CS50 : Final Project Documentation

Cing is Believing

- **Min Hyung Kang**
- **Young Hoon Kang**
- **Chris Livingston**
- **Tae Ho Sung**

Content :

I. Design Specifications

In the following design specifications, we will discuss the input, output, data flow, data structures of the Maze. We will also include the Pseudo Code for main parts of the program.

1. Input

Command input : there should be three parameters

./maze –n [nAvatars] -d [Difficulty] –h[Hostname]

Example input :
./maze –n 3 –d 5 –h pierce.cs.dartmouth.edu

Note : the parameters do not need to be in order as long as the options (-n,-d,-h) are specified

[nAvatars] Indicates the number of Avatars to be placed inside the maze
Requirement : The value should be a positive integer between 2 and 10.
Usage : AMStartup takes in this parameter and sends a request to the server stating that we have this many avatars. It also creates specified amount of avatars by forking Avatar.c.

[Difficulty] Indicates the difficulty of the maze
Requirement : The value should be an integer between 0 and 9
Usage : AMStartup takes in this parameter and sends a request to the server asking for the maze of specified difficulty. It also passes on the information to Avatar.c to let each Avatar know what kind of peril they are dealing with.

[Hostname] Indicates the host that will run the maze
Requirement : For this specific maze program, the host should always be pierce.cs.dartmouth.edu
Usage : AMStartup takes in this parameter and sends a request to the server at this host, from which it will receive the port number to run the maze on. The information will then be passed on to each Avatar.

Avatar Input : there are 8 parameters

[Avatar ID] [nAvatars] [Difficulty] [IP] [Mazeport] [Filename] [Mazewidth] [Mazeheight]

Example input :
./avatar 0 5 3 129.170.213.200 33952 25 25

Note : This input is done by AMStartup.c, and hence defensive coding is not implemented in Avatar.c to check the parameters. It is assumed that the user will not change the content of AMStartup.c or Avatar.c.

[AvatarId] Indicates the ID number of an avatar
Requirement : The value should be an integer between 0 and 9
Usage : Each avatar is assigned a unique integer ID, specified by this parameter.

[nAvatars] Same as the input for AMStartup.c
[Difficulty] Same as the input for AMStartup.c

[IP] Indicates the IP address of the server
Requirement : The value should be valid IP of pierce.cs.dartmouth.edu
Usage : Each avatar uses this address to access the server

[Mazeport] Indicates the TCP/IP port number used to communicate with the maze
Requirement : Should be port number assigned by the server upon request
Usage : Each avatar uses this port to access the specific maze

[Filename] The logfile that each Avatar prints out to
Requirement : Should be the same file used by AMStartup to print out the log
Usage : Each avatar prints out its actions to this log

[Mazewidth] [Mazeheight] Indicates the size of the maze
Requirement : Should be size of the maze provided by the server

Usage : Each avatar uses this information to solve the maze

Graphics Input : there are 3 parameters

[MazeWidth] [MazeHeight] [Difficulty]

Example input :
./graphics 20 20 3

[Mazewidth] [Mazeheight] [Difficulty] Same as above

2. Output
-Result : The overall running of the program should be behind the scene. Each Avatar will search through the maze to find each other.

-Logfile : The log files will be created in the form of Amazing_$USER_N_D.log. The first line of the log will include the user, the mazeport, and the date and time. Then the log will contain the moves made by each avatar until the end of the maze.

-Graphics : Our team went on to implement the graphics using GTK+. It will show the moves of the avatars, how they close off the walls, etc. Also, it shows how many times the avatar has passed a cell by changing its color. It will announce that the maze was solved once all the avatars converge at one point.

3. Data Flow

The user calls on AMStartup with specified number of avatars, difficulty, and the host to connect to . AMStartup starts to write a log which is later concatenated by the avatars. AMStartup then sends a message to the host requesting a port one can use to connect to maze of specified difficulty. Upon recieving the message back from the port, AMStartup allocates a shared memory segment for each avatar to maintain information about an individual avatar's last move, which will later be used by the used to update each avatar's individual representation of the maze. It also initiates graphics program, which will show the movements of the Avatars. Then, it forks specified number of processes of Avatars, giving each of them enough information to live by themselves.
The avatars, once all of them are initiated, each sends a message to the server telling it they are ready. They also start appending to the log, stating their movements. Each of them have a graph, which will represent the maze, and linked list for each of other avatars, saving each path that other avatars took. Once all the avatars are ready, the server informs the avatars whose turn it is as well as the location of

each avatar. An Avatar only reacts when it is its turn to move, and it makes a move in the following way.
First, it accesses the shared memory initiated by AMStartup. Accessing it, it updates all the information of other avatars on its own graph. It then looks for a best next step. The best step involves finding its way to the nearest path of other avatar, whose path is not connected with the avatar's own. It also remembers the walls surrounding it and the ones others encountered through graph, so that it can make an informed decision. Also, when it meets a dead end and backtraces, it closes off the the dead-end so that we make the maze "smaller". Once it decides on its move, it updates its next move and current location to shared memory so that other avatars can know if there was a wall or not in that direction once the server returns the results. It then sends the move to the server, and next avavtar takes its turn.

While all this is being processed by each Avatar, graphics program depicts such movements. This done via message queues. Each avatar, before telling the server its move, sends a message to the graphics program, which receives the message and depending on its memory of the user's previous moves, updates its graph so that it can print out the current state of the maze.

When the avatars meet, the avatars exit as well as AMStartup, and the log file is closed. The graphics program states that the maze was solved, and the user can click to close the program.

4. Data structures

graph : graph is used to store the whole maze
- Graph : contains dynamic two dimensional array of graphnodes, representing the whole maze
- GraphNode : contains info about a single "cell" within the maze. It has information about what its neighbor cells are (as long as avatars explored them), and whether it was visited by an avatar or not

hashtable : hashtable is used for BFS once all the paths are connected
- HashTable : includes HashTableNodes, used for implementing BFS and finding short distance between paths.
- HashTableNode : Contains GraphNode so that a certain hash value points at a certain GraphNode

linkedList : used to store the paths that each avatar takes
- List : includes head and tail of a list, so that user can access the list
- ListNode : each ListNode is a graphNode, representing the path an avatar takes

move : used to store information about the last move made by an avatar
- x : the starting x position of the previous move made by an avatar
- y: the starting y position of the previous move made by an avatar
- dir : the direction of the previous move made by an avatar

The specific data structures and their implementations will be covered in section II.

5. Main Pseudocode

AMStartup

{

//check command line arguments

Check how many arguments there are, and check validity of input as specified in section (I.1) of this document.

//Extract the ip address for the host

//Sends AM_INIT message to the server, specifying the number of avatars and difficutly

//Checks for received message

=>If the message is AM_INIT_FAILED, prints out error message for the user and exits

=>If the message is AM_INIT_OK

-Extract Mazeport (TCP/IP port number) from the message

-Saves the returned Mazewidth and Mazeheight

// Create and allocate key array of each avatar's shared memory segment to shared memory

// Allocate an individual shared memory segment for each avatar to hold a move struct to represent the last move made by that avatar.

// Start graphics program

// Fork specified number of processes of avatar

// Exits

}

Avatar{

// Check all the parameter arguments

// Create necessary data structures

=> a graph to represent the maze

=> n number of linked lists for each avatar's path

=> the array of shared memory keys to reference the shared memory segments of each avatar

// Access all the shared memory

// Connect to the Mazeport

// Send a AM_INIT messgage to the server

```
        // Keep listening to the port until it is its turn

        // Upon its turn
                => Update the position of each avatar in the linked list
                => Update the current version of avatar's graph
                => Decides on a move
                => Update the positions on the graph
                => Send its move to via message queue to graphics
                => Makes the move

        // Exits upon receiving the message that the maze was solved
}

Graphics{
        // Decides on the dimensions of the image in the screen
        // Creates a window and places a drawing pane on it
        // Define personalized color scheme for the maze
        // Connect the signal to drawing area to allow redrawing
        // Starts a timer so that the maze is redrawn every certain interval
        // For every interval, performs the following actions
                => Waits for the message from avatars
                => Upon getting a message, decide on which component to draw on the maze
                => When the avatars are all on the same location, declare the maze was solved
}
```

II. Implementation Specifications

1. Data Structures
2. Function Prototypes
3. Constants

We define major data structures and important global variables.

1. Data structures.

(1) HashTable

```
// Nodes that compose the hashtable
typedef struct HashTableNode{
        GraphNode *node;       // point of hashvalue
        struct HashTableNode* next // pointer to next node
} HashTableNode;
```

```
// Avatar-specific data
typedef struct Hashtable{
        HashTableNode* Table[MAX_SLOT]; // Actual hashtable
} HashTable;
```

(2) Graph

```
// A node that shows information about a particular cell in maze
typedef struct GraphNode{
        XYPos *position; // Indicates the coordinate of this cell
        int visited; // Indicates if this node had been visited
        int current; // Indicates if an avatar is currently here
        struct GraphNode *north; // Indicates the what is on each direction
        struct GraphNode *east;
        struct GraphNode *south;
        struct GraphNode *west;
} GraphNode;

// A graph that include dynamic list of nodes, which is defined during runtime
typedef struct Graph{
        int width; // The size of the maze
        int height;
        GraphNode*** nodes; // Dynamic two dimensional array of pointers to the cells
} Graph;

//Graph extern constants
extern XYPos wallPosition; // all the walls' address are save as to be placed outside the maze
extern  GraphNode Wall; // Represents a wall for any cell
```

(3) Linked List

```
// Node that composes the list : each include a GraphNode
typedef struct ListNode{
        GraphNode* node; // A certain cell in maze saved in this node
        struct ListNode* next; // Pointer to the next node
} ListNode;

//Linked list which include information about a path an avatar took.
typedef struct List{
        ListNode* head; // the start and the end of the list
        ListNode* tail;
}
```

(4) Move

```
// Move struct with starting x,y pos and direction of move (direction int as defined in amazing.h)
typedef struct Move {
   int x;
   int y;
   int dir;
} Move;
```

Note: The above struct is maintained in a shared memroy segment unique to each avatar.

2. Function Prototypes

Classes in ./lib

(1) avatar_func.h : include functions used by Avatar to solve the maze

```
// Returns distance between two positions
double getDist(XYPos* one, XYPos* two);


// Looks for the closest unvisited position in the direction of closest unconnected path
XYPos *nearestUnvisitedPosition(XYpos *CurrentPosition, int AvatarID, int nAvatars, List **lists, int
*connected);

// Get the optimal direction towards the goal
int getDirection(XYPos *source, XYPos *goal);

// Add the number of times the avatar's been to adjacent locations to place priority on places it's been
to less frequently
int sumAdjacentVisited(XYPos *position, int degree, Graph *graph);

// Comes up with the best move for the avatar
int move(XYPos *currentPosition, XYPos *nearestPosition, Graph *graph);
```

(2) Graph

```
//Create a new graph
Graph* createGraph(int width, int height);
//Clears all the memory for Graph
int clearGraph(Graph *graph);

//Places the Avatar on the specfied location
int initialVisitNode(int x, int y, Graph *graph);

//Move to the node specified
```

```
int visitNode(GraphNode *prev, int direction, Graph *graph);

//Add a wall at given location
int addWall(GraphNode *prev, int direction, Graph *graph);

//Checks if it is a dead end (three sides are walls)
int isDeadEnd(GraphNode *node, Graph *graph);

//Blocks the dead end by creating a virtual wall
int blockDeadEnd(GraphNode *current, Graph *graph);
```

(3) hashTable

```
//Sets the size of the maze
void setSize(int w, int h)

//Compute the hashvalue for a given point peculiar to the HashTable
int computeHash(GraphNode *current)

//Add the given point to the Hash
int addToHash(GraphNode *current, GraphNode* before, HashTable* userTable)

//Just looks at the point which was added most recently for this current point
GraphNode* peekLastPoint (GraphNode *current, HashTable* userTable);

//Retrieves the last point which was added most recently for this current point
GraphNode *getPoint(GraphNode *current, HashTable *userTable);

//Destroys the table, freeing all the associated memory
int destroyTable(HashTable* userTable);
```

(4) myList

```
// Creates a new List
List *newList();

// Destroys the list
int destroyList(List *list)

//Push the given node onto the list
int push(GraphNode *node, List *list);

//Adds the given element to the head of the list
int addToHead(GraphNode *node, List *list);
```

```
//Peeks at the tail of the list
GraphNode* peekTail(List* list);

//Checks if the list is empty
int isEmpty(List *list)

//Return the head of the list
GraphNode *pop(List *list)

//Combines two lists into one
List *combine(List *list1, List list2);

//Checks if two lists are connected
int isConnected(List *currentList, List *otherLIst);
```

(5) shmAMStartup

```
//Creates the keys for avatars to use
int* createKeys(int nAvatars);

//Share the created keys with users/those who seek
int shareKeys(int* keys, int nAvatars);

//allocates the memory for each avatar
int allocShm(key_t *keys, size_t size, int nAvatars);

//deallocates the memory that was allocated
int deallocShm(int * keys, int nAvatars);

//deallocate the keys used by avatars
int deallocKeys(int nAvatars);
```

(6) shmAvatar

```
// returns array of ints that represents keys to memory segments of each individual avatar
int* getKeys(int nAvatars);

//Returns the x member of Move struct of shared memory
int getX(int avatarID, int* keys, int nAvatars);

//Returns the y member of Move struct of shared memory
int getY(int avatarID, int* keys, int nAvatars);

//Returns the direction of Move struct of shared memory
```

int getDir(int avatarID, int* keys, int nAvatars);

//Sets the X member of Move struct of shared memory
int setX(int newVal, int avatarID, int* keys, int nAvatars)

//Sets the Y member of Move struct of shared memory
int setY(int newVal, int avatarID, int* keys, int nAvatars)

//Sets the direction of Move struct of shared memory
int setDir(int newVal, int avatarID, int* keys, int nAvatars)

3. Constants

(1) Graph

extern XYPos wallPosition; // The position of the wall : defined as (-1,-1), which is out of the maze
extern GraphNode WALL; // the "WALL" that can represent every wall in every part of maze

III. Summary of Error Conditions Detected and Reported

For a list of error conditions that are handled by the program, refer to the next section.

Note the following assumptions :

-Number of Avatars : should be between 2 and 10
-Difficulty : should be between 0 and 9
-Host : should always be pierce.cs.dartmouth.edu

The user should not be running any other program than AMStartup. Avatar and Graphic are not intended to be run for user purpose by themselves.

To run the graphics, the user should be able to run GTK+ code in his/her computer. If the environment is not set up, the user will not be able to run it

IV. Test Cases

The following cases were tested :

- Number of arguments for AMStartup
- Difference in order of parameters for AMStartup
    - Note that validity of parameters were not tested for avatar or graphics, as they are not supposed to be run by the user

- Unit testing for each data structure
- Different maze sizes
- Different number of avatars

---- END OF DOCUMENTATION ---