**Multifunctional Maze Generator**

**ES 111 Final Project**

**Abstract:**

The purpose of this project is to present a program that performs the following:

1. Generate a maze with user-defined dimensions (X by Y), giving the user 2 different maze generation algorithm choices:
	1. Depth-First Search
	2. Recursive Division
2. Output the maze as a plot and/or image
3. Give the user the option to save or load mazes
4. Using depth-first search, solve a maze that has been generated by this program, and output the solution as a plot or image
5. ~~(If time allows) Accept a maze image, solve the maze, and output the solution~~ (omitted due to time/complexity constraints)

**Introduction:**

Mazes exist in a wide variety of dimensions, tessellations, types of routing, and textures. In fact, there may very well be an infinite number of combinations of the above; the biggest limiting factors are imagination, and technological/mental capabilities (for example, try visualizing a 5-dimensional maze, let alone a 4-dimensional maze). For this reason, generalizations will be made in order to simplify things a bit.

The type of maze we are probably all most familiar with is the 2-dimensional maze, which can easily be printed onto a piece of paper and distributed for you and your nerdy/elderly friends to solve. Take things a step further, and you get a 3-dimensional maze, which essentially stacks a bunch of 2-d mazes on top of each other, adds vertical walls, and opens entrances between each level. As one may imagine, they are much more difficult to solve. 4th dimension and higher mazes take this concept a step further, but since we humans only perceive things in three dimensions, these mazes can be difficult to impossible to visualize, much less solve. It may be possible to visualize a 4-dimensional maze by imagining having the ability to stop time, move through a 3-dimensional maze, finding where connects to the next maze, then switching the time to the increment that the adjoining maze matches at that particular spot, or by imagining a set of 3-dimensional mazes with portals between each other at different points. Simply put, few are bold enough to venture into the territory of higher-dimension mazes.

Tessellation refers to the specific geometry, or shape of the maze and its constituent cells. As you can probably imagine, there are likely as many tessellation classes as you can imagine. A few types most people are familiar with include orthogonal (square), delta (triangle), and theta (circular) mazes, though are some pretty abstract mazes out there, such as crack mazes. Some example of different types of mazes can be seen below.

|  |  |
| --- | --- |
| Crack | Delta |
| Orthogonal | Theta |

There are several types of routing seen among mazes. The most prominent is the “perfect” maze, in which there is only one possible path from one cell to any other cell within the maze. With this type of maze, there is only one solution, and there are no inaccessible areas or “loops” in the maze. The second type of maze is the “braid” maze, which contains no dead ends. With this type of maze, there can be multiple solutions, and there is the possibility of going in circles within the maze. The sparse maze is effectively a sub-category of the above two types of mazes. It effectively leaves some areas inaccessible, so that some cells may have no path to other cells (e.g. a closed off room within the maze).

Independent from the above categories, texture is another term that can be used to describe a maze. It refers to specific, recognizable patterns that appear within the maze itself. For example, a maze may have many long corridors that have many corners, or it may have many short, but straight corridors. The maze may also have some sort of design embedded into it, such as symmetry.

MAZE ALGORITHMS

In the automated generation of mazes, there are a number of well-known algorithms that can be used. The algorithm used can have a significant effect on both the speed of the maze generation, and on several parameters of the maze, such as texture. By default, maze-generation algorithms create perfect mazes, and must be modified to produce specific types of routing, such as braided routing. One of the quickest and most common algorithms is depth-first search. A similar algorithm is recursive division. These two algorithms have been implemented in the final project.

Depth-first search is an algorithm that results in a perfect maze with long corridors and many corners. To generate the maze, the algorithm begins by thinking of the maze as an X by Y grid of cells with all walls up. Starting at a random cell, the algorithm checks its adjacent cells to make sure they are valid. If at least one valid cell exists, it picks one of them, and moves into that cell, breaking down walls in between. If no valid cells exist, backtrack until it finds a cell that has valid adjacent cells. Repeat this process until all cells have been visited. The process is summarized below.

* Start with an X by Y grid, with all walls up
* Pick a random cell in the X by Y grid, set it as visited
* Algorithm loop
	+ Check adjacent cells for valid cells, if one exists:
		- Move into a random valid cell
		- Break down walls between the two cells
		- Set new cell as visited
	+ If a valid cell does not exist:
		- Backtrack to most recently visited cell
	+ Run loop until all cells have been visited

\*Note that a valid cell is defined as a cell that has not been visited, and is within the X by Y grid

Recursive division looks at generating a maze a bit differently than depth-first search. Instead of burrowing through walls to build a maze, recursive division starts with an X by Y grid with only the edge walls up. It in effect builds a maze additively. The algorithm first picks a random valid cell on the maze (this time, a cell is defined as empty space bounded by walls – like a rectangular room). This cell is then divided by either a horizontal or a vertical wall, then a hole that is one grid unit wide is broken into that wall, and the cell is divided into two smaller cells. This process repeats until all cells are too small to be further divided (until each cell is only one unit tall or one unit wide, or both). The interesting aspect about this type of algorithm is that it results in a lot of bias – long, straight vertical or horizontal walls. What is just as interesting is that this bias can be controlled quite simply, by just changing the likeliness that the split direction will be horizontal or vertical. The overall process is summarized below:

* Start with an X by Y grid, with all walls down
* Algorithm Loop
	+ Pick a random valid cell (one that is greater than 1 grid unit in each dimension)
		- Decide whether to put up a wall vertically or horizontally
		- Randomly put up a wall in the cell, dividing the cell into two parts
		- Break a wall 1 grid unit wide through the wall, making a passage between the two cells
	+ Run loop until all cells are invalid (are only 1 grid unit wide in at least 1 dimension)

MAZE SOLVING ALGORITHM

As with maze generation algorithms, there are many algorithms that can be used to solve mazes. Some methods are inappropriate for use with mazes with multiple solutions, and some are flat-out inefficient with respect to other methods. Also, some methods of solving mazes are only appropriate if the algorithm has access to the entire maze at once, while others can be used even if none of the maze is known (think of a person walking through the maze).

Perhaps the simplest effective method of maze solving is the wall follower method. To solve the maze, the algorithm starts at a point, and follows the wall on either its left-hand-side or right-hand side until it reaches an exit. The obvious pitfall with this method is that it cannot reliably solve mazes with loops. If it starts at a point that eventually loops, then it will end up infinitely going in circles around that loop.

A more effective simple method of maze solving is the depth-first search method. It is extremely similar to the depth-first method of maze generation, with only a few minor changes. The algorithm starts at the start point, and looks for a valid adjacent cell to move into (in this case, an unvisited cell that does not have a wall separating the two cells). The algorithm then moves to the new cell, marks it as visited, and continues. If all adjacent cells are invalid, the algorithm backtracks until it finds a cell with valid adjacent cells. The algorithm is finished when a cell that is marked as the finish cell is found. What makes this method effective is that only the backtrack cell list is kept, so for a perfect maze, a solution will always be found, and that solution will only include the cells that lead to that solution (no moving through dead ends, etc.) This method of maze solving works also for mazes with multiple solutions, but would need some modification to find the shortest solution.

**Project Overview/Analysis**

The project itself can be broken up into several categories: the maze generator(s), the maze solver, the plot tools, and the master tool. The generation functions create the maze based on user input, the maze solver solves the maze that has a given start point and end point, the plot tools plot the maze or write the maze image, and the master tool brings each tool together and enables saving and loading of mazes. The following gives the flow of the functions:

* Master tool (mazetool.m)
	+ Maze creation tools
		- Depth-first search (createmazedf.m)
		- Recursive division (createmazerd.m)
	+ Maze solving tool
		- Depth-first search (solvemaze.m)
	+ Maze plotting tools
		- Plot unsolved maze (plotmaze.m)
		- Plot solved maze (plotsolvedmaze.m)
	+ Maze drawing tools (for image output)
		- Draw unsolved maze (drawmaze.m)
		- Draw solved maze (drawsolvedmaze.m)

The following is a break-down/more detailed overview of each function, and how each works:

function[]=mazetool(X, Y, t, b, p, s, f, i)

*Overview:*

This function ties every other function together. It has the capability to define the maze dimensions, the generation algorithm to use, the bias of the maze, whether and how to plot the maze, whether to solve the maze, whether to save the maze, and whether to save an image of the maze. This tool also has the capability to load a saved maze that is in text format.

*Notes:*

This function uses a number of loops, if/else, and switch statements to make decisions based on user input. Because the drawmaze/plotmaze functions do not have the capability to decide whether the maze has been solved, or whether to plot the solved or unsolved maze, the *mazetool* function does this. For image saving, note that most formats supported by MATLAB should be supported. Also note that when filenames are inputted into the function (e.g. mazetool(X, Y, 1, 1, 1, 1, ‘filename.txt’, ‘imagename.jpg’)), apostrophes must be used. However, if a prompt comes up for user input, no apostrophes should be used.

function []=createmazedf(X,Y)

*Overview:*

This function generates a maze of dimensions X by Y, based on user input. It uses the depth-first search method, and puts the maze grid in the form of a matrix with X\*Y+1 rows, and 4 columns. The first X\*Y rows store the information for each cell, in vertically progressing order (e.g. for a 10x10 maze, the first 10 rows of the matrix would contain the cell information for all of the cells in the first column, not the first row). Each column of each cell contains the cell wall information; 0=wall is down, 1=wall is up. The form is [North East South West]. This setup is perhaps a bit less convenient than the use of a Y by X by 4 matrix, but it makes debugging simpler. The last row of the matrix contains the length, width, start index, and end index information in the form [X, Y, Start Index, End Index].

*Notes:*

This function uses the global variable “main” to store the final maze matrix. If you wish to load this variable into the local workspace so you can use other functions with it, or otherwise manipulate it, use the command “global main” to pull the variable from the global workspace. Alternatively, you can define a variable to the output of the function (e.g. a=createmaze(X, Y)) This is the case for the other maze generation function and the maze solver function.

function [main]=createmazerd(X,Y, bias)

*Overview:*

This function is very similar to the usage and output of the previous function, with the exception of the “bias” parameter, which determines the bias of the maze towards long straight horizontal corridors vertically or horizontally. The lower the number (as it approaches 0), the more vertical bias it will have, while the greater the number (as it approaches 100 or above), the more horizontal bias it will have. The major difference in this function however, is not in usage, but rather, the algorithm used to generate the maze. Instead of walls being torn down, walls are built and cells are divided, as the description of recursive division maze algorithms stated.

*Notes:*

Compare this generation method to the recursive division method, and you will see some very interesting differences.

function [main]=solvemaze(main)

*Overview:*

This function solves a maze that is in the format of the above maze generators, and outputs the solution onto a similar matrix as those seen above, with the addition of a 5th column that states whether the cell is part of the solution (0=not part of solution, 1=part of solution). The method used for solving mazes is depth-first search.

*Notes:*

This maze solver should work for mazes that are tweaked to have multiple solutions, or for start/end points that are not on the edges of the maze.

function []=plotmaze(main) / function []=plotsolvedmaze(main)

*Overview:*

These functions plot the maze using MATLAB’s plotting tool. They essentially go through for loops that sweep across the X and Y coordinates of the maze, check to see what walls are up on each cell, and put those walls up as lines in the plot tool. The plotsolvedmaze function is the same as the first, except it first determines whether a solution exists (by determining if a fifth column exists on the maze matrix), and if it does, plotting a point in the center of each cell that is a solution cell.

*Notes:*

The plotmaze function will plot only the maze portion of a maze that contains a solution, so it can be used if you wish to plot a maze that has already been solved. Each of these functions can be used for mazes that have been saved to variables as well.

function []=drawmaze(main) / function []=drawsolvedmaze(main)

*Overview:*

These functions are very similar to the plotmaze functions, except they take things a step further and instead of writing lines to the plot window, they write pixels to an image matrix. Like the plotsolvedmaze function, the drawsolvedmaze function first checks to see if a solution exists, then it sweeps the maze matrix for solution cells and plots them.

*Notes:*

It is possible to change the resolution of the generated maze by tweaking the cell size (cs) and wall size (ws) in the function. Both of these parameters are in pixels, so changing these values will change the resolution of the maze accordingly.

Other Notes

If you save the output of any applicable function to a variable, you can then use that variable as an input to applicable functions (solvemaze, plotmaze, drawmaze). You can also save a maze variable to a file by using the command save(‘filename.txt’, ‘variable’, ‘-ascii’) . Note that the parameter ‘-ascii’ is important because it omits any non-numerical information from the saved file. A maze can likewise be loaded using the command outvariable=load(‘filename.txt’) where outvariable is the variable you would like to define your saved maze as. Finally, you can save the maze image as a picture file by using the command imwrite(image, ‘filename.ext’) where ‘image’ is your image matrix. An example of usage for maze matrix ‘a’ and filename ‘maze22.bmp’ would be imwrite(drawmaze(a), ‘test22.bmp’) .

**Conclusion**

The possibilities to continue or build off of this project are limitless. There are at least a dozen more maze generation algorithms and maze solving algorithms that could each be uniquely implemented and compared. In addition, there are many algorithms that could conceivably work that have not yet been thought of. The plotting tool could be tweaked and customized to allow custom colors and dimensions. The solution drawing tool could be modified to draw a pretty line through the solution rather than displaying a cluster of red blocks. A function could be written that takes the function and mass-produces n number of mazes, and saves only the top i number of mazes that have the longest solutions. As originally planned in the abstract, the solving and plotting tools developed in this project could be used in conjunction with a tool that takes a maze image and converts it into the standard maze matrix format used in this project.

Outside of this project, path-finding is very important in the world of artificial intelligence, and even streamlining large-scale processes. For robotic/AI technology to progress, technology needs to be implemented that enables robots to not only find their way around, but to find the quickest, most efficient way to get from point A to point B. Being able to solve a maze may be a good starting point to base that technology on. For large-scale processes such as driving a delivery truck around a crowded town, it is desirable that the quickest route is found. Traffic and speed limit differentiations of different routes ignored, this is essentially a problem that involves a maze with multiple solutions. The possibilities for implementation of the path-finding logic that maze generation and solving involve are very diverse.

**Appendix A – Bibliography**

*Think Labyrinth*: extensive information on types and classifications of mazes, as well as maze generation and solving algorithms:

<http://www.astrolog.org/labyrnth/algrithm.htm>

*Wikipedia*: information and examples of maze generation algorithms:

<http://en.wikipedia.org/wiki/Maze_generation_algorithm>

*Wikipedia*: information and examples of maze solving algorithms:

<http://en.wikipedia.org/wiki/Maze_solving_algorithm>

*The Buckblogs*: specific information on recursive division maze generation:

<http://weblog.jamisbuck.org/2011/1/12/maze-generation-recursive-division-algorithm>

*Mazeworks*: specific information on the basics of maze generation and the recursive division maze generation algorithm:

<http://www.mazeworks.com/mazegen/mazetut/index.htm>

*Anonymous:* MATLAB code for maze generation using depth-first search: was used as an example – some of the ideas were heavily “borrowed” upon, namely the matrix format for the maze:

<http://www.cs.cmu.edu/~jkubica/code/matlabMaze.html>

**Appendix B – Code**

**function[]=mazetool(X, Y, t, b, p, s, f, i)**

function[main]=mazetool(X, Y, t, b, p, s, f, i)

%Multifunctional maze tool:

%--X & Y are horizontal and vertical maze dimensions

%--t is the type of maze to generate: 0=Depth-First Search; 1=Recursive Division;

%2=load saved maze file

%--b is the bias (0-100) towards length or width where applicable; higher gives more width bias

%--p is the plot type: 0=no plot; 1=standard Matlab plot; 2=image plot

%--s determines whether to solve: 0=don't solve, 1=solve, 2=ask whether to solve later

%--f determines whether to save: 0=don't save, 1=ask whether to save later, 'filename.ext' saves

%--i determines whether to save image: 0=don't save, 1=ask whether to save later, 'filename.ext' saves.

%--c is the color (currently unavailable)

%Calls the following functions:

%--createmazedf

%--createmazerd

%--solvemaze

%--plotmaze

%--plotsolvedmaze

%--drawmaze

%--drawsolvedmaze

%===============================Initialize Variables============================

tf=0; %while loop control

%Set important global variables

global main; %main maze matrix

global image; % b&w image matrix (unsolved mazes)

global colorimage; %rgb image matrix (solved mazes)

%Generate maze

%while tf~=1

 switch t

 case 0

 createmazedf(X,Y);

 %tf=1;

 case 1

 createmazerd(X, Y, b);

 %tf=1;

 case 2

 mazein=input('Enter the mazefile.txt you wish to load. (no apostrophes):', 's');

 main=load(mazein);

 %case 3

 %mazein=input('Enter the maze variable you wish to load: ');

 %main=mazein;

 otherwise

 %ext=t(strfind(t,'.'):end); %Find extension if one exists

 %if length(ext)==1 %if there is an extension, input must be a file

 %main=load(t);

 %else %if there is no extension, input must be a variable

 % main=t;

 %end

 disp('You didn''t enter a valid maze type. Try again.')

 end

%end

%Solve maze

solvemaze(main); %Solves maze

tf=0; %reset while loop control

%Plot maze, decide whether to plot solved maze

while tf~=1

 %determine if/when user wanted to solve maze(0=no, 1=yes, 2=ask later)

 if p ~= 0 && p ~=1 && p ~=2 %if plot type is undefined

 p=input('Determine plot type (0=no plot, 1=std matlab plot, 2=img plot): ');

 else %plot type is defined

 tf=1; %stop while loop

 tf2=0; %new tf determinant

 while tf2~=1

 if s==0 % DON'T SOLVE

 switch p %plot type?

 case 0 %No plot

 tf2=1;

 case 1 %matlab plot

 plotmaze(main)

 tf2=1;

 case 2 %img plot

 drawmaze(main)

 tf2=1;

 end

 elseif s==1 % SOLVE MAZE

 switch p %plot type?

 case 0 %No plot

 tf2=1;

 case 1 %matlab plot

 plotsolvedmaze(main)

 tf2=1;

 case 2 %img plot

 drawsolvedmaze(main)

 tf2=1;

 end

 else %s NEEDS TO BE DEFINED

 switch p %plot type?

 case 0 %No plot

 break

 case 1 %matlab plot

 plotmaze(main)

 case 2 %img plot

 drawmaze(main)

 end

 s=input('Determine whether to solve maze (0=don''t solve, 1=solve): ');

 end

 end

 end

end

tf=0; %reset while loop control

%Save maze

switch f

 case 0

 %no action

 disp('Not saving txt...')

 case 1

 tf=input('Enter yourfilename.txt or leave blank if not saving: ', 's');

 if strcmp('', tf)~=1

 save(tf, 'main', '-ascii')

 disp('Saving txt...')

 end

 otherwise

 save(f, 'main', '-ascii')

 disp('Saving txt...')

end

%Save image

saveimg=0; %define variable for whether to save image

switch i

 case 0

 tf=('');

 %no action

 case 1

 tf=input('Enter yourfilename.ext or leave blank if not saving: ', 's');

 if strcmp(tf, '')==1

 saveimg=0;

 else

 saveimg=1;

 end

 otherwise

 tf=i;

 saveimg=1;

end

imtest=0; %new variable

%Make sure user entered a valid filename

while imtest~=1

 ext=tf(strfind(tf,'.'):end); %Find extension

 if saveimg==0

 imtest=1;

 disp('Not Saving Image...')

 elseif strcmp(ext, '.bmp')==1 || strcmp(ext, '.gif')==1 || strcmp(ext, '.pcx')==1 ||... %If extension is valid, save

 strcmp(ext, '.jpg')==1 || strcmp(ext, '.jpeg')==1 || strcmp(ext, '.pbm')==1 ||...

 strcmp(ext, '.png')==1 || strcmp(ext, '.pnm')==1 || strcmp(ext, '.ppm')==1 ||...

 strcmp(ext, '.ras')==1 || strcmp(ext, '.tif')==1 || strcmp(ext, '.tiff')==1 ||...

 strcmp(ext, '.xwd')==1 || strcmp(ext, '.gif')==1

 saveimg=1;

 disp('Saving Image...')

 imtest=1;

 else

 disp('You either entered an invalid extension, or didn''t enter one at all.')

 tf=input('Enter yourfilename.ext or leave blank if not saving: ', 's');

 if strcmp(tf, '')==1

 saveimg=0;

 imtest=1;

 disp('Not saving Image...')

 end

 end

end

if saveimg==1

 if p==2 %if user has already done an image plot

 switch s

 case 0

 imwrite(image,tf);

 case 1

 imwrite(colorimage,tf);

 end

 else

 switch s

 case 0

 drawmaze(main);

 imwrite(image,tf);

 case 1

 drawsolvedmaze(main);

 imwrite(colorimage,tf);

 end

 end

end

disp('All tasks completed.')

**function []=createmazedf(X,Y)**

%Author: Sean Coss

%Start Date: 3-23-2013

function [main]=createmazedf(X,Y)

%Takes user input (X,Y) and generates a maze of X by Y size, using the depth-first search algorithm.

%X=input('enter width:') %Debugging

%Y=input('enter height:') %Debugging

%X=5; %Debugging

%Y=5; %Debugging

global main; %set main as a global variable

%--------------------------------------Set up maze & other matrices-------------------------------------------------

main=ones(X\*Y, 4); %Creates the maze matrix; each row is a consecutive cell, each column is a wall direction [N E S W]

%sets=(1:X\*Y); %Creates indeces for the maze

visited=zeros(1,X\*Y); %Keeps track of which cells where visited; values will be set to 0 as they are visited

backtrack=[];

c\_dirmap=[1 2 3 4]; % defines directions of walls for current cell [N E S W]

v\_dirmap=[0 0 0 0]; % defines which directions are valid in respect to current cell [N E S W]

n\_dirmap=[3 4 1 2]; % defines directions of walls for new cell [S W N E]

%--------------------------------------------Set start cell----------------------------------------------------------

c\_ind=floor(rand\*(X\*Y))+1; %Select random index

start\_cell=c\_ind;

[cX,cY]=c\_coordinate(X,Y,c\_ind); %Get x and y coordinate of that index

%==============================================Run algorithm===============================================

while min(visited) <1

%for i=1:5

 %--------------------------------------------Check adjacent cells-----------------------------------------

 %North Cell:

 if cY-1==0 %Check if north contains a cell (is within maze)

 v\_dirmap(1)=0; %set north validity to 0 (invalid)

 else

 if visited(c\_ind-1)==1 %check if north cell has been visited

 v\_dirmap(1)=0; %set north validity to 0 (invalid)

 else

 v\_dirmap(1)=1; %set north validity to 1 (valid)

 end

 end

 %East Cells

 if cX+1>X %check if east contains a cell (is within maze)

 v\_dirmap(2)=0; %set east validity to 0 (invalid)

 else

 if visited(c\_ind+Y)==1 %check if east cell has been visited

 v\_dirmap(2)=0; %set east validity to 0 (invalid)

 else

 v\_dirmap(2)=1; %set east validity to 1 (valid)

 end

 end

 %South Cells

 if cY+1>Y %check if south contains a cell(is within maze)

 v\_dirmap(3)=0; %set south validity to 0 (invalid)

 else

 if visited(c\_ind+1)==1 %check if south cell has been visited

 v\_dirmap(3)=0; %set south validity to 0 (invalid)

 else

 v\_dirmap(3)=1; %set south validity to 1 (valid)

 end

 end

 %West Cells

 if cX-1==0 %check if west contains a cell ( is within maze)

 v\_dirmap(4)=0; %set west validity to 0 (invalid)

 else

 if visited(c\_ind-Y)==1 %check if west cell has been visited

 v\_dirmap(4)=0; %set west validity to 0 (invalid)

 else

 v\_dirmap(4)=1; %set west validity to 1 (valid)

 end

 end

 %--------------------------------------------------Backtrack---------------------------------------------------

 if sum(v\_dirmap)==0 %If there are no valid directions

 c\_ind=backtrack(end); %set the current cell to the last cell on the backtrack list

 [cX,cY]=c\_coordinate(X,Y,c\_ind); %Get x and y coordinate of that index

 backtrack=backtrack(1:(end-1)); %remove the last backtrack entry from the list

 %----------------------------------------------Move to new cell------------------------------------------------------

 else

 %Pick direction & determine if direction is valid

 v\_dir=0; % initialize variable

 while v\_dir ~=1 %run this until a valid direction is found

 dir=floor(rand\*4)+1;

 if v\_dirmap(dir)==1

 v\_dir=1;

 end

 end

 %Set current cell as visited, and tear down wall before moving into new cell

 visited(c\_ind)=1;

 main(c\_ind, dir)=0;

 %Set new cell coordinates and index a current cell coordinates and index

 switch dir

 case 1 %North

 c\_ind=c\_ind-1;

 cY=cY-1;

 %cX=cX;

 case 2 %East

 c\_ind=c\_ind+Y;

 %cY=cY;

 cX=cX+1;

 case 3 %South

 c\_ind=c\_ind+1;

 cY=cY+1;

 %cX=cX;

 case 4 %West

 c\_ind=c\_ind-Y;

 %cY=cY;

 cX=cX-1;

 end

 %Set new cell as visited, tear down wall, and record cell on the backtrack record

 visited(c\_ind)=1; %new cell is visited

 diri=n\_dirmap(dir); %inverts direction so correct wall will be knocked down

 main(c\_ind, diri)=0; %knock down wall in the direction that you came from

 backtrack=[backtrack, c\_ind];

 end

end

%--------------------------------------------------Set start and end points--------------------------------------------------------

open\_ind=floor(rand\*(X\*Y))+1; %choose random index

open\_dir=round(rand); % decide whether to go left or up

[cX,cY]=c\_coordinate(X,Y,open\_ind); %define current coordinates

%Choose start & end points

switch open\_dir

 case 0 %use x-walls

 %Start coordinate

 cY=1; %start wall

 %cX=cX

 s\_ind=(cX-1)\*Y+cY; %set start index

 main(s\_ind,1)=0; %open north wall of start index

 %End coordinate

 cX=floor(rand\*X)+1; % end wall

 cY=Y;

 e\_ind=(cX-1)\*Y+cY; % set end index

 main(e\_ind, 3)=0; %open south wall of end index

 case 1 % use y-walls

 %Start coordinate

 cX=1;

 %cY=cY

 s\_ind=(cX-1)\*Y+cY;

 main(s\_ind,4)=0; % open west wall of start index

 %End coordinate

 cY=floor(rand\*Y)+1;

 cX=X;

 e\_ind=(cX-1)\*Y+cY;

 main(e\_ind,2)=0; %open east wall of end index

end

main((end+1),1:4)=[X,Y,s\_ind,e\_ind]; %Add x and y-values to the end of the maze matrix; also add start and end indeces

%main %debugger use

**function [main]=createmazerd(X,Y, bias)**

%Author: Sean Coss

%Start Date: 4-9-13

function [main]=createmazerd(X,Y, bias);

%Creates a maze of X\*Y dimensions based on user input

%Recursive division method of maze generation:

%-Picks a 'cell' within the maze, and puts a dividing wall in between,

%vertical or horizontal (semi-randomly chosen, with a slight bias towards

%keeping cells somewhat square). Repeats dividing process until cell is at

%its minimum size - 1 unit thick

%===============================Initial Setup==============================

%X=input('Enter width for maze: '); %number of cells wide

%Y=input('Enter hidth for maze: '); %number of cells high

%bias=input('Enter bias (0-100), higher is more X-bias: ');

biasY=bias/50;

global main;

%array in form of {[1=valid, 0=invalid], [X0, Xf, Y0, Yf]}

matrix=[1, 1, X+1, 1, Y+1]; %Define initial array of cells - contains cell which consists of entire maze; adds 1 for terms of walls instead of cells

C\_ind=1; %current index for array

validcells=sum(matrix(1:end, 1)); %set initual number of valid cells to 1 - the initial value defined by the matrix itself

%iscellvalid=0; %initially set validity of current cell to 0 (you technically don't know if it's valid)

%============================Main (standard maze format) Setup================

main=zeros(X\*Y, 4); %Creates the maze matrix with all walls down; each row is a consecutive cell, each column is a wall direction [N E S W]

%sets=(1:X\*Y); %Creates indeces for the maze

c\_dirmap=[1 2 3 4]; % defines directions of walls for current cell [N E S W]

n\_dirmap=[3 4 1 2]; % defines directions of walls for new cell [S W N E]

%Put walls up around perimeter

main(1:Y, 4)=1; %Set westmost walls to up

%main(end-(Y-1):end+1, 2)=1 %debugging

main(end-(Y-1):end, 2)=1; %Set eastmost walls to up

for i=1:X

 c\_ind=(i-1)\*Y+1; %Set c\_ind to northmost cell of current column

 main(c\_ind, 1)=1; %Set northmost wall to up

 c\_ind=(i\*Y); %Set c\_ind to southmost cell of current column

 main(c\_ind, 3)=1; %Set southmost wall to up

end

%---------------------------------Debugging------------------------------------

counter=0;

validcounter=0;

%============================Run Algorithm=================================

while sum(validcells)>0 %While there are still valid cells

 %---------------------Initial values for algorithm------------------------

 matrixsize=size(matrix); %Get matrix dimensions

 cells=matrixsize(1); %Get number of cells, which is the number of rows in the matrix

 %Pick random cell until a valid one is found

 iscellvalid=0; %initially set validity of current cell to 0 (you technically don't know if it's valid)

 while iscellvalid ~=1

 C\_ind=ceil(rand\*cells); %choose a random cell to try to build a wall in

 iscellvalid=matrix(C\_ind, 1); %Set iscellvalid to validity of indexed cell (1=valid)

 %validcounter=validcounter+1 %debugging

 %input('valid cell segment. Continue?') %debugging

 end

 %C\_ind %debugging

 %Define reference points for current cell

 sY=(matrix(C\_ind, 4)); %Get start Y wall position

 sX=(matrix(C\_ind, 2)); %Get start X wall position

 cY=(matrix(C\_ind, 5)-matrix(C\_ind, 4)+1); %Get the number of row walls in the current cell

 cX=(matrix(C\_ind, 3)-matrix(C\_ind, 2)+1); %Get number of column walls in the current cell

 eY=(matrix(C\_ind, 5)); %Get end Y wall position

 eX=(matrix(C\_ind, 3)); %Get end X wall position

 %------Determine whether to split cell vertically or horizontally------

 if biasY\*cY>cX

 cutdir=0; %set the cut direction to horizontal

 %Add code for probability of randomness vs. horizontal

 elseif cX>biasY\*cY

 cutdir=1; %set the cut direction to vertical

 %Add code for probability of randomness vs. vertical

 else

 cutdir=round(rand); %Randomly choose the cut direction

 end

 %============Cut the cell, add a new cell, and remap cell===============

 %Seperate cut case for vertical than for horizontal

 switch cutdir

 %-------------------------Horizontal Cut-----------------------------

 case 0

 %disp('Horizontal Cut') %debugging

 divY=(sY+1)+floor(rand\*(cY-2)); %Choose a random point along the Y within the cell to make the cut

 %Decide if divide has caused any cells to become invalid for further splitting (1 cell wide)

 %upper portion (start to divY)

 if divY-sY==1

 iscurrentvalid=0;

 else

 iscurrentvalid=1;

 end

 %lower portion (divY to end)

 if eY-divY==1

 isnewvalid=0;

 else

 isnewvalid=1;

 end

 %Split matrix up into subcells and re-define parameters

 matrix(C\_ind, 1:5)=[iscurrentvalid, sX, eX, sY, divY]; %Redefine current cell as the top of the original cell from the split

 matrix(end+1, 1:5)=[isnewvalid, sX, eX, divY, eY]; %Define the new cell as the bottom of the original cell from the split

 %Decide if

 %Decide which wall to break down (or in this case, not put up)

 brkXwall= floor(rand\*(cX-1))+sX; %Random x-position between the start wall and end wall of original cell

 %Put up wall in upper and lower unit cells of the dividing line

 for i=sX:(eX-1) %For colums starting to the right of the leftmost wall, and end ending at columns to the left of the rightmost wall

 upper\_ind=(i-1)\*Y+(divY-1); %set the upper index to the curent column, and the cell just above the div line

 lower\_ind=(i-1)\*Y+(divY); %set the lower index to the current column, and the cell just below the div line

 if i~=brkXwall %Put walls up if the current x-position is not the wall to take down

 main(upper\_ind, 3)=1; % Put up south wall of upper cell

 main(lower\_ind, 1)=1; %Put up north wall of lower cell

 end

 end

 %-------------------------Vertical Cut-----------------------------

 case 1 % if cut direction is vertical

 %disp('Vertical Cut') %debugging

 divX=(sX+1)+floor(rand\*(cX-2)); %Choose random point along the X within the cell to make the cut

 %Decide if divide has caused any cells to become invalid for further splitting (1 cell wide)

 %left portion (start to divY)

 if divX-sX==1

 iscurrentvalid=0;

 else

 iscurrentvalid=1;

 end

 %Right portion (divX to end)

 if eX-divX==1

 isnewvalid=0;

 else

 isnewvalid=1;

 end

 %Split matrix into subcells and redefine parameters

 matrix(C\_ind, 1:5)=[iscurrentvalid, sX, divX, sY, eY]; %Redefine current cell as left side of original cell from split

 matrix(end+1, 1:5)=[isnewvalid, divX, eX, sY, eY]; %Define new cell as right side of original cell from split

 %Decide which wall to break down (not put up in this case)

 brkYwall=floor(rand\*(cY-1))+sY; %Random y-position between the start wall and end wall of original cell

 %brkYwall=5; %debugging

 %Put up wall in left and right unit cells of the dividing line

 for i=sY:(eY-1) %For rows starting below the upper wall and above the bottom wall for original cell

 left\_ind=(Y\*(divX-2)+i);

 right\_ind=(Y\*(divX-1)+i);

 if i~= brkYwall %Put walls up if the current y-position is not the wall to take down

 main(left\_ind, 2)=1; %Put up east wall of left cell

 main(right\_ind, 4)=1; %Put up west wall of right cell

 end

 end

 end %end switch statement

 %------------------------------------------Finish up algorithm - reset validcells----------------------------------------------

 validcells=sum(matrix(1:end, 1)); %Update number of valid cells

 %matrix %debugging

 %counter=counter+1 %debugging

 %input('continue?') %debugging

end %end while loop

%=============================Final Stuff==================================

%--------------------------------------------------Set start and end points--------------------------------------------------------

%input('While Loop complete. Continue?') %debugging

open\_ind=floor(rand\*(X\*Y))+1; %choose random index

open\_dir=round(rand); % decide whether to go left or up

[cx,cy]=c\_coordinate(X,Y,open\_ind); %define current coordinates

%Choose start & end points

switch open\_dir

 case 0 %use x-walls

 %Start coordinate

 cy=1; %start wall

 %cx=cx

 s\_ind=(cx-1)\*Y+cy; %set start index

 main(s\_ind,1)=0; %open north wall of start index

 %End coordinate

 cx=floor(rand\*X)+1; % end wall

 cy=Y;

 e\_ind=(cx-1)\*Y+cy; % set end index

 main(e\_ind, 3)=0; %open south wall of end index

 case 1 % use y-walls

 %Start coordinate

 cx=1;

 %cy=cy

 s\_ind=(cx-1)\*Y+cy;

 main(s\_ind,4)=0; % open west wall of start index

 %End coordinate

 cy=floor(rand\*Y)+1;

 cx=X;

 e\_ind=(cx-1)\*Y+cy;

 main(e\_ind,2)=0; %open east wall of end index

end

%Add X and Y coordinates to main matrix

main((end+1),1:4)=[X,Y,s\_ind,e\_ind]; %Add x and y-values to the end of the maze matrix; also add start and end indeces

**function [main]=solvemaze(main)**

%Author: Sean Coss

%Start Date: 4-3-13

function [main]=solvemaze(main);

%Takes input (main) and solves it, expanding main matrix

global main;

%Break up input maze matrix

X=main(end,1); %Pull width

Y=main(end,2); %Pull height

s\_ind=main(end,3); %Pull start cell index

e\_ind=main(end,4); %Pull start cell index

maze=main((1:end-1), (1:4)); %Pull maze matrix

mazesize=size(maze);

maze=[maze, zeros(mazesize(1),1)]; %Add column to maze matrix to serve as solution matrix

v\_dirmap=[0 0 0 0]; % defines which directions are valid in respect to current cell [N E S W]

%Set up initial information

c\_ind=s\_ind; %Set current index to start index

[cX,cY]=c\_coordinate(X,Y,c\_ind); %Get x and y coordinate of that index

visited=zeros(X\*Y, 1);

backtrack=[c\_ind]; %Add first cell to backtrack path

visited(c\_ind)=1; %Add first cell to list of visited cells

%====================================================Solve Maze================================================

while c\_ind~=e\_ind

 %---------------------------------------------Check Adjacent cells------------------------------------------

 %North Cell:

 if cY-1==0 %Check if north contains a cell (is within maze)

 v\_dirmap(1)=0;

 elseif maze(c\_ind, 1)==1 ||... %Check if north cell has a wall up

 visited(c\_ind-1)==1 %Check if north cell has been visited

 v\_dirmap(1)=0;

 elseif c\_ind-1==e\_ind %Check if north cell is solution

 c\_ind=e\_ind; %set current cell to end cell

 backtrack=[backtrack, c\_ind]; %add to backtrack path

 break %break the loop

 else %If cell isn't invalid, and isn't end cell, it is valid

 v\_dirmap(1)=1;

 end

 %East Cell:

 if cX+1>X %Check if east contains a cell (is within maze)

 v\_dirmap(2)=0;

 elseif maze(c\_ind, 2)==1 ||... %Check if east cell has a wall up

 visited(c\_ind+Y)==1 %Check if east cell has been visited

 v\_dirmap(2)=0;

 elseif c\_ind+Y==e\_ind %Check if east cell is solution

 c\_ind=e\_ind; %set current cell to end cell

 backtrack=[backtrack, c\_ind]; %add to backtrack path

 break %break the loop

 else %If cell isn't valid, and isn't end cell, it is valid

 v\_dirmap(2)=1;

 end

 %South Cell:

 if cY+1>Y %Check if south contains a cell (is within maze)

 v\_dirmap(3)=0;

 elseif maze(c\_ind, 3)==1 ||... %Check if south cell has a wall up

 visited(c\_ind+1)==1 %Check if south cell has been visited

 v\_dirmap(3)=0;

 elseif c\_ind+1==e\_ind %Check if south cell is solution

 c\_ind=e\_ind; %set current cell to end cell

 backtrack=[backtrack, c\_ind]; %add to backtrack path

 break %break the loop

 else %If cell isn't valid, and isn't end cell, it is valid

 v\_dirmap(3)=1;

 end

 %West Cell:

 if cX-1==0 %Check if west contains a cell (is within maze)

 v\_dirmap(4)=0;

 elseif maze(c\_ind, 4)==1 ||... %Check if west cell has a wall up

 visited(c\_ind-Y)==1 %Check if west cell has been visited

 v\_dirmap(4)=0;

 elseif c\_ind-Y==e\_ind %Check if west cell is solution

 c\_ind=e\_ind; %set current cell to end cell

 backtrack=[backtrack, c\_ind]; %add to backtrack path

 break %break the loop

 else %If cell isn't valid, and isn't end cell, it is valid

 v\_dirmap(4)=1;

 end

 %--------------------------------------------------Backtrack---------------------------------------------------

 if sum(v\_dirmap)==0 %If there are no valid directions

 c\_ind=backtrack(end-1); %set the current cell to the last cell on the backtrack list

 [cX,cY]=c\_coordinate(X,Y,c\_ind); %Get x and y coordinate of that index

 backtrack=backtrack(1:(end-1)); %remove the last backtrack entry from the list

 %-----------------------------------------------Move to new cell------------------------------------------------------

 else

 %Pick direction & determine if direction is valid

 v\_dir=0; % initialize variable

 while v\_dir ~=1 %run this until a valid direction is found

 dir=floor(rand\*4)+1;

 if v\_dirmap(dir)==1

 v\_dir=1;

 end

 end

 %Set new cell coordinates and index a current cell coordinates and index

 switch dir

 case 1 %North

 c\_ind=c\_ind-1;

 cY=cY-1;

 %cX=cX;

 case 2 %East

 c\_ind=c\_ind+Y;

 %cY=cY;

 cX=cX+1;

 case 3 %South

 c\_ind=c\_ind+1;

 cY=cY+1;

 %cX=cX;

 case 4 %West

 c\_ind=c\_ind-Y;

 %cY=cY;

 cX=cX-1;

 end

 %Set new cell as visited, add to backtrack path

 visited(c\_ind)=1; %new cell is visited

 backtrack=[backtrack, c\_ind]; %Add new cell to backtrack path

 end %end backtrack/move if loop

 %backtrack %debugging

end %end while loop

 %==========================================Add maze solution to maze matrix============================

 for i=1:length(backtrack) %Run loop for each entry of backtrack path

 c\_ind=backtrack(i);

 maze(c\_ind, 5)=1;

 end

fprintf('Maze solved. \n')

%backtrack %debugging

maze;

main=[maze; X, Y, s\_ind, e\_ind, 0];

**function []=plotsolvedmaze(main)**

%Author: Sean Coss

%Date Started: 3-23-2013

function []=plotsolvedmaze(main);

%Takes the input maze and x and y coordinates, and plots the solved maze

clf %clear figure

%X=ans(end,1) %Pull width

%Y=ans(end,2) %Pull height

%maze=ans((1:end-1),(1:4)) %Pull maze data from main matrix

X=main(end,1); %Pull width

Y=main(end,2); %Pull height

maze=main((1:end-1),(1:end)); %Pull maze data from main matrix

%reset cX, cY, c\_ind values

cX=1;

cY=Y;

c\_ind=1;

axis ([-1, (X+1),-1,(Y+1)])

%---------------------------------------------Plot Maze Loop--------------------------------------------------------------

for cX=1:X

 for cY=Y:-1:1

 %Draw North Wall

 if maze(c\_ind, 1)==1

 DX=[(cX-1), cX];

 DY=[cY,cY];

 hold on

 plot(DX,DY,'k')

 end

 %Draw East Wall

 if maze(c\_ind, 2)==1

 DX=[cX,cX];

 DY=[(cY-1),cY];

 hold on

 plot(DX,DY,'k')

 end

 %Draw South Wall

 if maze(c\_ind, 3)==1

 DX=[(cX-1), cX];

 DY=[(cY-1),(cY-1)];

 hold on

 plot(DX,DY,'k')

 end

 %Draw West Wall

 if maze(c\_ind, 4)==1

 DX=[(cX-1),(cX-1)];

 DY=[(cY-1),cY];

 hold on

 plot(DX,DY,'k')

 end

 %c\_ind

 c\_ind=c\_ind+1;

 %axis ([-1, (X+1),-1,(Y+1)])

 drawnow

 axis equal

 %pause (2/(X\*Y))

 %cY

 %cX

 end

end

%===========================================Maze Solution===============================================

 %reset cX, cY, c\_ind values

cX=1;

cY=Y;

c\_ind=1;

mazesize=size(main);

if mazesize(2)==5

 for cX=1:X

 for cY=Y:-1:1

 %Check cell for solution parameter

 if maze(c\_ind, 5)==1

 DX=(cX-.5);

 DY=(cY-.5);

 hold on

 plot(DX,DY,'r\*')

 end

 %Increase c\_ind, and display point

 %c\_ind

 c\_ind=c\_ind+1;

 %axis ([-1, (X+1),-1,(Y+1)])

 drawnow

 axis equal

 %pause (2/(X\*Y))

 %cY

 %cX

 end

 end

else

 disp('No solution available.')

end

**function []=drawsolvedmaze(main)**

%Author: Sean Coss

%Date Started: 3-23-2013

function [colorimage]=drawsolvedmaze(main);

%Takes the input maze and x and y coordinates, and plots the solved maze

clf %clear figure

global colorimage

%X=ans(end,1) %Pull width

%Y=ans(end,2) %Pull height

%maze=ans((1:end-1),(1:4)) %Pull maze data from main matrix

X=main(end,1); %Pull width

Y=main(end,2); %Pull height

maze=main((1:end-1),(1:end)); %Pull maze data from main matrix

%Determine img parameters

cs=32; %determine cell size in pixels

ws=4; %determine cell wall size in pixels

imgX=(cs\*X)+(2\*ws);

imgY=(cs\*Y)+(2\*ws);

%Generate image

image=ones(imgY, imgX);

%reset cX, cY, c\_ind values

cX=1;

cY=Y;

c\_ind=1;

%---------------------------------------------Plot Maze Loop--------------------------------------------------------------

for cX=1:X

 for cY=1:Y

 %Draw North Wall

 if maze(c\_ind, 1)==1

 DXi=ws+(cX-1)\*cs+1-ws;

 DXf=ws+(cX-1)\*cs+cs+ws;

 DYi=ws+(cY-1)\*cs+1;

 DYf=ws+(cY-1)\*cs+ws;

 image(DYi:DYf,DXi:DXf)=0;

 end

 %Draw East Wall

 if maze(c\_ind, 2)==1

 DXi=ws+cX\*cs-ws+1;

 DXf=ws+cX\*cs;

 DYi=ws+(cY-1)\*cs+1-ws;

 DYf=ws+(cY)\*cs+ws;

 image(DYi:DYf,DXi:DXf)=0;

 end

 %Draw South Wall

 if maze(c\_ind, 3)==1

 DXi=ws+(cX-1)\*cs+1-ws;

 DXf=ws+cX\*cs+ws;

 DYi=ws+cY\*cs-ws+1;

 DYf=ws+cY\*cs;

 image(DYi:DYf,DXi:DXf)=0;

 end

 %Draw West Wall

 if maze(c\_ind, 4)==1

 DXi=ws+(cX-1)\*cs+1;

 DXf=ws+(cX-1)\*cs+ws;

 DYi=ws+(cY-1)\*cs+1-ws;

 DYf=ws+(cY)\*cs+ws;

 image(DYi:DYf,DXi:DXf)=0;

 end

 %c\_ind

 c\_ind=c\_ind+1;

 %axis ([-1, (X+1),-1,(Y+1)])

 %pause (2/(X\*Y))

 %cY

 %cX

 %imshow(image)

 %pause(wait)

 end

end

%----------------------------------------------Fill in borders--------------------------------------------

for cX=1:imgX

 %Check top border

 if image (ws+1,cX)==0

 image(1:ws,cX)=0;

 end

 %Check bottom border

 if image((ws+Y\*cs),cX)==0

 image((ws+Y\*cs+1):(ws+Y\*cs+ws),cX)=0;

 end

end

for cY=1:imgY

 %Check left border

 if image(cY,ws+1)==0

 image(cY,1:ws)=0;

 end

 % Check right border

 if image(cY,(ws+X\*cs))==0

 image(cY,(ws+X\*cs+1):(ws+X\*cs+ws))=0;

 end

end

imshow(image)

%-------------------------------------------Convert maze into multi-color format---------------------

%choice='no';

%while strcmp(choice,'yes')~=1

 %Choose color and background color for maze

 %backcolor=input('Enter a background color for the maze')

 backcolor=[1 1 1];

 %linecolor=input('Enter a line color for the maze')

 linecolor=[0 0 0];

 %Set up color image to be same as original, with 3 layers

 colorimage=ones(imgY,imgX,3);

 colorimage(:,:,1)=image;

 colorimage(:,:,2)=image;

 colorimage(:,:,3)=image;

 for cX=1:imgX

 for cY=1:imgY

 if image(cY,cX)==1

 colorimage(cY,cX,:)=backcolor;

 elseif image(cY,cX)==0

 colorimage(cY,cX,:)=linecolor;

 end

 end

 end

 imshow(colorimage)

 %choice=input('Are you happy with your color choce? ','s');

%end

%===========================================Maze Solution===============================================

%reset cX, cY, c\_ind values

cX=1;

cY=Y;

c\_ind=1;

mazesize=size(maze); %Get dimensions of maze matrix; used for determining if sol'n exists

if mazesize(2)==5

 for cX=1:X

 for cY=1:Y %sweep across maze

 %Switch solution blocks to red

 if maze(c\_ind, 5)==1

 DXi=ws+(cX-1)\*cs+1+ws;

 DXf=ws+(cX-1)\*cs+cs-ws;

 DYi=ws+(cY-1)\*cs+1+ws;

 DYf=ws+(cY-1)\*cs+cs-ws;

 colorimage(DYi:DYf,DXi:DXf, 1)=1;

 colorimage(DYi:DYf,DXi:DXf, 2:3)=0;

 end

 %Increase c\_ind, and display point

 %c\_ind

 c\_ind=c\_ind+1;

 %axis ([-1, (X+1),-1,(Y+1)])

 %pause (2/(X\*Y))

 %cY

 %cX

 %imshow(colorimage)

 end

 end

else

 disp('No solution available.')

end

imshow(colorimage)

%Simplified determine whether to dave image file

% filename=input('Enter name to save maze as (include extension, leave blank if not saving):','s')

% if strcmp(filename,(''))==1

% else

 % imwrite(image,filename)

% end

%End of program

**\*Note: To eliminate redundancy, functions plotmaze and drawmaze were omitted, as all code from these functions is in the functions plotsolvedmaze and drawsolvedmaze.**