# Intelligent Control System for Collection, Survival, and Mapping

Michael Silverman
Department of Computer Science
University of Rochester
msilver3@mail.rochester.edu
March 10, 2006

## Abstract

To implement intelligent control, it is necessary to use a sophisticated control system. The control algorithm for this agent uses the JESS production system to perform three tasks. The first task, collection, is achieved by a greedy move-to-target algorithm. The second task, survival, simply moves the agent away from danger. The third task, mapping, is achieved using a zigzag algorithm in which the agent moves as far as it can see in a given state, and then rotates. Two methods of mapping, the zigzag and random walk are compared. The three tasks are controlled by a hierarchical algorithm that ranks the tasks in order of importance. The agent is compared to a simple random search agent. The agent controller is written in Java and interfaces with the Quagents engine and a GUI that displays the agent's mental state. The display includes all walls that have been seen, the path the agent has taken, and the locations of all seen objects, which are stored in absolute world coordinates.

# Background and Motivation

Sophisticated control systems need structure as they grow more complex. Therefore, it is necessary to formalize and organize controlling agents. JESS is a logical choice for providing this structure. It has the capabilities necessary for basic decision making. Essentially JESS allows for the declaration of rules that can be seen as logical predicates. It has tools for matching symbols it knows are true and applying various results which can change values and call functions. Since rules for decision making in JESS can be stored in files, it makes comparing and changing agents easy and efficient.

Another issue with controlling an agent is its mental state. When an agent is running, it must store a large amount of data about the world. Not only are there objects and the environment to keep track of, but actions as well. Furthermore, it is difficult to calculate the absolute position of these items because agents can only perceive information relative to themselves. To organize this information, it is necessary to create an interface that stores all relevant information on images, and displays them. The interface can take information the agent provides and convert it into absolute positional data. Also, since the interface has all the information about objects and walls, it can provide information back to the agent when it needs to remember something about the world.

To test the system, it must be used to control a complicated task. The first task, mapping, is difficult because it requires searching without any real information about the world. It is hard to find an efficient path through an unknown area. Another task is collecting objects. To collect objects, their locations must be stored and the correct commands must be executed to reach them and pick them up. Finally, avoiding danger is almost the opposite of collecting objects. The agent needs to find a dangerous object and move away from it.

There is also an implicit task of organizing these behaviors. It would be foolish, for example, to try to map an area where there is a dangerous object nearby. These tasks require complex organization, making them good choices for organization by a productions system like JESS.

# Methods

To create the agent control system, the JESS production system must be combined into the Java client for the Quagent. To combine the two, simply compile the JESS source into the Quagent client code. Since JESS is object based, all that is needed is to instantiate the JESS interpreter. Then the Quagent client can load rules into the JESS interpreter and display the results. Also, the Quagent client must be bound into the JESS rules. This is done by adding the Quagent client into JESS and making a JESS rule that finds all Quagent clients and stores them in a global JESS variable. Once that is done, JESS commands and rules can call various helper functions in the Quagent client class.

To store information about the locations of objects and walls in the Quagent world, the agent must call the "get where" command. The command causes the agent to send its global position and direction. Once the position of the agent is updated, it is possible to locate any object because all object coordinates are sent relative to the agent position. The object information is accessible through helper functions, so the JESS rules can reason about the locations of objects and walls.

Once the world is represented, JESS only needs rules to interpret agent behavior. The agents have a state, and plan variable. State is set by the Quagent client when commands finish or information about the world changes. Also, information about the dangers and objects nearby are embedded in JESS assertions, so the rules can reason about which plan the agent should execute. See the Appendix for agent rules.

To test the JESS system's robustness and the efficiency of the mapping algorithm, it is necessary to create two sets of JESS rules for mapping and compare results. Since JESS can load agent behaviors from rule files, it is possible to compare the random walk agent to the zigzag agent.

In the random agent, the algorithm for mapping is simply to move and turn a random amount and then cast several rays to find the walls. The zigzag agent is equally simple, but performs much better. The zigzag algorithm tells the agent to cast rays, and then move to the farthest ray's impact point. Then the agent rotates ten degrees and repeats. The agent winds up zigzagging across the map because it moves as far as possible, and then, due to the rotation, moves back almost to where it started.

Aside from the differences in the mapping algorithms, the two agents are basically the same. When an object is near, JESS calls the helper functions to determine the nearest object and then executes a move command to go to it, and then pick it up. When dangerous objects are near, JESS calls the same movement functions, but instead of rotating the correct amount, it rotates the correct amount plus 180, thus moving in the reverse direction from the dangerous object.

To compare the two algorithms, they were run head to head on the same complicated maze map. The maze has loops and tight passageways that could throw off the algorithms. The maze has no objects, insuring the test only compares the mapping behavior.

# Results

The zigzag algorithm did surprisingly well. As Figure 1 demonstrates, it uncovered a much higher percent of the loopTest map than did the random traversal. Also, on the square map, the zigzag traversed the space much more evenly than the random. The pattern is due to the nature of the algorithm. Zigzag ensures an even pattern because it always tries to move as far as possible. By moving long distances, zigzag gets stuck in the same region less often. Essentially, it tries to escape any container. Random traversal leaves everything to chance.

The other behaviors functioned correctly. When items are near, the agent, as demonstrated in figure 2, goes directly to them. Rather than wandering, the desire to get items overrides mapping behavior. While this function succeeds in the case demonstrated in figure 2, the agent could get stuck due to the greedy nature of the item grabbing behavior. If, for example, a wall blocked the path to an object, the agent would keep moving into the wall forever in an effort to get to the object.

The same is true for the danger avoidance procedure. Shown in figure 3, object grabbing is overridden by the avoid danger behavior and some items are left behind. The agent runs from the danger rather than take the objects near it. Also, like item grabbing, the avoidance behavior is greedy and can get stuck if the danger is near a corner. The agent would run into the corner forever trying to escape the danger.

The results show that the hierarchy works correctly, and that the agent takes intelligent courses of action. When mapping, the agent searches through the maze until it finds objects. It grabs the objects unless there is danger. Furthermore, the results show that the architecture is modular, because it is possible to interchange behavior algorithms without changing overall behavior.

Figure 1:

The left Column is the zigzag algorithm; the right is the random walk. The top row is on the square map, and the bottom row is on the loopTest map. Blue lines are the path the agent took, and black dots are walls. The green spot is the starting point.

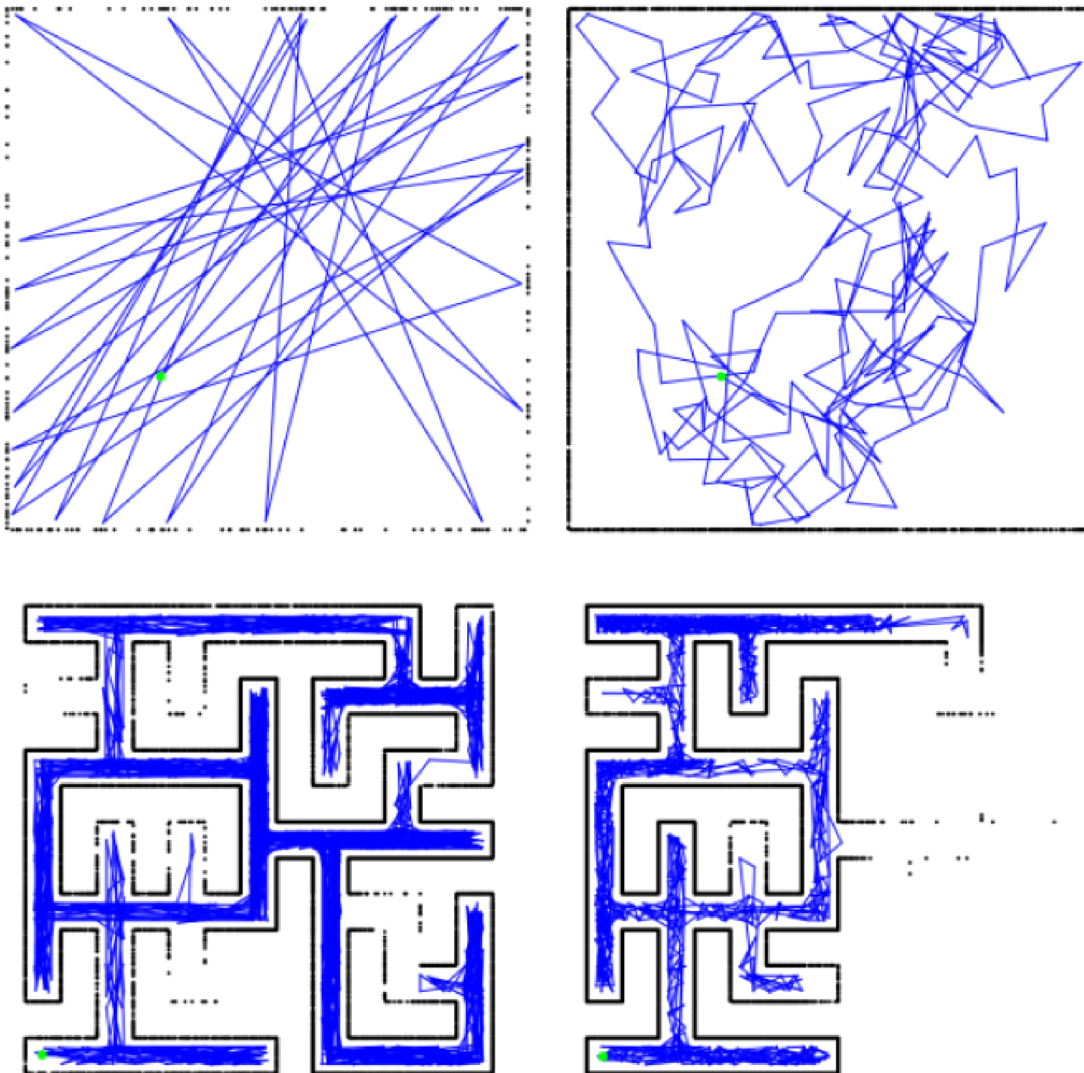

Figure 2:

This is the path and object display of the random walker run on the square map with a path of tofu. Green is the starting point, tofu is marked in red, and the agent path is blue

Figure 3:

This is the path and object display of the random walker run on the square map with a path of tofu. Green is the starting point, tofu is marked in red, and the agent path is blue. There is also a kryptonite, but otherwise this is the

# Discussion and Further Work

The agent described here completes its task, but several modifications could prove beneficial. The most obvious is another set of rules to move around walls that block the path to goals. In addition, a set of rules to prevent getting cornered while escaping kryptonite also will be beneficial. Finally, the zigzag algorithm could be enhanced to update world information more frequently. While zigzag covers a lot of space, it does not check for nearby objects as often as random walk, thus letting some objects slip through. These slight modifications could create a more powerful intelligent agent. The agent described here, however, clearly demonstrates that the JESS production system can power such tasks.

JESS provides an effective control system for complex agents. It is capable of reasoning about a hierarchy of behaviors. Furthermore, the behaviors are interchangeable, and expandable. It provides the structure necessary to create an agent that intelligently maps, grabs items, and avoids dangers. That control, combined with an internal model of the agent's world allows for a complex and robust agent capable of many intelligent tasks.

# Appendix

## JESS rules for random walker:

```
(defglobal ?*current-plan* = (assert(wander)))
(defglobal ?*current-state* = (assert(idle)))

(defglobal ?*current-objects* = (assert(noobjects)))
(defglobal ?*current-danger* = (assert(nodanger)))

(defglobal ?*dist-to-goal* = 0)
(defglobal ?*theta-to-goal* = 0)
(defglobal ?*quagent-class* = 0)

;wander rules
(defrule start-looking
        (and (wander) (idle))
        =>
        (retract ?*current-state*)
        (printout t "I'm Gonna Look Arround" crlf )
        (bind ?*current-state* (assert(looking))))

(defrule start-wandering
        (and (wander) (finishedlooking))
        =>
        (retract ?*current-state*)
        (printout t "I'm Gonna Wander" crlf )
        (call ?*quagent-class* turnBy (?*quagent-class* randF 360.0))
        (call ?*quagent-class* walkBy (+ 50 (?*quagent-class* randF 250.0)))
        (bind ?*current-state* (assert(waiting))))

;update position data rules can be done while in any plan
(defrule find-pos
        (looking)
        =>
        (printout t "Where am I?" crlf )
        (retract ?*current-state*)
        (call ?*quagent-class* doCommand "DO GetWhere")
        (bind ?*current-state* (assert(waiting))))
(defrule find-radius
        (finishedgetwhere)
        =>
        (printout t "What's near me?" crlf )
        (retract ?*current-state*)
        (call ?*quagent-class* doCommand "ASK RADIUS 400.0")
        (bind ?*current-state* (assert(waiting))))
(defrule find-rays
        (finishedradius)
        =>
        (printout t "Where are the walls?" crlf )
        (retract ?*current-state*)
        (call ?*quagent-class* doCommand "ASK RAYS 10.0")
        (bind ?*current-state* (assert(waiting))))

;avoiding cryptonite rules
;if danger is near STOP, then look arround
(defrule get-path-away
        (and (dangernear) (not(avoiddanger)))
        =>
```

```
                (printout t "OH NO not KRYPTONITE! STOP!!" crlf )
                (retract ?*current-plan*)
                (bind ?*current-plan* (assert(avoiddanger))))

(defrule false-alarm
                (and (avoiddanger) (finishedlooking) (not(dangernear)))
                =>
                (printout t "WAIT! there is no Kryptonite, it was all a dream" crlf )
                (retract ?*current-plan*)
                (retract ?*current-state*)
                (bind ?*current-plan* (assert(wander)))
                (bind ?*current-state* (assert(idle))))

;move away from danger
(defrule move-away
                (and (avoiddanger) (finishedlooking))
                =>
                (printout t "I'll turn around and move away" crlf )
                (retract ?*current-state*)
                (bind ?*theta-to-goal*
                        (+ 180 (call ?*quagent-class* thetaToObject
                                        (?*quagent-class* getNearestObject "kryptonite" ))))
                (bind ?*dist-to-goal* 300.0)
                (bind ?*current-state* (assert(gotogoal))))

;moved away from kryptonite, now back to wandering
(defrule threat-over
                (and (avoiddanger) (idle))
                =>
                (printout t "Thank Goodness I got away" crlf )
                (retract ?*current-plan*)
                (retract ?*current-state*)
                (bind ?*current-plan* (assert(wander)))
                (bind ?*current-state* (assert(idle))))

;goal seeking rules
(defrule set-target
                (and (objectsinrange) (not(avoiddanger)) (not(dangernear)))
                =>
                (retract ?*current-state*)
                (retract ?*current-plan*)
                (printout t "I'm Gonna Pick Something Up" crlf )
                (bind ?*theta-to-goal*
                        (call ?*quagent-class* thetaToObject
                                        (?*quagent-class* getNearestObject "tofu")))
                (bind ?*dist-to-goal*
                        (call ?*quagent-class* distToObject
                                        (?*quagent-class* getNearestObject "tofu")))
                (bind ?*current-plan* (assert(getobject)))
                (bind ?*current-state* (assert(gotogoal))))

(defrule pick-up
                (and (getobject) (idle))
                =>
                (retract ?*current-state*)
                (call ?*quagent-class* doCommand "DO PICKUP tofu" )
                (bind ?*current-state* (assert(pickup))))

(defrule got-object
                (and (getobject) (finishedpickup))
                =>
                (printout t "Yum, I got some tofu" crlf )
```

```
                    (retract ?*current-state*)
                    (bind ?*current-state* (assert(looking)))
                    (retract ?*current-plan*)
                    (bind ?*current-plan* (assert(wander))))

(defrule move-to-goal
                    (gotogoal)
                    =>
                    (retract ?*current-state*)
                    (printout t "I'm Gonna Go To My Target" crlf )
                    (call ?*quagent-class* turnBy ?*theta-to-goal* )
                    (call ?*quagent-class* walkBy ?*dist-to-goal*)
                    (bind ?*current-state* (assert(waiting))))
```

JESS rules for zigzag (Replace start-wandering in the random walker with the following:

```
(defrule start-wandering
                    (and (wander) (finishedlooking))
                    =>
                    (retract ?*current-state*)
                    (printout t "I'm gonna go as far as I can see!" crlf )
                    (call ?*quagent-class* turnBy (call ?*quagent-class* thetaToObject
                                                        (?*quagent-class* getMaxRayObject )))
                    (call ?*quagent-class* walkBy (call ?*quagent-class* distToObject
                                                        (?*quagent-class* getMaxRayObject )))
                    (bind ?*current-state* (assert(waiting))))
```

# Citations


[1] Russell S, Norvig P (1995) *Artificial Intelligence: A Modern Approach*, Prentice Hall Series in Artificial Intelligence. Englewood Cliffs, New Jersey