Assignment 3: Multi-Tap Plant Watering

Bar-Ilan CS 89-570
Introduction to Artificial Intelligence

January 9, 2026

1 Before you start

- **Use of AI / online sources.** You may use them, but you *must* clearly note *everywhere* you used them and for what purpose. Failure to disclose will reduce your grade.
- In this assignment, submit only the file `ex3.py`, **but rename it before submission to** `ex3_id.py`, where `id` is your student ID.
- If you have any questions or need clarification about the assignment, contact me at yifatyank@gmail.com.

2 Introduction

In this exercise you will continue working with the *Plant Watering* problem from previous assignments, but in the Reinforcement Learning (RL) setting. Which means: the result of the robot's actions and the rewards accumulated from watering the plants are non-deterministic **and the distributions of the results are unknown tho the agent.**

Your task stays the same as in *assignment 2*, to implement a controller function that, given the current state of the task and the number of steps left, chooses the next action. The goal is to **maximize the expected accumulated reward** over the episode within the number of steps (*horizon*) provided.

The full problem description is provided below (Figure 1), changes from the last assignment are marked in blue.

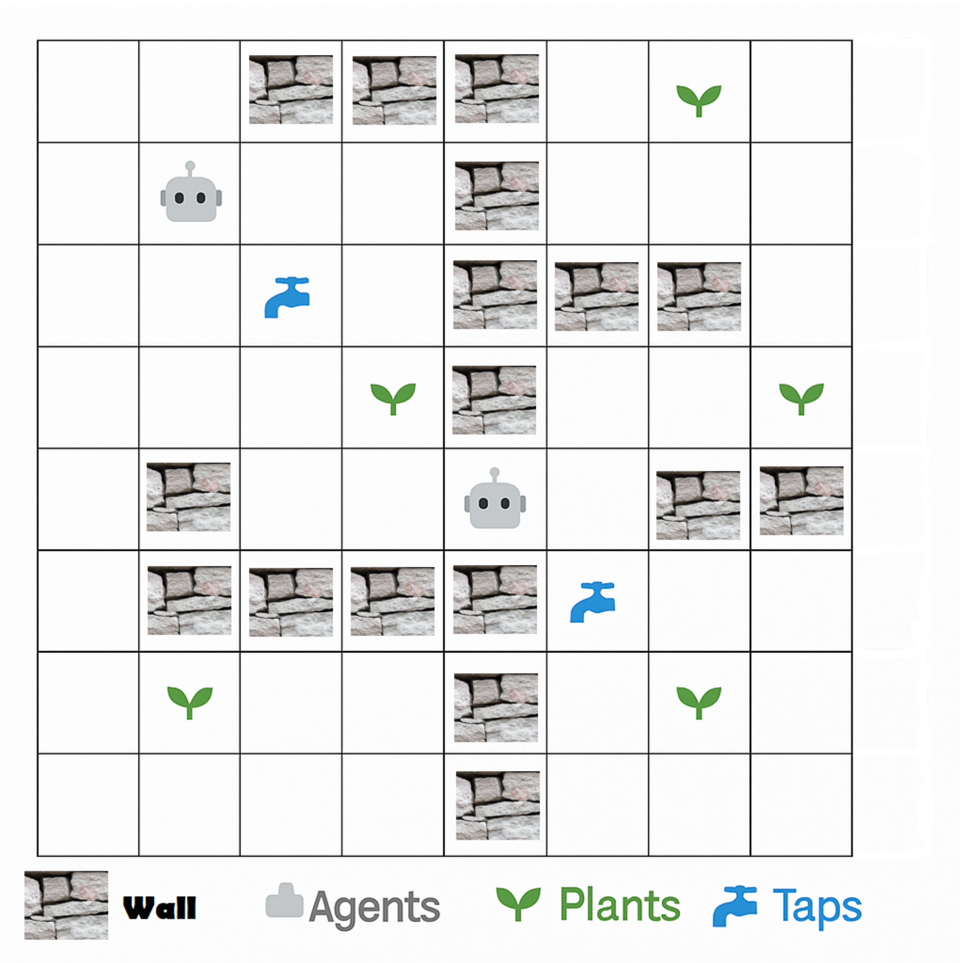


Figure 1: The original *Plant Watering* domain.

3 Problem Description

3.1 Deterministic core:

- The world is an $N \times M$ grid with walls, taps, plants, and robots.
- A tap cell contains a certain number of water units (WUs).
- A plant cell requires a certain number of WUs to be fully satisfied.
- Each robot has a location, a current load (how many WUs it carries), and a capacity (the maximal WUs it can carry).

3.2 Actions:

Robots can execute the following basic actions:

- **UP, DOWN, LEFT, RIGHT**: move one cell in the corresponding direction, if the target cell is inside the grid and is not a wall.
- **LOAD**: if a robot is standing on a tap, it can load 1 WU from the tap (if there is water left) into its own load, as long as it does not exceed its capacity.
- **POUR**: if a robot is standing on a plant and carries at least 1 WU, it can pour 1 WU into that plant, reducing the plant's remaining need by 1 and its own load by 1.

In addition, the environment has the action RESET.

- The action **RESET** can be performed from each state of the world.
- The action returns the world to it's initial state(robots' locations, number of water units in taps, number of water units each plant needs).
- The accumulated rewards stay the same.(The rewards do not reset).

3.3 Stochastic results of actions and rewards

- **Defective robots.** Each robot r has an unknown success rate. This value is the probability that the robot will *successfully* execute the action it chooses at a given step. With the remaining probability, the robot "misbehaves" and the intended action fails in a specific way (described below). Pay attention, in this setting the success rate is not included in the representation of the world's state. The failure probabilities are unknown and may vary between problems, but always the same for different episodes of the same problem.
- **Plant rewards are stochastic.** Each plant (i, j) has its own reward distribution which is unknown to the agent. Whenever a robot *successfully* pours one WU into that plant, the reward for this action is drawn from that distribution. Although the reward distribution of each plant is not unknown to the agent, we do provide you with the highest reward each plant gives(for pouring one WU into that plant). And the rewards given by pouring each plant are always positive rewards.
- **Goal reward.** When *all* plants in the instance have received all the water they need, the episode is considered completed, and an additional deterministic `goal_reward` is given.
- **Horizon.** This is the **overall** budget of steps the agents has to accumulate the reward.

3.4 Actions and their outcomes

For each chosen action, the environment code performs the following steps:

1. **Legality check.** First, the task checks that the chosen action is legal in the current state:
   - A move action (UP, DOWN, LEFT, RIGHT) must be in the set of applicable actions from the robot's current cell (i.e., it cannot go into a wall or outside the grid).
   - LOAD is only legal if the robot is on a tap cell, the tap still has water, and the robot's current load is strictly less than its capacity.
   - POUR is only legal if the robot currently carries at least 1 WU and there is a plant on the same cell which still needs water.

   If the action is illegal, the engine raises an error – something that should never occur with a correct controller.
2. **Success vs. failure.** If the action is legal, the controller flips a biased coin (according to the distribution) for that robot:
   - With the *robot's success probability* action succeeds.
   - With the remaining probability the robot is "defective" on this step and the action fails, leading to a different outcome.

We now specify the exact effect of each action in both cases.

Move actions (UP, DOWN, LEFT, RIGHT).

- **On success:** the robot moves one cell in the intended direction.
- **On failure:** the robot chooses randomly between:
  - moving in a *different* legal direction (any other move from the applicable actions of its current cell), and
  - staying in place.

  One of these options is chosen uniformly at random, and the robot's position is updated accordingly.

LOAD.

- **On success:** the robot loads 1 WU from the tap into its own load, and the tap's remaining water decreases by 1. If the tap's remaining water reaches 0, it is removed from the state (it is no longer considered a tap).
- **On failure:** nothing changes: the robot's load and the tap's water remain the same, and no reward is obtained.

POUR.

- **On success:**
  - the robot's load decreases by 1,
  - the plant's remaining water need decreases by 1,
  - if the plant's remaining need reaches 0, the plant is removed from the state, and
  - a reward is sampled uniformly at random from the plant's reward list and added to the total reward.
  - the total remaining water need in the instance is updated accordingly.
- **On failure:** the robot loses 1 WU from its load (it "spills" the water), but the plant does not receive any water and no reward is obtained.

RESET. At any step, you may choose to reset the state to the initial state. This action always succeeds (probability 1), gives no reward, and costs 1 step. So, if your steps budget was x before applying reset, it will be $x - 1$ after applying RESET.

Episode termination and goal reward. Whenever the total remaining water need over all plants becomes 0, the agent receives the additional `goal_reward`, the episode is considered finished, and the task is reset-ed to the initial state.
Note that applying any action always costs 1 step (RESET included).

In the next section we formalize the initial state representation and the global optimization goal.

4 Initial State and the Goal

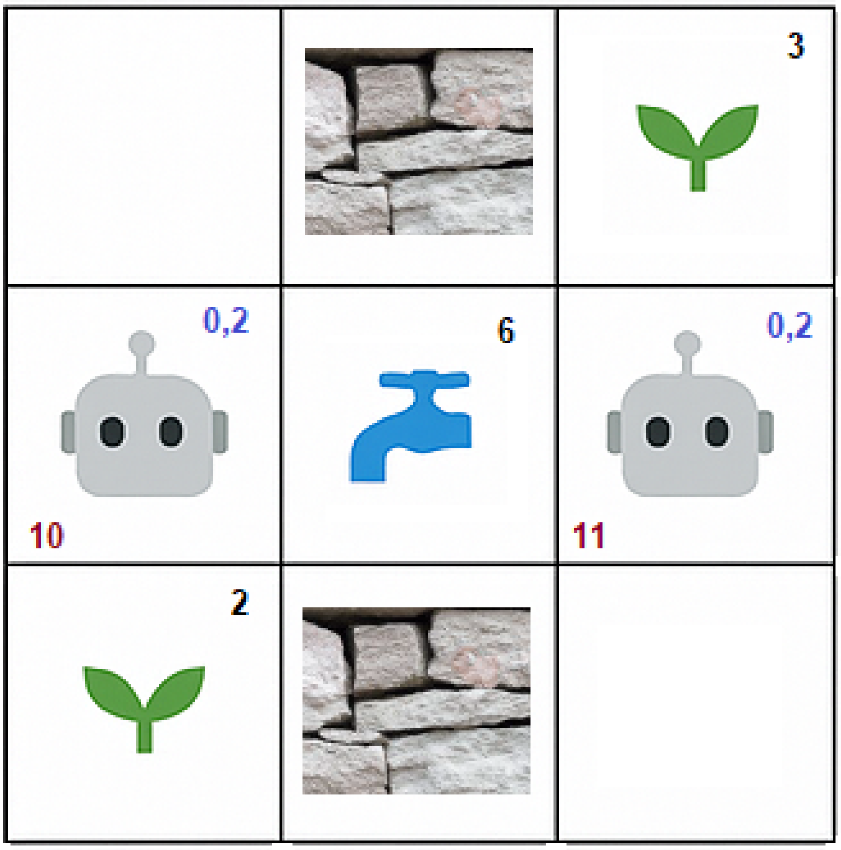
4.1 Initial state representation.

The initial state of the problem is given as a Python dictionary `init_state`, with the following entries:

1. **Size** – a pair (N, M), where N is the number of rows and M is the number of columns.
2. **Walls** – a set of coordinates of the form (i, j). If $(i, j) \in$ `Walls`, then the cell in row i and column j contains a wall and cannot be entered.
3. **Taps** – a dictionary. Each key is a coordinate (i, j) of a tap, and the value is an integer giving the number of available water units (WUs) in that tap.
4. **Plants** – a dictionary. Each key is a coordinate (i, j) of a plant, and the value is an integer giving the number of WUs required by that plant.
5. **Robots** – a dictionary. Each key is a robot ID. The value is a tuple of four values $(i, j, load, capacity)$, where (i, j) is the robot location, `load` is the current number of WUs carried by the robot, and `capacity` is the maximal number of WUs this robot can carry.
6. **plant_max_reward** – a dictionary. Each key is a plant coordinate (i, j), and the value is the maximum reward that can be gained by pouring one WU into the plant.
7. **goal_reward** – a number giving the reward received when *all* plants have received all the water they need (i.e., when the global goal is achieved).
8. **seed** – an integer seed used to initialize the random number generator for this problem instance (so that runs are reproducible).
9. **horizon** – the overall number of steps the agents have to accumulate the reward in task.

The success rate of each robot and the distribution of rewards given after watering each plant are hidden and are not included in the state.

4.1.1 Example




```
init_state = {
Size: (3,3),
Walls : {(0,1),(2,1)},
Taps : {(1,1): 6},
Plants: {(2,0): 2, (0, 2): 3},
Robots: {10: (1,0,0,2), 11: (1,2,0,2)},
plant_max_reward: {(0, 2) : 5 ,(1, 2) : 10},
goal_reward: 20,
seed: 45
horizon: 80
}
```

4.2 Goal description.

As in assignment 2 the goal is *maximizing reward*. Formally, the goal of your controller is to choose actions that **maximize the expected total reward** obtained during the episode, where the reward comes from:

- stochastic rewards when pouring water into plants, and
- a deterministic `goal_reward` once all plants have received all required WUs. After you received the `goal_reward` the state is reset to the initial state.

Your implementation will not construct a full plan in advance, but will decide, at each step, which action to take based on the current state.

5 State Representation

In this assignment, the state representation is fixed and you must not change it:

`(robots_t, plants_t, taps_t, total_water_need)`

Where:

- *robots_t* is a tuple of robots. Each robot is represented as $(rid, (r, c), load)$, where rid is the robot's ID, (r, c) is the robot's position, and $load$ is the amount of water the robot is currently carrying. Note: You can obtain a robot's capacity using a provided function.
- *plants_t* is a tuple of plants. Each plant is represented as $(pos, need)$, where $pos = (r, c)$ is the plant's position, and $need$ is the amount of water the plant still needs at the current time.
- *taps_t* is a tuple of taps. Each tap is represented as $(pos, water)$, where $pos = (r, c)$ is the tap's position, and $water$ is the amount of water currently available in the tap.
- *total_water_need* is the sum of the water needs of all plants.

6 Get Functions

- A function that returns `self._done`, which indicates whether the task has finished (e.g., when `steps == max_steps`):

  ```
  def get_done(self):
      return self._done
  ```

- A function that returns the number of steps performed so far:

  ```
  def get_current_steps(self):
      return self._steps
  ```

- A function that returns the problem description (i.e., the model):

  ```
  def get_model(self):
      return self._model
  ```

- A function that returns the reward gained after the last action was performed.

  ```
  get_last_gained_reward
      return self._last_gained_reward
  ```

- A function that returns the current accumulated reward obtained from the actions taken so far:

```
def get_current_reward(self):
    return self._reward
```

- A function that returns the current state after the steps performed so far:

```
def get_current_state(self):
    return self._state
```

- A function that returns the horizon of the problem (maximum number of steps):

```
def get_max_steps(self):
    return self._max_steps
```

- A function that returns a dictionary of robot capacities (`rid : capacity`):

```
def get_capacities(self):
    return self._capacities
```

- A function that returns a dictionary of the maximum reward that can be gained by pouring 1 WU into each plant.

```
def get_plants_max_reward(self):
    return self._plant_max_reward
```

7 What You Must Implement

We provide a starter codebase. Most infrastructure is already implemented. Your job is to complete the methods so that the checker can solve instances.

Files and required functions (in `ex3.py`)

You *must* implement the following (you may add helpers as you see fit):

1. **`choose_next_action(state) -> str`**
   - **Input:** a state tuple `state` $= (robots\_t, plants\_t, taps\_t, total\_water\_need)$.
   - **Output:** a *single* legal action encoded as a string. For robot actions, the format is `'ACTION(robot_id)'` (e.g., `'UP(3)'` or `'POUR(7)'`).
     Note: If you choose the reset action, return the string `'RESET'` (without a robot ID).
   - **Possible actions (depending on legality):** UP, DOWN, LEFT, RIGHT, LOAD, POUR, RESET.

8 Scoring & Evaluation

- There will be **8 problem instances**
- Each instance will be evaluated using **30 different random seeds**. For each seed, we run the instance once, record the total reward, and then compute the instance score as the average reward across all tested seeds (sum of rewards divided by the number of seeds).

- Time-limit for each instance and seed is: $20 + 0.5 \cdot \text{horizon}$
- To receive a passing grade (60):
  - For each problem in the test set provided with the starter code, **your reward must be higher than that of a random policy**.
  - There is a **validation check in `ext_plant.py`**. Make sure that every action you return from `choose_next_action` passes this check.
- To receive a grade higher than 80, your code must surpass the performance of our baseline, which we will publish next week.
- The last 20% of the grade will be determined on the basis of performance. *This is not a competition: More than one student can receive 100.*

9 Attached files

The starter code includes the following files:

1. **`ex3.py`** — *the only file you submit, and the only file that will be graded.* Implement `__init__`, `choose_next_action`, and set your `id` variable. Do not rename or move this file.
2. **`ex3_check.py`** — a local *self-checker*. Use it to run your code on the provided instances. You may add additional problems here for your own testing, but the grading system will *not* use your added problems. Changes to this file do not affect grading.
3. **`ext_plant.py`** — implementation code that runs your action-selection function on the current state. You do not submit this file; do not rely on modifying it for your solution.

10 Rules & Submission

- **Edit only `ex3.py`, change its name to `ex3_id.py`, e.g. ex3_123456789.py** Do not modify or submit any other file.
- **Individual work.** Discussing ideas is allowed, but *sharing or copying code is prohibited.*
- **Use of AI / online sources.** You may use them, but you *must* clearly note *everywhere* you used them and for what purpose. Failure to disclose will reduce your grade.
- **Academic integrity.** We take academic integrity **very seriously**. Cheating of any kind will result in a failing grade for the course at a minimum, as well as additional disciplinary actions. Please do not make us deal with this.
- **Questions.** Ask during office hours or in the assignment forum only.
- **Extensions.** No extensions will be granted except for documented exceptional circumstances via email.
- **Deadline:** 28/1/2026.
- **Identification.** Set your ID in the variable `id` in `ex3.py`.
- **What to submit & where:** Submit `ex3.py`, **but rename it before submission to** `ex3_id.py`, where `id` is your student ID. Upload the renamed file to the Bar-Ilan submission website.

Rules (summary).

- Unit action cost (i.e., each action costs one step).
- At any given time step, only one robot acts, while the others do not.
- To load a WU, a robot must stand on the same cell as a tap.
- To pour into plant, a robot carrying water must stand on the same cell as plant.
- At most one robot per cell. At most one tap per cell. At most one plant per cell.
- Stay within grid bounds.
- Each robot start with load = 0.
- The function you need to implement, `choose_next_action(state) -> str`, takes a state as input and returns a string-coded action.
- The goal is to maximize the rewards accumulated within the given number of steps (horizon).