Marsquake Rover Simulator Technical Documentation

Akshay Gahlot
Ekansh Mahendru
Hansin Ahuja

Table of Contents

Technology Stack

- Programming languages:
    - Python 3.7 for backend services
    - JavaScript for user interface
- Flask 1.1.1
- Azure App Services

Agent - Environment Interaction through Object Oriented Programming

- The environment, agent and their interaction in the AI system has been emulated through OOP concepts**.**
- Class **Environment** is the template for an environment object, and class **Agent** is the template for an agent object in this environment.
- Algorithms detailed in the actions directory are the **member functions** of an object of class Agent and **act as actuators** for this AI system.
- Other member functions act as **sensors and effectors** to simulate the system.
- We follow the condition - action rule, where the state of the environment is mapped to an action performed by an agent.
- The environment object projects percepts throughout its lifetime, which an agent object detects through its sensors. The agent manifests its **rationality** following the aforementioned condition - action rule, and exercises the actuators in the form of several path-finding algorithms. The effectors then effectuate some change, which affects the state of the environment, and the process repeats.
- The mapping from various components of an AI system to our objects and their members has been shown in Fig. 1.

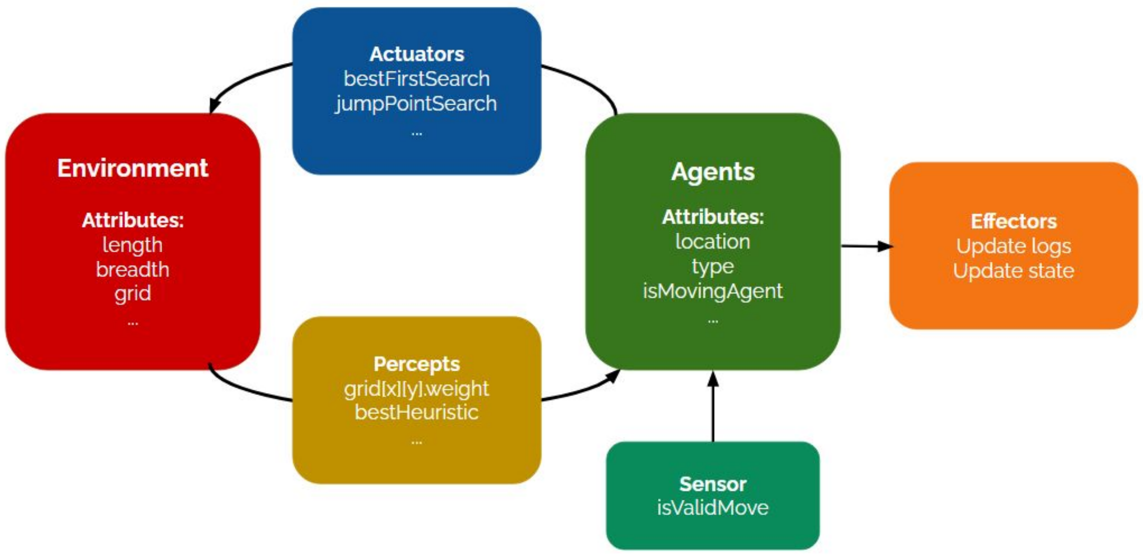


Fig 1: Interaction between the objects of class Agent and Environment through member functions and member attributes.

Class Environment:

- The environment class has attributes **length**, **breadth** and **grid** that define its state. The grid attribute of class Environment uses two utility classes:
    - **class Cell:** Represents one cell of the grid, which has attributes such as location, weight, type, etc.
    - **class Location:** A wrapper class to encapsulate the properties associated with a location, namely the x-coordinate, y-coordinate and neighbours. Note that the neighbour of a wormhole entry point will be the wormhole exit point, and not its physical neighbours.
- An environment object has the following methods and attributes that act as the percepts in the system:
    - **bestHeuristic():** It estimates the remaining distance between the agent and the destinations.
    - **cell.weight:** Weight of the cell agent is currently in.
    - **cell.srcAgent:** The first source agent to reach the current cell.
    - **cell.destAgent:** The first destination agent to reach the current cell.
    - **cell.type:** The type of the current cell, which can be an obstacle, wormhole, destination, etc.
- These methods and attributes, among others, provide important information to the agent and are essential for decision making, hence act as percepts.

Class Agent:

- The Agent class has attributes such as **location** (that tells the current position in the environment) and **isMovingAgent** (that tells if sensors and actuators are to be used).
- Its method **isValidMove()** acts as a sensor and perceives the percepts of the environment. It helps the agent make valuable decisions about valid moves that it can make.
- Based on this information, the actuator (algorithm selected) animates the agent by moving it appropriately in the environment. On every move, the information about the environment is updated by compiling the various information provided by different agents on the grid, conveyed through the agent's member attribute 'logs', which effectuates necessary changes in the environment.
- Once the percept is received that destination has been reached, through **cell.srcAgent** (an attribute possessed by each cell object of the grid attribute in the environment object), sensor and percepts are no more required and the followed path is traced back by the agent.

Flow of information:

- The API request contains grid configurations, which is sent to **findPath()** which validates the input first, and throws an exception on failure.
- If the configuration is valid, it is either sent to checkpointMode() or nonCheckpointMode(), depending on which mode is selected.
- **nonCheckpointMode()** initializes the agents, environment, flags, etc. and acts as the driver for the entire process. It calls on the agents to execute required algorithms until a path is found, or it is determined that a path isn't possible. It returns the following:
    - **path:** The final path to be reported.
    - **gridChanges:** The changes registered in the grid in the entire run.
    - **timeTaken:** Time taken to execute the function call.
    - **activatedCells:** Currently activated cells in the environment. Returned only if called by checkpointMode().
- **checkpointMode():** The first checkpoint can be treated as the destination for the source, the second checkpoint as the destination for the first checkpoint, and so on. So this function makes the necessary changes and calls **nonCheckpointMode()** until the destination is reached, or if it has been determined that the destination cannot be reached.
- The flow of information has been shown in the form of a flowchart in Fig. 2.

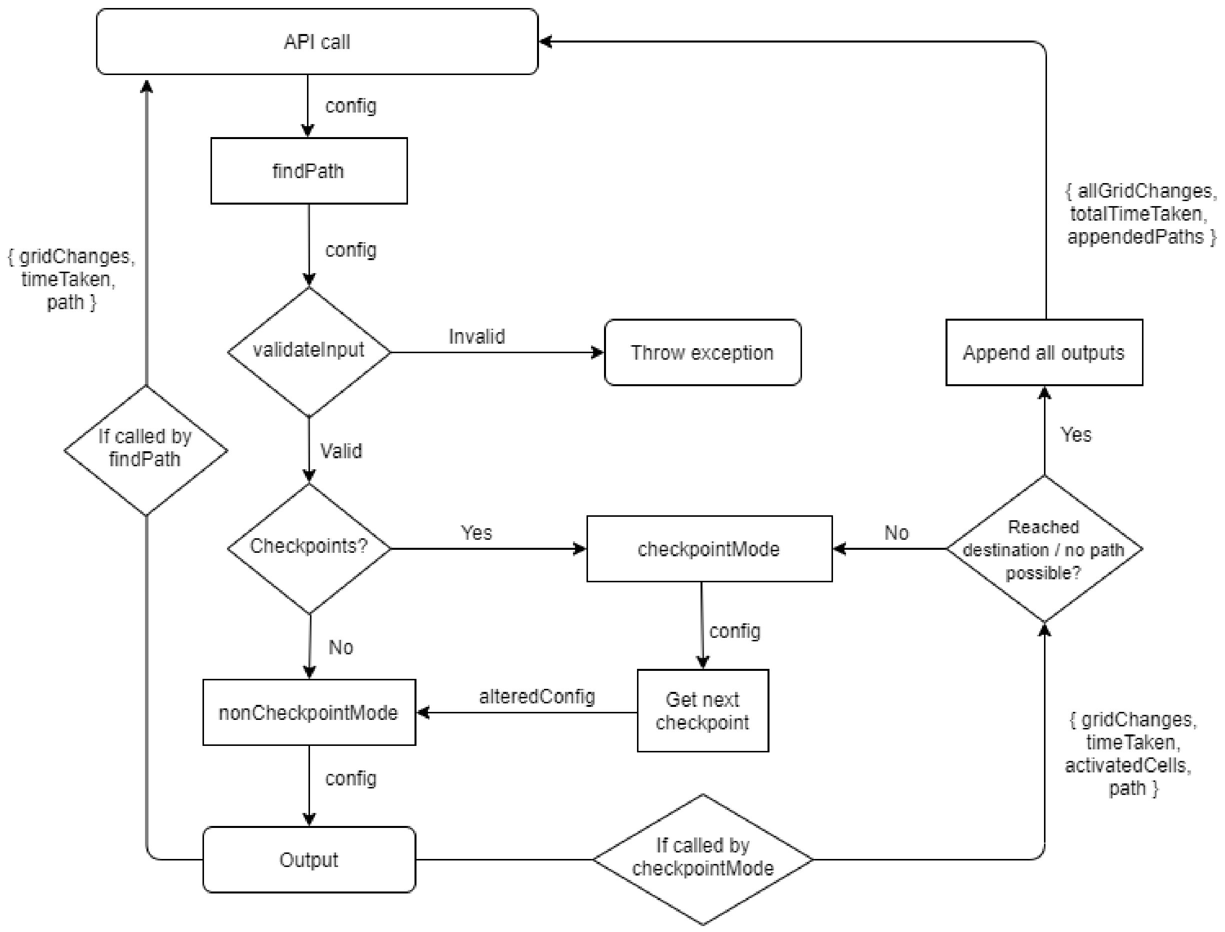
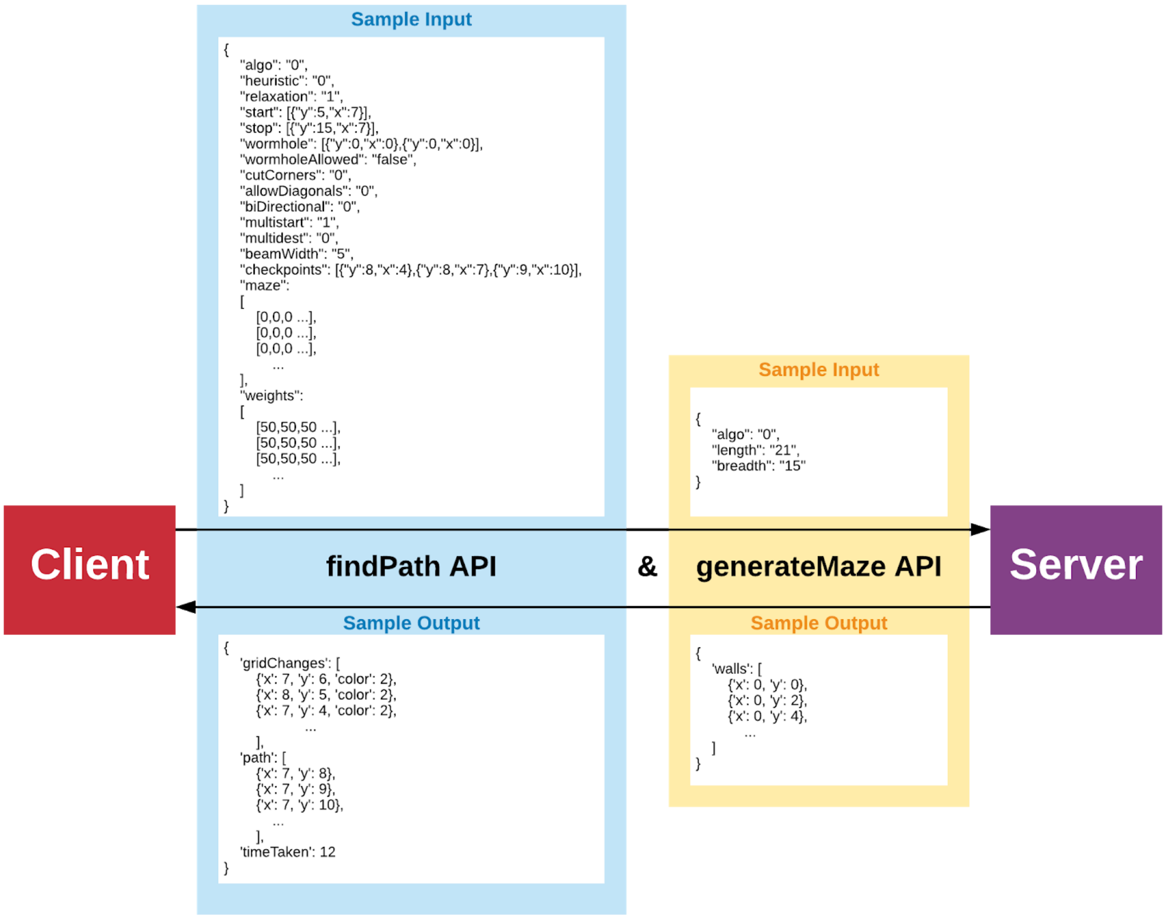


Fig 2: Flow of information from the API request to its response

The Application Program Interface:

- There are two endpoints exposed using the Flask server, one for finding a path in the maze and one for generating the random maze.
- For a request to the former, all the configurations and other options are sent along with the maze and cell weights to find the path. The response consists of 3 parameters: gridChanges: the sequence in which each cell is processed, path: the path to be drawn at the end and timeTaken: the time taken to simulate path-finding.
- For generating a maze, only the algorithm, length and breadth of the maze are required and the API returns the wall locations.
- The sample input and output of both the endpoints, along with the interaction between the server and client, has been shown in Fig. 3.



```
{
    "algo": "0",
    "heuristic": "0",
    "relaxation": "1",
    "start": [{"y":5,"x":7}],
    "stop": [{"y":15,"x":7}],
    "wormhole": [{"y":0,"x":0},{"y":0,"x":0}],
    "wormholeAllowed": "false",
    "cutCorners": "0",
    "allowDiagonals": "0",
    "biDirectional": "0",
    "multistart": "1",
    "multidest": "0",
    "beamWidth": "5",
    "checkpoints": [{"y":8,"x":4},{"y":8,"x":7},{"y":9,"x":10}],
    "maze":
    [
        [0,0,0 ...],
        [0,0,0 ...],
        [0,0,0 ...],
            ...
    ],
    "weights":
    [
        [50,50,50 ...],
        [50,50,50 ...],
        [50,50,50 ...],
            ...
    ]
}
```

```
{
    "algo": "0",
    "length": "21",
    "breadth": "15"
}
```

```
{
    'gridChanges': [
        {'x': 7, 'y': 6, 'color': 2},
        {'x': 8, 'y': 5, 'color': 2},
        {'x': 7, 'y': 4, 'color': 2},
                    ...
        ],
    'path': [
        {'x': 7, 'y': 8},
        {'x': 7, 'y': 9},
        {'x': 7, 'y': 10},
            ...
        ],
    'timeTaken': 12
}
```

```
{
    'walls': [
        {'x': 0, 'y': 0},
        {'x': 0, 'y': 2},
        {'x': 0, 'y': 4},
            ...
    ]
}
```

Fig 3: Sample I/O of the endpoints