Assignment #2

Will Tekulve

September 19, 2019

1 Extended Ant Colony Design

The premise of my extended ant colony design involves adding a second class of ant agent, "Queen". In the original ant colony simulation, the ants were instructed to find food by randomly searching throughout the environment and eventually using pheromone hints and food was found and paths established. Each ant then brought that food back to "home", however that food had no purpose. My extended simulation would include a "Queen" ant that consumed the food brought back to home. This Queen would have the role of managing the size of the overall ant population, perhaps by increasing it's size when food is plentiful and allowing it to decrease when food is scarce. This means that the basic worker ants that are a part of the original simulation must also have an additional mechanic, death. Adding death as part of the simulation may have the side effect of allowing programmers to test different parameters within the simulation to see which variation produces the most successful ant colony.

1.1 Calculation of Lifespan

In order to add a death mechanic it is critical to define a lifespan, and in this case I will be basing my logic off of common garden ants. Basing Google results show that worker ants may live up to four years, while Queens have a significantly longer lifespan at 15 years. Due to this extended (and much longer than expected!) lifespan, I feel like it is apt to describe the passage of time within the simulation in terms of days. I will use a Gaussian distribution, assuming that $m_{worker} = 1460$ days and $\sigma_{worker} = 84$ days (approximately 3 months). I choose to ignore the potential death of a queen and instead use food supply to dictate whether or not more workers can be produced. Growth of new workers will be ignored and it will be assumed that workers are spawned as fully grown adults.

1.2 World Time

The passage of time in most simulations I am familiar with is governed by a ticking clock, where every entity or action moves one step per tick. Obviously, this methodology is not desirable when using the base Clojure simulation as all agents are operating truly independently. Instead I will create a new var containing a world time actor. This actor will send an action to all living worker ants that indicates the passage of a day and will modify the ant state accordingly such as killing it. To accommodate this, ant struct will require an additional attribute to track the number of days lived. Not all of these actions will be executed simultaneously although the speed of execution should be fast enough for any differences between ants to be negligible. We could also consider the differences in speed of execution to be "genetic outliers". The world time actor will also be responsible for cleaning up dead ants from the world vector.

1.3 The Queen

Only one Queen will exist in the world for all time. I will be ignoring any death mechanic for the Queen to limit the scope of these design changes. The Queen will consume food brought back to the base by workers and produce new workers. Production of new workers will use a Poisson distribution based on the amount of food available at home. The distribution will be organized such that bringing too much food to home will result in fewer workers being produced. Too little food, however, and the Queen will produce new workers at a slower rate which may lead to colony collapse.

2 Pseudo-code

2.1 World Time

Since the ants are available by a global var and through mapping associations in the world vector, the world time agent doesn't actually require any internal state to perform it's duties. As such, the agent will be defined similarly to the `evaporator`.

```
(def worldtimer (agent nil))
```

The `worldtimer` agent has two actions logically assigned, `tick` and `decompose`. The `tick` action will be responsible for distributing ticks to all living ants.

```
(def death-gauss-mean 1460)
(def death-gauss-stddev 84)

(defn chance-death [days-lived]
    "return true/false based on gaussian probability of death"
    false)

(defn tick
    "action performed by ant agent, will also need to change
    :alive attribute"
    [loc]
    (let [p (place loc)
          ant (:ant @p)]
        (dosync
            (alter ant assoc
                   :days-lived (inc (:days-lived 1)))
            (if (chance-death (:days-lived @ant))
                (alter ant assoc :alive false))))
    nil)

(defn tick-distribution [x]
    (when running
        (send-off *agent* #'tick-distribution))
    (dorun (map #(send-off % tick) ants))
    (. Thread (sleep tick-sleep-ms))
    nil)
```

The `decompose` action will be responsible for removing ants from the world that have died.

```
(defn decompose []
    (dorun
        (for [x (range dim) y (range dim)]
            (dosync
                (let [p (place [x y])
                      ant (:ant @p)]
                    (if (not (:alive @ant))
                        (alter p dissoc :ant))))))
    nil)

(defn decomposition [x]
    (when running
        (send-off *agent* #'decomposition))
    (decompose)
    (. Thread (sleep decompose-sleep-ms))
    nil)
```

The `use` section of the original code must have the following lines added to trigger these actions.

```
(send-off worldtimer tick-distribution)
(send-off worldtimer decomposition
```

2.2 Food Store

I'll need to centralize food storage due to my lack of experience with Clojure. This means I will be modifying the `drop-food` function to add food to a shared store if the ant is at "home". This agent can be completely removed with some smarter logic in `add-food` and `consume-food`.

```
(defstruct food-store :amount)
(def food-manager (agent nil))

(defn add-food [amount]
    (dosync
        (alter food-store :amount (inc amount))))

(defn consume-food [amount]
    (dosync
        (alter food-store :amount (dec amount))))

(defn drop-food [loc]
    "Drops␣food␣at␣current␣location.␣Must␣be␣called␣in␣a
␣␣transaction␣that␣has␣verified␣the␣ant␣has␣food"
  (let [p (place loc)
        ant (:ant @p)]
    (cond
        (:home @p)
            (add-food (:food @p))
    :else
        (alter p assoc
               :food (inc (:food @p))))
    (alter p assoc
           :ant (dissoc ant :food))
    loc))
```

2.3 The Queen

Similarly to `worldtimer` we need to define a singleton actor for the queen.

```
(def queen (agent nil))
```

Much like the ant agents, I am choosing to define a `queen-behave` function that takes care of managing the queen agent.

```
(def food-for-worker 75)

;this will need to be a location in the home that is always
;empty
(def queen-spawn-loc [NaN NaN])

(defn behave-queen [queen]
    (. Thread (sleep ant-sleep-ms))
    (when running send-off *agent* #'behave-queen)
    (if (> (stored-food-count) food-for-worker)
        (consume-food food-for-worker)
        (let [loc queen-spawn-loc]
            (create-ant loc 0))))
```

2.4 Ant Behavior

A minor change needs to be made to the ant `behave` function and struct to support the previous behaviors.

```
;in the behave function, the ''when" conditional must be
;modified to the following
(when (and running (:alive @ant))
       (send-off *agent* #'behave))
```

3 Shared Mutable Data Structures

A single shared data structure was added in my extension of the original code, the shared `food-store`. However, every access to this food store goes through transactions within the consume and add food functions. In this manner, consistency and atomicity is enforced through Clojure's built in transactions as part of the STM in the language.