Amazing Maze Project
Group: Rabosa
Group Members: Benjamin Holland,
Benjamin Page, Xinran Xiao

AM Design Specification

1 OVERVIEW

Essentially broken up into two main parts: the AMStartup and the Avatar Client. AMStartup is in charge of initializing the maze and making the initial connection to the server. It is also responsible for generating the log file where the avatars will record their movements and launching each individual avatar client. The Avatar Client receives turn messages from the server and then calculates and delivers a move message until either the maze has been solved or an error message is received.

2 AMSTARTUP

2.1 INPUT

Command input

```
./AMStartup [NUMBER OF AVATARS] [DIFFICULTY] [HOSTNAME]
```

Example command input

```
./AMStartup 4 9 stratton.cs.dartmouth.edu
```

[NUMBER OF AVATARS] 4
Requirement: (int) This must be a positive integer, capped at the size of the maze.
Usage: AMStartup will inform the user is the number is not valid. This number will be sent to the server and will also determine how many Avatars to initialize.

[DIFFICULTY] 9
Requirement: (int) This must be an integer from 0 to 9.
Usage: AMStartup will inform the user if the difficulty is invalid. This number will be sent to the server to determine how big of a maze to create.

[HOSTNAME] stratton.cs.dartmouth.edu
Requirement: (char *) This must be a valid hostname for the client to connect to.
Usage: AMStartup will inform the user if it cannot connect to the server.

2.2 OUTPUT

AMStartup will create a new log file with the name: Amazing_$USER_N_D.log, where $USER is the current userID, N is the number of avatars, and D is the difficulty. The first line of the file will contain $USER, the maze port, and the date.

AMStartup will also create N avatar_client processes and those will run until an error or occurs or the end of the program is signaled.

2.3 DATA FLOW

Three main messages are flowed through this program: AM_INTIALIZE, AM_INITIALIZE_OK, and AM_INITIALIZE_FAIL.

On startup, AMStartup connects to the specified server and sends the AM_INITIALIZE message. If the server is ready to start, an AM_INITIALIZE_OK is sent back and upon receiving this message, AMStartup will create the avatar client(s).

2.4 DATA STRUCTURES

No major data structures are used for AMStartup. Variables that are kept during the lifespan of the program include an increment of the avatarIDNumber, the log file name, and server information.

2.5 PSEUDOCODE

```
// Input processing logic
(1) Command line processing on arguments
        Inform the user if arguments are not present
        if(invalid number of avatars || invalid difficulty || invalid hostname)
        {
                Inform user of usage and exit failed
        }

// Initialization of any variables or data structures
(2) Initialize()

// Connect to the server
(3) ConnectToServer(hostname)
        if (fail to connect)
        {
                inform user and exit failed
        }
```

```
// Build and send initialize message
(4) SendIntializeMessage()
        if (failed to send)
        {
                inform user and exit failed
        }

// Listen to the server for a message back. If OK, proceed, else quit.
if (receive AM_INITIALIZE_OK)
{
        logName = createLogFile()

        for (number of avatars)
        {
                createAvatarClient(avatarIDNumber, nAvatars, difficulty, IP address of
server, MazePort, logName)
        }
}
else
        inform user and exit failed
```

3 AVATAR_CLIENT

3.1 INPUT

Command input

```
./avatar_client [AVATAR ID] [TOTAL NUMBER OF AVATARS] [DIFFICULTY]
[SERVER IP] [MAZE PORT] [LOG FILENAME]
```

Example command input

```
./avatar_client 0 3 2 129.170.212.235 10829 Amazing_3_2.log
```

[AVATAR ID]
Requirement: (int) An integer generated by AMStartup between 0 and total number of avatars.
Usage: avatar_client will return an error message and a bad return code if the ID is invalid.

[TOTAL NUMBER OF AVATARS]
Requirement: (int) A positive integer, capped by maze size.
Usage: avatar_client will return an error message and a bad return code if this field is invalid.

[DIFFICULTY]
Requirement: (int) An integer between 0 and 9 inclusive.
Usage: avatar_client will return an error message and a bad return code if this field is invalid.

[SERVER IP]
Requirement: (char *) A valid IP address in a valid format (X.X.X.X)
Usage: avatar_client will return an error message and a bad return code if this field is invalid.

[MAZE PORT]
Requirement: (int) A valid port number returned by the server in the AM_INITIALIZE_OK message.
Usage: avatar_client will return an error message and a bad return code if this field is invalid.

[LOG FILENAME]
Requirement: (char *) A valid filename that points to a valid file that already exists.
Usage: avatar_client will return an error message and a bad return code if this field is invalid.

3.2 OUTPUT

The avatar_client will append to the log file made by AMStartup (see above). Each avatar_client will append its progress into the log file with the following format for each line:

```
[AVATAR ID NUMBER] [TURN #] [CURRENT POSITION] [NEXT MOVE]

Sample:        Avatar: 5  |  Turn 39  |  (5, 5)  |  Next move: East
```

If an avatar_client receives the AM_MAZE_SOLVED message, then *one of them* will write that message into the log file.

3.3 DATA FLOW

The main flows of data can be split into two sections: server-client and process-process.

On initialization of avatar_client, it will send the AM_AVATAR_READY message to the server. From this we start our main loop of interactions between the server and the client. For every iteration, the server will send an AM_AVATAR_TURN message to the client. From this, the avatar_client will calculate its next move and send back an AM_AVATAR_MOVE message until the maze is solved or some error occurs.

When calculating the next move, the client will use inter-process communication to build a common 'map' of the maze that will detect walls and record common paths. For each iteration, each avatar_client will consult this map in its decision-making process and add on to it as it travels along. From this, the data is shared between all the avatar_clients.

3.4 DATA STRUCTURES

LAST_MOVE: This structure keeps track of the recent data of each avatar. It keeps track of where the avatar was last time it went in the loop, and then remembers the last successful move that the avatar made. Each avatar client creates an array of type LAST_MOVE in order to keep track of all the moves of all the people. That way everyone know all the relevant information about everybody.

WALLS[MAX_DIM][MAX_DIM][4]: this array of type int keeps track of the walls in the maze. When an avatar attempts a move and is unsuccessful, then it writes a wall in that box in that direction. such as, if there is a northern wall in a box, then it writes walls[X][Y][M_NORTH]=1. It also writes an opposing wall in the adjacent box so that an avatar knows from the other side that there is a wall there without ever having to attempt an unsucessfull move.

CLUES[MAX_DIM][MAX_DIM][2]: this array of type int keeps track of the bread crumbs at all locations in a maze. It also keeps track of who originally left that bread crumb. This allows us to add the intelligence not to follow people who are following you, and it also allows us to know when the avatars are 'linked' meaning that they follow the same avatar.

Some kind of finite-state automaton to keep track of the states of the avatars while they travel so that they will remember the moves they previously made and not continuously run into walls.

Shared memory maze (2-D int array)

- This structure will be shared among all the avatars.
- Each avatar will update walls on this 2-D int array so that other avatars (if they end up in the same spot) will be able to save some moves by recognizing the wall.

3.5 Pseudocode

```
// Input processing logic
(1) Command line processing on arguments
       Inform the user if arguments are not present
       if (any of the arguments are invalid)
       {
              Inform user of usage and exit failed
       }
```

```
// Initialization of any variables or data structures
(2) Initialize()
        if (fail)
        {
                Inform user and exit failed
        }

// Connect to the server and send the ‘go’ signal
(3) ConnectToServer(ip)
(4) if (connect successful)
        {
                sendReadySignal()
        }

// Listen to the maze port
(5) Listen(mazePort)

// Main processing loop. For each received turn, make a move and send it back until
termination message.
(6) while (true)
        {
                if (AM_AVATAR_TURN)
                {
                        if (this avatar’s turn)
                        {
                                initialize AM_AVATAR_MOVE
                                move = calculate(avatar positions)
                                AM_AVATAR_MOVE.direction = move
                                sendMove(AM_AVATAR_MOVE)
                                continue;
                        }
                }
                else if (AM_NO_SUCH_AVATAR)
                {
                        log(error message);
                        break;
                }
                else if (AM_AVATAR_OUT_OF_TURN)
                {
                        log(error message);
                        break;
                }
                else if (AM_TOO_MANY_MOVES)
                {
                        log(error message);
                        break;
                }
                else if (AM_MAZE_SOLVED)
                                {
                                        log(maze_solved);
                                        break;
                                }
                                else
                                {
                                        continue;
                                }
                        }

        // Cleanup data structures, etc.
        (7) Cleanup()
```

4 Heuristics For Maze Solving

The avatars operate with a default move as a left hand wall follower. Once a wall is detected, it will not attempt a move into the wall. Since the basic move is so simple, we can keep track of all the avatars at once. By knowing their positions and knowing how their positions changed, and knowing that they all want to move to with the ranked preference of left, forward, right, back. Thus, if I'm avatar number one, and I see that avatar 2 went to the right, we keep track that there must be walls to the left and forward, and we will not attempt to move into those walls. And since we can keep track of the movement of each avatar, we are able to leave bread crumbs in each box where an avatar moves. The bread crumbs say who left the bread crumb and in which direction the breadcrumbs point. If an avatar comes upon a bread crumb then we follow it (unless the person who left the bread crumb is also following us, then we ignore it, and try to go left). That explains the motion of the bread crumbs. The next item of importance is how the avatars know when to stop. At the beginning of the maze, we set no finish line; rather, they all travel around in the process described above. Then we keep track of who is following who's bread crumbs. Using that information, we can know when everyone is following the same avatar ie the leader. (or following a person who is following a person who is following the leader). Once everyone is following the same leader, the leader comes to a stop, and all the other avatars keep moving until they catch up. The path to the leader is going to then be a topologically straight line path directly to the leader. Then the program comes to a stop once all the other avatars catch up.