Assignment 1

Usage

Setup

Carefully follow Rahul Shome's instructions

In this directory, run the following command

```
chmod 755 setup.sh && ./setup.sh
```

Running a Maze

Modify the heuristic function and weight in `$PRACSYS_PATH/python/config.py` to meet your needs

Now watch our program navigate any maze by running the following command

```
roslaunch prx_core assignment_1.launch env:="PATH_TO_MAZE_FILE"
```

- PATH_TO_MAZE_FILE must point to a file that follows the maze format outlined in this assignment

Generate a Report

You can start a report server by running

```
python3 $PRACSYS_PATH/python/report_server.py numMazes pathsPerMaze
outputFile
```

- `numMazes` is the number of mazes you want to report
- `pathsPerMaze` is the number of paths per maze you want to report
- `outputFile` is the file where your report will be saved after all paths of each maze complete
- mazes are differentiated by the file of the maze, the heuristic function used, and the heuristic weight used

The default port is 8080, if you want to use another port, change the value of `port` in `$PRACSYS_PATH/python/report.py`

In this directory, run the following command to automate generating the report outlined in the assignment:

```
chmod 755 report.sh && ./report.sh
```

A new terminal window will open displaying the server. Make sure you keep an eye on the server terminal window because you would need to manually close a maze when the server indicates to move onto the next maze. When all data is received, the server will generate a report in `$PRACSYS_PATH/report.txt` and the window will close.

Report

Results

- maze10x10x10%
    - Manhattan heuristic
        - Weight 1
            - Number of nodes in 10 paths: 72
            - Number of expanded nodes in 10 paths: 126
        - Weight 10
            - Number of nodes in 10 paths: 74
            - Number of expanded nodes in 10 paths: 65
    - Zero heuristic
        - Weight 1
            - Number of nodes in 10 paths: 72
            - Number of expanded nodes in 10 paths: 355
        - Weight 10
            - Number of nodes in 10 paths: 72
            - Number of expanded nodes in 10 paths: 355
- maze10x10x20%
    - Manhattan heuristic
        - Weight 1
            - Number of nodes in 10 paths: 67
            - Number of expanded nodes in 10 paths: 117
        - Weight 10
            - Number of nodes in 10 paths: 69
            - Number of expanded nodes in 10 paths: 60
    - Zero heuristic
        - Weight 1
            - Number of nodes in 10 paths: 67
            - Number of expanded nodes in 10 paths: 302
        - Weight 10
            - Number of nodes in 10 paths: 67
            - Number of expanded nodes in 10 paths: 302

Observations

The most noticeable observation in the report is that the weight has no effect on paths that were found by the zero heuristic search. This is also intuitive given that 0 * weight will always equal 0. The zero heuristic search is basically the same as uniform cost search because they both only take the true cost from start into account which also guarantees optimality. Due to the absence of heuristics, expanding nodes had no sense of direction thus more nodes had to be expanded.

The Manhattan heuristic search, especially with higher weight, expanded fewer nodes than the zero heuristic search because it had a sense of direction for which nodes to expand. A drawback of increasing the weight very high is that you loose optimality. When the h value has such a high weight, the g value is almost neglected, thus nodes in the path are selected based on a heuristic rather than the true cost. Since the Manhattan heuristic function in our world was admissible, the true cost can be greater than the heuristic value, and thus, high weight lowers optimality. Depending on your problem, expanding fewer nodes while sacrificing optimality might be what you need. In other problems, optimality cannot be sacrificed.

Implementation

We implemented an A* Search algorithm that finds an optimal path from start to goal. You can see the algorithm implemented in the `aStar()` function in `astar.py`.

`cpp-io.py` calls `search()` in `search.py` passing in appropriate arguments it got from the c++ program. `cpp-io.py` translates and sends `search()`'s return value back to the c++ program. `search()` defines the start and goal nodes, gets the enviornment from the file, calls `aStar()`, and constructs a path list from `aStar()`'s return value.

All files and functions are well commented so you can read them to understand the code in more detail.

Python-C++ Inter-Process Communication

To implement our program in python and passing the data to and back from c++ to python, we:

1. Create a pipe in c++
2. Run the python script, `$PRACSYS_PATH/python/cpp-io.py`, as a child process, passing inputs as arguments
3. Write data to the pipe to send back computed output

Report Generator Client-Server Communication

To store report data over multiple instances of our A* search program, we had to create a server that stays alive during execution of the multiple instances.

The client:

1. After completing the search tries to connect to the server
2. If the server is running send the data to the server
3. If the server is not running, don't report the data

The server:

1. Accepts a connection from a socket
2. Receives data from the client
3. Organizes data in a data structure
4. Keeps accepting connections as long as it needs more data
5. Saves data to file once all data has been received
6. Exits

Protocols

Definitions

An **integer** is a 4-byte network-byte-order integer.

A **string** is a message that starts off as an **integer** that tells the length of the string to come.

A **intTupPairList** is a message that starts as an **integer** that tells the number of **tuple**s that will follow. Each **tuple** is 8-bytes where the first 4-bytes represent the first **integer** in the tuple and the last 4-bytes represent the second **integer** in the tuple.

Python-C++ Protocol

These are the command-line arguments passed to the python program from c++

```
cpp-io.py write_fd filename initial_i initial_j goal_i goal_j
```

The python program sends an **intTupPairList** representing the path

Report Generator Client-Server Protocol

1. Client sends server an **integer** as the heuristic weight
2. Client sends the maze name as a **string**
3. Client sends the heuristic function name as a **string**
4. Client sends the path as an **intTupPairList**
5. Client sends the expanded nodes as an **intTupPairList**

Extra Credit

To reuse information between repeated, chained searches in the same environment run the following before starting running a maze:

```
python3 $PRACSYS_PATH/python/remember.py
```

This will not work for multiple environments. You must exit the server before running another maze.

Extra Credit Implementation

1. Store paths in a list
2. Search path list that contains start and goal nodes
3. Create a new path using the found path

Our implementation is in `remember.py` and is called by `search.py`

Credits

- Ammaar Muhammad Iqbal (ami76)
- Maury Johnson (mlj92)

- Shaan Kalola (spk103)
- Michael Parrilla (map565)