In Computational Intelligence Journal, Volume 26, Issue 1, pp. 84-119

On-line Case-Based Planning

Santi Ontañón and Kinshuk Mishra and Neha Sugandh and Ashwin Ram
CCL, Cognitive Computing Lab,
Georgia Institute of Technology,
Atlanta, GA 30322/0280, USA
{santi,kinshuk,nsugandh,ashwin}@cc.gatech.edu

Abstract

Some domains, such as real-time strategy (RTS) games, pose several challenges to traditional planning and machine learning techniques. In this paper, we present a novel on-line case-based planning architecture that addresses some of these problems. Our architecture addresses issues of plan acquisition, on-line plan execution, interleaved planning and execution and on-line plan adaptation. We also introduce the Darmok system, which implements this architecture in order to play WARGUS (an open source clone of the well-known RTS game WARCRAFT II). We present empirical evaluation of the performance of Darmok and show that it successfully learns to play the WARGUS game.

Keywords: Case-Based Reasoning, Case-Based Planning, Computer Game AI

1 Introduction

Computer games have been classified as the "Human-level AI's Killer Application" [27]. State-of-the-art computer games recreate real-life environments with a surprising level of detail. However, even though there have been enormous advances in computer graphics, animation and audio for games, most games contain very basic artificial intelligence (AI) techniques. There are several reasons for this: first, there is a disconnect between the goals of academic AI and game AI (where academic AI focuses in achieving stronger AI, game AI focuses in achieving AI that is more fun to play with). Other reasons include, the low percentage of CPU allocated for the AI, or the reluctance towards AI techniques from game designers (since they lose control of the behavior of the game). However, there is increasing interest in the game industry for some AI techniques such as hierarchical planning, nominated one of the top 5 game AI trends for 2008. The *AI Game Programming Wisdom* series [36, 37] provides a good overview of current state of the art AI techniques used in the game industry.

AI techniques have been successfully applied to several computer games such as chess, backgammon or checkers. However, traditional AI techniques fail to

perform well in most current commercial computer games because these games have vast search spaces in which the AI has to make decisions in real-time, rendering traditional search-based techniques inapplicable. Real-time strategy (RTS) games are one good example of such games. RTS games have several characteristics that make the application of traditional planning approaches difficult: they have huge decision spaces [2], they are adversarial domains, they are non-deterministic, non fully-observable, and finally it is difficult to define postconditions for operators (actions don't always succeed, or take different amount of time, and have complex interactions that are difficult to model using typical planning representation formalisms). Section 4 presents a more detailed explanation of the difficulties of RTS games.

Machine learning (ML) cannot directly handle RTS games either, since the huge state space of RTS games makes learning state-action mappings unfeasible. However, several successful approaches exist that can learn small subproblems inside an RTS game (optimal resource harvesting, optimal city location, etc.). A lot of knowledge engineering is required to provide the system with a compact representation of the state space. However, even with such knowledge, the problem is still not trivial. In order to apply machine learning techniques we need a method to classify actions as correct or incorrect in order to build training examples. This is not easy in a complex game, since a player might execute hundreds of actions over the course of a game, and it's hard to determine which ones were "correct" and which ones were "incorrect" — the well known blame assignment problem.

Case-based reasoning (CBR), and in particular, case-based planning (CBP), is a promising family of techniques that combine aspects of planning with aspects of machine learning. Case-based planning techniques [42] work by reusing previous stored plans for new situations instead of planning from scratch. Thus, CBP combines both planning and learning. However, several issues also arise when trying to deal with RTS games using case-based planning. First, CBP systems require a library of plans in order to plan. Second, RTS games require real-time interaction, and thus planning must be interleaved with execution, monitoring the plan execution and informing the planner of possible failures. Third, CBP techniques require *plan adaptation* (since it is very unlikely that a plan will fit the solution exactly) and, due to the time constraints imposed by RTS games, such adaptation techniques must be efficient.

In this paper we propose a novel on-line case-based planning (OLCBP) architecture able to deal with the complexity of RTS domains and propose an extension of the traditional CBR cycle, that we call the OLCBP cycle. We explain in detail the issues that arise when trying to deal with domains such as RTS games and propose solutions to these issues in our OLCBP framework. In particular, we will present solutions to the case-acquisition problem in CBP by analyzing expert demonstrations and extracting cases from them. We will introduce a new case-based planning technique with interleaved planning and execution. We will also introduce a new case-based planning adaptation technique that is domain independent and efficient. Finally, we will introduce the idea of delayed adaptation. In order to demonstrate the feasibility of the pro-

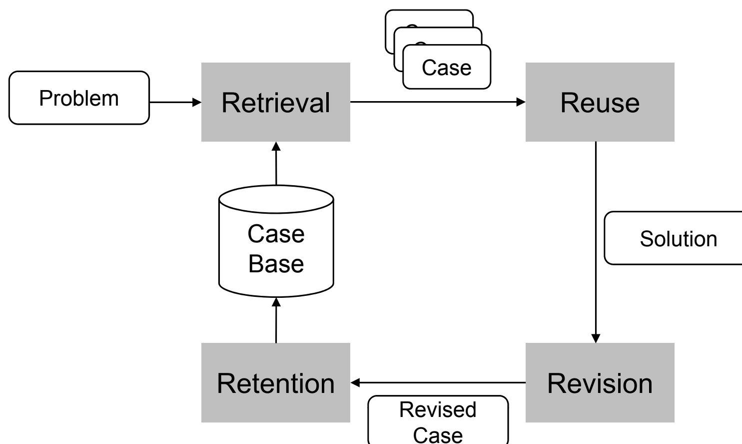

Figure 1: The traditional case-based reasoning cycle.

posed tecniques, we present the Darmok[1] system, a system that implements the proposed OLCBP cycle in order to play WARGUS. We evaluate the Darmok system in a collection of maps, and show that the proposed OLCBP architecture allows Darmok to successfully learn to play WARGUS.

The remainder of this paper is organized as follows. Section 2 presents previous work on case-based reasoning, planning and case-based planning. Section 3 analyzes the CBR cycle and presents a new OLCBP cycle. Then, Section 4 introduces the issues that RTS games involve. Sections 5 to 10 present the Darmok system and how all the components of the OLCBP architecture are instantiated on it. Finally, Section 11 presents an empirical evaluation of Darmok. The paper closes with conclusions and future work.

2 Background

In this section we introduce some background in case-based reasoning, case-based-planning, real-time domains, and computer games.

2.1 Case-Based Reasoning and Planning Approaches

Case-based reasoning (CBR) [1, 24] is a problem solving methodology based on reutilizing specific knowledge of previously experienced and concrete problem situations (cases). The activity of a case-based reasoning system can be summarized in the CBR cycle [1], shown in Figure 1. The CBR cycle consists of four stages: Retrieve, Reuse, Revise and Retain. In the Retrieve stage, the

[1] Darmok is an episode of *Star Trek: The Next Generation* in which a race called *The Children of Tamar* makes appearance. This race uses a metaphorical language that is reminiscent of CBR processes.

system selects a subset of cases from the case base that are relevant to the current problem. The Reuse stage adapts the solution of the cases selected in the retrieve stage to the current problem. In the Revise stage, the obtained solution is verified (either by testing it in the real world or by examination by an expert), which provides feedback about the correctness of the predicted solution. Finally, in the Retain stage, the system decides whether or not to store the new solved case into the case base.

Case-based planning is planning as remembering [17], and involves reusing previous plans and adapting them to suit new situations. There are several motivations for case-based planning techniques [42]: first, it inherits the *psychological plausibility* from case-based reasoning, and second, they have the potential to increase the *efficiency* with respect to generative planners. Although, under certain conditions [32], reusing plans has the same or even higher worst-case complexity than planning from scratch [34], case-based planning can exploit regularities in the problems being solved, and thus potentially greatly increase the efficiency.

A large number of systems and approaches for case-based planning have been presented in the past (see [42] for a complete overview). One of the first case-based planning systems is CHEF [17], that works on the domain of Szechwan cooking, being able to build new recipes based on user's request for dishes with particular ingredients and tastes. CHEF makes use of memory whenever possible. It contains a memory of past failures to warn of problems and also a memory of successful plans from which to retrieve plans. Both memories are updated after each planning episode, whether successful or failed. One of the novel capabilities of CHEF with respect to classical planning systems is its ability to learn. Each time CHEF experiences a planning failure, it means that understanding has broken down and it has to be fixed. Thus, planning failures tell the system when it needs to learn. CHEF performs three kinds of learning: plan learning (new plans), expectation learning (model of the world), and critic learning (plan fixes). A key feature in CBP is how to perform plan adaptation, CHEF performs this task by a set of domain-specific rules called TOPs.

Prodigy/Analogy [45] by Veloso et al. is an architecture that integrates planning and learning. Specifically, the main difference between Prodigy/Analogy and other CBP systems is that instead of reusing previous plans, it reuses previous planning decisions. Prodigy/Analogy stores the reasoning trace of planning episodes, including information of which decisions were taken while planning, why other choices for the decision were not considered, etc. Then, when planning for a new problem, Prodigy/Analogy *replays* the stored traces. This transforms the planner, from a module that performs expensive search through the space of alternatives into a module that tests the validity of choices proposed by past experience and follows equivalent search directions.

CHEF and Prodigy/Analogy illustrate the two approaches to plan adaptation (*transformational* in the case of CHEF and *derivational* in the case of Prodigy/Analogy) typically used in the CBR community. For an overview of case-based plan adaptation techniques see [32]. Moreover, there has been also a considerable amount of work on this topic coming from the planning community.

Domain-independent planning has been shown to be intractable (PSPACE-complete when the set of operators is known in advance, and worse if they are given as the input of the planning problem [11]), and thus, approaches to improve computational cost in planning in general are constantly pursued. One such approach is to reuse previous plans instead of planning from scratch. PRIAR [21] is a planner system designed to support plan reuse. PRIAR internally uses NONLIN [43], and works by annotating generated plans with a *validation structure* that contains an explanation of the internal causal dependencies. The focus of PRIAR is only on plan reuse, not on plan retrieval. However, the authors propose a heuristic that estimates the cost of adaptation as a good basis for plan retrieval (see the work of Cesta and Romano [8] for another proposal for case retrieval using PRIAR). Kambhampati and Hendler showed speedups up to 79% in the blocks world domain using PRIAR compared to planning from scratch.

Finally, Fasciano presented the MAYOR system [12] for playing the popular Sim City game, consisting of a collection of *advocates* that are responsible for different tasks in the domain. Some of those advocates are case-based planners. One of the most interesting aspects of MAYOR is the way it learns from failures. MAYOR has a concept net of all the factors in the game (money, crime, pollution, etc.) and how they are related. Each plan in the plan library has some expectations on those factors (e.g. "decreased crime"). If the expectations are not met, the concept net is used to locate "controllable factors" that could be manipulated to satisfy the expectation failure. These fixes are stored with the plan in the plan library, and the next time the plan is retrieved, the system will check to see whether any of the fixes stored with the plan have to be executed.

2.2 CBR and Planning in Real Time Domains

Traditionally, case-based reasoning or planning techniques have been applied to "static" domains, i.e. domains in which the system has unlimited amount of time to solve each problem, and during this time, the "world state" does not change. However, most real-world domains are dynamic. Systems have time constraints, and must deal with a dynamic environment that changes over time. There have been several approaches both in the CBR community and in the planning community to deal with this issue. Particularly interesting is the relation between the case-based planning research coming from the planning community and the on-line planning research since, in both areas, plan adaptation and reuse is a key capability.

A common application area is robotics, where typically fast and reactive techniques such as Brooks's subsumption architecture [6], are needed to successfully control the robot. A good example of CBR approach to this domain is the Continuous CBR (CCBR) approach by Ram and Santamaria [39] implemented in the SINS system. CCBR is characterized by implementing continuous representations of the domain, having continuous performance, and continuous learning. SINS controls the parameters of a navigation system for a robot by continuously executing the CBR cycle to update such parameters.

The cases in SINS store sequences of observed inputs associated with observed outputs after executing specific actions (parameter configurations). SINS uses a reinforcement-learning mechanism to update the content of cases in the case base at each cycle by matching the current situation with previous cases and analyzing whether cases properly predicted the current situation or not. The CCBR approach handles real-time domains by having a very quick CBR cycle, continuously executed and monitoring the world in order to set the appropriate control parameters for a robot's navigation system.

In the planning community, the concept of a *reactive planner* was developed to deal with dynamic domains. A reactive planner is a system that builds or changes its plans in response to the shifting situations at execution time. Reactive Action Packages (RAPs) were presented by Firby [14] as a model of reactive planning. RAPs follow three principles: a) all decisions are taken based only in the current world state, b) when a RAP finishes it guarantees to have satisfied its goal, and c) should a RAP fail it is because it has exhausted every possible avenue of attack for the problem. A RAP consists of three things: a goal success check, a validity check (whether a RAP is applicable or not) and a set of possible task nets to achieve the goal. Firby presented an execution module that could plan to achieve goals reactively using RAPs by hierarchically combining them (the task nets of a RAP might decompose a task into other RAPs or into basic actions).

Related to planning in real time domains is the relation between planning and execution, explored by several authors, such as the NASL formalism by McDermott [30] or the IPEM architecture by Ambros and Steel [3]. Other work on planning and execution includes the work concerning re-planning or plan reuse in the planning community, such as the PRIAR system (presented in the previous section) or RepairSHOP [46], that can adapt hierarchical task network (HTN) plans.

In the following section, we discuss how all the ideas introduced about case-based planning and execution in real-time domains affects the CBR cycle.

3 On-Line Case-Based Planning: The CBR cycle Revisited

The CBR cycle, shown in Figure 1, makes two assumptions that are not suited for strategic real-time domains involving on-line planning. The first assumption is that problem solving is modeled as a single-shot process, i.e. a "single loop" in the CBR cycle solves a problem. In Case-Based Planning, solving a problem might involve solving several subproblems, and also monitoring their execution (potentially having to solve new problems along the way). The second assumption is that execution and problem solving are decoupled, i.e. the CBR cycle produces a solution, but the solution execution is delegated to some external module. In strategic real-time domains, executing a problem is part of solving it, specially when the internal model of the world is not 100% accurate, and en-

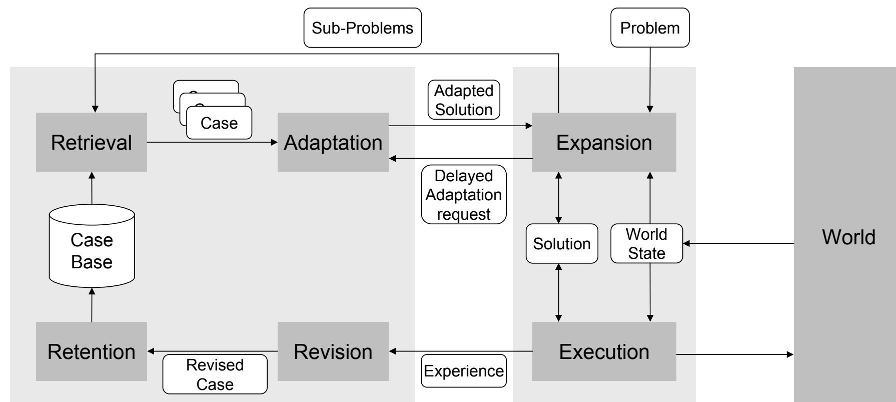

Figure 2: The on-line case-based planning cycle (OLCBP).

suring that the execution of the solution succeeds is an important part of solving problems. For instance, while executing a solution the system might discover low level details about the world that render the proposed solution wrong, and thus another solution has to be proposed.

Of the systems presented in the previous section, only five consider execution failures explicitly in their problem solving cycle: RAPs, NASL, SINS, CHEF and Mayor. Other systems (such as PRIAR) can handle execution failures by replanning, but in such cases, the system itself does not incorporate execution failure handling. Using those systems as a starting point and having the goals of our own Darmok system in mind, we redesigned the original CBR cycle to incorporate functions needed for real-time strategic domains. Let us present it in some detail, and explain how some of the previously presented systems fit this new version of the cycle.

Figure 2 presents an extension of the CBR cycle, called the OLCBP (On-Line Case-Based Planning) cycle, with two added processes needed to deal with planning and execution of solutions in real-time domains, and some other small variations. The two new added processes are:

- *Expansion*: this process takes the current solution proposed by the system for a problem (i.e. the *current plan*), and tries to find open sub-problems (sub-goals) in it. If there are any, these sub-problems are sent to the retrieve process so that they can be solved. Another responsibility of the expansion module is to monitor the world state for changes, and send plans to the adaptation module again in case the world state changes enough so that plans have to be changed. We call this a *delayed adaptation*, since adaptation is delayed and performed at run-time with the latest game state. This is an important feature for systems working in dynamic environments as shown in [35].

- *Execution*: this process is in charge of executing the current plan and

updating its status according to the result of execution. If a particular step in the plan fails when executed, and that causes a particular sub-problem to fail, then the execution process will update the current plan to reflect this. When that happens the expansion module will be responsible to find an alternative plan for such a sub-problem.

The flow in the OLCBP cycle is as follows. Problems arrive to the *expansion* process, that decomposes it into sub-problems if need be. Each of these sub-problems are sent to the *retrieval* process, that retrieves relevant cases from the case base. The *adaptation* process generates a solution to the sub-problems by reusing and adapting the solutions in the retrieved cases. These solutions are sent back to the *expansion process*, which incorporates them into the current solution. At the same time that the *expansion* process constructs the solution, the *execution* process executes the parts of the solution that have been fully expanded. As a result of execution these solutions are sent to the *revision* process that verifies the solutions proposed, based on their outcome in the world, and finally the *retention* process decides whether to retain the new experiences or not.

There are four other refinements with respect to the original CBR cycle:

1. Problems "enter the cycle" through the *expansion* process. When a new problems arrives to the system, this problem is set as the current plan, i.e. the current plan consists of a single open problem. Thus, the first thing the expansion module will do is to send this problem to the retrieve process.

2. The CBR cycle is divided in two parts: a first part composed of retrieval and adaptation (or reuse), and a second part composed of revision and retention. The first part is in charge of finding solutions to new problems (i.e. it corresponds to the problem solving capability of the CBR cycle), and the second part is responsible for learning from experience. Revision takes as input the outcome of executing the solutions provided by the system and revises them to verify they achieve their goals. Revised solutions are handed to retention, that will decide whether to retain new cases or not, and update any internal indexes or similarity metrics when required.

3. The new cycle incorporates the *world* in it's design, since it is an important part of any real-time problem solving process.

4. The new cycle features a new *delayed adaptation* cycle (notice that there is a loop between adaptation and expansion). In a domain where the domain changes dynamically, we want to delay adaptation till the last moment to ensure that plans are adapted with the latest information. Thus, the expansion component may send back plans to adaptation if the environment changed too much since the last time the plan was adapted. Moreover, notice that for real-time domains it is important for adaptation techniques to be efficient.

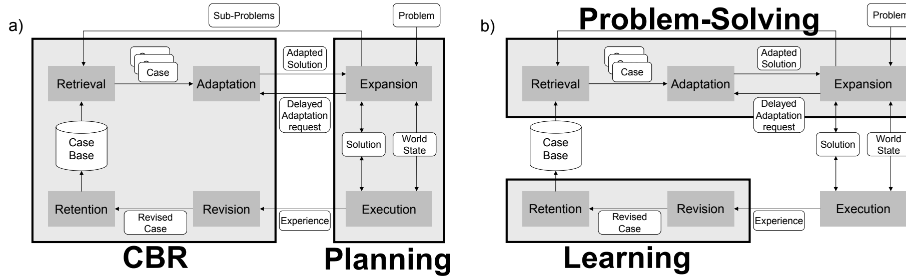

Figure 3: Two interpretations of the OLCBP cycle.

Figure 3 presents an analysis of the OLCBP cycle. Figure 3.a shows how the retrieval, adaptation, revision and retention boxes in the OLCBP cycle actually correspond to the original CBR cycle, while the expansion and execution processes compose the on-line planning and execution cycle. Figure 3.b shows that the top three processes in the cycle (retrieval, adaptation and expansion) provide the problem-solving capabilities to the system, while the retention plus revision processes provide for the learning capabilities. If we look carefully at Figure 3.b, we can see that the OLCBP cycle breaks up nicely in three parts: problem-solving, execution and learning.

Let us show that the proposed cycle is representative of previous CBP approaches by analyzing some of them using the proposed cycle. If we consider the CHEF, PRIAR, or MAYOR systems, they share a common feature: when solving a new planning problem, they retrieve a single plan and adapt it. Thus, in those systems solving a problem consists of a “single loop” through the CBR cycle (with a highly computationally expensive adaptation process in the case of PRIAR or SPA). Moreover, out of those systems only MAYOR considers execution, thus the CHEF, PRIAR systems do not incorporate any expansion or execution processes. Another system we might consider is the Systematic Plan Adaptor (SPA) [18], that like PRIAR, only considers one plan and does not consider execution. The Multi-Plan Adaptor (MPA) [38] is an evolution of SPA that combines pieces of multiple plans. One can think of MPA as a way to allow SPA to contain an expansion process by merging several cases retrieved with the current plan. MAYOR, however, incorporates an execution component that monitors the execution of plans, and provides this information to the revise process that can properly learn from those experiences. The TOLTEC system does implement an expansion process, since it executes the CBR cycle recursively, expanding the plan more and more each time. However, TOLTEC does not contain an execution process, and thus does not deal with interleaved planning and execution. Another system to be considered is SINS, which does not contain an expansion process (since SINS does not perform planning) but does contain an execution process in the form of the controller (that directly controls the robot) and the constant monitoring of the environment to learn about the effects of the actions taken in the environment. A completely different example

Figure 4: A screenshot of WARGUS a popular real-time strategy game.

is Prodigy/Analogy, where cases are used to remember previous planning decisions. One could interpret Prodigy/Analogy as having an expansion process that is a full planner, which constantly consults the case base to retrieve cases that contain similar planning situations and are used to constrain the search space of the planner.

The rest of this paper presents how this architecture has been implemented for the particular problem of RTS games in the Darmok system.

4 On-Line Case-Based Planning in Real-Time Strategy Games

Real-time strategy (RTS) games have several characteristics that make the application of traditional planning approaches difficult:

- They have huge decision space (i.e. the set of different actions that can be executed in any given state is huge).
- Huge state space (the combination of the previous bullet and this bullet makes them not suitable for search based AI techniques [2, 7]).
- They are non-deterministic.
- They are incomplete information games, where the player can only sense the part of the map he has explored and include unpredictable opponents.
- They are real-time. Thus, while the system is deciding which actions to execute, the game continues executing and the game state changes constantly.

- They are difficult to represent using classical planning formalisms since postconditions for actions cannot be specified easily.

For example, Wargus (Figure 4) 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 troops, and troops are required to attack the enemy. In addition, players can also build defensive buildings such as towers (or even use farms as walls to block the enemy). Therefore, Wargus involves complex reasoning to determine where, when and which buildings and troops to build. A standard playing strategy for a "standard" map involves: building up a resource infrastructure (gold, wood and oil), developing some defenses (towers, archers or footmen), building some attacking units and finally sending them to attack the enemy. However, standard strategies are not effective against a human expert in most maps, since humans exploit the unique features of each map to come up with interesting strategies. For example, the map shown in Figure 4 is a 2-player version of the classic map "Nowhere to run Nowhere to hide" (NWTR), with a wall of trees that separates the players. The NWTR map is a popular map played by humans, and recognized by human experts as being highly strategic. Different strategies commonly employed are:

- Building long range units (such as catapults or ballistae) to attack the other player before the wall of trees has been destroyed,
- tunneling early in the game through the wall of trees trying to catch the enemy by surprise,
- blocking the wood supplies of the opponent by defending the wall of trees with ranged units,
- even an air attack using flying units might be possible if resources are managed properly and the player manages to block the enemy for the amount of time needed to produce flying units.

Thus, a standard playing strategy won't work on this map against an expert player.

There are several reasons why traditional search-based planning approaches cannot be directly applied to domains such as Wargus. For instance, if we follow the analysis performed in [2], the approximate number of different commands that can be issued in the situation depicted in Figure 4 is about 280,000. Thus, classical adversarial search using a minimax kind of algorithm is not feasible.

Traditional STRIPS [13] planning cannot be directly applied since the problem space is too large. HTN planning [33] can handle larger problems than traditional domain independent planning algorithms such as STRIPS. However the benefit comes with the cost of having to define a certain amount of domain

knowledge (in the form of domain dependent methods that an HTN planner uses to decompose tasks in subtasks). Moreover, even if those techniques were applicable, there is still the problem that WARGUS is an adversarial domain and, thus, adversarial planning techniques should be used instead. Adversarial planning techniques have the disadvantage that plans must be a tree of contingencies for all possible ordering of the opponent actions [4], thus greatly increasing the complexity of the task with respect to traditional planning.

Moreover, even if the computational complexity of the planning algorithms was low enough to be applied to WARGUS, classical planning algorithms assume deterministic domains. WARGUS (as most real-time strategy games) is non deterministic. Thus, probabilistic planning techniques are required. Probabilistic planning techniques are typically based on MDPs (Markov Decision Processes) or POMDPs (Partially Observable Markov Decision Processes) [20]. Such techniques provide a firm foundation for planning in probabilistic domains but have the downside of being computationally expensive. In addition, they model the problem as a synchronous interaction between an agent and the world, whereas in RTS games such as WARGUS interactions are not synchronous. Blum and Langford [4] showed that planning in probabilistic domains can be done with the same computational complexity as in deterministic domains if we sacrifice optimality. However, even that is not enough to fully deal with domains such as WARGUS, since that only states that we will have the same computational complexity as traditional planners (PSPACE-complete). Finally, in addition to being non-deterministic, real-time strategy games are also non-fully observable domains. Most games feature the "fog-of-war", that makes a player only able to observe the subset of the map where he has units. Planners that support non-fully observable domains exist (an example is ZANDER [29]) but their computational complexity makes them inapplicable to domains such as WARGUS unless we use some level of abstraction both in the action space and in the state space. The IPEM system [3] can also deal with incomplete information domains by means of sensing actions, but has also the problem of being a systematic planner, not suitable for real-time domains.

Finally, there is a key difference that makes domains such as WARGUS difficult to deal with using traditional planning techniques. This difference is that unlike in traditional planning domains, it is very difficult to define the effect of actions in WARGUS using preconditions and postconditions. In STRIPS-like planning representations, actions specify preconditions and postconditions, so that a planner can match postconditions with preconditions using forward or backwards chaining and construct a plan. However, in complex domains, such matchings are not easy to perform. Let us consider the action "attack" in the WARGUS domain. The only post-condition that we can specify for the attack action is that if a unit is commanded to attack, it will be in the "attack" mode afterwards. But it is difficult to determine if such a unit will succeed in its attack, or if it will reach its goal, or how much time will the unit take to destroy the target. Moreover, if we command several units to attack the same target, the probability of success increases. If one of the conditions our planner has to satisfy at a given moment is "destroy a particular enemy unit", it won't find

any operator, or action, with post conditions "destroy enemy unit", since such post-condition cannot be specified.

The previous example could be addressed using a probabilistic framework where the possible different outcomes of an action will be assigned some probabilities (since in this particular example we might just be interested in predicting whether a particular unit would be killed or not, and thus there are only two possible outcomes). However, in general, the number of possible outcomes are huge so none of the existing planning algorithms for MDPs or nondeterministic planning domains would be able to handle this problem. A more complicated example can be seen with the resource gathering actions. If a precondition states that we need to have "1000 gold units" to perform a task, there is no operator with a post-condition of "1000 gold units", since the "harvest" operators can only ensure that a unit will start the resource gathering process. Moreover, it will be difficult for the planner to figure out how many units to assign for harvesting in order to have the resources "on time" (where "on time" is also difficult to assess, since it depends on what the opponent actions are).

Our OLCBP cycle has been designed with domains such as RTS games in mind. This approach has been implemented in the Darmok system, designed to play the full WARGUS game. The only aspect of WARGUS still not covered by Darmok is the "fog-of-war", that has been disabled in our experiments.

5 Darmok

In this section we will present the Darmok system, that implements the previously described architecture for case-based planning for the WARGUS domain. Darmok is an evolution of the system reported in [35], that didn't consider plan adaptation or learning from experience.

The Darmok system learns how to play WARGUS by observing how humans play. Darmok learns what we call *plan snippets* by observing a human play, and stores those snippets in the system in the form of cases. Such snippets are then retrieved and composed together to form *plans*. A snippet is similar to the concept of a method in HTN [33] planning, where methods are composed to satisfy tasks and subtasks in a HTN, and the whole network corresponds to an abstract plan, however, as we will see later, there are several key differences between HTN planning and Darmok. Figure 5 shows an overview of Darmok's architecture. Darmok's execution can be divided in two main stages:

- *Learning*: During learning Darmok observes a game trace to learn plan snippets that will be stored in the case base. In our experiments, an expert plays WARGUS to generate a trace. However, notice that the system could learn from any trace, even from traces of itself playing, or observing other systems play. Then, the trace is annotated by the expert, explaining the goals he was pursuing with the actions he took while playing. Using those annotations, a set of snippets are extracted from the trace and stored as a set of cases. For each snippet, the situation in which it was executed, the goal it was pursuing, and its success or failure are stored in the case base.

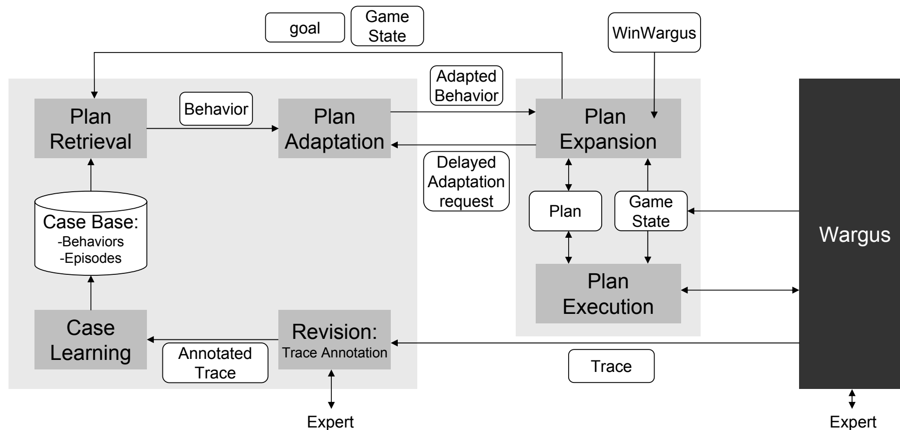

Figure 5: Overview of our case-based planning approach.

- *Execution*: *Plan Retrieval*, *Plan Adaptation*, *Plan Expansion* and *Plan Execution* are in charge of maintaining a current plan to win the game. The *Plan Execution* module is in charge of executing the current plan, and update its state (e.g., marking which actions succeeded or failed). The *Plan Expansion* module is in charge of identifying open goals in the current plan and expand them. In order to do that it relies on the *Plan Retrieval* module, which given an open goal and the current game state retrieves the most appropriate plan snippet to fulfill the open goal. Finally, we have the *Plan Adaptation* module in charge of adapting the retrieved snippets according to the current game state.

One of the key aspects of the system is that it interleaves planning with execution in order to deal with dynamic domains. Moreover, unlike traditional HTN planning systems, Darmok does not perform any search process (which is not suitable for real-time domains), instead, Darmok uses a combination of case-based plan retrieval with search-free plan adaptation in order to find suitable snippets to satisfy each of the goals in the plan. Additionally, another difference with respect to HTN planning is the type domain knowledge that has to be provided to the system. HTN planners require a set of tasks, a set of primitive actions, a vocabulary for conditions (for preconditions and postconditions), and a collection of methods to decompose each task. Darmok, on the other hand, requires: a set of goals, a set of primitive actions, a vocabulary for conditions, a set of features to represent the game state (used for plan retrieval), a set of annotated expert traces, and (as we will explain later) a set of rules to help the system perform precondition-postcondition matching. Notice that the knowledge required by Darmok defines the characteristics of the domain and not the strategies to use (corresponding to the methods in HTN planning) which are learned by Darmok.

In the remainder of this paper we will present all the modules in detail.

Sections 6, presents the language we use to represent plans. Then, Section 7 describes the learning stage, including the revision process and the case learning process. Section 8 presents the plan retrieval module. Section 9 describes the plan expansion and plan execution modules and Section 10 presents the plan adaptation module.

6 Plan Representation in Darmok

In this section we will present the plan representation formalism used by Darmok, designed to allow a system to learn plans, represent them, and to reason about them and their intended and actual effects. Our language is based on the classic STRIPS [13] planning language, but further extended to allow more expressiveness and reasoning and learning capabilities over the language.

The basic constituent piece is the *snippet*. Snippets are composed of three elements:

- A set of *preconditions* that must be satisfied before the plan can be executed. For instance, a snippet 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 plan for it to have chances of success (also known as "maintenance goals" in the planning literature). If at some moment during the execution, the alive conditions are not met, the plan 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.
- The plan itself. which 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 snippet that has to be executed to satisfy that particular subgoal.

Also, snippets are associated with *goals*. A *goal* is a representation of the intended goal of the snippet. For every domain, an ontology of possible goals has to be defined. For instance, a snippet might have the goal of "having a tower". Snippets are similar to the concept of a macro-operator in planning. However, a snippet is a much simpler structure that simply contains one possible decomposition of a goal, while traditionally a macro-operator is a more complicated structure that is able to internally select which is the right decomposition of a goal in more primitive operators.

Notice that unlike classical planning approaches, postconditions cannot be specified for snippets, since a snippet is not guaranteed to succeed. Thus, we can only specify the goal a snippet pursues, i.e., its *success conditions*. It is

important to realize the difference between a postcondition and a success condition. A postcondition is a condition that we can ensure is going to be true after the execution of a snippet (or an action), while a success condition is a condition that when satisfied we can consider the snippet (or action) to have completed. For example, a side effect of an action is a postcondition but not a success condition: "enemy killed" is a success condition of an attack, but not a postcondition since we cannot ensure that after the attack is done the enemy would be killed. If the possible effects of an action could be enumerated, they could be modeled using a probabilistic planning framework such as MDPs. However, in complex RTS games the list of possible effects of an action is huge (or even infinite), and thus not representable using such frameworks. Our use of *success conditions* instead of postconditions defines an abstraction over the notion of a nondeterministic action in planning handled by interleaving planning and execution.

Specifically, three things need to be defined for using Darmok 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*. For uniformity, in Darmok actions are treated as standard snippets, and thus have a goal, preconditions and alive conditions (so that the system can reason about them too).
- A set of *sensors*, that are used to obtain information about the current state of the world, and are used to specify the preconditions, alive conditions and goals of snippets. 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 set of *success conditions* is defined. For instance, *HaveUnits(TOWER)* is a valid goal in our gaming domain and it has as success condition: *UnitExists(TOWER)*. Therefore, the goal definition can be used by the system to reason about the intended result of a snippet, while the success conditions are used by the execution engine to verify whether a particular snippet succeeds at run time. In the next section we will provide a more in depth description of how goals are defined in our system.

7 Plan Acquisition

Case-based planning systems require a set of plans in its case-base in order to function. However, there has not been much emphasis in previous work on how such a case-base can be acquired. We propose to acquire such cases by analyzing game traces. In Darmok, this is done in two processes, as shown in Figure 5: trace annotation (revision) and case learning.

Cycle	*Player*	*Action*	*Annotation*
8	1	Build(2,“pig-farm”,26,20)	-
137	0	Build(5,“farm”,4,22)	SetupResourceInfrastructure(0,5,2) WinGame(0)
638	1	Train(4,“peon”)	-
638	1	Build(2,“troll-lumber-mill”,22,20)	-
798	0	Train(3,“peasant”)	SetupResourceInfrastructure(0,5,2) WinGame(0)
878	1	Train(4,“peon”)	-
878	1	Resource(10,5)	-
897	0	Resource(5,0)	SetupResourceInfrastructure(0,5,2) WinGame(0)
1118	1	Resource(12,‘5)	-
1126	0	Build(11,“farm”,6,22)	SetupResourceInfrastructure(0,5,2) WinGame(0)
...	...	...	...

Table 1: Snippet of a real trace generated after playing Wargus.

The first step in the process consists of the expert providing a demonstration to the system. In our particular application domain, Wargus, an expert simply plays a game (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 fragment of a real trace from playing a game of Wargus. As the table shows, each 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 5 shows, the next step is to annotate the trace (Revision). Annotation consists of associating which goals were being pursued by each of the actions executed by the expert. Annotation is needed because if the system was to learn snippets simply by observing a human play, it will need to implement plan recognition techniques in order to identify what the human is intending to do at every moment. Thus, annotations provide a way to avoid complex plan recognition techniques. However, as Section 13.1 explains, plan recognition is one of the ways we plan to extend the system in the future to allow it to autonomously learn by simply observing people playing the game.

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:

- $WinGame(player)$: representing that the action had the intention of making the *player* win the game.
- $KillUnit(unit)$: representing that the action had the intention of killing the unit *unit*.
- $Resources(gold, wood, oil)$: the action had the intention of increasing the resource levels to at least the specified levels in the parameters.
- $SetupResourceInfrastructure(player, peasants, farms)$: indicates that the expert wanted to create a good resource infrastructure for player

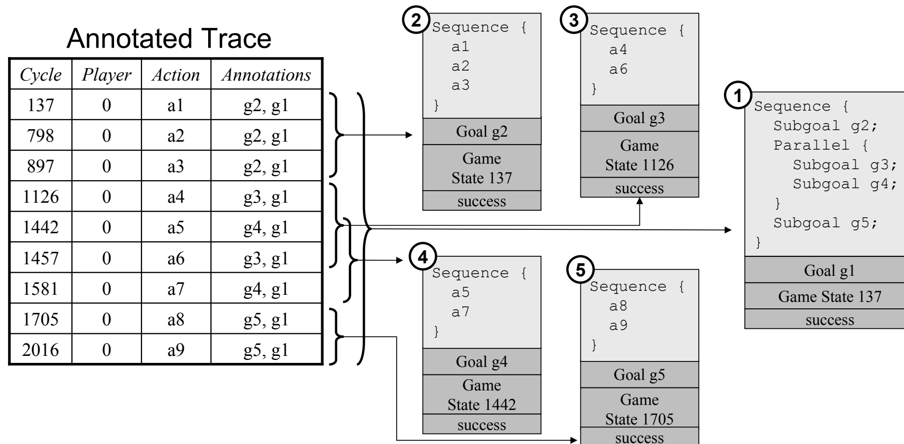

Figure 6: Extraction of cases from the annotated trace.

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 5 shows, the annotated trace is processed by the *case learning* 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 the case base is composed of two structures: *snippets* and *episodes*. A snippet stores just a plan, and an episode stores the outcome of having applied a particular snippet in a particular context to achieve a particular goal. See Section 8 for a more detailed explanation of our case base formalism. In the following we will use the term "cases" when we want to refer to both snippets and episodes without distinguishing between them.

The *case learning* module analyzes the annotated trace 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 6, 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 [15], 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. Darmok determines those relations in the following way:

- If most (90%) of the actions associated with a goal g happen *before* the first action of another goal g', then g and g' are considered to happen in

sequence. Notice that Darmok does not require 100% of the actions to happen before in order to recover from anomalies in the expert traces (it is common in traces generated by humans that the first actions associated with a goal is executed before the previous goal is finished).

- If all the actions associated with a goal g are also associated with another goal g' and the goal g' has some action not associated with g, then g is considered to be a subgoal of g'.
- Otherwise, two goals are considered to be in parallel.

For instance, in Figure 6, *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 6, snippet number 1 (shown in the figure) can be extracted, specifying that to achieve goal *g1* the expert first tried to achieve goal *g2*, then attempted *g3* and *g4* in parallel, and after that *g5* was pursued. Also, we can construct an episode that says that the snippet number 1 was applied in the game state in which the game was at cycle 137 and it succeeded. Then, for each one of the subgoals a similar analysis is performed, leading to four more snippets and four more episodes. For example, case 3 states that a possible way to achieve goal *g2*, is to execute basic actions *a4* and *a6* sequentially.

Darmok doesn't attempt any generalization of the expert actions: generalization is delayed until the adaptation phase. If an expert action is $Build(5, "farm", 4, 22)$, that is exactly the action stored in a snippet. Thus, using the learned snippets to play a new scenario in WARGUS, the particular values of the parameters in the action might not be the most appropriate for the new scenario. In Darmok, the plan adaptation component is in charge of adapting the parameters of an action to a new scenario (see Section 10). Darmok might extract multiple snippets associated with the same goal from the set of traces it has access to. Each of these snippets will be kept as cases in the case base. During game-play, the retrieval module will be responsible of selecting the most appropriate case given the current game state. If the number of cases in the case base increases, the cost of retrieving snippets increases too (see [31] for a technique to improve the efficiency of retrieval in large case bases applied to Darmok).

Increasing the complexity of annotations we could make the learning process easier for Darmok. For instance, the relation among goals could be specified directly by the expert, and thus Darmok would have no need to infer it. Also, experts could annotate the features of the map that helped them decide which strategy to use (so that Darmok could learn feature weights or indexes for cases), or conditions under which they would consider a plan failed or succeeded (to help learning alive conditions and success conditions). However, one of our goals was to minimize the amount of annotation effort of the experts, thus we opted for a simple annotation schema.

Finally, our plan learning component is not yet able to infer alive conditions and preconditions for snippets, which is part of our future work (this might

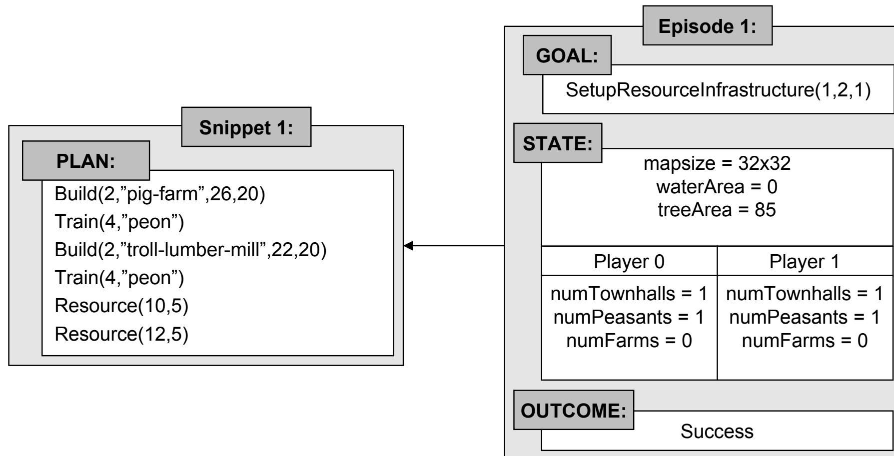

Figure 7: Example of a snippet and an episode extracted from an expert trace for the WARGUS domain.

have the negative effect that the adaptation module will have to do some extra work). Thus, the cases learned consist only of the procedural information in the snippets and a goal, game state and outcome in the episodes, as Figure 7 shows.

8 Plan Retrieval

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 the case base we might have several cases that contain different snippets to solve each one of these problems under different circumstances. Therefore, for any non trivial planning task we will have a *heterogeneous* case base, with snippets that are suited to solve different kind of goals. Additionally, some snippets might work properly in some situations and not as well in other situations, thus in our case base we will also have to include information concerning in which situations a particular snippet has succeeded in the past. We propose to organize the case base using two kinds of elements:

- Snippets: a snippet is a procedure composed of a collection of actions, and subgoals composed in sequence or parallel.
- Episodes: an episode is a tuple $e = \langle p, G, S, O \rangle$ where $e.p$ is a snippet, $e.G$ is a goal, $e.S$ is a situation (i.e. a game state), and $e.O$ is the outcome of applying $e.p$ in $e.S$ to achieve $e.G$. In particular $e.O$ is a real number

between 0 and 1 representing how well the snippet achieved its goal in the situation $e.S$.

After the plan learning process, each snippet in the case base is associated with one episode. Moreover, as the system plays, new episodes will be acquired, and a snippet might be associated with an unbounded number of episodes, storing all the experiences of executing such snippet in different situations. In that way, the system will learn from experience which snippets are suited for which situations and goals.

Figure 7 shows an example of a snippet with an associated episode, with the four elements: a goal (in this case, building the resource infrastructure of player "1") a procedure to achieve the specified goal in the given map in the snippet, a game state (with general features about the map and information about each players) and the outcome for the episode. In particular, in Darmok, the game state is defined by 35 features that represent the state of a WARGUS game. Twelve features represent the number of troops (number of fighters, number of peasants, and so on), four represent the resources the player owns (gold, oil, wood and food), fourteen represent the number 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, number of gold mines).

Ideally, given a game state and a goal, we would like to retrieve the snippet which could have the highest performance in such a goal in the given game state. To predict the performance of a snippet p, the plan retrieval module uses the episodes in the case base associated with p. To retrieve episodes, Darmok uses the *episode relevance* measure $ER(e, S, G)$ that computes the relevance of a given episode e given the current game state S and goal G, and is defined as:

$$ER(e, S, G) = \alpha GS(e.G, G) + (1 - \alpha)SS(e.S, S)$$

where GS is the *goal similarity* of the goal in the snippet p and the goal G, and SS is the *state similarity.* α is a parameter that has been set to 0.75 in our experiments.

The distance between two goals $g_1 = name_1(p_1, ..., p_n)$ and $g_2 = name_2(q_1, ..., q_m)$ is measured as follows:

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

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.

$SS(gs_1, gs_2)$ computes the similarity between two given game states, and returns a number between 0 and 1 (where 0 means identical, and 1 means

totally different). SS is computed as the inverse of a simple Euclidean distance among the game states, where each feature is normalized between 0 and 1.

Finally, to predict the performance of a snippet, we define the set of *relevant episodes* $RE(p, S, G) = \{e_1, ..., e_k\}$, as the set of k episodes associated with the snippet p, and that have the maximum relevance ER. In our experiments, we have set $k \leq 5$, i.e. if there is fewer or equal to 5 episodes associated with p in the case base, then $RE(p, S, G)$ contains all of them, but if there is more than 5, then only the best 5 are included in $RE(p, S, G)$. Using that definition we can now define the predicted performance of a snippet as:

$$PP(p, S, G) = \frac{1 + \sum_{e \in RE(p,S,G)} ER(e, S, G) \times e.O}{2 + \sum_{e \in RE(p,S)} ER(e, S, G)}$$

In other words, the predicted performance is a weighted average of the outcomes that snippet p has had in similar game states to S for similar goals. We add 1 to the numerator and 2 to the denominator following the Laplace probability estimation rule (which biases the predicted performance towards 0.5 when the number of episodes we have to predict the performance is small.)

The result of the retrieval process is the snippet p that has the highest predicted performance $PP(p, S, G)$, i.e. the snippet that has had the best performance in the past for similar goals in a similar game state.

The snippet retrieved then needs to go through the adaptation process. However, real-time domains require *delayed adaptation*. The game state changes with time and it is important that adaptation is done with the most up to date game state (ideally with the game state just before the snippet starts execution). For that reason, in Darmok, when the plan execution module is just about to start the execution of a particular snippet, the snippet is sent to the plan adaptation module for adaptation. Darmok cannot guarantee that the snippet is adapted with the latest game state since in a real-time domain the domain changes continuously, however, delaying adaptation as much as possible minimizes the adaptation error of Darmok.

9 On-Line Plan Expansion and Execution

During execution, the plan expansion, plan execution and plan adaptation occur in parallel in order to maintain a current *partial plan tree* that the system is executing. A *partial plan tree* (or simply the "plan") is represented as a tree consisting of two types of nodes: *goals* and *snippets* (following the same idea of the task/method decomposition [9] or HTNs [33]).

Initially, the plan consists of a single goal corresponding to the planning task at hand. In particular, in Darmok the initial goal is always "win the game". Then, the plan expansion module asks the plan retrieval module to retrieve a snippet for that goal. That snippet might have several subgoals, for which the plan expansion module will again ask the plan retrieval module to retrieve snippets, and so on. For instance, at the top of Figure 8 we can see a sample plan, where the top goal is to "win". The snippet assigned to the "win" goal has

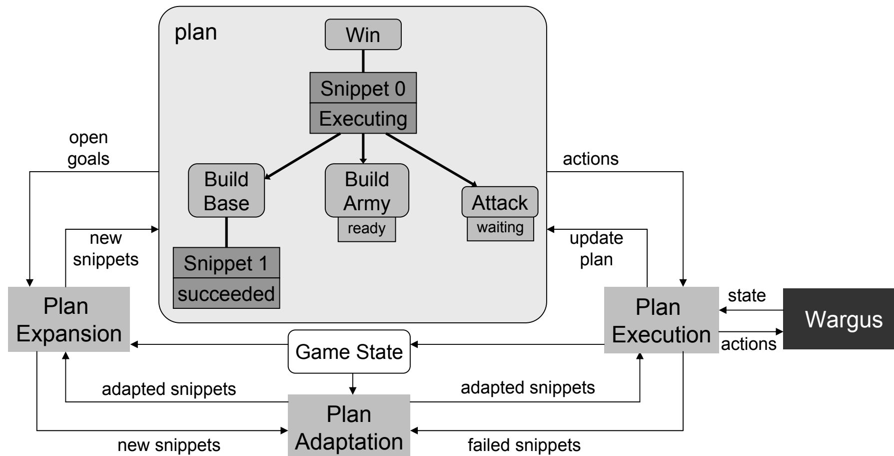

Figure 8: Interleaved plan expansion and execution.

three subgoals, namely "build base", "build army" and "attack". The "build base" goal has already a snippet assigned that has no subgoals, and the rest of subgoals still don't have an assigned snippet. When a goal still doesn't have an assigned snippet, we say that the goal is *open*.

Additionally, each snippet in the plan has an associated state. The state of a snippet can be: *pending*, *executing*, *succeeded* or *failed*. A snippet 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 snippet assigned and where the snippet has failed is also considered to be open (since a new snippet has to be found for this goal).

Open goals can be either *ready* or *waiting*. An open goal is ready when all the snippets that had to be executed before this goal have succeeded, otherwise, it is waiting. For instance, in Figure 8, "snippet 0" is a sequential plan 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.

Figure 9 shows the algorithm the plan expansion module executes at every execution cycle given a plan p and a game state S. The plan expansion module is constantly querying the current plan to see if there is a ready open goal. When this happens, the open goal is sent to the plan retrieval module to retrieve a snippet for it. Then, that snippet is sent to the plan adaptation module, and then inserted in the current plan, marked as pending.

Figure 10 shows the algorithm that the plan execution module executes at each execution cycle, composed of four steps:

- *startReadySnippets*: For each pending snippet, the execution module evaluates the preconditions, and as soon as they are met, the snippet starts its execution. If the current game state has changed since the time the

```
Function PlanExpansion(p, S)
    G = GetReadyOpenGoals(p)
    For g ∈ G Do
        p_g = RetrievePlan(g, S)
        p'_g = AdaptPlan(p_g, g, S)
        p = InsertSnippetInPlan(p, g, p'_g)
    EndFor
    Return p
EndFunction
```

Figure 9: Plan expansion algorithm used in Darmok, where p is the current plan and S is the current game state.

```
Function PlanExecution(p, S)
    p = startReadySnippets(p,S)
    p = sendActionsToExecution(p,S)
    p = updateSnippetStatus(p,S)
    p = updateActionStatus(p,S)
    Return p
EndFunction
```

Figure 10: Plan execution algorithm used in Darmok, where p is the current plan and S is the current game state.

plan retrieval module retrieved it, the snippet is handed back to the plan adaptation module to make sure that the plan is adequate for the current game state.

- *sendActionsToExecution*: If any of the executing snippets have any basic actions, and those actions have all its preconditions satisfied, then they are sent to WARGUS to be executed. If the preconditions of the actions are not satisfied, the snippet is sent back to the plan adaptation module to see if the plan can be repaired. If after a certain amount of time t_1 (set to 2000 game cycles in our experiments) it cannot, then the snippet is marked as failed, and thus its corresponding goal is open again (thus, the system will have to find another plan for that goal).

- *updateSnippetStatus*: The execution module periodically evaluates the alive conditions and success conditions of each snippet. If the alive conditions of an executing snippet are not satisfied, the snippet is marked as failed, and its goal is open again. If the success conditions of a snippet are satisfied, the snippet is marked as succeeded.

- *updateActionStatus*: Whenever a basic action succeeds or fails, the execution module updates the status of the snippet that contained it. When a basic action succeeds, the executing snippet can continue to the next step. When a basic action fails, the snippet is marked as failed, and thus its corresponding goal is open again.

The next section focuses on the plan adaptation algorithms.

10 Plan Adaptation

Plan adaptation can be seen as two different subprocesses: *parameter adaptation* and the *structural plan adaptation*. The former is in charge of adapting the parameters of the basic actions, e.g. the coordinates and specific units that will perform the actions, and the latter is in charge of adapting the structure of a plan, e.g. inserting or removing actions in a plan. Some previous systems, such as PRIAR [21] combine these two processes in a single one by performing search in the space of plans. However, in order to perform plan adaptation efficiently, and on-line, we propose to break adaptation into two processes, since efficient adaptation techniques can be designed if they are treated separately.

10.1 Parameter Adaptation

In Darmok, the parameter adaptation process consists of a series of rules that are applied to each one of the actions of a snippet 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 snippet shown in Figure 7 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 (*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). Since the map in WARGUS is represented as a grid, in order to adapt the coordinates, the parameter adaptation module gets, from the case, how the map in the particular coordinates looked like by retrieving the content of the map in a 7x7 window surrounding the specified coordinates. Then, it

looks in the current map for a spot that is the most similar to that 7x7 window, and uses those coordinates.

Notice that parameter adaptation is a difficult problem if we want to solve it optimally, since it involves computing the optimal placement of buildings, attacking locations, and so on. For example, if the expert placed a tower in a "choke point", Darmok has to understand that, and also select a "choke point" in the current game as the placement of the tower (detection of choke points is a computationally expensive problem), and not just any choke point, but one that serves the same function as the one in the original map. However, these 2 simple rules are enough for Darmok to adapt actions between similar maps with an acceptable accuracy, using the parameters that the expert demonstrated as guidelines.

10.2 Structural Plan Adaptation

As said in Section 6, in our formalism, plans are composed of four basic types of elements: primitive actions; parallel plans which consist of component plans which can be executed in parallel; sequential plans which consist of component plans which need to be executed in sequence; and subgoals which require further expansion. Darmok's structural plan adaptation module specifically considers plans which are only composed of actions, sequential plans and parallel plans. This implies that only those plans which are completely expanded are considered. We do this because the plan adaptation technique we propose for on-line plan adaptation relies on constructing a *plan dependency graph* among the basic actions that compose a plan. In order to generate such a graph, a plan must be fully expanded.

10.2.1 Plan Dependency Graph Generation

A *Plan Dependency Graph* G is a directed graph where each node in the graph is a primitive action and each link (p_i, p_j) indicates that p_j depends on p_i. For simplicity, in the remainder of this section we will represent this graph as a set of links.

Figure 11 shows the algorithm for plan dependency graph generation. Each action within a plan has a set of preconditions and a set of success conditions. The plan dependency graph generator analyzes the preconditions of each of these primitive actions. Let p' be an action in the plan which contributes to satisfying the preconditions of another action p. Then, a directed edge from p' to p is formed (function *FindDependencies*, shown in Figure 11). This directed edge can be considered as a dependency between p' and p. Here, we assume that actions in different parts of a parallel plan are independent of each other (a strong assumption, subject to improvement in future work). A pair of actions might have a dependency between them only if their closest common parent is a sequential plan. This is what is effectively done by using the set of actions D, in Figure 11. For any action p' when the function *FindDependencies* is called,

```
Function GeneratePlanDependencyGraph(p, D)
    G = ∅
    ForEach p′ ∈ p.subPlans
        If p′ is sequential or parallel Then
            G = G ∪ GeneratePlanDependencyGraph(p′, D)
        ElseIf p′ is a primitive action Then
            G = G ∪ FindDependencies(p′, D)
        EndIf
        If p is sequential Then D := D ∪ p′.allPrimitiveActions
    EndForEach
    Return G
EndFunction

Function FindDependencies(p, D)
    G = ∅
    ForEach p′ ∈ D
        If p′ statisfied any condition of p Then
            G = G ∪ (p′, p)
        EndIf
    EndForEach
    Return G
EndFunction
```

Figure 11: Algorithm for Plan Dependency Graph Generation. Where p is the plan to be adapted, and D is the set of actions on which any sub-plan in p might depend (and it's equal to $\emptyset$ in the first call to the algorithm). $p.subPlans$ refers to the set of sub-plans directly inside p in case p is sequential or parallel. And $p.allPrimitiveActions$ refers to all the primitive actions inside p or in any sub-plan inside p.

D contains exactly the set of actions on which p' might be dependent. The set of primitive actions contained in a subplan p' of p are added to D only if p is a sequential construct. The recursive call to *GeneratePlanDependencyGraph* ensures that nested parallel and sequential constructs can be processed. This process results in the formation of a plan dependency graph G with directed edges between actions that have dependencies.

A challenge in our work is that simple comparison of preconditions of an action p with success conditions of another action p' is not sufficient to determine whether p' contributes to achievement of preconditions of p. This is because there isn't necessarily a direct correspondence between preconditions and success conditions. An example is the case where p has a precondition testing the existence of a single unit of a particular type. p' may have a success condition testing the existence of a given number of units of the same type. Another

example is with attacking: the success condition of a goal might specify that a particular enemy unit has to be killed, but the attack actions have no postcondition named "killed", since we cannot guarantee that an attack will succeed (the success condition of the attack action is that a particular unit will be in the "attacking status").

For that purpose, the plan graph generation component needs a precondition-success condition matcher (*ps-matcher*). In our system, we have developed a simple rule-based ps-matcher that incorporates a collection of rules for the appropriate condition matching. For example, our system has six different conditions which test the existence of units or unit types. Thus the ps-matcher has rules that specify which conditions can be matched. In some cases it is not clear whether a relation exists or not. However it is necessary for our system to capture all of the dependencies, even if some non-existing dependencies are included. If a dependency was not detected by our system, a necessary action in the plan might get deleted. However, if our system adds extra dependencies that do not exist the only thing that can happen is that the system ends up executing some extra actions that would not be required. Clearly, executing extra actions is better than missing some needed actions (notice that by "extra actions" we don't mean that the adaptation module will insert extra actions, but that some actions that were already in the retrieved snippet won't be removed when they could have been removed).

10.2.2 Removal of unnecessary actions

Figure 12 shows the algorithm for the removal of unnecessary or redundant actions. Every plan p has a root node that is always a goal g. The removal of unnecessary actions begins by taking the success conditions of the goal g and finding out which of the actions in the plan contribute to the achievement of those conditions. These actions are called *direct actions* for the subgoal, and are obtained by the function *GetDirectActions* in Figure 12. Then the plan dependency graph for p is generated using the *GeneratePlanDependencyGraph* function in Figure 11. The algorithm works by maintaining a set of *active actions* A. At the end of the algorithm, all the actions not in A will be removed from the plan. The removal of actions proceeds using the plan dependency graph and the set of direct actions, B. The success conditions of each action in B are evaluated for the game state at that point of execution. Each of these actions p with unsatisfied success conditions is added to the list of active actions. The set of actions B' on which the action p has a dependency according to the dependency graph G are recursively checked to see if they have to be activated. Such actions are obtained using the function *GetParentActions* in the algorithm (that can be implemented to have constant time). The result of this process is a set A of actions whose success conditions are not satisfied in the given game state and which have a dependency to a direct action, which also has success conditions not satisfied in the given game state. Actions that are not active (not in A) are removed from the plan.

```
Function RemoveRedundantPlans (p, g)
    B = GetDirectActions(p, g)
    G = GeneratePlanDependencyGraph(p, ∅)
    A = BackPropagateActivePlans(B, G, ∅)
    remove from p all the actions not in A
    Return p
EndFunction

Function BackPropagateActivePlans (B, G, A)
    ForEach p ∈ B
        If p's success conditions are not satisfied Then
            A = A ∪ {p}
            B' = GetParentACtions(p, G)
            A = BackPropagateActivePlans(B', G, A)
        EndIf
    EndForEach
    Return A
EndFunction
```

Figure 12: Algorithm for Removal of Unnecessary Actions. Where p is the plan to be adapted, and g is the goal corresponding to p. GetParentACtions(p, G) is a simple function that returns all the actions that have a causal direction with a given action p, according to a plan dependency graph G. GetDirectActions(p, g) is a function that returns those actions in p that are *direct actions*.

10.2.3 Adaptation for unsatisfied preconditions

Figure 13 shows the algorithm for adaptation for unsatisfied preconditions. If the execution of an action fails because one or more of its preconditions are not satisfied, the system needs to act so that the execution of the plan can proceed. To do this, each unsatisfied condition is associated with a corresponding *satisfying goal*. A satisfying goal is such that when a plan to achieve the goal is retrieved and executed, the success of the plan implies that the failed precondition is satisfied. Thus, for every possible condition type that might fail, the adaptation component must know how to define a corresponding satisfying goal (in Darmok, this is part of the domain knowledge).

Initially, all the unsatisfied preconditions of the plan p to adapt are computed, resulting in a set C. For each condition $c \in C$, a satisfying goal is obtained, using the function *GetSatisfyingGoal*. This gives a set of goals G which need to be achieved before the action p can be executed. A parallel plan q is generated where each of the goals in G can be achieved in parallel. q is inserted as the first step of plan p.

After the modified plan is handed back to the plan execution module, it is inserted into the current plan. In the next execution cycle the plan expansion

```
Function AdaptForUnsatisfiedConditions(p)
    C = GetUnsatisfiedPreconditions(p)
    G = ∅
    ForEach c ∈ C
        G = G ∪ GetSatisfyingGoal(c)
    EndForEach
    Initialize q as an empty parallel plan
    ForEach g ∈ G
        add SubGoalPlan(g) to q
    EndForEach
    insert q at the beginning of p
    Return p
EndFunction
```

Figure 13: Algorithm for Adding Goals for Unsatisfied Preconditions, where p is the primary action to be adapted. GetUnsatisfiedConditions(p) is a function which returns the set of those preconditions of p which are not satisfied. GetSatisfyingGoal(c) is a function which returns a goal whose success satisfies the condition c. SubGoalPlan(g) is a function which returns a sub-goal plan with goal g.

module will expand the newly inserted goals in G.

Notice that the structural plan adaptation module performs two basic operations: delete unnecessary actions (which is performed by an analysis of the plan dependency graph), and insert additional actions needed to satisfy unsatisfied preconditions. This last process is performed as a collaboration between several modules: the plan execution module identifies actions that cannot be executed, the adaptation component identifies the failed preconditions and generates goals for them, and the plan expansion and plan retrieval modules expand the inserted goals.

11 Experimental results

In order to evaluate our on-line case-based planning techniques, we implemented them in Darmok and evaluated them in a suite composed of 12 Wargus maps. Eleven of them are different variations of the map known as "No where to run, nowhere to hide" (NWRT1 to NWTR11) and one of them is a variation of the map known as "Garden of War" (GoW1). Both NWTR and GoW are well known maps from *battlenet*, popular among human players. Figures 14 and 15 show the NWTR1 and GoW1 maps respectively. We selected these two kinds of maps because we wanted maps with a high strategic component, i.e. maps where a good strategy is the key for winning, rather than micro-management. NWTR maps feature a wall of trees that separates the two players,

Figure 14: “No where to run, no where to hide” map.

and that feature allows for multiple interesting strategies as we mentioned in Section 4. Thus, NWTR is interesting strategy-wise. GoW, on the other hand, represents the standard Wargus maps: large terrain with trees and open areas, the enemy is far away at the start, etc. However, it also has an interesting strategic component: there are two unclaimed gold mines in the center of the map. The first player to claim them has a huge advantage.

Moreover, the different variations of each map are key to the strategy to be applied in that map. Some of the variations in the NWTR maps include changing the width and shape of the wall of trees (that makes some of the strategies useless), having a hole in the barrier, changing the size of the map, or even partially substituting the wall of trees by a rock-wall. These are key modifications that the system has to be aware of in order to pick the appropriate strategy.

In order to evaluate our system, we generated 10 demonstrations in some of the maps. Each demonstration shows a different strategy: rushing, towering, ballistae attack, air-attack, and variations. To generate a demonstration, a member of our team played the game using a particular strategy, and then annotated each action with the goals being pursued at each point. In average, the time required to generate a demonstration is less than 30 minutes: between 10 to 15 minutes to play a game (in the maps we use for our experiments), and about 10 minutes to annotate a trace with a proper trace annotation tool (plus 5 minutes of time spent in launching the game, opening the annotation tool, saving the files in the proper folders, etc.). Thus, generating a demonstration

Figure 15: “Garden of War” map.

is a very easy process. The 12 maps plus 10 demonstrations constitute our experimental setup.

Table 2 summarizes the results of our experiments. Each row of the table shows the results of the system playing from a different number of expert traces. In the first row we show the performance of the system learning from one expert demonstration, the second from two expert demonstrations and so on. For each row, we run the system in the 12 maps four times (a total of 48 games per row). Each row is split in two parts: results with structural plan adaptation on and with structural plan adaptation off. The number of wins, draws, and losses (W, D, and L) are shown. At the end of each row we emphasize the percentage of wins (WP). Finally, the bottom row summarizes the results. Let us analyze the results.

The first result we observe in Table 2 is that learning from 10 expert demonstrations and turning on structural plan adaptation the system wins about 41% of the times against the built-in AI. This number might not look impressive, but we have to take into account that Darmok is playing the game at the same level of detail that a human would (i.e. it takes every single decision in the game), and that Darmok barely has any domain knowledge built in (only the two simple coordinate and unit adaptation rules and the set of features used for case retrieval). Looking at the numbers it might seem that Darmok achieves a similar level of play to the built-in AI, however a more detailed analysis of the behavior of Darmok is needed to understand the results.

There are several strengths and weaknesses between Darmok and scripted AI techniques such as the built-in AI of Wargus. The main strength of the built-in AI are its tactics. The built-in AI has a very fast and reactive low-level unit control loop that makes sure that units are attacking the appropriate enemies and that attacked units are defended. In contrast, the built-in AI has

Table 2: Performance of Darmok with and without structural plan adaptation playing against the built-in AI of WARGUS.

	Adaptation				No Adaptation			
NT	*W*	*D*	*L*	*WP*	*W*	*D*	*L*	*WP*
1	17	4	27	**35.42%**	9	7	32	**18.75%**
2	16	5	27	**33.33%**	15	2	31	**31.25%**
3	18	6	24	**37.5%**	10	6	32	**20.83%**
4	19	3	26	**39.58%**	8	4	36	**16.67%**
5	11	6	31	**22.91%**	7	6	35	**14.58%**
6	14	2	32	**29.17%**	3	5	40	**6.25%**
7	20	0	28	**41.67%**	9	6	33	**18.75%**
8	15	3	30	**31.25%**	6	3	39	**12.50%**
9	21	4	23	**43.75%**	7	3	38	**14.58%**
10	20	0	28	**41.67%**	2	0	10	**16.67%**
	171	33	276	35.63%	82	42	356	17.08%

very limited strategic abilities. It uses a very similar strategy for every single map (since it just executes a script) that is not adaptive. Darmok, on the other hand, has a very poor low-level tactical unit control (that could be improved by making Darmok sit on top of a scripted lower level tactical layer), and thus often loses games due to poor low level unit control (in order to win a battle, Darmok usually needs double the number of units over the built-in AI). However, Darmok compensates that with a much stronger strategic ability. Darmok can plan ahead and retrieve strategies observed by humans, it can attempt different strategies in the same map when other strategies fail, etc. For instance, Darmok uses ranged attacks when there is a wall of trees separating the enemy from our base, and attempts a rush when the wall does not exist. However, the main aim of Darmok is to capture strategic knowledge of humans and reuse it using case-based planning. Therefore, we consider Darmok a success in the sense that only using strategic knowledge it is able to make up for poor tactics and win 41% of the games.

Let us analyze the results in Table 2 in more detail. If we compare the results obtained with and without structural adaptation, we can see that structural plan adaptation multiplies by a factor of two the average number of wins of Darmok: 35.63% vs 17.08% in average, and 41.67% versus 16.67% in the scenario with 10 expert demonstrations. Moreover, this difference increases with the number of traces that Darmok learns from. When Darmok learns from a few expert demonstrations, the gains achieved by structural plan adaptation are smaller. This can be clearly seen in Figure 16, which shows a visualization of the percentage of wins with and without adaptation as the number of demonstrations that the system learns from increases. The effect can be explained by the fact that when Darmok has learned from a few demonstrations, it is very

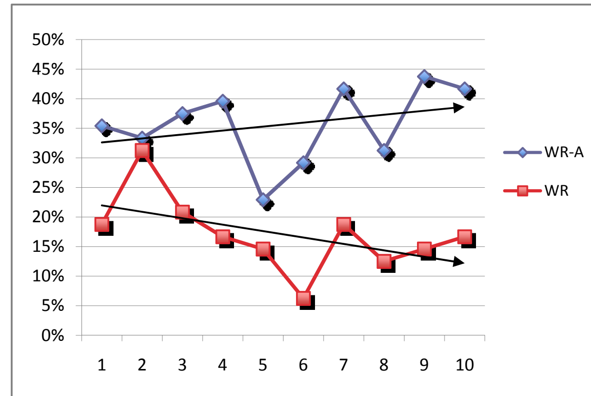

Figure 16: Average number of wins with (WR-A) and without adaptation (WR) versus the number of expert traces observed by Darmok.

likely that individual snippets will fit together, however, when Darmok learns cases from different demonstrations, these cases are unlikely to fit together, and they will need adaptation. Thus, adaptation provides Darmok with a way to glue different plan snippets learned from different demonstrations into a sound strategy.

We can also see that the performance of the system varies as Darmok observes more expert demonstrations. Specifically, the performance goes up when there is plan adaptation, and down when there is not. Performance goes down to 16.67% of wins with no adaptation and 10 expert demonstrations and up to 41.67% with adaptation. This result can be explained by understanding that the performance of Darmok depends on a number of factors. Two of those factors are: the selection of proper plan snippets to achieve goals, and the ability to glue different plan snippets together into a single plan. When the system has few expert demonstrations, selecting which plan snippet to use is an easy task, since there's only a few of them, and splicing together plan snippets is an easy task, since all of them come from the same expert demonstration. However, when there are more expert demonstrations, the problem of selecting plan snippets and splicing them together becomes harder. Plan adaptation can solve the second problem, but not the first. As an overall effect, the performance of Darmok goes slightly up with more expert demonstrations. However, the expected effect would have been the performance to increase more than observed. The explanation for this (after a careful observation of the replays of the games) is that the set of features that Darmok uses to retrieve plan snippets is not good enough. Thus, the selection of an appropriate strategy for the map is not excellent. Darmok still does a good job putting together different plan snippets into a meaningful plan, but it might not be the best for a given map. The conclusion is that in order to boost the performance of Darmok, a better subset of features has to be found (for instance, we observed that a key feature in the maps we

Table 3: Performance of Darmok in different maps using different expert traces.

map	T1a	T1b	T1c	T3	T6	T7	T10	T11	TGa	TGb
NWTR1	0	1	1	0	1	0.5	0	1	0	0
NWTR2	1	1	0	0	0	0.5	0	0	0	0
NWTR3	0	1	0	1	0	0.5	0	0	0	0
NWTR4	0	0	1	0	1	0.5	0	0	0	0
NWTR5	0	0	1	0	0	0	0	0	1	0
NWTR6	0	1	0	1	1	0.5	0	0.5	0	0
NWTR7	1	1	1	1	0	0	0	1	0	0
NWTR8	1	1	1	1	0	1	0	0	0	0
NWTR9	0	1	0	1	0	0	0	0	0	0
NWTR10	0	0	0	0	0	0	0	0	0	0
NWTR11	0	0	0	0	0	0	0	0	0	0
GoW	0	0	0	0	0	0	0	0	1	0
average	0.25	0.58	0.42	0.42	0.25	0.29	0.00	0.20	0.17	0.00

used is whether there is a path between Darmok's base and the enemy base, if we add that feature to the cases, the performance would greatly increase). As mentioned in the next section, we are already performing experiments to solve that issue, and in early runs we have observed the performance of Darmok to multiply by a factor two when the appropriate features are used for case retrieval.

In order to have a better insight on the performance results presented before, we performed a more detailed analysis by running Darmok with each possible combination of expert trace and map we had. That is, we run the system learning from each one of the 10 expert traces we have (one at a time) and we made it play each of the 12 maps we had. The results are presented in Table 3, that shows maps in the vertical axis, versus traces in the horizontal axis. Each cell contains a 1 if the game was won, 0 if it was lost, and 0.5 if it was drawn. The bottom row shows an average score of each trace (1.00 would mean that the trace wins all the maps, and 0.00 that it loses all the maps). The name of the traces represent the map in which they were demonstrated, so traces T1a, T1b and T1c where all played in map NWTR1, T3 in NWTR3, etc. The traces labeled as TGa and TGb were demonstrated in the GoW map.

The first thing that can be observed in Table 3 is that some traces are clearly better than others. For instance, while trace T1b (that demonstrates a strategy that combines towers and ballistas) wins on 58% of the maps (i.e. average score 0.58), trace TGb (that encodes a quick footmen rush) does not win in any map. Thus, it is clear that the performance of Darmok depends on the quality of the traces presented to it. It is interesting to see that some strategies do not win even in the maps they were demonstrated. For example, TGb loses in the GoW map. This is so, since the strategy demonstrated in TGb is very similar to the one the built-in AI uses (which relies on the simple idea of sending units quickly to the opponent to outnumber him), thus, it comes down to whoever has the better low level unit control (in which Darmok performs poorly, as mentioned earlier). The same situation happens with trace T10 (which also encodes a kind of rushing strategy).

The previous explanation can be applied to explain why Darmok cannot win any game in the maps NWTR10 and NWTR11. Those maps have a hole in the wall of trees, and thus require a totally different strategy than the rest. However, the maps are small enough, so that a long-term rush (such as the "knights rush" demonstrated in trace TGa) does not work either. This leads again to a low-level control battle, in which Darmok is not good. Moreover, when Darmok plays learning from all the expert traces, we saw that it managed to win some games in maps NWTR10 and NWTR11, thus showing that Darmok is able to combine different demonstrations to come up with new strategies.

If we analyze the matrix in Table 3, we can see that, when learning from a single expert trace, Darmok wins 50.00% of the times when it plays in the same map where the trace was demonstrated. Whereas it wins only 25.83% of the time in a different map. If we analyze how many times Darmok wins in a "similar" map (assuming NWTR maps are similar among each other, and GoW is different) than the one in the trace, then Darmok wins 41.47% of the times. Thus, clearly, the similarity between the map in which Darmok plays and the one for which it has a trace plays a key role in its performance.

Finally, we performed an additional experiment where we selected by hand 3 expert traces that we thought would work together: T1b, T1c and TGa and trained Darmok with those traces. The result is that Darmok managed to win 66.67% of the games (a higher percentage than when Darmok learns from all 10 traces). The explanation is that the current version of Darmok still does not learn from experience (i.e. episode retention is switched off). Therefore, case retrieval is done purely based on goal and game state similarity (without taking into account previous experiences of applying different strategies to different maps). Because of this T1a and T1b are almost indistinguishable for Darmok, and thus, when the map being played looks like NWTR1 (where those two traces were demonstrated), Darmok selects at random the strategy in T1a or the one in T1b (when clearly T1b is a better strategy). Therefore, when Darmok contains in the case base good and bad strategies mixed, and that were demonstrated in similar game states, it cannot properly select which is the right one to apply. Both learning preconditions for plans and learning from experience should help solving this problem, which are part of our future work.

12 Related Work

Several areas are related to our work. In particular case-based planning, plan reuse, the relation of planning and execution, and the problem of learning plans from expert traces.

Concerning case-based planning, we presented a small overview in Section 2.1, but see Spalazzi [42] for a more comprehensive review. However there are several systems related to Darmok that are worth mentioning in this section. The TOLTEC planner [44] is a case-based planner inspired on the *Dynamic Memory* [40] theory by Roger Schank. As such, TOLTEC stores previous planning experiences as scenes, scripts, MOPs (Memory Organization Packages) and

TOPs (Thematic Organization Packages). TOLTEC recursively refines the plan it is generating by retrieving new cases and inserting them into the plan. It uses MOPs to predict previous failures during planning, and thus select the appropriate expansions. Moreover, TOLTEC can also learn "soft constraints" that reflect user preferences. Darmok performs a similar plan expansion to the one presented by TOLTEC. The main difference being that Darmok uses a more advanced plan adaptation, and can backtrack its decisions (TOLTEC performs no backtracking) based on execution failures.

CaPER [22] is a case-based planning system that focuses on improving the plan retrieval phase. Previous CBP systems used indexing techniques to store cases in the case-base. Kettler et al. argue that indexing techniques might limit the ways in which cases can be retrieved. For example, they argue that in CHEF cases are not indexed by time; thus CHEF cannot understand queries of the type "I want a recipe that takes less than 10 minutes to cook". CaPER [22] uses a frame-based representation for the cases, and cases can be accessed by any concept or relation in them. However, these generic techniques for retrieval have a high computational cost. To solve the potential computational cost of this operation, they propose to use parallel computing to increase the efficiency of retrieval. Darmok could greatly benefit from such complex case representation, although the high computational cost makes them not suitable for real-time domains.

In Section 2.1 we presented the PRIAR [21] system (based in NONLIN), a planner that was designed to exploit plan reuse. In the same line of PRIAR, the SPA (Systematic Plan Adaptor) system by Hanks and Weld [18] is also based on NONLIN. The key highlights of SPA is that it is complete and systematic, while PRIAR is complete but not systematic and CHEF is neither complete nor systematic (CHEF uses a set of heuristic rules for adaptation that are very efficient but not complete).

Extending SPA, Ram and Francis [38] present MPA (Multi-Plan Adaptor), that extends SPA with three mechanisms: a *goal deriver* that extracts goal statements, a *plan clipper* that prepares plans for merging and a *plan splicer* that can merge two plans. MPA's main capability is the ability to combine several plans into one by splicing them into pieces and merging them. The main difference between PRIAR, SPA and MPA and Darmok is that they are off-line planners, while Darmok is an on-line system that plans and executes plans in real time. Additionally, neither PRIAR, SPA nor MPA address the problem of plan learning, or plan retrieval (MPA was connected to the NICOLE system for retrieval [38])

Another relevant work is the logical treatment of plan reuse by Koehler [23] (they refer to it as "planning from second principles") where they define a four stage process, analogous to the CBR process, consisting of: determination (retrieve a plan specification from the plan library), interpretation (analyze the retrieved plan in terms of the new goal), refitting (adapt the plan using a planner), update (incorporate the new plan into the plan library). However, Koehler only focuses in off-line plan reuse, not addressing plan learning, retrieval or on-line planning.

Concerning the relationship between planning and execution. An early formalization of this problem is McDermott's NASL [30], focusing on how planning and execution are interleaved, and how errors in both planning and execution affect the process. An important consideration is that if there is a model of the world, and the "state of the world" is thought of as an internal data structure, then search can be done. Otherwise, it cannot be done. Since advancing in a branch of the search tree changes the state, and it is not possible to go back to a previous state in order to do backtracking. Another important consideration introduced by McDermott is the differentiation between *planning failures* and *execution failures*, that we will also make in our system.

The IPEM system (Integrated Planning, Execution and Monitoring) [3] by Ambros and Steel, is another framework for integrated planning and execution. In IPEM, execution is delegated to a scheduler, that considers decisions such as "when to start an action" part of the normal scheduling process. One interesting characteristic of IPEM is that it does not assume that the planner knows the complete state of the world, and can deal with "information gathering actions". Moreover, IPEM assumes a classic STRIPS planning representation framework with preconditions and postconditions of operators, that is not adequate for domains such as RTS games.

Finally, concerning the problem of learning cases from traces, the work on learning in HTN planning and learning macro operators is specially relevant. Hogg et al. [19] present the HTN-MAKER algorithm, that given a set of solved planning problems and a set of tasks is able to generate HTN methods for them. Notice that Darmok's snippets are very similar to HTN methods, and thus an HTN-MAKER-like algorithm could be used to improve Darmok's snippet learning method (allowing Darmok to also learn the preconditions of snippets), which is part of our future work.

Könik and Laird present a *Relational Learning from Observation* technique [25] able to learn how to decompose a goal into subgoals based on observing annotated expert traces (with the same annotation schema used by Darmok). Könik and Laird's technique uses relational machine learning techniques to learn how to decompose goals, and the output is a collection of rules (that would be executed by the SOAR [26] architecture). The main differences between Könik and Laird's technique and Darmok's learning methods are that: a) Darmok uses lazy learning to decide which snippet to use for each goal (it delegates the decision to problem solving time), while Könik and Laird's technique does it using induction during learning time, generating a set of rules. Also, Darmok learns sequences of actions in snippets, while Könik and Laird's technique learn srules that spawn sub-goals or individual actions one by one when their head is satisfied by the current state of the world.

Other work on learning macro operators focuses on speeding up planning: assuming that we have a system that can already solve a problem, the goal is to learn macros that can be used at planning time to reduce the amount of search needed. For example, Langley and Choi [28] propose to learn such macros as *teleoreactive logic programs* that can be run by the ICARUS architecture [10]. They propose to store a new macro every time the system performs a search

process to solve a new problem containing the resulting search. They call these macros "skills" (in opposition to "primitive-skills"). Botea et al. [5] propose to learn macro operators as a way to speed up planners by both analyzing the domain description and also analyzing solutions to sample problem instances. Both approaches differ from Darmok's learning process in that they assume that there is an existing system that can solve problems from scratch even if macros are not present (the goal is only speed-up). The only way Darmok can solve problems is by retrieving and adapting snippets, so without snippets, Darmok cannot solve any problem, and thus these techniques are not directly applicable to Darmok unless a generative planner is added to the architecture as a back-up for when no snippet is available.

Most of the work on case-based planning does not focus on how to learn the cases in the case base. So for instance, systems like MAYOR or CHEF use a predefined case-base, while Darmok attempts to solve the case acquisition problem.

13 Conclusions and Future Work

In this paper we have proposed a novel architecture for on-line case-based planning, captured in the OLCBP cycle. In order to illustrate the proposed OLCBP cycle, we presented a system, Darmok, that implements such a cycle. Darmok implements several key ideas. First, the case acquisition problem is solved by observing human demonstrations, from which the case-base is populated. Second, Darmok contains a heterogeneous case-base where cases are indexed both by the goal they achieve and by the particular context in which they are applicable. Third, Darmok interleaves planning and execution on-line in order to achieve efficient on-line planning that is reactive to the current situation. Fourth, we introduced the idea of *delayed adaptation* where the adaptation of plan snippets retrieved from the case-base is delayed until they have to be executed, to ensure they are adapted with the most up-to-date information from the domain. Finally, Darmok implements plan adaptation techniques that are efficient to be executed on-line. Darmok also has learning-from-experience capabilities (it is able to retain new episodes from experience), but the evaluation of this capability will be subject of future work.

In our experimental evaluation, we compared the performance of Darmok against the built-in AI of Wargus. That evaluation pointed out which are the strengths and weaknesses of the system. First of all, we can conclude that the on-line case-based planning approach of Darmok succeeds, since the system is able to play the game in real-time (it can actually play the game more than 10 times the default speed without performance loss) while taking every single decision in the game. We saw that by observing some human demonstrations, the performance of Darmok can easily reach that of the built-in AI of Wargus. The merit being that Darmok "learned" and "planned" how to play Wargus by itself, while the built-in AI is hard coded and fixed (it is actually the responsibility of the author of a particular map to make sure that the built-in AI will

not execute non-sensical actions in that map, because it simply executes a fixed script, selected while authoring a map, while Darmok will select an adequate strategy by itself). The results showed that Darmok has very strong strategic abilities, but is weak at the lower level tactical control. This is in contrast with other systems presented in the literature. For instance, the case-based tactician (CaT) [2] or CARL [41] use hand-coded scripts as a lower level layer to execute the low level unit control, and thus they are expected to have better tactical control than Darmok. However, such low level layer could easily be integrated in Darmok too. Moreover, one of Darmok's strengths is the ability to train the system with virtually no effort (by just exposing the system to some annotated expert demonstrations).

Finally, we'd like to comment on the applicability of the techniques presented in this paper to domains other than WARGUS. Darmok itself could be adapted with few changes to play any RTS game given that the adequate knowledge structures are defined (goals, actions, etc.). Moreover, the same ideas should be applicable to any strategic real-time task for which expert demonstrations can be easily captured and annotated. Moreover, since Darmok's planner performs a very limited amount of backtracking (only when plans fail during execution), Darmok won't be suitable for domains where the individual snippets that compose a plan interact in ways that force the system to perform extensive search to find the right combinations. Darmok assumes that the retrieval mechanism will retrieve a good enough snippet, and that the adaptation module can solve the remaining conflicts of the retrieved snippet (since it was designed to run under tight real-time constraints). Another characteristic of the techniques presented in this paper is that they are aimed at domains that are big enough so that finding optimal plans is not feasible, and the only aim is to obtain plans that are good enough.

13.1 Future work

Darmok provides us with an excellent platform in which to develop future lines of research. Specifically, we would like to develop case-based planning algorithms that can plan with more general plan representations. For instance, the current plan representation used by Darmok cannot represent conditional plans or loops. We are currently designing improved learning and planning modules that can learn generic petri-nets from human demonstrations, being able to capture richer behaviors than the current representation used by Darmok.

Learning from demonstrations gives Darmok the ability to easily incorporate domain knowledge. Currently, Darmok requires the demonstrations to be annotated. We plan to investigate plan recognition techniques applied to the case extraction process to avoid the annotation step. This will allow the system to learn from merely observing humans playing games (i.e. Darmok could potentially be observing on-line games and accumulating cases continuously without any authoring effort). Also related to learning, we want to explore the learning from experience capabilities of the Darmok architecture.

The selection of the appropriate plan snippets is a key step in Darmok.

As the experimental evaluation showed, the set of features used currently in Darmok (that was never fine tuned) is not good enough. In case-based planning systems, case retrieval has to take into account three factors: if the retrieved case will satisfy the goal at hand (goal similarity), if it is adequate for the current situation (game state similarity), and if the case can be adapted successfully (adaptation cost). Currently Darmok only takes into account the first two factors. We envision two lines of future work to improve this process: first, there is existing research in case retrieval for case-based planning using the adaptation cost as a guideline, we would like to incorporate those techniques into our system. To achieve that goal, the learning system of Darmok will have to be enhanced in order to be able to learn preconditions, since they can be used to estimate the adaptation cost. We are currently investigating similar techniques to the ones reported in [19] to achieve this goal. Second, we are exploring situation assessment techniques that will perform retrieval as a two step process. During the first step the system will try to figure out in which "situation" it is in (attacking, defending, etc.), and then from the situation, an extended set of relevant features will be selected with which to perform the final case selection (early experiments with this technique show a two fold increase in performance with a limited computational cost increase).

The planning system will also be subject of future research. Merging multiple plan snippets into a single one, in order to achieve a goal for which we had no good plan in the library using techniques similar to those ones in MPA [38] or considering scheduling [16] are two of the techniques we want to explore. Scheduling is important in RTS games, since the timing in which actions are performed might have a big influence in the final result. In several games we observed that Darmok lost a game because of simple bad timing (sending attack actions one by one instead of several in a block, or sub-optimal coordination of parallel plans are some examples).

Another aspect that we want to improve in our Darmok architecture is the reactivity. Currently Darmok blindly executes a plan as long as the preconditions and postconditions of all the goals, sub-goals and primitive actions in the plans are being satisfied. However, in RTS games, is is very important to observe the opponent actions to be able to react *before* some part of our plan fails. We are envisioning a simulator module inside the architecture that will allow the system to estimate what will be the outcome of applying a particular plan, so that the failures can be detected beforehand. Such simulator requires Darmok to incorporate world-model learning and opponent model techniques, that will automatically learn an internal model of the domain that Darmok can use to improve its planning abilities.

Acknowledgements The authors would like to thank Kane Bonnette, for the java-WARGUS interface; DARPA for their funding under the Integrated Learning program TT0687481; and the anonymous reviewers for their suggestions that contributed to increase the quality of this paper.

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] Joseé A. Ambros-Inerson and Sam Steel. Integrating planning, execution and monitoring. In *AAAI'88*, pages 83–88, 1988.

[4] Avrim Blum and John Langford. Probabilistic planning in the graphplan framework. In *ECP*, pages 319–332, 1999.

[5] A. Botea, M. Enzenberger, M. Mueller, and J. Schaeffer. Macro-ff: Improving ai planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24:581–621, 2005.

[6] Rodney A. Brooks. A robust layered control system for a mobile robot. Technical Report 864, MIT/AIL, 1985.

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

[8] Amedeo Cesta and Ciovanni Romano. Using abstraction-based similarity to retrieve reuse candidates. In *Proceedings of the first international conference on Artificial intelligence planning systems, AIPS 92*, pages 269–270, 1992.

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

[10] Dongkyu Choi, Matt Kaufman, Pat Langley, Negin Nejati, and Daniel Shapiro. An architecture for persistent reactive behavior. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 988–995, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning: A detailed analysis. Technical Report CS-TR-2797, 1991.

[12] Mark Fasciano. Everyday-world plan use. Technical Report TR-96-07, University of Chicago, 1996.

[13] 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.

[14] R. James Firby. An investigation into reactive planning in complex domains. In *AAAI-87*, pages 202–206, 1987.

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

[16] Melinda Gervasio and Gerald Dejong. Completable scheduling: An integrated approach to planning and scheduling. In *AAAI 1992 Spring Symposium on Practical Approaches to Scheduling and Planning*, pages 122–126, 1992.

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

[18] Steve Hanks and Daniel S. Weld. A domain-independednt algorithm for plan adaptation. *Journal of Artificial Intelligence Research*, 2:319–360, 1995.

[19] Char M. Hogg, Héctor Muñoz-Avila, and Ugur Kuter. Htn-maker: Learning htns with minimal additional knowledge engineering required. In *AAAI-2008*, 2008.

[20] Leslie Pack KAelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.

[21] Subbarao Kambhampati and James A. Hendler. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, 55(2):193–258, 1992.

[22] B. P. Kettler, J. A. Hendler, A. W. Andersen, and M. P. Evett. Massively parallel support for case-based planning. *IEEE Expert*, pages 8–14, 1994.

[23] Jana Koehler. Towards a logical treatment of plan reuse. In *Proceedings of the first international conference on Artificial intelligence planning systems, AIPS 92*, pages 285 – 286, 1992.

[24] Janet Kolodner. *Case-based reasoning.* Morgan Kaufmann, 1993.

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

[26] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: an architecture for general intelligence. *Artif. Intell.*, 33(1):1–64, 1987.

[27] John E. Laird and Michael van Lent. Human-level AI's killer application: Interactive computer games. In *AAAI 2000*, pages 1171–1178, 2000.

[28] Pat Langley and Dongkyu Choi. Learning recursive control programs from problem solving. *J. Mach. Learn. Res.*, 7:493–518, 2006.

[29] Stephen M. Majercik and Michael L. Littman. Contingent planning under uncertainty via stochastic satisfiability. In *AAAI/IAAI*, pages 549–556, 1999.

[30] Drew McDermott. Planning and acting. *Cognitive Science*, 2:71–109, 1978.

[31] Kinshuk Mishra, Santi Ontañón, and Ashwin Ram. Situation assessment for plan retrieval in real-time strategy games. In *Proceedings of ECCBR-2008*, pages 355–369, 2008.

[32] Héctor Muñoz-Avila and Michael Cox. Case-based plan adaptation: An analysis and review. *IEEE Intelligent Systems*, 2007.

[33] D. Nau, T.C. Au, O. Ilghami, U. Kuter, D. Wu, F. Yaman, H. Muñoz-Avila, and J.W. Murdock. Applications of shop and shop2. *Intelligent Systems*, 20(2):34–41, 2005.

[34] Bernhard Nebel and Jana Koehler. Plan modifications versus plan generation: A complexity-theoretic perspective. Technical Report RR-92-48, 1992.

[35] Santi Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. Case-based planning and execution for real-time strategy games. In *Proceedings of ICCBR-2007*, pages 164–178, 2007.

[36] Steve Rabin. *AI Game Programming Wisdom*. Charles River Media, 2002.

[37] Steve Rabin. *AI Game Programming Wisdom II*. Charles River Media, 2004.

[38] Ashwin Ram and Anthony Francis. Multi-plan retrieval and adaptation in an experience-based agent. In David B. Leake, editor, *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. AAAI Press, 1996.

[39] Ashwin Ram and J. C. Santamaria. Continuous case-based reasoning. *Artificial Intelligence*, 90(1-2):25–77, 1997.

[40] Roger Schank. *Dynamic Memory Revisited*. Cambridge University Press, 1999.

[41] 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.

[42] L. Spalazzi. A survey on case-based planning. *Artificial Intelligence Review*, 16(1):3–36, 2001.

[43] A. Tate. Generating project networks. In *International Joint Conference on Artificial Intelligence, IJCAI-77*, pages 888–893, 1977.

[44] R.L. Tsatsoulis, Costas; Kashyap. Case-based reasoning and learning in manufacturing with the toltec planner. *IEEE Transactions on Systems, Man and Cybernetics*, 23:1010–1023, 1993.

[45] Manuela Veloso, Jaime Carbonell, A. Pérez, Daniel Borrajo, and E. Fink an J Blythe. Integrating planning and learning: The prodigy architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7, 1995.

[46] Ian Warfield, Chad Hogg, Steplien Lee-Urban, and Héctor Muñoz-Avila. Adaptation of hierarchical task network plans. In *FLAIRS 2007, Special Track on Case-Based Reasoning*, 2007.