UNIVERSITY OF BARI ALDO MORO

DEPARTMENT OF INFORMATION TECHNOLOGY

Master's Degree in Computer Science

Fundamentals of Artificial Intelligence

Case study

GreMaES

a Prolog-based Greenhouse Management Expert System

Francesco Peragine

f.peragine@studenti.uniba.it

m. 737873

https://github.com/francescoperagine/GreMaES

A.Y. 2021/22

Sommario

Introduction

The present case study illustrates the development details of a Prolog-based *Expert System* (ES) within the domain of the ambient intelligence, in the context of the management of a greenhouse.

The plant domain has been inevitably oversimplified to be addressed in both size and complexity, and several assumptions also take place, like those relative to light (intensity, duration, spectrum), water (composition, hardness, pH) and so on.

The ES can be use in three ways: **interacting** by submitting queries, by **monitoring** the greenhouse itself and **consulting** the knowledge base.

Interacting

The user is guided by an interactive shell interface to submit the symptoms that are manifested on a plant, to obtain a diagnosis of its health problems. The user may provide any number of captions.

For the sake of explainability, at the end of the reasoning process the system asks the user to consult the inference steps that led to the actual results.

Monitoring

To ensure safety and the correct lifecycle of hosted plants, the agent carries out the following actions:

- continuous monitoring by the means of sensor devices
- storing environment percepts
- identification of conditions that may affect the plants based on stored environment percepts
- managing the environment thanks to actuator devices
- showing reports of the hosted plants' overall status through the Command Line Interface
- log observations and actuators activity in external files in the **logs** folder.

Sensors devices provides the inputs: readings for temperature and humidity, image captions for textual descriptions of any abnormal manifestation occurred on plants. All readings are simulated through simple random sampling strategy performed with timed cadence. At the end of every batch the user is asked if he wishes to continue with the sampling. To smooth the process the sampling size is fixed to 10 and the interval is fixed to 0.

Problems that have been identified and that can be directly addressed through actuator devices are automatically fixed, like a low humidity level can be fixed by activating the watering system.

At the end of the process, a detailed report is shown.

Consulting

The user can explore rules and treatments stored in the knowledge base.

Inference

The implementation of the ES is based on both forward and backward chaining.

Even if not necessary, it was my intent to explore both reasoning processes, mostly to deepen my understanding and satisfy my curiosity.

The forward reasoning strategy is implemented as first step of the reasoning process, to induce from basic manifestations of symptoms the health conditions, which is the foundation of the **monitor mode**, whilst the backward strategy is used mostly in the **user mode** in which, starting from the output of the forward reasoning process, the most general health issue is diagnosed.

Plant domain

The knowledge base used in this case study has been made from public resources available on the Internet. Being the product of an autonomous research conducted without any prior knowledge of the domain itself and not being verified by any domain expert, it may contain errors and omissions.

All the gathered informations are listed in the Glossary – Knowledge Base section.

Requirements

The software was developed with YAP (Yet Another Prolog) v.6.2.2.

Some SWI Prolog libraries were also included.

Domain

Due to the simplifications that take place in this case study, there is no clear correspondence with literature about the terminology, therefore I defined *health issues*, *health problems* and *health conditions* it as follows:

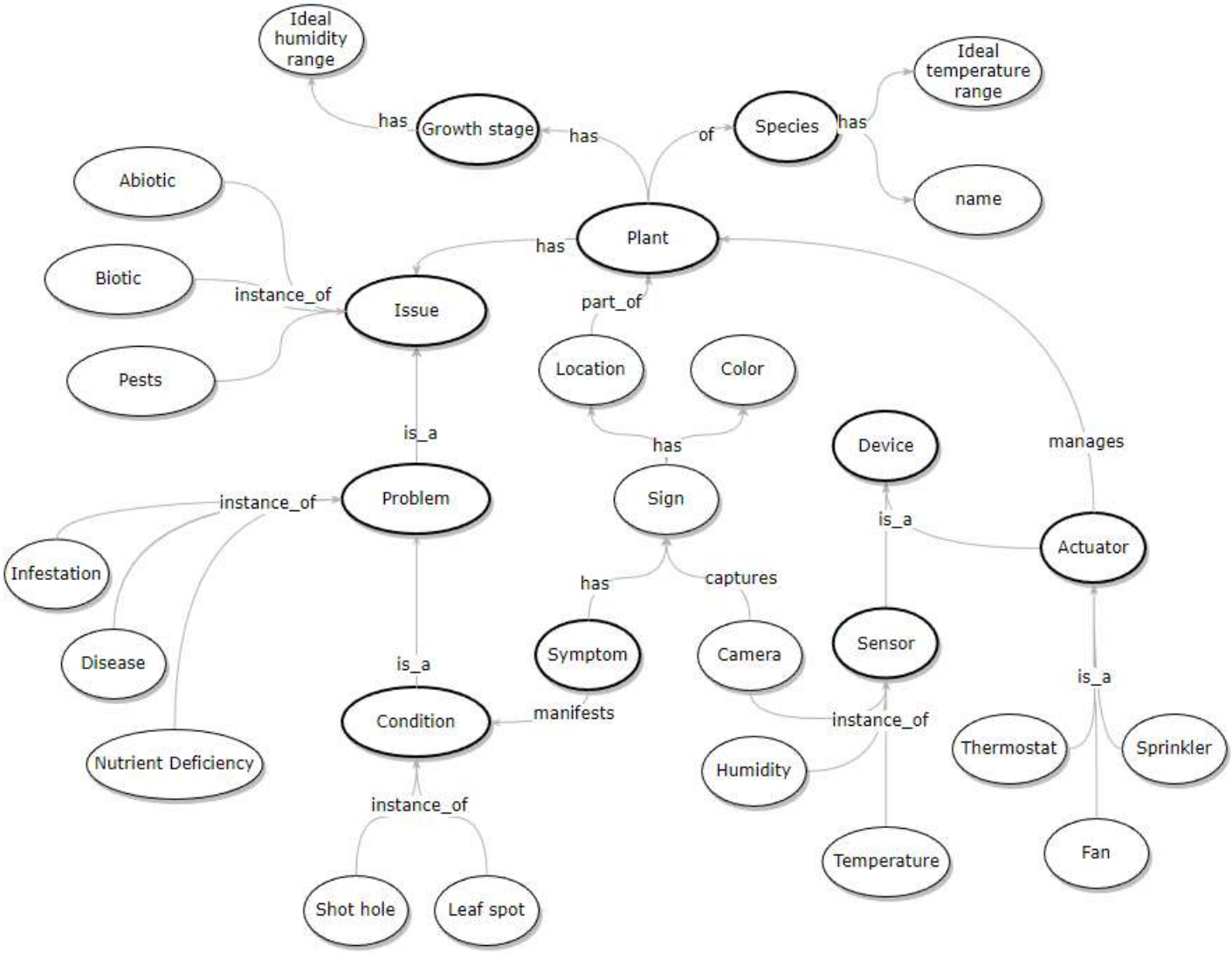
Issues are families of problems that may affects the plants. There are two main categories:

- Abiotic disorders caused by nonliving factors (climate, nutrient deficiencies, overwatering, ...)
- Biotic diseases caused by living organisms (diseases, pests).

Every *problem* has several *conditions,* which has one or more *symptoms* and may have multiple *treatments*, if any.

Symptoms are manifested through *signs* on *sections* of the plants and may show peculiar *colors*.

Ontology



Conceptualization

Domain: Plants

Goal: diagnosis about the health status of a plant.

Analysis

- Each **plant** is characterized by:
  - name
  - species
  - growth stage
  - health status
  - set of connected sensor devices
  - set of connected actuator devices
- **Species** have:
  - name
  - ideal temperature range
- **Growth stage:**
  - identifier
  - ideal humidity range
- **Sensor devices** acquire percepts from the environment
  - identifier
  - metric (temperature, humidity)
- **Actuator devices** alter the properties of the environment:
  - identifier
  - metric
  - activation status
- **Symptoms** are physical manifestations of health problems, are denoted by:
  - sign
  - section (location)
  - color

Predicates

- **plant**(Plant,Species,GrowthStage) = Plant is of Species and has GrowthStage
    - Constants: {trinidad1, sunflower, p1, p2, …}
- **species**(Name,TemperatureMin,TemperatureMax) = species has ideal range of temperature
    - Constants: {capsicum_chinense, …,
      -5, …, 40}
- **growth_humidity**(GrowthStage,HumidityMin,HumidityMax) = stage S has ideal range of humidity [X - Y]
    - Constants: {flowering_mature, vegetative_growing, seed_germination,
      …,
      0 - 100}
- **sign_location**(Sign,Location) = Sign occurs on Location
    - Constants: {none, altered_color, angular_lesions, black_leathery_spot, …
      all, branches, leaves, lower_leaves, roots, … }
- **sign_color**(Sign,Color) = Sign has Color
    - Constants: {altered_color, cotton_like_downy_substance, flies, …
      chlorotic, dark_green, blotchy_chlorosis, interveinal_chlorosis, …}
- **treatment**(Condition,Treatment) = Condition may be solved with Treatment
    - Constants: {'spray with neem oil and insecticidal soap', 'cut off and remove the infected leaves or flowers', 'shower the plant once a week', …}
- **plant_sensor**(Plant,SensorID,Metric) = Plant is connected to SensorID that acquires environment property Metric
    - Constants: {t11, t12, t13, t14, …
      temperature, humidity, caption}
- **plant_actuator** (Plant,ActuatorID,Metric,Activation) = Plant is connected to ActuatorID that manages environment property Metric with Activation status
    - Constants: {act1, act2, act3, …
      cold, hot, wet, dry
      thermostat, fan, sprinkler}

Rules

Conditions are characterized by a limited set of symptoms (up to three within the case study), which are matched with the provided inputs, by users or sensors, to reason about the health status of plants.

Main

The program starts by initializing the index, saving in the working memory the most used lists and removing old logs. After that it welcomes the user, cleaning up the working memory and asking which mode is to be executed.

The choices represented are between the diagnostic *user mode*, the *knowledge base mode* and the automatic *monitor mode*.

The selection ensures the loading of the respective module and its initialization.

Once the selected task is completed, the user is asked if he wants to run the program again.

init

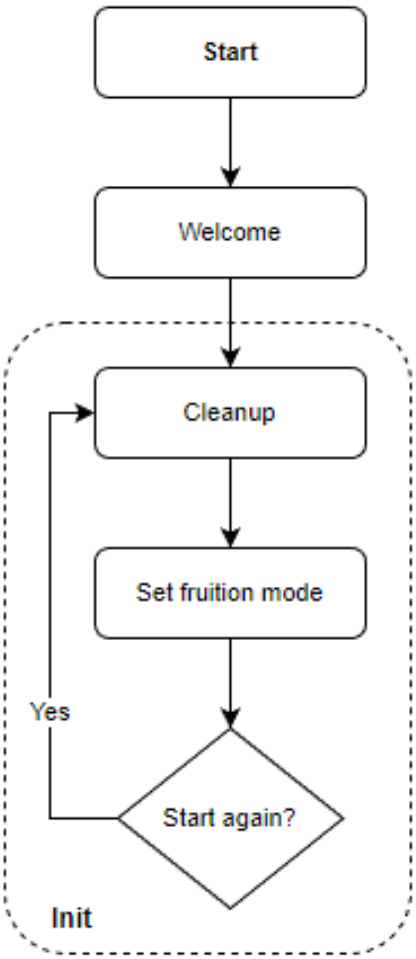
```
:- initialization(init).

init :-
    engine_init,
    utils_init,
    logger_init.
```

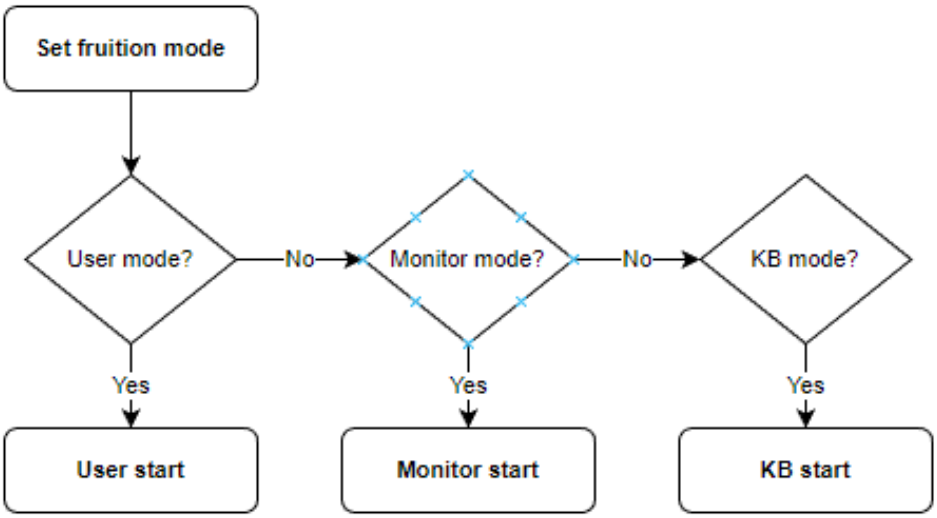
start

```
start :-
    cleanup,
    welcome,
    fruition_mode,
    restart.
```

cleanup

```
cleanup :-
    unset(asked/2),
    unset(fact_history/2),
    unset(fact/2),
    unset(usedfact/2),
    unset(actuator_status/2),
    unset(observation/6).
```





set_fruition_mode

```
fruition_mode :-
    mode_user,
    ensure_loaded(mode_user),
    user_start.
fruition_mode :-
    \+ (mode_user),
    mode_monitor,
    ensure_loaded(mode_monitor),
    monitor_start.
fruition_mode :-
    \+ (mode_user),
    \+ (mode_monitor),
    mode_kb,
    ensure_loaded(mode_kb),
    kb_start.
```

```
mode_user :- askif(fruition_mode(mode_user)).
mode_monitor :- askif(fruition_mode(mode_monitor)).
mode_kb :- askif(fruition_mode(mode_kb)).
```

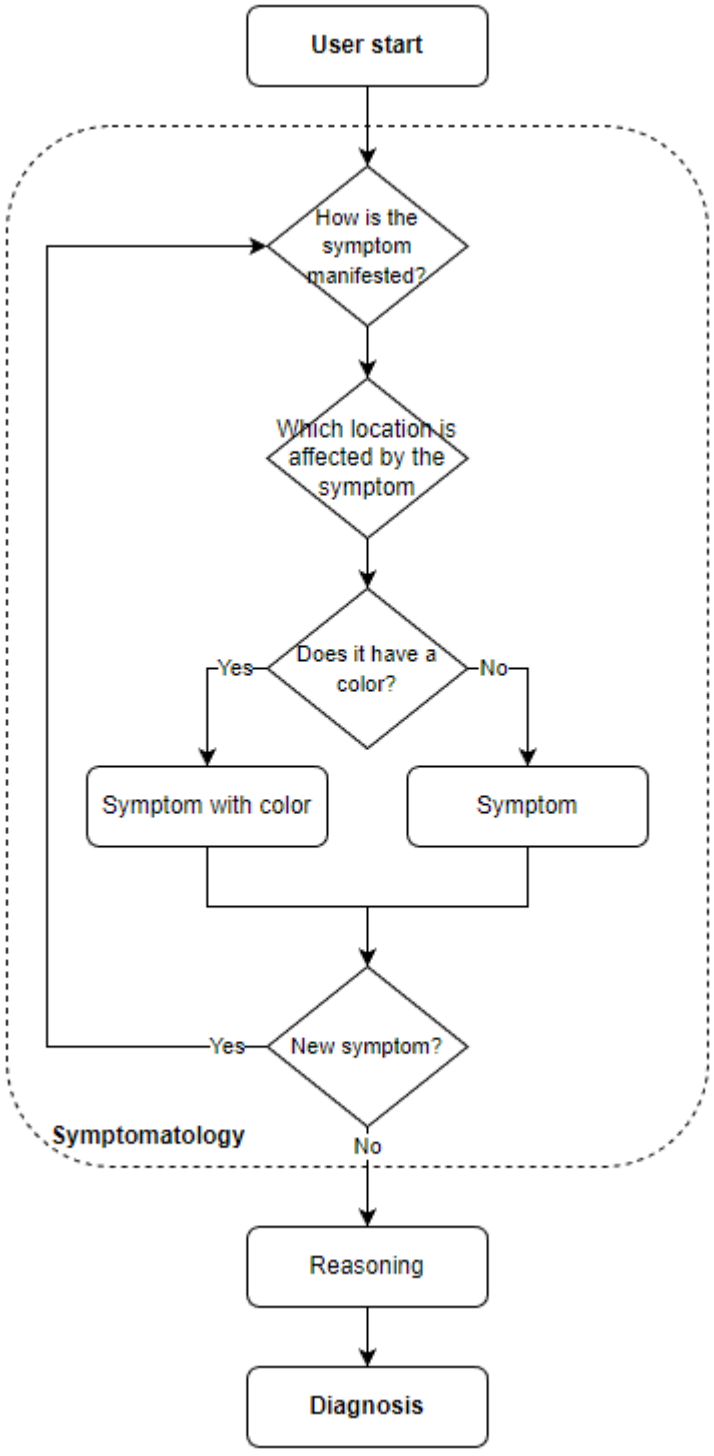
restart

```
restart :-
    start_again,
    cleanup,
    start.
restart :-
    \+ start_again,
    goodbye.
start_again :- askif(start_again).
```

User mode

user_start

```
user_start :-
    symptomatology,
    forward,
    backward,
    diagnosis.
```



The user mode starts by asking how the issue has been manifested and of which color it was, if any.

The **sign** is build by asking questions about its localization and color.

To reduce the search (and not overwhelm the user), every choice that is provided to the user will show only the combinations of signs, locations and color present in real symptoms. If a list of possible elements is reduced to one, the value is selected without further prompts.

Once the symptom has been saved, the user is asked to register more.

The diagnostic process is performed at the end of the gathering phase.

symptomatology

Asks repeatedly the user to provide symptoms, until the user specifies otherwise.

```
symptomatology :-
    repeat,
    build_symptom,
    !,
    not(once_again).

once_again :- askif(new_symptom).
```

build_symptom

Initializes the questioning for the user to provide informations about the manifested symptoms.

```
    unset_asked(new_symptom),
    ask_sign(Sign),
    ask_location(Sign,Location),
    ask_color(Sign,Color),
    save_observation(curr,symptom(Location,Sign,Color)).
```

Reads the previously stored signs' list and initializes the menu for the selection from the list of elements. Signs are independent from each other, whereas locations and colors are always dependent on signs. All signs are manifested on a location, but not all the signs show a peculiar color. If the options' lists are reduced to only one element, it's selected by default.

ask_sign (-Sign)

```
ask_sign(Sign) :-
    signs(Signs),
    ask_menu(signs,Signs,Sign).
```

ask_location (+Sign,-Location)

```
ask_location(Sign,Location) :-
    all(Location,sign_location(Sign,Location),Locations),
    length(Locations,L),
    (L > 1 -> ask_menu(sign_locations,Locations,Location) ;
nth1(1,Locations,Location)).
```

ask_color (+Sign,-Color)

```
ask_color(Sign,Color) :-
    all(Color,sign_color(Sign,Color),Colors),
    length(Colors,L),
    (L > 1 -> ask_menu(sign_colors,Colors,Color) ; nth1(1,Colors,Color)).
% ask_color (+Sign,-Color)
ask_color(Sign,none) :-
    \+ sign_color(Sign,_).
```

After the user provided the sign, the KB is consulted to check whether that sign is associated to a color. If so, the user is asked to also provide it.

After that, the user is asked to select the section on which the sign occurs.

ask_menu (+MessageCode,+Menu,-Selection

Displays a menu and get user's selection.

```
ask_menu(MessageCode,Menu,Selection) :-
    writeln_message(MessageCode),
    display_menu(Menu,1),
    repeat,
    read(Index),
    ask_menu_forward(Menu,Index,Selection).
```

ask_menu_forward (+Menu,+Index,-Selection)

Returns Index's Selection.

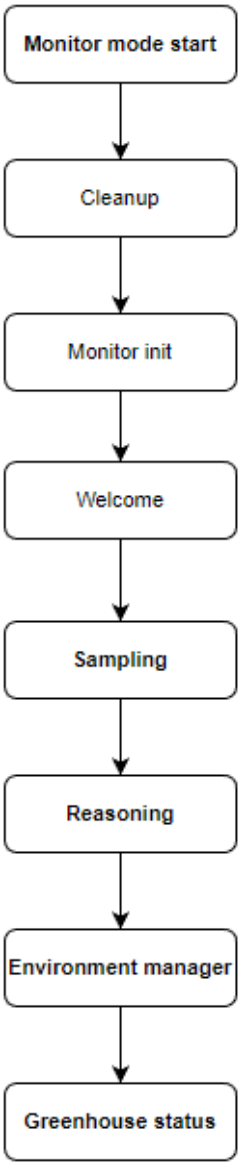
```
ask_menu_forward(Menu,Index,Selection) :-
    nth1(Index,Menu,Selection).
ask_menu_forward(Menu,Index,Selection) :-
    \+ nth1(Index,Menu,Selection),
    writeln_message(not_recognized_value),
    !,
    fail
```

display_menu (+List,+Index)

Helper predicate to display the menu options with their indexes.

```
display_menu([],_).
display_menu([Option|Rest],Index) :-
    write(Index),write('. '),write(Option),nl,
    NewIndex is Index + 1,
    display_menu(Rest,NewIndex).
```

Monitor mode



As for the user mode, the monitor mode starts by welcoming the user and cleaning up the cached informations.

It then acquires the ideal values for the environment temperature and humidity for every hosted plant in the greenhouse, printing them out for the users, and setting all connected actuators to the *off* status.

monitor_init

Initializes sensors' list and actuators.

```
monitor_init :-
    sensors_init,
    actuators_init.
```

sensors_init

Saves a list of all sensors to speed up the sampling.

```
sensors_init :-
    all(SensorID,plant_sensor(_,SensorID,_),Sensors),
    saveln(sensors(Sensors)).
```

actuators_init

Sets all connected actuators to the off status.

```
actuators_init :-
    all(ActuatorID,plant_actuator(_,ActuatorID,_,_),Actuators),
    forall(member(ActuatorID,Actuators),actuator_init(ActuatorID)),
    writeln('Actuators initialized.').
```

monitor_start

```
monitor_start :-
    monitor_cleanup,
    monitor_init,
    welcome_monitor,
    greenhouse_mapping,
    monitor_loop_start,
    !,
    forward,
    backward,
    environment_manager,nl.
    greenhouse_status.
```

monitor_cleanup

```
monitor_cleanup :-
    unset(observation/6),
    unset(sensors/1).
    unset(actuator_status/2).
```

greenhouse_mapping

```
greenhouse_mapping :-
    all(Plant-Species-'temp_range'-Tmin-Tmax-'stage'-GrowthStage-Hmin-
Hmax,  (plant(Plant,Species,GrowthStage),species(Species,Tmin,Tmax),growth_humidity
(GrowthStage,Hmin,Hmax)),
        Plants),
    sort(Plants,SortedPlants),
    maplist(writeln,SortedPlants),nl.
```

To make the system as realistic and as customizable as possible few inputs parameters can be modified:

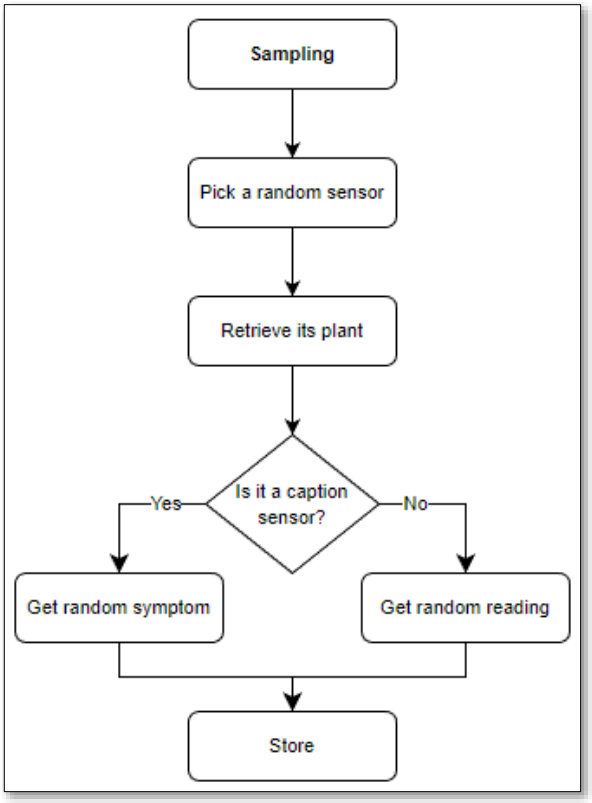
- the size of each batch of sampling
- the number of seconds between each sampling
- the probability of acquiring a sign from a camera sensor
- the variability of each metric reading

```
sampling_size(10).
sampling_interval(0).
reading_variability(1.3).
symptom_probability(1).
```

monitor_loop_start

Deletes the last confirmation about continuing the loop and initializes the loop with specified size and interval.

```
monitor_loop_start :-
    unset_asked(continue_monitor_loop),
    sampling_size(Repetitions),
    sampling_interval(Interval),
    monitor_loop(Repetitions,Interval).
```



monitor_loop (+Repetitions,+Interval)

Performs a fixed number of timed samplings.

```
monitor_loop(Repetitions,Interval) :-
    Repetitions > 0,
    sampling_init,
    (Interval > 0 -> sleep(Interval) ;
true),
    RepetitionsNew is Repetitions - 1,
    monitor_loop(RepetitionsNew,Interval).
```

At the end of the loop the user is asked to continue the sampling.

```
monitor_loop(0,_) :-
    continue_sampling,
    monitor_loop_start.
```

The loop ends with a negative answer from the user.

```
monitor_loop(0,_) :-
    \+ continue_sampling.
% continue_sampling/0
continue_sampling :-
    askif(continue_monitor_loop).
```

sampling_init

Selects a random sensor from those already active on plants and initializes the sampling which includes a timestamp.

```
sampling_init :-
    sensors(Sensors),
    random_list_element(Sensors,RandomSensorID),
    plant_sensor(Plant,RandomSensorID,SensorType),
    timestamp_utc(Timestamp), % evaluated here to fix the timestamp
    sampling(Plant,Timestamp,SensorType,RandomSensorID).
```

random_list_element (+List,-Element)

Randomly selects an element from a list

```
random_list_element(List,Element) :-
    length(List,Length),
    random(0,Length,RandomNumber),
    nth0(RandomNumber,List,Element).
```

sampling (+Plant,+Timestamp,+SensorType,+SensorID)

Observations are stored in both working memory and logs.

Performs a sampling of an environment metric randomly selecting values in a extended range.

```
sampling(Plant,Timestamp,SensorType,SensorID) :-
    SensorType \= caption,
    reading_variability(Variability),
    plant_range_values(Plant,SensorType,Min,Max),
    MinV is Min / Variability,
    MaxV is Max * Variability,
    % humidity values are fixed in a [0-100] range.
    % not proportionally scaled to exploit the reading_variability.
    (SensorType = humidity,MinV < 0 -> MinV1 = 0 ; MinV1 = MinV),
    (SensorType = humidity,MaxV > 100 -> MaxV1 = 100 ; MaxV1 = MaxV),
    random(MinV1,MaxV1,RandomValue),
    Value is floor(RandomValue),
    reading_status(Plant,SensorType,Value,ReadingStatus),
    save_observation(Plant,reading(SensorType,ReadingStatus)),
    saveln(observation(Timestamp,Plant,SensorType,SensorID,Value,ReadingStatus)),
    atomic_concat([Timestamp,',',Plant,',',SensorType,',',SensorID,',',Value,',',Re
adingStatus],Message),
    log(observation,Message).
```

Performs a random sampling from a camera device that returns a symptom.

```
sampling(Plant,Timestamp,caption,SensorID) :-
    symptom_probability(SymptomProbability),
    random(RandomValue),
    sampling_symptom(SymptomProbability,RandomValue,Symptom),
    save_observation(Plant,Symptom),
    term_to_atom(Symptom,S),
    atomic_concat([Timestamp,',',Plant,',',caption,',',SensorID,',',S],Message),
    log(observation,Message).
```

plant_range_values (+Plant,+SensorType,-Min,-Max)

Returs environment metric's SensorType range values

```
plant_range_values(Plant,temperature,Min,Max) :-
    plant(Plant,Species,_),
    species(Species,Min,Max).
plant_range_values(Plant,humidity,Min,Max) :-
    plant(Plant,_,GrowthStage),
    growth_humidity(GrowthStage,Min,Max).
```

reading_status (+Plant,+SensorType,+Value,-ReadingStatus)

Reads the range values of the SensorType and returns the status of the current reading.

```
reading_status(Plant,SensorType,Value,ReadingStatus) :-
    plant_range_values(Plant,SensorType,Min,Max),
    Value < Min,
    ReadingStatus = low.
reading_status(Plant,SensorType,Value,ReadingStatus) :-
    plant_range_values(Plant,SensorType,Min,Max),
    Value > Max,
    ReadingStatus = high.
reading_status(Plant,SensorType,Value,ReadingStatus) :-
    plant_range_values(Plant,SensorType,Min,Max),
    between(Min,Max,Value),
    ReadingStatus = normal.
```

sampling_symptom (+SymptomProbability,+RandomValue,-Symptom)

Performs the sampling of a symptom based on the probability that it happens.

This is to simulate an actual environment and eventually to perform stress tests of the system.

For the case study purpose, the probability of a symptom happening is 100% to test out the inference system when multiple symptoms are acquired for each plant.

```
sampling_symptom(SymptomProbability,RandomValue,Symptom) :-
    SymptomProbability >= RandomValue,
    signs(Signs),
    random_list_element(Signs,Sign),
    caption_forward(Sign,Symptom).
```

If no symptom is acquired, the neutral symptom is returned.

```
sampling_symptom(SymptomProbability,RandomValue,symptom(none,all)) :-
    RandomValue > SymptomProbability.
```

caption_forward (+Sign,-Symptom)

Matches Location and Color with a random Sign.

```
caption_forward(Sign,symptom(RandomLocation,Sign,RandomColor)) :-
```

```
    random_sign_location(Sign,RandomLocation),
    random_sign_color(Sign,RandomColor).
```

For those symptoms which do not have any associated color, sets the 3rd argument to none.

```
caption_forward(Sign,symptom(RandomLocation,Sign,none)) :-
    \+ sign_color(Sign,_),
    random_sign_location(Sign,RandomLocation).
```

random_sign_location (+Sign,-RandomLocation)

Unifies Sign with a random location related to it.

```
random_sign_location(Sign,RandomLocation) :-
    all(Location,sign_location(Sign,Location),Locations),
    random_list_element(Locations,RandomLocation).
```

random_sign_color (+Sign,-RandomColor)

Unifies Sign with a random color related to it.

```
random_sign_color(Sign,RandomColor) :-
    all(Color,sign_color(Sign,Color),Colors),
    random_list_element(Colors,RandomColor).
```

monitor_diagnosis

Retrieves all plants related conditions, removes duplicates and initializes the explanation.

```
monitor_diagnosis :-
        findall(Plant-Condition,
usedfact(_,condition(Plant,Condition)),PlantsConditionsList),
        list_to_ord_set(PlantsConditionsList,PlantsConditionsSet),
        maplist(explain_diagnosis,PlantsConditionsSet).
```

environment_manager

Initializes the retrieving of climate related conditions.

```
environment_manager :-
    plants(Plants),
    forall(member(Plant,Plants), plant_reading(Plant)).
```

plant_reading (+Plant)

Retrieves all the readings of the plant and initializes the actuator handler.

```
plant_reading(Plant) :-
    findall(reading(Timestamp,Plant,Metric,Condition,Reading),
        (observation(Timestamp,Plant,Metric,_,_,Reading),
        rule(_,condition(Plant,Condition),[manifests(Plant,reading(Metric,_))])),
        List),
    actuator_handler(List).
```

actuator_handler (+List)

Retrieves actuators able to handle the environment Metric and starts them.

```
actuator_handler([]).
actuator_handler([reading(Timestamp,Plant,Metric,Condition,Reading)|T]) :-
    findall((Timestamp,Plant,ActuatorID,Metric,Condition,Reading,ActivationStatus),
            (plant_actuator(Plant,ActuatorID,Metric,ActivationStatus)),
            Actuators),
    (Actuators \= [] -> maplist(actuator_start,Actuators) ; true),
    actuator_handler(T).
```

actuator_start

If the reading is not normal and if the actuator is able to handle the condition, activates it.

Logs the update only if the actual status was modified.

```
actuator_start((Timestamp,Plant,ActuatorID,Metric,Condition,Reading,ActivationStatu
s)) :-
    Reading \= normal,
    Condition = ActivationStatus,
```

```
    actuator_save(ActuatorID,on,Action),
    (Action \= none -> actuator_log(Timestamp,Plant,Metric,Reading,ActuatorID,on) ;
true).
```

The actuator is turned off if the reading is normal or if it cannot handle the condition.

```
actuator_start((Timestamp,Plant,ActuatorID,Metric,Condition,Reading,ActivationStatu
s)) :-
    actuator_save(ActuatorID,off,Action),
    (Action \= none -> actuator_log(Timestamp,Plant,Metric,Reading,ActuatorID,off)
; true).
```

actuator_log (+Timestamp,+Plant,+Metric,+Reading,+ActuatorID,+Status)

```
actuator_log(Timestamp,Plant,Metric,Reading,ActuatorID,Status) :-
    atomic_concat([Timestamp,',',Plant,',',Metric,',',Reading,',',ActuatorID,',',St
atus], Message),
    log(actuator,Message).
```

greenhouse_status

Shows a report for all plants' conditions.

```
greenhouse_status :-
    plants(Plants),
    all(Plant-Condition,
(member(Plant,Plants),usedfact(_,condition(Plant,Condition))),PlantConditions),
    maplist(writeln,PlantConditions).
```

Diagnosis

diagnosis

No symptoms case

```
diagnosis :-
    not(usedfact(_,_)),
    writeln_message(no_symptom).
```

If there are symptoms but there's no clear diagnosis

```
diagnosis :-
    usedfact(_,_),
    not(usedfact(_,condition(_,_))),
    writeln_message(no_condition).
```

If there are clear conditions,explains them

```
diagnosis :-
    usedfact(_,condition(_,_)),
    diagnosis_forward.
```

diagnosis_forward

```
diagnosis_forward :-
    history(HistoriesList),
    (member([Plant|History],HistoriesList),
    plant_history(Plant,History,Facts),
    all(Plant-Condition,(member(ID,History),
usedfact(ID,condition(Plant,Condition))),PlantsConditions),
    explain(PlantsConditions),
    explain_inference(Facts)).
```

explain_inference (+Facts)

```
explain_inference(Facts) :-
    need_explanation,
    writeln_message(inference),
    maplist(writeln,Facts).
explain_inference(_) :-
    \+ need_explanation.

% need_explanation/0
need_explanation :- askif(need_explanation).
```

explain (+List)

```
explain([]).
explain([Plant-Condition|T]) :-
    explain_diagnosis(Plant-Condition),
    explain_treatment(Condition),
    explain(T).
```

explain_diagnosis (+Plant-+Condition)

```
explain_diagnosis(Plant-Condition) :-
    rule(_,problem(_,Problem),[condition(_,Condition)]),
    rule(_,issue(_,Issue),[problem(_,Problem)]),
    usedfact(_,issue(Plant,Issue)),
    (mode_user -> X = '\nYour plant'; X = Plant),
    atomic_concat([X,' is affected by the ',Issue,' disorder - ',Condition,'
',Problem], Message),
    writeln(Message).
```

explain_treatment (+Condition)

```
explain_treatment(Condition) :-
    rule(_,problem(_,'climate'),[condition(_,Condition)]),
    writeln_message(preserve_environment).
explain_treatment(Condition) :-
    rule(_,problem(_,'nutrient deficiency'),[condition(_,Condition)]),
    writeln_message(missing_nutrient).
explain_treatment(Condition) :-
    \+ rule(_,problem(_,'climate'),[condition(_,Condition)]),
    \+ rule(_,problem(_,'nutrient deficiency'),[condition(_,Condition)]),
    \+ treatment(Condition,_),
    write_message(treatment_none), writeln(Condition).
explain_treatment(Condition) :-
    \+ rule(_,problem(_,'nutrient deficiency'),[condition(_,Condition)]),
    all(Treatment,treatment(Condition,Treatment),Treatments),
    write('* How to treat '),write(': '),
    maplist(writeln,Treatments).
```

Engine

engine_init

```
engine_init :-
    not(last_index(_)),
    all(ID,rule(ID,_,_),IDs),
    max_list(IDs,LastID),
    NextID is LastID + 1,
    save_index(NextID).
```

next_index(-NextID)

```
next_index(NextID) :-
    last_index(LastID),
    NextID is LastID + 1,
    save_index(NextID).
```

backward(+Fact)

Backward chaining uses as KB the output of the forward one, therefore starts from evaluating conditions.

```
backward(Fact) :-
    usedfact(_,Fact),
    \+ functor(Fact,manifests,_).
backward(Fact) :-
    usedfact(_,Fact),
    functor(Fact,condition,_).
```

Evaluates all rules and stores the proved facts on top of the KB then adds them to the plant history.

```
backward(Fact) :-
    rule(RuleID,Fact,[Condition]),
    backward(Condition),
    \+ usedfact(_,Fact),
    next_index(FactID),
    saveln_a(usedfact(FactID,Fact)),
    Fact =.. [_,X,_],
    save_history(X,RuleID),
    save_history(X,FactID).
```

Initializes the inference.

```
backward :-
    all(Fact,(backward(Fact)),Facts).
```

forward

```
forward :- done.
forward :-
    fact(ID,Fact),
    not(pursuit(ID,Fact)),
    save_usedfact(ID,Fact),
    forward.
done :- not(fact(_,_)).
```

pursuit (+FactID,Fact)

```
pursuit(FactID,Fact) :-
    rule(RuleID,Head,Conditions),
    functor(Head,condition,_),
    rule_pursuit(FactID,Fact,RuleID,Head,Conditions),
    fail.
```

rule_pursuit (+FactID,+Fact,+RuleID,+Head,+Conditions)

Searches through rules conditions, deleting entries that match the fact.

```
rule_pursuit(FactID,Fact,RuleID,Head,Conditions) :-
    match(Fact,Conditions),
    delete_fact(Fact,Conditions,ConditionsNew),
    new_rule(FactID,RuleID,Head,ConditionsNew).
```

delete_fact (+X,+List,-Rest)

Unifies the fact by binding the variable within the conditions and it always succedees.

```
delete_fact(X,[],[]).
delete_fact(X,[X|L],M) :- delete_fact(X,L,M).
delete_fact(X,[Y|L],[Y|M]) :-
    not(X=Y),
    delete_fact(X,L,M).
```

new_rule/4 (+FactID,+RuleID,+Head,+List)

When the right-hand sided of a rule is empty a new fact is made,otherwise the rule is updated.

```
new_rule(FactID,RuleID,Head,[]) :-
    save_fact(Head,FactIDNew),
    save_trail(FactID,[RuleID,FactIDNew]).
new_rule(FactID,RuleID,Head,Conditions) :-
    not(Conditions=[]),
    save_rule(RuleID,Head,Conditions),
    save_trail(FactID,RuleID).
new_rule(_,RuleID,Head,Conditions) :-
    fact(_,Head),
    Conditions=[].
```

Utils

```
debug(off).
```

is_debug

```
is_debug :- debug(on).
```

timer (+Goal,-Time)

Estimates the CPU time for evaluating a Goal.

```
timer(Goal, Time) :-
    statistics(runtime, [Start|_]),
    Goal,
    statistics(runtime, [End|_]),
    Time is End - Start.
```

utils_init

Initializes the most frequently used lists to improve performances and improve tracing readability.

```
utils_init :-
    signs_init,
    plants_init.
```

signs

Stores in the working memory a list of all signs for future samplings.

```
signs_init :-
    all(Sign,sign_location(Sign,_),Signs),
    delete(Signs,'none',Signs1),
    list_to_ord_set(Signs1,Set),
    saveln(signs(Set)).
```

plants_init

Stores in the working memory a list of all plants for future samplings.

```
plants_init :-
    all(Plant,plant_sensor(Plant,_,_),Plants),
    list_to_ord_set(Plants,Set),
    saveln(plants(Set)).
```

history

```
history :-
    all([P|H2],
            (fact_history(P,H1),
```

```
            usedfact(ID,manifests(P,S)),
            memberchk(ID,H1),
            reverse(H1,H2)),
        Hs),
    maplist(writeln,Hs).
```

history (-Hs)

Returns the reversed plants' histories

```
history(Hs) :-
    all([P|H2],
            (fact_history(P,H1),
            usedfact(ID,manifests(P,S)),
            memberchk(ID,H1),
            reverse(H1,H2)),
        Hs).
```

plant_history_id (+P,-H1)

Returns the reversed plant's history

```
plant_history_id(P,H1) :-
    fact_history(P,H),
    reverse(H,H1).
```

plant_history (+Plant,-History,-Facts)

Returns plant's history and facts

```
plant_history(Plant,History,Facts) :-
    plant_history_id(Plant,History),
    all(X-Fact,(
        member(X,History),
        (usedfact(X,Fact) ; (rule(X,(Head),Body),Fact = Head-Body))
        ),Facts).
```

timestamp_utc (-Now)

```
timestamp_utc(Now) :-
    datime(datime(Year, Month, Day, Hour, Minute, Second)),
    add_zero(Month, PaddedMonth),
    add_zero(Day, PaddedDay),
    add_zero(Hour, PaddedHour),
    add_zero(Minute, PaddedMinute),
    add_zero(Second, PaddedSecond),
    atomic_list_concat([Year, PaddedMonth, PaddedDay], '-', Date),
    atomic_list_concat([PaddedHour, PaddedMinute, PaddedSecond, 'Z'], ':', Time),
    atomic_list_concat([Date, 'T', Time], DateTime),
    format(atom(Now), '~w', [DateTime]).
```

add_zero

```
add_zero(N, Padded) :-
    (N < 10 -> atomic_concat([0, N], Padded) ; Padded = N).
```

write_message/1 (+MessageCode)

Retrieves the string message from its code and prints it out.

```
write_message(MessageCode) :-
    message_code(MessageCode,Message),
    write(Message).
```

writeln_message/1

```
writeln_message(MessageCode) :-
    write_message(MessageCode),nl.
```

match/2 (+X,+L)

Checks whether the X matches any element within L.

```
match(X,[X|L]).
match(X,[Y|L]) :- match(X,L).
```

match/2 (+L1,+L1)

Checks every member of L1 in L2

```
match([], _).
match([X|L1], L2) :-
    member(X, L2),
    match(L1, L2).
```

Logger

Logs are, currently, used only in the monitor_mode and are stored in the GreMaES.log file. All logs files are deleted at start.

logger_init

```
logger_init :-
    all(LogFile,(log_file(File,LogFile),
(file_exists(LogFile),delete_file(LogFile))),LogFiles).
```

log (+File,+Message)

```
log(X,Message) :-
    log_file(X,File),
    open(File,append,Stream),
```

```
    write(Stream,Message),
    nl(Stream),
    close(Stream),
    writeln(Message).
```

Store

unset(+FunctorName/+Arity)

```
unset(N/A) :-
    abolish(N,A).
```

unset_asked (+Asked)

```
unset_asked(Y) :-
    (asked(Y,_) -> retract(asked(Y,_)) ; true).
```

save_debug (+DebugMode)

```
save_debug(X) :-
    (debug(X) -> retract(debug(_))),
    assert(debug(X)),
    !.
```

save_index (-NextID)

Returns the next available ID

```
save_index(NextID) :-
    \+ last_index(_),
    assert(last_index(NextID)),
    !.
save_index(NextID) :-
    last_index(_),
    retract(last_index(_)),
    assert(last_index(NextID)),
    !.
```

save_observation (+X,+Observation)

```
save_observation(X,Observation) :-
    save_fact(manifests(X,Observation),ID),
    save_history(X,ID).
```

save_fact (+Fact,+ID)

```
save_fact(Fact,ID):-
    \+ fact(_,Fact),
    next_index(ID),
```

```
    asserta(fact(ID,Fact)),
    !,
    (is_debug -> writeln(fact(ID,Fact)) ; true).
save_fact(_,_).
```

save_history (+X,+ID)

```
save_history(X,ID) :-
    \+ fact_history(X,_),
    assert(fact_history(X,[ID])),
    !,
    (is_debug -> writeln(fact_history(X,[ID])) ; true).
save_history(X,ID) :-
    fact_history(X,History),
    retract(fact_history(X,History)),
    assert(fact_history(X,[ID|History])),
    !,
    (is_debug -> writeln(fact_history(X,[ID|History])) ; true).
```

asserta forces the focus-of-attention on new facts, therefore last found facts will be pursued first.

save_usedfact

```
save_usedfact(ID,Fact) :-
    (fact(_,Fact) -> retract(fact(_,Fact)) ; true),
    assert(usedfact(ID,Fact)),
    !,
```

save_rule

```
save_rule(ID,Head,Conditions) :-
    \+ rule(_,Head,Conditions),
    asserta(rule(ID,Head,Conditions)),
    !,
    (is_debug -> (writeln(rule(ID,Head,Conditions))) ; true).
```

save_trail (+Prev,+Curr)

Stores Curr in the same fact_history as Prev.

```
save_trail(Prev,Curr) :-
    \+ is_list(Curr),
    fact_history(X,History),
    memberchk(Prev,History),
    retract(fact_history(X,History)),
    assert(fact_history(X,[Curr|History])),
    !,
    (is_debug -> writeln(fact_history(X,[Curr|History])) ; true).
```

save_trail (+Prev,+List)

Appends the elements of the list to Prev history.

```
save_trail(Prev,[]).
save_trail(Prev,[H|T]) :-
    save_trail(Prev,H),
    save_trail(Prev,T).
```

actuator_init/1 (+Actuator)

Initializes the status.

```
actuator_init(Actuator) :-
    assert(actuator_status(Actuator,off)),
    !.
```

actuator_save (+Actuator,+Status,-Action)

Changes the status and returns the performed action.

```
actuator_save(Actuator,Status,changed) :-
    actuator_status(Actuator,OldStatus),
    OldStatus \= Status,
    retractall(actuator_status(Actuator,_)),
    assert(actuator_status(Actuator,Status)),
    !.
```

actuator_save

If it's already in the right status does nothing.

```
actuator_save(Actuator,Status,none) :-
    actuator_status(Actuator,Status).
```

saveln

```
saveln(X) :-
    \+ call(X),
    assert(X),
    !,
    (is_debug -> writeln(X) ; true).
saveln(X) :-
    call(X).

% saveln_a/1
saveln_a(X) :-
    asserta(X),
    !.
```

Knowledge Base mode

The kb_mode allows the user, straightforwardly, to browse the knowledge base rules and treatments.

kb_start

```
kb_start :-
    L = [rules,treatments],
    maplist(kb_browse,L).
```

kb_browse(+X)

```
kb_browse(X) :-
    askif(view(X)),
    browse(X).
kb_browse(X) :-
    asked(view(X),A),
    negative(A).
```

browse(X)

```
browse(X) :-
    X = rules,
    listing(rule).
browse(X) :-
    X = treatments,
    all(Condition-Treatment,treatment(Condition,Treatment),Treatments),
    maplist(writeln,Treatments).
```

Conclusions and future developments

The system acts as an expert to help the user identify plants' health statuses, continuously monitor a greenhouse and provide diagnostics. Even with a limited knowledge base it is a useful tool and could easily be improved.

As it is only a small application of ambient intelligence, there could be many ways to extend its functionalities, like using real sensors and actuators, sending email notifications or setting alarms, setting up an Internet server and allow the use through the cloud, build statistics to perform ML tasks on a larger scale, and so on.

The project also shows how forward and backward chaining work and how flexible is a logic programming language: declarative approach, knowledge base and automatic inference all in one place.

It has been a very long journey, but I daresay a very interesting one.

Glossary

Knowledge Base

Nutrient deficiencies

Nitrogen

- pale yellow color (chlorosis)
- older leaves turn completely yellow.
- flowering, fruitings, protein and starch contents are reduced
- reduction in protein results in stunted growth and dormant lateral buds

Phosphorus

- smaller leaf sizes
- lessened number of leaves
- slower rate maturation
- leaves and stems appear dark green or purple
- older leaves are affected first

Potassium

- reduced growth
- chlorosis and necrosis occurring in older leaves in later growth stages
- older leaves show mottled or chlorotic areas with leaf burn at the margins, usually leaving the midrib alive and green
- brown scorching and curling of leaf tips as well as chlorosis (yellowing) between leaf veins
- purple spots may also appear on the leaf undersides
- plant growth, root development, and seed and fruit development are usually reduced in potassium-deficient plants

Sulfur

- resembles nitrogen deficiency except yellowing occurs in new, younger leaves, rather than old, lower leaves.

Magnesium

- interveinal chlorosis with green mid-ribs

- Leaf margins become yellow or reddish-purple

Boron

- chlorotic young leaves and death of the main growing point
- leaves may develop dark brown, irregular lesions
- whitish-yellow spots may form at the base of the leaves
- leaves may become thickened, distorted and curled
- stems may be stunted
- flower buds may fail to form or be misshapen

Calcium

- localized tissue necrosis leading to stunted plant growth
- necrotic leaf margins on young leaves or curling of the leaves, and eventual death of terminal buds and root tips
- new growth and rapidly growing tissues of the plant are affected first
- the mature leaves are rarely if ever affected
- reduced height, fewer nodes, and less leaf area

Chloride

- chlorotic and necrotic spotting along leaves with abrupt boundaries between dead and alive tissue
- wilting of leaves along margins
- highly branched roots

Copper

- chlorotic younger leaves
- stunted growth
- delayed maturity
- excessive tillering
- lodging and sometimes brown discoloration

Iron

- yellowing (Chlorosis) occurs in the newly emerging leaves instead of the older leaves and usually seen in the interveinal region
- fruit would be of poor quality and quantity
- the yellowing may turn a pale white or the whole leaf may be affected

Manganese

- plant disorder that is often confused with, and occurs with, iron deficiency
- most common in poorly drained soils, also where organic matter levels are high
- manganese may be unavailable to plants where pH is high
- yellowing of leaves with smallest leaf veins remaining green to produce a 'chequered' effect
- younger leaves may appear to be unaffected
- brown spots may appear on leaf surfaces
- severely affected leaves turn brown and wither

Zinc

- growth is limited because the plant
- cannot take up sufficient quantities of this essential micronutrient from its growing medium.
- chlorosis
- necrotic spots
- bronzing of leaves
- resetting of leaves
- stunting of plants
- dwarf leaves
- malformed leaves

Pests

Aphids

Aphids live only about a week, but a mature female can reproduce rapidly. The tiny sucking pests, often found growing en masse on the underside of leaves, emit a sticky substance that draws ants and attracts sooty mold. Control aphids with neem oil or insecticidal soap.

- mass in large number
- sticky honeydew deposits
- white or grey "husks" littering the soil
- leaves become chlorotic in random patches
- growth may become distorted
- Treatment
- spray with neem oil and insecticidal soap

Thrips

Thrips are tiny flying insects with fringed wings. The sap-sucking insects discolor and distort nearly any type of plant. They leave tiny black specks of excrement on the leaves and often create white patches on leaves and petals. Thrips are difficult to control and often require a combination of methods such as sticky traps and insecticidal soap or neem oil.

- mottling, streaking, browning or yellowing on the leaves
- Treatment
- cut off and remove the infected leaves or flowers.
- spray with neem oil or natural pyrethrum

Spider mites

Spider mites are difficult to see with the naked eye, but they are easily recognized by the fine webs. The pests cause streaking, spotting and discolored leaves that may fall off the plant if not controlled. Neem oil and insecticidal soap are effective. Water properly, as mites are drawn to dry, dusty conditions.

- sticky webbing
- mottled leaves with lots of brown dots

Treatment

- shower the plant once a week
- purchasing the predatory mite Phytosieulus persimilis
- spray with neem oil and insecticidal soap

Scale insects

Scale damage can be devastating, as the tiny pests suck out the sweet nectar. There are two types of scale: hard scale, found primarily on woody tissue such as branches, trunks and twigs; and soft scale, which has a waxy protective covering. Control can be difficult, but neem oil works well by suffocating the pests. Regular use of insecticidal soap is also effective.

- sticky honeydew

Treatment

- spray neem oil and insecticidal soap
- dab individual scales with alcohol

White flies

Whiteflies are yet another type of sap-sucking pest. Small numbers are relatively harmless but large infestations can cause yellow or dry leaves that may fall off the plants. Like other sap-sucking pests, the sweet substance created by whiteflies attracts ants and sooty mold. To control whiteflies, try sticky traps and insecticidal soap or neem oil.

- leaves chlorosis
- dry leaves

Treatment

- spray neem oil and insecticidal soap
- sticky trap

Cutworms

Cutworms are the larval stage of certain moths. The destructive pests hide under leaves or other plant debris, emerging to lay masses of eggs on plants. They eat nearly anything in their paths, often cutting through stems of young plants at ground level. Remove plant debris. Pick off the pests by hand in late afternoon or evening. Create barriers with cardboard collars or gritty substances like eggshells, coffee grounds, or diatomaceous earth. Encourage birds to visit your garden.

Fungus gnats

Fungus gnats are tiny, annoying pests that wreak havoc on houseplants or in gardens or greenhouses. The swarms of flying insects are annoying, but it's the larvae that does the most harm by eating plant roots. Fungus gnats may also carry disease from plant to plant. Control adults with bright yellow sticky traps and/or insecticidal soap.

- small black flies around 2mm long

Treatment

- keep the soil less moist until they leave
- use the bottom watering method
- mix the nematodes with water and water directly

Mealy bugs

Mealybugs are common both indoors and outdoors, where they cause stunted growth, withering and yellowing of plants. The pests are easily recognized by the cottony protective covering. Insecticidal soap works well against the pests. Light infestations on indoor plants can also be removed with a toothpick or a cotton swab dipped in rubbing alcohol.

- clustering cottony covering under leaves and in the leaf joints
- plants look dehydrated
- plants may lose leaves rapidly
- stunted growth
- chlorotic leaves eventually drop off
- sticky honeydew residue

Treatment

- poke them off with a shake
- spraying with water
- spray neem oil and insecticidal soap

Diseases

Black spot

fungal, black round spots, upper side leaves (lower ones infected first). Infected leaves turn yellow and fall off. It occurs in extended wet weather periods or when leaves are wet for 6+ hours.

Tips for Controlling Black Spots on Leaves

- Plant in well-draining soil. Keep your plants healthy by providing regular feedings of organic fertilizer. This will help prevent fungal disease in plants.
- The fungus spores overwinter in plant debris. Remove dead leaves and infected canes from around the plants and disguard in the trash. Do not add to the compost pile.
- Disinfect your pruners with a household disinfectant after every use. Ethanol or isopropyl alcohol can be used straight out of the bottle.
- Because water (not wind) spreads the fungal spores, avoid applying water on the leaves. When you water, apply water directly to the roots. Use a soaker hose to water plants prone to the disease.

Leaf Spots

Fungal leaf spot disease can be found both indoors on houseplants, and outdoors in the landscape. This occurs during warm, wet conditions. As the disease progresses, the fungal spots grow large enough to touch each other. At this point the leaf surface appears more like blotches than spots. Leaf spot may result in defoliation of a plant. Follow the same tips as the ones to control black spot.

Powdery Mildew

Powdery mildew is a fungal disease that affects many of our landscape plants, flowers, vegetables and fruits. Powdery mildew is an easy one to identify. Infected plants will display a white powdery substance that is most visible on upper leaf surfaces, but it can appear anywhere on the plant including stems, flower buds, and

even the fruit of the plant. This fungus thrives during low soil moisture conditions combined with high humidity levels on the upper parts of the plant surface. It tends to affect plants kept in shady areas more than those in direct sun.

Tips for Controlling Powdery Mildew

- Inspect plants that you buy from a greenhouse before purchasing for mildew (and insects).
- Wiping off the leaves is not an effective powdery mildew treatment as it will return within days of cleaning.
- Because spores overwinter in debris all infected debris should be removed. Trim and remove infected plant parts.
- Do not till the debris into the soil or use in the compost pile.
- Space plants far enough apart to increase air circulation and reduce humidity.

Downy Mildew

Downy mildews produce grayish fuzzy looking spores on the lower surfaces of leaves. To identify downy mildew, look for pale green or yellow spots on the upper surfaces of older leaves. On the lower surfaces, the fungus will display a white to grayish, cotton-like downy substance. Downy mildew occurs during cool, moist weather such as in early spring or late fall. Spore production is favored by temperatures below 65°F and with a high relative humidity.

Tips for Downy Mildew Treatment

- Downy mildew needs water to survive and spread. It there is no water on your leaves, the disease cannot spread. Keep water off leaves as much as possible.
- Because the disease overwinters on dead plant debris, be sure to clean around your plants in the fall to help prevent the disease in the following spring.

Blight

Blight is a fungal disease that spreads through spores that are windborne. For this reason, spores can cover large areas and rapidly spread the infection. Blight can only spread under warm humid conditions, especially with two consecutive days of temps above 50°F, and humidity above 90% for eleven hours or more. No cure exists. Prevention is the only option.

Tips for Preventing Blight

- If growing potatoes, grow early varieties because blight occurs during mid-summer and you can harvest your crop before the blight.
- Plant resistant varieties: Sarpo Mira and Sarpo Axona are two varieties that show good resistance. Practice good garden hygiene.
- Destroy any blight-infected plant parts. Keep the area clean of fallen debris from your diseased plants and discard in the trash. Do not add to your compost pile.

Canker

Canker is often identified by an open wound that has been infected by fungal or bacterial pathogens. Some cankers are not serious while others can be lethal. Canker occurs primarily on woody landscape plants. Symptoms may include sunken, swollen, cracked or dead areas found on stems, limbs or trunk. Cankers can girdle branches and kill foliage. Cankers are most common on stressed plants that have been weakened by cold, insects, drought conditions, nutritional imbalances or root rot. Rodents can also spread the pathogens.

Tips for Controlling Canker in Plants

- Remove diseased parts in dry weather.
- Grow resistant varieties whenever possible.
- Avoid overwatering and overcrowding; avoid mechanical wounds such as damage from lawn mowers.
- Wrap young, newly planted trees to prevent sunscald. Sunscald creates dead patches that form on trunk and limbs of young trees if the trunks have been shaded, then transplanted to sunny areas.

- Keep plants healthy by planting in healthy soils and maintaining nutritional requirements.

Shot Hole

This disease of peach, apricot, plum and cherry spreads in warm wet weather infecting buds, blossoms, leaves, fruit and twigs (not large branches). Leaves develop numerous small, tan to purplish spots about 6 mm in diameter that drop out causing a shot hole appearance. Red to purplish spots also form on the fruit and can be accompanied by a clear, gummy substance. Gummy twig and small branch cankers also occur.

Shot Hole Control

plant resistant varieties. Rake up and destroy fallen leaves and prune out and destroy infected twigs and branches. To prevent twig and bud infections spray with Copper Spray: Peaches after harvest and all other trees in September before fall rains start.

Late Blight, Early Blight

Late Blight and Early Blight these are fungal diseases of tomatoes, potatoes and other related plants. Early blight appears as dark brown to black leaf spots with concentric rings. Black spots develop on stems and large, black, leathery, sunken spots on the fruit. Infections often occur in May or June in wet years. Late blight forms irregular greenish black, water-soaked blotches first on older leaves or stems quickly spreading to the fruit. This disease usually doesn't appear until August in wet years, but it can destroy entire plants overnight.

Late Blight and Early Blight Control

Space and prune plans for good air circulation. Avoid overhead watering. If Early blight starts to appear, pick off and destroy the infected leaves. If chemical control is required apply a copper spray at 7 to 10 day intervals. If late blight starts to appear remove diseased leaves or entire plants immediately, seal in a plastic bag and send to the landfill. Do not compost late blight infected plants. Apply a copper spray at every 5 to 10 days till allowed days before harvest.

Botrytis Blight or Grey Mold

Botrytis Blight or Grey Mold is a grey fuzzy mold develops on dead and dying plant tissue spreading to healthy tissue when conditions are wet. Infections first appears as water-soaked spots or areas on soft or senescent foliage, flower parts and young stems. On flowering plants, woody ornamentals and small fruit this disease can cause flower, leaf and shoot blights as well as stem and fruit rots. Very susceptible plants include: peonies, roses, hostas, strawberries and raspberries.

Botrytis Blight or Grey Mold Control

Plant resistant cultivars. Thoroughly clean and discard garden debris and refuse in the fall to reduce the level of grey mold in your garden. Susceptible plants (that are sun loving) should be grown in sunny areas with good air circulation. If practical water at the base of plants not over the foliage. If botrytis appears, remove infected leaves and fruit. It is rarely worth applying fungicides to control this disease.

Verticilium Wilt

Verticilium Wilt is a serious fungal disease of many deciduous trees, herbaceous perennials, berries and vegetables. It is of particular concern for flowering cherries. It enters roots from the soil moving upwards in the plant, plugging up the plants transportation system. Visible indication that there is a problem starts with yellowing, wilting and dying back of young twigs and branches often on one side of plant or tree. Many other problems look the same, however Verticillium wilt gets worse from year to year. Cutting into a woody stem with a knife reveal black or brown streaks in the wood are vascular cambium just under the bark.

Verticilium Wilt Control

Control is all preventative as there is no cure once a plant is infected. Avoid drought stress or flooding on mature landscape trees. Remove dead and dying plants including the infested roots and the soil and replant with tolerant or resistant species. When pruning trees that may have this disease, sterilize your pruning tools between trees to prevent spreading it to an and noninfected tree. Rubbing alcohol, Lysol or a 10% household bleach solution (corrosive) can be used to disinfect pruning tools. Once an area is infected with Verticilium Wilt, we generally suggest not planting the same species in that area for several years.