In 9th European Conference on Case-Based Reasoning (ECCBR 2008), Trier, Germany

Real-Time Plan Adaptation for Case-Based Planning in Real-Time Strategy Games

Neha Sugandh, Santiago Ontañón, and Ashwin Ram

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

Abstract. Case-based planning (CBP) is based on reusing past successful plans for solving new problems. CBP is particularly useful in environments where the large amount of time required to traverse extensive search spaces makes traditional planning techniques unsuitable. In particular, in real-time domains, past plans need to be retrieved and adapted in real time and efficient plan adaptation techniques are required. We have developed real time adaptation techniques for case based planning and specifically applied them to the domain of real time strategy games. In our framework, when a plan is retrieved, a plan dependency graph is inferred to capture the relations between actions in the plan suggested by that case. The case 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. Our techniques have been implemented in the Darmok system (see [8]), designed to play WARGUS, a well-known real-time strategy game. We analyze our approach and prove that the complexity of the plan adaptation stage is polynomial in the size of the plan. We also provide bounds on the final size of the adapted plan under certain assumptions.

1 Introduction

Traditional planning techniques are inapplicable in real-time domains with vast search spaces. Specifically, we are interested in real-time strategy (RTS) games that have huge decision spaces that cannot be dealt with search based AI techniques [1]. Case-based planning (CBP) can be useful in such domains since they can potentially reduce the complexity of traditional planning techniques. CBP techniques [10] work by reusing previous stored plans for new situations instead of planning from scratch. However, plans cannot be replayed exactly as they were stored in any non trivial domain. Therefore, CBP techniques require plan adaptation to adapt the information contained in plans. More specifically, CBP techniques for RTS games need adaptation techniques that are suitable for dynamic and unpredictable domains, and that have a low complexity to be useful for real-time situations. In this paper we present Darmok, a case-based planning

Fig. 1. A screenshot of the WARGUS game.

architecture that integrates planning and execution and is capable of dealing with both the vast decision spaces and the real-time component of RTS games. Then, we will focus on the problem of how to adapt plans stored in the knowledge base of our system to suit new situations in real-time. We further analyze the algorithms and establish bounds on their complexity, thus proving that the algorithms presented are suitable for a real-time situation.

It is hard to approach RTS games using traditional planning approaches: RTS games have huge decision spaces [1], they are adversarial domains, they are non-deterministic and non fully-observable, and finally it is difficult to define postconditions for actions (actions don't always succeed, or take a different amount of time, and have complex interactions that are difficult to model using planning representation formalisms). To address these issues, we developed Darmok [8], a case-based planning system that is able to deal with domains such as WARGUS. We apply our plan adaptation techniques to Darmok.

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). In this paper, we will present a domain independent and search-free structural plan adaptation technique based on two basic ideas: a) removing useless operations from a plan can be done by analyzing a *dependency graph* and b) the insertion of new operations in the plan can be delegated to the case-based planning cycle itself. Thus, the plan adaptation will state that some new operations to achieve a particular goal must be inserted, and the CBP engine will generate a plan for that goal. Our plan adaptation approach has been implemented in the Darmok system with promising results.

In the rest of this paper we introduce the Darmok system in Section 2, and then we focus on plan adaptation in Section 3. Then, we analyze the complexity of the adaptation algorithms in Section 4. After that, we report experimental results in Section 5. The paper closes with related work and conclusions.

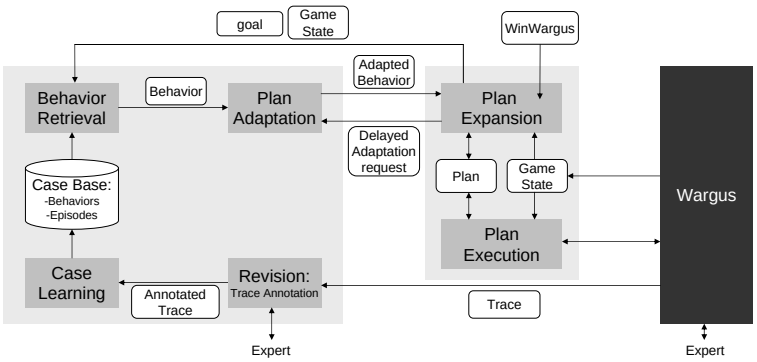

Fig. 2. Overview of the Darmok system.

2 Case-Based Planning in WARGUS

Figure 1 shows a screen-shot of WARGUS, a RTS 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 this section we will briefly describe the Darmok system, in which we have implemented our plan adaptation techniques. In order to play WARGUS 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:

- *Plan Learning*: performed by the *Revision* and *Case Learning* modules. Each time a human plays a game, a trace is generated (containing the list of actions performed in the game). During revision, the human annotates that trace stating which goals he was pursuing with each action. This annotated trace is processed by the case learning module that extracts plans in form of cases.
- *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 *Behavior Retrieval* module, that retrieves the most appropriate behavior to fulfill an open goal. Finally, the *Plan Adaptation* module adapts the retrieved plans.

Cases in Darmok consist of two parts: *behaviors* and *episodes*. A Behavior contains executable code to achieve a particular goal, and an episode contains information on how successful a behavior was in a particular situation.

A behavior has two main parts: a *declarative* part and a *procedural* part. The declarative part has the purpose of providing information to the system about

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

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

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* (primitive 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*. For instance, *AbsoluteHaveUnits(TOWER,1)* is a valid goal in our gaming domain that has the following success condition: *UnitExists(TOWER)*.

2.1 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* that the system is executing. A *partial plan tree* in our framework is represented as a tree consisting of *goals* and *behaviors* (similar to HTN planning [6]). Initially, the plan consists of a single goal: "win the game". Then, the plan expansion module asks the behavior retrieval module for a behavior for that goal. That behavior might have several subgoals, for which the plan expansion module will again ask the behavior retrieval module for behaviors, and so on. When a goal still does not have an assigned behavior, we say that the goal is *open*.

Additionally, each behavior 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 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 retrieval module. The retrieved 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:

- Pending behaviors with satisfied preconditions change status to executing.
- Basic actions that are ready and with all their 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 retrieval module retrieved it, the behavior is handed back to the plan adaptation module.

In the remainder of this paper we will focus on the plan adaptation component. See [8] for a more detailed explanation of the rest of the system.

3 Real-Time 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 [8] for an explanation on how that module works). In this section we will focus on the structural plan adaptation module.

We specifically consider plans which are only composed of actions, sequential constructs and parallel constructs. This implies that we consider only those plans which are completely expanded and do not contain a sub-goal which further needs to be expanded. We generate a *plan dependency graph* using the preconditions and success conditions of the actions. The structural plan adaptation process has two sub-processes: elimination of unnecessary actions, and insertion of required 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.

3.1 Plan Dependency Graph Generation

Figure 3 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 3). 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 3. For any action p' when the function *FindDependencies* is called D contains exactly the set of actions on which p' might be dependent. The set of primitive actions for

```
Function GeneratePlanGraph(p, D)
    G = ∅
    ForEach p' ∈ p.subPlans
        If p' is sequential or parallel Then
            G = G ∪ GeneratePlanGraph(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
End-Function

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

Fig. 3. Algorithm for Plan Dependency Graph Generation. Where p is the plan to be adapted, and D is the set of plans on which any sub-plan in p might depend (and it is 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.

a subplan p' of p are added to D only if p is a sequential construct. The recursive call to *GeneratePlanGraph* 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 a plan p with success conditions of another plan 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 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 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. 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.

```
Function RemoveRedundantPlans (p, g)
    B = GetDirectActions(p, g)
    G = GeneratePlanGraph(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' = GetParentPlans(p, G)
            A = BackPropagateActivePlans(B', G, A)
        EndIf
    EndForEach
    Return A
EndFunction
```

Fig. 4. Algorithm for Removal of Unnecessary Actions. Where p is the plan to be adapted, and g is the goal corresponding to p. GetParentPlans(p, G) is a simple function that returns all the plans that have a causal direction with a given plan p, according to a graph G. GetDirectActions(p, g) is a function that returns those primitive actions in p that are *direct actions*.

3.2 Removal of unnecessary actions

Figure 4 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. This is done by the function call to *GetDirectActions* in Figure 4. These actions are called *direct actions* for the subgoal. Then the plan dependency graph for p is generated using the *GeneratePlanGraph* function in Figure 3. 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 plans are obtained using the function *GetParentPlans* in the algorithm (that can be easily 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

```
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
End-Function
```

Fig. 5. Algorithm for Adding Goals for Unsatisfied Preconditions, where p is the primitive 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.

have a dependency to a direct plan, also with success conditions not satisfied in the given game state. Actions that are not active (not in A) are removed.

3.3 Adaptation for unsatisfied preconditions

Figure 5 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. The 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. Initially, all the unsatisfied preconditions of the action 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* in Figure 5. 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 module will expand the newly inserted goals in G.

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.

4 Complexity Analysis

In the following sections we will analyze the complexity of our structural adaptation techniques. We analyze both the removal of redundant actions through plan dependency graph generation as well as the addition of goals to the partial plan to satisfy unsatisfied conditions. In the first case the time complexity of plan dependency graph generation and removal of actions is obtained. In the case of goal additions, the goal addition for a single condition happens in constant time. Here, the time complexity is not as important as the number of goals added during a game play, because the addition of goals has an impact on the size of the plan for winning the game and thereby on the time taken to win the game. We obtain a bound on the number of goals that are added during the plan adaptation stage for satisfying any unsatisfied preconditions.

4.1 Complexity of Removal of Unnecessary Actions through Plan Dependency Graph Generation

Theorem 1. *The time complexity for removal of unnecessary actions through plan dependency graph generation is $O(N(n))$, where $N(n) = kl(n-1)(n-2)/2 + l^2n + (n-1)(n-2)/2$. Where n is the size of the plan, k is the maximum number of pre-conditions and l is the maximum number of success conditions of actions.*

We derive Theorem 1 in the following discussion. A plan dependency graph is generated for a plan which has been expanded to the level of primitive actions. Let the number of pre-conditions in any action be bounded by k and the number of success conditions for any subgoal or action be bounded by l. The maximum number of comparisons that can occur while comparing the preconditions of any action with the success conditions of another action is bounded by $N_c^{max} = kl$.

Now consider a retrieved plan with n actions. If the plan is a sequential plan, to obtain the dependencies we compare the preconditions of each action with the success conditions of the preceding actions. Thus the preconditions of the second action are compared with the success conditions of the first action, the preconditions of third action are compared with the success conditions of the first action and the second action and so on. The total number of comparisons at the level of plans is thus:

$$N_c^P(n) = 1 + 2 + ... + (n-1) = \frac{(n-1)(n-2)}{2} \tag{1}$$

In case the plan retrieved is a parallel plan we do not try to obtain the dependencies as we assume that the component plans are independent of each other. When the retrieved plan is a combination of sequential plans and parallel

plans, some comparisons take place but the number of comparisons will be less than that in the case of a sequential plan. The sequential plan thus provides an upper bound on the number of comparisons. Within each action comparison, there can be N_c^{max} condition comparisons and these condition comparisons take constant time.

When obtaining the direct actions we compare the success conditions of each primitive action with the success conditions for the goal. The maximum number of condition comparisons possible here is l^2 and this is done for each of the n actions. Thus the number of condition level comparisons for obtaining direct plans is bounded by $N_d^{max}(n) = nl^2$.

Once the dependencies between plans have been determined we remove redundant actions. Only those direction actions with unsatisfied success conditions are initially considered active. Then we propagate it to the plans on which the direct plans depend, we also recursively do this for the plans made active. We do the propagation only once for each action. If the total number of actions is n, and the position of an action in a sequential plan is i, the maximum number of actions the action can depend upon is $i-1$. Thus, the number of links we follow is bound by $N_l^{max}(n)$.

$$N_l^{max}(n) = 1 + 2 + ... + (n-1) = \frac{(n-1)(n-2)}{2} \tag{2}$$

The complexity of adaptation through plan dependency graph generation is thus $O(N(n) = N_c^{max} * N_c^P(n) + N_d^{max}(n) + N_l^{max}(n))$, proving Theorem 1.

4.2 Analysis of Adaptation for Unsatisfied preconditions

Considering the maximum number of preconditions for any action to be k and that each precondition when not satisfied leads to the addition of a single goal, the maximum number of goals inserted to satisfy an action's preconditions is also k. Each goal is expanded into a plan. The primitive actions present in this new plan can further have unsatisfied conditions during execution. This may lead to the creation of cycles i.e it is possible that a goal g_1 has a precondition c_1 which leads to goal g_2 and the goal g_2 has a precondition c_2 which leads to the goal g_1, it might lead to the continuous addition of goals. No bound regarding the size of the final partial plan can be obtained if such cycles can occur (Darmok incorporates a simple cycle detection mechanism that prevents these situations).

Theorem 2. *Assuming all plans succeed upon execution and goals do not form cycles, the number of goals added by the adaptation module is* $O(M_G^{max})$*, where:* $M_G^{max} = n_{max}k * (n_{max}^{G-1}k^{G-1} - 1)/(n_{max}k - 1)$*. Where* n_{max} *is the maximum size of any plan in the case base,* k *is the maximum number of preconditions in any plan and* G *is the number of different goals possible in the domain.*

If the goals in a real-time planning system cannot form a cycle i.e any plan for a goal g_1 will never lead to a goal g_2 such that the plan for g_2 leads to the goal g_1, a bound can be established on the number of goals added. If the total number

of possible different goals in the system is G, the number of possible goals a plan for a top level goal g can lead to is $G-1$. Let g' be one such goal. The number of goals a plan for this goal can lead to is $G-2$, and so on. Additionally, the number of goals added for any action is limited by k. Consider the maximum number of actions in any fully expanded plan for a goal to be n_{max}. If the goals added by plan adaptation for the first time are considered level one goals, the goals added within a plan for a first level goal as second goals, and so on, the maximum number of goals that can be added at level l for a level $l-1$ goal is $n_G^l = min(G-l, k)$. The maximum number of goals added is N_G^{max}:

$$N_G^{max} = \sum_{i=1\ldots G-1} \left((n_{max})^i \prod_{j=1\ldots i} n_G^j \right) \tag{3}$$

If $k < G$ and we replace n_G^l by k we get an upper bound on N_G^{max}, $N_G^{max} < M_G^{max}$:

$$M_G^{max} = \sum_{i=1\ldots G-1} (n_{max})^i k^i = n_{max} k * \frac{((n_{max})^{G-1} k^{G-1} - 1)}{(n_{max} k - 1)} \tag{4}$$

Thus the number of goals added by the adaptation module is bounded by M_G^{max}. This is clearly not a tight upper bound. Better upper bounds can be obtained introducing domain related constraints.

In the case of WARGUS domain, the goals inserted by the plan adaptation module are either to build certain units or buildings or to gather resources. Consider the term *units* to refer to all units other than peasants and the term *buildings* to refer to all buildings other than farms (peasants and farms need to be considered separately). For the further analysis we assume that the opponents have not destroyed any buildings. Let n_0 be the number of primitive actions in the completely expanded plan without adaptation. Let b, f, u and p be the number of buildings, farms, units and peasants inserted by the adaptation module respectively. Farms are required to train peasants and units, each farm allows training of four peasants and units. Considering each of the n_0 actions can produce at most one peasant or unit, the number of units trained is at most $n_0 + u + p$. We know that before executing the plan, the number of farms was enough for the number of units and peasants we had. Thus, the number of farms f that the adaptation component will insert must satisfy the following inequality (because it will not insert more farms than needed): $4f \leq n_0 + u + p$.

Now, consider the peasants inserted by the adaptation module. Peasants are trained to gather resources or build buildings or farms. There are three kinds of resources in WARGUS:wood, gold and oil. Farms require only two of these resources(wood and gold), while a building or unit might require any of the three resources. Farms and buildings also require peasants for building them. Thus, a building can require 4 peasants - one to build it and three to gather the different resources. Similarly, a farm or a unit can require 3 peasants. The maximum number of peasants required by the n_0 primitive actions is $4n_0$. Thus we get

the following inequality (because it will not insert more peasants than needed): $p \leq 4n_0 + 4b + 3f + 3u$.

Solving these inequalities gives us: $f \leq (5n_0 + 4b + 4u)$ and $p \leq (19n_0 + 16b + 15u)$. Further, units can only be required by one of the n_0 actions, since they are not required for building or gathering resources. Thus, $u \leq n_0$. The number of goals added is thus $O(N_g)$:

$$N_g = b + u + f + p \leq 24n_0 + 21b + 20u \leq 44n_0 + 21b \tag{5}$$

Thus, the number of extra goals added due to plan adaptation is $O(21b + 44n_0)$. Notice also that b is bounded by the number of different buildings (other than farms), because adaptation will never insert duplicate buildings since duplicate buildings are never *required* (although it might be convenient to have them, it is never required). Thus the number of goals added is linearly bounded in the number of building types and the number of actions which were originally present in the plan. The analysis presented, provides a bound on the size of the plan after the adaptation module has finished adapting it, assuming that there are no goal cycles (easily detected in the case of WARGUS).

4.3 Single Cycle Complexity

In order to have a real-time system, it is important that a single execution cycle takes a short time. Each cycle of Darmok involves the expansion of open goals and then execution of actions ready to execute. If the number of open goals is r and the number of actions ready to execute is e, r plans are retrieved and the plan dependency graph is generated for these plans. If n_{max} is the maximum number of actions in any of the retrieved plans, the complexity for the adaptation of retrieved plans is $O(rN(n_{max}))$ (See Theorem 1). While plan execution, adaptation due to failed preconditions of an action can lead to the addition of maximum k goals in a single cycle. Thus the total per cycle complexity of the Darmok planning adaptation module is polynomial, $O(rN(n_{max}) + ke)$.

5 Experimental results

To evaluate our plan adaptation techniques, we conducted two sets of experiments turning the plan adaptation on and off respectively. The experiments were conducted on 12 maps: 11 different variations of the well known map "Nowhere to run nowhere to hide" (NWTR) and 1 version of "Garden of War" (GoW). 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.). GoW maps are large maps with an empty area in the middle where a lot of gold mines are located. 10 different expert demonstration were used for evaluation; 8 of the expert traces are on maps from NWTR maps while the other 2 expert traces were for GoW maps. Each one of the expert demonstrations exemplified

Table 1. Effect of Plan Adaptation on Game Statistics

	Adaptation						No Adaptation						
NT	*W*	*D*	*L*	*ADS*	*AOS*	*WP*	*W*	*D*	*L*	*ADS*	*AOS*	*WP*	*improvement*
1	17	4	27	2158	1514	**35.42%**	9	7	32	1701	1272	**18.75%**	88.75%
2	16	5	27	2798	1828	**33.33%**	15	2	31	1642	1342	**31.25%**	6.66%
3	18	6	24	1998	1400	**37.5%**	10	6	32	1633	1491	**20.83%**	80.00%
4	19	3	26	2343	1745	**39.58%**	8	4	36	1358	1663	**16.67%**	137.40%
5	11	6	31	2141	1842	**22.91%**	7	6	35	1310	1607	**14.58%**	57.13%
6	14	2	32	1709	1695	**29.17%**	3	5	40	1475	1788	**6.25%**	366.72%
7	20	0	28	1941	1448	**41.67%**	9	6	33	1800	1564	**18.75%**	122.24%
8	15	3	30	1887	1465	**31.25%**	6	3	39	1598	1671	**12.50%**	150.00%
9	21	4	23	2110	1217	**43.75%**	7	3	38	1449	1681	**14.58%**	200.07%
10	5	0	7	1533	1405	**41.67%**	2	0	10	1158	1555	**16.67%**	150.00%
	156	33	255	20618	15559	35.14%	76	67	301	15124	15634	17.12%	105.38%

different techniques with which the game can be played: fighter's rush, knights rush, ranged attacks using ballistas, or blocking the enemy using towers.

We conducted the experiments using different combinations of the traces. We report the results in 12 games (one per map) using all 10 traces. We also report the results in 48 games in nine different scenarios where the system learnt from 1, 2, 3, 4, 5, 6, 7, 8 and 9 traces respectively. For conducting these experiments, for any number of traces, n, we randomly chose four sets containing n traces and ran our system against the built in AI with each set on all 12 maps.

Table 1 shows the results of the experiments with and without adaptation. NT indicates the number of traces. For each experiment 6 values are 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 adaptation with respect to no adaptation. The bottom row shows a summary of the results.

The results show that plan adaptation 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. When several traces are used cases belonging to different traces are retrieved and executed, thus, 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. In some cases the system performs well even without adaptation. This may be because the cases retrieved "tie in" together as they are and do not require adaptation. For instance, in the experiment where the system learnt 10 traces, we can see how the system managed to improve performance from 16.67% wins without adaptation to 41.67% wins with adaptation. Finally, 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, and that also results depend on the quality of the demonstration traces provided to the system. Thus, with better demonstrations (by true experts), the performance could greatly improve.

6 Related Work

Case-based planning is the application of CBR to planning, and as such, it is planning as remembering [2]. CBP involves reusing previous plans and adapting them to suit new situations. There are several motivations for case-based planning [10], the main one being that it has the potential to increase the *efficiency* with respect to generative planners (although, in general, reusing plans has the same or even higher complexity than planning from scratch [7]).

One of the first case-based planning systems was CHEF [2], 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, it 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 [4] 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 [3]. The key highlight of SPA is that it is complete and systematic (while PRIAR is not systematic, and CHEF is not either complete nor systematic), but uses a simpler plan representation than PRIAR. Extending SPA, Ram and Francis [9] 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. A thorough review on plan adaptation techniques was presented in [5].

7 Conclusions

In this paper we have presented real-time structural plan adaptation techniques for RTS 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. We implemented our algorithm inside the Darmok system that can play the game of WARGUS. The experiments conducted gave promising results for the techniques introduced, however our techniques are domain-independent. 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. The complexity analysis performed shows that the adaptation techniques do not have a significant overhead and are suitable for real time situations.

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 is fully instantiated. Because of that, plan adaptation as presented in this paper can only work at the lower levels of the plan, where everything is instantiated. This could be addressed by reasoning about interactions between higher level goals, by estimating which are the preconditions and postconditions 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 [9] in order to be able to merge several plans into a single plan. This can increase the flexibility of the approach since sometimes no single plan in the case base can achieve a goal, but a combination will.

References

1. 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.
2. Kristian F. Hammond. Case based planning: A framework for planning from experience. *Cognitive Science*, 14(3):385–443, 1990.
3. Steve Hanks and Daniel S. Weld. A domain-independednt algorithm for plan adaptation. *Journal of Artificial Intelligence Research*, 2:319–360, 1995.
4. Subbarao Kambhampati and James A. Hendler. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, 55(2):193–258, 1992.
5. Héctor Muñoz-Avila and Michael Cox. Case-based plan adaptation: An analysis and review. *IEEE Intelligent Systems*, 2007.
6. 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.
7. Bernhard Nebel and Jana Koehler. Plan modifications versus plan generation: A complexity-theoretic perspective. Technical Report RR-92-48, 1992.
8. 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.
9. 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.
10. L. Spalazzi. A survey on case-based planning. *Artificial Intelligence Review*, 16(1):3–36, 2001.