It Knows What You're Going To Do: Adding Anticipation to a Quakebot

John E. Laird

University of Michigan
1101 Beal Ave.
Ann Arbor, Michigan 48109-2110
laird@umich.edu

Abstract

The complexity of AI characters in computer games is continually improving; however they still fall short of human players. In this paper we describe an AI bot for the game Quake II that tries to incorporate some of those missing capabilities. This bot is distinguished by its ability to build its own map as it explores a level, use a wide variety of tactics based on its internal map, and in some cases, anticipate its opponent's actions. The bot was developed in the Soar architecture and uses dynamical hierarchical task decomposition to organize it knowledge and actions. It also uses internal prediction based on its own tactics to anticipate its opponent's actions. This paper describes the implementation, its strengths and weaknesses, and discusses future research.

AI bots in first-person shooter (FPS) computer games have continually gotten more complex and more intelligent. The original bots in FPSs were completely oblivious to their environment and used fixed scripts to attack the human player. Current bots, such as those found in Quake III or Unreal Tournament, are beginning to approximate the game play of humans. They collect health and other powerups, and they have a variety of tactics such as circle-strafing and popping in and out of doorways. What, if anything, are they missing? Although they can react to different situations and opponents, as of yet they do not anticipate or adapt to the behavior of other players. The following quote from Dennis (Thresh) Fong, the Michael Jordon of Quake, gives some insight into the importance of anticipation (Newsweek, November 1999):

> Say my opponent walks into a room. I'm visualizing him walking in, picking up the weapon. On his way out, I'm waiting at the doorway and I fire a rocket two seconds before he even rounds the corner. A lot of people rely strictly on aim, but everybody has their bad aim days. So even if I'm having a bad day, I can still pull out a win. That's why I've never lost a tournament.

A related example is when you see an enemy running down a hallway far away. Because the enemy has only the blaster (an inferior weapon), you realize he is probably looking for the hyperblaster (a much better weapon), which is just around the corner from you. You decide to go get the hyperblaster first and directly confront the enemy, expecting that your better firepower will win the day.

Each of these tactics can be added manually for specific locations in a specific level of a game. We could add tests that if the bot is ever in a specific location on a specific level and hears a specific sound (the sound of the enemy picking up a weapon), then it should set an ambush by a specific door. Unfortunately, this approach requires a tremendous effort to create a large number of tactics that work only for the specific level.

Instead of trying to encode behaviors for each of these specific situations, a better idea is to attempt to add a general capability for anticipating an opponent's actions. From an AI perspective, anticipation is a form of planning; a topic researchers in AI have studied for 40 years. The power of chess and checkers programs comes directly from their ability to anticipate their opponent's responses to their own moves. Anticipation for bots in first-person shooters (FPS) has a few twists that differentiate it from the standard AI techniques such as alpha-beta search.

1. A player in a FPS does not have access to the complete game state as does a player in chess or checkers.
2. The choices for action of a player in a FPS unfold continuously as time passes. At any time, the player can move, turn, shoot, jump, or just stay in one place. There is a breadth of possible actions that make search intractable and requires more knowledge about which actions might be useful.

This paper describes the Soar Quakebot and how anticipation was added to it. The original Soar Quakebot (Laird & van Lent, 1999) was designed to be a human-like expert at playing Quake deathmatches. It did not incorporate any planning, and was designed to be a reactive system that integrated large bodies of tactics via hierarchical goals based on the techniques we used to successfully model the behavior of military pilots (Jones et al. 1999; Tambe et al. 1995). However, as we developed the Quakebot, we found that in order to improve the behavior of the bot, we were forced to add more and more specialized tactics. In addition, when we presented our work to game developers, they invariably asked, "Does it anticipate the human players actions? If it did, that would be really cool." Given that the underlying goal of all of our research is to be "really cool" (which may be hard to believe given that we are nerdy AI researchers), we finally

got around to looking at adding anticipation, which is the subject of this paper.

The remainder of the paper is as follows. First, we present the design of the Soar Quakebot sans anticipation. Next we describe how anticipation was added to the Quakebot and present examples of its behavior. To our surprise, it was straightforward to add anticipation to the Soar Quakebot, and it also provided a general approach to encoding many of the tactics that originally required specialized knowledge. Finally, we describe future work to extend the work on anticipation with the main emphasis on learning opponent-specific models of behavior.

The Soar Quakebot

The Soar Quakebot plays the death match version of Quake II. In a death match, players exist in a "level", which contains hallways and rooms. The players can move through the level, picking up objects, called powerups, and firing weapons. The object of the game is to be the first to kill the other players a specified number of times. Each time a player is shot or is near an explosion, its health decreases. When a player's health reaches zero, the player dies. A dead player is then "spawned" at one of a set of spawning sites within the level. Powerups, which include weapons, health, armor, and ammo, are distributed throughout the level in static locations. When a powerup is picked up, a replacement will automatically regenerate in 30 seconds. Weapons vary according to their range, accuracy, spread of damage, time to reload, type of ammo used, and amount of damage they do. For example, the shotgun does damage in a wide area if used close to an enemy, but does no damage if used from a distance. In contrast, the railgun kills in a single shot at any distance, but requires very precise aim because it has no spread.

The Soar Quakebot controls a single player in the game. We have attempted to make the perceptual information and motor commands available to the bot similar to those that a human has playing the game. For example, a bot can see only unobstructed objects in their view cone and they can hear only nearby sounds. One issue is that bots cannot sense the walls in a level as coherent objects because they consist of many polygons that are displayed to the user to give the appearance of solid walls, open doorways, etc. To navigate a level, the Quakebot explores the level and deliberately builds up a map based on range data to walls. The Quakebot uses this internally generated map to know where walls, rooms, hallways, and doors are when it is running through a level. To make the internal map construction code easier to implement, the Soar Quakebot can map only two-dimensional levels that are made up of rectangular rooms connected by rectangular hallways. Once a map is built, it can be saved for later use when the Soar Quakebot replays the same level.

As shown in Figure 1, the Soar Quakebot reasoning code currently runs on a separate computer and interacts with the game using the Quake II interface DLL (dynamically loaded library). C code, which implements the Soar Quakebot's sensors and motor actions, is embedded in the DLL along with our inter-computer communication code, called Socket I/O. Socket I/O provides a platform independent mechanism for communicating all perception and action information between the Quakebot and the game and has also been used to interface Soar to Descent 3. The Quakebot uses Soar (Laird et al., 1987) as its underlying AI engine. All the knowledge for playing the game, including constructing and using the internal map, is encoded in Soar rules. The underlying Quake II game engine updates the world and calls the DLL ten times a second (the graphics engine updates much more often than the game engine). On each of these cycles, all changes to the bots sensors are updated and any requested motor actions are initiated.

In this configuration, Soar runs asynchronously to Quake II and executes its basic decision cycle anywhere from 30-50 times a second, allowing it to take multiple reasoning steps for each change in its sensors. Soar runs as fast as possible, consuming 5-10% of the processing of a 400MHz Pentium II running Windows NT.

Soar is an engine for making and executing decisions - selecting the next thing the system should do and then doing it. In Soar, the basic objects of decision are call *operators*. An operator can consists of primitive actions to be performed in the world (such as move, turn, or shoot), internal actions (remember the last position of the enemy), or more abstract goals to be achieved (such as attack, get-item, go-through-door) that in turn must be dynamically decomposed into simpler operators that ultimately bottom out in operators with primitive actions. These primitive actions are implemented by if-then rules.

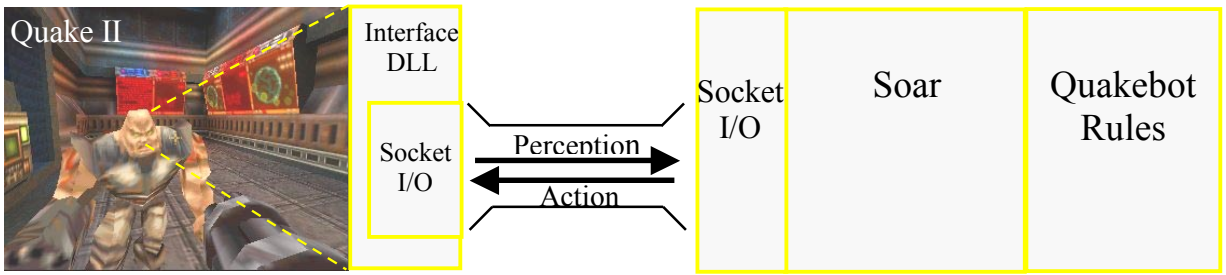

Figure 1: Interface between Quake II and the Soar Quakebot

The basic operation of Soar is to continually propose, select, and apply operators to the current state via rules that match against the current state. When an abstract operator is selected that cannot be applied immediately, such as get-item, then a substate is generated. For this substate, additional operators are then proposed selected and applied until the original operator is completed, or the world changes in such a way as to lead to the selection of another operator. Figure 2 shows a typical trace of operators being selected. Indentation indicates that a substate has been created and that operators are then selected to pursue the operator that led to the substate.

```
10:    O: O11 (collect-powerups)
11:    ==>S: S14
12:       O: O12 (get-item)
13:       ==>S: S16
14:          O: O19 (goto-item)
15:          ==>S: S17
16:             O: O23 (face-item)
17:             ==>S: S20
18:                ==>S: S21
22:                   O: O25 (wait)
23:                   O: O24 (wait)
24:             O: O28 (move-to-item)
25:             ==>S: S23
26:                ==>S: S24
27:                   O: O29 (wait)
28:                   O: O30 (wait)
29:                   O: O31 (wait)
```

Figure 2: Trace of operator selections

The trace starts with the selection of the collect-powerups operator (O11). This operator immediately becomes a goal, as it is not possible to apply it immediately. In the resulting substate (S14), many rules can fire (not shown) to propose getting specific items that the bot needs. Additional rules fire (also not shown) that create preferences for the operators based on the worth of the item, its distance, etc. At decision 12, one instance is selected, which in this case is to get the supershotgun in the current room. Get-item is further decomposed into suboperators go-through-door, when the item is not in the current room, and goto-item, when the item is in the current room. The supershotgun is in the room, so goto-item is selected, which is then implemented in a substate by face-item and move-to-item. The proposal for face-item tests that if the bot is not facing the item being picked up, then the bot should turn toward it. Facing an item is not instantaneous, and decisions 17-23 show how the bot just waits until the turning is complete. Once the bot is facing the item, the proposal for move-to-item fires, and move-to-item is selected, which also takes time to complete.

Figure 3 shows the underlying organization of operators that gave rise to the trace in Figure 2. This is just a small part of the overall hierarchy, but includes some of the top-level-operators, such as wander, explore, attack, and those that are used in the substate that can arise to apply the collect-powerups operator.

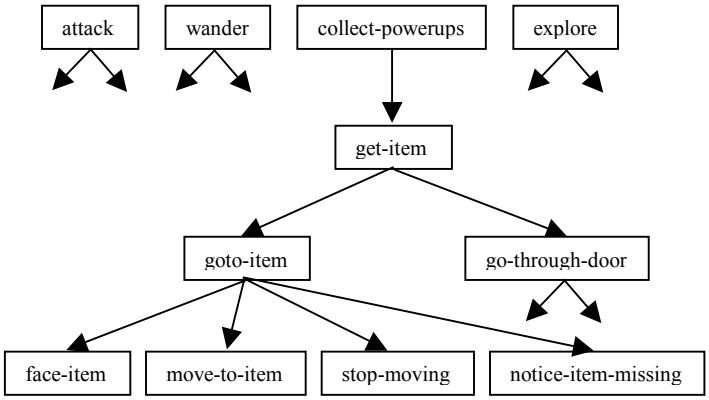

Figure 3: Partial operator hierarchy

Soar does not use any pre-defined ordering to determine which operators to select and apply. As mentioned earlier, the knowledge in Soar to propose, select, and apply operators is encoded as if-then rules. The rules are Soar's long-term procedural knowledge, and they are matched against the states stored in Soar's global declarative *working memory*. Working memory holds all of the bot's information about the current situation, including perception, elaborations of perception, data structures representing the map of the game, etc. All rules that successfully match working memory *fire* in parallel to change working memory by either adding or deleting declarative structures. There is no underlying program counter that inexorably moves execution from one statement to the next, independent of changes to the situation, possibly performing an action that has become obsolete. Instead, each action is selected by rules that continually test the current situation.

Soar's underlying processing cycle that selects and applies operators consists of five phases as shown in Figure 4. The following paragraphs describe the processing cycle to a level of detail that can be skipped for those only interested in anticipation.

1. Sensing: Updating the available perceptual information in the top state. The creation and updating of perceptual information is done automatically by C routines in the Quake DLL that simulate the sensing of a human. For example, the Quakebot can "see" unobstructed objects in a forward facing view cone out to a pre-set range. The Quakebot can "hear" movement, explosions, and other sounds to a pre-set range. The Quakebot can also sense its own state, such as the items it has, its health, its position, orientation, speed, etc.
2. Elaboration, operator proposal, and operator evaluation:

a) Elaboration: Based on the contents of working memory, rules may fire to monotonically elaborate the sensory information with task-specific data. For example, rules might test current health level and then create a new structure in working memory that classifies it as critical, low, medium, or high. Additional rules can test for the presence of these structures.
b) Operator Proposal: Based on the contents of working memory, rules may fire to propose operators to be selected for the current states (the origin of states besides the top state is described below). The action of these rules is to create special working memory elements that signal Soar that the operator should be considered for selection for a specific state. You can think of these rules as testing the pre-conditions of the operators and proposing the operators when it is legal to apply them to the current situation.
c) Operator Evaluation: Additional rules can test which operators have been proposed, and then create *preferences* for them. There is a fixed set of preferences that can specify partial orders among operator, that some operators should not be selected, that it doesn't matter which operator is selected, and so on. The rules that create the preferences can test other aspects of the state for which the operator is proposed, making it easy to encode heuristic selection knowledge.

All of these three types of rules fire in parallel - there is no separate phase for elaboration or proposal or evaluation - the ordering of rule firing is completely data driven. Because of data dependencies, elaboration rule firings will usually lead to proposal and then evaluation rule firings. In addition, these rules retract their actions when their conditions no longer match working memory so that only elaborations, proposals, and evaluations relevant to the current situation are maintained in working memory. Soar stays in this phase until quiescence is reached and no more rules fire or retract. This usually happens in two to three waves of parallel rule firing.

3. Operator Selection: Based on the created preferences, a fixed decision procedure picks the best operator for each state. Once an operator is selected, it is installed as the current operator for the current state in working memory. In many cases, the result of the decision procedure will be to maintain the current operator, especially when it takes time for an operator to apply. If the preferences are inconsistent (one operator is better than another and the second is also better than the first), incomplete (the preferences do not distinguish between the available operators or there are no operators proposed), or do not lead to the selection of a new operator, then an *impasse* is reached, signifying that more knowledge is required. Whatever the cause of the impasse, Soar automatically creates a new *substate* in which the goal of the problem solving is to resolve the impasse. As problem solving progresses, an impasse may arise in the substate, leading to a stack of states. Soar continually fires rules and attempts to select operators for every state in the stack during each loop through the decision cycle. When a different operator selection can be made for an impassed state (through the creation of results in the substate or because through changes in perception that in turn lead to changes in which operators are proposed), then the impasse is resolved, the substate (and all of its substates) is removed from working memory and problem solving continues.
4. Operator Application: Once an operator is selected, rules that match against it can fire, changing working memory, possibly including commands to the motor system. Because the actions are implemented as rules, Soar directly supports conditional operators as well as operators whose actions unfold over time based on feedback from the perceptual system. Rules that apply operators do not retract their actions when they no longer match, but create persistent data structures in working memory that are removed only when they are explicitly removed by another operator application rule or become disconnected from the state because of some other change. The operator application phase continues until no additional rules fire.
5. Output: Following application, all newly created output commands, such as turn, move, shoot, are sent to the motor system.

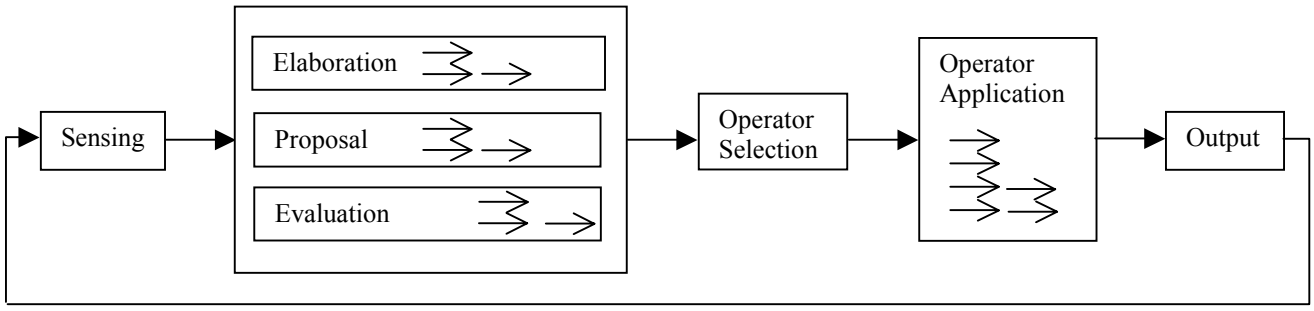

Figure 4: The Soar Decision Cycle

The Soar Quakebot is designed based on the principles developed early on for controlling robots using Soar (Laird and Rosenbloom 1990) and then extended in our research on simulating military pilots in large scale distributed simulations (Jones, et al. 1999). For more details on the structure of the Soar Quakebot than provided below, see the Soar Tutorial (Laird 2000).

Below is a list of the main tactics the Quakebot uses. These are implemented across the top-level operators. Excluding the anticipation capability, the current Soar Quakebot has 100 operators, of which 20 have substates, and 715 rules.

- Collect-powerups
  - Pick up items based on their spawn locations
  - Pick up weapons based on their quality
  - Abandon collecting items that are missing
  - Remember when missing items will respawn
  - Use shortest paths to get objects
  - Get health and armor if low on them
  - Pickup up other good weapons/ammo if close by
- Attack
  - Use circle-strafe
  - Move to best distance for current weapon
- Retreat
  - Run away if low on health or outmatched by the enemy's weapon
- Chase
  - Go after enemy based on sound of running
  - Go where enemy was last seen
- Ambush
  - Wait in a corner of a room that can't be seen by enemy coming into the room
- Hunt
  - Go to nearest spawn room after killing enemy
  - Go to rooms enemy is often seen in

Finally, the Soar Quakebot is has many numeric parameters that determine the details of behavior, such as how long does it hide for an ambush, how far should it try to get for a certain weapon. We have grouped some of these parameters together to create different styles of Quakebots that vary in the tactics in terms of aggressiveness, reaction time, aiming skill, and overall intelligence (where certain tactics are disabled or enabled).

Anticipation

Our approach to anticipation is to have the Quakebot create an internal representation that mimics what it thinks the enemy's internal state is, based on its own observation of the enemy. It then predicts the enemy's behavior by using its own knowledge of tactics to select what it would do if it were the enemy. Using simple rules to internally simulate external actions in the environment, the bot forward projects until it gets a prediction that is useful, or there is too much uncertainty as to what the enemy would do next. The prediction is used to set an ambush or deny the enemy an important weapon or health item.

In adding a general capability like anticipation to the Soar Quakebot, one of the goals is that it really is general. There are many different ways it should be general. It should be independent of the level the bot is in. Moreover it should be as independent as possible of the specific tactics the bot already has. That does not mean it can't make use of them when doing anticipation, but it does mean that we should be able to add anticipation with as few as changes to the existing bot as possible. This makes the addition easier, and also gives us some confidence that the anticipation capability can be used for other bots that play other games. Therefore, as part of the discussion of anticipation, we will report on the number and character of the rules needed to be added, modified, or deleted, and whether these rules were task-dependent (test or modify working memory elements specific to Quake) or domain-independent.

Anticipation requires adding knowledge about when it should be used, how it is done, and how the results are used to change behavior. These map on to the following:

1. Proposal and selection knowledge for a predict-enemy operator.
2. Application knowledge for applying the predict-enemy operator.
3. Proposal knowledge for selecting operators that will use the predictions to set ambushes.

Proposal and Selection

When should the Soar Quakebot attempt to predict the enemy's behavior? It should not be doing it continually, because of the computational overhead and the interference with other activities. It shouldn't do it when it has absolutely no idea what the state of the other bot is and it also shouldn't do it when any prediction will be ignored because the bot already knows what to do. The Soar Quakebot attempts to anticipate an enemy when it senses the enemy (so it knows some things about the enemy's state), and the enemy is not facing the bot and is far away (otherwise the bot should be attacking). When the predict-enemy operator is proposed, a rule fires that prefers it to all other top-state tactical operators (such as wander, collect-powerups, and retreat).

Figure 5 shows an example where the Quakebot (lower left) sees its enemy (upper center) heading north, on its way to get a desirable object (the heart). This corresponds to the situation described above and causes the Quakebot to propose and select the predict-enemy operator.

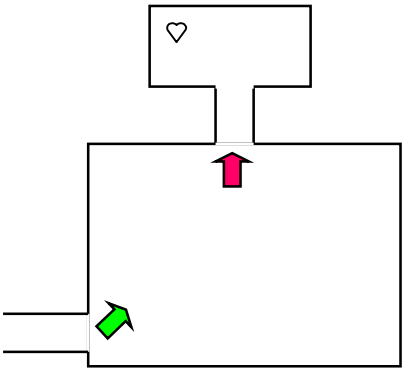

Figure 5: Initial situation in which the predict-enemy operator is selected.

One important aspect of Soar is that if the enemy turns toward the Quakebot instead of continuing north, the predict-enemy operator will be retracted so that the Quakebot can select the attack operator and not be caught napping.

The proposal and selection rules are not specific to a given level, nor are they even specific to the game of Quake. However, they do restrict anticipation so that it is only used in limited cases. Later we will discuss the possibility of an even more general approach that extends the use of anticipation. Overall, there are 4 rules used to propose and select the predict-enemy operator.

Application

Once the decision has been made to predict the enemy's behavior (via the selection of the predict-enemy operator), the next stage is to do it. Our approach is straightforward. The Quakebot creates an internal representation of the enemy's state based on its perception of the enemy and then uses its own knowledge of what it would do in the enemy's state to predict the enemy's actions. Thus, we will assume that the enemy's goals and tactics are essentially the same as the Quakebot's. This is the same approach that is taken in AI programs that play most games, such as chess or checkers. However, in this case the actions that are taken are not moving a piece on a board but are the movement of a Quakebot through its world using perception and motor commands.

The first step is to create the internal representation of the enemy's situation so that the Quakebot's tactics can apply to them. This is easy to do in Soar because Soar already organizes all of its information about the current situation in its state structure in working memory. All that needs to be done is that when the predict-enemy operator is selected and a substate is created, that state needs to be transformed into a state that looks like the top-level state of the enemy. This is done using an operator (create-enemy-state) that creates structures on the substate that corresponds to what the Quakebot thinks the enemy is sensing and has in its working memory, such as the map. The internal representation of the enemy's state is only approximate because the Quakebot can sense only some of it and must hypothesize what the enemy would be sensing. Surprisingly, just knowing the enemy's position, health, armor level, and current weapon are sufficient to make a plausible prediction of high-level behavior of players such as the Soar Quakebot. Sixteen rules are involved in creating the enemy state, nine that are specific to the Quakebot data structures and seven that are general. Three existing rules had to be modified to inhibit them from firing during the initialization of the prediction, so that the internal representation for the enemy's state did not include sensory information from the Quakebot itself.

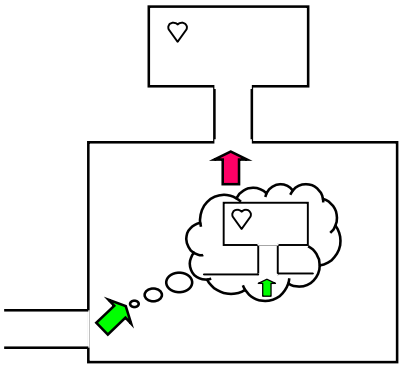

Figure 6: The Quakebot creates internal representation of enemy's situation.

The second step involves letting the Quakebot's tactics work on its representation of the enemy's state. In the internal simulation of the example in the figures, rules would propose the collect-powerups operator in order to get the heart powerup. The Quakebot knows that the powerup is in the room to the north from prior explorations and attributes that knowledge to the enemy. Once collect-powerups is selected, a substate will be created, and then get-item, which in turn will have a substate, followed by go-through-door. If this was no an internal simulation, go-through-door would lead to a substate in which goto-door is selected. However, for tactical purposes, the Quakebot does not need to simulate to that level of detail. To avoid further operator decompositions, a rule is added that tests that a prediction is being done and that the go-through-door operator is selected. Its actions are to directly change the internal representation so that the Quakebot (thinking it is the enemy), thinks it has moved into the hall. Similar rules are added to short-circuit other operator decompositions. Additional rules are needed to update related data structures that would be changed via new perceptions (frame axioms), such as that health would go up if a health

item was picked up. One additional rule is added to keep track of how far the enemy would travel during these actions. This information is used later to decide when to terminate the prediction. Altogether, nine rules are added to simulate the effects of abstract operators. All of these rules are specific to the operators used in Quake, but independent of the details of the specific level. However, if we ever wanted to add the ability of the Quakebot to plan its own behavior, these rules would be necessary. Figure 7 shows the updated internal representation of the Quakebot.

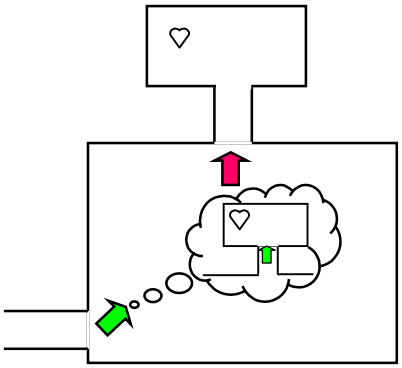

Figure 7: The Quakebot projects that enemy will move into hallway in pursuit of powerup.

The selection and application of operators continues until the Quakebot thinks that the enemy would have picked up the powerup. At that point, the enemy is predicted to change top-level operators and choose wander. Because there is only one exit, wander would have the enemy leave the room, going back into the hallway and finally back into the room where the enemy started (and where the Quakebot is).

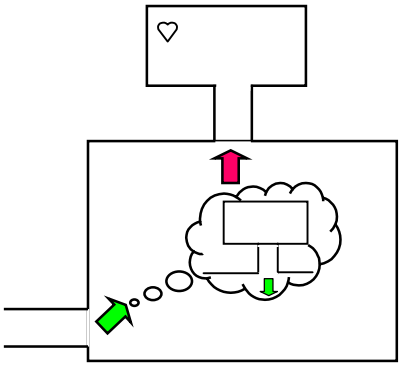

Figure 8: The Quakebot projects that enemy will return to the current room.

Predicting

Throughout this process, the Quakebot is predicting the behavior of the enemy. That prediction is only useful if the Quakebot can get into a tactical position that takes advantage of the prediction. Up until the enemy returns to the room, the prediction does not help the Quakebot. However, if the Quakebot hides by the hallway, it can get off a shot into the back or side of the enemy as it comes into the room. Thus, following the prediction, the Quakebot can set an ambush.

What are the general conditions for using the prediction: that is, what advantage might you get from knowing what the enemy is going to do? For Quake II, we've concentrated on the case where the bot can predict that it can get to a room before the enemy, and either set an ambush or deny the enemy some important powerup. This is done by continually comparing the distance that the enemy would take to get to its predicted location to the distance it would take for the Quakebot to get to the same location. For the current system, the number of rooms entered is used a rough distance measure. In the example above, the Quakebot predicts that it will take the enemy four moves to get back to the current room, and it knows it is already in that room. Why doesn't the Quakebot stop predicting when the enemy would be coming down the hallway, which is three moves for it vs. one for the bot? The reason is that the Quakebot knows that it cannot set an ambush in a hallway, and thus waits until the predicted location is a room.

A prediction can also terminate when the Quakebot (thinking as the enemy) comes across a situation in which there are multiple possible actions for which it does not have a strong preference. This would have arisen in the previous example if there had be three doors in the north most room - with only two doors, the prediction would have gone forward because of the preference to avoid going back where you came from. When this type of uncertainty arises, the Quakebot abandons the prediction. A total of five rules are used to detect that a relevant prediction has been created. These are specific to the approaches of using distance or uncertainty to decide when to terminate the prediction.

One possible extension to our approach is to have the bot maintain explicit estimates or probabilities of the different alternatives and search forward, predicting all possible outcomes and their probabilities. There are two reasons this is not done. First, the Quakebot does not need the probability estimates in order to make its own decision. Second, the added time to do such an extensive prediction could make the prediction meaningless as the enemy will have already moved through the environment by the time the prediction completes.

Using the Prediction

In the Soar Quakebot, three operators make use of the predictions created by predict-enemy: hunt, ambush, and deny-powerups. When a prediction is created that the enemy will be in another room that the Quakebot can get to sooner, hunt is proposed and it sends the bot to the correct room. Once in the same room that the enemy is predicted to be in, ambush takes over and moves the bot to an open location next to the door that the enemy is predicted to come through. In general, the bot will try to shoot the

enemy in the back or side as it enters the room (shown below in the figure). But if the bot has the rocket launcher, it will take a pre-emptive shot when it hears the enemy getting close (a la Dennis Fong, who was quoted earlier). Both of these ambush strategies have time limits associated with them so that the bot waits only a bit more time than it thinks the enemy will take to get to the room in which the bot has set the ambush. Deny-powerups is selected when the enemy is predicted to attempt to pick up a powerup that the bot can get first.

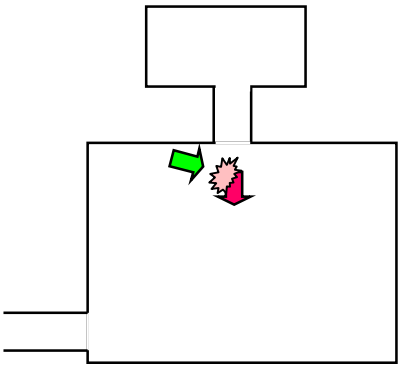

Figure 9: The Quakebot executes an ambush based on the results of its prediction.

Learning predictions

Inherent to Soar is learning mechanism, called chunking, that automatically creates rules summarizes the processing within impasses as rules. Chunking creates rules that test the aspects of the situation that were relevant during the generation of a result. The action of the chunk creates the result. Chunking can speed up problem solving by compiling complex reasoning into a single rule that bypasses the problem solving in the future. Chunking is not used with the standard Quakebot because there is little internal reasoning to compile out; however, with anticipation, there can be a long chain of internal reasoning that takes significant time (a few seconds) for the Quakebot to generate. In that case chunking is perfect for learning rules that eliminate the need for the Quakebot to regenerate the same prediction. The learned rules are specific to the exact rooms, but that is appropriate because the predictions are only valid under special circumstances.

Below is an English version of a rule learned by the Quakebot.

If predict-enemy is the current operator and
there is an enemy with health 100,
using the blaster, in room #11 and
I am distance 2 from room #3
then
predict that the enemy will go to room #3
through door #7.

Compiled into the prediction is that the bot can get to room #3 before the enemy.

Once this rule is learned, the bot no longer needs to go through any internal modeling and will immediately predict the enemy's behavior when it sees the enemy under the tested situations. The impact is that as the bot plays the game, it will build up a set of prediction rules, and it will make fast predictions in more situations. In fact, it might turn out that originally when it does prediction, the time to do the prediction sometimes gets in the way of setting an ambush or denying a powerup, but with experience that time cost will be eliminated. One possibility to create more challenging opponents is to pre-train Quakebots so that they already have an extensive set of prediction rules.

Limitations and Extensions

In this section we discuss various limitations and possible extensions to the anticipation capabilities of the Soar Quakebot.

Generality of Anticipation Capability

Our goal was to create a general anticipation capability that could be used by a given Quakebot for different game levels. In that we've succeeded. None of the knowledge added to support anticipation needs to be customized for a specific level. The power comes from reusing the bot's tactics and knowledge of the structure of a level (which it gains by mapping the level on its own).

A more general goal is for the anticipation capability to be useful by other bots in completely different games. Many parts of the capability are completely independent of Quake. However, some are not. Below is a recap of the different parts of the anticipation capability and the types of game-specific or game-independent knowledge they require.

- Deciding when to predict an enemy's behavior.
  General across games but restrict to a set of situations.
- Predicting the enemy's behavior.
  - Creating the internal representation of the enemy's state.
    Specific to the structure of the perceptual data and important internal state features.
  - Proposing and selecting operators for the enemy.
    General: uses existing bot knowledge.
  - Simulating the execution of operators.
    Specific to the operators, but part of planning knowledge would be available if bot planned some of its own actions.
  - Deciding that a prediction is useful.
    Specific to the situations that the bot expects to be useful: places the bot can get to first.
- Using the prediction to select other tactics/operators.
  Specific to those tactics.

The minor weaknesses are in terms of adding knowledge about the perceptual data and the abstract execution of

operators. They are usually easy to add and in no way restrict the use of anticipation.

The more troubling issue arises from the need for knowledge that determines when the enemy's behavior should be predicted and how will the predictions be useful. This restricts anticipation to being used only under situations that the designer has deemed worthwhile. This is important because anticipation could be used as the generator for many of the tactics that would otherwise be coded by hand. For example, during a fight the bot could predict that an injured enemy would attempt to pick up a nearby health. The bot could use this to either get the health first, or direct its weapon toward the health, making it more likely that the enemy will be hit. Another example is where the bot uses its knowledge about the expected path of the enemy to avoid the enemy when the bot is low on health or has inferior weapons. Similarly, when the bot kills an enemy, it could predict that the enemy will be recreated at a spawn location with only a mediocre weapon. It could use that prediction to move toward the closest spawn point in hope of engaging the enemy before it gets a better weapon. This second tactic is currently hard coded in the Soar Quakebot, but could arise from the appropriate use of anticipation.

The obvious approach would be to always predict the enemy and then always attempt to plan what actions the bot could perform that would get the bot to a preferred state (such as the enemy being dead, or having full health and a good weapon). This has the potential of even further simplifying the current structure of the bot by having the planning mechanism generate plans for getting powerups, attacking the enemy, etc. Many of the current tactics could be discarded. However, this approach comes with significant costs. There is the cost of planning and the fact that the planning would interfere with the bot's ability to react quickly to its environment - although chunking would gradually decrease with practice. Unfortunately, forward planning in these environments in intractable. As mentioned earlier, the bot has a huge space of possible moves it can take at each moment, with the most obvious culprit being which direction it should face. The more subtle cost is the cost of developing the planning knowledge that is used to generate the tactics, which in practice can be very difficult. An alternative is to be more selective, using some knowledge about the goals the bot is trying to achieve and means-ends problem solving to constrain the search.

Recursive Anticipation

The Quakebot anticipates what the enemy does next. An obvious extension is for the Quakebot to anticipate the enemy anticipating its own actions. This recursion can go on to arbitrary depths, but the usefulness of it is probably limited to only a few levels. Recursive anticipation could lead the Quakebot to actions that are deceptive and confusing to the enemy. Although this might be useful in principle and for non-real-time computer games, such as real-time strategy games where there is more global sensing and a less frantic pace, it might be of only limited use for the Quakebot. The reason is that the bot must sense the enemy in order to have some idea of what the enemy's state is, and the enemy must sense the bot in order to have some idea of what the bot's state is. In Quake, there are only rare cases where the bot and the enemy can sense each other and one will not start attacking the other. However, we plan to do some limited investigation of recursive anticipation to find out how useful it is.

Enemy-Specific Anticipation

The current anticipation scheme assumes that the enemy uses exactly the same tactics as the Quakebot. However, there may be cases where you know beforehand that an opponent has different tactics, such as preferring different weapons. By incorporating more accurate models of an enemies weapon preferences, the Quakebot can decide to ambush an enemy in completely different (and more appropriate) rooms.

This is easily handled by adding rules that encode the given tactics or weapon preferences. These rules must test the name of the opponent so that they apply only for the reasoning about the appropriate enemy. Additional rules may be required that reject the bot's tactics that conflict with the enemy's tactics, or it may be necessary to add conditions to the rules that encode those conflicting tactics so that they do not match when the enemy is being simulated.

Adaptive Anticipation

Unfortunately, an enemy's tactics and preference are rarely known beforehand. It is only through battle that one learns about the enemy. Thus, an important area of future research will be building models of enemy tactics that can be used to predict enemys' behavior. Our expected approach will be to learn how the enemy's tactics differ from the Quakebot's own knowledge. We will examine at a variety of learning approaches. The simplest is to pre-enumerate different "styles" or groups of tactics and explicitly gather data on the enemy in order to build up a better internal model. For example, noticing which weapons are used most often should give some idea as to those preferences. This approach is simple to implement, has low computational overhead, but has limitations in that the bot can only learn about the set of styles that have already been enumerated. Although this limitation is of concern to us as researchers, it may be completely sufficient for computer games. Similar approaches have already been successfully used in football games to track and adjust to the play-calling behavior of human players (Whatley, D. 1999).

A more general, but more difficult approach is to have the bot modify its knowledge each time the enemy does something unpredictable. The bot would continually try to build up its knowledge so that it can successfully predict the enemy. One final complexity is that the enemy will not be static, but will be adapting to the bot's tactics, and even

to the bot's use of anticipation and it adaptation to the enemy. For example, after the first time an enemy is ambushed after getting the powerup from a dead-end room, it will probably anticipate the ambush and modify its own behavior. Our research into these issues will build on previous research we've done on learning from experience with dynamic environments (Laird, J. E., Pearson, D. J, and Huffman, S. B. 1997).

Summary and Perspective

The goal of our research is to create synthetic characters for computer games with human-level intelligence. Incorporating anticipation is a critical part of human-level intelligence and we have demonstrated how it can be added to an existing computer bot.

From our perspective, this has been a success because it was added with only minimal changes to our existing bot, it added significantly new capabilities and behavior, and it points the way to many additional research issues.

From an AI perspective, our work is a bit of a rehash of research on opponent modeling, planning, and reactive planning. Its contribution to AI is that it pursues these topics within the context of a complex, dynamic, and competitive environment, where planning and execution efficiency are of utmost importance as well as ease of implementation.

From a computer games perspective, our work points the way for where commercial bots could be in a few years, not just "thinking" on their own, but predicting what you are thinking.

Acknowledgments

The author is indebted to the many students who have worked on the Soar/Games project, most notably Michael van Lent, Steve Houchard, Joe Hartford, and Kurt Steinkraus.

References

Jones, R.M., Laird, J.E., Nielsen, P.E., Coulter, K.J., Kenny, P.G., and Koss, F.V. (1999) Automated Intelligent Pilots for Combat Flight Simulation, AI Magazine, 20(1), 27-42.

Keighley, G. (1999) The Final Hours of Quake III Arena: Behind Closed Doors at id Software, GameSpot, http://www.gamespot.com/features/btg-q3/index.html.

Laird, J. E. (2000) The Soar Tutorial, Part V. The Soar Quakebot, http://ftp.eecs.umich.edu/~soar/tutorial.html.

Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987), Soar: An architecture for general intelligence. Artificial Intelligence, 33(3), 1-64.

Laird, J. E., Pearson, D. J., and Huffman, S. B., (1997) Knowledge-directed Adaptation in Multi-level Agents. Journal of Intelligent Information Systems. 9, 261-275.

Laird, J. E. and Rosenbloom, P. S. (1990) Integrating Execution, Planning, and Learning in Soar for External Environments. In *Proceedings of National Conference of Artificial Intelligence*, Boston, MA, 1022-1029.

Laird, J. E. and van Lent, M. (1999) Developing an Artificial Intelligence Engine. In *Proceedings of the Game Developers' Conference*, San Jose, CA, 577-588.

Tambe, M., Johnson, W. L., Jones, R. M., Koss, F., Laird, J. E., Rosenbloom, P. S., and Schwamb, K. (1995), Intelligent Agents for Interactive Simulation Environments, AI Magazine, 16 (1), 15-39.

Whatley, D. (1999) Designing Around Pitfalls of Game AI. In *Proceedings of the Game Developers' Conference*, San Jose, CA, 991-999.