Modeling Priority Management Strategies

Using UR-Quagents and JESS

Tom O'Neill Harry Glaser Leland Aldridge

CSC242, Dept. of Computer Science, University of Rochester

{toneill, hglaser, la002k}@mail.rochester.edu

Abstract

Complex agents with intelligent behaviors often have multiple potential behaviors to employ at any one time. Selecting the best behavior is usually non-trivial because it can depend on the situation. Using JESS and the UR-Quagent environment, we model three priority management strategies using goal-oriented agents in a simulated environment. Our results are inconclusive due to weaker than expected payoffs for certain actions in the environment, thus severely limiting our ability to compare strategies built for use under different environmental assumptions. Future trials should adjust the payoffs of certain actions based on these results.

Introduction

Production systems are ideal environments for programming behaviors into agents without enforcing a rigid control structure. One way to implement simple intelligent behavior in a production system is to use behavior states (Maes & Brooks, *Learning to Coordinate Behaviors*). In this scenario, several behavior states may be viable options at any time, and the agent needs to choose between them. To guide the agent with this task, the behavior states are ranked by their priority, thus the agent will always choose the available state with the highest priority.

The complexity of the problem arises at the point where priorities are assigned. Our approach identifies three potential strategies and applies them to the agent's behaviors. Our agent's behaviors are set up such that the complexities of more intelligent agents are abstracted away and our agent's behavior is entirely dependent on the priority strategy used on a particular choice. The choice the agent is presented with involves choosing which of two actions to perform in the environment, one option representing a short term payoff and the other a long term payoff. The strategies are evaluated based on the agents' relative success when compared to a common goal known by the agent.

We expect that a strategy that satisfies a 'compromise' between conflicting behaviors will lead to the best performance toward the goal, where strategies that concentrate on satisfying a single behavior's requirements will not lead to optimal agent behavior.

Methods

We used the University of Rochester Quake II Quagent environment to simulate intelligent agents in a virtual world. The JESS production system was used to both create and control the behavior of the agents.

Environment

At the start of each behavior session, a quagent was placed in the center of a room containing a 7x9 randomized grid of items that the quagent could interact with. For each trial the same randomized grid was used. Each item was in the center of a grid square, and grid points were 150 units apart. The grid contained 52 pieces of gold, 8 battery packs, and 3 pieces of kryptonite. The gold acted as a source of wealth, the batteries acted as a source of energy, and the kryptonite caused harm to the quagent if the agent traveled within 200 units of it. The distribution of items in the agent's environment is shown in Figure M.1. The agent always started in the middle of the grid.

G	G	G	K	G	G	B	G	B
G	G	B	G	G	G	G	G	G
G	G	G	G	G	G	G	G	G
G	G	B	G	G	G	G	B	G
G	G	G	G	B	G	G	G	G
G	G	G	K	B	G	G	G	G
G	G	B	G	G	G	G	G	B

Figure M.1
(G = gold, N = batteries, K = kryptonite)

Priorities

To allow the agents to model the implemented priority management strategies, the agent's behavior priorities were split into to three groups:

1. The Greater Priority Behaviors (GPB) – this group contains all of the behavior priorities in effect above the experimental behaviors and hence dominates the agent's behavior when in use.
2. The Strategy Priority Behaviors (SPB) – this group contains the two behaviors (long term benefit and short term benefit) that are managed using the current priority management strategy.
3. The Lesser Priority Behaviors (LPB) – this group contains the behaviors that only execute after both previous groups cannot act.

These three groups allowed the priority management strategies to influence each agent in a predicable way, thus allowing for the evaluation of each strategy in the context of the agent's environment and goal.

Behaviors

While in the environment, the quagents acted out one of four behaviors at a time. Each agent had the same four behaviors to choose from:

1. Avoid kryptonite – this constituted the GPB for each quagent. This behavior forced quagents to avoid nearness to kryptonite.
2. Collect gold – this is the first of two behaviors in the SPB. While executing this behavior, quagents sought out piles of gold and collected them.

3. Collect batteries – the second behavior in the SPB. While executing this behavior, quagents sought out battery packs and collected them.
4. Explore – the default action of the quagent, representing the LPB. Agents in explore mode turned a random number of degrees and walked random distances around the environment.

The goal of each quagent was to collect as much gold as possible before running out of energy. At no point were quagents allowed to sacrifice health in order to collect gold. Each action required a small amount of energy, thus leading to the development of three different strategies to handle the SPB. (Refs to Books-style papers?)

Sensors

Each agent could view the immediate area around it (1000 unit radius from the agent's position). The agent was able to determine the location and object type of every object in its radius, as well as use its own position and heading to calculate how to move toward or away from any viewable object. The agent did not remember previously seen objects. Thus, each action by the agent is determined by what is sees in its viewable radius at any moment.

Strategies

All agents had the same first last priorities, avoid kryptonite and explore respectively. The three priority management strategies implemented to handle

the two SPB, collecting gold and collecting batteries while avoiding kryptonite, were:

1. Greedy Agent (GA) - Ignore energy requirements and always seek gold.
2. Frantic Agent (FA) – Collect gold until energy runs low, then explore until enough batteries are found to restore energy levels.
3. Pragmatic Agent (PA) – Collect gold until energy runs low, then collect gold or batteries, with batteries given precedence.

Several agents operating under each strategy were placed in the environment (one at a time) and each acted out the behaviors deemed as having the highest priority. Each agent was allowed to act in the environment until its energy fell to zero. At that point, the results were recorded and the simulation reset.

Results

Agents were evaluated on several criteria: maximum wealth collected, time taken to collect the wealth, and the lifetime of the agent. The average values of these criteria for each agent are shown in Figure R.1, along with some calculated results. In this environment, time is measured in 'ticks', each tick represents a period of movement. Quagents could turn and pickup items between movement ticks, depending on their behavior state.

	Wealth	Time to Max Wealth	Lifetime	Lifetime Collection Rate
Frantic Agent	160	21	100.5	1.6

Greedy Agent	520	72	90.5	5.7
Pragmatic Agent	485	71.5	87	5.6

Figure R.1

Wealth is the total amount of gold the agent collected. Time to Max Wealth is the number of ticks in the environment until the last time the quagent picked up gold. Lifetime is the total number of ticks spent in the environment before death. Lifetime Collection Rate is the rate at which the quagent collected gold during its lifetime (Wealth / Lifetime). Graphical versions of the above chart follow in Figures R.2 – R.5.

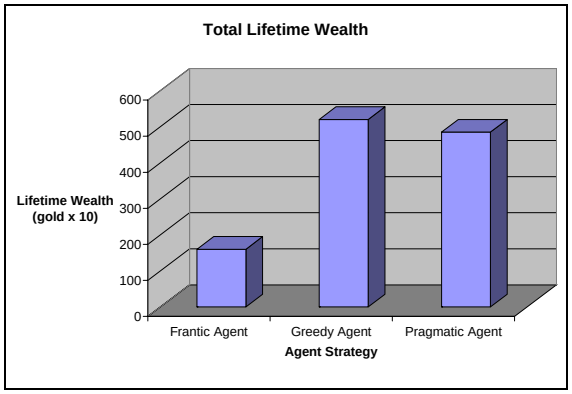


Figure R.2

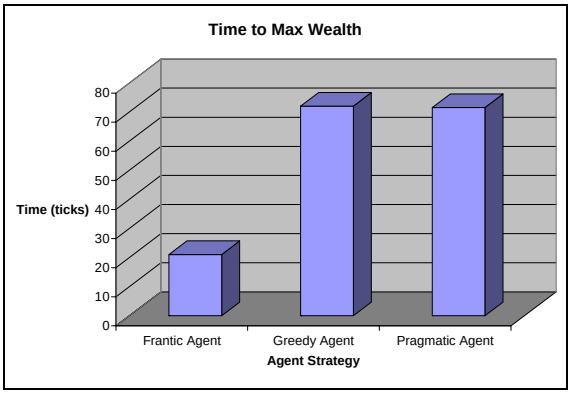


Figure R.3

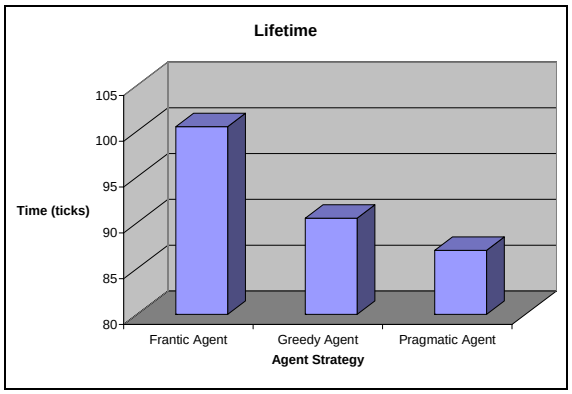


Figure R.4

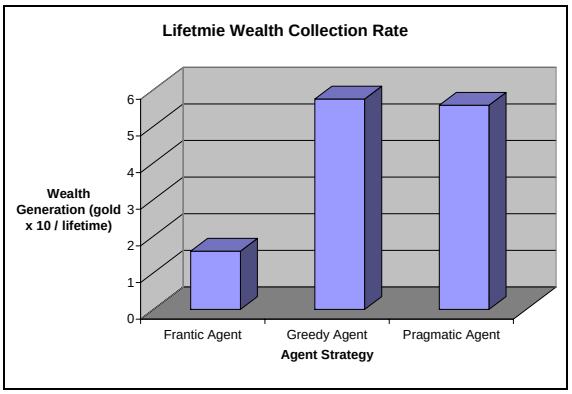


Figure R.5

In addition to the above measurements, each tick of each quagent was graphically mapped. The 5th, 20th, 40th, and 80th ticks for representative trials of each agent strategy are in Appendix 1.

Discussion

Our hypothesis suggests that the Pragmatic Agent would have the most success in attaining the goal. However, this is not the case since it does not acquire wealth at the highest rate nor does it attain the most wealth compared to the other two strategies. In an effort to explain this interesting phenomenon, we analyze each agent's performance. In all cases, the agents did not use any planning mechanisms to determine optimal paths, their actions were based entirely on the immediately viewable radius.

The Frantic Agent

The most glaring difference between the three strategies separates the Frantic Agent from the other two. Its strategy of fulfilling the secondary objective first (i.e. replenish energy before pursuing more gold) dramatically hurt its performance compared to the other agent strategies. The relative differences between the Frantic Agent and the other two agents in both Max Wealth and Time to Max Wealth suggest that it accrued wealth at an equal pace to the other two agents early on, and then no longer accrued any wealth once it ran low on energy. Appendix 1 shows that the exploratory nature of the Frantic Agent actually causes it to leave the main area of the environment and get lost near the boundaries.

The most likely cause for this error on the part of the agent is that the battery packs were either too sparse, or did not contain enough energy to offset the cost of exploration and thus were essentially useless.

However, the Frantic Agent does perform slightly better than the other two agents regarding life spans. It appears as if the Frantic Agent conserves energy in an effort to have enough time to explore and find more battery packs.

The frantic agent seems better suited for environments with stronger batteries, or higher densities of batteries, or environments where lifespan is a factor in achieving the goal.

The Greedy Agent

This agent ignored the requirement for energy and strictly collected gold. For this environment, this strategy was the most successful because this agent strategy collected more gold than any other agent strategy. This result could be the effect of an experiment that was too short, thus allowing an agent seeking short term gains to take the lead. However, that does not appear to be the case since the Pragmatic Agent, a design that should outperform a greedy strategy in the long run, collected slightly less gold.

This situation lends more evidence to the idea that the battery packs do not carry enough energy to be worth picking up. This strategy shows that energy did not play an important enough role in this environment to be worth picking up.

The Pragmatic Agent

This agent strategy was expected to perform the best of the three agents. Due to potential issues related to the power quantity of the battery packs throughout the environment, this agent performed worse than expected.

This agent strategy produced the shortest lifetime of all three strategies, yet it performed almost as well as the greedy agent regarding the total amount of gold collected. This is likely the result of the battery packs actually hampering its ability to collect gold because they do not seem to positively affect the quagent in any measurable way.

We believe the problem regarding the performance of these agents is related to the battery packs not carrying enough energy to make them useful in aiding quagent behavior. For this reason, we cannot accurately evaluate the effectiveness of the behavior priority strategies because the intended behaviors were not emergent in the agents. We discuss potential fixes and ideas for further exploration of priority strategy management in the next section.

Future Work

Future efforts to model priority management strategies in the Quagents/JESS environment need to account for the potential weakness of the battery packs in the environment. We suggest recompiling the Quake engine to provide for larger energy values on battery packs, or placing many more battery packs (potentially in the same location) in the environment.

After the energy issue has been corrected, future work could include testing the effectiveness of these strategies on several more environments, especially those that are not grid based. Environments that are engineered to force the quagents into varying levels of behavior complexity would be useful in

testing more intelligent behaviors. For example, the environment could be set up with kryptonite near batteries and experimenters could allow strategies that account for a desperation factor, thus the quagent could travel closer to kryptonite/battery dense areas if the quagent was especially low on energy.

Work Distribution

This project was worked on by Tom O'Neill, Harry Glaser, and Leland Aldridge, their contributions are as follows:

Tom O'Neill wrote this paper and programmed most of the JESS Quagent behaviors and Java interfacing, and ran the behavior tests.

Harry Glaser set up the Jess/Java framework and performed initial development and functional testing.

Leland Aldridge is responsible for developing the quagent behaviors and the frsmework for priority strategy modeling.

Appendix 1: Graphical Displays of Quagent Movements

The following are graphical printouts of representative behavior sequences of each quagent strategy. The colored dots on the map represent all of the objects the quagent can see (quagents do not have memories). The blue arrows represent the agent's successive moves.

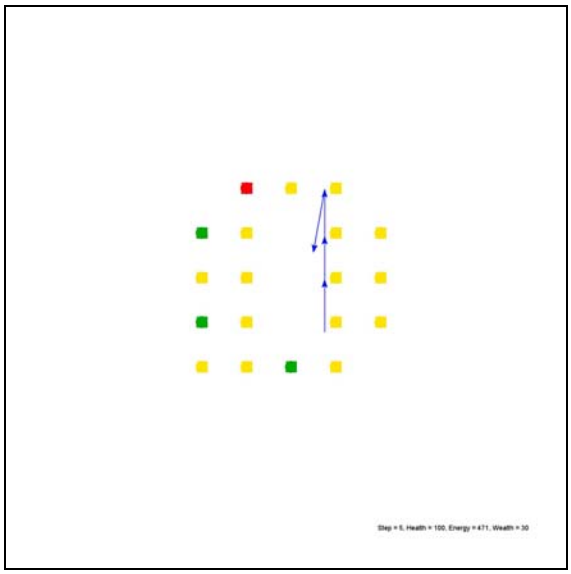
Dot Key:

Red = kryptonite

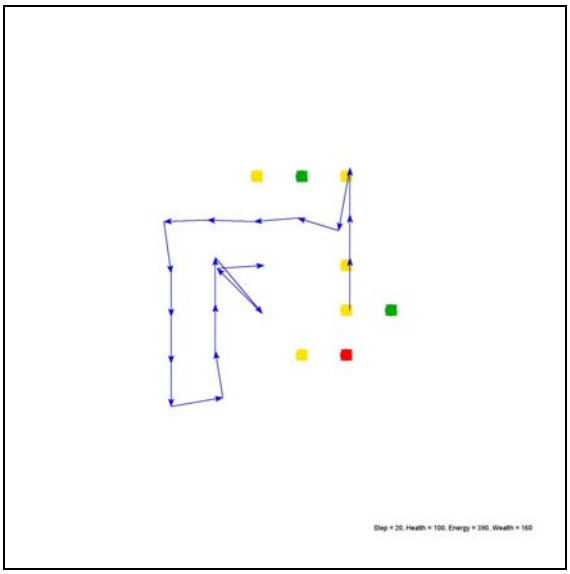
Green = batteries

Yellow = gold.

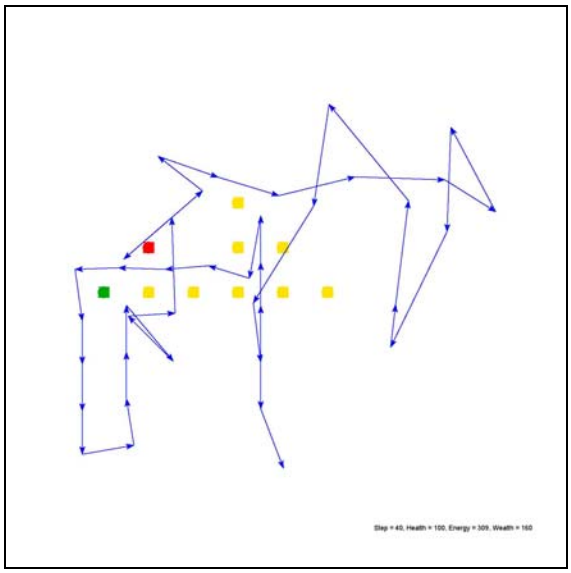
Frantic Agent



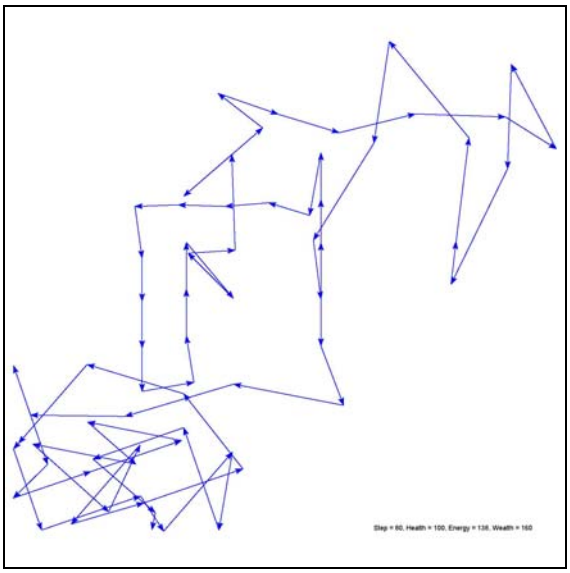
Step 5



Step 20



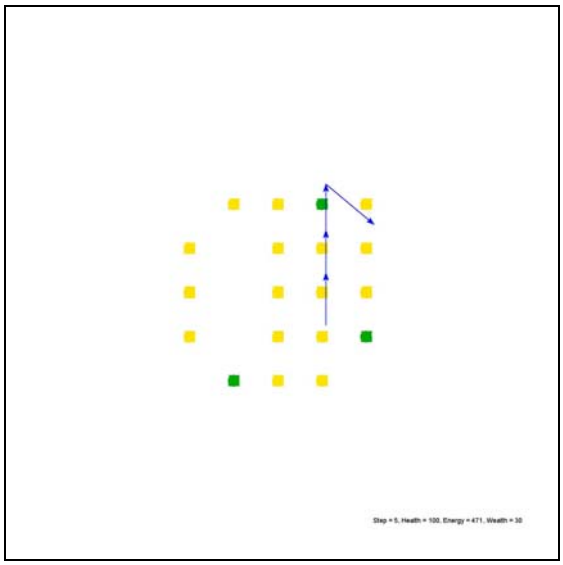
Step 40



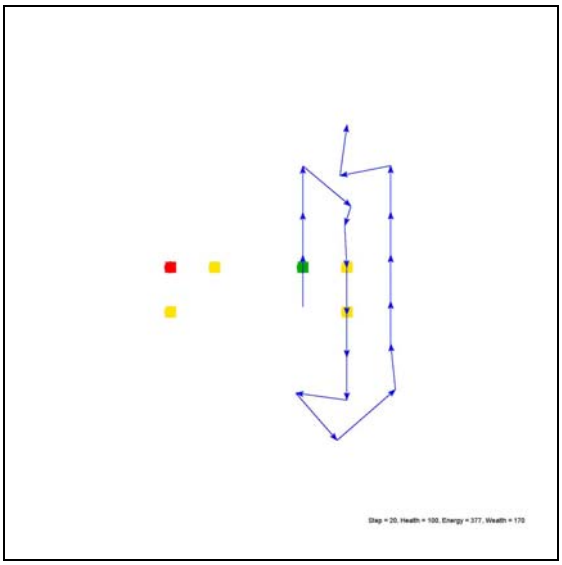
Step 80

Time slices from a sample run of the Frantic Agent's behavior trials. By step 40 he has left field of objects and by step 80 he is lost near the border of the environment.

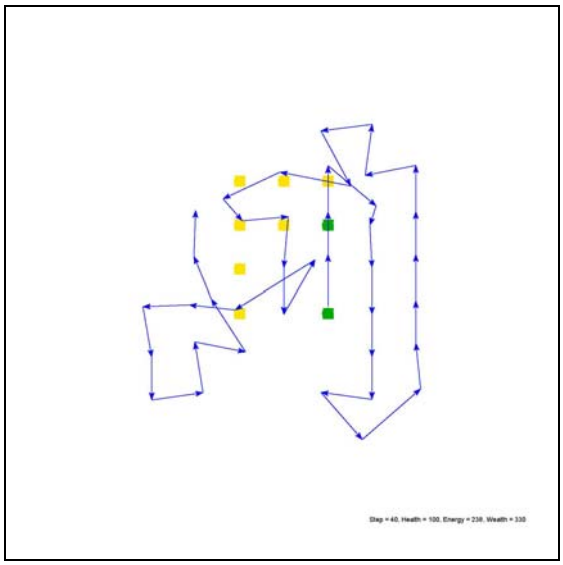
Greedy Agent



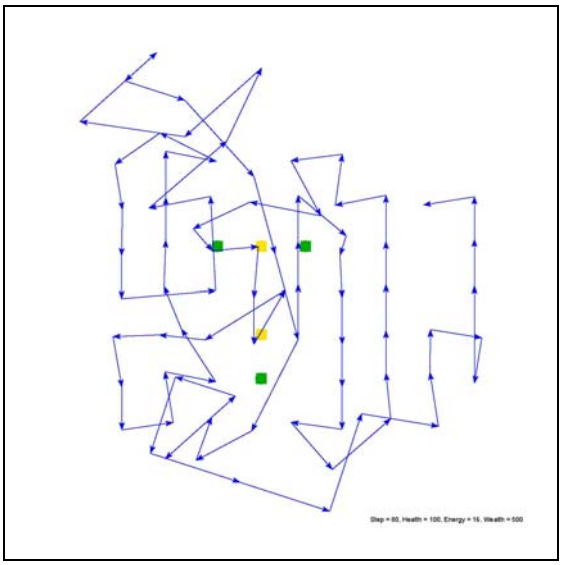
Step 5



Step 20



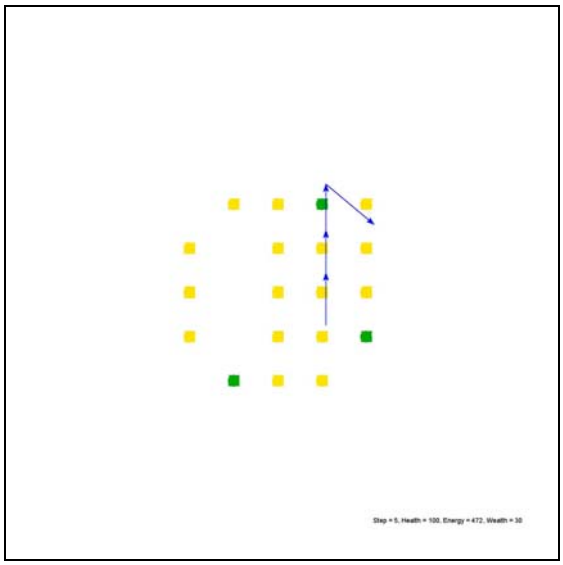
Step 40



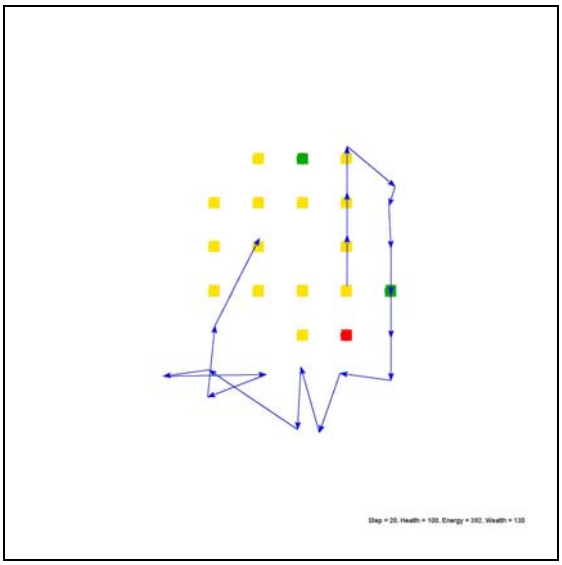
Step 80

Time slices from a sample run of the Greedy Agent's behavior trials. Step 80 shows the methodical nature of the agent as it collects gold and ignores everything else but kryptonite.

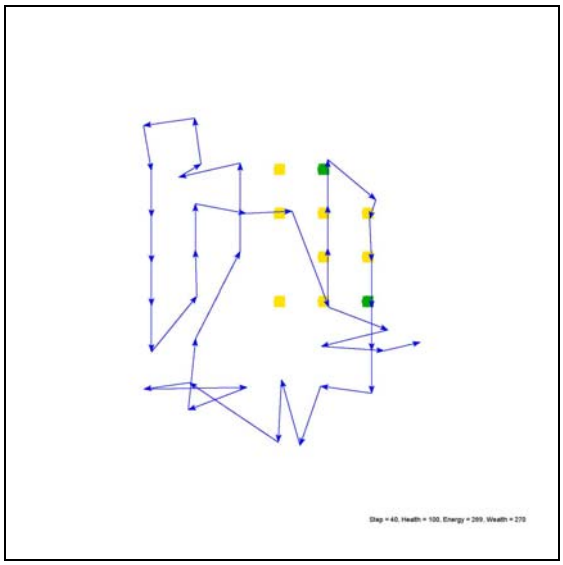
Pragmatic Agent



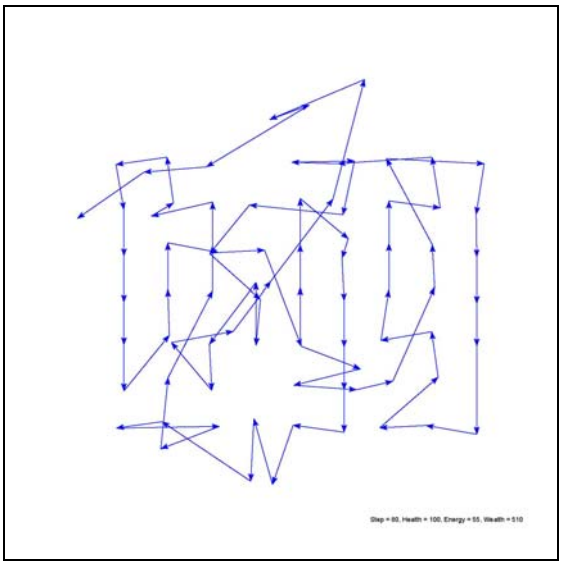
Step 5



Step 20



Step 40



Step 80

Time slices from a sample run of the Pragmatic Agent's behavior trials. Like the Greedy Agent, step 80 shows a methodical process of collecting items, with a few excursions for collecting batteries.

Appendix 2: JESS Code for each agent

The following pages are the complete JESS code files used to orchestrate the behaviors of the agents.

Frantic Agent

```
(defglobal
    ?*quagent* = 0 ;mr quagent
    ?*actionState* = 0 ;state of the quagent

    ?*currentVision* = 0 ; array of objects in local area

    ?*target* = 0 ;target index into vision array
    ?*energy* = -1 ;energy level of quagent, inits to 1000

    ;constants
    ?*seekEnergyAt* = 400 ;level at which quagent will start seeking energy
    ?*fleeDistance* = 200 ;distance to back away when quagent sees something bad
    ?*searchRadius* = 1000 ;radius of probe to seek objects (max: 1000)
    ?*minDistToKrypt* = 200 ;flee at this radius
    ?*noTarget* = -1 ;when no target exsts in the vision array

    ;jess into mapper names:
    ?*mapperName* = "myMapper"
    ?*addMovement* = "acceptPos" ;(x,y)
    ?*clearViewedItems* = "resetSpecial" ;()
    ?*addViewedPoint* = "addSpecialPos" ;(x,y,"type")
    ?*setHealth* = "setHealth" ;(val)
    ?*setEnergy* = "setEnergy" ;(val)
    ?*setWealth* = "setWealth" ;(val)
)

(deffunction init ()
    (printout t "#Function: init" crlf)

    (bind ?*quagent* (new ControlledQuagent))
    (?*quagent* initialize)
    (bind ?*actionState* (assert(lookAround)))
)

(deffunction close ()
    (?*quagent* close)
)

;scan max radius for potential targets and store scan for later
;also, check/update current energy level and health
(defrule lookForTargets
    (lookAround)
    =>
    (printout t "#Rule: lookForTargets" crlf)

    ;mapper output - movement!
    ;(bind ?pos (?*quagent* pos))
    ;(printout t ?*mapperName* "." ?*addMovement* "((" (?pos getX) ", " (?pos getY) "))"
crlf)

    (bind ?*currentVision* (?*quagent* probe ?*searchRadius*))
    (bind ?*energy* ((?*quagent* getWellbeing) getEnergy))
    (bind ?health ((?*quagent* getWellbeing) getHealth))
    (bind ?wealth ((?*quagent* getWellbeing) getWealth))

    ;mapper output - radius
    ;(printout t ?*mapperName* "." ?*clearViewedItems* "()" crlf)
    ;(bind ?len (- (?*currentVision* numItems) 1))
    ;(while (>= ?len 0)
    ;    (printout t ?*mapperName* "." ?*addViewedPoint* "(" (?*currentVision* getX ?len)
", " (?*currentVision* getY ?len) ", \"" (?*currentVision* getName ?len) "\")" crlf)
```

```
    ;    (bind ?len (- ?len 1))
    ;)

    ;draw quagent's state
    ;(printout t ?*mapperName* "." ?*setEnergy* "(" ?*energy* ")" crlf)
    ;(printout t ?*mapperName* "." ?*setHealth* "(" ?health ")" crlf)
    ;(printout t ?*mapperName* "." ?*setWealth* "(" ?wealth ")" crlf)
    ;(printout t ?*mapperName* ".draw()" crlf)

    (printout t "#    Current Energy: " ?*energy* crlf)
    (printout t "#    Current Health: " ?health crlf)
    (printout t "#    Current Wealth: " ?wealth crlf)

    (retract ?*actionState*)
    (bind ?*actionState* (assert(checkSurroundings)))
)

;searches scan for kyrptonite and if found, flees from it
;search only applies when kyptonite is seen
(defrule lookForKryptonite
    (declare (salience 30))
    (checkSurroundings)
    (test (neq (?*currentVision* getClosest "kryptonite" (?*quagent* pos)
 ?*minDistToKrypt*) ?*noTarget*))
    =>
    (printout t "#Rule: lookForKryptonite" crlf)

    (bind ?*target* (?*currentVision* getClosest "kryptonite" (?*quagent* pos)))

    (retract ?*actionState*)
    (bind ?*actionState* (assert(fleeTarget)))
)

;searches scan for gold and if found, goes there
;search only applies when bot has energy above it's
;  low-threshold and gold is in the scan
(defrule lookForGold
    (declare (salience 20))
    (checkSurroundings)
    (test (neq (?*currentVision* getClosest "gold" (?*quagent* pos)) ?*noTarget*))

    ;get gold only if not low on energy ;
    (test (> ?*energy* ?*seekEnergyAt*))
    =>
    (printout t "#Rule: lookForGold" crlf)

    (bind ?*target* (?*currentVision* getClosest "gold" (?*quagent* pos)))

    (retract ?*actionState*)
    (bind ?*actionState* (assert(seekTarget)))
)

;searches scan for batteries and if found, goes there
;search only applies if the agent either cannot find gold or
;  needs energy and batteries are in the scan
(defrule lookForEnergy
    (declare (salience 10))
    (checkSurroundings)
    (test (neq (?*currentVision* getClosest "battery" (?*quagent* pos)) ?*noTarget*))
    =>
    (printout t "#Rule: lookForEnergy" crlf)
```

```
    (bind ?*target* (?*currentVision* getClosest "battery" (?*quagent* pos)))

    (retract ?*actionState*)
    (bind ?*actionState* (assert(seekTarget)))
)

;handles moving the agent to the desired target
(defrule goToTarget
    (seekTarget)
    =>
    (printout t "#Rule: goToTarget" crlf)
    (printout t "#    Target = " (?*currentVision* getName ?*target*) crlf)

    (?*quagent* turn (* ((?*quagent* pos) getYaw) -1));reset yaw to 0, makes math easier

    (bind ?dist  (?*currentVision* getDist ?*target* (?*quagent* pos)))
    (bind ?turn  (?*currentVision* getAngle ?*target* (?*quagent* pos)))

    (?*quagent* turn ?turn)
    (?*quagent* run ?dist)

    (retract ?*actionState*)
    (bind ?*actionState* (assert(atTarget)))
)

;handles moving the agent to the desired target
(defrule fleeFromTarget
    (fleeTarget)
    =>
    (printout t "#Rule: fleeFromTarget" crlf)
    (printout t "#    Target = " (?*currentVision* getName ?*target*) crlf)

    (?*quagent* turn (* ((?*quagent* pos) getYaw) -1));reset yaw to 0, makes math easier

    ;flee in opposite direction +/- 30 degrees to avoid loops
    (bind ?dist  ?*fleeDistance*)
    (bind ?turn  (+ (?*currentVision* getAngle ?*target* (?*quagent* pos)) 180))
    (bind ?turn  (- ?turn (* (?*quagent* randInt 3) 30)))

    (?*quagent* turn ?turn)
    (?*quagent* run ?dist)

    (retract ?*actionState*)
    (bind ?*actionState* (assert(lookAround)))
)

;picks up wanted objects when standing over them
(defrule pickUpTarget
    (atTarget)
    =>
    (printout t "#Rule: pickUpTarget" crlf)

    (?*quagent* pickup (?*currentVision* getName ?*target*))

    (retract ?*actionState*)
    (bind ?*actionState* (assert(lookAround)))
)

;when there's nothing else to do, explore
(defrule exploreSurroundings
    (declare (salience 0))
    (checkSurroundings)
```

```
    =>
    (printout t "#Rule: exploreSurroundings" crlf)

    (?*quagent* turn  (* (?*quagent* randInt 12) 30)) ;random multiple of 30 degrees
    (?*quagent* run  (+ (* (?*quagent* randInt 6) 50) 200)) ;random multiple of 50 in
[200, 450]

    (retract ?*actionState*)
    (bind ?*actionState* (assert(lookAround)))
)




(reset)
(init)
(run)
(close)
```

Greedy Agent

```
(defglobal
    ?*quagent* = 0 ;mr quagent
    ?*actionState* = 0 ;state of the quagent

    ?*currentVision* = 0 ; array of objects in local area

    ?*target* = 0 ;target index into vision array
    ?*energy* = -1 ;energy level of quagent, inits to 1000

    ;constants
    ?*seekEnergyAt* = 400 ;level at which quagent will start seeking energy
    ?*fleeDistance* = 200 ;distance to back away when quagent sees something bad
    ?*searchRadius* = 1000 ;radius of probe to seek objects (max: 1000)
    ?*minDistToKrypt* = 200 ;flee at this radius
    ?*noTarget* = -1 ;when no target exsts in the vision array

    ;jess into mapper names:
    ?*mapperName* = "myMapper"
    ?*addMovement* = "acceptPos" ;(x,y)
    ?*clearViewedItems* = "resetSpecial" ;()
    ?*addViewedPoint* = "addSpecialPos" ;(x,y,"type")
    ?*setHealth* = "setHealth" ;(val)
    ?*setEnergy* = "setEnergy" ;(val)
    ?*setWealth* = "setWealth" ;(val)
)

(deffunction init ()
    (printout t "#Function: init" crlf)

    (bind ?*quagent* (new ControlledQuagent))
    (?*quagent* initialize)
    (bind ?*actionState* (assert(lookAround)))
)

(deffunction close ()
    (?*quagent* close)
)

;scan max radius for potential targets and store scan for later
;also, check/update current energy level and health
(defrule lookForTargets
    (lookAround)
    =>
    (printout t "#Rule: lookForTargets" crlf)

    ;mapper output - movement!
    ;(bind ?pos (?*quagent* pos))
    ;(printout t ?*mapperName* "." ?*addMovement* "((" (?pos getX) ", " (?pos getY) "))"
crlf)

    (bind ?*currentVision* (?*quagent* probe ?*searchRadius*))
    (bind ?*energy* ((?*quagent* getWellbeing) getEnergy))
    (bind ?health ((?*quagent* getWellbeing) getHealth))
    (bind ?wealth ((?*quagent* getWellbeing) getWealth))

    ;mapper output - radius
    ;(printout t ?*mapperName* "." ?*clearViewedItems* "()" crlf)
    ;(bind ?len (- (?*currentVision* numItems) 1))
    ;(while (>= ?len 0)
    ;    (printout t ?*mapperName* "." ?*addViewedPoint* "(" (?*currentVision* getX ?len)
", " (?*currentVision* getY ?len) ", \"" (?*currentVision* getName ?len) "\")" crlf)
```

```
    ;    (bind ?len (- ?len 1))
    ;)

    ;draw quagent's state
    ;(printout t ?*mapperName* "." ?*setEnergy* "(" ?*energy* ")" crlf)
    ;(printout t ?*mapperName* "." ?*setHealth* "(" ?health ")" crlf)
    ;(printout t ?*mapperName* "." ?*setWealth* "(" ?wealth ")" crlf)
    ;(printout t ?*mapperName* ".draw()" crlf)

    (printout t "#    Current Energy: " ?*energy* crlf)
    (printout t "#    Current Health: " ?health crlf)
    (printout t "#    Current Wealth: " ?wealth crlf)

    (retract ?*actionState*)
    (bind ?*actionState* (assert(checkSurroundings)))
)

;searches scan for kyrptonite and if found, flees from it
;search only applies when kyptonite is seen
(defrule lookForKryptonite
    (declare (salience 30))
    (checkSurroundings)
    (test (neq (?*currentVision* getClosest "kryptonite" (?*quagent* pos)
 ?*minDistToKrypt*) ?*noTarget*))
    =>
    (printout t "#Rule: lookForKryptonite" crlf)

    (bind ?*target* (?*currentVision* getClosest "kryptonite" (?*quagent* pos)))

    (retract ?*actionState*)
    (bind ?*actionState* (assert(fleeTarget)))
)

;searches scan for gold and if found, goes there
(defrule lookForGold
    (declare (salience 20))
    (checkSurroundings)
    (test (neq (?*currentVision* getClosest "gold" (?*quagent* pos)) ?*noTarget*))
    =>
    (printout t "#Rule: lookForGold" crlf)

    (bind ?*target* (?*currentVision* getClosest "gold" (?*quagent* pos)))

    (retract ?*actionState*)
    (bind ?*actionState* (assert(seekTarget)))
)

;searches scan for batteries and if found, goes there
;search only applies if the agent either cannot find gold or
;  needs energy and batteries are in the scan
(defrule lookForEnergy
    (declare (salience 10))
    (checkSurroundings)
    (test (neq (?*currentVision* getClosest "battery" (?*quagent* pos)) ?*noTarget*))
    =>
    (printout t "#Rule: lookForEnergy" crlf)

    (bind ?*target* (?*currentVision* getClosest "battery" (?*quagent* pos)))

    (retract ?*actionState*)
    (bind ?*actionState* (assert(seekTarget)))
)
```

```
;handles moving the agent to the desired target
(defrule goToTarget
    (seekTarget)
    =>
    (printout t "#Rule: goToTarget" crlf)
    (printout t "#    Target = " (?*currentVision* getName ?*target*) crlf)

    (?*quagent* turn (* ((?*quagent* pos) getYaw) -1));reset yaw to 0, makes math easier

    (bind ?dist  (?*currentVision* getDist ?*target* (?*quagent* pos)))
    (bind ?turn  (?*currentVision* getAngle ?*target* (?*quagent* pos)))

    (?*quagent* turn ?turn)
    (?*quagent* run ?dist)

    (retract ?*actionState*)
    (bind ?*actionState* (assert(atTarget)))
)

;handles moving the agent to the desired target
(defrule fleeFromTarget
    (fleeTarget)
    =>
    (printout t "#Rule: fleeFromTarget" crlf)
    (printout t "#    Target = " (?*currentVision* getName ?*target*) crlf)

    (?*quagent* turn (* ((?*quagent* pos) getYaw) -1));reset yaw to 0, makes math easier

    ;flee in opposite direction +/- 30 degrees to avoid loops
    (bind ?dist  ?*fleeDistance*)
    (bind ?turn  (+ (?*currentVision* getAngle ?*target* (?*quagent* pos)) 180))
    (bind ?turn  (- ?turn (* (?*quagent* randInt 3) 30)))

    (?*quagent* turn ?turn)
    (?*quagent* run ?dist)

    (retract ?*actionState*)
    (bind ?*actionState* (assert(lookAround)))
)

;picks up wanted objects when standing over them
(defrule pickUpTarget
    (atTarget)
    =>
    (printout t "#Rule: pickUpTarget" crlf)

    (?*quagent* pickup (?*currentVision* getName ?*target*))

    (retract ?*actionState*)
    (bind ?*actionState* (assert(lookAround)))
)

;when there's nothing else to do, explore
(defrule exploreSurroundings
    (declare (salience 0))
    (checkSurroundings)
    =>
    (printout t "#Rule: exploreSurroundings" crlf)

    (?*quagent* turn  (* (?*quagent* randInt 12) 30)) ;random multiple of 30 degrees
    (?*quagent* run  (+ (* (?*quagent* randInt 6) 50) 200)) ;random multiple of 50 in
```

```
[200, 450]

    (retract ?*actionState*)
    (bind ?*actionState* (assert(lookAround)))
)



(reset)
(init)
(run)
(close)
```

Pragmatic Agent

```
(defglobal
    ?*quagent* = 0 ;mr quagent
    ?*actionState* = 0 ;state of the quagent

    ?*currentVision* = 0 ; array of objects in local area

    ?*target* = 0 ;target index into vision array
    ?*energy* = -1 ;energy level of quagent, inits to 1000

    ;constants
    ?*seekEnergyAt* = 400 ;level at which quagent will start seeking energy
    ?*fleeDistance* = 200 ;distance to back away when quagent sees something bad
    ?*searchRadius* = 1000 ;radius of probe to seek objects (max: 1000)
    ?*minDistToKrypt* = 200 ;flee at this radius
    ?*noTarget* = -1 ;when no target exsts in the vision array

    ;jess into mapper names:
    ?*mapperName* = "myMapper"
    ?*addMovement* = "acceptPos" ;(x,y)
    ?*clearViewedItems* = "resetSpecial" ;()
    ?*addViewedPoint* = "addSpecialPos" ;(x,y,"type")
    ?*setHealth* = "setHealth" ;(val)
    ?*setEnergy* = "setEnergy" ;(val)
    ?*setWealth* = "setWealth" ;(val)
)

(deffunction init ()
    (printout t "#Function: init" crlf)

    (bind ?*quagent* (new ControlledQuagent))
    (?*quagent* initialize)
    (bind ?*actionState* (assert(lookAround)))
)

(deffunction close ()
    (?*quagent* close)
)

;scan max radius for potential targets and store scan for later
;also, check/update current energy level and health
(defrule lookForTargets
    (lookAround)
    =>
    (printout t "#Rule: lookForTargets" crlf)

    ;mapper output - movement!
    ;(bind ?pos (?*quagent* pos))
    ;(printout t ?*mapperName* "." ?*addMovement* "((" (?pos getX) ", " (?pos getY) "))"
crlf)

    (bind ?*currentVision* (?*quagent* probe ?*searchRadius*))
    (bind ?*energy* ((?*quagent* getWellbeing) getEnergy))
    (bind ?health ((?*quagent* getWellbeing) getHealth))
    (bind ?wealth ((?*quagent* getWellbeing) getWealth))

    ;mapper output - radius
    ;(printout t ?*mapperName* "." ?*clearViewedItems* "()" crlf)
    ;(bind ?len (- (?*currentVision* numItems) 1))
    ;(while (>= ?len 0)
    ;    (printout t ?*mapperName* "." ?*addViewedPoint* "(" (?*currentVision* getX ?len)
", " (?*currentVision* getY ?len) ", \"" (?*currentVision* getName ?len) "\")" crlf)
```

```
    ;    (bind ?len (- ?len 1))
    ;)

    ;draw quagent's state
    ;(printout t ?*mapperName* "." ?*setEnergy* "(" ?*energy* ")" crlf)
    ;(printout t ?*mapperName* "." ?*setHealth* "(" ?health ")" crlf)
    ;(printout t ?*mapperName* "." ?*setWealth* "(" ?wealth ")" crlf)
    ;(printout t ?*mapperName* ".draw()" crlf)

    (printout t "#    Current Energy: " ?*energy* crlf)
    (printout t "#    Current Health: " ?health crlf)
    (printout t "#    Current Wealth: " ?wealth crlf)

    (retract ?*actionState*)
    (bind ?*actionState* (assert(checkSurroundings)))
)

;searches scan for kyrptonite and if found, flees from it
;search only applies when kyptonite is seen
(defrule lookForKryptonite
    (declare (salience 30))
    (checkSurroundings)
    (test (neq (?*currentVision* getClosest "kryptonite" (?*quagent* pos)
 ?*minDistToKrypt*) ?*noTarget*))
    =>
    (printout t "#Rule: lookForKryptonite" crlf)

    (bind ?*target* (?*currentVision* getClosest "kryptonite" (?*quagent* pos)))

    (retract ?*actionState*)
    (bind ?*actionState* (assert(fleeTarget)))
)

;searches scan for gold and if found, goes there
;search only applies when bot has energy above it's
;  low-threshold and gold is in the scan
(defrule lookForGold
    (declare (salience 20))
    (checkSurroundings)
    (test (neq (?*currentVision* getClosest "gold" (?*quagent* pos)) ?*noTarget*))

    ;get gold if low on energy and don't see energy ;
    (test (or (> ?*energy* ?*seekEnergyAt*) (eq (?*currentVision* getClosest "battery"
(?*quagent* pos)) ?*noTarget*)))
    =>
    (printout t "#Rule: lookForGold" crlf)

    (bind ?*target* (?*currentVision* getClosest "gold" (?*quagent* pos)))

    (retract ?*actionState*)
    (bind ?*actionState* (assert(seekTarget)))
)

;searches scan for batteries and if found, goes there
;search only applies if the agent either cannot find gold or
;  needs energy and batteries are in the scan
(defrule lookForEnergy
    (declare (salience 10))
    (checkSurroundings)
    (test (neq (?*currentVision* getClosest "battery" (?*quagent* pos)) ?*noTarget*))
    =>
    (printout t "#Rule: lookForEnergy" crlf)
```

```
    (bind ?*target* (?*currentVision* getClosest "battery" (?*quagent* pos)))

    (retract ?*actionState*)
    (bind ?*actionState* (assert(seekTarget)))
)

;handles moving the agent to the desired target
(defrule goToTarget
    (seekTarget)
    =>
    (printout t "#Rule: goToTarget" crlf)
    (printout t "#    Target = " (?*currentVision* getName ?*target*) crlf)

    (?*quagent* turn (* ((?*quagent* pos) getYaw) -1));reset yaw to 0, makes math easier

    (bind ?dist  (?*currentVision* getDist ?*target* (?*quagent* pos)))
    (bind ?turn  (?*currentVision* getAngle ?*target* (?*quagent* pos)))

    (?*quagent* turn ?turn)
    (?*quagent* run ?dist)

    (retract ?*actionState*)
    (bind ?*actionState* (assert(atTarget)))
)

;handles moving the agent to the desired target
(defrule fleeFromTarget
    (fleeTarget)
    =>
    (printout t "#Rule: fleeFromTarget" crlf)
    (printout t "#    Target = " (?*currentVision* getName ?*target*) crlf)

    (?*quagent* turn (* ((?*quagent* pos) getYaw) -1));reset yaw to 0, makes math easier

    ;flee in opposite direction +/- 30 degrees to avoid loops
    (bind ?dist  ?*fleeDistance*)
    (bind ?turn  (+ (?*currentVision* getAngle ?*target* (?*quagent* pos)) 180))
    (bind ?turn  (- ?turn (* (?*quagent* randInt 3) 30)))

    (?*quagent* turn ?turn)
    (?*quagent* run ?dist)

    (retract ?*actionState*)
    (bind ?*actionState* (assert(lookAround)))
)

;picks up wanted objects when standing over them
(defrule pickUpTarget
    (atTarget)
    =>
    (printout t "#Rule: pickUpTarget" crlf)

    (?*quagent* pickup (?*currentVision* getName ?*target*))

    (retract ?*actionState*)
    (bind ?*actionState* (assert(lookAround)))
)

;when there's nothing else to do, explore
(defrule exploreSurroundings
    (declare (salience 0))
```

```
    (checkSurroundings)
    =>
    (printout t "#Rule: exploreSurroundings" crlf)

    (?*quagent* turn  (* (?*quagent* randInt 12) 30)) ;random multiple of 30 degrees
    (?*quagent* run  (+ (* (?*quagent* randInt 6) 50) 200)) ;random multiple of 50 in
[200, 450]

    (retract ?*actionState*)
    (bind ?*actionState* (assert(lookAround)))
)




(reset)
(init)
(run)
(close)
```