Documentation

(1) **Input**

Command Input
```
./AM_Startup -n [Number of Avatars] –d [Difficulty] –h [Hostname]
```

Example command input

```
AM_Startup –n 5 –d 4 –h stowe.cs.dartmouth.edu
```

[Number of Avatars] 5
Requirement: The number of avatars must be between 1 and 10, inclusive.
Usage: The program will inform the user and quit if the number of avatars is not in range.

[Difficulty] 4
Requirement: The difficulty must be between 0 and, inclusive.
Usage: The program will inform the user and quit if the difficulty is not in range.

[Hostname] stowe.cs.dartmouth.edu
Requirement: The hostname must be valid. The maze should be able to send details for the creation of the maze (Maze height, Maze width, MazePort).
Usage: The program will inform the user if the URL is invalid or if a connection to the server could not be established. Otherwise, the program will run.

---

(2) **Output**

The program will create an ASCII maze showing the avatars (labeled with their avatar IDs), solving the maze. Depending on whether the maze is solved or not, the program will inform the user of the same.

As the program runs, a log file is created. The first line of the log file has the MazePort, AvatarID, and the date and time. The rest of the log file contains all the attempted moves that the Avatars have made, their current positions, and details when the maze is solved.

---

(3) **Data Flow**

1. When AM_Startup is run, it creates a client socket, which connects to the server.
2. Once successfully connected, the client sends an AM_INIT message that contains the number of avatars and difficulty.
3. After the message has been sent, the server responds with an AM_INIT_OK message.

4. The Mazeport, Maze Height and Maze Width are extracted from this AM_INIT_OK message.
5. AM_Startup now creates a log file with the name `AMAZING_$USER_nAvatars_Difficulty`. The first line of this file contains the Mazeport, Avatar ID, date and time.
6. Next, AM_Startup initializes all the memory that needs to be shared among the threads by allocating the relevant data structures to the heap. This includes initializing the list of last moves, the array of walls, and the array of visits of each avatar to each cell.
7. Finally, AM_Startup, allocates memory for the threads. For each thread, the AM_Args struct is initialized and its members (avatarId, nAvatars, difficulty, mazePort, ipAddress, logfile, width, height, walls, lastMoves) are given the appropriate values.
8. Each thread is created using the `pthread_create()` function which takes the `AM_Args` struct and the `new_amazing_client` which actually interacts with the server and sends avatar moves to the server.
9. The `new_amazing_client` sends the AM_AVATAR_READY via the MazePort.
10. While the message received from the server is not AM_MAZE_SOLVED or AM_TOO_MANY_MOVES or AM_SERVER_TIMEOUT:
    I. The client waits for AM_AVATAR_TURN from server
    II. Checks past moves and writes message about the move to the logfile.
    III. Then it draws the move using ASCII graphics through the draw() function.
    IV. The lastMoves list is updated.
    V. The array of visits is checked and updated
        a. If the avatar that last made a move already visited its current cell, a wall is added where it came from. If not, the avatar is added to the visits array at its current cell
    VI. The avatar with the current AvatarID makes its move and sends the AM_AVATAR_MOVE message. The move is made using the left hand traversal algorithm, checking for walls locally before sending a move to the server.
    VII. Once the maze is solved, avatar 0 writes the number of moves, hash, difficulty, and number of avatars to the logfile.

11. Once the maze is solved, the logfile is closed, the memory allocated in the heap is freed within AM_Startup.

---

(4) **Data Structures**

Walls: A 3 dimensional array that stores each maze-cell (the X,Y position) and direction (N,S,E,W) of a wall. Each cell contains a string with some subset of "NSEW" indicating the walls found around that cell. A wall may exist on one cell and not the adjacent cell (a one way wall) if an avatar is back-tracing and needs to indicate a dead end. A two way wall (the current cell and its adjacent cell) would not allow other avatars in the dead end to exit.

Last Moves: An array containing with space for each avatar, allowing constant time access to the most recent X and Y position of the avatar as well as the last successful move and the last attempted move. This is used to compute the next move and to determine if a move was made successfully.

Visits: A 3 dimensional array similar in structure to "walls", except each cell contains an nAvatars array of integers. At the slot in the array for each avatar is a 1 or a 0 to indicate if that avatar has visited the space or not, respectively. Since the mazes are perfect, and we are following around the left-hand wall, visiting a cell twice would indicate that an avatar is backing out a dead end. We block off dead ends as described in the walls data structure so that other avatars can avoid them.

---

(4) **Pseudocode**

1. Check that the arguments are valid
    I. Correct number of arguments
    II. Valid number of avatars
    III. Valid difficulty
    IV. Valid option characters
2. Use `gethostbyname` to acquire the server information.
3. Connect to the server using the 'sockfd' obtained from the `socket` command and the information obtained from `gethostbyname`.
4. Construct the AM_INIT message, send it to the server and then receive the message from the server.
5. If the message received is not AM_INIT_OK, exit out and inform the user.
6. Otherwise, acquire the maze information: Maze Height, Maze Width and MazePort from the AM_INIT_OK message.
7. Convert the maze information to the Host Byte Order using `ntohl` function.
8. Use `getenv` to get the user ID, calculate the length of the log file name, `calloc` memory for the name and then use `sprinf` to write the name to a char array.
9. Use `curtime` to get the time of the function call.
10. Allocate memory for the structs that will be shared among the threads : walls, lastMoves.
11. In a for loop, that loops as many number of times as there are number of avatars, initialize a struct that contains all the relevant information that each thread needs and then pass that information along with the new_amazing_client through the pthread_create() function.
12. Once all the threads have been created, use `pthread_join` for all the threads.
13. In new_amazing_client
    a. Check arguments are valid
    b. Open a socket to the server and send AM_AVATAR_READY
    c. While the maze is not solved
        i. Check that the received message is not an error

ii. Generate the next move based on information in walls, visits, and lastMoves if the TurnId matches the thread's avatar
iii. Send the next move to the server

14. Close the `logFile` and free up heap allocated memory.