CS 242 Project 2: Quagent Control

Darcey Riley, Adina Rubinoff, Greg Wilbur

February 28, 2010

Abstract

An intelligent agent must have some means of determining what it should do in different situations. One way to model this involves using a production system, which is a series of rules that dictate which actions an agent should take when faced with a particular situation. In this paper we describe a number of production systems that we have written in the Jess language for the purpose of controlling quagents.

1 Introduction

A production system is essentially a set of rules that tell an agent how to behave. These rules specify situations that can occur, and the actions that should be taken when these situations are encountered. The Jess language is a Java-based production system language which follows rules efficiently using the Rete algorithm, and thus is a good choice for programming production systems for intelligent agents.

Using production systems, we have developed five different "personalities" for quagents (see Appendix for actual production systems). All of them interact with objects in their environment, other quagents, or both. In this paper we will describe each quagent "personality" individually, then discuss how these different types of quagents interact with one another. Finally, we will talk about possible future work using these quagents.

2 Individual Quagents

2.1 ShyQuagent

The ShyQuagent production system is designed to model a shy or fearful quagent. It thus attempts to avoid other quagents whenever it sees them. When it is in a safe spot, it slowly turns in order to keep a lookout in all directions. As it turns, the ShyQuagent uses the RADIUS command to scan for other quagents. Although the RADIUS command looks in all directions, and thus the quagent has no actual need to turn around, we added in the turning to better model real-life behavior. Also, the turning gave us a way to tell if the quagent had crashed or was just silently looking for other quagents.

If another quagent is seen, the ShyQuagent quickly turns and runs in the opposite direction. Once it has run far enough, it once again stops and begins looking around.

One issue that may occur is that the RADIUS command will detect quagents that are separated from the ShyQuagent by a wall. When this occurs, the ShyQuagent will still run away from the detected quagent. This is a departure from human behavior, since a wall would normally be considered good protection from something one wants to avoid. However, this is not a problem for the quagent, since we can just assume the quagent has superior senses, and does not feel protected by walls.

2.2 ChaseQuagent

The ChaseQuagent, on the other hand, is designed to model an extremely friendly quagent, which walks towards other quagents whenever it seems them. Like the ShyQuagent, it slowly turns in place, using the Radius command to scan for other quagents. However, rather than running away from other quagents when they are detected, it either walks or runs towards them, depending on how far away they are. Once it has reached the other quagent, it stops and begins to look around once more.

As noted earlier, the RADIUS command will detect quagents that are on the other side of a wall from the ChaseQuagent. In this case, the ChaseQuagent will try to run through the wall to reach the other quagent. Since the wall is in its way, the ChaseQuagent will end up stuck at the wall, running in place, until it thinks it's run far enough. As long as the other quagent stays on the other side of the wall, and no other quagent comes closer, the ChaseQuagent will keep running towards it, trying and failing to run through the wall. However, we don't anticipate this being a large problem, since the ChaseQuagent only runs after other quagents, which also move around, and thus the environment will not remain the same for a long period of time.

2.3 SimpleQueen

As the name suggests, the SimpleQueen is an extremely simplistic quagent. It stands still until it detects gold using the BackPack's built-in probe() method, at which point it walks toward the gold and picks it up; it then continues to stand still until it detects more gold.

It should be noted that the probe() method can return items that are separated from the SimpleQueen by a wall. When this happens, the quagent will attempt to get to the gold anyway. Although we have not tested exactly what would happen in this situation, based on our experience with quagents we suspect that the SimpleQueen would walk until it reached the wall, think it picked up the gold, and search for more gold again. Because it always attempts to reach the nearest gold, the SimpleQueen would forever attempt to walk through the wall to get to that particular piece of gold.

2.4 GiverQuagent

The GiverQuagent is designed to model a very generous quagent. When it does not have gold, it searches for gold, using the BackPack's built-in probe() method. When it finds gold, it walks to the nearest piece of gold and picks it up. After the GiverQuagent has collected gold, it looks for the nearest "michael" quagent, walks towards it, and gives it the gold it has collected; it then probes for gold again and repeats this process.

Our first version of the GiverQuagent would drop gold near a "michael" quagent, then immediately look for more gold to pick up. As a result, it would pick up the gold it dropped before the "michael" quagent got the chance to take it for itself. For this reason, the final version of the GiverQuagent code has the GiverQuagent move a small distance away, then wait for a small amount of time, before looking for more gold to pick up.

As with the SimpleQueen, it should be noted that the probe() method can return items that are separated from the GiverQuagent by a wall. Again, the GiverQuagent will attempt to get to the gold anyway. We have not tested exactly what happens in this situation either, but we suspect that the quagent will think it reached the gold, think it picked the gold up, and bring the nothing it is actually carrying back to a "michael" quagent. Similarly, if it attempts to reach a "michael" that is behind a wall, we suspect it will simply deposit its gold at the wall where it is stopped and think it has given that gold to a "michael".

2.5 PirateQuagent

The PirateQuagent, much like an actual pirate, is driven by the need to collect as much gold as possible and hoard it all in one location. To do this, it must find a good location for hoarding gold. Rather than selecting some shadowy corner where no one else will find its stash, the PirateQuagent simply takes the location of the first gold it sees to be the hoarding location.

The first thing that the PirateQuagent does it look around for gold. In order to find gold, the PirateQuagent searches its environment using the visibleProbe() method. visibleProbe() calls the BackPack's probe() method, which in turn uses a RADIUS command; visibleProbe() then removes items which are behind walls from the BackPack's list of seen items, and, if a hoarding location has been chosen, it also removes items which are already in the hoarding location. If the PirateQuagent has detected any reachable gold that is not in the hoarding location already, it walks to that gold; otherwise, it walks to a random location and looks around again.

When the PirateQuagent reaches a piece of gold, if it has no hoarding location, it sets the hoarding location to be the location of that piece of gold. Otherwise, it picks up the piece of gold. While the

PirateQuagent has less than a certain amount of gold (set in the code to be 10, but easily changeable), it repeats the process of looking for gold, walking to the gold, and picking it up. When it has collected enough gold, it returns to the hoarding location to deposit it.

Rather than using any sort of fancy path-finding algorithm to determine the best path for getting back to the hoarding location, the PirateQuagent simply keeps a stack of all the locations it has visited since it left the hoarding location; then, when it has accumulated enough gold, it simply pops locations off of the stack and walks back to them until the stack is empty; it then has reached the hoarding location and can deposit the gold.

At first, we had the PirateQuagent push its current location after depositing gold at the hoarding location, but due to the imprecision of quagent movement, it deposited items further and further from the actual hoarding location each time it returned; in order to correct for this error, we now have the PirateQuagent push the actual hoarding location onto the stack after depositing gold.

However, we are still slightly concerned about this imprecision while walking, as in rare cases it may lead to the quagent getting stuck behind a wall. This would probably only happen if the PirateQuagent were trying to walk through an extremely narrow opening.

A last issue with the PirateQuagent involves the issue of "invisible gold". During our testing of the PirateQuagent, it tended to walk to locations where gold had previously been, then attempt to pick up gold, though none was actually there at that point in time. As a result it would return to the hoarding location before actually gathering a sufficient amount of gold. Though we examined our code and the existing code thoroughly we were unable to determine the source of this problem. We haven't seen this problem since we changed some of our code as well as our config files, so it may have been an anomaly; however, we are not certain that the issue of "invisible gold" will not reappear.

Also note that the PirateQuagent usually crashes after the program has been running for a long time; we suspect that the problem is somewhere in the probeVisible() method in AGDQuagent but are not positive. Despite this bug, the program runs for long enough to see the PirateQuagent behaving as it is supposed to.

3 Quagent Interaction

Many of our quagents are specifically designed to interact with other quagents. Thus, we now describe what happens when specific quagents are spawned in the general vicinity of one another.

3.1 Test 1: ShyQuagent and ChaseQuagent (Party Scenario)

The ShyQuagent and ChaseQuagent have essentially opposite behaviors; for this reason, we were particularly interested to see how they would interact. Thus, for our first test, we spawned a number of ShyQuagents near a number of ChaseQuagents in the Arena map. We chose the Arena map for testing in order to avoid problems with walls.

In this scenario, the ShyQuagents immediately run away from the other quagents, ending up near a wall. Once there, they stay by the wall for the rest of the scenario, avoiding the other quagents. The ChaseQuagents, on the other hand, all run towards each other, ending up in a large group in the center. The scenario stabilizes here, with a group of ChaseQuagents in the center, and the ShyQuagents scattered around the walls. This is much like a party scenario in real life; the more gregarious partygoers gather in the middle to socialize, while the shyer ones stay by the walls and just watch the party. Therefore we decided to name this scenario the Party scenario.

3.2 Test 2: ShyQuagent and ChaseQuagent (Tag Scenario)

A ShyQuagent and a ChaseQuagent should ideally play a small game of tag when spawned alone in an environment, with the ShyQuagent running away and the ChaseQuagent giving chase. When tested in the Arena map, however, the ShyQuagent immediately moved farther than the ChaseQuagent could detect, and then both stood in one place and just scanned. This shows that the ShyQuagent is better at its given task than the ChaseQuagent. However, it is likely that in a smaller environment, where the ShyQuagent doesn't have much room to escape, the ChaseQuagent would be better able to pursue, and a true game of tag would take place.

3.3 Test 3: SimpleQueen and GiverQuagent

The SimpleQueen and GiverQuagent are, in some sense, designed to work together: the GiverQuagent loves to share its gold, and the SimpleQueen wants to collect as much gold as possible. In order to model a symbiotic relationship between these two Quagents, we used a quagent.config file to spawn a "michael" SimpleQueen with the standard, extremely limited Wisdom and a GiverQuagent with extremely high Wisdom in a very large room, the Arena map, filled with gold. Because of the SimpleQueen's extremely limited Wisdom, it is unable to detect all of the gold around it and thus stands still most of the time. The GiverQuagent, which has extremely high Wisdom, runs around the room, collecting gold and dropping it near the SimpleQueen.

Most of the time, this interaction works as intended. However, quagent movement is imprecise, and the BackPack's drop() method, which is used to drop the gold, results in the gold being thrown a small distance away from the GiverQuagent dropping it. For these reasons, the SimpleQueen with its limited wisdom occasionally doesn't see the gold that has been dropped near it. Depending on where the GiverQuagent walks after dropping the gold, and whether there is gold there, it sometimes picks up the same piece of gold and attempts to bring it to the SimpleQueen again; other times it goes to get more gold. Sometimes this builds up such that the GiverQuagent essentially collects a pile of gold near the SimpleQueen, who hasn't moved.

4 Future Directions

4.1 Updates to Our Quagents

Obviously our first task for improving our program would be to eliminate any final bugs, such as the one that makes the PirateQuagent crash. Also, described above are a number of "imperfections" in the quagents' behaviors, largely resulting from quagents detecting items beyond walls and imprecision in quagent movement. Thus, another important future direction would be adding more complicated code that solves these problems.

4.2 More Tests to Run

Earlier, we described tests that we performed on our quagents; however, all of the quagents we tested together were essentially designed to interact with one another. It would be very interesting to see how other quagents might interact with each other; unfortunately we have not gotten a chance to test this yet. Thus, an important future direction would involve putting two different types of quagents in an arbitrary situation and seeing if/how they interact.

5 Appendix

5.1 Jess Rules for ShyQuagent

```
(defrule run-away
    "agent turns and runs from enemies"
    (runAway)
    =>
    (printout t "the agent sees an enemy and runs away....")
    (bind ?angle (+ (?*quake* getTurnItem ?*enemy*) 180.0))
    (bind ?*return-value* (?*quake* turn ?angle))
    (bind ?*return-value* (?*quake* runBy 1000.0))
    (retract ?*current-goal*)
    (bind ?*current-goal* (assert (get-situation)))
    (printout t "Agent runs away from enemy agent" crlf))

(defrule find-enemies
     "agent looks around to see if there are any enemies in its vicinity"
```

```
    (get-situation)
    =>
    (bind ?*current-situation* (?*quake* enemyProbe 1000))
    (bind ?danger (?*quake* getNumEnemies))
    (bind ?*enemy* (?*quake* getEnemy))
    (retract ?*current-goal*)
    (if (= ?danger 0) then
          (bind ?*current-goal* (assert (turn)))
       else
          (bind ?*current-goal* (assert (runAway))))
    )

(defrule turn
    "agent waits and then turns a slight amount"
    (turn)
    =>
    (?*quake* sleep 2000)
    (bind ?*return-value* (?*quake* turn 45.0))
    (retract ?*current-goal*)
    (bind ?*current-goal* (assert (get-situation)))
    (printout t "Agent turns" crlf))
```

5.2 Jess Rules for ChaseQuagent

```
(defrule find-friends
     "agent looks around to see if there are any friends in its vicinity"
    (get-situation)
    =>
    (bind ?*current-situation* (?*quake* enemyProbe 1000))
    (bind ?friends (?*quake* getNumEnemies))
    (retract ?*current-goal*)
    (if (= ?friends 0) then
          (bind ?*current-goal* (assert (turn)))
       else
          (bind ?*friend* (?*quake* getEnemy))
          (bind ?*current-goal* (assert (runTo))))
    )

(defrule run-to
     "agent runs towards friends"
    (runTo)
    =>
    (printout t "the agent sees a friend and walks towards it....")
    (bind ?*return-value* (?*quake* chaseItem ?*friend*))
    (retract ?*current-goal*)
    (bind ?*current-goal* (assert (get-situation)))
    (printout t "Agent runs toward friend agent" crlf))

(defrule turn
    "agent waits and then turns a slight amount"
    (turn)
    =>
    (?*quake* sleep 2000)
```

```
    (bind ?*return-value* (?*quake* turn 45.0))
    (retract ?*current-goal*)
    (bind ?*current-goal* (assert (get-situation)))
    (printout t "Agent turns" crlf)
)
```

5.3 Jess Rules for SimpleQueen

```
(defrule getItamz
    (empty)
    =>
    (?*queen* probe 100)
    (if ((?*queen* getMyBackPack) hasSeen gold) then
        (bind ?gold ((?*queen* getMyBackPack) getItem gold))
        (?*queen* walkToItem ?gold)
        (?*queen* pickUp ?gold)
    )
    (retract ?*state*)
    (bind ?*state* (assert (empty)))
)
```

5.4 Jess Rules for GiverQuagent

```
(defrule getItamz
    "If we are empty, we need to find gold"
    (empty)
    =>
    (?*giver* probe 1000000)
    (if ((?*giver* getMyBackPack) hasSeen gold) then
        (bind ?gold ((?*giver* getMyBackPack) getItem gold))
        (?*giver* walkToItem ?gold)
        (?*giver* pickUp ?gold)
        (retract ?*state*)
        (bind ?*state* (assert (full)))
    )
)

(defrule giveItamz
    "If we are full, we need to give the gold to somebody"
    (full)
    =>
    (?*giver* probe 1000000)
    (if ((?*giver* getMyBackPack) hasSeen michael) then
        (bind ?person ((?*giver* getMyBackPack) getItem michael))
        (?*giver* walkToItem ?person)
        (?*giver* drop gold)
        ; turn around and go away so we don't get in the way of the
        ; person we give the gold to.
        (?*giver* turn 180.0)
        (?*giver* walkBy 200)
        (?*giver* sleep 3000)
        (retract ?*state*)
        (bind ?*state* (assert (empty)))
```

```
    )
)
```

5.5 Jess Rules for PirateQuagent

```
(defrule look-for-gold1
    "checks if any gold is visible"
    (find-gold)
    (need-hoarding-loc)
    =>
    (?*quake* visibleProbe 500)
    (if (?*quake* seenGold)
        then
            (bind ?*nearest-gold* (?*quake* nearestGold))
            (retract ?*current-goal*)
            (bind ?*current-goal* (assert (have-found-gold)))
        else
            (retract ?*current-goal*)
            (bind ?*current-goal* (assert (go-somewhere-else)))
    )
)

(defrule look-for-gold2
    "checks if any gold is visible"
    (find-gold)
    (have-hoarding-loc)
    =>
    (?*quake* visibleProbe 500 ?*hoarding-loc*)
    (if (?*quake* seenGold)
        then
            (bind ?*nearest-gold* (?*quake* nearestGold))
            (retract ?*current-goal*)
            (bind ?*current-goal* (assert (have-found-gold)))
        else
            (retract ?*current-goal*)
            (bind ?*current-goal* (assert (go-somewhere-else)))
    )
)

(defrule go-elsewhere
    "no more gold in this area; look for more somewhere else"
    (go-somewhere-else)
    =>
    (bind ?*goto-x* (+ (?*quake* randFloat 400.0) (?*quake* getPlayerX)))
    (bind ?*goto-y* (+ (?*quake* randFloat 400.0) (?*quake* getPlayerY)))
    ;(printout t ?*goto-x* crlf)
    ;(printout t ?*goto-y* crlf)
    (retract ?*current-goal*)
    (bind ?*current-goal* (assert (test-if-new-loc-reachable)))
)

(defrule reachable-test
    "have determined a goal location; need to find out if it's reachable"
    (test-if-new-loc-reachable)
```

```
    =>
    (?*quake* turnTo ?*goto-x* ?*goto-y*)
    (printout t ?*goto-x* crlf)
    (printout t ?*goto-y* crlf)
    (bind ?*dist-to-walk* (?*quake* getDist ?*goto-x* ?*goto-y*))
    (if (< (?*quake* getDistanceAhead) ?*dist-to-walk*)
        then
            (retract ?*current-goal*)
            (bind ?*current-goal* (assert (go-somewhere-else)))
        else
            (retract ?*current-goal*)
            (bind ?*current-goal* (assert (goto)))
    )
)

(defrule need-to-move
    "walk to a specified location, known to be reachable"
    (goto)
    =>
    (?*quake* walkBy ?*dist-to-walk*)
    (?*path* push (?*quake* getWhere))
    (retract ?*current-goal*)
    (bind ?*current-goal* (assert (find-gold)))
)

(defrule found-gold
    "there is gold in range"
    (have-found-gold)
    =>
    (?*quake* walkTo (?*nearest-gold* getX) (?*nearest-gold* getY))
    (?*path* push (?*quake* getWhere))
    (retract ?*current-goal*)
    (bind ?*current-goal* (assert (at-the-gold)))
)

(defrule at-gold1
    "at location of gold, don't have hoarding loc"
    (at-the-gold)
    (need-hoarding-loc)
    =>
    (bind ?*hoarding-loc-x* (?*nearest-gold* getX))
    (bind ?*hoarding-loc-y* (?*nearest-gold* getY))
    (bind ?*hoarding-loc* ?*nearest-gold*)
    (retract ?*hoarding-loc-found*)
    (bind ?*hoarding-loc-found* (assert (have-hoarding-loc)))
    (while (not (?*path* empty))
        do (?*path* pop)
    )
    (?*path* push (?*quake* getWhere))
    (retract ?*current-goal*)
    (bind ?*current-goal* (assert (find-gold)))
)

(defrule at-gold2
```

```
    "at location of gold"
    (at-the-gold)
    (have-hoarding-loc)
    =>
    (bind ?loc-x (?*quake* getPlayerX))
    (bind ?loc-y (?*quake* getPlayerY))
    (?*quake* pickUp ?*nearest-gold*)
    (bind ?*amount-gold* (+ ?*amount-gold* 1))
    (if (< ?*amount-gold* 10)
        then
            (retract ?*current-goal*)
            (bind ?*current-goal* (assert (find-gold)))
        else
            (retract ?*current-goal*)
            (bind ?*current-goal* (assert (hoard)))
    )
)

(defrule back-to-hoard
    "walk back to the hoard"
    (hoard)
    =>
    (bind ?next-loc (?*path* pop))
    (?*quake* walkTo (?next-loc getX) (?next-loc getY))
    (if (?*path* empty)
        then
            (retract ?*current-goal*)
            (bind ?*current-goal* (assert (drop)))
        else
            (retract ?*current-goal*)
            (bind ?*current-goal* (assert (hoard)))
    )
)

(defrule drop-all
    "drop all items at hoarding location"
    (drop)
    =>
    (?*quake* dropAll)
    (?*path* push ?*hoarding-loc*)
    (bind ?*amount-gold* 0)
    (retract ?*current-goal*)
    (bind ?*current-goal* (assert (go-somewhere-else)))
)
```