CPSC 565 Term Project Development Guide

Neil Tallim

December 2, 2008

Contents

Part I

Introduction

1 Overview

What I am proposing as a term project is a system that simulates a terrarium, containing an arbitrary number of insect-like factions, loosely modeled after ant colonies, and a number of threats, which seek to eat the ant-like insects. This system will demonstrate the following features, all of which are modular and open to reimplementation and reimagination in the future to create a more accurate and interesting simulation (however, they are simplified right now because of time constraints):

- Agent reproduction and death
- Agent generation based on need
- Resource competition
- Survival behaviour
- Pathfinding
- Per-agent behaviour triggered by environmental hints

2 Why this project idea was developed

The goal of this project is to provide the implementor with an understanding of how the workings of independent agents can affect, and even restructure, the operations of an entire culture. Additionally, the effects of tiny changes to the environment, such as the addition of a single wall space, a change in the rate of resource replenishment, or an adjustment to the aggressiveness of a predator, will be studied to see how they cascade through the system and how all of the affected agents behave as a result.

This system will give insight into the nature of emergent patterns with multiple variables at play, via simulation of a liberal adaptation of a real-world system. The knowledge learned here will be applied to part of an extracurricular project currently being formulated.

3 How this system will work

This system will run as a 0-player game with two main active layers: cellular automata to propagate signals like pheromones, and agents that interact with objects and each other, and which derive information from the cellular automata layer. (Note, however, that each layer is comprised of multiple elements, described in 3.1 on the following page) Every iteration (tick), the current state of the field will be evaluated and transformed to create the next state. Most information will be visible to the observer (user) in the form of statistics that may be viewed by pausing the simulation and investigating the properties of any object, and a graphical field rendition will show where everything in the system is in near-real-time.

A very early conceptual example of what this system might look like in a custom engine is presented below. However, the final system will likely be implemented using Breve on a strictly 2D plane.

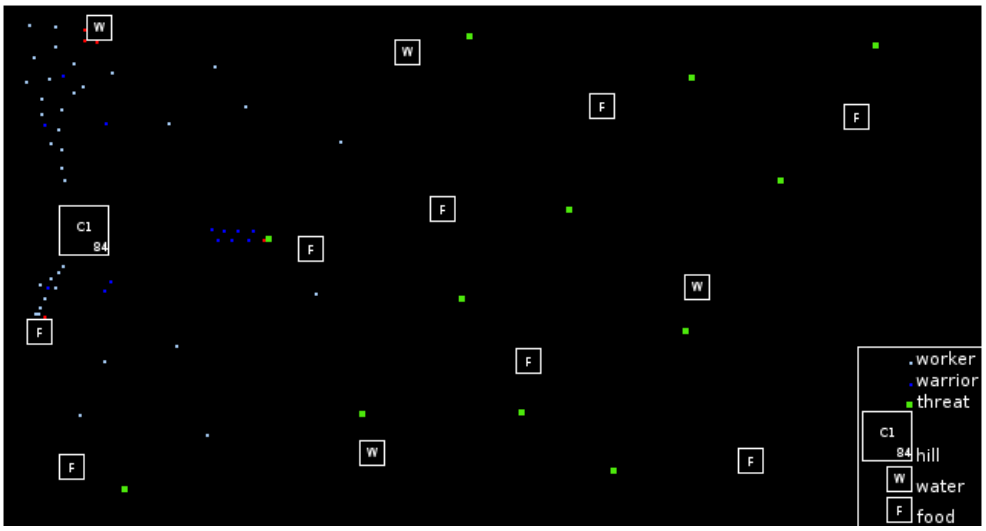

Figure 1: Early conceptual rendering of the system

3.1 How this system will work on a technical level

To prevent the "turn order" of agents from being a factor when transitioning between iterations, each element of the field – signals, entities, unts (the ant-like agents, see 4.1 on the next page), threats (see 4.3 on page 7) – will be discretely written to the new state-field in order. When writing the new state information, completed elements will be overlaid on the previous state-field (so unts will see the new signal map, but they won't see their neighbours' new positions) when updating.

All threats and all unts, as members of two isolated pools, will block on their movement events until all of their peers are ready; this will ensure that they all have fair views of the world around them before interacting with it. Lastly, threats need to move after unts so that they actually have a chance of killing them (this way, the threats will be able to move directly towards unts that may be trying to escape, giving them better attack vectors).

3.2 How observers will interact with this system

This system will load configuration data that specifies global environment variables, referenced by all agents, and per-colony/threat-class environment variables that are referenced by specific entities. Additionally, configuration data specifying the size of the field and the location of objects within it will also be fully customizable.

All scenarios will be perfectly reproducible given the same input data, so to simplify the process of revisiting interesting developments, every time the system is started, a copy of all of its configuration data will be replicated. To remove the effects of entropy, the configuration options will include a parameter for seeding the RNG.

Live interaction is not currently planned, but it may be implemented on a level that allows, for example, walls to be added or removed.

Part II

Components of the system

Note: All logic is entirely unoptimized, and serves only as a reference for thought. The code samples provided are hardly production-quality, either – in fact, they're procedural, which is definitely not how an implementation of a system like this should be written.

4 Entities and agents

As an example of emergent logic, this system will be driven by its agents. However, there will be a minor degree of "external" organization, enforced by a quasi-elective governing system (the elective attributes come from the fact that all decisions will be made based on the existence of agents, and agents spawn and die over time in reaction to their environments).

On a basic level, the mobile objects within the system are its dominant agents, and the immobile ones are entirely inert agents that merely help to shape the environment around themselves. Entities are immobile agents that influence and react to the behaviour of the mobile agents around them, providing such logical features as intuitive, though non-explicit, waypoints in pathfinding, resource transportation infrastructure, and reproduction.

Put in practical, real-life-like terms, the mobile agents are things like ants and spiders, immobile agents are things like streams (walls), and entities are like food sources and ant nests that include queens.

4.1 Unts

To cover the fact that this system will only be loosely based on the workings of ant colonies, the "protagonists" will be referred to as 'unts' (singular: unt, pronounced as 'unit').

Each class of unts that contains more than one role will have its roles assigned based on the rules in 5 on page 9.

There are several classes of unts planned for this system; these are described in the following sections.

4.1.1 Workers

Workers are unts that, for their entire lifespans, repeatedly leave hills in search of resources, which they need to harvest for the sake of the colony's survival.

Notable traits include the following:

- Most workers will be attracted to harvesting signals (some may only want food and some may only want water, depending on the needs of the colony); however, some will be repelled by the signals of others, which will ensure that the colony will always search for new sources. (See 5.1.5 on page 10)
- Workers have a boldness property, which determines how they will behave when threats are around – whether they will prioritize their own survival or act as martyrs to draw the attention of the colony's warriors.
- Workers will, when exploring for resources, never move towards the hill from which they originated until they have something to return. This will model semi-systematic behaviour and help to keep resources flowing.
- When a worker is carrying resources, it will return to the nearest accessible hill that belongs to its colony. This will help to spread unts based on regional need.
- Workers that are carrying resources will deposit pheromones on every space that they reach while heading to the nearest hill. (See 6.3 on page 15)

4.1.2 Builders

Builders are inside-workers, unts that reinforce the structure of a colony and hill. However, they exhibit no direct influence on anything in this simulation. Rather, they serve as a metric for evaluating how mature a hill is, allowing architects to be spawned (see 4.1.4 on the following page), which allows new hills to be formed.

When a new hill is formed, the builders in the hill that "reproduced" will implicitly construct tunnels to the new location and set it up for use, then change their roles to those of workers and warriors as directed by the colony's current distribution rules (see 7.1.3 on page 19) to settle it.

4.1.3 Warriors

Warriors are unts that exist solely to protect workers.

If workers get slaughtered, their colony will have fewer resources, and threatened hills will become a liability. To prevent this, warriors serve in two roles:

- Escorts, which follow any gathering-related signal pheromones they sense, keeping threats away from supply lines.
- Patrollers, which disperse throughout a colony's "territory" and deter threats and invasions before they can become a problem.

All warriors will respond to any attack signals that they perceive, no matter what role they hold. This will ensure that threats are dealt with as rapidly as possible. (However, it might also lead to weaknesses in the colony's defenses, which could possibly allow other threats to enter. The implications of this behaviour are sure to be an interesting area of study)

4.1.4 Architects

Architects are unts that have the sole task of finding an appropriate place to construct a new hill. They have above-average terrain awareness and no return-to-base logic, so they will just keep going until they find a suitable location or die trying (see 7.1.4 on page 19 for an explanation of why this suicidal behaviour is not a problem).

Unfortunately, architects are a necessary pacemaker (directing where expansion will occur, rather than letting it happen as a result of swarm consensus) due to the lack of time that can be dedicated to this project. However, they only replace one non-primary logical process, so their pacemaker effect is reasonably minimal overall.

4.2 Unt colonies and hills

Unts exist as part of a colony, which consists of one or more hills and some information that describes the classes that make up its race. In fact, that's really all there is to the definition of a colony within this system: a collection of hills; collected, shared communal resources; the archetypes used to generate new unts; information about when to spawn a new generation. Colonies have no physical presence.

Hills are actual entities that offer shelter and pathfinding targets for unts, as well as locations where new unts will appear. However, beyond being gathering points, hills exert no control over how individual unts will behave.

Colonies and hills are effectively analogous to countries that consist of a number of cities, all of which are in a state of quasi-anarchy, with a decentralised government, yet which all freely share information and resources as needed, like a pure communism. So, while the placement of hills may have a dramatic impact on how colonies will evolve, they do not affect the more primitive patterns of the ecology in which their constituents live.

4.3 Threats

Threats include anything that has the capacity to harm unts, including warrior unts from other colonies. Non-unt threats may be generically conceptualized as spiders or other predatory insects.

Each class of unt will take a different amount of time for non-unt threats to kill (unt threats attack and kill instantaneously). This is necessary to allow the defending colony's warriors time to react:

- Workers will take $ENVIRONMENT_{KILL.TIME_{WORKER}}$ iterations
- Warriors will take $ENVIRONMENT_{KILL.TIME_{WARRIOR}}$ iterations
- Architects will take $ENVIRONMENT_{KILL.TIME_{ARCHITECT}}$ iterations

After non-unt threats' lifespans have expired, depending on how many unts they consumed while alive, they will be survived by one or more of their kind:

$$children = floor(unts : consumed / ENVIRONMENT_{threat_{NOURISHMENT}}) + 1$$

Non-unt threats will not target each other.

4.3.1 Predators

Predators are simple agents that randomly wander the field, looking for unts to kill. They will run away from attack pheromones to avoid their own demise. They will attack any unt that happens to enter their sight as long as, up 'til the time of the kill, the set of all unts they can see, excluding their target victim, is free of warriors.

4.3.2 Hunters

Hunters behave just like predators, except they will emit random resource pheromones to lure workers to them. They will stop emitting pheromones upon selecting a victim, and they will only resume the process of depositing them once they have resumed wandering the field. (In terms of behavioural states (see 6.2 on page 11), they deposit pheromones only while wandering)

4.3.3 Stalkers

Stalkers behave just like predators, except they will respond to resource pheromones, which should lead them towards paths that are travelled by workers.

4.4 Obstacles

Obstacles are any objects on the field that are not hostile to unts, but which impede movement or senses in some fashion.

4.4.1 Walls

A wall is a space through which physical movement and pheromone detection are impossible. The effect of these objects is three-fold:

1. Objects on the other side of walls cannot be detected, even if they're within the agent's sight radius.
2. Signals on the other side of walls cannot be detected, even if they're within the agent's smell radius.
3. Unts will need to apply special pathfinding rules (see 6.4 on page 16) to find ways around walls, especially when heading back to their colony.

4.4.2 Sponges

A sponge does not impede physical movement, but it will absorb all pheromones that reach it. However, sight through sponges will not be affected. They are, essentially, walls that do not carry the first and third impacts.

4.5 Resources

Resources are required for a colony to survive. The primary goal of any colony is securing additional resources and growing as much as possible.

Neither food nor water is inherently more valuable than its counterpart. However, availability and consumption properties can easily change the priority of either as a simulation progresses.

4.5.1 Food

Food is one of two resources required for a colony to survive. It appears in the field as a single entity that has a maximum quantity value ($food_{CAPACITY}$), which is replenished by $food_{REPLENISHMENT} * food_{CAPACITY}$ every $food_{COOLDOWN}$ iterations.

4.5.2 Water

Water is the second of two resources required for a colony to survive. It appears in the field as a single entity that has a maximum quantity value ($water_{CAPACITY}$), which is replenished by $water_{REPLENISHMENT} * water_{CAPACITY}$ every $water_{COOLDOWN}$ iterations.

5 Role assignment

This section describes how unts receive the roles and properties that govern their behaviour in the field.

5.1 Worker roles

Workers are responsible for gathering food and water to keep their colony alive. Depending on need, the number of workers assigned to either task may be disproportionate, or workers may be told to find whatever resource they can.

Note that some workers may exhibit stochastic behaviour, which will cause them to distance themselves from known supply paths. However, their priorities will remain the same.

5.1.1 Seeking food

Food-seekers will react to food-related pheromone signals and food resources, heading towards these leads to begin harvesting so they can return what they can find to the nearest hill.

If they reach a depleted resource, they will try to find an alternative.

5.1.2 Seeking water

Water-seekers will react to water-related pheromone signals and water resources, heading towards these leads to begin harvesting so they can return what they can find to the nearest hill.

If they reach a depleted resource, they will try to find an alternative.

5.1.3 Seeking any resource

These workers will react to any resource-related pheromone signals or resource sites, bringing whatever they can find back to the nearest hill. To other unts, they will appear to be seekers of either food or water; the only difference is that these types of unts will be of much more use in establishing local patterns around new hills because they will swarm on resources much more readily.

If they reach a depleted resource, they will try to find another.

5.1.4 Role assignment

When a state transition occurs, any workers in a hill will be dispatched. At this point, each worker will individually receive a role based on whether the colony would be in danger of running out of one type of resource if all of its unts were to feed at the same time (by determining whether such an event would lead to the starvation of any unts, were it to happen twice):

- $consumption_{food} = \sum colony_{unts_{consumption:food}}$
- $consumption_{water} = \sum colony_{unts_{consumption:water}}$
- $risk_{food} = -(colony_{food:available} - (consumption_{food} * 2))$
- $risk_{water} = -(colony_{water:available} - (consumption_{water} * 2))$

```
i = unt_being_evaluated

if risk.food <= 0 and risk.water <= 0: #No risk.
 if random.random() < ENVIRONMENT.colony.WORKER.NO_FOCUS:
  i.role = None
 else:
  i.role = random.choice((ENUMERATIONS.FOOD, ENUMERATIONS.WATER))
elif risk.food > 0 and risk.water > 0: #Dire risk.
 if random.random() < risk.food / (risk.food + risk.water): #Proportional bias to the weaker supply.
  i.role = ENUMERATIONS.FOOD
 else:
  i.role = ENUMERATIONS.WATER
elif risk.food > 0: #Low food.
 if random.random() < (risk.food / consumption.food):
  i.role = ENUMERATIONS.FOOD
 else:
  i.role = random.choice((ENUMERATIONS.WATER, None))
elif risk.water > 0: #Low water.
 if random.random() < (risk.water / consumption.water):
  i.role = ENUMERATIONS.WATER
 else:
  i.role = random.choice((ENUMERATIONS.FOOD, None))
```

5.1.5 Stochastic behaviour

Each worker has a $ENVIRONMENT_{colony_{WORKER_{STOCHASTIC.PROBABILITY}}}$ chance of being a stochastic element. This property is assigned when the unt is created.

5.2 Warrior roles

Warriors are responsible for keeping a colony's workers safe, ensuring that they can gather vital resources and explore the colony's "territory" to find new resources and suitable locations for expansion.

5.2.1 Escorting workers

Escorts try to stay near supply paths by following the pheromones they sense. In this role, they will help to prevent threats from disrupting the most vital behaviour of their colony because threats will stay away if they notice that the swarm is well-protected.

5.2.2 Patrolling territory

Patrollers offer a means of guarding a colony's territory. They will spread out from the outlying hills of their colonies and disperse to shape a perimeter that will deter invasion by threats and other colonies.

5.2.3 Role assignment

Upon creation, each unt will have a $ENVIRONMENT_{colony_{WARRIOR_{ESCORT}}}$ chance of being an escort, rather than a patroller.

6 Simulation rules

These are rules that cannot be changed by configuration variables. While they may be adaptive and react according to the influences of configuration variables, they cannot be directly altered.

These rules are what govern how this simulation is unique.

6.1 Unt energy rules

Each class of unt for each colony has a certain amount of energy, like stamina, that determines how long it can survive before eating again.

Each iteration, each unt's remaining energy will decrease by one.

All workers will make returning to base their top priority when their energy has fallen to 50%. All warriors will do the same, unless they are in pursuit of a threat, since protecting workers takes precedence over their own lives. Builders, never leaving their hills, will ignore their energy level until it hits 0%.

If an unt is outside of a hill (or it is a starved builder) when its energy hits 0%, it dies instantly.

6.1.1 Energy recovery

Whenever an unt leaves a hill (or whenever a builder's energy hits 0%), it eats to replenish its strength; this is done by consuming its colony's resources. The following algorithm explains how this works. Note that it is quite simple, and it allows for some fuzziness because accuracy here is not terribly important, at least relative to the overhead that would be required to deal with whether one unt eats or not.

```
i = unt_being_evaluated
restored_halves = 0

if colony.resources(ENUMERATIONS.FOOD) > 0:
 colony.resources(ENUMERATIONS.FOOD) -= \
  i.consumption(ENUMERATIONS.FOOD) * (1 - (i.remaining_energy / i.max_energy))
 restored_halves += 1
 if colony.resources(ENUMERATIONS.FOOD) < 0:
  colony.resources(ENUMERATIONS.FOOD) = 0

if colony.resources(ENUMERATIONS.WATER) > 0:
 colony.resources(ENUMERATIONS.WATER) -= \
  i.consumption(ENUMERATIONS.WATER) * (1 - (i.remaining_energy / i.max_energy))
 restored_halves += 1
 if colony.resources(ENUMERATIONS.WATER) < 0:
  colony.resources(ENUMERATIONS.WATER) = 0

i.remaining_energy += ((1 - (i.remaining_energy / i.max_energy)) / 2) * restored_halves
```

6.2 Behavioural triggers

This section describes how agents in the system will alter their behaviour based on things they detect in the field.

6.2.1 Workers

Workers operate in one of two basic modes: wandering and harvesting. When wandering, also referred to as exploring, workers will move about the field, trying not to approach the hill from which they departed. When harvesting, which is triggered by any lead that hints at the presence of resources that they seek, they will close in on the lead until they either reach the resource or lose track of it. Upon reaching a resource, they will return directly to their hill with as much of it as they can carry in tow.

To make this system interesting, some workers exhibit non-social behaviour, as stochastic elements, which causes them to actively avoid known resources in hopes of discovering new ones.

Workers also have the curious property of aggression: depending on the nature of the colony, workers may exhibit varying degrees of aggressiveness, which will dictate whether they will seek to initiate combat, directing the colony's warriors to areas where they are needed, or whether they will run to safety, keeping the number of worker deaths down at the cost of productivity.

Behavioural logic

```
i = unt_being_evaluated

#Clear unneeded avoidances.
for avoid in i.getAvoidances():
 if not i.canSense(avoid):
  i.clearAvoidance(avoid)

if i.status == ENUMERATIONS.FOLLOWING:
 if not exists(i.target):
  i.status = ENUMERATIONS.WANDERING

if i.status == ENUMERATIONS.WANDERING or (i.status == ENUMERATIONS.FOLLOWING and isSignal(i.target)):
 suicide_run = False #Used to prevent distractions while charging a threat.
 threats = [agent <- i.agentsInLoS() | isThreat(i, agent)]
 if threats:
  if i.aggression == ENUMERATIONS.BOLDNESS.AGGRESSIVE:
   i.follow(threats[0]) #Switches to following.
   i.clearAvoidances() #Make the path straight.
   suicide_run = True
  elif i.aggression == ENUMERATIONS.BOLDNESS.ASSERTIVE:
   pass #Whatever happens happens.
  elif i.aggression == ENUMERATIONS.BOLDNESS.PASSIVE:
   i.avoid(threats[0]) #Doesn't change state, but affects pathfinding.

 if not suicide_run: #Not charging a threat.
  threat_signals = [signal <- i.signalsInLoS() | signalType(signal) == ENUMERATIONS.THREAT]
  food_signals = [signal <- i.signalsInLoS() | signalType(signal) == \
   ENUMERATIONS.FOOD, -90 < abs(angleOffset(i, i.nearestHill()) - angleOffset(i, signal)) > 90]
  water_signals = [signal <- i.signalsInLoS() | signalType(signal) == \
   ENUMERATIONS.WATER, -90 < abs(angleOffset(i, i.nearestHill()) - angleOffset(i, signal)) > 90]]
  if threat_signals:
   i.avoid(strongestSignal(threat_signals)) #Doesn't change state, but affects pathfinding.
  if i.role == ENUMERATIONS.FOOD:
   if food_signals:
    if not i.isStochastic():
     i.follow(strongestSignal(food_signals)) #Switches to following.
    else:
     i.avoid(strongestSignal(food_signals)) #Affects pathfinding.
  elif i.role == ENUMERATIONS.WATER:
   if water_signals:
    if not i.isStochastic():
     i.follow(strongestSignal(water_signals)) #Switches to following.
    else:
     i.avoid(strongestSignal(water_signals)) #Affects pathfinding.
  else:
   if food_signals or water_signals:
    if not i.isStochastic():
     i.follow(strongestSignal(food_signals + water_signals)) #Switches to following.
    else:
     i.avoid(strongestSignal(food_signals + water_signals)) #Affects pathfinding.

i.move() #Update position.

if i.status == ENUMERATIONS.RETURNING: #Deposit appropriate pheremones.
 i.producePheremone(resourceType(i.carrying))
else:
 food = [resource <- i.resourcesInLoS() | resourceType(resource) == ENUMERATIONS.FOOD]
 water = [resource <- i.resourcesInLoS() | resourceType(resource) == ENUMERATIONS.WATER]

 if i.role == ENUMERATIONS.FOOD:
```

```
  if food:
   if distance(i, food[0]) == 1:
    if i.harvest(food[0]):
     i.status = ENUMERATIONS.RETURNING #Switches to returning.
     i.clearAvoidances()
    else:
     i.avoid(food[0]) #Look for another source.
     i.status = ENUMERATIONS.WANDERING #Switches to wandering.
   else:
    i.follow(food[0]) #Switches to following.
 elif i.role == ENUMERATIONS.WATER:
  if water:
   if distance(i, water[0]) == 1:
    if i.harvest(water[0]):
     i.status = ENUMERATIONS.RETURNING #Switches to returning.
     i.clearAvoidances()
    else:
     i.avoid(water[0]) #Look for another source.
     i.status = ENUMERATIONS.WANDERING #Switches to wandering.
   else:
    i.follow(water[0]) #Switches to following.
 else:
  if food or water:
   if distance(i, (food + water)[0]) == 1:
    if i.harvest((food + water)[0]):
     i.status = ENUMERATIONS.RETURNING #Switches to returning.
     i.clearAvoidances()
    else:
     i.avoid((food + water)[0]) #Look for another source.
     i.status = ENUMERATIONS.WANDERING #Switches to wandering.
   else:
    i.follow((food + water)[0]) #Switches to following.
```

6.2.2 Warriors

Any warrior will attack the closest threat within range. This applies to any warriors that are patrolling or escorting.

Threats that are active (fellow unts are attacking them or they have attacked a fellow unt) will be located based on pheromone signals and sight.

Behavioural logic

```
i = unt_being_evaluated

if i.status == ENUMERATIONS.FOLLOWING:
 if not exists(i.target):
  i.status = ENUMERATIONS.WANDERING

if i.status in (ENUMERATIONS.WANDERING, ENUMERATIONS.FOLLOWING):
 threats = [agent <- i.agentsInLoS() | isThreat(i, agent)]
 if threats:
  i.follow(threats[0]) #Switches to following.
 else: #Check pheremones.
  threats = [signal <- i.signalsInLoS() | signalType(signal) == ENUMERATIONS.THREAT]
  if threats:
   i.follow(strongestSignal(threats)) #Switches to following.
  else:
   if i.role == ENUMERATIONS.ESCORT: #Stay near worker trails.
    signals = [signal <- i.signalsInLoS() | signalType(signal) in (ENUMERATIONS.FOOD, ENUMERATIONS.WATER)]
    if signals:
     i.follow(strongestSignal(signals)) #Switches to following.

i.move() #Update position.

if i.status == ENUMERATIONS.FOLLOWING and isAgent(i.target):
 if distance(i, i.target) <= 1:
   i.attack(i.target) #May be a suicide attack to inflict damage.
   if i.isAlive(): #Won and survived.
    i.status = ENUMERATIONS.RETURNING #Go back to base and recover.
```

6.2.3 Architects

Architects just move until they find a suitable building location. This is determined by proximity to resources and distance from hills (architects can sense friendly hills $2r$ away, where r is the radius of their resource-sensing

capabilities. This will prevent two hills from focusing on the same resource).

When moving, architects always move away from their base hill.

Behavioural logic

```
i = unt_being_evaluated

i.move() #Update position.

#Make sure there are no hills nearby.
if not [hill <- i.colony.hills | distance(i, hill) < ENVIRONMENT.MIN_BUILD_DISTANCE]:
 #Check to see whether food or water is preferred and build if in range.
 food_sources = [resource <- i.resourcesInLoS() | resourceType(resource) == ENUMERATIONS.FOOD]
 water_sources = [resource <- i.resourcesInLoS() | resourceType(resource) == ENUMERATIONS.WATER]

 if i.role == ENUMERATIONS.FOOD:
  if food_sources:
   i.build() #Ends life.
 elif i.role == ENUMERATIONS.WATER:
  if water_sources:
   i.build() #Ends life.
 elif food_sources or water_sources:
  i.build() #Ends life.
```

6.2.4 Threats

When a threat is on the prowl, it will choose any available prey by applying rules similar to the unts' warrior algorithm. If it finds no prey, it will behave according to whatever is normal for its species (see 4.3 on page 7).

Escaping, target-following, and attacking threats will not exhibit pheromone-based behaviour. Note that threats that follow signals will only go towards concentration points to keep them near prey paths, but they will not sit directly on prey paths (to avoid escorting warriors).

Behavioural logic

```
i = threat_being_evaluated

if i.status == ENUMERATIONS.WANDERING:
 prey = [agent <- i.agentsInLoS() | isUnt(agent)]
 if len(prey) == 1:
  i.attack(prey[0]) #Nothing threatening nearby.
 elif len(prey) > 1:
  if not containsWarrior(prey[1:]): #Make sure there's nothing threatening.
   i.follow(prey[0]) #Sets status to following.
elif i.status == ENUMERATIONS.FOLLOWING:
 if exists(i.target):
  threats = [agent <- i.agentsInLoS() | not agent == i.target and isWarrior(agent)]
  if threats:
    i.status = ENUMERATIONS.WANDERING #Abandon the hunt.
 else:
  i.status = ENUMERATIONS.WANDERING
elif i.status == ENUMERATIONS.KILLING:
 i.progressKill() #Count down the ticks until movement is possible. Switches to retreating when done.
elif i.status == ENUMERATIONS.RETREATING: #While retreating, movement is straight.
 if signalStrength(i.location, THREAT) == 0:
  i.status = ENUMERATIONS.WANDERING #Safe enough.
 else:
  threats = [agent <- i.agentsInLoS() | isThreat(i, agent)]
  if threats: #Retreat from the closest one.
   i.rotate(angleOffset(i, threats[0]) + 180) #Run directly away from the nearest threat.

i.move() #Update location.

if i.status == ENUMERATIONS.FOLLOWING:
 if not isSignal(i.target):
  if distance(i, i.target) <= 2:
   i.kill(i.target) #Sets status to killing.
elif i.status == ENUMERATIONS.WANDERING and species(i) == Stalker:
 leads = [signals <- i.signalsInLoS() | signalType(signal) in (ENUMERATIONS.FOOD, ENUEMRATIONS.WATER)]
 if leads:
  i.follow(strongestSignal(leads)) #Sets status to following.
```

6.3 Pheromones

Pheromones, also referred to as signals, are a medium by which information about the field and its agents can travel. They offer an imprecise method of directing agents towards areas of interest.

Each space may have any quantity of any number of different pheromones; they do not interfere with one another.

Each colony produces its own set of pheromones that are meaningless to every other colony; threats produce and follow pheromones in a colony-independent manner.

Every unt has the ability to sense pheromones nearby, just as they can sense agents. An unt does not necessarily need to be in physical contact with pheromones, unless it has a sensing range of 0.

6.3.1 Dispersion

Every cycle, the pheromones in each space will weaken, and the pheromones in adjacent spaces will increase slightly, until the concentration is too weak to persist. Pheromones strength will degrade according to the following formula:

$$strength_{new} = floor(strength_{old} * ENVIRONMENT_{SIGNALS_{DISPERSION.FACTOR}})$$

At the same time, based on the old strength of the pheromone, every surrounding space in the surrounding von Neumann neighbourhood will be given a quantity of the pheromone based on the following formula:

$$strength = floor(strength_{source} * ENVIRONMENT_{SIGNALS_{SPREAD.FACTOR..NEUMANN}})$$

Lastly, every non-overlapping space in the Moore neighbourhood will be given a quantity of the pheromone based on this formula:

$$strength = floor(strength_{source} * ENVIRONMENT_{SIGNALS_{SPREAD.FACTOR.MOORE}})$$

6.3.2 Accumulation

Because pheromones have the potential to spread, they will collide in practice, and this, naturally, will cause them to become more intense. The following formula deals with this property, based on a list, *signals*, of all pheromones that hit any given space during the current cycle transition:

$$strength = floor(max(signals) + \sum[ENVIRONMENT_{SIGNALS_{COLLISION.FACTOR}} * (signals - max(signals))])$$

This means that the strongest pheromone is used as the space's base, then a fraction of every other pheromone is added to it to get reasonable results.

6.3.3 Food pheromones

Food pheromones are dropped by worker unts who are carrying food back to a hill. Every cycle, a concentration of $ENVIRONMENT_{colony_{PHEREMONES}}$ is deposited in the unt's current space.

Threats may deposit $ENVIRONMENT_{HUNTERS_{PHEREMONES}}$ each cycle, if they are hunters.

6.3.4 Water pheromones

Water pheromones are dropped by worker unts who are carrying water back to a hill. Every cycle, a concentration of $ENVIRONMENT_{colony_{PHEREMONES}}$ is deposited in the unt's current space.

Threats may deposit $ENVIRONMENT_{HUNTERS_{PHEREMONES}}$ each cycle, if they are hunters.

6.3.5 Attack pheromones

Attack pheromones are left behind whenever any unt is killed. They alert other unts to the presence of danger. $ENVIRONMENT_{colony_{PHEREMONES.ATTACK}}$ will be dropped on the space of the agent that killed the unt.

6.4 Pathfinding and movement

Pathfinding rules dictate what happens during agents' movement phases. It is here that navigation to objects is described, as well as how avoidance logic functions.

6.4.1 Following targets

The logic used to follow targets, whether agents or other entities, is quite straightforward: turn to face the goal and advance; if the target cannot be seen, just go straight, since the current direction was its last-known location; if going straight is impossible, pick a random direction no more than 90 degrees from the current heading and go that way.

Movement logic

```
i = agent_being_evaluated

if i.target in i.agentsInLoS():
 i.face(i.target)

if not i.advance(): #There's a wall that is blocking progress.
 i.rotate(random.choice((-45, 45))) #Pick a new direction.
 i.advance() #Try taking one step in the new direction before ending the turn.
```

6.4.2 Following pheromones

The logic used to follow pheromones is similar to that used to follow targets: keep going straight (since pheromones don't move, the agent does not need to realign itself at every step) until the target has been reached; if going straight is impossible (because of a wall), the strongest like pheromone in the Moore neighbourhood is followed instead.

Movement logic

```
i = agent_being_evaluated

if i.target.location == i.location: #Destination reached.
 if not i.status == ENUMERATIONS.RETURNING:
  i.status = ENUMERATIONS.WANDERING #Stop following the signal.
 else:
  i.status = ENUMERATIONS.RETURNING #If the unt was using this as a guide, go back to returning.
elif not i.advance(): #There's a wall that is blocking progress.
 i.follow(strongestSignal([signal <- i.signalsInMoore() | signalType(signal) == \
  signalType(i.target)])) #Follow the strongest close signal.
  i.advance() #Walk to the new target.
  i.status = ENUMERATIONS.WANDERING #New target reached.
```

6.4.3 Wandering randomly

When wandering, agents will go straight with a $ENVIRONMENT_{WANDER.VARIANCE}$ probability of turning 45 degrees to either side with each step. This process will continue unless a wall is hit or the agent is no longer wandering.

Movement logic

```
i = agent_being_evaluated

if random.random() < ENVIRONMENT.WANDER_VARIANCE: #Decide whether to turn.
 i.rotate(random.choice((-45, 45)))

i.advance()
```

When a wall is encountered during an advance(), agents will randomly choose a new direction in which to travel, though workers will behave slightly differently because it is not in their interest to head back towards their colony if avoiding it is at all possible.

Wall-collision logic

```
i = agent_being_evaluated

paths = [space <- i.spacesInMoore() | isOpen(space)] #Survey the area.
if isWorker(i):
 paths = [space <- paths | \
  abs(angleOffset(i, space) - angleOffset(i, i.departed_hill)) > 45] #Try to move laterally.

if paths: #Randomly choose one.
 i.rotate(angleOffset(i, random.choice(paths)))
else:
 i.rotate(180) #Go backwards.
```

6.4.4 Wandering while avoiding objects

When an agent is trying to avoid an object (always the closest one to it, at least in the early stages of implementation), it will attempt to maintain at least the same distance from it after moving that it held prior to moving – it will either move away from it, or try to move in a circular ring around it. Workers that are avoiding an object, like a threat, will scatter and become disoriented until the threat passes; this can cause major changes in supply chains – changes that have the potential to be very interesting.

Movement logic

```
i = agent_being_evaluated

paths = [space <- i.spacesInMoore() | isOpen(space), \
 abs(angleOffset(i, space) - angleOffset(i, nearest(i, i.avoid))) > 45] #Move laterally.
if paths: #Pick one to take.
 i.rotate(angleOffset(i, random.choice(paths))
else: #Impossible to get away, so try to move past the object at an angle.
 risky_paths = [space <- i.spacesInMoore() | isOpen(space), angleOffset(i, space) in (135, 225)]
 if risky_paths: #Pick one to take.
  i.rotate(angleOffset(i, random.choice(risky_paths)))
 else: #The only path open leads to the object that must be avoided.
  return #Stay put and hope a path opens.

i.advance()
```

6.4.5 Returning to the colony

When an unt starts its return trek, it will turn to face the nearest hill (don't ask how it knows where that is; it just knows). It will then proceed to move in a straight line as far as possible, stopping only if it hits a wall. Upon recovering, it continues towards the nearest hill from its new location.

Movement logic

```
i = agent_being_evaluated

if i.state == ENUMERATIONS.DETOURING:
 nearest_hill = i.locateNearestHill()
 if clearPath(i, nearest_hill): #No obstacles in the way.
  i.rotate(angleOffset(i, nearest_hill)) #Change trajectory.
 else:
  nearby_signals = [signal <- i.signalsInLoS() | signalType(signal) in (ENUMERATIONS.FOOD, ENUMERATIONS.WATER), \
   -45 <= angleOffset(i, signal) <= 45]
  if nearby_signals:
   #Other paths are nearby (in front of the unt, to avoid backtracking),
   #so use their location as a guide.
   i.follow(strongestSignal(nearby_signals))
elif i.state == ENUMERATIONS.BACKTRACKING: #Stop backtracking ASAP.
 directions = [space <- i.spacesInMoore() | isOpen(space), angleOffset(i, space) not in (0, 180)]
 if directions: #It's possible to change course, so do it.
  i.rotate(angleOffset(i, random.choice(directions)))
  i.state = ENUMERATIONS.DETOURING

i.advance()
```

When a wall is encountered during an advance(), the unt will make a decision. The outcome will determine how it tries to find a path back to its colony.

Wall-collision logic

```
fork_paths = [space <- i.spacesInMoore() | isOpen(space), -90 < angleOffset(i, space) < 90]
if fork_paths: #Try to go as straight as possible.
 i.rotate(angleOffset(i, random.choice(fork_paths)))
 i.state = ENUMERATIONS.DETOURING
else:
 directions = [space <- i.spacesInMoore() | isOpen(space), not angleOffset(i, space) == 180]
 if directions: #Go any way possible.
  i.rotate(angleOffset(i, random.choice(directions)))
  i.state = ENUMERATIONS.DETOURING
 else: #Go backwards; this is a dead end.
  i.rotate(180)
  i.state = ENUMERATIONS.BACKTRACKING
```

7 Environmental rules

This section describes rules that are entirely dependent upon configuration parameters and the net result of their interactions over time. These rules include such things as how different classes are selected during reproduction, and how new hills are established. It is here that the emergent properties of the system flourish and come together to produce complex, nigh-unpredictable results.

7.1 Hills and colonies

A colony is defined as a related collection of independently managed hills. Post-instantiation, no parent-child relationship exists between hills. To make this property sustainable, each hill will be responsible for raising generations of unts that are distributed to suit the environment that surrounds it.

7.1.1 Spawning a new generation

A colony may spawn a new generation of unts every time the tick-count $colony_{reproduction}$ reaches 0, with the quantity of new unts being decided by the following formula:

$$unts_{new} = min(surplus_{food}, surplus_{water}) * colony_{population}$$

- $surplus_{food} = (colony_{food:available} - \sum colony_{untz_{consumption:food}}) * ENVIRONMENT_{RESOURCES.RESERVE}$
- $surplus_{water} = (colony_{water:available} - \sum colony_{untz_{consumption:water}}) * ENVIRONMENT_{RESOURCES.RESERVE}$

If the number of new unts is less than $ENVIRONMENT_{GENERATION.MINIMUM}$ of the colony's current population, then $colony_{reproduction}$ will temporarily be set to $ENVIRONMENT_{REPRODUCTION.DELAY}$, to prevent a colony from imploding when it should just wane a little. Else, $colony_{reproduction} = ENVIRONMENT_{REPRODUCTION}$ in preparation for the next cycle.

7.1.2 Allocating a new generation

All spawned unts will be initially distributed among their colony's hills based on their current populations relative to the population of the colony. The following logic will be employed:

1. Each hill will be assigned a $colony_{hill_{priority}}$ score equal to the result of the following formula, with each variable computed since the last spawn:

$$priority = priority_{control} + priority_{survival} + priority_{race} + priority_{size}$$

- $priority_{control} = ENVIRONMENT_{IMPORTANCE_{EXPANSION}} * colony_{hill_{builders}}$
- $priority_{survival} = ENVIRONMENT_{IMPORTANCE_{TERRITORY}} * (colony_{hill_{workers:lost}} + colony_{hill_{warriors:lost}})$
- $priority_{race} = ENVIRONMENT_{IMPORTANCE_{RESOURCES}} * ((colony_{hill_{food:gathered}} / colony_{food:gathered}) + (colony_{hill_{water:gathered}} + colony_{water:gathered}))$
- $priority_{size} = ENVIRONMENT_{IMPORTANCE_{GROWTH}} * (colony_{population} / colony_{hill_{population}})$

2. Each hill will receive $unts_{assigned} = (colony_{hill_{priority}} / \sum colony_{hills_{priority}}) * unts_{new}$ new unts.

7.1.3 Assigning classes to a new generation

When a hill receives a new generation of unts, each one will be cast into a class depending on the needs of the hill. The following rules will be applied to determine the class breakdown, with each value being counted since the creation of the last generation (resource counts are per-return event, not per-unt):

- $killed = (colony_{hill_{workers:killed}} + colony_{hill_{warriors:killed}})$
- $available = unts_{assigned} - killed$
- $harvesting = (1 - (colony_{hill_{workers:returned.without}} / colony_{hill_{workers:returned}})) * ENVIRONMENT_{colony_{WORKER.GROWTH}}$
- $insecurity = \begin{cases} safe & \text{if } killed = 0 \\ unsafe & \text{otherwise} \end{cases}$
  - $safe = \begin{cases} 1 & \text{if } colony_{hill_{warriors}} \leq colony_{hill_{population}} * ENVIRONMENT_{colony_{WARRIOR_{POPULATION.MINIMUM}}} \\ ENVIRONMENT_{colony_{WARRIOR_{DECAY}}} & \text{otherwise} \end{cases}$
  - $unsafe = ENVIRONMENT_{colony_{WARRIOR_{GROWTH}}} + (killed / (colony_{hill_{workers}} + colony_{hill_{warriors}} + killed))$
- $target_{workers} = harvesting * colony_{hill_{workers:lastgen}}$
- $target_{warriors} = insecurity * colony_{hill_{warriors:lastgen}}$
- if $target_{workers} + target_{warriors} \leq available$:
  - $unts_{workers} = colony_{hill_{workers:killed}} + target_{workers}$
  - $unts_{warriors} = colony_{hill_{warriors:killed}} + target_{warriors}$
  - $unts_{builders} = available - unts_{workers} - unts_{warriors}$
- else:
  - $unts_{workers} = colony_{hill_{workers:killed}} + available * (target_{workers} / (target_{workers} + target_{warriors}))$
  - $unts_{warriors} = colony_{hill_{warriors:killed}} + available * (target_{warriors} / (target_{workers} + target_{warriors}))$
  - $unts_{builders} = 0$

If the number of unts being replaced due to unnatural death exceeds the number of new unts being added, the new ones will be allocated proportionally between workers and warriors, with no unts cast into other classes. The numbers cast in this case follows:

- $unts_{workers} = unts_{new} * (colony_{hill_{workers:killed}} / (colony_{hill_{workers:killed}} + colony_{hill_{warriors:killed}}))$
- $unts_{warriors} = unts_{new} * (colony_{hill_{warriors:killed}} / (colony_{hill_{workers:killed}} + colony_{hill_{warriors:killed}}))$

7.1.4 Expansion

If $colony_{hill_{unts:builders}} \geq ENVIRONMENT_{colony_{ARCHITECT_{SPAWINING.BUILDER.RATIO}}} * colony_{hill_{population}}$, then the hill will spawn an architect independently of the other units it raises; only one architect may exist per hill at any given time. Architects are not guaranteed to find a suitable building site before they expire and, in fact, may never find such a site if they come from the middle of a well-established colony.

A Configuration variables

A.1 Global environment variables ($ENVIRONMENT$)

A.1.1 General

$MIN.BUILD.DISTANCE = 100$

A.1.2 Pathfinding

$WANDER.VARIANCE = 0.1$

A.1.3 Reproduction

$GENERATION.MINIMUM = 0.25$
$IMPORTANCE_{EXPANSION} = 0.5$
$IMPORTANCE_{GROWTH} = 0.25$
$IMPORTANCE_{RESOURCES} = 1.0$
$IMPORTANCE_{TERRITORY} = 0.5$
$REPRODUCTION = 10,000$
$REPRODUCTION.DELAY = 1,000$
$RESOURCES.RESERVE = 0.15$

A.1.4 Signals

$SIGNALS_{DISPERSION.FACTOR} = 0.75$
$SIGNALS_{SPREAD.FACTOR.NEUMANN} = 0.5$
$SIGNALS_{SPREAD.FACTOR.MOORE} = 0.33$
$SIGNALS_{COLLISION.FACTOR} = 0.1$

A.2 Colony-specific environment variables ($ENVIRONMENT_{colony}$)

A.2.1 General

$PHEREMONES = 75$
$PHEREMONES.ATTACK = 100$
$LIFESPAN = ENVIRONMENT + 2,500$

A.2.2 Architects ($ARCHITECT$)

$ENERGY = 500$
$SPAWNING.BUILDER.RATIO = 0.25$
$SIGHT = 20$

A.2.3 Builders ($BUILDER$)

$CONSUMPTION_{FOOD} = 0.75$
$CONSUMPTION_{WATER} = 0.75$

A.2.4 Warriors ($WARRIOR$)

$CONSUMPTION_{FOOD} = 1.1$
$CONSUMPTION_{WATER} = 1.1$
$DECAY = 0.75$
$ENERGY = 800$
$ESCORT = 0.25$
$GROWTH = 1.10$
$POPULATION.MINIMUM = 0.1$
$SIGHT = 4$
$SMELL = 15$

A.2.5 Workers ($WORKER$)

$BOLDNESS = ENUMERATIONS_{BOLDNESS_{PASSIVE}}$
$CARRYING.CAPACITY = 10$
$CONSUMPTION_{FOOD} = 1$
$CONSUMPTION_{WATER} = 1$
$ENERGY = 750$
$GROWTH = 1.10$
$NO.FOCUS = 0.25$
$SIGHT = 3$
$SMELL = 10$
$STOCHASTIC.PROBABILITY = 0.05$

A.3 Threat-specific environment variables ($ENVIRONMENT$)

A.3.1 Universal ($THREAT$)

$KILL.TIME_{ARCHITECT} = 3$
$KILL.TIME_{WARRIOR} = 7$
$KILL.TIME_{WORKER} = 5$

A.3.2 Predators ($PREDATOR$)

$HEALTH.POINTS = 5$
$NOURISHMENT = 3$
$SIGHT = 5$
$LIFESPAN = 3,000$

A.3.3 Hunters ($HUNTERS$)

$HEALTH.POINTS = 4$

$NOURISHMENT = 3$

$PHEREMONES = 50$

$SIGHT = 4$

$LIFESPAN = 2,500$

A.3.4 Stalkers ($STALKER$)

$HEALTH.POINTS = 5$

$NOURISHMENT = 3$

$SIGHT = 3$

$SMELL = 10$

$LIFESPAN = 3,000$

B Function definitions

Note: Only functions that need to be explained will be documented here.

Note: When resolving objects in line-of-sense (LoS), the resulting list will always be enumerated in terms of increasing distance.

B.1 Global functions

int angleOffset(agent, object) : returns the number of degrees the agent would need to turn to face the object.

bool clearPath(object, object) : true if there are no walls on a straight line between the two objects..

bool containsWarrior(list<agent>) : true if the provided list contains a warrior unt.

int distance(object, object) : computes the distance between two objects. If the second object is at the same co-ordinates (x, y) as the first, this is 0; if $abs(x_1 - x_2) \leq 1$ and $abs(y_1 - y_2) \leq 1$, this is 1; else, this is $abs(x_1 - x_2) + abs(y_1 - y_2)$ with 1 subtracted if $abs(x_1 - x_2) > 0$ and $abs(y_1 - y_2) > 0$.

bool exists(agent) : true if the specified agent is still present on the field.

bool exists(signal) : true if the space occupied by the signal still contains the signal's pheromone type.

bool isAgent(object) : true if the object referenced is an instance of an agent.

bool isOpen(space) : true if the space specified is open.

bool isSignal(object) : true if the object referenced is an instance of a signal.

bool isThreat(agent, agent) : true if the second agent is a threat to the first one.

bool isUnt(agent) : true if the agent specified is an unt.

bool isWarrior(agent) : true if the agent specified is a warrior unt.

type resourceType(resource) : returns the type of the specified resources.

int signalStrength(location, type, colony=None) : returns the strength of the specified type of pheromone at the specified location. colony, if specified, serves to filter the signals evaluated.

type signalType(signal) : returns the pheromone type of the specified signal.

signal strongestSignal(list<signal>) : returns the strongest signal in the provided list.

signal weakestSignal(list<signal>) : returns the weakest signal in the provided list.

B.2 Agent functions

bool advance() : causes the agent to walk one space forward along its current heading; success is returned.

list<agent> agentsInLoS() : builds a list of all agents within the active agent's line of sight, ordered by proximity.

void attack(agent) : causes the agent to attack the target, inflicting damage or killing it, while also taking appropriate damage itself.

void avoid(object) : causes the agent to add the object to the list of things it needs to stay away from.

bool canSense(agent) : true if the specified agent is in the active agent's line of sight.

bool canSense(signal) : true if the space occupied by the signal still contains the signal's pheromone type and the space is in the active agent's line of smell.

void follow(object) : causes the agent to turn directly towards the specified object, sets the object as the agent's *target*, and sets the agent's *status* to $ENUMERATIONS_{FOLLOWING}$.

bool harvest(resource) : causes the agent to harvest the specified resource, and sets the resource type as the agent's *carrying*. If it fails because the resource is depleted, *carrying* isn't set and false is returned.

hill locateNearestHill() : locates the hill closest to the agent that belongs to the agent's colony.

void move() : initiates pathfinding logic.

void producePheremone(type) : causes the agent to deposit pheromones of the specified type at its current location.

list <resource> resourcesInLoS() : builds a list of all resources within the active agent's line of sight, ordered by proximity.

list <signal> signalsInLoS() : builds a list of all signals within the active agent's line of smell, ordered by proximity.

list <signal> signalsInMoore() : builds a list of all signals within the active agent's Moore neighbourhood, with signals returned in strictly random order.

list <space> spacesInMoore() : builds a list of all spaces within the active agent's Moore neighbourhood, with spaces returned in strictly random order.