International Conference on Case-based Reasoning (ICCBR 2007)

Case-Based Planning and Execution for Real-Time Strategy Games

Santiago Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram

CCL, Cognitive Computing Lab
Georgia Institute of Technology
Atlanta, GA 303322/0280
{santi,kinshuk,nsugandh,ashwin}@cc.gatech.edu

Abstract. Artificial Intelligence techniques have been successfully applied to several computer games. However in some kinds of computer games, like real-time strategy (RTS) games, traditional artificial intelligence techniques fail to play at a human level because of the vast search spaces that they entail. In this paper we present a real-time case based planning and execution approach designed to deal with RTS games. We propose to extract behavioral knowledge from expert demonstrations in form of individual cases. This knowledge can be reused via a case based behavior generator that proposes behaviors to achieve the specific open goals in the current plan. Specifically, we applied our technique to the WARGUS domain with promising results.

1 Introduction

Artificial Intelligence (AI) techniques have been successfully applied to several computer games. However, in the vast majority of computer games traditional AI techniques fail to play at a human level because of the characteristics of the game. Most current commercial computer games have vast search spaces in which the AI has to make decisions in real-time, thus rendering traditional search based techniques inapplicable. For that reason, game developers need to spend a big effort in hand coding specific strategies that play at a reasonable level for each new game. One of the long term goals of our research is to develop artificial intelligence techniques that can be directly applied to such domains, alleviating the effort required by game developers to include advanced AI in their games.

Specifically, we are interested in real-time strategy (RTS) games, that have been shown to have huge decision spaces that cannot be dealt with search based AI techniques [2, 3]. In this paper we will present a case-based planning architecture that integrates planning and execution and is capable of dealing with both the vast decision spaces and the real-time component of RTS games. Moreover, applying case-based planning to RTS games requires a set of cases with which to construct plans. To deal with this issue, we propose to extract behavioral knowledge from expert demonstrations (i.e. an expert plays the game and our system observes), and store it in the form of cases. Then, at performance time,

the system will retrieve the most adequate behaviors observed from the expert and will adapt them to the situation at hand.

As we said before, one of the main goals of our research is to create AI techniques that can be used by game manufacturers to reduce the effort required to develop the AI component of their games. Developing the AI behavior for an automated agent that plays a RTS is not an easy task, and requires a large coding and debugging effort. Using the architecture presented in this paper the game developers will be able to specify the AI behavior just by demonstration; i.e. instead of having to code the behavior using a programming language, the behavior can be specified simply by *demonstrating* it to the system. If the system shows an incorrect behavior in any particular situation, instead of having to find the bug in the program and fix it, the game developers can simply demonstrate the correct action in the particular situation. The system will then incorporate that information in its case base and will behave better in the future.

Another contribution of the work presented in this paper is on presenting an integrated architecture for case-based planning and execution. In our architecture, plan retrieval, composition, adaptation, and execution are interleaved. The planner keeps track of all the open goals in the current plan (initially, the system starts with the goal of winning the game), and for each open goal, the system retrieves the most adequate behavior in the case base depending on the current game state. This behavior is then added into the current plan. When a particular behavior has to be executed, it is adapted to match the current game state and then it is executed. Moreover, each individual action or sub-plan inside the plan is constantly monitored for success or failure. When a failure occurs, the system attempts to retrieve a better behavior from the case base. This interleaved process of case based planning and execution allows the system to reuse the behaviors extracted from the expert and apply them to play the game.

The rest of the paper is organized as follows. Section 2 presents a summary of related work. Then, Section 3 introduces the proposed architecture and its main modules. After that, Section 4 briefly explains the behavior representation language used in our architecture. Section 5 explains the case extraction process. Then sections 6 and 7 present the planning module and the case based reasoning module respectively. Section 8 summarizes our experiments. Finally, the paper finishes with the conclusions section.

2 Related Work

Concerning the application of case-based reasoning techniques to computer games, Aha et al. [2] developed a case-based plan selection technique that learns how to select an appropriate strategy for each particular situation in the game of WARGUS. In their work, they have a library of previously encoded strategies, and the system learns which one of them is better for each game phase. In addition, they perform an interesting analysis on the complexity of real-time strategy games (focusing on WARGUS in particular). Another application of case based reasoning to real-time strategy games is that of Sharma et al. [15], where they

present a hybrid case based reinforcement learning approach able to learn which are the best actions to apply in each situation (from a set of high level actions). The main difference between their work and ours is that they learn a case selection policy, while our system constructs plans from the individual cases it has in the case base. Moreover, our architecture automatically extracts the plans from observing a human rather than having them coded in advance.

Ponsen et al [14] developed a hybrid evolutionary and reinforcement learning strategy for automatically generating strategies for the game of WARGUS. In their framework, they construct a set of rules using an evolutionary approach (each rule determines what to do in a set of particular situations). Then they use a reinforcement learning technique called *dynamic scripting* to select a subset of these evolved rules that achieve a good performance when playing the game. There are several differences between their approach and ours. First, they focus on automatically generating strategies while we focus on acquiring them from an expert. Moreover, each of their individual rules could be compared to one of our behaviors, but the difference is that their strategies are combined in a pure reactive way, while our strategies are combined using a planning approach. For our planner to achieve that, we require each individual behavior to be annotated with the goal it pursues.

Hoang et al. [9] propose to use a hierarchical plan representation to encode strategic game AI. In their work, they use HTN planning (inside the framework of Goal-Oriented Action Planning [13]). Further, in [11] Muñoz and Aha propose a way to use case based planning to the same HTN framework to deal with strategy games. Moreover, they point out that case based reasoning provides a way to generate explanations on the decisions (i.e. plans) generated by the system. The HTN framework is very related to the work presented in this paper, where we use the task-method decomposition to represent plans. Moreover, in their work they focus on the planing aspects of the problem while in this paper we focus on the learning aspects of the problem, i.e. how to learn from expert demonstrations.

The work presented in this paper is strongly related to existing work in case-based planning [8]. Case Based Planning work is based on the idea of planning by remembering instead of planning from scratch. Thus, a case based planner retains the plans it generates to reuse them in the future, uses planning failures as opportunities for learning, and tries to retrieve plans in the past that satisfy as many of the current goals as possible. Specifically, our work focuses on an integrated planning and execution architecture, in which there has been little work in the case based planning community. A sample of such work is that of Freßmann et al. [6], where they combine CBR with multi-agent systems to automate the configuration and execution of workflows that have to be executed by multiple agents.

Integrating planning and execution has been studied in the search based planning community. For example, CPEF [12] is a framework for continuous planning and execution. CPEF shares a common assumption with our work, namely that plans are dynamic artifacts that must evolve with the changing

Fig. 1. A screenshot of the WARGUS game.

environment in which they are executing changes. However, the main difference is that in our approach we are interested in case based planning processes that are able to deal with the huge complexity of our application domain.

3 Case-Based Planning in WARGUS

WARGUS (Figure 1) is a real-time strategy game where each player's goal is to remain alive after destroying the rest of the players. Each player has a series of troops and buildings and gathers resources (gold, wood and oil) in order to produce more troops and buildings. Buildings are required to produce more advanced troops, and troops are required to attack the enemy. In addition, players can also build defensive buildings such as walls and towers. Therefore, WARGUS involves complex reasoning to determine where, when and which buildings and troops to build. For example, the map shown in Figure 1 is a 2-player version of the classical map "Nowhere to run nowhere to hide", with a wall of trees that separates the players. This maps leads to complex strategic reasoning, such as building long range units (such as catapults or ballistas) to attack the other player before the wall of trees has been destroyed, or tunneling early in the game through the wall of trees trying to catch the enemy by surprise.

Traditionally, games such as WARGUS use handcrafted behaviors for the built-in AI. Creating such behaviors requires a lot of effort, and even after that, the result is that the built-in AI is static and easy to beat (since humans can easily find holes in the computer strategy). The goal of the work presented in this paper is to ease the task of the game developers to create behaviors for these games, and to make them more adaptive. Our approach involves learning behaviors from expert demonstrations to reduce the effort of coding the behaviors, and use the learned behaviors inside a case-based planning system to reuse them for new situations. Figure 2 shows an overview of our case-based planning approach. Basically, we divide the process in two main stages:

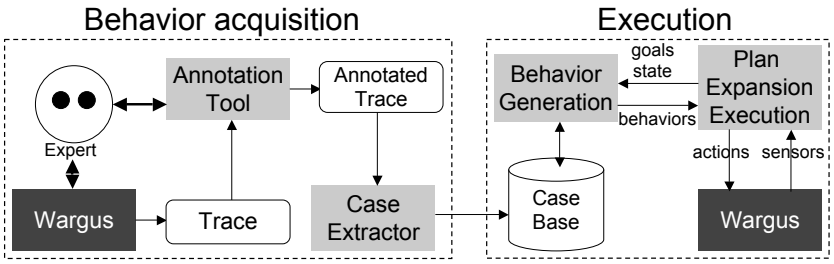

Fig. 2. Overview of the proposed case-based planning approach.

- *Behavior acquisition*: During this first stage, an expert plays a game of WARGUS and the trace of that game is stored. Then, the expert annotates the trace explaining the goals he was pursuing with the actions he took while playing. Using those annotations, a set of behaviors are extracted from the trace and stored as a set of cases. Each case is a triple: situation/goal/behavior, representing that the expert used a particular behavior to achieve a certain goal in a particular situation.
- *Execution*: The execution engine consists of two main modules, a real-time plan expansion and execution (RTEE) module and a behavior generation (BG) module. The RTEE module maintains an execution tree of the current active goals and subgoals and which behaviors are being executed to achieve each of the goals. Each time there is an open goal, the RTEE queries the BG module to generate a behavior to solve it. The BG then retrieves the most appropriate behavior from its case base, and sends it to the RTEE. Finally, when the RTEE is about to start executing a behavior, it is sent back to the BG module for adaptation. Notice that this *delayed adaptation* is a key feature different from traditional CBR required for real-time domains where the environment continuously changes.

In the following sections we will present each of the individual components of our architecture.

4 A Behavior Reasoning Language

In this section we will present the Behavior Reasoning Language used in our approach, designed to allow a system to learn behaviors, represent them, and to reason about the behaviors and their intended and actual effects. Our language takes ideas from the STRIPS [5] planning language, and from the ABL [10] behavior language, and further develops them to allow advanced reasoning and learning capabilities over the behavior language.

The basic constituent piece is the *behavior*. A behavior has two main parts: a *declarative* part and a *procedural* part. The declarative part has the purpose of providing information to the system about the intended use of the behavior,

and the procedural part contains the executable behavior itself. The declarative part of a behavior consists of three parts:

- A *goal*, that is a representation of the intended goal of the behavior. For every domain, an ontology of possible goals has to be defined. For instance, a behavior might have the goal of "having a tower".
- A set of *preconditions* that must be satisfied before the behavior can be executed. For instance, a behavior can have as preconditions that a particular peasant exists and that a desired location is empty.
- A set of *alive conditions* that represent the conditions that must be satisfied during the execution of the behavior for it to have chances of success. If at some moment during the execution, the alive conditions are not met, the behavior can be stopped, since it will not achieve its intended goal. For instance, the peasant in charge of building a building must remain alive; if he is killed, the building will not be built.

Notice that unlike classical planning approaches, postconditions cannot be specified for behaviors, since a behavior is not guaranteed to succeed. Thus, we can only specify what goal a behavior pursues.

The procedural part of a behavior consists of executable code that can contain the following constructs: *sequence*, *parallel*, *action*, and *subgoal*, where an *action* represents the execution of a basic action in the domain of application (a set of basic actions must be defined for each domain), and a *subgoal* means that the execution engine must find another behavior that has to be executed to satisfy that particular subgoal. Specifically, three things need to be defined for using our language in a particular domain:

- A set of *basic actions* that can be used in the domain. For instance, in WARGUS we define actions such as*move*, *attack*, or *build*.
- A set of *sensors*, that are used in the behaviors to obtain information about the current state of the world. For instance, in WARGUS we might define sensors such as *numberOfTroops*, or *unitExists*. A sensor might return any of the standard basic data types, such as boolean or integer.
- A set of *goals*. Goals can be structured in a specialization hierarchy in order to specify the relations among them.

A goal might have parameters, and for each goal a function *generateSuccessTest* must be defined, that is able to generate a condition that is satisfied only when the goal is achieved. For instance, *HaveUnits(TOWER)* is a valid goal in our gaming domain and it should generate the condition *UnitExists(TOWER)*. Such condition is called the *success test* of the goal. Therefore, the goal definition can be used by the system to reason about the intended result of a behavior, while the success test is used by the execution engine to verify whether a particular behavior succeeds at run time.

Summarizing, our behavior language is strongly inspired by ABL, but expands it with declarative annotations (expanding the representation of goals and defining alive and success conditions) to allow reasoning.

Cycle	*Player*	*Action*	*Annotation*
8	1	Build(2,"pig-farm",26,20)	-
137	0	Build(5,"farm",4,22)	SetupResourceInfrastructure(0,5,2) WinWargus(0)
638	1	Train(4,"peon")	-
638	1	Build(2,"troll-lumber-mill",22,20)	-
798	0	Train(3,"peasant")	SetupResourceInfrastructure(0,5,2) WinWargus(0)
878	1	Train(4,"peon")	-
878	1	Resource(10,5)	-
897	0	Resource(5,0)	SetupResourceInfrastructure(0,5,2) WinWargus(0)
...	...	...	...

Table 1. Snippet of a real trace generated after playing WARGUS.

5 Behavior Acquisition in WARGUS

As Figure 2 shows, the first stage of our case-based planning architecture consists of acquiring a set of behaviors from an expert demonstration. Let us present this stage in more detail.

One of the main goals of this work is to allow a system to learn a behavior by simply observing a human, in opposition to having a human encoding the behavior in some form of programming language. To achieve that goal, the first step in the process must be for the expert to provide the demonstration to the system. In our particular application domain, WARGUS, an expert simply plays a game of WARGUS (against the built-in AI, or against any other opponent). As a result of that game, we obtain a game trace, consisting of the set of actions executed during the game. Table 1 shows a snippet of a real trace from playing a game of WARGUS. As the table shows, each trace entry contains the particular cycle in which an action was executed, which player executed the action, and the action itself. For instance, the first action in the game was executed at cycle 8, where player 1 made his unit number 2 build a "pig-farm" at the (26,20) coordinates.

As Figure 2 shows, the next step is to annotate the trace. For this process, the expert uses a simple annotation tool that allows him to specify which goals was he pursuing for each particular action. To use such an annotation tool, a set of available goals has to be defined for the WARGUS domain.

In our approach, a *goal* $g = name(p_1, ..., p_n)$ consists of a goal name and a set of parameters. For instance, in WARGUS, these are some of the goal types we have defined:

- $WinWargus(player)$: representing that the action had the intention of making the player *player* win the game.
- $KillUnit(unit)$: representing that the action had the intention of killing the unit *unit*.

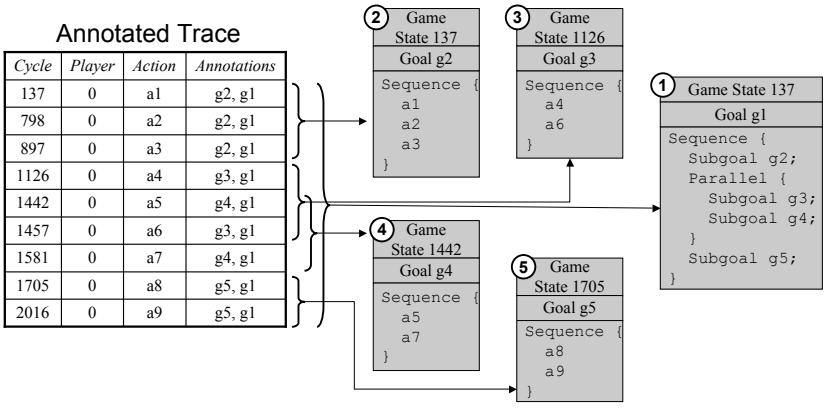

Fig. 3. Extraction of cases from the annotated trace.

– $SetupResourceInfrastructure(player, peasants, farms)$: indicates that the expert wanted to create a good resource infrastructure for player *player*, that at least included *peasants* number of peasants and *farms* number of farms.

The fourth column of Table 1 shows the annotations that the expert specified for his actions. Since the snippet shown corresponds to the beginning of the game, the expert specified that he was trying to create a resource infrastructure and, of course, he was trying to win the game.

Finally, as Figure 2 shows, the annotated trace is processed by the *case extractor* module, that encodes the strategy of the expert in this particular trace in a series of cases. Traditionally, in the CBR literature cases consist of a problem/solution pair; in our system we extended that representation due to the complexity of the domain of application. Specifically, a case in our system is defined as a triple consisting of a game state, a goal and a behavior. See Section 7 for a more detailed explanation of our case formalism.

In order to extract cases, the annotated trace is analyzed to determine the temporal relations among the individual goals appearing in the trace. For instance, if we look at the sample annotated trace in Figure 3, we can see that the goal *g2* was attempted *before* the goal *g3*, and that the goal *g3* was attempted *in parallel* with the goal *g4*. The kind of analysis required is a simplified version of the temporal reasoning framework presented by Allen [7], where the 13 basic different temporal relations among events were identified. In our framework, we are only interested in knowing if two goals are pursued in sequence, in parallel, or if one is a subgoal of the other. We assume that if the temporal relation between a particular goal g and another goal g' is that g happens *during* g', then g is a subgoal of g'. For instance, in Figure 3, *g2*, *g3*, *g4*, and *g5* happen *during g1*; thus they are considered subgoals of *g1*.

From temporal analysis, procedural descriptions of the behavior of the expert can be extracted. For instance, from the relations among all the goals in Figure 3,

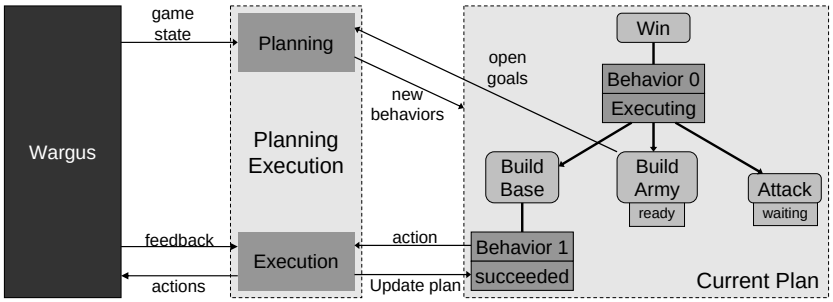

Fig. 4. Interleaved plan expansion and execution.

case number 1 (shown in the figure) can be extracted, specifying that to achieve goal *g1* in the particular game state in which the game was at cycle 137, the expert first tried to achieve goal *g2*, then attempted *g3* and *g4* in parallel, and after that *g5* was pursued. Then, for each one of the subgoals a similar analysis is performed, leading to four more cases. For example, case 3 states that to achieve goal *g2* in that particular game state, basic actions *a4* and *a6* should be executed sequentially.

6 Real-Time Plan Expansion and Execution

During execution time, our system will use the set of cases collected from expert traces to play a game of WARGUS. In particular two modules are involved in execution: a real-time plan expansion and execution module (RTEE) and a behavior generation module (BG). Both modules collaborate to maintain a current *partial plan tree* that the system is executing.

A *partial plan tree* (that we will refer to as simply the "plan") in our framework is represented as a tree consisting of two types of nodes: *goals* and *behaviors* (following the same idea of the task/method decomposition [4]). Initially, the plan consists of a single goal: "win the game". Then, the RTEE asks the BG module to generate a behavior for that goal. That behavior might have several subgoals, for which the RTEE will again ask the BG module to generate behaviors, and so on. For instance, on the right hand side of Figure 4 we can see a sample plan, where the top goal is to "win". The behavior assigned to the "win" goal has three subgoals, namely "build base", "build army" and "attack". The "build base" goal has already a behavior assigned that has no subgoals, and the rest of subgoals still don't have an assigned behavior. When a goal still doesn't have an assigned behavior, we say that the goal is *open*.

Additionally, each behavior in the plan has an associated state. The state of a behavior can be: *pending*, *executing*, *succeeded* or *failed*. A behavior is pending when it still has not started execution, and its status is set to failed or succeeded after its execution ends, depending on whether it has satisfied its goal or not.

A goal that has a behavior assigned and where the behavior has failed is also considered to be open (since a new behavior has to be found for this goal).

Open goals can be either *ready* or *waiting*. An open goal is ready when all the behaviors that had to be executed before this goal have succeeded, otherwise, it is waiting. For instance, in Figure 4, "behavior 0" is a sequential behavior and therefore the goal "build army" is ready since the "build base" goal has already succeeded and thus "build army" can be started. However, the goal "attack" is waiting, since "attack" has to be executed after "build army" succeeds.

The RTEE is divided into two separate modules, that operate in parallel to update the current plan: the *plan expansion* module and the *plan execution* module. The plan expansion module is constantly querying the current plan to see if there is any ready open goal. When this happens, the open goal is sent to the BG module to generate a behavior for it. Then, that behavior is inserted in the current plan, and it is marked as pending.

The plan execution module has two main functionalities: a) check for basic actions that can be sent directly to the game engine, b) check the status of plans that are in execution:

- For each pending behavior, the execution module evaluates the preconditions, and as soon as they are met, the behavior starts its execution.
- If any of the execution behaviors have any basic actions, the execution module sends those actions to WARGUS to be executed.
- Whenever a basic action succeeds or fails, the execution module updates the status of the behavior that contained it. When a basic action succeeds, the executing behavior can continue to the next step. When a basic action fails, the behavior is marked as failed, and thus its corresponding goal is open again (thus, the system will have to find another plan for that goal).
- The execution module periodically evaluates the alive conditions and success conditions of each behavior. If the alive conditions of an executing behavior are not satisfied, the behavior is marked as failed, and its goal is open again. If the success conditions of a behavior are satisfied, the behavior is marked as succeeded.
- Finally, if a behavior is about to be executed and the current game state has changed since the time the BG module generated it, the behavior is handed back to the BG and it will pass again through the *adaptation* phase (see Section 7) to make sure that the plan is adequate for the current game state.

7 Behavior Generation

The goal of the BG module is to generate behaviors for specific goals in specific scenarios. Therefore, the input to the BG module is a particular scenario (i.e. the current game state in WARGUS) and a particular goal that has to be achieved (e.g. "Destroy The Enemy's Cannon Tower"). To achieve that task, the BG system uses two separate processes: *case retrieval* and *case adaptation* (that correspond to the first two processes of the 4R CBR model [1]).

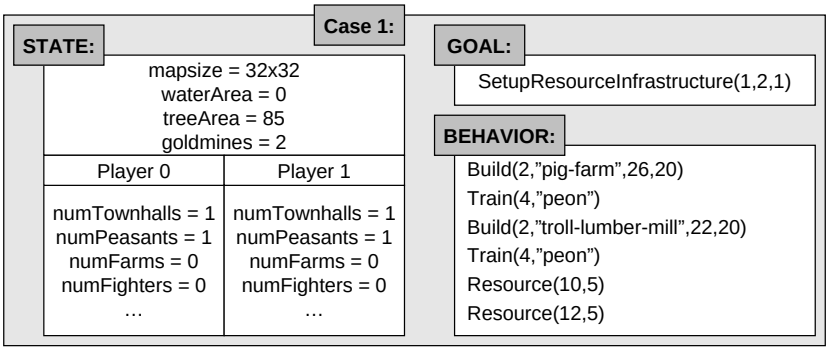

Fig. 5. Example of a case extracted from an expert trace for the WARGUS game.

Notice that to solve a complex planning task, several subproblems have to be solved. For instance, in our domain, the system has to solve problems such as how to build a proper base, how to gather the necessary resources, or how to destroy each of the units of the enemy. All those individual problems are different in nature, and in our case base we might have several cases that contain different behaviors to solve each one of these problems under different circumstances. Therefore, in our system we will have an *heterogeneous* case base. To deal with this issue, we propose to include in each case the particular *goal* that it tries to solve. Therefore we represent cases as triples: $c = \langle S, G, B \rangle$, where S is a particular game state, G is a goal, and B is a behavior; representing that $c.B$ is a good behavior to apply when we want to pursue goal $c.G$ in a game state similar to $c.S$.

Figure 5 shows an example of a case, where we can see the three elements: a game description, that contains some general features about the map and some information about each of the players in the game; a particular goal (in this case, building the resource infrastructure of player "1"); and finally a behavior to achieve the specified goal in the given map. In particular, we have used a game state definition composed of 35 features that try to represent each aspect of the WARGUS game. Twelve of them represent the number of troops (number of fighters, number of peasants, and so on), four of them represent the resources that the player disposes of (gold, oil, wood and food), fourteen represent the description of the buildings (number of town halls, number of barracks, and so on) and finally, five features represent the map (size in both dimensions, percentage of water, percentage of trees and number of gold mines).

The case retrieval process uses a standard nearest neighbor algorithm but with a similarity metric that takes into account both the goal and the game state. Specifically, we use the following similarity metric:

$$d(c_1, c_2) = \alpha d_{GS}(c_1.S, c_2.S) + (1 - \alpha) d_G(c_1.G, c_2.G)$$

where d_{GS} is a simple Euclidean distance between the game states of the two cases (where all the attributes are normalized between 0 and 1), d_G is the distance metric between goals, and α is a factor that controls the importance of the game state in the retrieval process (in our experiments we used $\alpha = 0.5$). To measure distance between two goals $g_1 = name_1(p_1, ..., p_n)$ and $g_2 = name_2(q_1, ..., q_m)$ we use the following distance:

$$d_G(g_1, g_2) = \begin{cases} \sqrt{\sum_{i=1...n} \left(\frac{p_i - qi}{P_i}\right)^2} & \text{if } name_1 = name_2 \\ 1 & \text{otherwise} \end{cases}$$

where P_i is the maximum value that the parameter i of a goal might take (we assume that all the parameters have positive values). Thus, when $name_1 = name_2$, the two goals will always have the same number of parameters and the distance can be computed using an Euclidean distance among the parameters. The distance is maximum (1) otherwise.

The result of the retrieval process is a case that contains a behavior that achieves a goal similar to the requested one by the RTEE, and that can be applied to a similar map than the current one (assuming that the case base contains cases applicable to the current map). The behavior contained in the retrieved case then needs to go through the adaptation process. However, our system requires *delayed adaptation* because adaptation is done according to the current game state, and the game state changes with time. Thus it is interesting that adaptation is done with the most up to date game state (ideally with the game state just before the behavior starts execution). For that reason, the behavior in the retrieved case is initially directly sent to the RTEE. Then, when the RTEE is just about to start the execution of a particular behavior, it is sent back to the BG module for adaptation.

The adaptation process consists of a series of rules that are applied to each one of the basic operators of a behavior so that it can be applied in the current game state. Specifically, we have used two adaptation rules in our system:

- *Unit adaptation*: each basic action sends a particular command to a given unit. For instance the first action in the behavior shown in Figure 5 commands the unit "2" to build a "pig-farm". However, when that case is retrieved and applied to a different map, that particular unit "2" might not correspond to a peon (the unit that can build farms) or might not even exist (the "2" is just an identifier). Thus, the unit adaptation rule finds the most similar unit to the one used in the case for this particular basic action. To perform that search, each unit is characterized by a set of 5 features: owner, type, position (x,y), hit-points, and status (that can be *idle*, *moving*, *attacking*, etc.) and then the most similar unit (according to an Euclidean distance using those 5 features) in the current map to the one specified in the basic action is used.
- *Coordinate adaptation*: some basic actions make reference to some particular coordinates in the map (such as the *move* or *build* commands). To adapt the coordinates, the BG module gets (from the case) how the map in the

	map1	*map2*	*map3*
trace1	3 wins	3 wins	1 win, 1 loss, 1 tie
trace2	1 loss, 2 ties	2 wins, 1 ties	2 losses, 1 tie
trace1 & trace2	3 wins	3 wins	2 wins 1 tie

Table 2. Summary of the results of playing against the built-in AI of WARGUS in several 2-player versions of "Nowhere to run nowhere to hide".

particular coordinates looks like by retrieving the content of the map in a 5x5 window surrounding the specified coordinates. Then, it looks in the current map for a spot in the map that is the most similar to that 5x5 window, and uses those coordinates.

8 Experimental results

To evaluate our approach, we used several variations of a 2-player version of the well known map "Nowhere to run nowhere to hide", all of them of size 32x32. As explained in Section 3, this map has the characteristic of having a wall of trees that separates the players and that leads to complex strategic reasonings. Specifically, we used 3 different variations of the map (that we will refer as *map1*, *map2* and *map3*), where the initial placement of the buildings (a gold mine, a townhall and a peasant in each side) varies strongly, and also the wall of trees that separates both players is very different in shape (e.g. in one of the maps it has a very thin point that can be tunneled easily).

We recorded expert traces for the first two variants of the map (that we will refer as *trace1* and *trace2*). Specifically, *trace1* was recorded in *map1* and used a strategy consisting on building a series of ballistas to fire over the wall of trees; and *trace2* was recorded in *map2* and tries to build defense towers near the wall of trees so that the enemy cannot chop wood from it. Each trace contains 50 to 60 actions, and about 6 to 8 cases can be extracted from each of them. Moreover, in our current experiments, we have assumed that the expert wins the game, it remains as future work to analyze how much the quality of the expert trace affects the performance of the system.

We tried the effect of playing with different combinations of them in the three variations of the map. For each combination, we allowed our system to play against the built-in AI three times (since WARGUS has some stochastic elements), making a total of 27 games.

Table 2 shows the obtained results when our system plays only extracting cases from *trace1*, then only extracting cases from *trace2*, and finally extracting cases from both. The table shows that the system plays the game at a decent level, managing to win 17 out of the 27 games it played. Moreover, notice that when the system uses several expert traces to draw cases from, its play level increases greatly. This can be seen in the table since from the 9 games the system played using both expert traces, it won 8 of them and never lost a game, tying only once. Moreover, notice also that the system shows adaptive behavior

since it was able to win in some maps using a trace recorded in a different map (thanks to the combination of planning, execution, and adaptation).

Finally, we would like to remark the low time required to train our system to play in a particular map (versus the time required to write a handcrafted behavior to play the same map). Specifically, to record a trace an expert has to play a complete game (that takes between 10 and 15 minutes in the maps we used) and then annotate it (to annotate our traces, the expert required about 25 minutes per trace). Therefore, in 35 to 40 minutes of time it is possible to train our architecture to play a set of WARGUS maps similar to the one where the trace was recorded (of the size of the maps we used). In contrast, one of our students required several weeks to hand code a strategy to play WARGUS at the level of play of our system. Moreover, this are preliminary results and we plan to systematically evaluate this issue in future work. Moreover, as we have seen our system is able to combine several traces and select cases from one or the other according to the current situation. Thus, an expert trace for each single map is not needed.

9 Conclusions

In this paper we have presented a case based planning framework for real-time strategy games. The main features of our approach are a) the capability to deal with the vast decision spaces required by RTS games, b) being able to deal with real-time problems by interleaving planning and execution in real-time, and, c) solving the knowledge acquisition problem by automatically extracting behavioral knowledge from annotated expert demonstrations in form of cases. We have evaluated our approach by applying it to the real-time strategy WARGUS with promising results.

The main contributions of this framework are: 1) a case based integrated real-time execution and planning framework; 2) the introduction of a behavior representation language that includes declarative knowledge as well as procedural knowledge to allow both reasoning and execution; 3) the idea of automatic extraction of behaviors from expert traces as a way to automatically extract domain knowledge from an expert; 4) the idea of heterogeneous case bases where cases that contain solutions for several different problems (characterized as *goals* in our framework) coexist and 5) the introduction of *delayed adaptation* to deal with dynamic environments (where adaptation has to be delayed as much as possible to adapt the behaviors with the most up to date information).

As future lines of research we plan to experiment with adding a case retention module in our system that retains automatically all the adapted behaviors that had successful results while playing, and also annotating all the cases in the case base with their rate of success and failure allowing the system to learn from experience. Additionally, we would like to systematically explore the transfer learning [15] capabilities of our approach by evaluating how the knowledge learnt (both from expert traces or by experience) in a set of maps can be applied to a different set of maps. We also plan to further explore the effect of adding more

expert traces to the system and evaluate if the system is able to properly extract knowledge from each of them to deal with new scenarios.

Further, we would like to improve our current planning engine so that, in addition to sequential and parallel plans, it can also handle conditional plans. Specifically, one of the main challenges of this approach will be to detect and properly extract conditional behaviors from expert demonstrations.

Acknowledgements The authors would like to thank Kane Bonnette, for the java-WARGUS interface; and DARPA for their funding under the Integrated Learning program TT0687481.

References

[1] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *Artificial Intelligence Communications*, 7(1):39–59, 1994.

[2] David Aha, Matthew Molineaux, and Marc Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In *ICCBR'2005*, number 3620 in LNCS, pages 5–20. Springer-Verlag, 2005.

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

[4] B. Chandrasekaran. Design problem solving: a task analysis. *AI Mag.*, 11(4):59–71, 1990.

[5] Richard Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intellicence*, 2(3/4):189–208, 1971.

[6] Andrea Freßmann, Kerstin Maximini, Rainer Maximini, and Thomas Sauer. CBR-based execution and planning support for collaborative workflows. In *ICCBR Workshop*, pages 271–280, 2005.

[7] Allen F. G. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[8] Kristian F. Hammond. Case based planning: A framework for planning from experience. *Cognitive Science*, 14(3):385–443, 1990.

[9] M. Hoang, S. Lee-Urban, and H Muñoz-Avila. Hierarchical plan representations for encoding strategic game ai. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*. AAAI Press, 2005.

[10] Michael Mateas and Andrew Stern. A behavior language for story-based believable agents. *IEEE intelligent systems and their applications*, 17(4):39–47, 2002.

[11] H. Muñoz-Avila and D. Aha. On the role of explanation for hierarchical case-based planning in real-time strategy games. In *Proceedings of ECCBR-04 Workshop on Explanations in CBR*, 2004.

[12] Karen L. Myers. CPEF: A continuous planning and execution framework. *AI Magazine*, 20(4), 1999.

[13] J. Orkin. Applying goal-oriented action planning to games. In *AI Game Programming Wisdom II*. Charles River Media, 2003.

[14] M. Ponsen, H. Muñoz-Avila, P. Spronck, and D. Aha. Automatically acquiring adaptive real-time strategy game opponents using evolutionary learning. In *Proceedings of the 20th National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence*, pages 1535–1540, 2005.

[15] Manu Sharma, Michael Homes, Juan Santamaria, Arya Irani, Charles Isbell, and Ashwin Ram. Transfer learning in real time strategy games using hybrid CBR/RL. In *IJCAI'2007*, page to appear. Morgan Kaufmann, 2007.