Spring | 14

Amazing Project Design Specification

Jason Feng, Chris Leech, & Jordan Hall

Includes Input, Output, Dataflow, Data Structures, Variables, and Psuedocode

CS50 Project Team - Aramon

1. Input

Startup Input: -h [HOSTNAME] -d [DIFFICULTY] -n [NUMBER_AVATARS]

-h hostname
-d difficulty
-n nAvatars

Example**:** startup -h pierce.cs.dartmouth.edu -d 3 -n 5

hostname (pierce.cs.dartmouth.edu):

This is the location of the server that will be handling the maze.

difficulty (3):

This is the difficulty of the maze, must be a value from 1 to 9. The maximum difficulty level is defined in a constant, in this case 9. Mazes 0-4 are fixed while mazes 5-9 are different every time they are generated.

nAvatars (5):

This is the number of avatars that will be put into the maze. The maximum number of avatars is defined in a constant, in this case 10.

Avatar Input: [Avatar ID] [Number of Avatars] [Difficulty Level] [IP Address] [Mazeport] [Filename for log file]

avatarID:

This is the avatar ID assigned to each avatar. The IDs start from 0 up until the number of avatars designated by the user in startup. The master always is the avatar with ID 0.

nAvatars (5):

This is the number of avatars that will be put into the maze. The maximum number of avatars is defined in a constant, in this case 10.

difficulty (3):

This is the difficulty of the maze, must be a value from 1 to 9. The maximum difficulty level is defined in a constant, in this case 9. Mazes 0-4 are fixed while mazes 5-9 are different every time they are generated.

IP Address:

This is the ip address obtained in startup. It is used to set up the connection between the avatar and the server.

Mazeport:

This is the mazeport used to set up the connection between the avatar and the server.

Filename:

This is the file name of the log file used to write information such as the time the program was ran, the user, the number of moves it took to succeed and the hash returned when the program was successful.

2. Output

Creates a log file with the Mazeport, User, and time the program started. The log also contains the hash and the number of moves it took to solve.

The program will write to a “picture” file a series of ASCII depictions of the graph, with the location of the avatars and the walls of the maze.

3. Data Flow

Startup:

We first get the hostname, difficulty, and number of avatars from the command line arguments. Then the next step is to establish the connection with the server. We want to setup the INIT message using the arguments provided. Then we want to get the ip address of the hostname and setup the socket we are communicating through.

After this initialization is completed, then we establish a connection using the socket and send the server the initialization message. Using the return message, we check if the connection is successful and that there is a successful INIT_OK message returned by the server.

From the INIT_OK message we need to get the mazeport that we are using and then we setup the filename of the log file we are writing to. The filename includes the user of the program, the difficulty, and the number of avatars.

The final process is to create the avatars. We need to create a command that includes the avatarID, the number of Avatars total, the difficulty, the ip address, the mazeport, and the filename. For each avatar we set up one command and send it.

We create the avatar processes and free memory.

Avatar:

The first step is to get all the varibles and initialize them from the input arguments, this includes all the information in the command sent from startup: the avatarID, the number of Avatars total, the difficulty, the ip address, the mazeport, and the filename.

Similar to how we established a connection in startup, we need to establish a connection for each avatar. We setup the socket and check for valid connections and then said and INIT message.

After the master is setup, the master needs to write the first line of the log which includes the filename of the log, the maze port we are using, the number of avatars, and the difficulty.

Then we use the move function to move the avatar until all the avatars find each other. For movement, we need to setup positions and directions for each avatar and also for the master. Each avatar will always know where the master was. After every move, we check the connection and check the message for errors. If the message is a MAZE_SOLVED type, then we can exit and write the hash to our log file. If a slave meets the master, then the slave will begin moving with the master. Until the maximum number of moves is reached, we will keep sending move messages based on the left hand and right hand maze traversal algorithms. The move algorithm is described below.

Left Hand Algorithm for the Slave - “Keep my left hand on the maze and walk”:

We first have an algorithm to determine the direction that the avatar is facing. We store the previous position of the avatar and the current position of the avatar. We compare the x, and y values of those two positions.

If the x coordinates are the same then we know the avatar moved along the y axis, and if the y coordinates are the same, then we know the avatar moved along the x axis. So given that we moved along one axis, then we compare the coordinates of the other axis.

If y of the current position is less than y of the previous position, then the avatar is going or facing south. If y of the current position is greater than y of the previous position, then the avatar is going or facing north. If x of the current position is less than x of the previous position, then the avatar is going or facing west. If y of the current position is greater than y of the previous position, then the avatar is going or facing east.

The slave will move using the left hand algorithm and the master will move using the right hand algorithm (which is the opposite of the left hand). The left hand algorithm logic is that if the avatar can move in the left direction, it will always move in that direction, if not then the other priorities in greatest to least order are move front, move to the right, and move backwards. The right hand algorithm uses the same logic except instead of moving left first, it will choose to move right first. Once an avatar reaches the maximum number of moves it's allocated, it will stop moving and return an error stating that it reached the maximum number of moves.

If a slave and master find each other by landing on the same position, they the slave will follow the master and also use the right hand algorithm instead of its original left hand algorithm.

4. Data Structures

XYPos – Position structure with an x and y coordinate.

Avatar – Maze avatar structure with a XYPos presenting the position of the avatar in the maze and an integer representing the id.

AM_Message – A message with a type defining the type of message was sent. Each message has various variables depending on its use.

Int *map: Shared memory representing the maze for graphics output. It is not convenient to declare it as a normal two-dimensional array using brackets. Thus, the array is represented as a single-dimensional array, where any element at (x, y) could be accessed at the index i calculated as follows: i = y*total_columns + x
Additionally, because the maze has to have walls in between spots, as well as a border, the representation maze in memory was around 4 times wider and around 2 times taller than the width and height received from the server. TranslateX() and translateY() take coordinates in "server world" and translate them to coordinates in "avatar world".

5. Variables

Startup:

```
int opt;                    // Commands
char *hostname;               // Name of the host we are using
uint32_t difficulty;          // Difficulty level
```

```
uint32_t nAvatars;            // Number of avatars
AM_Message initial;           // Message for initialization
struct hostent *ipFind;       // Ip address struct
char *ipAddress;              // Ip address
struct sockaddr_in servaddr;  // Addresse
int res;                      // Connection error
int out_socket;               // Socket
AM_Message okMessage;         // Successful init
uint32_t mazePort;            // Mazeport
char *fileName;               // Filename for log
char *command;                // Command to send for avatar
uint32_t type;                // Message type
```

Avatar:

```
int id;                       // Avatar ID
int nAvs;                     // Number of avatars in the maze
int difficulty;               // the difficulty of the maze
char *ip;                     // IP Address
int port;                     // MazePort
char *fileName;               // Filename of the log avatar writes to
struct sockaddr_in servaddr;  // For address
int out_socket;               // Socket
int res;                      // Check return for connection
```

6. Constants

Startup:

```
#define MAX_FILE 50             // Max file length for log file name
#define MAX_COMMAND 200         // Max command size for creating avatars
```

Avatar:

```
#define KEY 1993                // the shmget key
```

7. Psuedocode

Startup:

// Initialize variables such as hostname, ip address, command line arguments, commands, the name of the log file, etc.
// Parse command line arguments

```
// Setup init message for initialization
// Get ip address
// Create socket
// Get connection from server
        // Check if the connection is valid
// Send the init message to the server
        // Check return message from init
        // Check the message type for errors
// Get the mazeport
// Get the filename using the difficulty and number of avatars
// Create processes for each avatar
// Free memory
```

Avatars:

```
// Initialize variables similar to those in startup: Avatar ID, number of avatars, difficulty level,
maze port, name of log file, etc.
// Get time for log
// Get arguments from the message
// Setup the connection with the server
        // Check if the connection is valid
// Setup and sent the init message
// Have the master write the first line of the log file
// Move each avatar until the maze is solved. For moving:
        // Check the connection
        // Check the message for errors
        // Check if it is a MAZE_SOLVED message
                // If so then print the hash to the log
        // If we find the master, then start moving using the right hand algorithm
        // Check if it is the master's turn
                // Track location and direction of the master
        // Check the avatar ID
        // Get the current location of the avatar
        // Setup the next move
        // Check if we have moved from our previous position
                // If so get new direction
        // Get the direction using the left hand rule
        // We are moving right, then use right hand rule
        // Set up direction and send out the move message
        // Free memory and have error message checking
```

Move algorithm:

```
// Check the direction of the avatar
// Given the direction, check the number of unsuccessful moves already tried
// Return the set direction for that number of moves, based on the left hand algorithm
```

Ascii:

```
// Initialize Shared Memory
        // Create a mask depending on the provided width and height
        // Put white space in "2d array"
        // Add in walls with w
// When we are the master avatar, we update the map and print it to the console
```

8. Files

```
├── Makefile
├── README
├── results: logs of various runs will be kept here
│   ├── Amazing_cleech_0_5.log
│   ├── Amazing_cleech_1_5.log
│   ├── Amazing_cleech_2_5.log
│   ├── Amazing_cleech_3_5.log
│   ├── Amazing_cleech_4_5.log
│   ├── Amazing_cleech_5_5.log
│   ├── Amazing_cleech_6_5.log
│   ├── Amazing_cleech_7_5.log
│   └── Amazing_cleech_8_5.log
├── src
│   ├── amazing.h
│   ├── ascii.c
│   ├── ascii.h
│   ├── avatar.c
│   ├── avatar.h
│   ├── movealgorithm.c
│   ├── movealgorithm.h
│   ├── startup.c
│   └── startup.h
└── testing
    ├── avatar_test.c
    ├── BATS.sh
    ├── Makefile
    ├── startup_test.c
    ├── unitTest
    └── unitTest.c
```

9. Prototype Definitions

ascii.h:

```
/* initMap
 * @param width
 */
void initMap(int width, int height, int *map);

/* drawAscii - draws out the current position of the players in the maze
 *
 * @param turn - the turn message from the server
 * @param width - the width of the maze
 * @param height - the height of the maze
 * @param nAvs - how many avatars are in the maze
 *
 * This will use newlines to move output up the terminal,
 * then draw an ascii maze, with avatars in the maze represented by
 * their unique avatar id's
 *
 */
void drawAscii(int width, int height, int *map);

/*
 * translateX - this changes X values in the maze
 * world (for example, values we get back from the server)
 * into X values in the drawn world
 *
 * @x - the value we want to translate
 *
 */
int translateX(int x);

/*
 * translateY - this changes Y values in the maze
 * world (for example, values we get back from the server)
 * into Y values in the drawn world
 *
 * @y - the value we want to translate
 *
 */
int translateY(int y);
```

```
/*
 * getWallX - this function takes in an x value and
 * a direction, and figures out the X value of the
 * wall we just hit
 *
 * @x - our current x value
 * @direction - the direction we tried to go last move
 *
 *
 * returns the x value of the wall, pre-translated!
 */
int getWallX(int x, int direction);

/*
 * getWallY - this function takes in an Y value and
 * a direction, and figures out the Y value of the
 * wall we just hit
 *
 * @y - our current y value
 * @direction - the direction we tried to go last move
 *
 *
 * returns the y value of the wall, pre-translated!
 */
int getWallY(int y, int direction);
```

avatar.h

```
/* move
 *
 * @param id - The ID number of the avatar we want to move
 * @param out_socket - The socket we are writing to
 * @param fileName - The name of the log file
 * @param nAvs - The number of avatars
 *
 * Return: Returns failure for various errors, success when all the avatars
 * find each other.
 *
 * Psuedocode:
 * 1. Initalize variables by setting up positions, directions, moves, etc.
 * 2. Enter move loop - keep moving until all the avatars find each other
 * 3. Error checking within the loop for connection or errors from the server
 * 4. The avatars have left hand move, the masters have right hand move, if a
 * master finds an avatar, then the avatar will move with the master and also
 * use right hand rule.
 * 5. Free memory
 * 6. Error checking
 */
int move(int id, int out_socket, char *fileName, int nAvs, int height, int width);

/* initalizeLog
 * @param fileName - The name of the log file
 * @param mazeport - Mazeport we are using
 * @param nAvs - The number of avatars
 *
 * Return: Returns 0 on successful and 1 on failure
 *
 * Creates the log file for writing.
 */
int initializeLog(char *fileName, int mazeport, int nAvs, int diff);

/* printSuccess
 * @param fileName - The name of the log file
 * @param move - The successful move
 * @nAvs - The number of avatars
 *
 * Return: Returns 1 if successful and exits failure if it cannot open the file
 *
 * Writes the successful solved code to the log file as well as the number of
 * moves it took.
 */
int printSuccess(char *fileName, AM_Message move, int moves, int nAvs);
```

movealgorithm.h

```
/* getDirectionChange
 * @param prev - Previous position
 * @param curr - Current position
 *
 * Return - Returns the direction the avatar is facing
 *
 * Compares the previous and current position to determine which direction the
 * avatar is now facing.
 */
int getDirectionChange(XYPos *prev, XYPos *curr);
```

```
/* getNextMoveL
 * @param direction - the direction the avatar is facing
 * @param moves_attempted - the number of moves we've already tried
 *
 * Return - The direction the avatar should move.
 *
 * This algorithm will always move the avatar left if it can. Depending on what
 * direction the avatar is facing, "left" will be in a different direction
 * according to the defined North, South, East, West in amazing.h. We use the
 * current direction and the number of unsuccessful moves we've tried to
 * determine the direction we should move in. We will first try left, then
 * straight, then right, and finally backwards if none of those three work. This
 * algorithm is used by the slaves.
 */
int getNextMoveL(int direction, int moves_attempted);

/* getNextMoveR
 * @param direction - the direction the avatar is facing
 * @param movess_attempted - the number of moves we've already tried
 *
 * Return - The direction the avatar should move.
 *
 * This algorithm will always move the avatar right if it can. Depending on what
 * direction the avatar is facing, "right" will be in a different direction
 * according to the defined North, South, East, West in amazing.h. We use the
 * current direction and the number of unsuccessful moves we've tried to
 * determine the direction we should move in. We will first try right, then
 * straight, then left, and finally backwards if none of those three work. This
 * algorithm is used by the master. Once a master finds a slave, then the slave
 * will also use the right hand algorithm.
 */
int getNextMoveR(int direction, int moves_attempted);
```

startup.h

```
/*
 * convertToInt - converts a character buffer to an int
 * @str: the character buffer to inspect
 * @val: a pointer to the converted value
 *
 * Returns 0 if str is null or is a nul terminator. Otherwise, we check is the
 * str contains a non-numeric digit. If it doesn't, return 1 and val is set to
 * the the converted value. Else, return 0.
 *
 * Takes strings from messages and converts them to their numerical forms
 *
 */
int convertToInt(const char* const str, uint32_t *val);

/* getFileName
 * @param d - Difficulty level
 * @param n - Number of avatars
 *
 * Return - Returns a filename for the log file
 *
 * Constructs a filename for the log file using the specified difficulty level
 * and the number of avatars
 */
char *getFileName(int d, int n);

/* setUpCommand
 * @param id - Id of the avatar
 * @param nAvs - Number of avatars
 * @param dif - Difficulty level
 * @param ip - IP address
 * @param mport - Mazeport
 * @param file - fileName for log file
 * @param height - height of the maze
 * @param width - width of the maze
 *
 * return - Returns a command
 *
 * Creates a command using the specified parameters
 */
char *setUpCommand(int id, int nAvs, int dif, char *ip, int mport, char *file, int height, int
width);
```

10. Summary of Error Conditions Detected and Reported

Startup:

- Invalid Number of arguments with flags (7)
    - Non numeric arguments for -d and –n flags
    - A non –ndh flag
- Could not get host when attempting to get ipAddress

- Could not connect to host when connecting via socket
- Connection timed out
- Unsuccessful init due to bad difficulty level or too many avatars

Avatar:

- Failure to create shared memory – In the move function, we have to initialize shared memory in a map for the ASCII graphics to print
- Connection to server timed out – While moving, we check the AM_MESSAGE passed to us by the server incase it returns a connection timed out error
- If the AM_MESSAGE returned to us is an AM_ERROR we check the code for these possible errors:
    - AM_TOO_MANY_MOVES error message - maximum number of moves exceeded
    - AM_SERVER_DISK_QUOTA - Server disk quota reached
    - AM_SERVER_OUT_OF_MEM - Server out of memory
    - unknown error
- Failure to initialize log when the master writes the first line of the log
- Can't open file when printing to log when the maze is solved and the master attempts to write the hash code to the log file
- Incorrect number of arguments provided (9) when the avatar is initialized by startup
- Cannot connection to host when the avatar is sent a command by startup

11. Test Cases and Expected Results

Unit Testing: First we began with unit testing the functions of our 4 files ascii, avatar, move algorithm, and startup. We made sure that our functions worked properly and would properly exit when given an incorrect input. We also make sure that our functions would consider boundary cases. This is all implemented in unitTest.c

System Testing: Our final tests were testing startup, which would spawn avatar processes. We tested startup.c for its various command line inputs and made sure that it failed when given an incorrect input. Finally we tested for 3 sample runs and would produce 3 log files. This whole system testing can also be seen in our results folder that contains the logs of multiple runs on various difficulties.

Expected runtime: The heuristic we use for the solver is based on the fact that the path through a perfect maze can be deformed into a circle. Following the right-hand rule, the master moves one way along the circle. The slaves move the opposite way around the circle. If the path all the way through the maze is of length N, then in the worst-case, our avatars will solve the maze in N/2 moves, regardless of the number of avatars. Additionally, the expected number of moves, with N being the length of the path and M being the number of avatars, is: (M-1)N/M assuming the avatars are placed randomly in the maze.

12. Project Team

Here we will briefly go into the major focuses on each member of the group.

Jason Feng:

- Coding:
  - Implementing Left Hand and Right Hand Rule
  - Changing Directions
  - Move algorithm for avatar
  - Makefiles
- Testing:
  - Algorithm Debugging
  - Unit Testing
  - System Testing – BATS.sh
  - Valgrind memory testing
- Documentation:
  - Design Documentation

Chris Leech:

- Coding:
  - Startup
  - Sending Messages for avatar
  - Log functions for avatar
  - Move function for avatar
- Graphics:
  - ASCII Graphics
- Testing:
  - Algorithm Debugging
  - Graphics Debugging
  - Startup Debugging
- Documentation:
  - README

Jordan Hall:

- Graphics:
  - GTK/Cairo
  - ASCII
- Coding:
  - Startup – initialization of the server
  - Avatar – Sending messages and receiving messages