

Amazing Project Design Specifications

Input

Command Input:

amazing [NUM_AVATARS] [DIFFICULTY] [HOSTNAME]

Example command input:

amazing 5 4 129.170.214.115

[NUM_AVATARS] 5

Requirement: Must be a number between 2 and AM_MAX_AVATARS

Usage: Client will inform user if the number is invalid

[DIFFICULTY] 4

Requirement: Must be a number between 0 and 9

Usage: Client will inform user if the difficulty is invalid

[HOSTNAME] 129.170.214.115

Requirement: Must be the ip address of flume.cs.dartmouth.edu

Usage: Client will inform user if the hostname is incorrect

Output

A maze of difficulty [DIFFICULTY] will be created and [NUMBER AVATARS] avatars will be randomly placed in the maze. ASCII Graphics will print out the maze along with the avatars and the paths that they have traveled after every move. A log file consisting of each avatars' moves and the maze outcome will be written to a file in the testing directory.

Data Flow

The server is constantly running and waiting for clients. AMStartup sends a message to the server with [NUMBER AVATARS] and [DIFFICULTY]. The server creates a maze of given difficulty and randomly places the avatars in the maze, sending a response message with a unique MazePort, MazeHeight and MazeWidth. AMStartup receives the message, stores the information and starts AMClient. AMClient starts up a thread for each avatar, passing the parameters from the message. Once ready, each of the threads will send a message to the server indicating they are ready.

Once the server has received a ready signal from each thread it will send messages to every thread with avatar locations and the current avatar to move. The thread whose turn it is will use our algorithm to determine the optimal move for the current avatar turn and a message will be sent back to the server with the move that the avatar wishes to make. The server will receive the message, determine if it is a valid move and if so it will update the avatar position. The server will send messages to every avatar again with updated avatar positions and the current avatar to move. This will continue until one of four conditions is met: A thread loses connection with the server, the maximum number of moves have been made, the time limit has been reached or the avatars have all reached the same position in the maze.

Throughout the entire process, the avatars will write information about their moves to the log file and when one of the four conditions have been reached, the outcome is written to the log file and any dynamically allocated memory is freed. The maze is redrawn to stdout for every turn.

Data Structures

MazeNode structs are used to represent each cell within the array. The maze is represented using a 2D array of MazeNodes.

```
typedef struct MazeNode
{
    int *visited; // number of times a location has been visited, index corresponds to avatar id
    int north, east, south, west; // keeps track of invalid direction
    int whoLast; // keeps track of which avatar was here last
    int lastDir; // keep track of last direction faced
}

/* XY-coordinate position */
typedef struct XYPos
{
    uint32_t x;
    uint32_t y;
}

/* Maze avatar */
typedef struct Avatar
{
    int solved;
    int fd;
    int AvatarId;
    int face;
    XYPos *pos;
}
```

```

/* ANSI color palette */
char* palette[11];
    palette[0]="\033[22;31m"; // red
    palette[1]="\033[1;93m"; // yellow
    palette[2]="\033[22;32m"; // green
    palette[3]="\033[22;34m"; // blue
    palette[4]="\033[22;36m"; // cyan
    palette[5]="\033[01;34m"; // light blue
    palette[6]="\033[22;35m"; // magenta
    palette[7]="\033[01;31m"; // light red
    palette[8]="\033[01;32m"; // light green
    palette[9]="\033[22;30m"; //black
    palette[10]="\033[0m"; // normal

```

Pseudocode:

AMStartup.c

```

    Check arguments
        If any invalid arguments, print error and exit
    Allocate memory for each avatar and the rendezvous point (average position of all avatars)
    Set up socket for startup
        Connect client to the socket
    Send init messages to the server
        send num avatars and difficulty to server
    Get init response from the server
        Proceed if init succeeded
    Set maze variables: mazeHeight, mazeWidth, mazePort
    Create maze with initializeMaze
    Call AMClient to begin avatar processes

```

AMClient.c

AMClient

```

    Begin new logfile
    For each new Avatar
        Create new avatar thread with function newAvatar
        Put each Avatar in array "avatars"
        Fail and print to log if thread cannot be created
        Clean up threads
    If maze has terminated, check the last message for exit status
        In each case, print status to log
    Free all allocated memory

```

newAvatar

- Allocate and initialize new Avatar struct

- Create a new socket for this avatar

 - If there is a problem creating the socket, print error and exit

- Set up the new socket and connect to network order

- Connect the client to the socket

 - If there is a problem connecting to the server, print error and exit

- Create ready message and send to server

- Create move message and server response message structs

- If all avatars have been successfully initialized, make FIRST move with avatar 0

 - Calculate rendezvous point in maze where all avatars will meet

 - Update new XYPos location for this avatar

 - Update MazeNode at this location with whoLast and lastDir fields

 - Update log with initial positions of all the avatars

 - Get the next move using the traversal heuristic and move

- While in infinite loop

 - If the server response is a move message, move the avatar

 - If the avatar was just moved

 - Mark the MazeNode that the avatar visits and update maze

 - Update whoLast and lastDir fields for the MazeNode

 - If this is the first move the avatar is making in the MazeNode

 - Visit the MazeNode for the first time

 - updating walls after the move has been made

 - If it is the current avatar's turn to move

 - Calculate optimal direction from traversal heuristic

 - Update log with new position of this avatar

 - Send move to the server and update maze

 - Else, maze is either solved or received error message

 - terminate (exit program via switches in AMClient function)

Traversal logic (in amazing.c)

- For each turn, calculate next move with **getMove**

 - First we check for a productive path with **isProductive**

 - Productive paths decrease the Manhattan Distance of the avatar's position to the rendezvous position

 - Manhattan distance is the absolute difference between the cartesian coordinates of two points, AKA taxicab geometry

 - **If a PP doesn't exist, we know that every possible move have been visited once by the avatar. We then arbitrarily choose a direction to traverse, and block off the twice visited MazeNode by calling addDeadEnd, which creates a "one-sided mirror".

Once all avatars are at the same location, AM_MAZE_SOLVED is received and the client exits.

Graphics: drawMaze (in AMClient.c)

Prints ASCII graphical representation of the maze after each turn in AMClient.c

Each avatar is assigned a color according to the “palette” array of ANSI color escape sequences

For each MazeNode

Print black border character if the MazeNode is on the border of the maze


Print an avatar (represented by a colored star) if the avatar is at this MazeNode

Print spaces if the MazeNode is unvisited

Print a UNICODE arrow facing the direction that the last avatar visiting it traveled, in the color of that avatar

Unicode characters used:

\u256D - top left corner of maze	┌
\u256E - top right corner of maze	┐
\u2570 - bottom left corner of maze	└
\u256F - bottom right corner of maze	┘
\u2502 - horizontal border of maze	
\u2500 - vertical border of maze	—

\u26B7 - rendezvous point smiley face 

\u2606 - avatar star ☆

\u2190 - west arrow →

\u2191 - north arrow ↑

\u2193 - south arrow ↓

\u2192 - east arrow ←