Case Acquisition Strategies for Case-Based Reasoning in Real-Time Strategy Games

Santiago Ontañón
Computer Science Department
Drexel University
Philadelphia, PA, USA
santi@cs.drexel.edu

Abstract

Real-time Strategy (RTS) games are complex domains which are a significant challenge to both human and artificial intelligence (AI). For that reason, and although many AI approaches have been proposed for the RTS game AI problem, the AI of all commercial RTS games is scripted and offers a very static behavior subject to exploits. In this paper, we will focus on a case-based reasoning (CBR) approach to this problem, and concentrate on the process of case-acquisition. Specifically, we will describe 7 different techniques to automatically acquire plans by observing human demonstrations and compare their performance when using them in the Darmok 2 system in the context of an RTS game.

Introduction

Real-time Strategy (RTS) games are complex domains, offering a significant challenge to both human and artificial intelligence (AI). Designing AI techniques that can play RTS games is a challenging problem because of several reasons: RTS games have huge decision and state spaces, they are non-deterministic, partially observable and real time (Ontañón et al. 2010; Aha, Molineaux, and Ponsen 2005). Moreover, RTS games requires spatial and temporal reasoning, resource management, adversarial planning, uncertainty management and opponent modeling (Buro 2003).

Many approaches to deal with the problem of RTS game AI have been proposed, spurred by competitions like the AIIDE[1] or CIG Starcraft AI competitions[2] (next section provides a quick overview of the different challenges RTS games pose, and how have they been approached from an AI perspective). In this paper, we will focus on a case-based reasoning (CBR) approach to this problem.

Specifically, we will focus on the problem of where do cases come from in CBR approaches to RTS game AI, and approach it from a learning from demonstration perspective. Learning from demonstration has been used before in the context of CBR for case acquisition (Ontañón et al. 2010; Weber and Mateas 2009; Floyd, Esfandiari, and Lam 2008) with promising results. The main contribution of this paper is a description and comparison of 7 different case acquisition strategies in the context of RTS games, showing their strengths and weaknesses. The case acquisition techniques have been incorporated into the Darmok 2 (Ontañón et al. 2009) system, and evaluated in a RTS game, called S3, specifically developed for experimentation, but still including all the complexity of RTS games.

[1]http://skatgame.net/mburo/sc2011/

[2]http://ls11-www.cs.uni-dortmund.de/rts-competition/starcraft-cig2011

Background

As mentioned before, RTS games are a very challenging domain from the point of view of AI, mainly due to their huge state and decision space. In comparison, an average position in the game of Chess has in the order of 20 possible moves; in a standard RTS game the total number of possible moves is in the order of thousands or even millions. The size of the state and decision space and the fact that the games are real-time (rather than turn-based), non deterministic and partially observable, makes standard game tree search approaches non applicable (although they have been tried (Chung, Buro, and Schaeffer 2005)).

The most common approach is to hard-code human strategies. For example, the work of McCoy and Mateas (McCoy and Mateas 2008), or many of the entries to the latest editions of the Starcraft AI competition use this strategy. The main difficulties here is that humans combine a collection of different strategies at many different levels of abstraction, and it is not obvious how to make all of those strategies interact in a unified integrated architecture.

Given the complexity of hard-coding such strategies, automatic learning techniques have been proposed, however, they cannot cope with the complexity of the full game, and have to be applied to specific sub-tasks in the game, or to an abstracted version of the game. For example, Aha et al. (Aha, Molineaux, and Ponsen 2005) use case-based reasoning to learn to select between a repertoire of predefined strategies in different game situations. Another approach is that of reinforcement learning (Marthi et al. 2005; Sharma et al. 2007), although it only works for small instances of RTS games with a very limited number of units (typically less than 10).

Learning from demonstration approaches have the advantage that they do not have to explore the whole state space of the game. For example, Könik and Laird (Könik and Laird

2006) study how to learn goal-subgoal decompositions by learning from human demonstrations. Weber and Mateas (Weber and Mateas 2009) use learning from demonstration to predict the opponent strategy.

The work presented in this paper is closely related to the work of Floyd et al. (Floyd, Esfandiari, and Lam 2008) and Ontañón et al. (Ontañón et al. 2010). Both approaches use learning from demonstration as a case-acquisition strategy inside of a case-based reasoning system that uses the learned cases to play the game.

Learning from Demonstration, called sometimes "learning from observation" or "apprenticeship learning" (Ontañón, Montaña, and Gonzalez 2011), has been widely studied in robotics and offers an alternative to manual programming. Human demonstrations have also received some attention to speed-up reinforcement learning (Schaal 1996), and as a way of automatically acquiring planning knowledge (Hogg, Muñoz-Avila, and Kuter 2008). A recent overview of the state of the art can be found in (Argall et al. 2009). For the purposes of this paper it suffices to say that learning from demonstration in the context of case-based reasoning systems involves two steps: 1) acquiring cases from expert demonstrations, and 2) reusing those cases to actually play the game. This paper focuses on the first step.

Darmok 2

Darmok 2 (D2) (Ontañón et al. 2009) is a case-based reasoning system designed to play RTS games. D2 implements the *on-line case-based planning* cycle (OLCBP), a high-level framework to develop case-based planning systems that operate on real-time environments (Ontañón et al. 2010). The main focus of D2 is to explore learning from human demonstrations, and the use of adversarial planning techniques. The most important characteristics of D2 are:

In order to play an RTS game, D2 is given the high level task T of winning the game. Then, when the game starts, it retrieves cases looking for plans to execute in the current game state in order to achieve T. The retrieved plans might have to be adapted using transformational adaptation techniques. Moreover, the retrieved plans can be either directly executable, or might decompose T in some sub-tasks, for which further retrieval operations need to be performed. D2 tracks the execution status of the plan in execution, and if any part of the plan fails, it retrieves further cases to replace the failed part.

For a more in-depth description of D2, the reader is referred to (Ontañón et al. 2009; Ontañón et al. 2010). In this paper, we will mainly focus on strategies to automatically acquire cases from expert demonstrations. Next section elaborates on the notion of demonstrations and cases, before we describe the case acquisition strategies.

Demonstrations and Cases

An expert demonstration consists of a list of triples $[\langle t_1, S_1, a_1\rangle, ..., \langle t_1, S_n, a_n\rangle]$, where each triple contains a time stamp t_i game state S_i and a an action a_i. The triples represent the evolution of the game and the actions executed by the expert at different times of the game. The action a_i is the action executed by the expert at time t_i.

Actions in RTS games are durative, have non-deterministic effects, might or might not succeed, and might also have complex interactions between them. For example, the probability of an action to succeed might depend on which actions are executed immediately afterwards. An example of this is the *attack* action: the probability of success when sending one unit to attack depends on whether we issue more *attack* actions afterwards (back-up units).

Actions in D2 have a complex representation to allow the system to reason about their complex interactions. But for the purposes of case acquisition, we only need to focus on the following 3 elements:

- An action specifies a *name* and a list of *Parameters*, e.g.: $Build(U0, Barracks, (23, 18))$. Which represents an action commanding unit $U0$ to build a building of type $Barracks$ at position $(23, 18)$.
- *Preconditions*, which must be satisfied for an action to start execution. For example, the previous action has 5 preconditions in the domain used for our experiments:
  1. entity $U0$ must exist,
  2. position $(23, 18)$ must be free,
  3. a path exists from the current position of $P0$ to $(23, 18)$,
  4. the player must have enough gold to build $Barracks$,
  5. the player must have enough wood to build $Barracks$.
- *Success conditions*. If the success condition of an action succeeds, we know the action completed successfully. Actions might fail, so there is no guarantee of success conditions to be met after an action starts. Also, actions might have additional side-effects that are hard to predict (like peasants opening new paths after chopping trees in a map, that radically change the strategy to follow in the game). In the previous example, the success conditions of the action are that there will be a building of type $Barracks$ at position $(23, 18)$. Although not equivalent, in the rest of this paper, we will consider these success conditions to be the postconditions in traditional planning.

In our framework, in addition to the demonstrations, the learning agent has access to a definition of the task T to perform. This definition is in the form of a reward function that, given a state S, assigns a reward $T(S) \in [0, 1]$. $T(S) = 1$ when the task is fully achieved in state S and $T(S) = 0$ when the task is completely unfinished. Additionally, for complex tasks, the learning agent might have access to a collection of sub-task definitions $T_1, ..., T_k$, defined by their respective reward functions. This set of subtasks can be used by the learning agent to understand and structure the demonstrations provided by the expert. In a way, this set of tasks is equivalent to the list of *tasks* in the HTN planning framework. Many of the learning techniques presented in this paper assume the existence of such task definitions, but we also examine learning techniques that do not assume such task definitions exist.

We will represent a *case* as a triple: $\langle T, S, P\rangle$, where T is a task, S is a game state, and P is a plan. A case represents the fact that the plan P was demonstrated by the expert as the right thing to do to achieve goal T at game state S. Here,

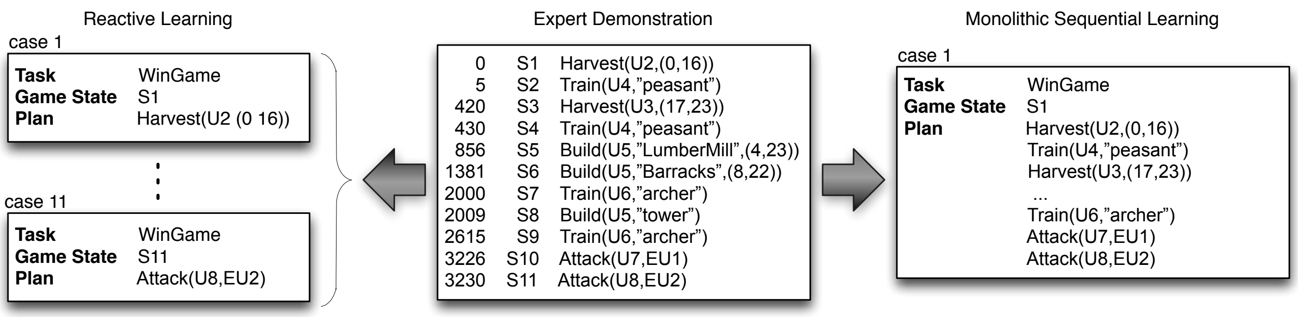


Figure 1: Comparison between a reactive case acquisition strategy and a sequential one.

T is just a symbol that identifies the task. In traditional CBR terms, one can see the combination of task and game state to be the *problem*, and the plan to be the *solution* of the case.

The solution of a case (the plan) can either be a single action or a complex plan. In order to represent complex plans, we will use the formalism of *petri nets*. Petri nets (Murata 1989) are very expressive can can represent plans with conditionals, loops or parallel sequences of actions.

Case Acquisition from Demonstration

In this section we will present 7 different techniques for learning cases from expert demonstrations. The 7 strategies are based on 4 basic ideas: learning sequences of actions rather than isolated actions (sequential learning), hierarchically decomposing the expert demonstration into tasks and subtasks (hierarchical learning), analyzing the preconditions-postconditions of actions (dependency graph analysis) and analyzing the timespan of executed actions (timespan analysis).

Reactive Learning

An expert demonstration is a sequence of triplets that record which actions did the expert execute in different game states at different times. Given an expert trace, the *reactive learning* strategy (RL) learns one case per each entry in the expert trace. Thus, from an expert trace consisting of n entries:

$$[\langle t_1, S_1, a_1\rangle, ..., \langle t_n, S_n, a_n\rangle]$$

it will learn the following n cases:

$$\{\langle WinGame, S_1, a_1\rangle, ..., \langle WinGame, S_n, a_n\rangle\}$$

This case acquisition strategy was introduced by Floyd et al. (Floyd, Esfandiari, and Lam 2008). The left hand side of Figure 1 shows an example of its execution with an expert trace consisting of 11 actions.

Monolithic Sequential Learning

As we will empirically show in the empirical evaluation, one of the issues of the previous strategy is that it's purely reactive and the resulting CBR system has problems properly sequencing actions in a way that their preconditions are satisfied. Notice that this is expected, since the one piece of information that is lost in the reactive learning strategy is precisely the order or the actions.

The *monolithic sequential learning* strategy (SML) takes the completely opposite approach,given an expert demonstration with n actions, it learns a single case:

$$\langle WinGame, S_1, sequence(a_1, ..., a_n)\rangle$$

where $sequence(a_1, ..., a_n)$ represents a sequential plan where all the actions are executed exactly in the same order as the expert executed them. Thus, this strategy learns a single case per demonstration. Figure 1 compares this learning strategy (right) with the reactive learning strategy (left).

Hierarchical Sequential Learning

The monolithic sequential learning strategy can capture the order in which actions were executed by the expert, but it has a major drawback that can be illustrated with the following example. Imagine that at the beginning of a game, a case is retrieved and fails after executing, say, 20 actions. Now, the CBR system retrieves a new case, but the new plan will contain another sequence of actions to play the complete game *from the beginning*, rather than from the current situation.

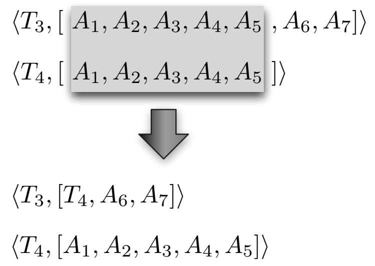
To prevent that from happening, the *hierarchical sequential learning* strategy (SHL) exploits the information available to the system in the form of subtasks $T_1, ..., T_k$ and tried to learn cases with plans to achieve each one of those tasks separately. It also tries to learn which tasks are subtasks of others. In this way, if a plan fails during execution, only a subset of the complete plan to win the game needs to be restarted. For example, the system might divide the trace in 3 parts: build base, build army, and attack. If during execution, the attack plan fails, another case for attack can be retrieved, and the actions to build the base and build the army do not have to be repeated.

In order to achieve that goal, this strategy uses the idea of a *task matrix* (Ontañón et al. 2009). A task matrix M for a given demonstration D, as illustrated in Table 1, contains one row per entry in the expert demonstration, and one column per task defined in the domain at hand. For example, in the RTS game used in our evaluation, we have 13 tasks defined (build a town hall, build a barracks, build a tower, train a footman, etc.), thus the task matrix has 13 columns. The matrix is binary and $M_{i,j} = 1$ if the task T_j is satisfied in the game state S_i, otherwise $M_{i,j} = 0$.

From each sequence of zeroes in a column that ends in a one, a plan can be generated. For example, five plans could

Demonstration	T_1	T_2	T_3	T_4	T_5
$\langle t_1, S_1, a_1 \rangle$	0	0	0	0	0
$\langle t_2, S_2, a_2 \rangle$	0	0	0	0	0
$\langle t_3, S_3, a_3 \rangle$	0	0	0	0	0
$\langle t_4, S_4, a_4 \rangle$	0	0	0	0	0
$\langle t_5, S_5, a_5 \rangle$	0	0	0	0	0
$\langle t_6, S_6, a_6 \rangle$	0	0	0	1	0
$\langle t_7, S_7, a_7 \rangle$	0	0	1	1	0
$\langle t_8, S_8, a_8 \rangle$	0	1	1	1	0
$\langle t_9, S_9, a_9 \rangle$	0	1	1	1	1
$\langle t_{10}, S_{10}, a_{10} \rangle$	0	1	1	1	1
$\langle t_{11}, S_{11}, a_{11} \rangle$	0	1	1	1	1
$\langle t_{12}, S_{12}, \emptyset \rangle$	1	1	1	1	1

Table 1: Task matrix for a set of five tasks $\{T_1, T_2, T_3, T_4, T_5\}$ and for a small trace consisting of only 12 entries (corresponding to the actions shown in Figure 3).



$$\langle T_3, [\boxed{A_1, A_2, A_3, A_4, A_5}, A_6, A_7] \rangle$$

$$\langle T_4, [\boxed{A_1, A_2, A_3, A_4, A_5}] \rangle$$

$$\langle T_3, [T_4, A_6, A_7] \rangle$$

$$\langle T_4, [A_1, A_2, A_3, A_4, A_5] \rangle$$

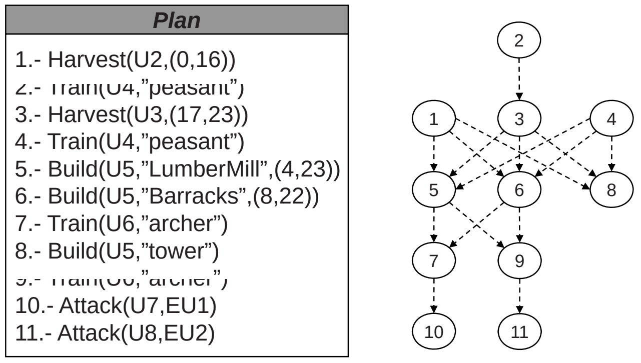
Figure 2: Illustration of the process of replacing a sequence of actions by a task to generate hierarchical plans.

be generated from the task matrix in Table 1. One for T_1 with actions $a_1, ..., a_{12}$, one for T_2 with actions $a_1, ..., a_8$, one for T_3 with actions $a_1, ..., a_7$, one for T_4 with actions $a_l, ..., a_6$, and one for T_5 with actions $a_l, ..., a_9$. Notice that the intuition behind this process is just to look at sequences of actions that happened before a particular task was satisfied, since those actions are a plan to complete that task.

Finally, if there are any two raw plans $p = \langle T_1, [a_1, ..., a_j] \rangle$ and $q = \langle T_2, [a'_l, ..., a'_l] \rangle$ such that the sequence of actions in p is a subset of the sequence of actions in q, such the sequence of actions in q is replaced by a subtask element T_1. The intuition is that the sequence of actions that was replaced are assumed to aim at achieving T_1. Figure 2 illustrates this idea. Notice that the order in which we attempt to substitute actions by subtasks in plans will result in different final plans. Finally, from each one of the resulting *raw plans* a case can be learned.

Dependency graph Learning

The two previous case acquisition strategies assume that the order in which the expert executed the actions is a total order that captures the precondition-postcondition dependencies between actions. However, this is not necessarily the case, and if the CBR system knows the proper dependencies, plans can be better executed and adapted.



Plan

1.- Harvest(U2,(0,16))
2.- Train(U4,"peasant")
3.- Harvest(U3,(17,23))
4.- Train(U4,"peasant")
5.- Build(U5,"LumberMill",(4,23))
6.- Build(U5,"Barracks",(8,22))
7.- Train(U6,"archer")
8.- Build(U5,"tower")
9.- Train(U6,"archer")
10.- Attack(U7,EU1)
11.- Attack(U8,EU2)

Figure 3: An example dependency graph constructed from a plan consisting of 11 actions in an RTS game.

A *dependency graph* (Sugandh, Ontañón, and Ram 2008) is a directed graph where each node represents one action in the plan, and edges represent dependencies among actions. A dependency graph is easily constructed by checking each pair of actions a_i and a_j, such that a_i was executed earlier in the plan, and checking if one of the postconditions of a_i matches any precondition of a_j, and there is no action a_k that has to happen after a_i that also matches with that precondition, then an edge is drawn from a_i to a_j in the dependency graph, annotating it with which is the pair of postcondition/precondition that matched. Figure 3 shows an example dependency graph (where the annotations in the edges have been omitted for clarity). The plan shown in the figure shows how each action is dependent on each other, and it is useful to determine which actions contribute to the achievement of particular goals.

The *dependency graph learning* strategy (DGML) is equivalent to the monolithic learning strategy, but instead of learning a sequential plan, the resulting plan has a partial order of its actions, based on their dependencies.

Dependency graph Hierarchical Learning

The *dependency graph hierarchical learning* strategy (DGHL) is equivalent to the hierarchical sequential learning strategy, but instead of learning sequential plans, the resulting plans have a partial order of its actions, based on their dependencies. This is the learning strategy presented in (Ontañón et al. 2009).

Timespan Learning

The idea of exploiting precondition-postcondition matching to create a dependency graph can greatly help in plan execution and in plan adaptation. However, since actions are not instantaneous in RTS games, but are durative, the simple task-matrix analysis presented in the previous sections may generate inexistent dependencies. For example if an action a started before another action b, but a didn't finish until after b was started, b can never have a dependency with a.

The idea of *timespan analysis* is to detect when did actions complete their execution in the expert demonstrations.

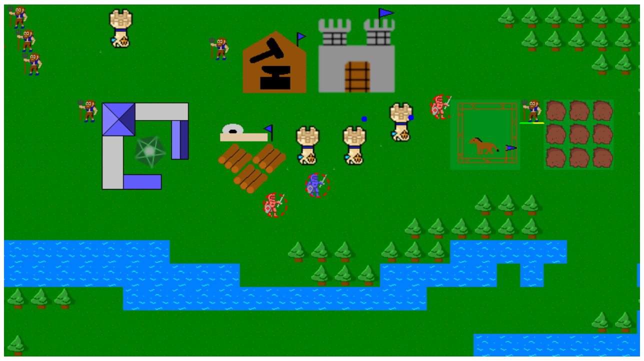


Figure 4: A screenshot of the S3 game.

To do so, it suffices with testing when the postconditions of the different actions got satisfied during the demonstration.

The *timespan learning* strategy (TSML) is identical to the DGML strategy, but after generating the dependency graphs, timespan analysis is used to remove those dependencies that are inconsistent with action duration.

Timespan Hierarchical Learning

Finally, the *timespan hierarchical learning* strategy (TSHL) is identical to the dependency graph hierarchical learning strategy, but timespan analysis is used to remove those dependencies that are inconsistent with action duration.

Experimental Evaluation

In order to evaluate the different case acquisition strategies presented in this paper, we used a strategy game called *S3*, which is a simplified version of the Warcraft RTS game, still capturing the complexity of the domain, but eliminating all of those aspects unrelated to AI research (e.g. animations, user interfaces, etc.). Figure 4 shows a screenshot of S3, where the red player is attacking the base of the blue player with two knights. In S3, players need to collect wood (by chopping trees) and gold (by mining gold mines), in order to construct buildings and attack units to defeat the opponent. To achieve those goals, there are 8 different action operators they can use, each of them with 2 to 5 parameters. We created a collection of 5 different maps with different characteristics. Two maps contained several islands connected by small bridges, two maps contained walls of trees that players had to cut through in order to reach the enemy, and one map consisted on a labyrinth of trees with each player on one side, and two large gold mines in the center. Maps in S3 are represented by a two-dimensional grid, where in each cell we can have grass, water or trees. The maps used in our evaluation had size 64x32 cells. S3 contains 4 built-in AIs, implementing 4 different strategies: footman-rush, archers-rush, catapults-rush and knights-rush.

We created a fifth AI, that we call the *expert*, implementing a defensive knights rush strategy (where first, a formation of defensive towers are created, and then knights are sent to attack the enemy) and generated expert demonstrations by making this strategy play against all of the other AIs

Strategy	Wins	Ties	Loses	Destroyed	Lost
Expert	8	6	11	2.53	3.37
RL	1	3	21	2.24	2.02
SML	2	2	21	2.23	3.29
SHL	2	2	21	0.94	0.96
DGML	4	4	17	4.40	4.08
DGHL	2	3	20	1.78	3.45
TSML	5	5	15	7.22	2.8
TSHL	3	5	16	2.55	3.02

Table 2: Experimental results of the 7 strategies evaluated against the built-in AIs in 5 different maps.

in all of the maps, and also against itself. Then, we randomly selected 5 of such demonstrations where the expert had won the game, constituting the training set. In each demonstration the expert executed an average of 121.6 actions (minimum 15, and maximum 496). In our experiments, the expert created an average of 40.4 units per game (having a maximum of 33 at once).

Seven different instances of the D2 system were created using each of the case acquisition strategies, and the resulting systems played against the 4 built-in AIs and the expert in all the maps. Thus, each instance of D2 played 25 games. If a game reached 100000 cycles, it was considered a tie. Table 2 show the number of wins, ties and loses as well as the average number of units that each of the instances of D2 destroyed (D2 killing units from the other AI) or lost (D2's units being killed by the other AI).

Table 2 shows that none of the methods was able to perform as well as the expert, although some of the strategies, like TSML, got close. The results show that the reactive learning (RL), sequential monolithic learning (SML) and hierarchical monolithic learning (SHL) obtained the worst results (21 defeats). RL's problem was that it had troubles sequencing actions in order to satisfy their preconditions. As reported in (Floyd, Esfandiari, and Lam 2008), for this strategy to work, some times we require large amounts of cases. From the 5 provided traces, about 1200 cases could be generated by RL, which probably was not enough, given the complexity of the domain. The problem of SML is that it did not know how to recover from plan failures, and had to restart execution from the beginning. Finally, SHL has problems because the simple technique used to infer hierarchical structure creates odd substitutions caused by ignoring the duration or the dependencies of the actions.

Table 2 also shows that by incorporating dependency graph analysis and timespan analysis, the resulting monolithic strategies, DGML and TSML, obtain much better results than the base sequential one (SML). This shows that having a deeper understanding of the relations among the expert actions is a clear help during the adaptation and execution phase. For example, TSML, only gets defeated 15 times, compared to 21 times of SML. We observe similar improvements with the hierarchical learning techniques, where DGHL and TSHL obtain better results than SHL.

Overall, the technique that obtains better results is TSML. Even if being a monolithic learning approach (having to restart a plan from scratch if it fails), since the case acquisition has learned a very good dependency graph between the actions, plans tend to execute successfully. The results also show that the hierarchical learning techniques tend to perform worse than their monolithic counterparts. This was a surprising result, given that previous work has shown hierarchical case acquisition techniques obtain good results in RTS games (Ontañón et al. 2010). The difference is that in (Ontañón et al. 2010) traces were spliced hierarchically by the expert himself, whereas here we use the task-matrix to automatically detect the hierarchical structure of a demonstration, and this method tends to obtain bad hierarchical decompositions. Finding a better way to learn hierarchical plans from expert demonstrations is part of our future work.

Conclusions and Future Work

This paper has presented a comparative evaluation of seven different case acquisition techniques for CBR systems that use learning from demonstration. Specifically, we have focused on the domain of real-time strategy games (RTS) due to their complexity. We incorporated all those techniques into the D2 CBR system to evaluate their performance.

We showed that the reactive learning strategy (RL) cannot produce coherent behavior because it does not reason about the order in which actions must be sequenced. This strategy requires a larger collection of cases in order to produce meaningful behavior. For that reason, strategies that extract as much information as possible from the expert demonstration (like action dependencies) obtain better results when learning from fewer traces (5 in our experiments). The other strategies can reason about action sequencing and perform better. However, they prevent the system from reacting to unexpected changes in the game: once a plan has started executing, the system will not change it unless it fails. An interesting new idea is that of *temporal backtracking* (Floyd and Esfandiari 2011), where a collection of reactive cases is learned, but each case has a *link* to the action that was executed previously (which is stored as a different case). At retrieval time, this information can be used to obtain some sort of reasoning about which actions to execute after other actions, while preserving the reactiveness of the system. However, this technique requires a different case retrieval mechanism, and thus, was out of the scope of this paper.

As part of our future work we would like to explore three main lines of research. First, how can we incorporate reactiveness into sequential case acquisition strategies by learning plans with conditionals. Second, how can we incorporate reasoning about sequences into reactive case acquisition strategies (temporal backtracking seems to be a promising like of work). And third, we would like to experiment with new techniques to obtain the hierarchical structure of an expert demonstration that improve the results obtained by the task-matrix. Additionally, evaluation with other RTS domains, such as Starcraft, is also part of our future work.

References

Aha, D.; Molineaux, M.; and Ponsen, M. 2005. Learning to win: Case-based plan selection in a real-time strategy game. In *ICCBR'2005*, number 3620 in LNCS, 5–20. Springer-Verlag.

Argall, B. D.; Chernova, S.; Veloso, M.; and Browning, B. 2009. A survey of robot learning from demonstration. *Robot. Auton. Syst.* 57:469–483.

Buro, M. 2003. Real-time strategy games: A new AI research challenge. In *IJCAI'2003*, 1534–1535. Morgan Kaufmann.

Chung, M.; Buro, M.; and Schaeffer, J. 2005. Monte carlo planning in rts games. In *IEEE Symposium on Computational Intelligence and Games (CIG)*.

Floyd, M. W., and Esfandiari, B. 2011. Learning state-based behaviour using temporally related cases. In *Proceedings of the Sixteenth UK Workshop on Case-Based Reasoning*.

Floyd, M. W.; Esfandiari, B.; and Lam, K. 2008. A case-based reasoning approach to imitating robocup players. In *In: Proceedings of FLAIRS-2008.*

Hogg, C. M.; Muñoz-Avila, H.; and Kuter, U. 2008. Htn-maker: Learning htns with minimal additional knowledge engineering required. In *AAAI-2008*, 950–956.

Könik, T., and Laird, J. E. 2006. Learning goal hierarchies from structured observations and expert annotations. *Mach. Learn.* 64(1-3):263–287.

Marthi, B.; Russell, S.; Latham, D.; and Guestrin, C. 2005. Concurrent hierarchical reinforcement learning. In *International Joint Conference of Artificial Intelligence, IJCAI*, 779–785.

McCoy, J., and Mateas, M. 2008. An integrated agent for playing real-time strategy games. In *AAAI 2008*, 1313–1318.

Murata, T. 1989. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* 77(4):541–580.

Ontañón, S.; Bonnette, K.; Mahindrakar, P.; Gmez-martn, M. A.; Long, K.; Radhakrishnan, J.; Shah, R.; and Ram, A. 2009. Learning from human demonstrations for real-time case-based planning. In *IJCAI 2009 Workshop on Learning Structural Knowledge From Observations (STRUCK).*

Ontañón, S.; Mishra, K.; Sugandh, N.; and Ram, A. 2010. On-line case-based planning. *Computational Intelligence* 26(1):84–119.

Ontañón, S.; Montaña, J. L.; and Gonzalez, A. J. 2011. Towards a unified framework for learning from observation. In *IJCAI 2011 Workshop on Agents Learning Interactively from Human Teachers (ALIHT).*

Schaal, S. 1996. Learning from demonstration. 1040–1046.

Sharma, M.; Homes, M.; Santamaria, J.; Irani, A.; Isbell, C.; and Ram, A. 2007. Transfer learning in real time strategy games using hybrid CBR/RL. In *IJCAI'2007*, 1041–1046. Morgan Kaufmann.

Sugandh, N.; Ontañón, S.; and Ram, A. 2008. On-line case-based plan adaptation for real-time strategy games. In *AAAI 2008*, 702–707.

Weber, B. G., and Mateas, M. 2009. A data mining approach to strategy prediction. In *IEEE Symposium on Computational Intelligence and Games (CIG).*