Micromouse Maze Simulator Documentation

Release 0.1.6

The Micromouse Maze Simulator contributors.

Jun 19, 2018

Contents

CHAPTER 1

What is Micromouse Maze Simulator?

Micromouse Maze Simulator is a micromouse search algorithm simulator. It acts as a server, so any client can connect to it and send requests to read walls and log the current exploration state. Watch it in action!

CHAPTER 2

Contents

2.1 Usage

2.1.1 Requirements

In order to use the Micromouse Maze Simulator you need:

- Python 3.5 (or higher).

2.1.2 Installation

Installation is very straight-forward using Python's `pip`:

```
pip3 install --user mmsim
```

This will install all the required dependencies and will provide you with the `mmsim` command.

2.1.3 Launching

To run the simulator simply:

```
mmsim
```

This will, the first time, download a set of mazes from micromouseonline.

If you have a local maze files collection in text format you may use that instead:

```
mmsim your/local/collection/path/
```

2.2 Protocol

2.2.1 Communication

The Micromouse Maze Simulator acts as a server, which means any client can connect to it and communicate with it using message passing.

Using message passing for communication means you can use any programming language to implement the client, as long as you correctly implement the communication protocol.

Communication is implemented using ØMQ, which is available for almost every programming language and implements multiplatform inter-process communication in a much more convenient way than raw TCP sockets.

To see which interface/port the server binds to by default:

```
mmsim --help
```

Note, however, that you can specify your preferred interface/port when launching the server:

```
mmsim --host 127.0.0.1 --port 1234
```

The server binds a single REP socket, which means the client is expected to communicate with the server sending requests, and the server will always send a reply back. This is important, as you must remember to receive and process that reply from the client. ØMQ forces the request-reply communication pattern to be correct and complete.

2.2.2 Implementing a basic client

To better understand how this works, we will start by implementing a very simple client in Python, to make sure the connection is well established:

```
import zmq


ctx = zmq.Context()
req = ctx.socket(zmq.REQ)
req.connect('tcp://127.0.0.1:6574')

req.send(b'ping')

reply = req.recv()
print(reply)
```

This simple client will send a `ping` request to the server and the server will reply back with a `pong` message.

Note: If you start the client before the server, it will wait for the server to be available.

In C it would look like this:

```
#include <stdio.h>

#include <zmq.h>


int main (void)
```

(continues on next page)

(continued from previous page)

```
{
    char buffer[4];
    void *context = zmq_ctx_new();
    void *requester = zmq_socket(context, ZMQ_REQ);

    zmq_connect(requester, "tcp://127.0.0.1:6574");
    zmq_send(requester, "ping", 4, 0);
    zmq_recv(requester, buffer, 4, 0);
    printf("%s\n", buffer);

    zmq_close(requester);
    zmq_ctx_destroy(context);
    return 0;
}
```

Which can be compiled with:

```
gcc client.c -o client -lzmq
```

And should result in the same `pong` reply being printed when executed.

2.2.3 Protocol

We have already seen part of the protocol implemented. In this section we will describe each of the requests the client can send to the server.

Ping

When the client sends 4 bytes with the word `ping` to the server, the server replies back with the word `pong` (another 4 bytes).

This request is for testing purposes only, but should be useful if you are starting to implement a client from zero.

Reset

When the client sends 5 bytes with the word `reset` the server will reset the simulation, which means that any information related to the last or current simulation will be deleted.

This request is useful to be executed always when starting the client, to make sure the server starts with a client state too.

The server always replies back with an `ok`.

Here is a simple `reset` example implemented in Python, which is almost the same as with the `ping` request:

```
import zmq


ctx = zmq.Context()
req = ctx.socket(zmq.REQ)
req.connect('tcp://127.0.0.1:6574')

req.send(b'reset')
```

(continues on next page)

(continued from previous page)

```
reply = req.recv()
print(reply)
```

Reading walls

The client can read walls at the current position. In order to do so, it must send the current position to the server:

```
<W><x-position><y-position><orientation>
```

The request is formed with 4 bytes:

1. `W`: is the W byte character, idicating a request to read walls at the current position.
2. `x-position`: is a byte number indicating the x-position of the mouse.
3. `y-position`: is a byte number indicating the y-position of the mouse.
4. `orientation`: a byte character, indicating the mouse orientation (`N` for North, `E` for East, `S` for South and `W` for West). Indicates where the mouse is heading to.

Positions are defined considering:

- Starting cell (x=0, y=0) is at the South-West.
- When going from West to East, the x-position increments.
- When going from South to North, the y-position increments.

The server replies with 3 bytes indicating the walls around the mouse.

1. First byte (boolean byte) indicates whether there is a wall to the left.
2. Second byte (boolean byte) indicates whether there is a wall to the front.
3. Third byte (boolean byte) indicates whether there is a wall to the right.

Here is an example in Python to read walls just after exiting the starting cell, when we are at (x=0, y=1) position and heading north:

```
import struct
import zmq


ctx = zmq.Context()
req = ctx.socket(zmq.REQ)
req.connect('tcp://127.0.0.1:6574')

req.send(b'W' + struct.pack('2B', 0, 1) + b'N')

left, front, right = struct.unpack('3B', req.recv())
print(left, front, right)
```

Sending exploration state

This is probably the most important and complex request. It basically sends the current state of the client including the mouse current position, the discovered walls so far and all the weights assigned to each cell in the maze.

The request looks like this:

1. `S`: is the S byte character, idicating we are sharing the state.
2. `x-position`: is a byte number indicating the x-position of the mouse.
3. `y-position`: is a byte number indicating the y-position of the mouse.
4. `orientation`: a byte character, indicating the mouse orientation (`N` for North, `E` for East, `S` for South and `W` for West). Indicates where the mouse is heading to.
5. `C`: a byte character indicating how the cell numbers matrix is being transmitted. `C` means C-style. If that does not work well for you, try `F`, for Fortran-style.
6. `numbers`: a byte array of 256 bytes. Each byte represents a number.
7. `C`: a byte character indicating how the walls matrix is being transmitted. `C` means C-style. If that does not work well for you, try `F`, for Fortran-style.
8. `walls`: a byte array of 256 bytes. Each byte represents a number.

Walls are defined with a bitmask:

- 2^0: less significant bit. Set it to 1 to mark the cell as visited.
- 2^1: Set it to 1 to specify East wall is present.
- 2^2: Set it to 1 to specify South wall is present.
- 2^3: Set it to 1 to specify West wall is present.
- 2^4: Set it to 1 to specify North wall is present.

Note: The simulation server will store every state received until a reset occurs. This allows you to execute the full simulation as fast as possible and then navigate through the state history using the graphical interface.

Here is an example in Python to send a state with fake walls and cell numbers. We set the same wall and an increasing number for all cells:

```
import struct
import zmq


ctx = zmq.Context()
req = ctx.socket(zmq.REQ)
req.connect('tcp://127.0.0.1:6574')

req.send(b'reset')
req.recv()

numbers = list(range(256))
walls = [2] * 256

state = b'S' + struct.pack('2B', 0, 1) + b'N'
state += b'C'
state += struct.pack('256B', *numbers)
state += b'C'
state += struct.pack('256B', *walls)

req.send(state)

reply = req.recv()
print(reply)
```

Now try and play a bit with that script:

- Change the mouse position and orientation.
- Change the walls sent.
- Change the numbers sent.
- Change `C` and `F` in the numbers sent to understand the differences.
- Remove the `reset` and see how the state history increases and how you can navigate through it.

2.3 Examples

2.3.1 Protocol tester in Python

The Micromouse Maze Simulator project includes an example to test the communication protocol. Only the code required for the client requests is implemented, with no actual mouse search logic.

The standalone tester example can be found as a script in the project.

2.3.2 Simple solver in Python

The Micromouse Maze Simulator project includes a fully working solver example implemented in Python. This client implements a robot that is able to explore the maze (i.e.: read and update walls, move...) while making decisions on which move to make on each step.

The decision making is very simple:

- Each time the robot passes through a cell, it increments its associated weight by 1.
- It then looks around to see the possible moves it can make (i.e.: having a wall will prevent a movement in that direction).
- Among the possible moves, it will look at the weights of the next possible cells and select the lowest weight.
- It will move to the cell that has the lowest weight.

The standalone simple solver example can be found as a script in the project.

Note: There is a lot of code just to implement maze storage, adding walls, client-server communication... The decision making and the actual search algorithm is implemented in the last functions. In particular, the `run_search()`, `best_step()` and `recalculate_weights()` functions.

Some functions start with an underscore. Those, you probably do not need to edit them nor directly call them from your client.

Using this client template you may try your own search algorithms. For example, if you want to try a left-wall follower robot, simply change the `best_step()` function to:

```
def best_step():
    prefer = ['left', 'front', 'right', 'back']
    allowed_steps = [step for step in prefer if is_allowed_step(step)]
    return allowed_steps[0]
```

Although this search algorithm will fail to reach the goal in many cases.

2.3.3 A real micromouse client in C

A real, complete micromouse client implemented in C can be found in the Bulebule micromouse project. This client executes the search algorithm that effectively runs in the Bulebule micromouse robot, implementing only the required functions to communicate with the Micromouse Maze Simulation server.

To try that client you need to first download the Bulebule repository:

```
git clone https://github.com/Theseus/bulebule.git
```

Then change to the `scripts/` directory and compile the client:

```
cd bulebule/scripts/
make
```

Note: You need to have ZMQ libraries installed in your system in order to compile the project.

Now simply run the client while the Micromouse Maze Simulator is running!

2.4 Developers

2.4.1 Installing dependencies

To install the required dependencies for developing Micromouse Maze Simulator, you can make use of the provided `requirements.txt` file:

```
pip install -r requirements.txt
```

2.4.2 Running tests

Running the tests locally is very simple, using Tox from the top level path of the project:

```
tox
```

That single command will run all the tests for all the supported Python versions available in your system or environment.

For faster results you may want to run all the tests just against a single Python version. This command will run all tests against Python 3.5 only:

```
tox -e py35
```

Note that those tests include style and static analysis checks. If you just want to run all the behavior tests (not recommended):

```
pytest -n 8
```

If you just want to run a handful of behavior tests (common when developing new functionality), just run:

```
pytest -k keyword
```

Note: Before submitting your changes for review, make sure all tests pass with `tox`, as the continuous integration system will run all those checks as well.

2.4.3 Generating documentation

Documentation is generated with Sphinx. In order to generate the documentation locally you need to run `make` from the `docs` directory:

```
make html
```

2.5 Indices and tables

- genindex
- search

Index