23rd AAAI Conference on Artificial Intelligence (AAAI 2008), Chicago, USA, pp 702-707

On-Line Case-Based Plan Adaptation for Real-Time Strategy Games

Neha Sugandh and **Santiago Ontañón** and **Ashwin Ram**
CCL, Cognitive Computing Lab
Georgia Institute of Technology
Atlanta, GA 303322/0280
{nsugandh,santi,ashwin}@cc.gatech.edu

Abstract

Traditional artificial intelligence techniques do not perform well in applications such as real-time strategy games because of the extensive search spaces which need to be explored. In addition, this exploration must be carried out on-line during performance time; it cannot be precomputed. We have developed on-line case-based planning techniques that are effective in such domains. In this paper, we extend our earlier work using ideas from traditional planning to inform the real-time adaptation of plans. In our framework, when a plan is retrieved, a plan dependency graph is inferred to capture the relations between actions in the plan. The plan is then adapted in real-time using its plan dependency graph. This allows the system to create and adapt plans in an efficient and effective manner while performing the task. The approach is evaluated using WARGUS, a well-known real-time strategy game.

Introduction

Artificial Intelligence (AI) techniques have been successfully applied to several computer games. However, in the vast majority of computer games traditional AI techniques fail to perform well because of the characteristics of the game. Most current commercial computer games have vast search spaces in which the AI has to make decisions in real-time, rendering traditional search based techniques inapplicable. In particular, 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 (Aha, Molineaux, & Ponsen 2005), they are adversarial domains, they are non-deterministic and 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).

To address these issues, we developed Darmok (Ontañón *et al.* 2007), a case-based planning system that is able to deal with the complexity of domains such as WARGUS. Case-based planning (CBP) techniques (Spalazzi 2001) work by reusing previous stored plans for new situations instead of planning from scratch. A key problem in CBP is plan adaptation (Muñoz-Avila & Cox 2007); plans cannot be replayed exactly as they were stored, specially in games of the complexity of modern RTS games. More specifically, CBP techniques for RTS games need adaptation techniques that are suitable for dynamic and unpredictable domains. Plan adaptation techniques can be classified in two categories: those adaptation techniques based on domain specific rules (domain specific, but fast) and those based on domain independent search-based techniques (domain independent, but slow). However, most previous work on plan adaptation focus on one-shot adaptation where a single plan or set of plans are retrieved and adapted before execution. In this paper we are interested on developing domain independent and search-free plan adaptation techniques that can be interleaved with execution and that are suitable for RTS games.

Our plan adaptation technique is based on two ideas: a) removing useless operations from a plan can be done by analyzing a *dependency graph* without performing search and b) the insertion of new operations in the plan can be delegated to the case-base planning cycle itself, thus also getting rid of the search. Our plan adaptation approach has been implemented in the Darmok system with promising results.

In the remainder of this paper we first introduce related work on plan adaptation. Then we present some of the issues of planning in RTS games. After that, we introduce the Darmok system focusing on plan adaptation. The paper closes with experimental results and conclusions.

Related Work

Case-based reasoning (CBR) (Aamodt & Plaza 1994) is a problem solving methodology based on reutilizing specific knowledge of previously experienced and concrete problem situations (cases). Case-based planning is the application of the CBR methodology to planning, and as such, it is planning as remembering (Hammond 1990). CBP involves reusing previous plans and adapting them to suit new situations. There are several motivations for case-based planning techniques (Spalazzi 2001): first, it inherits the *psychological plausibility* from case-based reasoning, and second, it has the potential to increase the *efficiency* with respect to generative planners (although, in general, reusing plans has the same or even higher worst-case complexity than planning from scratch (Nebel & Koehler 1992)).

Figure 1: A screenshot of the WARGUS game.

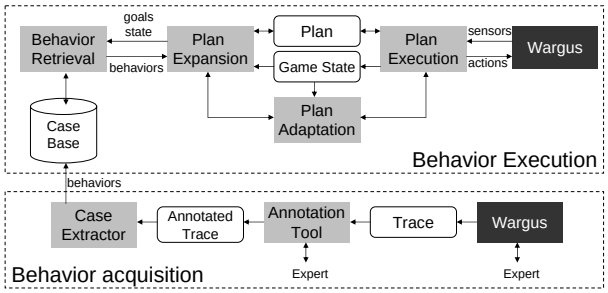

Figure 2: Overview of the Darmok system.

One of the first case-based planning systems was CHEF (Hammond 1990), able to build new recipes based on user's request for dishes with particular ingredients and tastes. CHEF contains a memory of past failures to warn about problems and also a memory of succeeded plans from which to retrieve plans. 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, that means that understanding has broken down and something has to be fixed. Thus, planning failures tell the system when it needs to learn. CHEF performs plan adaptation by a set of domain-specific rules called TOPs.

Domain-independent nonlinear planning has been shown to be intractable (NP-hard). PRIAR (Kambhampati & Hendler 1992) was designed to address that issue. PRIAR works by annotating generated plans with a *validation structure* that contains an explanation of the internal causal dependencies so that previous plans can be reused by adapting them in the future. Related to PRIAR, the SPA system was presented by Hanks and Weld (Hanks & Weld 1995). The key highlight of SPA is that it is complete and systematic (while PRIAR is not systematic, and CHEF is neither complete nor systematic), but uses a simpler plan representation than PRIAR. Extending SPA, Ram and Francis (Ram & Francis 1996) presented MPA (Multi-Plan Adaptor), that extended SPA with the ability to merge plans. The main issue with all these systems is that they are all based on search-based planning algorithms, and thus are not suitable for real-time domains, where the system has to generate quick responses to changes in the environment.

For an extensive overview of case-based plan adaptation techniques see (Muñoz-Avila & Cox 2007).

Case-Based Planning in WARGUS

Figure 1 shows a screen-shot of WARGUS, a real-time strategy game where each player's goal is to remain alive after destroying the rest of the players. Each player has a series of troops and buildings and gathers resources (gold, wood and oil) in order to produce more troops and buildings. Buildings are required to produce more advanced troops, and troops are required to attack the enemy. In addition, players can also build defensive buildings such as walls and towers. Therefore, WARGUS involves complex reasoning to determine the proper sequence of actions. A more detailed discussion on the complexity of WARGUS as a planning domain can be found in (Ontañón *et al.* 2007).

In this section we will briefly describe the Darmok system, in which we have implemented our plan adaptation techniques. Darmok was designed to play the game of WARGUS . In order to achieve this, Darmok learns behaviors from expert demonstrations, and then uses case-based planning to play the game reusing the learnt behaviors. Figure 2 shows an overview of our case-based planning approach. Basically, we divide the process in two main stages:

- *Behavior acquisition*: During this first stage, an expert plays a game of WARGUS and the trace of that game is stored. Then, the expert annotates the trace explaining the goals he was pursuing with the actions he took while playing. Using those annotations, a set of behaviors are extracted from the trace and stored as a set of cases.
- *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 in the *Behavior Retrieval* module, that given an open goal and the current game state retrieves the most appropriate behavior to fulfill the open goal. Finally, the *Plan Adaptation* module (the focus of this paper) adapts the retrieved plans according to the current game state.

In the remainder of this paper we will focus on the plan on-line adaptation component. However, for completeness, the next sections present a brief overview of the plan expansion and plan execution modules, see (Ontañón *et al.* 2007) for a more detailed explanation of those modules.

Plan Representation Language

The language that Darmok uses to represent behaviors has been designed to allow a system to learn behaviors, represent them, and to reason about the behaviors and their intended and actual effects. The basic constituent piece is the *behavior*. A behavior has two main parts: a *declarative* part and a *procedural* part. The declarative part has the purpose of providing information to the system about the intended use of the behavior, and the procedural part contains the ex-

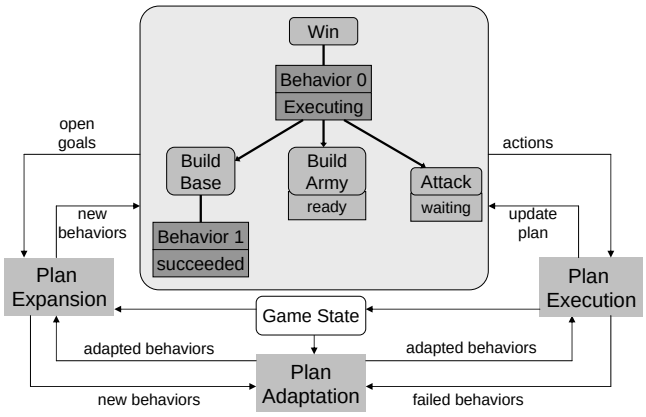

Figure 3: Interleaved plan expansion and execution.

ecutable behavior itself. The declarative part of a behavior consists of three parts:

- A *goal*, that is a representation of the intended goal of the behavior.
- A set of *preconditions* that must be satisfied before the behavior can be executed.
- A set of *alive conditions* that represent the conditions that must be satisfied during the execution of the behavior for it to have chances of success.

Notice that unlike classical planning approaches, postconditions cannot be specified for behaviors, since a behavior is not guaranteed to succeed. Thus, we can only specify the goal a behavior pursues. The procedural part of a behavior consists of executable code that can contain the following constructs: *sequence*, *parallel*, *action* (basic actions in the application domain), and *subgoal* (that need to be further expanded). A goal may have parameters, and must define a set of *success conditions*. A goal is a symbolic non-operational description of the goal, while success conditions are actual predicates that can be checked in the game state to evaluate whether a goal was satisfied or not. Notice that a success condition is distinct from a postcondition. Postconditions are conditions to be expected after a behavior executes while success conditions are the conditions that when satisfied we can consider the behavior to have completed its execution.

Run-Time Plan Expansion and Execution

During execution, the plan expansion, plan execution and plan adaptation modules collaborate to maintain a current *partial plan tree*, as depicted in Figure 3.

A *partial plan tree* in our framework is represented as a tree consisting of two types of nodes: *goals* and *behaviors* (notice the similarity with HTN planning (Nau *et al.* 2005)). In the remainder of this paper we will refer to the partial plan tree simply as the "plan". Initially, the plan consists of a single goal: "win the game". Then, the plan expansion module asks the behavior generation module to generate a behavior for that goal. That behavior might have several subgoals, for which the plan expansion module will again ask the behavior generation module to generate behaviors, and so on. When a goal still doesn't have an assigned behavior, we say that the goal is *open*. The top of Figure 3 shows an exemplification of a partial plan tree.

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

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 behavior generation module to generate a behavior for it. Then, that behavior is sent to the behavior 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:

- For each pending behavior, the execution module evaluates the preconditions, and as soon as they are met, the behavior starts its 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 behavior is sent back to the adaptation module to see if the plan can be repaired. If it cannot, then the behavior is marked as failed.
- Whenever a basic action succeeds or fails, the execution module updates the status of the behavior that contained it. When a basic action fails, the behavior is marked as failed, and thus its corresponding goal is open again.
- If the alive conditions of an executing behavior are not satisfied, the behavior is marked as failed. If the success conditions of a behavior are satisfied, the behavior is marked as succeeded.
- Finally, if a behavior is about to be executed and the current game state has changed since the time the behavior generation module generated it, the behavior is handed back to the plan adaptation module to make sure that the plan is adequate for the current game state.

On-Line Case-Based Plan Adaptation

The plan adaptation module is divided in two submodules: the *parameter adaptation module* and the *structural plan adaptation module*. The first one is in charge of adapting the parameters of the basic actions, i.e. the coordinates and specific units (see (Ontañón *et al.* 2007) for an explanation on how that module works). In this section we will focus on the structural plan adaptation module.

Plans are composed of four basic types of elements: actions, which are the basic actions that can be executed; 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 sub-goal plans requiring further expansion. A sequential

plan specifically defines the order in which the component plans need to be executed. We can further deduce dependencies between different plans using their preconditions and success conditions. We specifically consider only plans which are completely expanded and do not contain a subgoal which further needs to be expanded. We generate a *plan dependency graph* using the preconditions and success conditions of the actions. This plan dependency graph informs the plan adaptation process.

Plan Dependency Graph Generation

Each action has preconditions and success conditions, and each goal has only a set of success conditions. Further, every plan has a root node that is always a sub-goal. The plan dependency graph generation, begins by taking the success conditions of the root of the plan and finding out which of the component actions in the plan contribute to the achievement of these success conditions. These actions are called *direct sub-plans* for the subgoal. All *direct sub-plans* are added to the plan dependency graph. Then the plan dependency graph generator analyzes the preconditions of each of these direct sub-plans. Let B_1 be an action in the plan which contributes to satisfying the preconditions of a direct sub-plan D_1. Then, it adds B_1 to the plan dependency graph and forms a directed edge from B_1 to D_1. This directed edge can be considered as a dependency between B_1 and D_1. This is done for each of the direct sub-plans. Further this is repeated for each of the actions which are added to the graph, for example B_1 until the graph no longer expands. This process results in the formation of a plan dependency graph with directed edges between actions representing that one action contributes to the achievement of the preconditions of another action.

Actions have success conditions along with preconditions. However, a challenge in our work is that simple comparison of preconditions of a plan P_1 with success conditions of another plan P_2 is not sufficient to determine whether P_2 contributes to achievement of preconditions of P_1. This is because there isn't necessarily a direct correspondence between preconditions and success conditions. An example is the case where P_1 has a precondition testing the existence of a single unit of a particular type. However, P_2 may have a success condition testing the existence of a given number n of units of the same type. An edge should clearly be formed from P_2 to P_1, however a direct comparison of the conditions will not yield this relation.

For that purpose, the plan dependency graph generation component needs a precondition-success condition matcher (ps-matcher). In our system, we have developed a 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. All of these can be compared to each other, and thus the ps-matcher has rules that specify that all those 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.

The plan adaptation following the creation of the plan dependency graph has two sub-processes: elimination of unnecessary actions, and insertion of extra actions. The first one is performed as soon as the plan is retrieved, and the second one is performed on-line as the plan executes.

Removal of unnecessary actions

The removal of actions proceeds using the plan dependency graph. Given a plan for a goal, the plan dependency graph is generated and the success conditions of each direct plan are evaluated for the game state at that point of execution. This gives a list L of direct plans whose all success conditions are satisfied and hence do not need to be executed. Now, consider L as the list of actions that can be removed from the plan corresponding to the plan dependency graph. All actions B which are present only on paths terminating on an action D such that $D \in L$ can be removed and hence can be added to L. This is repeated until L becomes stable and no new plan is added to it. All actions in L are removed from the plan dependency graph. The resulting plan dependency graph has only those actions whose success conditions are not satisfied in the given game state and which have a path to a direct plan, also with success conditions not satisfied in the given game state.

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 create a game state where the given preconditions are satisfied so that the execution of the plan can proceed. To enable this, the adaptation module inserts *satisfying goals* in the plan (one goal per unsatisfied precondition). The satisfying goals are such that when plans to achieve the goals is retrieved and executed, the success of the plans implies that the preconditions of the failed action are satisfied. When an action P_1 fails to proceed because a precondition C_1 is not satisfied, a satisfying goal G_1 for C_1 is formed. P_1 is replaced by a sequential plan containing a subgoal plan with goal G_1 followed by P_1.

When the modified plan is handed back to the plan execution module, it is inserted into the current plan. Then, the plan expansion module expands the new goal G_1 by asking the behavior retrieval module to retrieve a behavior.

Notice that the 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.

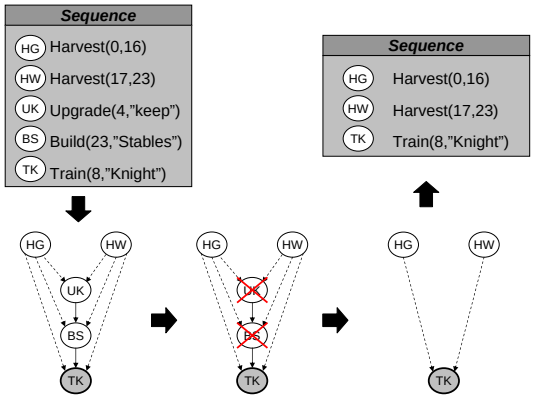

Figure 4: Exemplification of the plan adaptation action elimination process. On the top we can see the original and the adapted plan, and on the bottom we can see the generation and adaptation of the plan dependency graph.

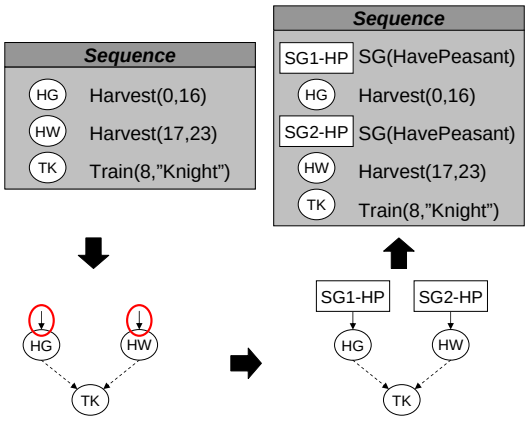

Figure 5: Exemplification of the plan adaptation precondition satisfaction process. On the top we can see the original and the adapted plan, and on the bottom we can see the generation and adaptation of the plan dependency graph.

Example

Imagine that in a particular game state the system needs to train a *knight*. To do so, the behavior shown in the top-left part of Figure 4 is retrieved, consisting of 5 steps: send a peasant to harvest gold, send a peasant to harvest wood, upgrade the townhall to a keep, then built stables, and finally train a knight. Assume that at the moment when the behavior was retrieved, the system has already built stables.

First, the plan dependency graph is constructed (shown in the bottom left of Figure 4). We can see that since the three actions *UK*, *BS*, and *TK* need gold and wood, we do not know if they depend or not on the first two actions (which harvest wood and gold), thus, the dependencies are included to be in the safe side. Moreover, in order to train knights, stables are needed, thus, there is a dependency between *BS* and *TK*. Finally, in order to build stables, we need a keep, so there is also a dependency between *UK* and *BS*.

The next step is to determine the set of *direct sub-plans*. In this case, it is *TK*, shaded to grey in Figure 4. Then, the set L of actions that can be removed is formed, and the actions in L are deleted from the plan. As we can see in the bottom middle of Figure 4 *BS* is not needed since its success conditions are already satisfied (the stables already exist), and thus *UK* is also not needed since it only contributes to *BS*. Thus, $L = \{UK, BS\}$, and *UK* and *BS* are removed from the plan, as we can see in the bottom right part of Figure 4. From the plan dependency graph, the adapted behavior is reconstructed and the final result can be seen in the top right of Figure 4, that consists only of the three actions that contribute to the achievement of the goal.

When the behavior is being executed, the adaptation module is still active. In particular, let us imagine that when the behavior in our example is being executed our system has no peasants (needed to execute the first two actions). The first two actions won't be able to be executed and that will be detected by the plan execution module, that will send the plan again to the adaptation module for further adaptation. The plan adaptation then identifies which are the preconditions that are not satisfied of the actions that cannot be executed. In our example, both actions require a peasant. Thus, the adaptation module will insert previous to each of those two actions a sub-goal consisting of having a peasant. An illustration of this process can be seen in Figure 5.

Experimental results

We conducted two sets of experiments turning the plan adaptation on and off respectively. The experiments were conducted on 10 different variations of the well known map "Nowhere to run nowhere to hide" (NWTR). 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.). The appropriate strategies for these maps depend upon multiple factors, especially the thickness of the wall of trees, some variations had openings in the wall, or others had very narrow sections that could be easily tunneled through. 7 different expert demonstration were used for evaluation; 5 of the expert traces are on different maps belonging to the set of 10 maps used for evaluation while the other 2 expert traces were for completely different "Garden of War" (GoW) maps (large maps with an empty area in the middle where a lot of gold mines are located). Each one of the expert demonstrations exemplified different strategies: "rushing" strategy, ranged attacks using ballistas or blocking the enemy from attacking using towers among others.

It is important to notice that the performance of the system greatly depends on the quality of the demonstrations, and it is part of our future work to fully evaluate that dimension. The traces used in this paper were recorded by an experienced (but non-expert) player. We conducted the experiments using different combinations of the traces. We report the results in 70 games where the system learnt from 1 trace, 2 traces, 3 traces and 6 traces at a time. We also report the results in 10 games using all 7 traces.

Table 1: Effect of Plan Adaptation on Game Statistics

No. of traces	Adaptation	Wins	Draws	Losses
1	**Yes**	**14**	**6**	**50**
1	No	14	10	46
2	**Yes**	**19**	**7**	**44**
2	No	11	10	49
3	**Yes**	**22**	**7**	**41**
3	No	21	9	40
6	**Yes**	**18**	**4**	**48**
6	No	9	4	57
7	**Yes**	**5**	**0**	**5**
7	No	1	1	8

Table 1 shows the results of the experiments. When using a single trace as the case library, plan adaptation did not lead to any improvement in the number of wins, while the number of draws decreased with plan adaptation. We observed that using adaptation, the plan adaptation component sometimes removed certain actions that seemed useless, but that had an impact further along in the game (and thus the adaptation component reinserted them again at the end), thus resulting in sub-optimal performance. Moreover, when learning from one trace, all the cases "tie in" together better, and adaptation has less value.

The results using more than one trace show a clear improvement in performance using plan adaptation. Here, cases belonging to different traces are retrieved and executed. Since the cases belong to different traces, there is a much greater chance of redundant or missing actions being present. Our plan adaptation deals with these problems, improving the performance of Darmok. When 3 traces were used the improvement in performance is not as significant. However, it was observed that most of the wins when adaptation was not being used resulted from cases retrieved from a single trace, effectively following a single trace.

In the experiment where the system learnt from all the 7 traces, we can see how the system managed to improve performance from 10% wins without adaptation to 50% wins with adaptation. When considering these numbers, we must take into account that our system is attempting to play the whole game of WARGUS at the same granularity as a human would play, taking every single decision. Moreover, it is important to emphasize that we are evaluating the performance increase of Darmok with and without plan adaptation, and not the absolute performance of Darmok.

Conclusions

We have presented on-line structural plan adaptation techniques for real-time strategy games. Specifically, our technique divides the problem in two steps: removal of unnecessary actions and addition of actions to fill gaps in the sequence of actions obtained from cases. We implemented our algorithm inside the Darmok system that can play the game of WARGUS. Our techniques are domain-independent and can be adapted for on-line plan adaptation in any domain. Moreover, one of the important aspects of our techniques is that they are efficient at the same time as effective, so they can be applied for real-time domains in which other search-based plan adaptation techniques cannot be applied.

Our techniques still have several limitations. Currently, our plan adaptation techniques require a plan to be fully instantiated in order to be adapted, thus we cannot adapt plans that are still half expanded. As a consequence, the high level structure of the plan cannot be adapted unless it's fully instantiated. This could be addressed by reasoning about interactions between higher level goals, by estimating which are the preconditions and success conditions of such goals by analyzing the stored plans in the case-base to achieve those goals. Another line of further research is to incorporate ideas from MPA in order to be able to merge several plans into a single plan. This can be very useful and can increase vastly the flexibility of the approach since sometimes no single plan in the case base can achieve a goal, but a combination will. Further formal analysis of the properties of the algorithm and of its completeness and soundness (depending on the content of the case-base) are also planned.

References

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

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

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

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

Kambhampati, S., and Hendler, J. A. 1992. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence* 55(2):193–258.

Muñoz-Avila, H., and Cox, M. 2007. Case-based plan adaptation: An analysis and review. *IEEE Intelligent Systems*.

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

Nebel, B., and Koehler, J. 1992. Plan modifications versus plan generation: A complexity-theoretic perspective. Technical Report RR-92-48.

Ontañón, S.; Mishra, K.; Sugandh, N.; and Ram, A. 2007. Case-based planning and execution for real-time strategy games. In *Proceedings of ICCBR-2007*, 164–178.

Ram, A., and Francis, A. 1996. Multi-plan retrieval and adaptation in an experience-based agent. In Leake, D. B., ed., *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. AAAI Press.

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