Team Raboso's Amazing Project

Avalon McRae, Ted Owens, Daniel Chen

May 27, 2014

Contents

1 Project Overview

This document outlines the design specification for team Raboso - Daniel Chen, Avalon McRae, Theodore Owens. Our goal is to develop an efficient algorithm for gathering avatars with no knowledge of the layout of the maze to a common location. We will begin with a discussion of our maze algorithm, and then move towards the issues of storing the requisite data and how the avatars will share knowledge via a threaded approach.

2 Maze Algorithm

All avatars have shared information of a Maze structure, which is stored as a pointer to a 2D array, in the variable MazeSquare ***Mazeview (for details on the MazeSquare struct, see the Data Storage section). They determine their next move based on the following two items:

1. Right-Hand Rule Heuristic
2. Accumulated Knowledge of Maze

2.1 Right-Hand Rule Heuristic

The first portion of the move-choosing algorithm begins by calculating the average position of all of the avatars. This location is established as a central meet-up location which all Avatars attempt to converge on. The calculation for the *Goal* location is only made once.

For each avatar's first move, a manhattan distance heuristic algorithm will take the position of the avatar and the position of the goal, and will return the possible moves (N, S, E, W) in order of priority. Thus if the goal is mostly to the southwest, this algorithm might return South and the best move, West, as the next best, East as the third option, and North as the last option.

After the first move, the right-hand rule heuristic is used. It takes in the last previously successful movement as the direction that the avatar is facing, and then prioritizes potential movements as received from the shared maze view in terms of right, straight, left, then backwards. So, for example, if the direction is North, the priority for movements (assuming no walls have been established at this position) is East, North, West and South. This makes is so that an avatar never visited the same square twice.

2.2 Accumulated Knowledge of Maze

The second portion of the algorithm allows the avatars to build a joint knowledge of the maze together, and make more informed moves based on this knowledge. The avatar will prioritize moving in such a way as to appease the right-hand-rule heuristic, but it will not move in directions which it knows to be walled off (this knowledge of a wall comes from, at some point, attempting to move in that direction and finding that the avatars position did not change in the course of the move).

Further, if an Avatar determines that a location is a dead-end, closed off with three walls, it will move out of the position and declare that position as a dead-end so that it will never return to that position and all other avatars will know not to go there as well. If an avatar is surrounded by two walls and knows that a third option is a dead-end, it can also declare the *current* position as a dead-end as well.

By this process of declaring dead-ends, we hope to prune the maze of it's unhelpful branches and keep avatars from falling into the same traps.

3 Data Storage

In order for the above algorithm to succeed, we must have some way of storing the knowledge of the maze layout. This will be accomplished through the following:

1. Position Structure: each position (MazeSquare) holds a 4 element array of walls. Its indices are the provided amazing.h direction macros, and its value will always be initialized to 0 (no wall). If an avatar encounters a wall in that maze square, the appropriate wall element stores a 1 in the array, so that other avatars can use this information to avoid said wall.

```
typedef struct MazeSquare {
  int walls[M_NUM_DIRECTIONS];
  int occupied;
} MazeSquare;

  \item Avatar Init Structure: The Avatar Init structure tracks the initial information
       that each avatar needs when their thread is initialized.
  typedef struct AvatarInit {
          int AvatarId;
          int nAvatars;
          int Difficulty;
          char *serverIP;
          int MazePort;
          char *logFile;
  } AvatarInit;
```

2. Informed Avatar Structure: the Informed Avatar structure tracks and updates each avatar's latest information as they traverse the maze. It stores there last position and there current position (thus, if the two positions are the same, you can infer that they tried to move and hit a wall). It also stores their previousmove as an integer, using the provided macros from amazing.h. This assists the heuristic because an avatar will not want to prioritize going back to where they just came from.

```
    typedef struct InformedAvatar {
      XYPos lastposition;
      XYPos currentposition;
      int previousmove;     // Contains an int (represented by the amazing.h macros)
  } InformedAvatar;
```

4 Thread-Based Approached

A key design choice in the Amazing Project is whether to use threads or processes to control each independent avatar. The question in large part boils down to, what information needs to be conveyed between avatars, and what is the best way to convey that information? We have chosen a threaded approach to the Amazing Project as it allows multiple threads to access the same global variables. Thus all avatar-threads would have access to a common maze structure.

In *AMStartUp*, we create a thread for each avatar. Each thread receives as an input an *AvatarInit* structure which contains the information vital to starting the avatar thread (such as the Avatar's unique ID and the MazePort to connect to). The thread then unpacks the information, attempts to connect to the MazePort, and begins making turns.

In *AmStartUp*, after the threads have been started, we wait for all threads to return before exiting the main function thread with the *pThreadJoin* function. This also takes care of freeing the memory needed to create the thread.

5 Components and Methods

The section below provides a brief outline of the devision between the Maze-centered methods and variables and the Avatar-centered methods and variables.

5.1 Maze

Has information shared and edited by all avatars.

- Variables: a multi-dimensional array of position structures. Each position structure has variables which track the presence (or absence) of wall on all sides, as well as a dead-end variable. The maze will also contain variables for a goal location, as well as macros for directional movements (N, S, E, W).
- Methods:
  - createGoalLocation: run at beginning of program, takes in the x and y coordinates of each avatar, averages their values to get the goal location.
  - freeMaze: frees up memory of the maze.
  - addWall: takes the location and direction of the wall, and returns success or failure.
  - setDeadEnd: this item sets the dead end variable of a MazeSquare to 1 if it is found to be a dead end.
  - getPossibleMoves: returns possible movements (NSEW) from a location based on maze view generated so far.

5.2 Avatar

There is one avatar for each person in the maze, and the avatar is responsible for the distance heuristic, action determiner function, and function to determine what walls to add to.

- Methods:
  - heuristic: simple distance heuristic.
  - getNextMove: calls getPossibleMoves using current location, then sorts the moves by the heuristic.
- Variables:
  - current location (x, y)
  - previous location (x, y)
  - previous move (NSEW)
  - lastPositionDeadEnd boolean initialized to false

6 Graphics

For our maze graphics, we used the *nurses* library to show the maze with an ASCII representation. The graphics pull in information from the shared maze knowledge created by the avatar, as well as the positions of the avatar, and the position of the goal, to show the state of the maze. Initially, the maze is displayed without any walls. As avatars do hit walls, they are printed to the screen.

The graphics also allow for scalability, and center the maze properly based on the height and width of the terminal, as well as the size of the maze itself, to ensure that a maze of any type is well-displayed.

For every move, the graphics functions clear the display and redraws the maze. Each position of the maze is drawn independently, and checks which wall should be marked around it.

7 Results

Table 1: Number of Moves per Avatar Count and Difficulty Level

$Q_{Avatars}$	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
1	0	0	0	0	0	0	0	0	0	0
2	39	306	2	1130	627	1907	3705	3658	19227	28563
3	107	395	296	2612	2311	2899	4775	10381	17914	38127
4	142	414	631	3190	6152	6081	10712	8848	16681	NR
5	152	520	804	4068	7670	7519	11660	12039	21681	NR
6	182	618	964	4802	3253	5243	11002	18736	14522	NR
7	212	721	1469	5602	10773	12452	12007	17661	41788	NR
8	242	824	1678	6363	4329	8022	15854	22273	36337	NR
9	272	927	1887	7445	6515	9024	19254	27466	53984	78332
10	312	1020	2096	8763	7797	10026	13524	21534	80975	NR

Keep in mind that mazes that are generated above level 4 are random, so cross-avatar count comparisons for those difficulties should not be made. NR stands for Not Retrieved.