Using Meta-reasoning to Improve the Performance of Case-Based Planning

Manish Mehta, Santiago Ontañón, and Ashwin Ram

CCL, Cognitive Computing Lab
Georgia Institute of Technology
Atlanta, GA 30332/0280
{mehtama1,santi,ashwin}@cc.gatech.edu

Abstract. Case-based planning (CBP) systems are based on the idea of reusing past successful plans for solving new problems. Previous research has shown the ability of meta-reasoning approaches to improve the performance of CBP systems. In this paper we present a new meta-reasoning approach for autonomously improving the performance of CBP systems that operate in real-time domains. Our approach uses *failure patterns* to detect anomalous behaviors, and it can learn from experience which of the failures detected are important enough to be fixed. Finally, our meta-reasoning approach can exploit both successful and failed executions for meta-reasoning. We illustrate its benefits with experimental results from a system implementing our approach called *Meta-Darmok* in a real-time strategy game. The evaluation of Meta-Darmok shows that the system successfully adapts itself and its performance improves through appropriate revision of the case base.

1 Introduction

Learning is a competence fundamental to intelligence, reflected in the ability of human beings to learn from their successes to make future progress and from their mistakes to improve themselves. Developing systems with learning abilities can therefore help us understand better the nature of intelligence. In order to create intelligent systems, it is thus important to provide systems with the ability to learn from their own experience (both successful and failed) and improve themselves. Failed experiences in problem solving play a role in learning, as they provide both humans and artificial systems with strong cues on what needs to be learned [5,9,10,15]. Likewise, successful experiences provide clues on ways to make progress and prosper in the problem domain. Meta-reasoning approaches have utilized successful and failed system experiences to create self-improving AI systems [1,6]. This paper investigates this ability of meta-reasoning approaches to improve the performance of case-based planning (CBP) systems that operate under real-time constraints.

Meta-reasoning systems are typically composed of a *base reasoner* that is in charge of the *performance task*, and a meta-reasoner that observes and modifies the base reasoner's plan. Meta-reasoning is the process of monitoring and

L. McGinty and D.C. Wilson (Eds.): ICCBR 2009, LNAI 5650, pp. 210–224, 2009.

controlling the reasoning process of the base reasoner, to adjust that reasoning process as needed. Previous research on introspective CBR (See Section 2) has shown that meta-reasoning can enable a CBR system to learn by refining its own reasoning process. In this paper we present a new meta-reasoning approach for case-based planning. Our approach is based on using a collection of author-defined *failure patterns*, which are explicit descriptions of anomalous situations. Failure patterns provide a case-based solution for detecting failures in the execution of the base reasoner. After failures have been detected, fixes have to be made in the system to prevent those failures from happening again. Our work has four main contributions: First, it can be used in real-time domains, and instead of modifying the plans in the case-base modifies the behavior of the base system by creating *daemons* that operate in real time. Second, we present a generic representation for *failure patterns* using finite state machines (FSM). Third, our system automatically learns which of the failures detected in the behavior of the system are important or not by comparing successful and unsuccessful past executions, whereas previous approaches required an explicit model of "correct behavior" in order to detect failures. Finally, our approach can learn both from successful and unsuccessful experiences.

In this paper we are going to use Darmok [13,14] as the base reasoning system, which is a case-based planning system designed to play real-time strategy (RTS) games. The system resulting from applying our approach to Darmok is called *Meta-Darmok*, with extends Darmok by giving it the ability to be introspectively aware of successful and failed execution, analyze the differences between the two executions and further adapt and revise the executing plans based on deliberating over the analyzed differences.

The rest of the paper is organized as follows. In Section 2, we present the related approaches that use meta-reasoning to improve the performance of CBR systems. Section 3 introduces our case based planning system, Darmok, and WARGUS, the RTS game used in our experiments. In Section 5, we present our meta-reasoning approach and Meta-Darmok. We present evaluation of Meta-Darmok in Section 6. Finally we conclude with future steps in Section 7.

2 Related Work

Application of CBR systems to real-world problems has shown that it is difficult for developers to anticipate all possible eventualities. Changing circumstances necessitate that a CBR system learns from its experiences and improve its performance over a period of time. Fox and Leake [8] developed a system to improve the retrieval of a CBR system using meta-reasoning. Their work used a model for the correct reasoning behavior of the system, and compared the performance of the model with the actual system performance to detect and diagnose reasoning failures. The approach, however, requires the development of a complete model of correct reasoning behavior, which is difficult to construct for a system operating in a complex real-time strategy game scenario like ours. Unlike their approach, our work doesn't depend upon an author created correct model of

Fig. 1. A screenshot of the WARGUS game

the system's performance but uses successful problem solving experiences as a benchmark model with which to compare the failed experience.

The DIAL system [11] uses introspective techniques to improve case adaptation. The system stores the traces of successful adaptation transformations and memory search paths and utilizes them to improve the system performance. Arcos [2] presents a CBR approach for improving solution quality in evolving environments. The approach however learns only from system's successes whereas in our approach we utilize both the successful and failed experiences. There are other system that have explored meta-reasoning approaches to identify system failures and improve system's performance. The Meta-Aqua system, for example, by Cox and Ram [7] uses a library of pre-defined patterns of erroneous interactions among reasoning steps to recognize failures in the story understanding task. The Autognostic system by Stroulia and Goel [16] uses a model of the system to localize the error in the system's element and and uses the model to construct a possible alternative trace which could lead to the desired but unaccomplished solution. The Introspect system [4] observes its own behavior so as to detect its own inefficiencies and repair them. The system has been implemented for the game of Go, a deterministic, perfect information, zero-sum game of strategy between two players. Ulam et. al. [18] present a model based meta-reasoning system for FreeCiv game that uses a self-model to identify the appropriate reinforcement learning space for a specific task. All these systems however are limited to learning from failed experience (and ignoring successful experiences) to improve the system performance.

3 Case-Based Planning in WARGUS

WARGUS is an open source clone of Warcraft II, a successful commercial real-time strategy game. Each player's goal in WARGUS is to survive and destroy the other players. Each player controls a number of troops, buildings,

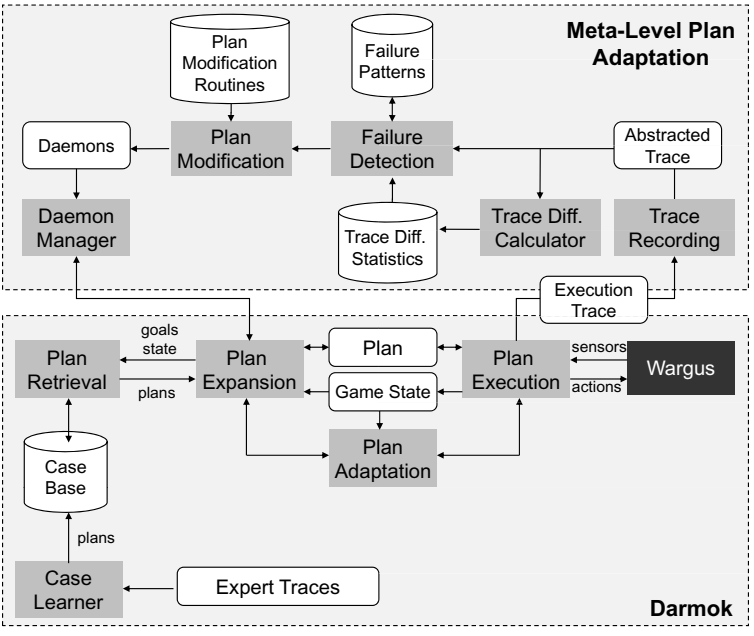

Fig. 2. Overview of the Meta-Darmok architecture

and workers (who gather resources such as gold, wood and oil in order to produce more units). Buildings are required to produce more advanced troops, and troops are required to attack the enemy. The calculations inherent in the combat system make the game non-deterministic. The game involves complex strategic reasoning, such as terrain analysis, resource handling planning, and scheduling, all of them under tight time constraints. For example, the map shown in Figure 1 is a 2-player version of the classical map "Nowhere to run, Nowhere to hide"(NWTR), one of the maps from *Battlenet* regularly player by human players, characterized by a wall of trees that separates the players. This map leads to complex strategic reasoning, such as building long range units (such as catapults or ballistas) to attack the other player before the wall of trees has been destroyed, tunneling early in the game through the wall of trees trying to catch the enemy by surprise, or other strategies, as explained in Section 6.

In this section we will briefly describe the Darmok [13,14] system, which serves as the base reasoner for our meta-reasoning approach. In order to play WARGUS Darmok learns plans from expert demonstrations, and then uses case-based planning to play the game reusing the learnt plans. The lower part of Figure 2 shows a simplified overview of the Darmok system. Basically, Darmok is divided in two main processes:

– *Case Acquisition*: involves capturing plans for playing the game using expert demonstrations. Each time a human plays a game, an *expert trace* is

generated (containing the list of actions performed in the game). The expert then annotates the trace stating which goals he was pursuing with each action. From the annotated traces, plans are extracted and stored in the case base.
- *Plan Execution*: The execution engine consists of several modules that together maintain a current plan to win the game. The *Plan Execution* module executes the current plan, and updates its state (marking which actions succeeded or failed). The *Plan Expansion* module identifies open goals in the current plan and expands them. In order to do that it relies on the *Plan Retrieval* module, which retrieves the most appropriate plan to fulfill an open goal. Finally, the *Plan Adaptation* module adapts the retrieved plans.

Let us present some additional detail on the Darmok system, required to understand our meta-reasoning approach.

Cases in Darmok consist of two parts: *plan snippets* and *episodes*. Snippets store plans, and episodes contain information on how successful was a plan in a particular situation. Specifically, a snippet has four main parts:

- A *goal*, which is a representation of the intended goal of the plan.
- A set of *preconditions* that must be satisfied before execution.
- A set of *alive conditions* that must be satisfied during the execution of the plan for it to have chances of success.
- The plan itself.

During execution, Darmok interleaves case-based planning and execution. The plan expansion, plan execution and plan adaptation modules collaborate to maintain a current *plan* that the system is executing. A plan in Darmok is represented as a tree consisting of *goals* and *plans* (like in HTN planning [12]). Initially, the plan consists of a single goal: "win the game". Then, the plan expansion module asks the plan retrieval module for a plan for that goal. That plan might have several subgoals, for which the plan expansion module will again ask the plan retrieval module for plans, and so on. When a goal still doesn't have an assigned plan, we say that the goal is *open*.

Additionally, each subplan in the plan has an associated state that can be: *pending* (when it still has not started execution), *executing*, *succeeded* or *failed*. A goal that has a plan assigned and where the plan has failed is also considered to be open. Open goals can be either *ready* or *waiting*. An open goal is ready when all the plans that had to be executed before this goal have succeeded, otherwise, it is waiting.

The plan expansion module is constantly querying the current plan to see if there is any ready open goal. When this happens, the open goal is sent to the plan retrieval module. The retrieved plan is sent to the plan adaptation module, and then inserted in the current plan, marked as pending.

The plan execution module has two main functionalities: check for basic actions that can be sent to the game engine and check the status of plans that are in execution. Basic actions that are ready and with all its preconditions satisfied are sent to WARGUS to be executed. If the preconditions are not satisfied, the

action is sent back to the adaptation module to see if it can be repaired. If it cannot, then the action is marked as failed. Whenever a basic action succeeds or fails, the execution module updates the status of the plan that contained it. When a plan is marked as failed, its corresponding goal is open again. If the alive conditions of an executing plan or action are not satisfied, it is also marked as failed, and if its success conditions are satisfied, then it is marked as succeeded.

4 Plan Adaptation

Plan adaptation in Darmok is divided into two difference processes: parameter adaptation and structure adaptation. When a plan is retrieved from the case base, structure adaptation is used (removing or inserting actions and or goals), and when each one of the actions in a plan is about to be sent to execution, parameter adaptation is used (which will adapt the parameters of the actions: locations and unit identifiers). Let us briefly explain these two processes.

During parameter adaptation, Darmok attempts to adapt the coordinates and unit identifiers present in the actions so that they can be applied to the game at hand. For instance, if in an action the expert demonstrated that " a farm has to be built at coordinates 26,25", Darmok will analyze which properties the coordinates 26,25 satisfy (e.g. they represent an empty extension of grass, far from the enemy and close to a forest), and look for a location in the current map that is the most similar. Darmok will do the same with the unit identifiers (looking to use the most similar units in the current game that the expert used in his demonstration game).

For structure adaptation, Darmok analyzes a plan and determines whether all the actions are required or not in the current game. For instance, maybe a plan specifies that a "barracks" has to be built, but in the current game, we already have "barracks". Darmok also determines whether additional actions have to be executed in order to make the plan executable in the current game. For instance maybe the plan makes reference to some "worker units" but in the current game we don't have them, so actions in order to obtain them are required. Structure adaptation is achieved by generating a *plan dependency graph* using the preconditions and success conditions of the actions. The plan dependency graph is a directed graph where each node in the graph is a primitive action and each link represents the dependency relationship between the parent and the child.

5 Meta-level Plan Adaptation

The Darmok system is able to play the game of WARGUS after observing expert traces. However, once it has learnt, the system is not able to modify the plans it has learnt. Although Darmok has the capability of learning from experience (by remembering which plans succeeded and which ones failed in different situations), it does not have any capability to fix the plans in its case base. If the expert that Darmok learnt from made a mistake in one of the plans, Darmok

will repeat that mistake again and again each time Darmok retrieves that plan. The meta-reasoning approach presented in this paper provides Darmok exactly with that capability, resulting in a system called *Meta-Darmok*, shown in Figure 2. By analyzing past performances, Meta-Darmok can fix the plans in the case base, improving the performance over Darmok.

Our meta-level plan adaptation approach consists of five parts: *Trace Recording*, *Trace Difference Calculation*, *Failure Detection*, *Plan Modification*, and the *Daemon Manager*. During trace recording, a trace holding important events happening during the game is recorded. Trace difference calculation involves keeping track of differences across successful and unsuccessful games. Failure detection involves analyzing the execution trace and differences across various games to find possible issues with the executing plans by using a set of *failure patterns*. These failure patterns represent a set of pre-compiled patterns that can identify the cause of each particular failure by identifying instances of these patterns in the trace. Once a set of failures has been identified, the failed conditions can be resolved by appropriately revising the plans using a set of *plan modification routines*. These plan modification routines are created using a combination of basic modification operators (called *modops*). The modops are in the form of modifying the original elements of the plan (i.e. actions), introduction of new elements or reorganization of the elements inside the plan. Finally, some of the modifications take form of *daemons*, which monitor for failure conditions to happen when Darmok retrieves some particular behaviors. The daemon manager triggers the execution of such daemons when required. We describe each of the different parts of the meta-level plan adaptation in detail next.

5.1 Trace Recording

In order to analyze the behavior of Darmok, the meta-reasoning module records an *execution trace* when Darmok is playing a game. The execution trace is used to record important events happening during the execution of Darmok. These events are in the form of information regarding the plans that were executing, their start and end times, their final execution status i.e. whether the plans started, failed or succeeded.

The trace also contains the external state of the game recorded at various intervals so that different information can be extracted from it during reasoning about the failures happening at execution time. The trace provides a considerable advantage in performing adaptation of plans with respect to only analyzing the instant in which the failure occurred, since the trace can help localize portions that could possibly have been responsible for the failure.

During its run, Darmok records a low level execution trace that contains information related to basic events including the identifier of the plan that was being executed, the corresponding game state when the event occurred, the time at which the plan started, failed or succeeded, and the delay from the moment the plan became ready for execution to the time when it actually started executing. All this information is recorded in the execution trace, which the system updates as events occur at runtime. The execution trace also records any modifications

performed by the various system components to the executing goal and plans. As explained earlier, this is carried out by the plan adaptation modules. The plan adaptation module makes various changes to the list of goals and plans that are currently executing.

Once a game finishes, an *abstracted trace* is created from the execution trace that Darmok generates. The abstracted trace is the one used by the rest of components of the meta-reasoning module. The abstracted trace consists of:

- Unit Data: information regarding units such as hit points, status (whether a unit is idle, for example), location, etc.,
- Idle Data: which units were idle and the cycle intervals for which they were idle.
- Kill Data: the cycles at which particular units were attacked and killed.
- Resource Data: for each entry in the trace, the corresponding resources that were available, such as the amount of food, gold, wood and oil.
- Attack Data: the units that were involved in an attack. For example, the enemy units that were attacking and the AI units that were under attack.
- Basic Plan Failure Data: the reason for plan failures, such as whether it was due to insufficient resources or not having a particular unit available to carry out a plan.
- Trace Data: contains information including the number of goals and plans that were pursued, number that succeeded, failed, didn't finish and didn't start. The data also includes number of times a goal was pursued, number of times it restarted, the total number of resource gathering plans that were used and type of resource that was gathered by those plans as part of satisfying that goal.

Once the abstracted trace is generated, it is both sent to the *failure detection* component and to the *trace difference calculation* component. The trace difference calculator records the differences between the stored repository traces and the new execution trace as we explain next, and the failure detection component uses this information to find failures in the trace.

5.2 Trace Difference Calculation

Each trace records various statistics as explained previously to help calculate differences across them. The differences among other things include various statistics:

- Actions level: holds the difference in terms of the actions that are carried out. These differences are for example, differences in parameters like location where the unit is built, number of units built by an action, number and type of resources gathered.
- Non-Existent Plan/Goal: holds goals and plans that were present in a particular game and absent in other.
- Plan/Goal Type: differences in the type of plans/goals that were used during execution.

- Cycle Level: differences in start and end cycle of plans and goals.
- Goal Count: comparison of number of times a goal was pursued.
- Action Plan Stats: differences in terms of number of actions that succeeded, failed, didn't start or finish.
- Resource Gathering Plans: holds the difference in terms of number of resource gathering plans that were used and the type of resource that was gathered, the number that succeeded and failed.
- Score differences: differences in scores at the end of completion of a basic operator plan and higher level goal and plan.
- Restart differences: differences in terms of the number of times a goal was restarted.
- Plan/Goal Count: differences in terms of number of goals/plans that were used and count of particular type of plans that were used.

The *trace difference calculator* module calculates the differences for the trace pairs involving a) Type *SuccVsUnsucc*: successful vs unsuccessful traces and b) Type *UnsuccVsUnSucc*: unsuccessful vs unsuccessful traces. As Darmok plays more games of WARGUS the trace difference calculator, keeps updating the trace difference statistics. These statistics contain the probability of the differences being present across both types of trace pairs. These probability estimates are used by failure detection as explained next.

5.3 Failure Detection

Failure detection involves localizing the fault points. Although the abstracted trace defines the space of possible causes for the failure, it does not provide any help in localizing the cause of the failure. Traces can be extremely large, especially in the case of complex games on which the system may spend a lot of effort, thus analyzing the trace looking for failures might be a complicated problem for two reasons: first, traces might be huge, and second, it is not clear what to look for in the trace, since it is not clear what a "failure" looks like in a trace.

In order to solve both problems, our meta-reasoning module contains a collection of *failure patterns*, which can be used to identify the cause of each particular failure by identifying instances of these patterns in the trace [3,5,17]. The failure patterns essentially provide an abstraction mechanism to look for typical patterns of failure conditions over the execution trace. The failure patterns simplify the blame-assignment process into a search for instances of the particular problematic patterns. Specifically, failure detection conceptually consists of two phases, matching and filtering:

1. Failure Pattern Matching: during this phase, all the failure patterns stored in the system are matched against the abstracted trace, and all the matches are recorded.
2. Filtering Non-important Failures: during this phase, the trace difference statistics are used to discern which failures are important and which ones

can be ignored. The result of this phase is the set of failures that are passed along to the plan modification component.

In our approach both phases are fused in a single process. Failure patterns are represented as finite state machines (FSM), which encode information needed to perform both phases mentioned earlier (matching and filtering). Figure 3 shows an example of one of the failure patterns defined in our system. Specifically, the pattern specifies that a failure is detected if a "plan restart" is found in the current trace, and if the probability of such failure satisfies some thresholds according to the trace difference statistics database.

An example of a failure detected from the abstracted trace, related to the functioning of plan adaptation module, is *Important Goals or Plans Removed failure*. The failure pattern detects whether important goals and plans have been removed from the trace. As explained earlier, the plan adaptation module potentially removes and/or adds goals and plans to existing list of goals and plans. This failure allows to detect whether, due to some failure in processing of plan adaptation module, some goals which are important for executing the current plan are removed. For example, the plan adaptation module may remove a "build tower goal" that is not considered necessary, which results in system losing in one of the maps. The fix for this failure would be to reinsert back the important goal or plan that has been removed back into the currently executing plan.

There are other failures that are detected based on the probability estimates from the trace pairs. The differences which have a high probability of presence in Type SuccVsUnsucc and low in Type UnsuccVsUnsucc are flagged for repair as part of the failure detection. This helps find the differences that are important and try to remove only those failures. These statistics help gauge probability values on whether a particular difference reflects some factor that is present or absent in successful as compared to unsuccessful games. An example of such a difference could be the *restart difference*. In Darmok, a goal restarts when it doesn't finish successfully in the previous attempt. In unsuccessful traces, it could happen that the goal restarts multiple times whereas in successful traces the goal finishes successfully in one or two attempts. This would lead to a lot of time and resources spent by the system in pursuing the restarted goal in unsuccessful traces. After analysis of the Type SuccVsUnsucc and Type UnsuccVsUnsucc traces, restart difference would have a high probability of presence in Type SuccVsUnsucc traces and low in Type UnSuccVsUnsucc. Once this difference is flagged, the failure pattern *restart failure* is detected (shown in Figure 3). A fix for this failure would be to add a constraint on the number of times the goal could be restarted. Other examples of failure patterns and their corresponding plan modification routines are given in Table 1.

5.4 Plan Modification

Once the cause of the failure is identified, it needs to be addressed through appropriate modification. These modifications are in the form of inserting or

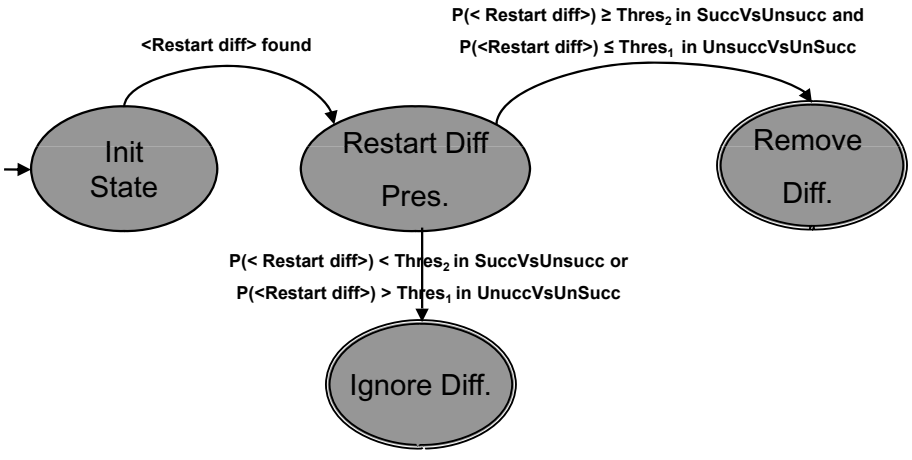

Fig. 3. The figure shows the failure pattern *restart failure* in WARGUS. The failure is addressed when there is a high probability (greater than $Thres_2$) of presence in Type succVsUnsucc and low probability (less than $Thres_1$) in Type UnSuccVsUnSucc.

Table 1. Some example failure patterns and their associated plan modification routine in WARGUS

Failure Pattern	**Plan Modification Routine**
Plan Composition failure (e.g., issues with plan feature like its parameters, its type etc)	Change the plan according to the particular composition issue i.e change the location, parameter etc
Goal/Plan count Failure	Constrain the number of times a plan/goal is pursued.
Restart failure	Constrain the number of time a plan/goal is restarted
Goal/Plan Missing failure	Add a plan/goal at appropriate time during the execution
Plan Step(s) Missing failure	Add step(s) to a particular plan
Basic Operator failure	Adding a basic action that fixes the failed condition

removing a new appropriate plan at the correct position in the failed plan, or removing some steps or changing some parameter of an executing plan. The plan modification step involves applying these operators to modify the failed plans appropriately.

Once the failure patterns are detected from the execution trace, the corresponding plan modification routines and the failed conditions are inserted as daemons for the map in which these failed conditions are detected. The daemons act as a meta-level reactive plan that operates over the executing plan at runtime. The conditions for the failure pattern become the preconditions of the

plan and the plan modification routine consisting of basic modops become the steps to execute when the failure pattern is detected. The daemons operate over the executing plan, monitor their execution, detect whether a failure is about to happen and repair the plan according to the defined plan modification routines.

Notice thus, that Meta-Darmok does not directly modify the plans in the case-base of Darmok, but reactively modifies those plans when Darmok is about to use them, in case some failure is about to happen. In the current system, we have defined 20 failure patterns and plan modification routines for WARGUS.

The adaptation system can be easily extended by writing other patterns of failure that could be detected from the abstracted trace and the appropriate plan modifications to the corresponding plans that need to be carried out in order to correct the failed situation. In order to understand the process better, let us look at an example.

5.5 Exemplification

In order to understand the working of the meta-level plan adaptation module let us look at an illustrated example. In some runs, we observed that the plan adaptation module of Darmok inserted a large amount of resource gathering goals (in particular sending peasants to gather wood) for the following reason. In some of the expert traces, the expert sent a very small set of peasants to gather wood at the very beginning of the game. Those few peasants would gather enough wood for the rest of the game, since they would start gathering wood very early. Sometimes, Darmok would reassign one of those peasants to another task. Thus, when the need for wood arises later in the game, Darmok will be low on wood. Since wood is harvested very slowly, Darmok sends several peasants to gather wood at the same time in order to have the wood on time. This situation is fine in the short term, but in the long term, in a map like the one shown in Figure 1 ("Nowhere to Run, Nowhere to Hide" or NWTR), a hole in the wall of trees will be opened quickly (since there are lots of peasants chopping wood). This allows the enemy to enter through the wall of trees and attack Darmok before Darmok is ready to reply. Let us see how the meta-level adaptation module can help fix this problem.

After playing several games, the trace difference calculator module accumulates statistics, and in particular one of them is that the average number of gathering resource goals for the NWTR map is much higher for unsuccessful traces than for successful traces. After the analysis of the Type SuccVsUnsucc and Type UnsuccVsUnsucc traces, the difference in statistics for the occurrence of the resource gathering goal will have a high probability of presence in Type SuccVsUnsucc traces and low in Type UnsuccVsUnsucc, thus the difference will be flagged. One of the failure patterns incorporated in our system is called *Goal/Plan Count Failure*. This failure pattern gets triggered when the difference in statistics for the occurrence of the goal or plan has a high probability of presence in Type SuccVsUnsucc traces and low in Type UnsuccVsUnsucc. When the meta-level is analyzing a new unsuccessful trace in the NWTR map, the Goal/Plan Count Failure pattern will be triggered.

The plan modification routine associated with the *Goal/Plan Count Failure* pattern consists of creating a new daemon that counts the number of times a particular goal appears during the course of a game, and if it is beyond some threshold, the daemon will delete such goals from the current plan. In particular, in our example, the daemon will be instantiated to look for resource gathering goals, and the threshold will be set to the typical value found in the successful traces in the trace difference statistics data base.

In this particular example, once the daemon was inserted, we observed that the daemon prevents addition of resource gathering goals beyond a certain point. In the short term, Darmok will struggle to have resources, since Darmok needs them to build buildings and train units. However, the daemon simply prevents the wall of trees to get destroyed. As a result resources are gathered at a slower pace and the few peasants gathering resources finally obtain enough wood, to enable Darmok to successfully complete the plan without destroying the wall of trees too early.

6 Meta-level Plan Adaptation Results

To evaluate Meta-Darmok, we conducted two different experiments turning the meta-reasoner on and off respectively. When the meta-reasoning is turned off, the execution traces are recorded as part of the execution. When the meta-reasoning is turned on, these execution traces are used as the starting point to calculate the trace statistics as new traces accumulate.

The experiments were conducted on 5 different variations of the NWTR map (shown in Figure 1). NWTR maps have a wall of trees separating the opponents that introduces a highly strategic component in the game (one can attempt ranged attacks over the wall of trees, or prevent the trees to be chopped by building towers, etc.). 3 different expert demonstrations were used for evaluation from NWTR maps (which lead to three different sets of plans to be used by Darmok). Moreover, Darmok can learn from more than one demonstration, so we evaluate results when Darmok learns from one, two and from all three demonstrations.

Table 2 shows the results of the experiments with and without meta-reasoning. NT indicates the number of expert traces. For each experiment 6 values are

Table 2. Effect of plan adaptation on game statistics

	No MetaLevel Adaptation						MetaLevel Adaptation						
NT	*W*	*D*	*L*	*ADS*	*AOS*	*WP*	*W*	*D*	*L*	*ADS*	*AOS*	*WP*	*improvement.*
1	4	4	7	1333	1523	**26.67%**	9	3	3	2096	582	**60.00%**	125.00%
2	5	2	8	1234	1010	**33.33%**	9	3	3	2628	356	**60.00%**	80.00%
3	5	3	7	1142	1762	**33.33%**	6	5	4	2184	627	**40.00%**	20.00%
	11	8	26	3709	4295	31.11%	24	11	10	6908	1565	53.33%	75.00%

shown: W, D and L indicate the number of wins, draws and loses respectively. ADS and AOS indicate the average Darmok score and the average opponent score (where the "score" is a number that WARGUS itself calculates and assigns to each player at the end of each game). Finally, WP shows the win percentage. The right most row presents the improvement in win percentage comparing meta-reasoning with respect to no meta-reasoning. The bottom row shows a summary of the results. The results show that meta-reasoning leads to an improvement of the percentage of wins as well as the player score to opponent score ratio. An improvement occurs in all cases irrespective of the number of traces used. Notice that in the case where Darmok learnt from 3 expert traces, the meta-reasoner is not able to bring the performance up to 60% like in the other two scenarios. This is because one of the traces used in our experiments was clearly inferior to the other two, and the performance of Darmok strongly depends on the quality of the expert traces [14]. However, meta-reasoning is still able to significantly improve the performance.

7 Conclusion

In this paper, we have presented a meta-reasoning approach to improve the performance of case-based planning systems, and a particular system, Meta-Darmok, that implements it. Meta-Darmok is based on the Darmok system, which plays an RTS game domain. We have shown that meta-reasoning can improve the performance of a CBP system that operates in a real-time domain. Failure patterns are useful to characterize typical failures. Moreover, by analyzing the differences between the successful and failed executions Meta-Darmok can determine which of the differences detected in failed executions with respect to successful executions are important or not. Finally, we have shown that *daemons* can be used to introduce reactive elements in the execution of the system that will adapt the behavior if failures are detected in real time. Our experimental results indicate that our approach improves the performance of the CBP system (overall improvement of 75%).

There were a few occasions when the meta-reasoning module introduced unwanted changes that degraded the system performance. However, in the few times it happened, the issue could be resolved if the system kept track of the system performance with the introduction of daemons for a particular map. If the system loses the map with the introduction of certain daemons, it could realize that the adaptation is causing unwanted changes in system performance. This might involve incorporating the actions of the meta-reasoning module into the execution trace to allow the meta-reasoner to introspect itself. We plan to explore this line in our future research. We also plan to apply our approach to other case-based planning systems to validate the generality of the approach. Finally, we also plan to investigate strategies to automatically generate failure patterns or tools for easy authoring of such failure patterns.

References

1. Anderson, M.L., Oates, T.: A review of recent research in metareasoning and metalearning. AI Magazine 28, 7–16 (2007)
2. Arcos, J.L.: T-air: A case-based reasoning system for designing chemical absorption plants. In: Aha, D.W., Watson, I. (eds.) ICCBR 2001. LNCS, vol. 2080, pp. 576–588. Springer, Heidelberg (2001)
3. Carbonell, J.G., Knoblock, A.C., Minton, S.: Prodigy: An Integrated Architecture for Planning and Learning. Lawrence Erlbaum Associates, Mahwah
4. Cazenave, T.: Metarules to improve tactical go knowledge. Inf. Sci. Inf. Comput. Sci. 154(3-4), 173–188 (2003)
5. Cox, M.T., Ram, A.: Failure-driven learning as input bias. In: Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society, pp. 231–236 (1994)
6. Cox, M.T.: Metacognition in computation: a selected research review. Artif. Intell. 169(2), 104–141 (2005)
7. Cox, M.T., Ram, A.: Introspective multistrategy learning: On the construction of learning strategies. Technical report (1996)
8. Fox, S., Leake, D.: Introspective reasoning for index refinement in case-based reasoning. Journal of Experimental and Theoretical Artificial Intelligence 13, 63–88 (2001)
9. Hammond, K.J.: Learning to anticipate and avoid planning problems through the explanation of failures. In: Proceedings of the Fifth National Conference on Artificial Intelligence, pp. 556–560 (1986)
10. Kolodner, J.L.: Capitalizing on failure through case-based inference. In: Proceedings of the Ninth Annual Conference of the Cognitive Science Society, pp. 715–726 (1987)
11. Leake, D.B., Kinley, A., Wilson, D.: Learning to improve case adaptation by introspective reasoning and CBR. In: Aamodt, A., Veloso, M.M. (eds.) ICCBR 1995, vol. 1010, pp. 229–240. Springer, Heidelberg (1995)
12. Nau, D., Au, T.C., Ilghami, O., Kuter, U., Wu, D., Yaman, F., Muñoz-Avila, H., Murdock, J.W.: Applications of shop and shop2. Intelligent Systems 20(2), 34–41 (2005)
13. Ontañón, S., Mishra, K., Sugandh, N., Ram, A.: Case-based planning and execution for real-time strategy games. In: Weber, R.O., Richter, M.M. (eds.) ICCBR 2007. LNCS, vol. 4626, pp. 164–178. Springer, Heidelberg (2007)
14. Ontañón, S., Mishra, K., Sugandh, N., Ram, A.: On-line case-based planning. Computational Intelligence (to appear)
15. Schank, R.C.: Dynamic memory: A theory of reminding and learning in computers and people. Cambridge University Press, Cambridge (1982)
16. Stroulia, E., Goel, A.K.: Functional representation and reasoning in reflective systems. Journal of Applied Intelligence 9, 101–124 (1995)
17. Sussman, J.G.: A Computational Model of Skill Acquisition. American Elsevier, Amsterdam (1975)
18. Ulam, P., Jones, J., Goel, A.K.: Combining model-based meta-reasoning and reinforcement learning for adapting game-playing agents. In: AIIDE (2008)