Agriculture automation simulated in JADE

Gabriele Dellepere, S4944557
gabriele.dellepere@gmail.com

February 15, 2025

1 Details of the proposal

1.1 Full specification of your proposal

The purpose of the proposal is using Jade to perform a smart farming simulation, where agents are composed of drones, land robots and tractors. The objective would be to maximize the land productivity minimizing the costs in terms of energy and water usage, by exploiting the collaboration between different kind of agents. Regarding the abilities of each kind of agent in the simulation, they would be the following: drones possess higher mobility and their job would be spreading pesticides and gathering information for the land robots. Land robots purpose is to fix any problem that raises within a farm (e.g. if some plants need weeding), while tractors are responsible for seeding and harvesting. Agents of the same kind share information about their relative position and tasks with each other in order to avoid collisions/conflicts of interests; drones, however, also share information about the state of the field/s they examined. Tractors are a bit of an exception as they operate on a whole field at a time and require other agents to move away from it while they work.

1.2 The kind of your proposal

Creative

1.3 The range of points/difficulty of your proposal

Medium/Hard

1.4 Changes with respect to the original proposal

Drones do not spray pesticides, they only care about checking the crops. Robots are the only agent that actively share their position, every other collision avoidance is either performed through planning or through environment detection of other agents (that are seen as obstacles)

1.5 Ways corners were cut during development

in the simulation:

- Agents never need to recharge nor to grab tools needed for their tasks
- There is no difference between day and night
- Climate never changes
- Plants have very limited needs
- At the start, the soil is already prepared and seeded

2 Introduction

2.1 The Setting

The project deals with some robotic agents whose objective is to maximize the yield of crops in a farm, while avoiding excessive waste of resources/energy. In order to achieve this, they have very different tasks to complete, depending on their **role**, but at the same time they must ensure to avoid fixed obstacles and not to collide with each other, nor damage crops that are not ready to be harvested yet.

2.1.1 Types of Agents

The agents are divided into three main categories, with different capabilities and movement speeds inside the environment:

- **Drones**: They are the fastest kind of agents and can fly. Due to these two properties, their main responsibility is to continuously check the crops, determine their current state, and inform other agents about their growth and possible issues. Each drone is responsible for only one portion of the farm, which is assigned to them by one particular drone that behaves as the Master and deals with load balancing.
- **Robots**: They are a bit slower than drones, and when inside a field they prefer to move in the direction of the furrows only (to avoid risk of accidentally trampling on some crops). However, they are the main problem solvers of the scenario: they stay idle until a drone sends them the request for assistance at some crops position, then they proceed to reach it and care for the plants (if more tasks arrive at the same time, they are put into a queue). Since more requests can arrive close in time, and there can be availability of more than one robot, they implement a Proposal-oriented conversation (**FIPA** approach), to inform the drones about their estimated level of occupancy, so that the drones can send each task to the currently least occupied robot,
- **Tractors**: They are autonomous self-driving tractors, whose main job is to keep track of the growth state of each field, and, when the expected yield becomes maximum, proceed with the harvest. Their main issues are that their the slowest kind of agent and when moving they try as much as possible to avoid every field that is not ready for harvest. Also, they need to send a warning to every agent inside a field that is going to be harvested soon, so that they can avoid it in their path planning.

Every agent has an internal **Belief** of how the environment is structured, which includes fixed obstacles and what kind of terrain is associated with a given position (e.g. it could be "path" or "farmland"). This belief is constantly updated through observations, but, of course, every agent has a limited view range only.

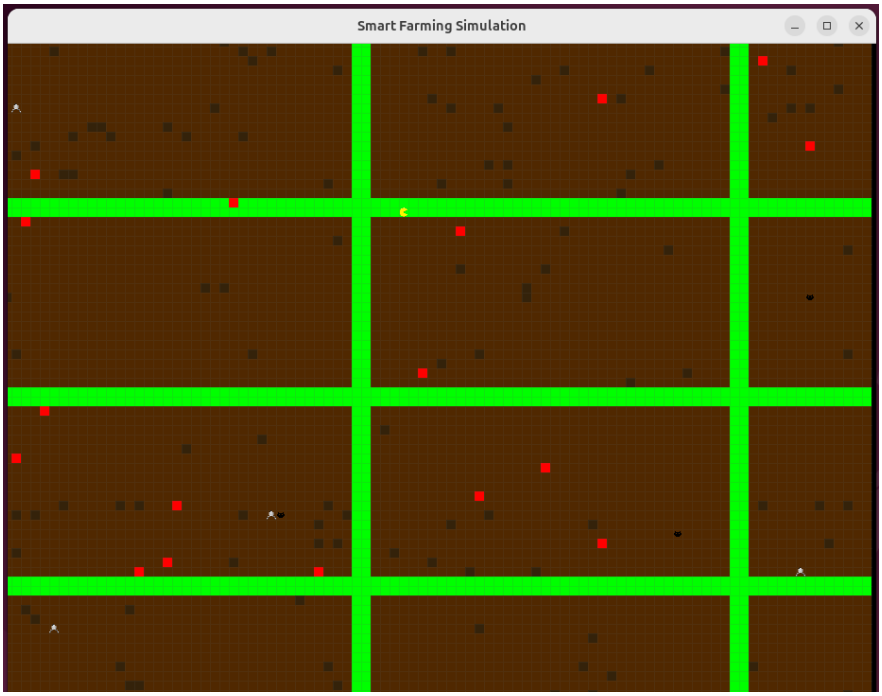
2.1.2 The Environment

The environment is structured as a bi-dimensional grid made out of **Tiles**. Each tile is one of 4 different kinds: path, farmland, obstacle and tall obstacle. While paths are intended for agents to walk freely on them, farmland is where the crops grow, and obstacles are tiles the agents cannot access. The only difference between "obstacle" and "tall obstacle" is that drones will actually fly over obstacles, but not over tall obstacles. We can think of obstacles as ponds, big rocks, gardening tools in the way and tall obstacles as trees or poles.
The crops in the farmland tile can be in either of 4 states:

- **Missing**: if they have been harvested/not replanted
- **Healthy**: if they are growing normally
- **Unwell**: if they require some intervention to return healthy

- **Dead**: if the plant is now rotten/completely dried

besides these, there are 2 additional continuous states, **growth** and **well-being**, that represent how much the plant has grown and how much its needs have been satisfied, plus a boolean hidden state, **is-decaying**, which can become true if the plant has reached maximum growth or when the well-being decreases below a certain **threshold**. A plant can go into the dead state "randomly" but only after it has entered its decay phase. During the simulation, at every tick each farmland tile is updated to simulate the crops evolution: to achieve this, the simulation uses a set of manually defined probability distributions, that while being far from providing precise results should be accurate enough to real life.



2.2 How to customize the setting

To customize the settings for the simulation, the user can define a specific set of environment variables (hence creating an env file and executing the *source* command on it is recommended). Examples of variables the user can define are:

- **SIMULATION_MAP_WIDTH** (defines the width of the grid)
- **SIMULATION_MAP_HEIGHT** (defines the height of the grid)
- **SIMULATION_DRONES_NUMBER** (defines the number of drones)
- **WINDOW_WIDTH** (defines the width of the window)
- **CROPS_HEALTHY_GROWTH_RATE** (defines the growth rate of crops when healthy)

The full list of exposed environment variables can be find in Appendix A.2.

3 Implementation

3.1 Used Tools

The project has been implemented in Java 17, using only the standard library plus two external ones: JADE and ELKI. JADE provides the framework for the agent initialization, control, and social interaction, while ELKI is a data mining library, whose only purpose inside the project is to provide the clustering algorithm implementation to perform the balancing of the workload for the drones. Because of the minimal external dependencies, the project can be compiled by simply using the correct version of the JDK and by specifying as classpath a folder containing the JADE and ELKI jar files.

3.2 Project Structure

The projects is divided into modules as shown by the following directory tree.

```
smartfarm/
|-- agents/
|-- common_behaviours/
|-- environment/
|-- models/
|-- settings/
|-- utils/
```

models contains the custom data types that are either needed for message contents (such as **AssistanceRequest**) or that are some common they needed to be placed in a shared folder (such as **Position**). *settings* contains the settings for the simulation, and *utils* contains wrapper classes for heavy mathematical or logical operations that would make the behaviours unreadable if they were placed in the middle of the other business logic.
Every main module will instead be described in the detail in the further subsections.

3.3 Creating the environment

```
environment/
|-- Environment.java
|-- EnvironmentController.java
|-- EnvironmentViewer.java
|-- ObservableEnvironment.java
|-- Observation.java
|-- ObservedEnvironment.java
|-- crops/
|   |-- Crops.java
|   |-- CropsNeeds.java
|   |-- CropsState.java
|-- tiles/
|   |-- FarmlandTile.java
|   |-- ObstacleTile.java
|   |-- PathTile.java
|   |-- TallObstacleTile.java
|   |-- Tile.java
|   |-- TileType.java
```

JADE does not provide a default implementation for environments. Because of this, the environment is a stand-alone component implemented in the project itself. It has been realized following the MVC design pattern, thus it is composed of three main classes:

- **Environment**: it handles the simulation map, by executing updates on it on demand and performing consistency checks (e.g. two agents cannot be in the same tile at the same time). It also implements the interface **ObservableEnvironment**, that is a Facade to let the agents extract observations from it and perform some actions like movement or curing/harvesting the crops.
- **EnvironmentViewer**: it provides a View of the environment state to the user. It is implemented using Java Swing and its only purpose is to show a window that displays a portion of the simulation map and the agents on it.
- **EnvironmentController**: it handles user input on the window (such as moving around the map on user drag) and also contains the "game loop" that updates the viewer and the environment at fixed intervals during the simulation.

As, anticipated, the simulation map contains tiles, which can be of the following implemented types: **PathTile**, **FarmlandTile**, **ObstacleTile**, **TallObstacleTile**. Only the FarmlandTile contains **Crops** and performs some operations inside its **update** method.
The **ObservedEnvironment** class, instead, represent the actual knowledge that an agent has about the environment (it basically constitutes its beliefs base). It would normally be learned through exploration of the environment, however due to lack of time to implement this initial step as well, every agent is already provided with an instance of a complete ObservedEnvironment at initialization.

3.4 Implementing the Agents

```
 agents/
|-- AgentType.java
|-- BaseFarmingAgent.java
|-- drone/
|-- robot/
|-- tractor/
|-- common_behaviours/
```

The agents share a common core of characteristics and functionalities, described by the **BaseFarmingAgent** class and by the behaviours in the *common_behaviours* folder. Such common features include:

- every agent has an **AgentType** (that identifies it as either Drone, Robot, or Tractor), a position on the map, a list of known agents, an reference to the ObservableEnvironment and an instance of ObservedEnvironment. Furthermore, they have a triple of field-related properties that are needed for them to plan their movements:
  - fields: the list of known farm fields
  - fieldsMap: a 2D array that maps tile position to field number
  - fieldsToAvoid: the fields that the agent must avoid (because a tractor is working on them)
- every agent has a situate() method that places them on the environment and a registerToYellowPages() method to become discoverable through the Directory Facilitator.

3.4.1 Common Behaviours

```
common_behaviours/
|-- AgentDiscoveryBehaviour.java
|-- FollowPathBehaviour.java
|-- InitBehaviour.java
|-- ReceiveHarvestNotificationBehaviour.java
```

The Common behaviours include:

- **AgentDiscoveryBehaviour**: a TickerBehaviour that once every tick sends a request to the Directory Facilitator to know if any new agent appeared or if some old agent is no longer available. In the special case of the "Master" Drone, this can trigger a new load balancing to optimize the efficiency of the available drones.
- **FollowPathBehaviour**: the most reused behaviour of the whole project, it is a TickerBehaviour, whose tickrate depends on the agent speed, and it is responsible to move the agent along a given path of consecutive tiles. If during the journey an unexpected obstacle is met, the behaviour uses the A* algorithm to avoid it and have the agent reach the next available tile on the path. A* is also used to connect any pair of tiles in the path that are not already adjacent.
- **InitBehaviour**: a OneShotBehaviour whose only purpose is to do the heavy part of the agent initialization, such as splitting the map into fields, computing the fieldsMap and adding the agent its initial set of behaviours. One thing to note is that while every agent computes the field splitting independently, the algorithm is deterministic, thus it does not require further communication to agree on the field ids.
- **ReceiveHarvestNotificationBehaviour**: a CyclicBehaviour that blocks whenever there is no message in the queue relative to fields getting harvested. Whenever a message of this kind is received, it proceeds to either add or remove the field from the fieldsToAvoid set of the agent depending on if the message notified the beginning or the ending of an harvest.

3.4.2 Drones

```
drone/
|-- DroneAgent.java
|-- behaviours/
|   |--CheckOnCropsBehaviour.java
|   |--DroneInitBehaviour.java
|   |--LoadBalancingBehaviour.java
|   |--PathPlanningBehaviour.java
|   |--RequestAssistanceBehaviour.java
```

Drones are responsible for the exploration part of the simulation; they need to check on every crop as frequently as possible, to avoid missing some needed maintenance for too long and causing the crop to decay. For this reason it is important that drones work on a partition of the simulation map that is as balanced as possible. Once a set of tiles is assigned to them, a drone saves it in its assignedTiles field and proceed to compute a plan to visit them all cyclically. To achieve this, drones require the following behaviours:

- **CheckOnCropsBehaviour**: an extension of FollowPathBehaviour, that reschedules itself upon completion to have the drone follow the planned path in a loop. Besides moving, at every tick it checks on the crops and sends an update to the tractors about the observed growth state and well-being. If the crop is in Unwell state, it also schedules a RequestAssistanceBehaviour.
- **DroneInitBehaviour**: an extension of InitBehaviour that ensures also PathPlanningBehaviour and CheckOnCropsBehaviour are added to the behaviour pool of the agent
- **LoadBalancingBehaviour**: a OneShotBehaviour responsible for splitting the workload for the drones and communicating it to them. The algorithm used is a modified version of KMeans++ that ensures that the clusters have almost equal size. It is implemented in the ELKI library and the project expose it through a wrapper function in the **BalancedKMeans** class, in the *utils* folder.

- **PathPlanningBehaviour**: a CyclicBehaviour that listens for updates from the Master Drone and updates the path plan upon notification. The algorithm used to plan the path is to simply visit one whole field at a time, and for the tiles belonging to the same field, sort them and visit them in the sorted order. The sorting with respect to the x axis swaps after at each row, while the sorting with respect to the y axis swaps after each field, to minimize as much as possible passing through the same tile twice, while ensuring that every farmland tile is visited. non adjacent tiles are connected using the A* algorithm.

- **RequestAssistanceBehaviour**: a Behaviour that is responsible of handling the conversation with the Robots, to choose the best candidate to solve the problem. While the notification for the tractor is a single non blocking message and can be handled directly in the CheckOnCropsBehaviour, in this case we need to send a Call For Proposal first, then wait for the robots to respond with either a Propose or a Refuse, and lastly select the robot with an Accept message. This behaviour has a final state that waits for the notification of completion or failure: in case of failure, the behaviour restarts from step one with the last contacted robot added to a temporary blacklist

3.4.3 Robots

```
robot/
|-- RobotAgent.java
|-- behaviours/
|   |--AcceptTaskBehaviour.java
|   |--CompleteTaskBehaviour.java
|   |--IdleBehaviour.java
|   |--MoveOutOfFieldsBehaviour.java
|   |--OfferAssistanceBehaviour.java
|   |--RobotInitBehaviour.java
```

Robots are needed to solve the issues related to the crops, but they can be a bit clumsy with the movement, so instead of exploring on their own they wait in an idle state until some drone tells them what to do. Once they receive some tasks, they complete them in order of assignment: for this reason, they need to store them in a queue. The robots also expose a "currentState" field, needed to manually restart them from the idle state (the awaken caused by message arrivals may not be enough). The list of behaviours that can be assigned to a robot is as follows:

- **AcceptTaskBehaviour**: a CyclicBehaviour that listens for "Accept" messages from the drones and either schedules a CompleteTaskBehaviour if the Robot has no tasks assigned or pushes the new task at the end of the queue otherwise. This behaviour also handles the computation of the path to be followed to reach the task destination, by taking as start position the expected position of the robot after their last task completes (this is needed to keep track of the occupancy level of the robot).

- **CompleteTaskBehaviour**: an extension of FollowPathBehaviour that, upon reaching destination, has the robot complete the task and then notify the requester drone that the plant has been provided for. If it fails to reach destination, a notification of failure is sent instead. After finishing, it schedules a new CompleteTaskBehaviour if the task queue is not empty, and a MoveOutOfFieldsBehaviour otherwise.

- **IdleBehaviour**: a Behaviour needed to avoid keeping a thread occupied with busy waiting while the robot has no tasks to complete. Its action just checks if the queue is not empty (in which case it schedules a CompleteTaskBehaviour and sets its done property to true), otherwise it blocks and waits to be restarted.

- **MoveOutOfFieldsBehaviour**: it should have been a third possible state for the robot, extending FollowPathBehaviour to move the agent out of fields when necessary (in order not to disturb their peers), but there was not enough time to develop it and test it.

- **OfferAssistanceBehaviour**: a CyclicBehaviour that listens for "Call for Proposal" messages from the drones and answers with a Propose, containing the estimated completion time of the requested task. Such estimated completion time is computed as the sum of the path lenghts for all the tasks the robot already needs to complete, plus the L1 distance of the last task destination from the position where the assistance is required.
- **RobotInitBehaviour**: an extension of InitBehaviour that ensures OfferAssistanceBehaviour, AcceptTaskBehaviour and IdleBehaviour are added as well to the behaviour pool of the agent.

3.4.4 Tractors

```
tractor/
|-- TractorAgent.java
|-- behaviours/
|   |--EvaluateFieldsBehaviour.java
|   |--HarvestCropsBehaviour.java
|   |--ReceiveMeasurementBehaviour.java
|   |--TractorInitBehaviour.java
|   |--UpdatePredictionsBehaviour.java
```

In this simulation tractors are responsible of only handling the harvesting of crops (there was not enough time to implement the land preparation and seeding part as well). The job of the tractors however is not just to physically harvest the fields, but to also determine when is the best time to do so. To explain how this is achieved, a few premises need to be made:

- A tractor can only harvest whole fields (but fields maximum size can be adjusted in the settings; we could even say that a field is defined by how little can a tractor harvest in one go)
- We assume the probability distributions to be known, or rather, we assume the probability distributions we define in the settings to be accurate/realistic: a more realistic simulation will lead to more realistic results also in terms of yield prediction
- The harvesting decision and the prediction accuracy are influenced by two further settings: **rewardPredictionSize**, how far in the future the prediction extends to, and **statesBinsAmount**, the amount of bins we use to discretize the continuous space of the expected yield.
- Right now the simulation can handle only one tractor. This is a real pity but there was no time left to implement a load balancing for the tractors as well, and since the harvest decision algorithm is deterministic, different tractors would end up trying to harvest the same field.

The harvest decision algorithm proceeds in this way:
- At every environment update, compute the expected reward predictions of each tile.
- Compute the expected reward predictions of a field as the sum of the expected reward of the tiles belonging to it.
- Schedule for harvesting the fields whose expected reward is currently at its expected maximum.

The prediction of the expected reward for each tile is not a simple procedure, and there is definitely room for improvement in the algorithm. Nevertheless, it is fully explained in appendix A.1.
Regarding the list of behaviours that can be associated to this kind of agent, their description is the following:

- **EvaluateFieldsBehaviour**: a OneShotBehaviour that is scheduled only upon demand. For every field, it scans the list of #rewardPredictionSize future expected rewards and if it determines that the maximum expected reward is at the current time step it adds the field number in the queue of fields to be harvested, and schedules an HarvestCropsBehaviour if the tractor is not doing anything.

- **HarvestCropsBehaviour**: an extension of FollowPathBehaviour where the tractor executes the harvest() method on the environment for the tiles belonging to the field currently being harvested. When it stops, if the queue of the fields to be harvested is not empty, it schedules a new HarvestCropsBehaviour.

- **ReceiveMeasurementBehaviour**: a CyclicBehaviour that waits for drones to communicate updates on the growth and well being of the crops. Upon receiving a notification, it re-estimates the expected rewards for the measured tile and replaces their contribution to the total field rewards with the updated ones. If no evalutation for the field is scheduled, this behaviour then proceeds with scheduling one.

- **TractorInitBehaviour**: an extension of InitBehaviour that ensures ReceiveMeasurementBehaviour and UpdatePredictionsBehaviour are added as well to the behaviour pool of the agent. It also initializes the expected rewards predictions using the distributions computed in the **Distributions** class in the *utils* folder.

- **UpdatePredictionBehaviour**: a TickerBehaviour whose purpose is to keep predictions aligned with the "real" simulation time. It behaves like an internal clock for the agent and when it determines that a tick is passed in the environment it polls all the internal queues the tractor is using for predicting the rewards (so that the expected value at the next tick becomes the expected value at the current tick). After an update a new EvaluateFieldsBehaviour is scheduled to determine if the current time is the appropriate one to harvest some of the fields.

4 Results

Because of the maximum tick speed of TickerBehaviour (1000 times per second) a realistic simulation can be run only up to 1000x speedup. The issue was, even with that speedup, every simulation would take more than two hours to complete, an amount of time not exactly available at this point of development. To mitigate the problem, most tests were run at 5000x speedup, taking into account that agents would be five times slower. To get more balanced results, half the tests were performed by reducing the chance of problems arising in the plants (with respect to the default value), but we must acknowledge applying that such a tweak made the simulation a little less realistic.
Mainly because of hardware limitations, the map used during the simulations was 60x60 tiles.

# drones	# robots	# tractors	crops nerf	avg. yield	max yield
4	6	1	0.2	2713.4	2800
4	6	1	0.5	2257.2	2800
4	6	1	1.0 (no nerf)	0.0	2800
4	12	1	1.0 (no nerf)	1420.5	2800

Table 1: Simulation results for a 60x60 map and 5000x speedup

We can notice that when the plants problems are reduced to one fifth, then the five times slower agents are able to maintain almost all the crops healthy without any issue. What is surprising, is that while raising the problem generation to one half only brought down the average yield by 20%, restoring it to the default values caused the yield to drop to 0.0. This is likely due to the agents prioritizing the tasks in order of arrival and nothing else: if they can't keep up, they will still try to handle every crop and cause them to die rather than focusing only on a sub-set of the fields.
As you can see, there was an attempt at solving the problem by increasing the number of robots (that were noticeably the type of agent that could not keep up the most), but just doubling the number of agents did not produce results equivalent to halving the crops problems: the main reason for this, as you may guess, is that with a small map (60x60) having that amount of agents introduces new delays due to them having to constantly avoid each other.
After these tests, a single run of the simulation with 1000x speedup was launched to have an idea of a realistic result, even though one simulation cannot really value as scientific proof. The results were the following:

# drones	# robots	# tractors	crops nerf	yield	max yield
4	6	1	1.0 (no nerf)	2601.3	2800

92.9% efficiency could be considered a success, but in the end whether this kind of approach is worth it or not mostly depends on how expensive it would be to actually deploy these kind of agents in real life.

5 Future Work

There is a lot of work that can be done to develop this project further. Here are listed a set of possible improvements that could help make the simulation more accurate to real life or make the agents act smarter:

- Introduce realistic features to the simulation, reducing the number of points in Section 1.5.
- Complete the MoveOutOfFieldsBehaviour, to ensure robots do not become idle in the middle of a field, obstructing the work of drones and tractors.
- Change the queue of the robots tasks so that tasks with close destinations are ensured to be scheduled one after the other: this is not trivial as this is a travelling salesman problem,

but maybe with some proper heuristics a partially greedy approach may provide better results.

- Implement load balancing for tractors as well; so that it makes sense to have more than one tractor (right now they compete with each other for the same fields which is pointless).
- Make it 3D.

References

[1] Erich Schubert, Arthur Zimek, o.c.: Same-size k-means variation, https://elki-project.github.io/tutorial/same-size_k_means

[2] Humoud, T., Alrushoud, M., Almutairi, F., Almutairi, W., Almarri, F., Kanj, H.: Smart agriculture monitoring and controlling system using multi-agents model. In: 2023 5th International Conference on Bio-engineering for Smart Technologies (BioSMART). pp. 1–4 (2023). https://doi.org/10.1109/BioSMART58455.2023.10162030

[3] Oracle: Trail: Creating a GUI With Swings, https://docs.oracle.com/javase/tutorial/uiswing/index.htm

[4] Shi, H., Zhang, R., Sun, G., Chen, J.: Clustering-based task coordination to search and rescue teamwork of multiple agents. International Journal of Advanced Robotic Systems **16**(2), 1729881419831154 (2019). https://doi.org/10.1177/1729881419831154, https://doi.org/10.1177/1729881419831154

[5] Telecom Italia: Jade programming for beginners (2009), https://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf

[6] Wikipedia: A* search algorithm, https://en.wikipedia.org/wiki/A*_search_algorithm

[2] [4] [1] [3] [5] [6]

A Appendices

A.1 Expected Reward Prediction

Here is presented how the expected reward for each farmland tile is computed in the project:

(legend)
$R_t = reward \quad at \quad time \quad t$
$G_t = growth \quad at \quad time \quad t$
$W_t = wellbeing \quad at \quad time \quad t$
$S_t = state \quad at \quad time \quad t$
$Thr = threshold \quad before \quad decay$
$H = healthy$
$U = unwell$
$D = dead$

At every time step t, the reward for harvesting a tile is 1.0 if the crop has matured (its growth is >= 1.0) and if its status is $\neq$ Dead, otherwise it is 0.0. Because of this, the expected reward can be computed as:

$$E[R_t] = p(G_t >= 1.0, S_t \neq D) = p(G_t >= 1.0 | S_t \neq D) \cdot p(S_t \neq D)$$

let's start with handling $p(S_t \neq D)$: to be dead at time t, a crop must either already have been dead at time $t-1$, or it must be in its decay phase and it can enter this state with probability $pDying$. This means we can give this probability a recursive definition, as such:

$$p(S_t \neq D) = p(S_{t-1} \neq D) \cdot (1.0 - pDying \cdot p(decay_t = true | S_{t-1} \neq D))$$

Sidenote: from this moment onwards, the pre-condition $S_{t-1} \neq D$ can be assumed for any probability computation, thus it has been removed from the formulas to make them more readable, however we must remember that it is there.
To be in a decaying state, the crop must either have its well being W_t below Thr, or being mature and landing in this phase with probability $pDecay$. So we can rewrite the above equation as:

$$p(S_t \neq D) = p(S_{t-1} \neq D) \cdot (1 - pDying \cdot p(W_t < Thr \quad or \quad (G_t >= 1.0, goes \quad in \quad decay)))$$

or

$$p(S_t \neq D) = p(S_{t-1} \neq D) \cdot (1 - pDying \cdot (1 - (1 - p(W_t < Thr)) \cdot (1 - p(G_t >= 1.0) \cdot pDecay))$$

By plugging in this probability in the expected reward equation, we can see that the expected reward only depends on $p(G_t >= 1.0)$ and $p(W_t < Thr)$ (the other probabilities are taken from fixed distributions that we set in the settings of the simulation) and can be computed recursively from t_0 to our desired time step t, as long as the distance $t - t_0$ is less than our rewardPredictionSize.
Now, the only issue is how to to get $p(G_t >= 1.0)$ and $p(W_t < Thr)$.

The proposed solution is to try to approximate the distributions $p(G_t)$ and $p(W_t)$ between 0 and 1, using #statesBinsAmount discretized bins, and see both $p(G_t >= 1.0)$ and $p(W_t < Thr)$ as Cumulative distribution functions. Since to compute the expected reward we're already performing a recursive computation from t_0, we can update $p(G_t)$ and $p(W_t)$ as well during the same recursion, by performing a convolution between the previous growth and well being state with their expected growth/decrease rates.

$p(G_t) = p(G_{t-1}) * growthRate$
$p(W_t) = p(W_{t-1}) * wellBeingDecreaseRate$

$growthRate = healthyGrowthRate \cdot p(S_t = H) + unwellGrowthRate \cdot p(S_t = U)$
$wellBeingDecreaseRate = unwellDecreaseRate \cdot p(S_t = U)$ // no decay if healthy

Fortunately, we are assuming $S_{t-1} \neq D$. This means that we can approximate the evolution process of S_t with a Markovian one, where we take into account only the transitions between the states H and U. It also means we can say $p(S_t = H) \approx 1 - p(S_t = U)$.
This brings us to the final piece of our probability distribution computation, updating $p(S_t = H)$ and $p(S_t = U)$ in our recursive process as:

$p(S_t = H) = p(S_{t-1} = H) \cdot (1 - p(S_t = U | S_{t-1} = H)) + p(St - 1 = U) \cdot p(S_t = H | S_{t-1} = U)$
$p(S_t = U) = 1 - p(S_t = H)$

Where $p(S_t = U | S_{t-1} = H))$ is the probability that the crop develops a problem at that particular time step (given the probabilities of each problem defined in the settings, it can be computed as 1− the joint probability of not getting any of them at the same time).
$p(S_t = H | S_{t-1} = U)$, instead, is the probability a plant will be treated by a robot if it is unwell. No amount of domain expertise can make us pick a perfectly accurate value for this probability at the first try: the suggested approach is to pick an initial value that is not

completely unreasonable (e.g. sampling it between 0.1 and 0.01) and running the simulation several times, updating the estimate with a Monte Carlo approach.

Implementation Notes: We discussed that the probability distributions $p(G_t)$ and $p(W_t)$ are computed recursively. What we must point out now, however, is that the probabilities at time t_0 are impulses (zero everywhere and one at their known initial values, 0.0 and 1.0 respectively). Furthermore, whenever a new measurement is received, we find ourselves in a situation similar to the one at t_0: the current probability distribution has once again become an impulse that is one somewhere and zero everywhere else. Because of this, we can see the measurement update as just resetting the probability computation with an initial G or W that is simply translated some amount with respect with the original G_{t_0} or W_{t_0}. The final consideration we need to take into account is that translations are preserved across convolutions (this can be easily proven by applying a Fourier Transform and noticing that a translation has now become a change of phase among multiplications). Since we use convolutions to update our probability distributions, to update the estimates for $p(G_t)$ and $p(W_t)$ once a new measurement comes in we just need to apply a shift to the already known distributions without recomputing them from scratch.

Secondly, since we're only interested in cumulative distribution functions and since applying a shift up to #statesBinsAmount elements could be cumbersome, instead of saving the distributions $p(G_t)$ and $p(W_t)$, we use them only during the initialization phase and then keep in memory only their cdf_s, easily obtainable by just running a prefix sum and a suffix sum on the two distributions respectively.

A.2 Environment variables to customize the simulation

Here follows the list of customizable environment variables, paired with their default value (default probabilities are uniform probability distributions whose mean has been taken from eos.com and Wikipedia)

AGENTS_VIEW_RANGE = 3
AGENTS_DRONE_SPEED = 500
AGENTS_ROBOT_SPEED = 250
AGENTS_TRACTOR_SPEED = 120
AGENTS_ROBOT_WATERING_DELAY = 8
AGENTS_ROBOT_WEEDING_DELAY = 12
AGENTS_TRACTOR_CLOCK_DELAY = 100
AGENTS_AGENT_DISCOVERY_INTERVAL = 4000
AGENTS_REWARD_PREDICTION_SIZE = 500
AGENTS_STATES_BINS_AMOUNT = 1000
AGENTS_IDEAL_FIELD_SIZE = 1000
MINIMUM_REWARD_PERCENTAGE = 0.1
MEASURED_AVERAGE_COMPLETION_TIME = 30.0
CROPS_DYING_CHANCE = 3e-6
CROPS_DECAY_CHANCE = 5e-6
CROPS_WELL_BEING_THRESHOLD = 0.4
CROPS_WELL_BEING_DECREASE = 2.0e-5
CROPS_HEALTHY_GROWTH_RATE = 1.9e-5
CROPS_UNWELL_GROWTH_RATE = 1.4e-5
CROPS_WATERING_NEED = 1.6e-4
CROPS_WEED_REMOVAL_NEED = 1.6e-4
SIMULATION_MAP_WIDTH = 60
SIMULATION_MAP_HEIGHT = 60
SIMULATION_TARGET_FPS = 80
SIMULATION_TARGET_UPS = 10

SIMULATION_MOUSE_SENSITIVITY = 1.0
SIMULATION_DRONES_NUMBER = 4
SIMULATION_ROBOTS_NUMBER = 6
SIMULATION_TRACTORS_NUMBER = 1
SIMULATION_GROWTH_CHEAT = false
WINDOW_WIDTH = 500
WINDOW_HEIGHT = 500
WINDOW_GRID_SIZE = 32
WINDOW_MIN_TILE_SIZE = 10
WINDOW_MAX_TILE_SIZE = 30