HackTrick Maze Environment Documentation

Maze Environment Description

The maze environment provided in the repo is comprised of multiple files, containing multiple classes. Together, these classes work to represent a Maze, allowing an agent to move inside. In specific positions inside the maze, there are “rescue items/ survivors” that the agent should try to rescue. Each rescue item in the maze, once reached, is blocked by a security “riddle”. Such riddles must be solved by an automated algorithm to successfully rescue the survivor. The maze environment exposes to the players the current state – giving contextual information related to the current position and related to each of the rescue items. By sending actions to the maze environment, an agent could navigate through the maze to reach a rescue item and/or reach the maze exit. To solve a riddle, the maze environment exposes a dedicated function to submit the solution, and automatically adds the rescue item if the solution is correct.

It is allowed to terminate the game at any given time and the server will in turn save the attempt and calculate the score. Otherwise, the server will not end the attempt until the timeout has been reached.

In this documentation, we describe the **important** files and classes in the environment that you will interact with to run and test your solution locally. Any other files in the environment, not mentioned in this document, are abstracted for your convenience and should not be modified during running locally. The documentation provided should be sufficient to allow your agent to interact with the environment with ease.

Important notes:

- **Each rescue item is guarded by a security riddle that is mandatory to solve to rescue a survivor at a given position.**
- **For each rescue item, the agent gets 1 and only 1 attempt to solve the riddle. In case the solution is incorrect, or the candidate’s solution fails to call the solve function correctly, the rescue item is removed and will not be available in the current submission to play in the maze.**
- **Once a new submission is started, the maze environment is reset. Therefore, the agent will start again at position (0,0), all the rescue items will be available once more and the score will be reset for the current submission.**
- **The maze environment that will run on the HackTrick server during your submission(s) are deployed and any modifications done locally on your environment will not be effective during submissions and may result in failed attempt(s).**
- **If the candidate’s solution does not terminate the game, the HackTrick server will not terminate the game until the timeout is reached. Therefore, the score will be calculated on**

the total number of steps taken. Terminating the game is the candidate's responsibility and failing to terminate your game correctly may affect your submission score.

Maze Manager File

The Maze Manager file holds the code responsible for instantiating a game for an agent, and exposes functions to navigate the maze, and solve any riddles encountered on the way. These functions can be utilized by your agent when running locally to develop and test your solution. The Maze Manager file provides a helpful abstraction of the whole Maze Environment, so you will not need to modify other files in the Maze Environment. The file includes multiple class implementations, and imports Maze Environment files. The classes implemented are documented here, and at the end of this documentation, we provide the steps needed to run the environment (python requirements, and setup instructions).

Class MazeManager

Attributes:

- maze_size: specifies maze size
- maze_map: maintains a dictionary that maps agent_id to MazeEnv object **(you must be using the id specified for you through email)**
- riddles_dict: dictionary to map agent id to RiddleContainer object.
- rescue_items_dict: map rescue item position to riddle type
- riddle_scores: specifies the weights assigned to each riddle to be used while scoring.

Functions:

- init_maze(agent_id, maze_cells): creates instantiation of a Maze and assigns it to the agent_id in the maze_map.
- init_riddles(agent_id): Instantiates a riddle container and assigns it to the agent_id in the riddles_dict
- pull_riddle(riddle_type, agent_id): retrieves the riddle container associated with the agent_id and returns the riddle question based on riddle_type
- solve_riddle(riddle_type, agent_id, solution): verifies the riddle solution submitted by the agent. The function validates the position of the agent before allowing them to solve, and verifies that the riddle is not already solved or attempted before.
- randomize_rescue_items(): determines random locations for the rescue items to be embedded in the maze.
- step(agent_id, action): moves the agent inside their corresponding maze, returns the resulting state, reward, terminated & truncated flags, and additional info.

- reset(agent_id): resets the maze corresponding to the agent_id, returns the resulting state, reward, terminated & truncated flags, and additional info.
- get_action_space(agent_id): returns the allowed action for the agent id in the current position.*
- get_observation_space(agent_id): returns the state information of the agent in the current position. *
- is_game_over(agent_id): returns a Boolean indicating whether the agent reached the exit or not. *
- render(agent_id, mode="human", close=False): renders the maze and the agent actions. *
- set_done(agent_id): terminates the game
- calculate_final_score(agent_id): calculates the final score compromised of the rescue score and the riddle score, and checks if the agent escaped the maze or not. If not, the score will be reduced to 80% of the original value.
- calculate_current_score(agent_id): calculates the current score comprised of the rescue score and the riddle score.
- get_rescue_items_status(agent_id): returns the rescue item status where: 0 represents not attempted, 1 represents solved, and 2 represents nulled.

* Methods that will be only used with the remote server during submission, you will not use them when testing locally.

Class RiddleContainer

Attributes:

- cipher_riddle: creates a cipher riddle object
- server_riddle: creates a server riddle object
- pcap_riddle: creates a pcap riddle object
- captcha_riddle: creates a captcha riddle object
- riddles: a dictionary that stores the riddle objects with their types as the keys

Functions:

- get_riddle(riddle_type): takes a riddle type as input and returns the corresponding riddle object. For example, if the input is "cipher", it returns the cipher_riddle object.
- reset_riddles(): resets the "solved" flag for all the riddle objects to False.

Class Riddle – base class

Attributes

- riddle_type: a string representing the type of riddle (e.g., "cipher").
- riddle_dir_path: a string representing the directory path where the riddle files are stored.
- riddle_question: represents the riddle question.
- riddle_solution: represents the correct answer to the riddle.
- _solved: a boolean flag that indicates whether the riddle has been solved or not.
- attempts: an integer representing the number of attempts made to solve the riddle.

Functions:

- load_riddle(): loads the riddle question and solution from the riddle directory
- get_question(): returns the riddle question
- set_solved(): sets the _solved flag to True
- solved(): gets the _solved flag
- solve_riddle(solution): receives solution and compares it to the correct solution. If the solution is correct, the _solved flag is set to True.

Class CipherRiddle(Riddle)

Class CaptchaRiddle (Riddle)

Class ServerRiddle (Riddle)

Class PcapRiddle (Riddle)

These classes inherit from the base Riddle class with some modifications to the load_riddle() method.

How To Run Locally

The ReadMe on the repository provides information on how to build the repository to run your solution. The *local_solver.py* file is an example solver provided for your convenience where you can modify and run your solution locally, and you can use it to submit your solution to the HackTrick competition.

To build the Maze Environment on your local device, you will need to navigate to gym-maze/ and run the following command:

```
python setup.py install
```

To run your solution on your local device, you will need to navigate to gym-maze/ and run the following command:

```
python local_solver.py
```

The environment will by default enable_rendering of the game in real time, this will help you visualize how your agent is performing. Therefore, it is recommended that you run the environment on a Windows system. Alternatively, if you require running on a Linux system, you will need to disable rendering by modifying the **enable_render** argument in maze_manager.py file **line 32** to **False.**

To run the agent locally, in the main function, you can run the function "local_inference()".

How To Run an attempt on HackTrick Server

The submission_solver.py file also provides an option to run the solution as a submission to the HackTrick competition.

To run the agent on our server, in the main function, you can run the function "submission_inference()".

```
python submission_solver.py
```

Solving a maze

The maze is considered solved when the agent reaches the exit which is located at (9,9) and calls the API to end the game. This needs to be done before the timeout set by the environment or reaching the maximum number of steps. More details regarding the timeout conditions can be found in the server/API documentation.

Solving a riddle

In order to solve a riddle, the agent must first parse the type of riddle from the received state. Then, the agent should call the dedicated function that you have implemented to solve this type of riddle and respond with the solution.
If your agent's solution matches the reference solution used for evaluation, the riddle will be marked as solved and the number of rescue items will be incremented.