The Maze Challenge Design

By: Core Dumped in a Maze
Larissa Chen, Benjamin Littlejohn, Benjamin Hannam

Overview and Requirements

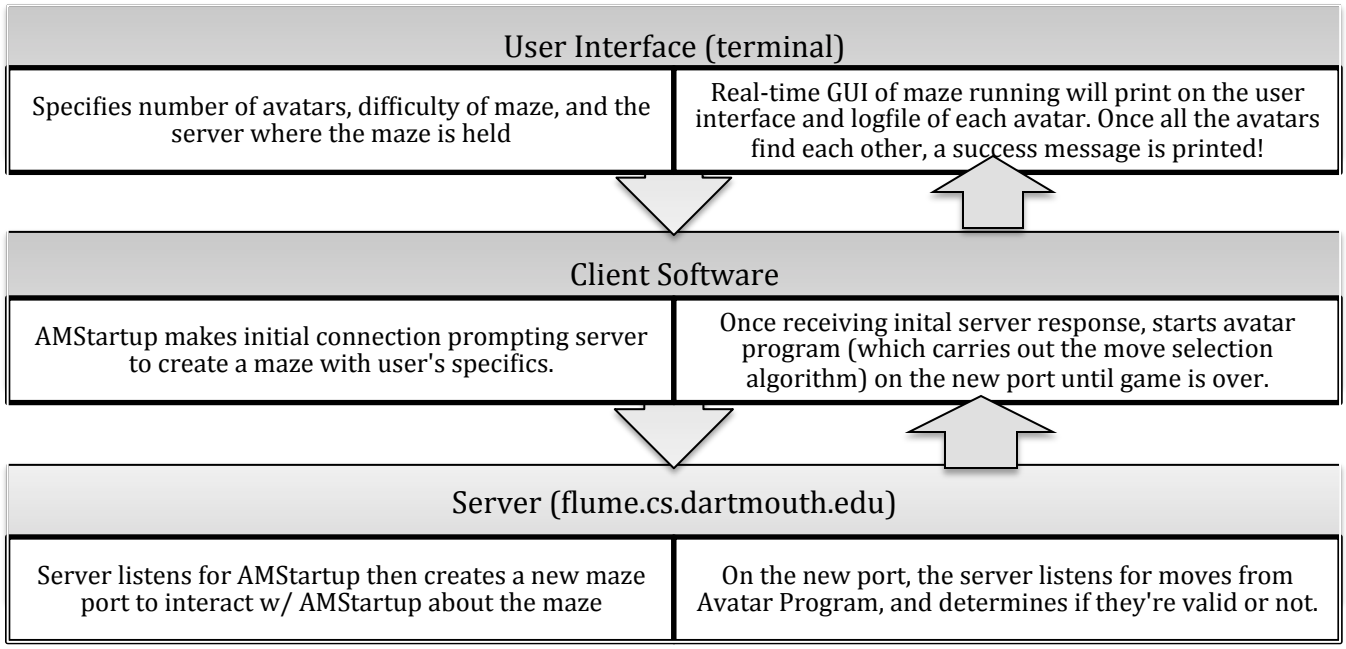
The maze challenge project is a client application that simulates a group of friends blindly being led into a maze and searching for each other. The client only initially knows that the maze is a perfect maze. This means only one path may exist from one point to any other, thus preventing any inaccessible sections, circular paths, and open areas.

The goal of the client application is to solve the challenge and bring together the friends in the fewest number of moves.

This design spec will cover the modules our team breaks the application into: an AMStartup script that drives the program flow and also an Avatar program that simulates move selection for each friend. This document also covers the GUI representation of the maze, the major data structures we utilize, and most importantly, the maze-solving heuristics optimized for this challenge.

Design Spec:

Maze Challenge Flowchart:



This is an overall flowchart displaying interactions between the 3 major players of the procedure: the user interface from the command-line, the client software that we write, and the server that holds the maze logistics on flume.cs.dartmouth.edu.

Description of Each Component and Module:

User Interface:

- Inputs: input arguments to provide to client software
  - –n n_avatars
  - –d difficulty
  - –h hostname
- Outputs: real-time GUI of maze running and logfile of format: Amazing_USER_n_d.log
  - Initial GUI displays an outline of the maze: known outer walls are marked off at the dimensions of the maze and corners of each "spot" are designated by a "+"
  - As avatars explore the maze, walls are dynamically inserted and marked off with a line "|" or " ---"
  - Avatars in the maze are represented by their corresponding unique id
  - When two avatars occupy the same spot, the lower number of the avatar is displayed
  - A dead spot, a spot that leads to a dead end, is marked off w/ a D and consequently its four walls are marked off closing off the spot
  - At the top of the maze, the last move is printed off e.g. "Avatar moved from x, y to x, y"
  - The GUI is also scalable and scales down when the initial width of the maze exceeds the available screen space

Client Software:

AMStartup Module (main):

- Parses input user arguments
  - Checks n_avatars is an int
  - Checks difficulty is an int
- Opens initial connection with server at hostname
- sends_init_message to server trying to create a maze with user's n_avatar and maze difficulty arguments
  - Maze sends an error if the user arguments are invalid according to the server-defined specs
  - Otherwise sends a success message with new maze_port where the maze will run
- creates n threads for n avatars
- Initializes global data structures that will be used by each avatar thread
  - A mazestruct_t to hold overall status of maze with known avatar and wall positions
- Passes control to thread_ops module also thought of as the "avatar program"
  - thread_ops controls the functionality of each avatar thread
- On return for thread_ops, prints any error return statuses and cleans up data structures

Thread_ops Module (helper module to handle each avatar thread) :

- Each thread avatar immediately opens connection with maze_port
- And carries out a function which:
  - Initializes an Avatar instance
  - Sends an AM_AVATAR_READY message to server
  - Enters solve_maze function, which in a loop…

    - Waits for its turn to make a move
    - On its turn figures the "best move" by running the heuristics we designed
    - Prints to the user the results of each move attempt and updates the maze graphic on the user interface to include a newly-found wall or open move etc.
    - Continues waiting and makings its turn until game is solved
    - Or if any other error occurs, the exit code is returned to the main function and main effectively parses the code and exits
- Upon a successful AM_MAZE_SOLVED message from the server, closes the threads and returns to AMStartup main function.
- If an error is returned, clean up still occurs and exit code contains the exit code

Data Structures:

mazestruct_t: The mazestruct_t defines the global maze variable available to each avatar. The mazestruct_t contains the real-time knowledge and status whose information is dynamically updated by the avatars.

mazestruct_t members include:

- a 2d array representation of all spots on the maze
- the maze width, the maze height
- number of avatars
- counter of completed moves
- array of every avatar's last move
- array of the leaders – array of the leader of every person
- array for each avatar containing who the avatar crossed path withs
- file pointer to logfile
- Boolean if maze is solved

spot_t is a struct that represents each individual spot on the maze. Its members include:

- four booleans that represent the walls for each cardinal directions. True indicating a wall and false if not
- a boolean representing if there is an avatar currently on the spot
- a boolean for if it is a dead spot, a dead spot is a spot that leads to a dead end
- a boolean for if the spot has been visited
- an integer array of size 10 which contains information about which avatars have visited the spot
- an array of size 10 that shows a list of which avatars are currently there

Methods of mazestruct_t:

- maze_new takes in the height and width of the maze and the number of avatars and allocs a new maze accordingly. maze_new allocates a new global maze in AMStartup before control is passed to the thread_ops module.
- Update methods: Everytime the avatar makes a move it updates the maze accordingly with….
    - place_avatar

    - insert_wall
    - insert_dead_spot
    - update_location
    - visited_spot
- The avatar can check the maze's status w/
    - check_wall
    - is_someone_adjacent
    - is_visited which checks if a spot has been visited
    - did_x_visit checks for if a specific avatar visit
- Methods to update who has crossed paths with whom
- maze_print which prints the GUI representation of the maze, and other print functions
- maze_delete which cleans up the maze structure and prints the number of moves to solve the maze.

Avatar:
avatar is a struct identified in amazing.h. Avatar contains…

- an integer that represents its unique id
- x, y position struct
- id of its leader (originally its own id, until avatars "come together" and it would be more convenient to just follow a nearby avatar)

Each avatar is initialized by each thread in thread_ops and interacts separately with the server directly trying different moves. Upon each move, the server sends the avatar feedback and updates the maze_struct accordingly by the feedback with the above methods mentioned in maze_struct. Each avatar decides where to move based on the maze-solving algorithm.

Maze-Solving Algorithm:

Our algorithm is computed in the following steps:

- On each avatar's move compute a static in-place evaluation of the four possible moves
- Loop through assigning a "move-value" for each possible move in each cardinal direction based on the following ranking system
    - 1 – if somebody's uncrossed with path exists in that spot
        - How to Detect: check that spot's visited array and crossed_with array
        - Implications of move: visit that space and effectively "cross paths" with any of the avatars whose path is on that spot
    - 2 - if it's an unvisited spot
        - How to Deteck: check that spot's visited Boolean
        - Implications of move: visit it and continue exploring
    - 3 – if somebody's already crossed with path exists in that spot
        - How to Detect: check that spot's visited array and crossed_with array
        - Implications of move: visit that space
    - 4 - if no other spots are available visit a space you have already visited marking off the current spot as dead and inserting walls
        - How to Detect: if the spot has already been visited by the same avatar
        - Implications of move: backtrack and mark the spot as dead

    - 5- there's a wall
        - Implications of move: Don't do it
- Choose a move based on the minimum score
- With this ranking system, we visit all possible non-dead spots "exploring" the maze
- When encountering somebody else's active path that ISN'T a dead spot or on a dead path, there are two options:
    - the higher ranked option is: explore the other way essentially extending that other avatar's clean, active path
    - or if no other moves exist: follow that avatar's path
- Keep "joining" or "extending" paths until all avatars are linked by one clear path void of any dead spots
    - Dead spots are marked off so effectively at the end there only exists one straightforward path
- Once all avatars are joined backtrack on the common path until the avatars all meet up at the common node

Crossing paths: To cross paths means to merge the paths of two different avatars. This happens when one avatar stumbles on a node in the maze that has already been visited. In doing so, cross paths is called with the avatar's id and the id of any other avatar who has visited that node. The crossed_with array for each avatar is "OR-ed" because any avatar A who crosses with avatar B who has also crossed with avatar C would mean each avatar A, B, C would contain crossed_with arrays of "A, B, C". Everytime cross_paths is called, its checked if all avatars have crossed_paths with each other, that is if any of the updated crossed_with arrays contains the id of every avatar. If so avatars can "avatar_unite" and "come together".

Coming Together: When all avatars are linked by the same path, they can backtrack on that path and meet at a common node. We do this by running the same static in-place evaluation except this time we change priorities. We ignore all unvisited nodes, making them the least priority, if an avatar is adjacent we join that avatar and that avatar becomes the "leader". If there are no adjacent avatars the point is to follow the path of visited nodes.

Leaders: Once assigned a leader, that avatar's move is then always defined as its leader's last move. Last moves are stored and retrieved from the last_move array in mazestruct.