrieved 26 September 2011.

Chan, H.; Fern, A.; Ray, S.; Wilson, N.; and Ventura, C. 2007. Online planning for resource production in real-time strategy games. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 65–72.

Cheng, D., and Thawonmas, R. 2004. Case-based plan recognition for real-time strategy games. In *Proceedings of the GAME-ON Conference*, 36–40. Reading, UK: University of Wolverhampton Press.

Chung, M.; Buro, M.; and Schaeffer, J. 2005. Monte carlo planning in RTS games. In Kendall, G., and Lucas, S., eds., *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 117–124.

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

Churchill, D., and Buro, M. 2012. Incorporating search algorithms into RTS game agents. In *Proceedings of the AIIDE Workshop on AI in Adversarial Real-Time Games*, 2–7. AAAI Press.

Churchill, D.; Saffidine, A.; and Buro, M. 2012. Fast heuristic search for RTS game combat scenarios. In *Proceedings of the AIIDE Conference*, 112–117.

Davis, I. L. 1999. Strategies for strategy game AI. In *Proceedings of the AAAI Spring Symposium on Artificial Intelligence and Computer Games*, 24–27.

Dereszynski, E.; Hostetler, J.; Fern, A.; Dietterich, T.; Hoang, T.; and Udarbe, M. 2011. Learning probabilistic behavior models in real-time strategy games. In *Proceedings of the AIIDE Conference*, 20–25. AAAI Press.

Dicken, L. 2011a. A difficult subject. `http://altdevblogaday.com/2011/05/12/a-difficult-subject/`. Retrieved 19 September 2011.

Dicken, L. 2011b. A turing test for bots. `http://altdevblogaday.com/2011/09/09/a-turing-test-for-bots/`. Retrieved 19 September 2011.

Dill, K. 2006. Prioritizing actions in a goal-based RTS AI. In Rabin, S., ed., *AI Game Programming Wisdom*, volume 3. Boston, MA: Charles River Media. 321–330.

Floyd, M., and Esfandiari, B. 2009. Comparison of classifiers for use in a learning by demonstration system for a situated agent. Presented at the Workshop on Case-Based Reasoning for Computer Games at the ICCBR.

Floyd, M., and Esfandiari, B. 2010. Toward a domain independent case-based reasoning approach for imitation: Three case studies in gaming. In *Proceedings of the Workshop on Case-Based Reasoning for Computer Games at the ICCBR*, 55–64.

Floyd, M. W., and Esfandiari, B. 2011a. A case-based reasoning framework for developing agents using learning by observation. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, 531–538.

Floyd, M., and Esfandiari, B. 2011b. Learning state-based behaviour using temporally related cases. Presented at the UK Workshop on CBR.

Gabriel, I.; Negru, V.; and Zaharie, D. 2012. Neuroevolution based multi-agent system for micromanagement in real-time strategy games. In *Proceedings of the Fifth Balkan Conference in Informatics*, 32–39. ACM.

Grollman, D., and Jenkins, O. 2007. Learning robot soccer skills from demonstration. In *Proceedings of the IEEE International Conference on Development and Learning*, 276–281.

Hagelbäck, J., and Johansson, S. J. 2008. The rise of potential fields in real time strategy bots. In *Proceedings of the AIIDE Conference*, 42–47. AAAI Press.

Hagelbäck, J., and Johansson, S. 2009. Measuring player experience on runtime dynamic difficulty scaling in an RTS game. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 46–52. IEEE.

Hostetler, J.; Dereszynski, E.; Dietterich, T.; and Fern, A. 2012. Inferring strategies from limited reconnaissance in real-time strategy games. In *Proceedings of the Annual Conference on Uncertainty in Artificial Intelligence*, 367–376.

Hsieh, J., and Sun, C. 2008. Building a player strategy model by analyzing replays of real-time strategy games. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, 3106–3111. Hong Kong, China: IEEE.

Huang, H. 2011. Skynet meets the swarm: how the Berkeley Overmind won the 2010 StarCraft AI competition. `http://arstechnica.com/gaming/news/2011/01/skynet-meets-the-swarm-how-the-berkeley-overmind-won-the-2010-starcraft-ai-competition.ars`. Retrieved 8 September 2011.

Jaidee, U.; Muñoz-Avila, H.; and Aha, D. 2011. Integrated learning for goal-driven autonomy. In *Proceedings of the IJCAI*, 2450–2455.

Judah, K.; Roy, S.; Fern, A.; and Dietterich, T. G. 2010. Reinforcement learning via practice and critique advice. In *Proceedings of the Association for the Advancement of Artificial Intelligence (AAAI) Conference on AI*.

Kabanza, F.; Bellefeuille, P.; Bisson, F.; Benaskeur, A.; and Irandoust, H. 2010. Opponent behaviour recognition for real-time strategy games. In *Proceedings of the AAAI Workshop on Plan, Activity, and Intent Recognition*.

Kitano, H.; Tambe, M.; Stone, P.; Veloso, M.; Coradeschi, S.; Osawa, E.; Matsubara, H.; Noda, I.; and Asada, M. 1998. The robocup synthetic agent challenge 97. In Kitano, H., ed., *RoboCup-97: Robot Soccer World Cup I*, volume 1395 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 62–73.

Laagland, J. 2008. A HTN planner for a real-time strategy game. Available: `http://hmi.ewi.utwente.nl/verslagen/capita-selecta/CS-Laagland-Jasper.pdf`.

Laird, J., and van Lent, M. 2001. Human-level AI's killer application: Interactive computer games. *AI Magazine* 22(2):15–26.

Lidén, L. 2004. Artificial stupidity: The art of intentional mistakes. In Rabin, S., ed., *AI Game Programming Wisdom*, volume 2. Hingham, MA: Charles River Media. 41–48.

Magnusson, M. M., and Balsasubramaniyan, S. K. 2012. A communicating and controllable teammate bot for RTS games. Master's thesis, School of Computing, Blekinge Institute of Technology.

Manslow, J. 2004. Using reinforcement learning to solve AI control problems. In Rabin, S., ed., *AI Game Programming Wisdom*, volume 2. Hingham, MA: Charles River Media. 591–601.

Marthi, B.; Russell, S.; Latham, D.; and Guestrin, C. 2005. Concurrent hierarchical reinforcement learning. In *Proceedings of the IJCAI*, 779–785.

Mateas, M., and Stern, A. 2002. A behavior language for story-based believable agents. *IEEE Intelligent Systems* 17(4):39–47.

Mehta, M.; Ontañón, S.; Amundsen, T.; and Ram, A. 2009. Authoring behaviors for games using learning from demonstration. Presented at the Workshop on Case-Based Reasoning for Computer Games at the ICCBR.

Mishra, K.; Ontañón, S.; and Ram, A. 2008. Situation assessment for plan retrieval in real-time strategy games. In Althoff, K.-D.; Bergmann, R.; Minor, M.; and Hanft, A., eds., *Advances in Case-Based Reasoning*, volume 5239 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 355–369.

Molineaux, M.; Aha, D.; and Moore, P. 2008. Learning continuous action models in a real-time strategy environment. In *Proceedings of the International Florida Artificial Intelligence Research Society (FLAIRS) Conference*, 257–262.

Molineaux, M.; Klenk, M.; and Aha, D. 2010. Goal-driven autonomy in a navy strategy simulation. In *Proceedings of the AAAI Conference on AI*. Atlanta, GA: AAAI Press.

Muñoz-Avila, H., and Aha, D. 2004. On the role of explanation for hierarchical case-based planning in real-time strategy games. In *Proceedings of ECCBR Workshop on Explanations in CBR*. Citeseer.

Nejati, N.; Langley, P.; and Konik, T. 2006. Learning hierarchical task networks by observation. In *Proceedings of the International Conference on Machine Learning*, 665–672.

Ontañón, S.; Mishra, K.; Sugandh, N.; and Ram, A. 2007. Case-based planning and execution for real-time strategy games. In Weber, R., and Richter, M., eds., *Case-Based Reasoning. Research and Development*, volume 4626 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 164–178.

Ontañón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. In press. A survey of real-time strategy game AI research and competition in StarCraft. *Transactions of Computational Intelligence and AI in Games* 5(4):1–19.

Ontañón, S.; Montana, J.; and Gonzalez, A. 2011. Towards a unified framework for learning from observation. Presented at the Workshop on Agents Learning Interactively from Human Teachers at IJCAI.

Ontañón, S. 2012. Case acquisition strategies for case-based reasoning in real-time strategy games. In *Proceedings of the International FLAIRS Conference*.

Orkin, J. 2004. Applying goal-oriented action planning to games. In Rabin, S., ed., *AI Game Programming Wisdom*, volume 2. Hingham, MA: Charles River Media. 217–227.

Palma, R.; Sánchez-Ruiz, A.; Gómez-Martín, M.; Gómez-Martín, P.; and González-Calero, P. 2011. Combining expert knowledge and learning from demonstration in real-time strategy games. In Ram, A., and Wiratunga, N., eds., *Case-Based Reasoning Research and Development*, volume 6880 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 181–195.

Perkins, L. 2010. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. In *Proceedings of the AIIDE Conference*, 168–173. AAAI Press.

Ponsen, M.; Muñoz-Avila, H.; Spronck, P.; and Aha, D. 2005. Automatically acquiring domain knowledge for adaptive game AI using evolutionary learning. In *Proceedings of the Innovative Applications of Artificial Intelligence Conference*, 1535–1540. AAAI Press.

Ponsen, M.; Muñoz-Avila, H.; Spronck, P.; and Aha, D. 2006. Automatically generating game tactics through evolutionary learning. *AI Magazine* 27(3):75–84.

Robbins, M. 2013. Personal communication. Software Engineer at Uber Entertainment, formerly Gameplay Engineer at Gas Powered Games.

Sailer, F.; Buro, M.; and Lanctot, M. 2007. Adversarial planning through strategy simulation. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 80 –87.

Sánchez-Pelegrín, R.; Gómez-Martín, M.; and Díaz-Agudo, B. 2005. A CBR module for a strategy videogame. In *Proceedings of the Workshop on Computer Gaming and Simulation Environments at the ICCBR*, 217–226. Citeseer.

Schaeffer, J. 2001. A gamut of games. *AI Magazine* 22(3):29–46.

Schwab, B. 2013. Personal communication. Senior AI/Gameplay Engineer at Blizzard Entertainment.

Scott, B. 2002. The illusion of intelligence. In Rabin, S., ed., *AI Game Programming Wisdom*, volume 1. Hingham, MA: Charles River Media. 16–20.

Shantia, A.; Begue, E.; and Wiering, M. 2011. Connectionist reinforcement learning for intelligent unit micro management in StarCraft. Presented at the International Joint Conference on Neural Networks.

Sharma, M.; Holmes, M.; Santamaria, J.; Irani, A.; Isbell, C.; and Ram, A. 2007. Transfer learning in real-time strategy games using hybrid CBR/RL. In *Proceedings of the IJCAI*.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement learning: An introduction*. Cambridge Massachusetts: MIT Press.

Synnaeve, G., and Bessière, P. 2011a. A bayesian model for plan recognition in RTS games applied to StarCraft. In *Proceedings of the AIIDE Conference*, 79–84. AAAI Press.

Synnaeve, G., and Bessière, P. 2011b. A bayesian model for RTS units control applied to StarCraft. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, 190–196.

Synnaeve, G., and Bessière, P. 2012. A dataset for StarCraft AI and an example of armies clustering. In *Proceedings of the AIIDE Workshop on AI in Adversarial Real-Time Games*.

Szczepański, T., and Aamodt, A. 2009. Case-based reasoning for improved micromanagement in real-time strategy games. Presented at the Workshop on Case-Based Reasoning for Computer Games at the ICCBR.

Tozour, P. 2002. The evolution of game AI. In Rabin, S., ed., *AI Game Programming Wisdom*, volume 1. Hingham, MA: Charles River Media. 3–15.

Turner, A. 2012. Soar-SC: A platform for AI research in StarCraft: Brood War. `https://github.com/bluechill/Soar-SC/tree/master/Soar-SC-Papers`. Retrieved 15 February 2013.

Uriarte, A., and Ontañón, S. 2012. Kiting in rts games using influence maps. In *Proceedings of the AIIDE Workshop on AI in Adversarial Real-Time Games*, 31–36.

Weber, B., and Mateas, M. 2009. A data mining approach to strategy prediction. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 140–147. IEEE.

Weber, B., and Ontañón, S. 2010. Using automated replay annotation for case-based planning in games. Presented at the Workshop on Case-Based Reasoning for Computer Games at the ICCBR.

Weber, B.; Mawhorter, P.; Mateas, M.; and Jhala, A. 2010. Reactive planning idioms for multi-scale game AI. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, 115–122. IEEE.

Weber, B.; Mateas, M.; and Jhala, A. 2010. Applying goal-driven autonomy to StarCraft. In *Proceedings of the AIIDE Conference*, 101–106. AAAI Press.

Weber, B.; Mateas, M.; and Jhala, A. 2011a. Building human-level AI for real-time strategy games. In *Proceedings of the AAAI Fall Symposium Series*, 329–336. AAAI.

Weber, B.; Mateas, M.; and Jhala, A. 2011b. A particle model for state estimation in real-time strategy games. In *Proceedings of the AIIDE Conference*, 103–108. AAAI Press.

Weber, B.; Mateas, M.; and Jhala, A. 2012. Learning from demonstration for goal-driven autonomy. In *Proceedings of the AAAI Conference on AI*, 1176–1182.

Wintermute, S.; Xu, J.; and Laird, J. 2007. SORTS: A human-level approach to real-time strategy AI. In *Proceedings of the AIIDE Conference*, 55–60. AAAI Press.

Woodcock, S. 2002. Foreword. In Buckland, M., ed., *AI Techniques for Game Programming*. Premier Press.

Ian Watson is Assoc. Prof. of Artificial Intelligence in the Dept of Computer Science at the University of Auckland, New Zealand. With a background in expert systems Ian became interested in case-based reasoning (CBR) to reduce the knowledge engineering bottleneck. Ian has remained active in CBR focusing on game AI along side other techniques. Ian also has an interest in the history of computing writing a popular science book called The Universal Machine.

Glen Robertson is a PhD candidate at the University of Auckland, working under the supervision of Ian Watson. Glen's research interests are in machine learning and artificial intelligence, particularly in unsupervised learning for complex domains with large datasets.