Combining Expert Knowledge and Learning from Demonstration in Real-Time Strategy Games ⋆

Ricardo Palma, Antonio A. Sánchez-Ruiz, Marco A. Gómez-Martín,
Pedro P. Gómez-Martín and Pedro A. González-Calero

Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
rjpalma@estumail.ucm.es, {antsanch,marcoa,pedrop,pedro}@fdi.ucm.es

Abstract. Case-based planning (CBP) is usually considered a good solution to solve the knowledge acquisition problem that arises when developing AIs for real-time strategy games. Unlike more classical approaches, such as state machines or rule-based systems, CBP allows experts to train AIs directly from games recorded by expert players. Unfortunately, this simple approach has also some drawbacks, for example it is not easy to refine an existing case base to learn specific strategies when a long game session is needed to create a new trace. Furthermore, CBP may be too reactive to small changes in the game state and, at the same time, do not respond fast enough to important changes in the opponent's strategy. We propose to alleviate these problems by letting experts to *inject* decision making knowledge into the system in the form of *behavior trees*, and we show promising results in some experiments using Starcraft.

1 Introduction

Real-time strategy (RTS) games are very demanding in terms of AI complexity. They require fast pathfinding algorithms for moving large numbers of units through extensive levels, which need to be manually or procedurally annotated with tactical information. Regarding decision making, RTS require a multi-tiered AI approach, with decisions made at a low level for individual characters, at an intermediate level for a formation of characters, and at the high level of a whole side in the game. Usually simple techniques, such as state machines, are used for low level decision making, while some form of rule-based system is the most common approach for decision making at higher levels [8] .

Building a rule-based system for decision making at the tactical and strategic level of an RTS game is a complex task for game designers, who are not used to deal with knowledge representation. In order to alleviate this authoring effort, there is an open line of research on the automatic acquisition of decision making knowledge from recorded traces of human experts playing the game. Such approaches, through the application of machine learning techniques, seek

⋆ Supported by the Spanish Ministry of Science and Education (TIN2009-13692-C03-03).

to make possible a form of *programming by demonstration*, where the human author shows to the game AI how to play.

We propose to bring the game designer back into the loop, by allowing him to explicitly inject decision making knowledge in the form of behavior trees to complement the knowledge obtained from the traces. Behavior trees are the technology of choice for representing decision making knowledge in commercial videogames. They can be built by both programmers and designers, and provide the ability to react to urgent goals or a fine-grained control over alternative courses of action.

The work presented in this paper seeks to extend the techniques demonstrated in the Darmok and Darmok 2 (D2) systems [10, 11], which are mature and well-tested systems within this line of research. We show through an experimental evaluation in the Starcraft game, how we can easily increase the efficacy of a case-based planner significantly. Moreover, we show that very simple behavior trees can make a big difference in terms of the AI quality.

The rest of the paper runs as follows. Sections 2 and 3 briefly describe the two technologies we propose to integrate: learning from demonstration and behavior trees. Section 4 discusses the integration of both technologies into a single architecture, which is empirically evaluated by means of some experiments described in Section 5. Last section presents related work and concludes the paper.

2 Learning from Demonstration in Real-Time Strategy Games

Learning from demonstration tries to replace the time-consuming and handmade task of programming behaviors by an automatic process in which an expert shows the system how to achieve some goal. Afterwards, a learning system analyses the actions performed by the expert and tries to extract useful knowledge. This type of techniques has traditionally been applied to build behaviors in robots, but the complexity of current video games makes them perfect candidates to apply these techniques as well.

There are many different approaches to extract decision making knowledge from the traces of expert players, for example to consider sequences of actions as reusable *plans*. In *case based planning* (CBP) these plans are stored in a plan base for later reuse, and the planner retrieves and adapts the most promising one according to the current goal and state of the problem. In our context, the goal is usually something like *win* and the state of the problem is the state of the game.

Being games dynamic, static CBP suffers in this context from lack of reactivity. In order to respond properly to the continuous changes in the game state, CBP systems usually include an *on-line* component that supervises the execution of the proposed plan. This way, instead of just considering the planning problem as a single-shot process where execution and problem solving are decoupled, *on-line case-based planning* (OLCBP) systems are also partly in charge of the plan execution. Note that OLCBP systems can discover low level aspects

at the plan expansion stage that were not considered originally and which might require to perform some changes in order to prevent failure.

Although planning in OLCBP is performed in *real-time*, planning decisions are based on an expensive *learning* process that is made previously off-line from game *traces* produced by human expert players. Depending on the CBP engine, the learning phase might require the assistance of a human expert to *label* the traces in order to enrich them with extra knowledge about, for example, pursued goals or important game state variables. The other option is to make the system intelligent enough to perform the labelling task automatically and thereby avoid that tedious task to the expert. Unfortunately, there is no magic bullet here; OLCBP systems using this last approach usually require a more detailed domain description in order to extract useful plans from plain traces.

Once the plan base has been fed with plans (cases) extracted (automatically or not) from traces, it is time for the *on-line* component of the system to take control. The game will be launched as normal and connected with an external AI that, in our architecture, is implemented using an OLCBP system. The game engine will broadcast its state using a game-dependent protocol, and, on the other way, the OLCBP system will inject the primitive actions that should be executed in the game environment.

One of such OLCBP systems is Darmok, presented in [11]. It has been used for playing real-time strategy games like Wargus (an open source clone of Warcraft II) and other games. As we will see in a later section, the combination of domain knowledge automatically extracted from game traces with hand-made expert knowledge is an effective solution to some of the problems identified in this type of OLCBP.

3 Expert knowledge in games

Regarding how to create behaviors for the non-player characters that appear in videogames, many techniques have been used. The goal here is to translate the knowledge that game experts have (those who know the best way to defeat the other player) into instructions to be executed by the machine. The approach is different depending on the level of abstraction and game genre but when focusing on low level decision making for controlling units on strategy games the most commonly used techniques have been finite state machines (FSMs) and more recently behavior trees (BTs).

Though BTs were initially proposed as a tool for programmers, they have been proved to be useful also for professional *game designers* to create the behaviors of the entities from scratch [4, 5]. One of the key points is that, just like FSMs, BTs open up the possibility of developing tools to create and edit behaviors by means of a graphical user interface, something that cannot be done in other techniques like scripts.

A BT is a hierarchical and declarative representation of a behavior where every node (and its subtree) can be seen as a simpler behavior. Therefore, BTs

promote reuse and allow designers to define complex behaviors in an incremental way.

Using visual tools, game designers and game programmers create the BTs that eventually will define the behavior of the game AIs. The available toolset for this authoring process depends on the actual AI engine implementation and every game studio has its own set of tree nodes that can be combined to encode complex behaviors from simpler ones. There are, therefore, many types of BT nodes but we will only cover those we have used in our experiments.

During the execution of the game, the behavior trees are loaded into memory and a module known as the *BT interpreter* controls its execution. The BT interpreter manages which branch of the current BT is *active* (complex systems may allow BTs to have more than one branch in execution) and how much CPU quantum should be assigned to each node. In addition, when a node complete its execution either successfully or failed, the BT interpreter decides, according to the semantics of its parent node in the tree, which branch should be expanded next.

In pure systems all the decisions are performed based on BTs, while mixed systems may have a higher level AI module to select the best BT for each entity according to some global policy or strategy. This high level module must receive continuous notifications to properly manage the behavior of the different entities in a coordinated manner. In turn, the BT execution model might rest either on a single global BT interpreter or there might be several BT interpreters, one for each entity. As we will see in Section 4, our system has been implemented using a mixed architecture and a single BT manager.

Regarding the types of behavior a BT may encode, its expressiveness and versatility depends on the amount and types of nodes available at design time. The simplest BT is compound by just one node representing a *primitive action*, that is, a game action to be executed by one character. These primitive actions may have parameters whose values are either hard-coded on the BT itself or computed at run time depending on the context. When the BT interpreter decides to execute one of these nodes, the primitive action is sent to the game engine and the interpreter must wait until its execution ends either with success or failure. Note that although from the point of view of the AI system the action is atomic, in a real-time game its execution might take several seconds. Besides, the action can fail if something happens during that time that prevents its execution.

The simplest way of combining behaviors is using *sequential nodes*, inner nodes whose child nodes (or subtrees) are executed sequentially. When all of them finish successfully their execution, the behavior succeeds, failing otherwise. Sequential nodes are usually represented as a node with an arrow inside.

Dynamic priority lists bring more expressive power to BTs, adding the idea of *conditions*. Every child node has not only a behavior but also a guard or condition that checks some aspect of the game state that is relevant for the sub-behavior. When the interpreter reaches a dynamic priority list, evaluates the guards of the children nodes from left to right and starts the execution of the

first behavior whose guard is met. If no condition is met, the entire behavior fails. Once a child node is active, the interpreter keeps evaluating the conditions of the left siblings and whenever one of them becomes true, the current behavior is aborted and the child with the higher priority takes control. In the figures shown later in the paper guards appears as nodes in dotted lines with one child representing the behavior that must be executed when the guard becomes true.

The last node we consider is the *query node* [3] that promotes even further reusability. All the nodes described so far are hard-coded in the BT, but this new node allows to dynamically attach BTs as subtrees of other BTs at run time. This way, BTs can delegate to achieve a particular goal to other BTs without having to specify to which one in particular. The idea is that different BTs are created independently during the game design phase and stored in a behaviors base and, at run time, the query nodes serve as joint points to combine behaviors like assembly parts.

The technology behind query nodes is borrowed from case-based reasoning. The behavior base may be seen as a case base storing different solutions to achieve a goal. These BTs are described using a semantic label from a behavior ontology, a set of variables and constraints which encode the game states where the BT is optimal, and the entities the BT is meant to (this is needed because a soldier may have a different behavior to protect/cover itself than a tortoise).

The *query node* in turn contains the CBR query which describes the desired behavior along with a number of variables and constraints using the same vocabulary that was used to index the behavior base.

At run time the query is extended with the current game state and the current entity executing the BT and, using similarity measures similar to those used on CBR, it retrieves the best BT for the current situation. Interested reader may refer to [3] for more details. As we will see later, we have used these nodes in different experiments. In the rest of the paper, this kind of nodes is drawn using thick lines.

4 Extending Case-Based Planning with Behavior Trees

Three main flaws [12] have been identified in on-line case based planning when used with plan extraction from expert traces:

- *Poor reactivity at the plan level:* when a plan is chosen, it will not be abandoned until it is completely performed or a *low level action* fails. World changes that should fire a new replanning phase are usually ignored because they are not recognized as a failure condition for the current action in the *working* plan.
- *Excessive reactivity at the action level:* when a small low level action fails, a big plan could be entirely discarded because the planner would assume that also the *complete plan* has failed and cannot be fixed. A new planning phase would be fired, which could end up with a complete opposite approach for the current goal. In other words, small local problems could have unreasonable big responses.

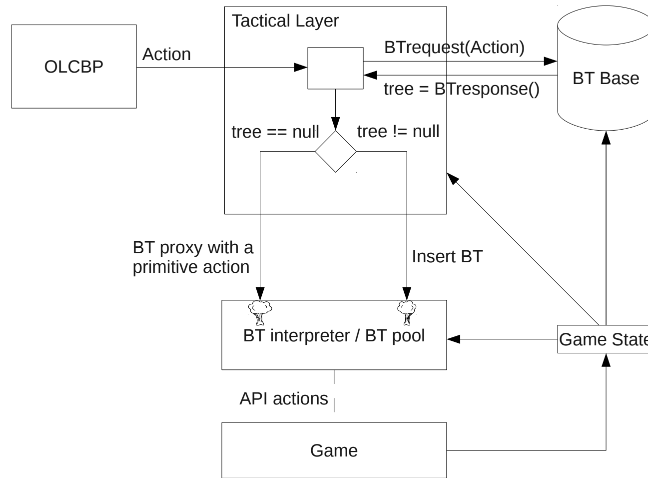

Fig. 1. Low level tactical layer

- *Learning by demonstration is hard to fine-tune:* When an expert identifies behaviors that should be improved, the nature of learning by demonstration and the fact that he has to provide new traces to have the new behavior learnt, makes the process really hard. This is due to the complexity and randomness inherent in strategy games, where it is difficult to get a suitable scenario where the expert's actions are relevant enough for being incorporated as a plan in the plan base.

Our solution is to incorporate expert knowledge into the process. This knowledge takes the form of behavior trees presented in the previous section. As we will describe shortly, BTs have such an important role in our approach that the BT interpreter is the only one that injects primitive actions into the game. This interpreter manages a *BT pool* that contains all the BTs that are currently in execution.

Between the OLCBP and this *BT pool* we place a *tactical layer*. The planner still emits primitive actions to be executed, but instead of directing them into the game they pass through this layer. Its functionality is based on a BT base that is populated by behaviors built previously by the experts. When the OLCBP wants to execute an action, the tactical layer retrieves the most suitable BT that performs that action in the current game state and sends it to be executed by the interpreter, adding it to the BT pool. If there is no such a BT, it builds a small BT with just the node that executes that action.

The tactical layer and the BT base allow the expert to *overwrite* the execution of primitive actions when needed. For example, there will be specific situations where a primitive action should be enriched with a more *error-safe* plan that is able to gracefully react to some small situations that would cause the primitive action to fail. This would avoid the problem of excessive reactivity at the action

level, but, unfortunately, *training* (and teach) the CBP system to use these so concrete plans is quite hard (if possible at all). This extension to the basic architecture where only the game and OLCBP system coexist does not require changes in any of them. Figure 1 shows a simplified view of the architecture so far. The OLCBP system emits a primitive action to the tactical layer which tries to get a BT through a query to the BT base. Should the query return *null*, the original primitive action is packed on a BT containing a single node. In other case the BT is inserted into the interpreter that manages its execution sending a primitive action to the game.

A second extension of pure OLCBP systems consists on allowing experts to overwrite *complete* plans. The idea is similar to that of low level actions but requires changes on the OLCBP system. The BTs created by the expert contain a description of the game state specifying when it is appropriate to use them. When the OLCBP system in its plan expansion reach a new goal, instead of just retrieving the most similar plan stored in the plan base, it performs a query to the BT base to check if a hand-made BT has been created for that specific situation. If such a BT exists the goal is replaced by it, proceeding through the plan base otherwise.

Therefore, with this extension the output of the OLCBP system may be just a primitive action (that goes through the tactical layer and it is converted to a BT) or an entire BT that is directed to the BT interpreter.

This extension aims to solve the poor reactivity at the plan level problem, because *BT guards* keep an eye on the high-level game conditions that makes the plan suitable for running, and fires a new replanning phase when they are not longer met, even if the primitive actions are not failing.

Figure 2 shows the detailed architecture regarding this last extension. When the plan expansion module is processing the plan for goal 1 and detects goal 3, instead of directly trying to use the plan library, it firstly query the BT base looking for a BT created by the expert suitable for the current situation of the game. In that case there is a BT that replaces the goal 3 and which eventually will be sent to the tactical layer to be inserted in the BT interpreter.

5 Experiments

Starcraft [2] is a famous real-time strategy game that has captivated millions of players in the last decade. Although the goal of the game is very simple (to build an army and to defeat the other players), this game offers hundreds of hours of fun thanks to the huge number of different strategies that can be created combining different types of units and technologies.

Players can choose among three different races (*Terran*, *Protoss* and *Zerg*), each one of them with its unique strengths and weaknesses. The chosen race will determine the type of units and technologies that will be available during the game. In order to win, players have to wisely manage a limited number of resources and invest them to get more resources, to develop new technology or to build defensive and offensive units.

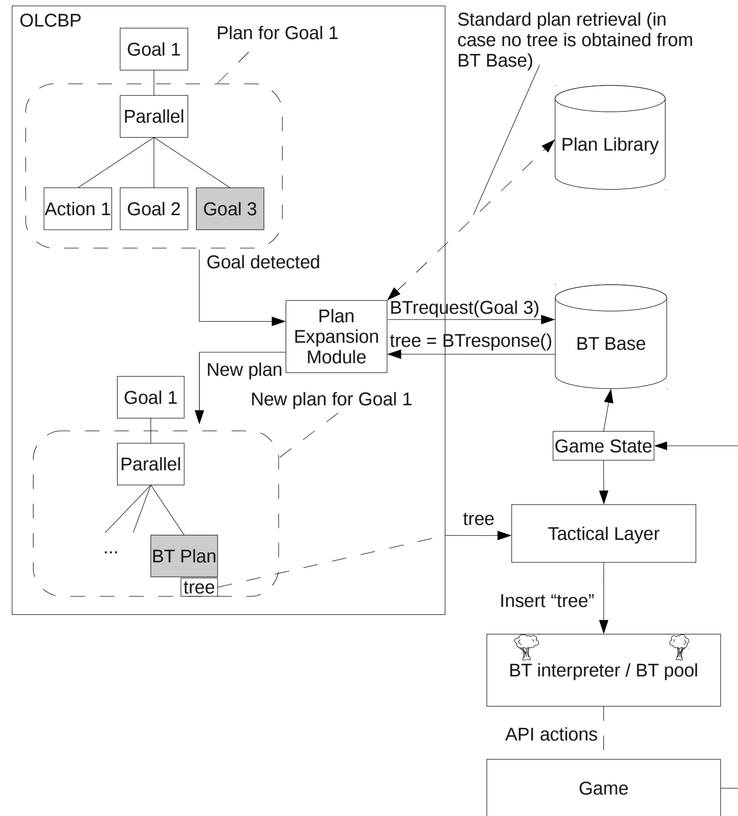

Fig. 2. High level tactical layer

A normal game goes through three different stages: to harvest raw materials, to build and develop your base, and to attack the enemy. During the *harvesting phase*, players focus on building a good number of gathering units or *workers* and, that way, to ensure a good income rate of raw resources. In the second phase, players use those resources to *build their bases*, that is, to create different types of buildings that will allow them to train stronger units and to research new technologies to improve their army. Finally, during the *fighting phase*, players try to destroy the enemy bases using their forces. Of course, these three stages are not really independent and players need to pay attention to the game development in order to decide the best course of action.

We focus on the battle aspects in the Starcraft game. It is consider one of the most challenging features in the game because it requires being to combine different units and skills in a effective way and, at the same time, to react quickly when the enemy changes his strategy.

In this section we describe three simple experiments that will let us evaluate: (1) the viability of the hybrid architecture presented in this paper; (2) the sig-

Fig. 3. Starcraft game (Blizzard Entertainment)

nificant improvement in terms of AI quality we obtain when we allow experts to complete the case base with simple tactical knowledge; and (3), how easily we can represent that type of knowledge using behavior trees. The experiments take place in three battle scenarios proposed in a recent Starcraft AI competition [1], and each one of them requires a different strategy in order to win.

5.1 Experiment 1: Marines Battle

This scenario involves a square battleground without obstacles in which two teams of 12 *terran marines* fight against each other until one of them is exterminated. The terran marine is a basic infantry unit with a medium range attack. Each team starts in a corner of the battlefield so the main tactical decisions in this scenario concern the movement of troops and which enemies should be the first targets.

In this scenario we find an example of the poor reactivity at the plan level problem. When Darmok decides to send one of its units A to kill an enemy unit B, and another enemy unit C intercept A on the way, it is not uncommon to see how A dies under the enemy fire without even reacting to the attack.

In order to improve this behavior we used the two BTs in figures 5 and 6. The first one uses a priority list to detect when a travelling unit is in danger and must defend itself. The second BT is used to attack the threat, giving priority to the initial target over other possible enemies. Finally, the *search BT* node is used to attach the second BT as a branch of the first one at run time. This on-line mechanism give us great flexibility because the search can use knowledge about the current state of the game in order to select the best BT to attack the enemy (there could be different BT to describe different attack strategies).

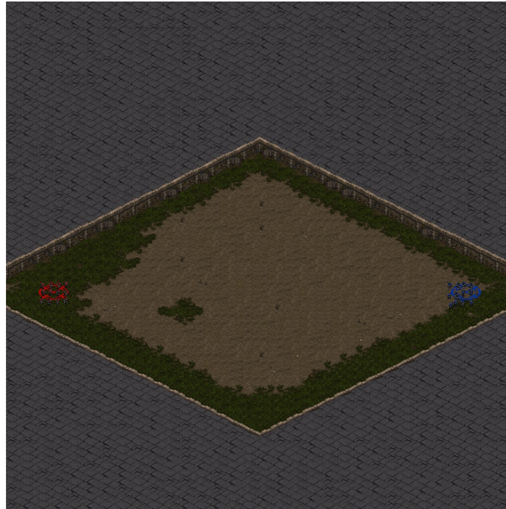

Fig. 4. Starcraft game

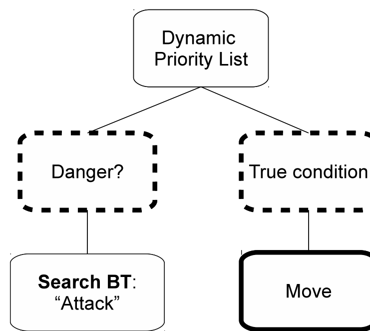

Fig. 5. BT to *move* troops

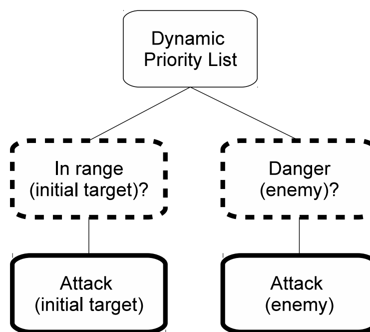

Fig. 6. BT to *attack* enemies

In this experiment Darmork was trained using the traces of 3 real games played in this scenario. Then, we made the Darmok system to play 1000 battles against the game AI, using Darmork with and without the BT layer. In this experiment, these two simple BTs produced an improvement of 61.94% in terms of battles won.

	Darmok	Darmok with BTs
victories	26.2% (262)	42.3% (432)
defeats	73.8% (738)	57.7% (577)
improvement	-	**61.94%**

5.2 Experiment 2: Bunker Defense

The goal of the second scenario is to defend a base as long as possible, while the enemy AI sends waves of enemies every now and then. In order to do it, the player counts with 18 terran marines and some *bunkers* strategically located in the only entrance to the base. A bunker is a defensive structure that can accommodate up to four terran infantry units. Units inside the bunker benefit from a longer range attack and suffer no damage until the bunker is destroyed and the units are expelled unharmed.

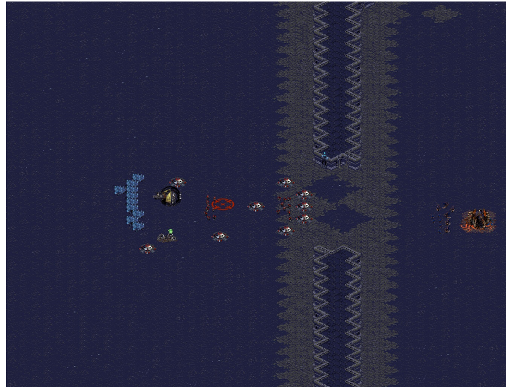

Fig. 7. Bunker Defense scenario

The best strategy in this scenario is to use the bunkers in the first line and to retreat to another bunker when the current one is destroyed. Darmok, however, does not always protect the troops using the bunkers and, sometimes, the system sends the marines to attack the enemy in the open field. In order to improve this behavior, we designed an extended version of the previous *attack BT* (figure 8) that takes into account the presence of near bunkers.

In this experiment Darmok was also trained using the traces of 3 real games. Then, we used Darmok to defend the base 200 times and we counted the number

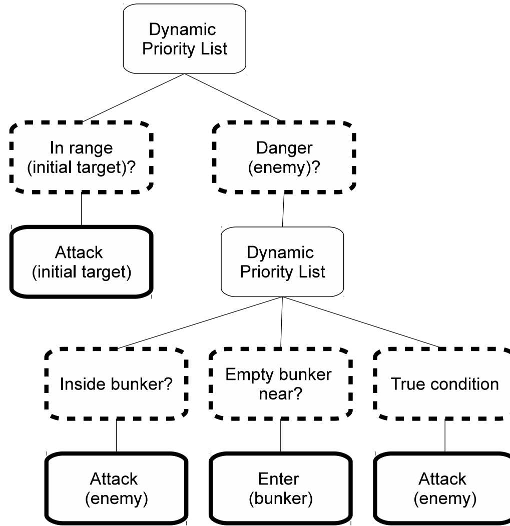

Fig. 8. BT to *attack* enemies using available *bunkers*

of waves that was able to resist before the base was destroyed. In order to make the simulation more real, the defender only sees the part of the map close to the base and, therefore, it does not know when the next wave will arrive.

In this experiment the use of a simple BT produced an improvement of 24.38% in terms of resisted waves.

	Darmok	Darmok with BTs
resisted waves	402	500
μ waves / simulation	2.01	2.5
σ waves / simulation	0.5	0.7
improvement	-	**24.38%**

5.3 Experiment 3: Vultures vs. Firebats

One essential skill to be a good Starcraft player is to know how to use the special powers of some units to produce several casualties in the enemy army. One of these special units is the *terran vulture*, an advanced motorbike and the fastest unit in the game. Vultures produce a very small amount of damage when they attack directly the enemy, but they have a very special weapon, the *spider mines*. Vultures can hide these mines in the ground, were the mines will stay unnoticed until some enemy passes near, producing a big explosion that will kill all the surrounding units (allays and enemies).

In this last experiment we brought face to face 6 terran vultures and 24 *firebats*. Firebats are close range terran units able to make a big amount of damage to all the enemies that stay in front of them and close enough. A good strategy for the player controlling the vultures is to scatter the mines on the

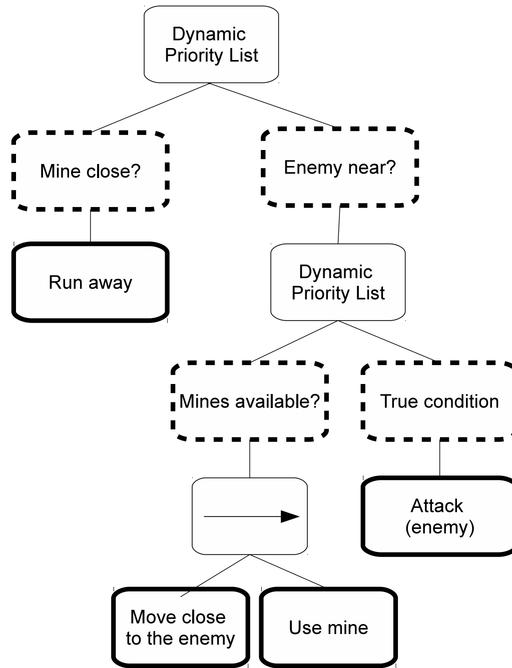

Fig. 9. BT to *attack* enemies using *spider-mines*

ground in front of the fire bats, and then to retreat to a safe position taking advantage of their speed.

The Darmok planner was trained using the traces of 3 games also in this case. It is interesting to realize how difficult is for Darmok to learn automatically how to use the spider mines from some game examples and, however, how easily this type of knowledge can be represented using a BT like the one in figure 9. This BT uses a priority list to check some conditions in a certain order and respond appropriately. First, we check if there is a mine close to the vulture so we can move the unit away to avoid the imminent explosion. Then, we check if the vulture has some mines left and in that case we move it closer to the enemy to use them. Finally, if none of the above conditions are met, we use the vulture to attack the enemy using its default (and weak) attack.

In this experiment we simulated 500 battles in which Darmok controlled the vultures, first without BTs and then using them. We should explain that it is very difficult for 6 vultures to defeat 24 firebats even using the mines properly, so we decided to measure the success of the experiment in terms of the number of firebats killed. In this experiment, just one BT produced an improvement of 34.10%.

	Darmok	Darmok with BTs
firebats killed	4927	6607
μ firebats killed / simulation	9.85	13.21
σ firebats killed / simulation	5.42	5.83
improvement	-	**34.10%**

6 Related Work and Conclusions

There is no, to the best of our knowledge, other work combining Case-based planning and BTs. Nevertheless, this approach can be considered as an example of a more general AI trend of combining domain theory and empirical data: BTs encode a partial view of the expert's domain theory, and cases in the plan library are empirical data. From this point of view, multiple examples of integrations of a domain theory, usually in the form of a set of rules, and Case-based reasoning (CBR) can be found in the research literature [13]. Some systems take the output of a rule-based component as the input for a case-based one, such as the one described in [6], a bank audit system that automatically detects abnormal, irregular, risky, and violated transactions from the standards at the first screening stage, and then applies CBR, which scrutinizes the detected transactions and provides the punishment levels at the second stage. Other systems take an approach closer to the one presented here, where the output of a case-based module feeds a rule-based one, such as the system described in [7], a medical system for Alzheimer's Disease patients, where the case-based module is invoked to determine whether a neuroleptic drug should be prescribed to a patient and if this is so, the rule-based is invoked to select one of five drugs.

Considering BTs as a kind of planning artifact that stores hand-written plans, we can also find related work on the combination of case-based planning and other planning approaches. The SiN system [9] uses a case-based planning algorithm that combines conversational case retrieval with generative planning. SiN can generate plans given an incomplete domain theory by using cases to extend that domain theory, which is given in the form of a planning domain. SiN can also reason with imperfect world-state information by incorporating preferences into the cases. While the case-based module and the domain theory are independently developed in SiN, we propose a more efficient approach by purposely developing a domain theory to fill the holes in the empirical data.

Also in the Starcraft domain, [14] proposes some preliminary ideas to combine expert knowledge in the form of hand-written ABL plans with knowledge automatically extracted from traces using statistical techniques. Although this work is very related to ours, it is at the same time in a very early stage. Besides, using BTs instead of a planning language we facilitate the process of injecting expert knowledge to the system because the experts, in this case game designers, are used to them.

Regarding future work, we intend to explore possible techniques for facilitating the task of identifying those areas in the plan library that require the expert intervention. At this point, a major drawback of the proposed approach is that

the expert needs to analyse the plan library in order to identify those plans, sub-plans or basic actions that require improvement. We envision a computer-assisted identification process, where by generating traces of the AI controlled by the plan library the system can automatically pinpoint actions and goals that usually fail as places for improvement.

References

1. AIIDE: StarCraft AI competition. As part of the program of the Artificial Intelligence and Interactive Digital Entertainment conference (2010)
2. Blizzard: Starcraft game (1998), http://us.blizzard.com/en-us/games/sc
3. Flórez-Puga, G., Llansó, D., Gómez-Martín, M.A., Gómez-Martín, P.P., Díaz-Agudo, B., González-Calero, P.A.: Artificial Intelligence for Computer Games, chap. Empowering Designers with Libraries of Self-validated Query-enabled Behaviour Trees, pp. 55–82. Springer Verlag (2011)
4. Isla, D.: Halo 3 - building a better battle. In: Game Developers Conference (2008)
5. Krajewski, J.: Creating all humans: A data-driven AI framework for open game worlds. Gamasutra (February 2009)
6. Lee, G.H.: Rule-based and case-based reasoning approach for internal audit of bank. Know.-Based Syst. 21(2), 140–147 (2008)
7. Marling, C.R., Whitehouse, P.: Case-based reasoning in the care of alzheimer's disease patients. In: Aha, D.W., Watson, I. (eds.) 4th International Conference on Case-Based Reasoning, ICCBR 2001, Proceedings. pp. 702–715 (2001)
8. Millington, I., Funge, J.: Artificial Intelligence for Games. Morgan Kaufmann, second edn. (2009)
9. Muñoz-Avila, H., Aha, D.W., Nau, D.S., Weber, R., Breslow, L., Yamal, F.: Sin: integrating case-based reasoning with task decomposition. In: IJCAI'01: Proceedings of the 17th international joint conference on Artificial intelligence. pp. 999–1004 (2001)
10. Ontañón, S., Bonnette, K., Mahindrakar, P., Gómez-Martín, M.A., Long, K., Radhakrishnan, J., Shah, R., Ram, A.: Learning from human demonstrations for real-time case-based planning. In: Kuter, U., Muñoz-Avila, H. (eds.) Proceedings of the IJCAI-09 Workshop on Learning Structural Knowledge From Observations (2009), `http://www.cs.umd.edu/~ukuter/struck09/index.html`
11. Ontañón, S., Mishra, K., Sugandh, N., Ram, A.: On-line case-based planning. Computational Intelligence 26(1), 84–119 (2010), `http://dx.doi.org/10.1111/j.1467-8640.2009.00344.x`
12. Palma, R., González-Calero, P.A., Gómez-Martín, M.A., Gómez-Martín, P.P.: Extending case-based planning with behavior trees. In: 24th Florida Artificial Intelligence Research Society Conference. p. To appear (2011)
13. Prentzas, J., Hatzilygeroudis, I.: Categorizing approaches combining rule-based and case-based reasoning. Expert Systems 24(2), 97–122 (2007)
14. Weber, B.: Integrating expert knowledge and experience. In: AAAI Doctoral Consortium (2010)