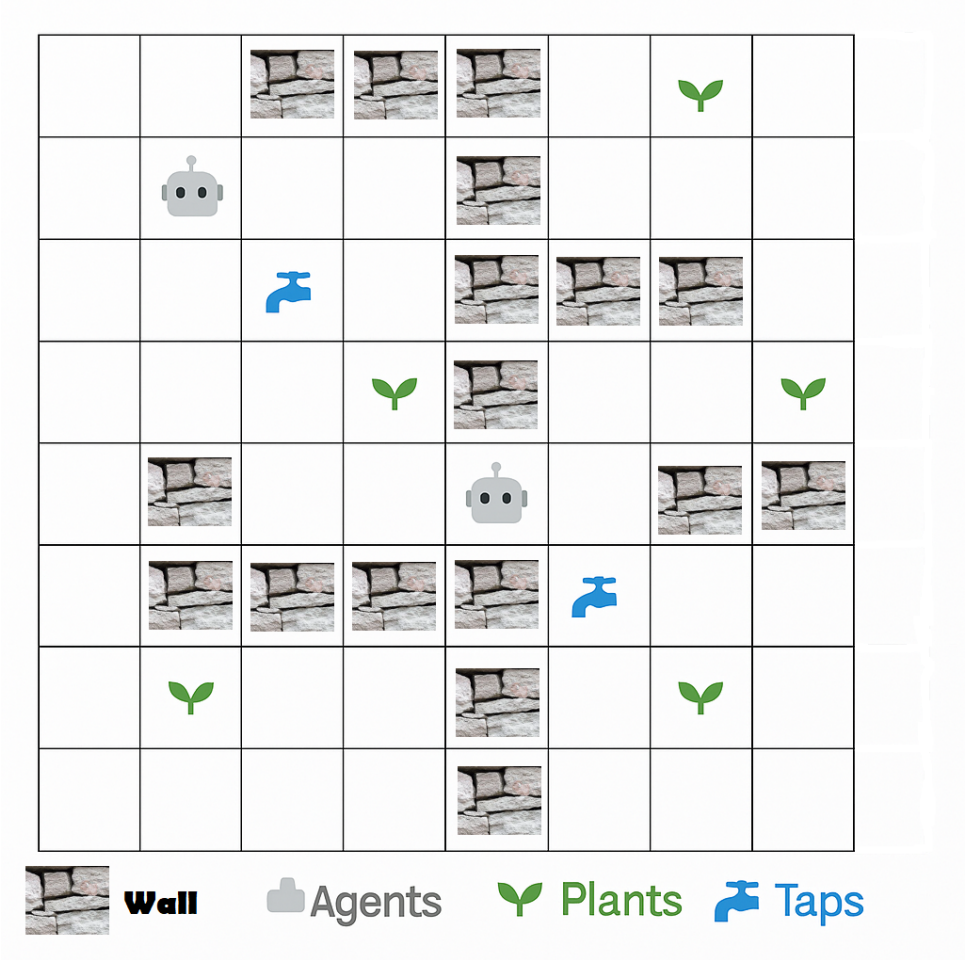
Assignment 1: Multi-Tap Plant Watering

Bar-Ilan CS 89-570
Introduction to Artificial Intelligence

November 16, 2025

1 Introduction

In this exercise you will *experiment with solving a planning/path-finding problem* using deterministic search algorithms we studied GBFS, A*). Our problem is called *Plant Watering.*



Summary of the Domain In this problem, robots move on a 2D grid, LOAD water at a tap, and POUR water on plants until each plant reaches its required amount (its target number of water units (WU)). While doing so, robots must respect their *carrying capacities*. Some cells on the grid may contain impassable walls.

This domain combines *grid navigation* with *resource management* (carried water, per-plant totals, and global totals).

2 Problem Description

The grid is represented by NxM cells, each cell can be empty, or contain one of the three types of immovable objects, additionally there are robots moving on the greed.

Robots You control a set of robots that move on a grid. Each robot has a unique ID number, a carrying capacity (the maximum number of WU it can hold), and a current water budget (the number of WU it currently carries). Each robot has six actions, executable under the appropriate preconditions:

- UP(ID), DOWN(ID), LEFT(ID), RIGHT(ID) – move the robot with the given ID accordingly on the grid. Robots cannot leave the grid, and cannot enter a cell containing a wall or another robot.
- LOAD(ID) – the robot loads one WU from a collocated non-empty *tap*. The robot's updated WU budget cannot exceed its carrying capacity. Loading a unit reduces the tap's remaining WU by one.
- POUR(ID) – the robot pours one unit of water into a collocated *plant*. To do this, the robot must have a non-empty water budget. As a result, one WU is removed from the robot's budget and added to the plant. Each plant requires a specified number of WUs.

No two robots can occupy the same grid cell (at the same type).
At each time step, exactly one robot acts.

Action representation (string format). Let i denote the **robot index**. Each action must be encoded as the following string:

- RIGHT(ID): `RIGHT{id}`
- LEFT(ID): `LEFT{id}`
- UP(ID): `UP{id}`
- DOWN(ID): `DOWN{id}`
- LOAD(ID): `LOAD{id}`
- POUR(ID): `POUR{id}`

This means that if you have the plan (up, right, load, down, pour) with only one robot (e.g., 10 if the robot has ID = 10), then the output should be the following: [`UP{10}, RIGHT{10}, LOAD{10}, DOWN{10}, POUR{10}`]. In the successor function, each child is represented as (action, next_state), and you should store this string in the action field.

Immovable objects There are three types of immovable objects on the grid.

- **Taps** – water containers from which robots obtain their water supply. A robot must be in the same cell to execute the action LOAD.
- **Plants** – each plant requires a specified amount of water. The problem is **SOLVED** when **all** plants have received their initially specified number of WUs.
- **Walls** – these cells are blocked and cannot contain taps, plants, or robots. *For clarity*: no robot can start in a wall cell or move into it.

Each cell cannot contain two objects of the same type.

Moving the robots (UP / DOWN / LEFT / RIGHT) Each actions moves one robot at each time (this is specified by the ID of the robot).

Let robot ID be located in a cell (x, y), where x is the row index and y is the column index. This robot four potential moves $(x + 1, y)$ (DOWN), $(x - 1, y)$ (UP), $(x, y + 1)$ (RIGHT), and $(x, y - 1)$ (LEFT). Such moves are legal only if the resulting cell is (1) within the grid limits, (2) there is no wall or another robot in the target cell.

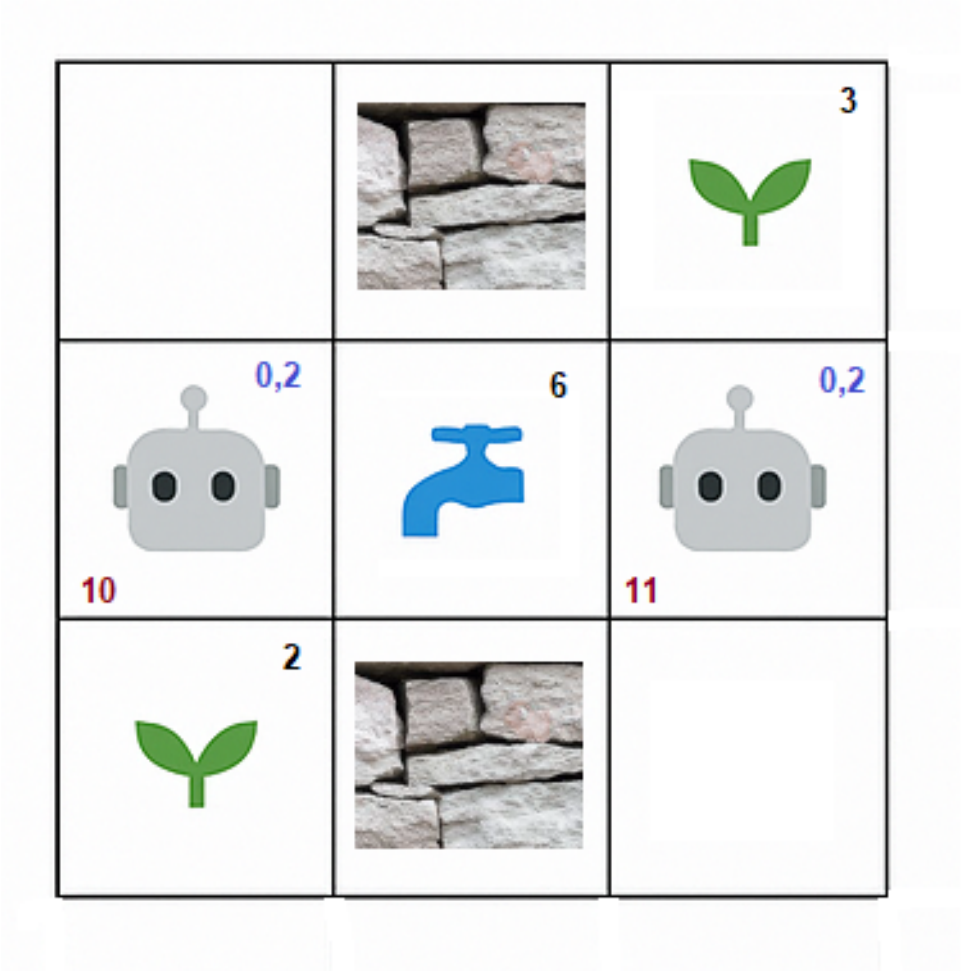
3 Initial State and the Goal

Initial state representation The initial state of the problem is represented as a *init_state* Python dictionary with the following entries:

1. **Size** – (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). This means that in the cell in the i'th row and j'th column there is a wall.
3. **Taps** – each entry in the dictionary has the key (i, j) as the coordinates of the tap, and the number (integer) of the available WUs in the tap.
4. **Plants** – each entry in the dictionary has the key (i, j) as the coordinates of the plant, and the number (integer) of the WUs required in the plant.
5. **Robots** – here the key is the robots ID. The values is a tuple of 4 values $(i, j, load, capacity)$, where i, j are the location of the robot, *load* is the current number of WU in the robot, and *capacity* the maximum number of WUs the robot can carry.

Goal description The robots should provide to **each plant the number of WU it requires**.

4 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)}
}
```

5 Tips

We provide a starting state *init_state*. Each student may represent the state however they like, but once you define the state representation in the Python initialization function, you must keep using the same representation for every state you create in the successor function. Note that each state must be *hashable*, i.e., all its components must be immutable! Examples of immutable structures in Python include: tuple, string, frozen set. You can also make hashable classes, to this end you need to implement the functions `__eq__`, `__hash__` in the class.

- Keeping the **A* heuristic admissible** (e.g., the blind heuristtic). Making it *informative* and still admissible is the real challenge. Note that if you have separate ideas for admissibile heuristic, you can always combine them.
- Make **GBFS heuristic** fast, not just and informative.

- Handle **multiple taps** correctly. This can be tricky!
- Generate only **legal actions** according to the instance settings (bounds, walls). **It will be checked**, so think – how can you check for this yourself?
- Keep the **successor function** as small and efficient as possible, to avoid **redundant successors**.
- Note that while LLMs can write you a heuristic, they cannot prove that this heuristic is admissible (or sound for this matter).

It must be noted that **A*** is typically slower than **GBFS**. Therefore, when you run the same problem with both algorithms (**A***, **GBFS**), they will behave differently. In general, **A*** will take more time and expand more nodes than **GBFS**, because it also considers the cost-so-far $g(s)$ in addition to the heuristic $h(s)$. In contrast, **GBFS** looks only at the heuristic $h(s)$, so it usually runs faster but may return suboptimal solutions.

6 Important

Since the problem can be represented the in many different ways (again – you are free to choose how to represent the state and the problem), we need the actions names and the IDs of the robots to be consistent among all submissions. Do not change the IDs of the robots you got as an input. Use the provided action names and format. For example, the action that moves up a robot with ID = 11 is written as `UP{11}`.

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.

Context. The state is your encoding, that you choose to represent the problem.

Files and required functions (in `ex1.py`)

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

1. `successor(state) -> List[(action, next_state)]`
   - **Input:** a `state` object (hashable) representing the current game configuration.
   - **Output:** a *list* of tuples (`action`, `next_state`), one for every applicable primitive action from the current state.
   - **Actions to consider (6 total):** UP(ID),DOWN(ID),LEFT(ID),RIGHT(ID),POUR(ID),LOAD(ID).
   - **Side effects (purely functional):** `next_state` must be a *new* object with updated positions/counters; do not mutate `state` in place.
2. `goal_test(state) -> bool`
   - **Input:** a `state`.
   - **Return:** `True` iff $\forall p \in \text{plants}: \ \text{poured}(p) = \text{target}(p)$.
3. `h_astar(node) -> int` (*A* heuristic; must be admissible*)
   - **Input:** a `node` whose property `node.state` holds the current state.

- **Output:** a nonnegative integer that *lower-bounds* the remaining cost to any goal.
- **Meaning:** this is the heuristic A* uses; it must never overestimate the optimal remaining cost. (Consistency is preferred but not required.)

4. `h_gbfs(node) -> int` (*GBFS heuristic; may be non-admissible*)
   - **Input/Output:** same as `h_astar`.
   - **Meaning:** this is the heuristic GBFS uses; it *ranks* states for speed and does not have to be a lower bound (can overestimate).

8 Scoring & Evaluation (How the checker works)

- There will be **10–20 problem instances** (the exact number is decided by the course staff).
- Each instance is tested **twice**: once with **A*** and once with **GBFS**, using your implementations of `successor`, `goal_test`, and the corresponding heuristic.
- The running time of each test instance is measured. If it is too long (timeout depends on the problem size, but typically no more than 30 secs), the maximum time is measured.
- The sum total (cumulative) running times of the test instances will be used as a basis for the score. We also take into program crashes, illegal moves (e.g., into walls), and similar failures.
- To receive a passing grade:
  - The number of program crashes, illegal moves, and other failures must be 0.
  - For each problem in the test set provided with the starter code, the running time must be below 30 seconds per problem.

What counts as "solved"? An instance is considered solved if your algorithm returns a *valid* action sequence that reaches a goal state and passes the automatic simulator checks (preconditions/effects and goal).

- **Additional requirement for A*:** the returned plan *must be optimal* with respect to the instance's action costs. The checker verifies optimality (e.g., by comparing the reported cost to a reference optimum or by ensuring no cheaper plan exists).
- **For GBFS:** any valid plan that reaches a goal and passes the simulator checks counts as solved. When GBFS solutions are compared (e.g., for tie-breaking or bonus), *fewer actions (lower cost)* is better.

9 Attached files

The starter code includes the following files:

1. `ex1.py` — *the only file you submit and the only one graded.* Implement `successor`, `goal_test`, `h_astar`, `h_gbfs`, and set your `id` variable. Do not rename or move this file.
2. `ex1_check.py` — local *self-checker*. Use it to run your code on the provided instances. You may add *more* 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. **`utils.py`** and **`search.py`** — auxiliary code (data structures and search drivers used by the assignment). You do not submit these files; do not rely on modifying them for your solution.

How to self-check (quick start). Run the checker from the assignment folder:

```
python ex1_check.py
```

Follow the comments inside `ex1_check.py` to select algorithms (A*/GBFS) and instances. You can run it with python3 if you want to.

10 Rules & Submission

- **Edit only `ex1.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 plagiarism seriously—please do not make us deal with it.
- **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:** 1/12/2025.
- **Identification.** Set your ID in the variable `id` in `ex1.py`. In addition, submit a file `details.txt` containing:

  (a) Your full name (English only)

  (b) Your ID number
  (one item per line)

- **Submission instructions will be send later.**

Rules (summary).

- Unit action cost (A* returns optimal plans under this metric).
- One robot performs an action at a time.
- 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.