**Project 3: Mazes on the Web**

***Random maze generation and solution using Partition and DFS***

**Version 03/13/19**

**Educational Objectives:** After completing this assignment, the student should be able to accomplish the following:

1. Describe and elucidate the API for the Partition classes (1 and 2)
2. State and give intuitive arguments for the asymptotic runtime of Partition operations
3. Describe the supporting data structure for Partition (1 and 2)
4. Explain how Union is implemented
5. Explain how Find is implemented without path compression
6. Explain how Find is implemented with path compression
7. Define a Maze
8. Define and explain the Walls Code used to define cells in a Maze
9. Define and explain the concept of self-consistency in a maze file
10. Use Partition as a bookkeeping device to generate random mazes
11. Explain how Maze relates to Graph
12. Explain how DFS or BFS can be applied to solve a Maze problem
13. Explain why DFS is better suited to model a single rat (or person!) attempting to escape a maze.

**Operational Objectives:**

1. Implement a random maze generator as a client of fsu::Partition
2. Implement a maze solver using fsu::Path\_DFS
3. Develop a web-based demo of 1 and 2

**Deliverables:**

ranmaze2t.cpp # client program of Partition <> generating random mazes

ranmaze2w.cpp # client program of Partition <> generating random mazes

solvemaze.cpp # solves a maze problem conforming to the course maze technology

maze\_demo # contains maze demonstration

log.txt # development and testing log, contains URL for maze\_demo

**Procedural Requirements**

1. The official development/testing/assessment environment is specified in the Course Organizer. Code should compile without warnings or errors.
2. In order not to confuse the submit system, create and work within a separate subdirectory cop4531/proj3.
3. Maintain your work log in the text file log.txt as documentation of effort, testing results, and development history. This file may also be used to report on any relevant issues encountered during project development.
4. **General requirement on genericity.** Use generics whenever possible. For example:
   1. If you need to sort, use a generic sort algorithm or a container with built-in sorting.
   2. If a sort requires a specialized notion of order, create a function object that captures the desired order property.
   3. If you need a graph algorithm, use a component of the fsu graph library.
   4. In general, whenever a generic algorithm exists that can be deployed, do not circumvent that with specialized one-off code.
   5. Carefully choose all containers as the most appropriate for a particular purpose.

In short: re-use components from LIB (or your own versions) whenever possible. Don't re-invent the wheels.

1. Begin by copying all of the files from LIB/proj3 into your proj3 directory along with relevant executables from area51. You should see at least the following:
2. deliverables.sh # submission configuration file
3. ranmaze2t\_i.x # sample executable
4. ranmaze2w\_i.x # sample executable
5. mazemaster\_i.x # maze analyzer
6. solvemaze\_i.x # maze solver
7. makefile # makefile for project
8. mazeprint.c # C source code for "printmaze.x" (for demo)
9. index.php # index for demo
10. mazegen.sh # bash script for demo

Execute ranmaze2w\_i.x to generate maze files and mazemaster.x to analyze the resulting maze files. You will follow the same process in testing your own ranmaze and its resulting maze files. Also pay careful attention to the detailed I/O behavior of ranmaze, this is how your program should behave.

1. Create the file ranmaze2t.cpp, a client of Partition<>. Using mazemaster2t.x, test thoroughly to be sure the mazes it generates are self-consistent and that the behavior mimics that of LIB/area51/ranmaze2t\_i.x, including the file name extensions for output files.
2. Create the file solvemaze.cpp, a client of Path\_DFS<>. Test with smaller mazes against results from mazemaster.
3. Activate your CS web space on ww2.cs.fsu.edu. Note this may require interaction with the system group.
4. Create your web space directory http://ww2.cs.fsu.edu/~[your\_user\_name]/maze\_demo
5. Populate your demo directory with the files
6. index.php
7. mazegen.sh
8. ranmaze2w.x
9. solevemaze.x
10. printmaze.x
11. Set permissions for initial creation of output files (owned by apache), run one time, and then reset permissions to a safer level.
12. Test the web demo (on web server) and the text-based system (on linprog).
13. Follow these steps when you are ready to submit:
    1. Be sure that you have established the submit script LIB/scripts/submit.sh as a command in your ~/.bin directory. [We are currently using version 2.0.]
    2. Copy the submit configuration script LIB/proj3/deliverables.sh into your project directory. [This is an important step, as deliverables may have changed.]
    3. Submit the project using the command submit.sh while in your project directory and logged in to shell or quake. Pay attention to the screen, which will warn you about missing deliverables and incorrect login.
    4. Check your CS email for feedback. If Robocheck is active for this project, read the results from automated testing.

***Warning:****Submit scripts do not work on the program and linprog servers. Use shell.cs.fsu.edu or quake.cs.fsu.edu to submit projects. If you do not receive the second confirmation with the contents of your project, there has been a malfunction.*

**Code Requirements and Specifications - ranmaze**

1. Output maze files according to the spec in the Union-Find notes Appendix: Maze Technology. Take care that all maze files are syntactically and semantically correct - mazemaster.x is helpful in testing your output.
2. The maze file should have the start cell at the beginning (left-most cell) of the middle row and the goal at the end (right-most cell) of the same row.
3. ranmaze should expect three command line arguments: number of rows, number of columns, and filename.
4. ranmaze2t.x should output two versions of the random maze: (1) the first time goal is reachable from start, output the maze to the file "filename.[components]" where [components] is the actual number of components in the maze; and (2) after all cells are reachable from start, output the maze to the file "filename.1". ("2t" is for "text version using partition2.h".)
5. ranmaze2w.x should output only the version of the random maze with all cells reachable from start (i.e., the maze graph is connected) and output file "filename.1". ("2w" is for "web version using partition2.h".)
6. Take care that your mazes are correctly "random" in the sense that all walls that can be legally up/down have the same probability of being selected for inspection at each step. This is mostly a matter of judicious design, especially at boundary cases. For example: be sure that a cell face on an interior cell has the same chance of selection as a cell face of a boundary cell, where fewer faces are eligible.

**Code Requirements and Specifications - solvemaze**

1. Input: a maze named "xxx"  
   Output: a copy of xxx, with the solution appended, in file named "xxx.dfs".  
   (Here, "xxx" stands for the name of any semantically correct maze file.)
2. Use the utilities to full advantage to follow this processing model:
   1. Load maze file into directed graph
   2. Check digraph for symmetry (same as checking maze for consistency)
   3. Find path from start to goal
   4. Create output file name and open file for write
   5. Copy contents of input file to output file - first the documentation then the numbers
   6. Write solution as cell numbers (should be same as vertex numbers)

Be ready to explain why applying Path\_DFS(v,w) is more efficient than calling DFSurvey::Search(v).

**Code Requirements and Specifications - web demo**

1. Behavior should mimic <http://www.cs.fsu.edu/~lacher/courses/COP4531/maze_demo/>.
2. Be sure you have read and understand the scripting code in the supplied files
3. index.php
4. mazegen.sh

**Hints - makefile**

* The supplied makefile can be used to create executables that can help in your maze study:
* agraph.x # graph analyzer
* fpath.x # computes paths (bfs and dfs)
* fsymm.x # detects symmetry in digraphs
* maze2graph.x # converts maze file to its isomorph
* mazemaster.x # maze analyzer
* printmaze.x # creates postscript (printable) representation of maze (with solution if present)
* rangraph.x # generates random graph (input vertex and edge counts)
* ranmaze2t.x
* ranmaze2w.x
* solvemaze.x

(The last 3 require your deliverable source code, the others build with no contrubution in your directory.)

It is important that you check the output from your executables to be sure it is correct. For example, be sure that ranmaze2t outputs semantically correct maze files (including the self-consistency condition) and that the number of components is correct. It's also important to check your solutions.

**Hints - Ranmaze**

* A very handy way to generate unsigned integers is with the fsu random object generator. Include these files:
* #include <xran.h>
* #include <xran.cpp>

and declare an object and use it like this:

static fsu::Random\_unsigned\_int ran;

...

x = ran(a,b); // x is a random unsigned int in the range a <= x < b

y = ran(0,4); // y is a random choice of 0,1,2,3 (N,E,S,W)

z = ran(0,2); // z is a random choice of 0 or 1 (coin flip)

This generator uses the Marsaglia KISS generator as its underlying engine.

The keyword static in this context has the meaning inherited from C: The object will be initialized the first time it comes into scope and remain alive during the remaining execution, going in and out of scope as needed.

* An effective way to organize ranmaze.cpp is to write a function
* void Connect(
* size\_t beg , size\_t end , // start and goal cells
* size\_t numrows, size\_t numcols, // maze dimensions
* fsu::Vector<uint8\_t>& maze , // walls codes for cells - passed by reference
* fsu::Partition<size\_t>& p // tracks component structure - passed by reference
* );

that can be called from main() once for the "first pass" maze and subsequently as often as needed for the "one component" maze. Note that the maze data and partition data are passed by reference so that modifications made in a call to Connect will affect the data in the calling process. The organization is then left to function main:

int main (int argc, char\* argv[])

{

...

size\_t numcells = numrows \* numcols;

start = ??;

goal = ??;

fsu::Vector<uint8\_t> maze (numcells, 15); // all closed boxes

fsu::Partition<size\_t> p (numcells); // all singletons

// ensure goal is reachable from start:

std::cout << " components after 0 passes: " << p.Components() << '\n';

Connect (start, goal, numrows, numcols, maze, p);

size\_t components = p.Components();

std::cout << " components after 1 pass: " << components << '\n';

if (components > 1) // output intermediate result the first time goal is reachable

{

// build filename filec

...

// output first pass maze

out1.open(filec);

WriteMaze(out1, start, goal, numrows, numcols, maze);

out1.close();

out1.clear();

std::cout << "1-pass maze written to file " << filec << '\n';

}

// continue until all cells are reachable

if (components > 1)

{

for (size\_t cell = 1; cell < numcells; ++cell)

{

Connect (0, cell, numrows, numcols, maze, p);

}

}

std::cout << " components after all passes: " << p.Components() << '\n';

out1.open(file1);

WriteMaze(out1, start, goal, numrows, numcols, maze);

out1.close();

out1.clear();

std::cout << "n-pass maze written to file " << file1 << '\n';

...

}

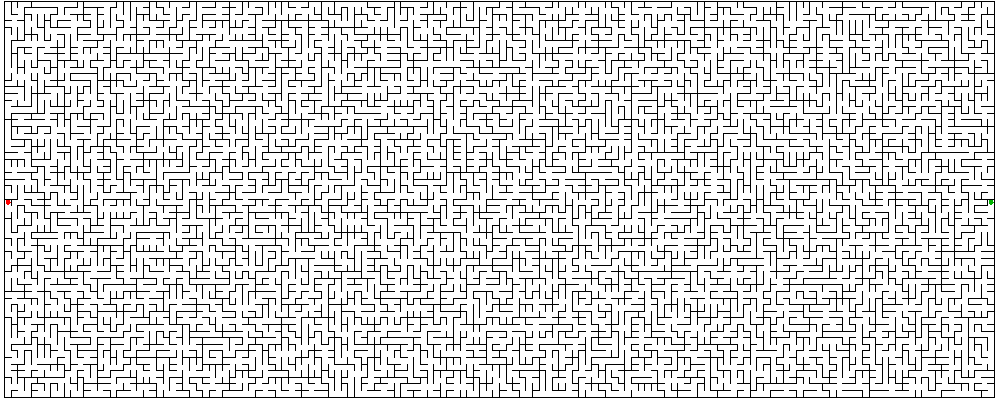
* Building the filename is just the metedious [meticulous + tedious] work of creating the correct size null-terminated character array and putting all the characters in the right place. You need two such file names "filec" and "file1" each of which is the user input filename "file" with an extension appended. To get the extension for filec, unfortunately there is no "itoa" in the library, so you need the number of digits = 1 + (size\_t)log10(components) and a loop that finds the digits:
* for (size\_t i = 0; i < digits; ++i)
* {
* char digit = '0' + (char)(components % 10);
* // put this digit in the correct place in filec
* components /= 10;
* }

This loop calculates the digits. You have to put them in the correct place in the file name array. Don't forget the null terminator. (You can also use std::stringstream if you prefer.)

* Here is a way to get started on the main program, including the handy constants for four atomic walls:
* const uint8\_t NORTH = 0x01;
* const uint8\_t EAST = 0x02;
* const uint8\_t SOUTH = 0x04;
* const uint8\_t WEST = 0x08;
* ...
* int main (int argc, char\* argv[])
* {
* if (argc < 4)
* {
* std::cout << "\*\* command line arguments required:\n"
* << " 1: number of rows\n"
* << " 2: number of cols\n"
* << " 3: file name for maze\n"
* return 0;
* }
* size\_t numrows = atoi(argv[1]);
* size\_t numcols = atoi(argv[2]);
* ...
* }

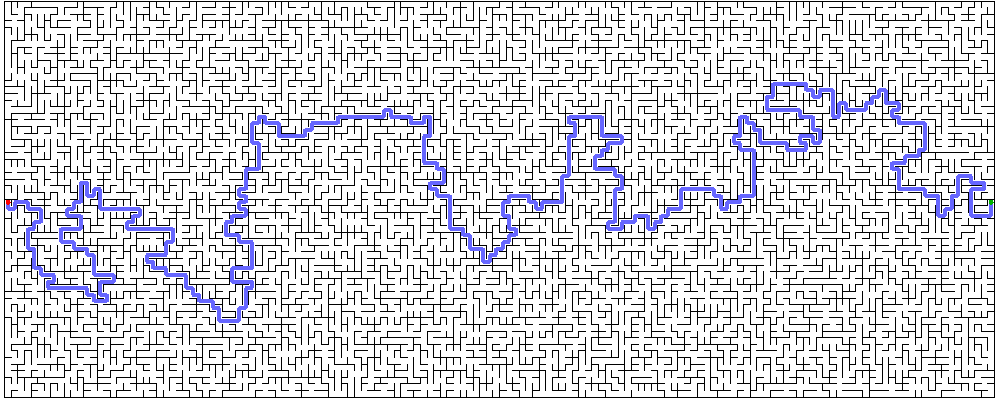
The wall constants are global, hence accessible from Connect()

* The following image is of a maze generated by ranmaze. Start and goal cells marked red/green, respectively.



**Hints - Solvemaze**

* The deeper steps outlined in the requirements section can each be implemented with a call to a template in LIB/graph:
  1. Load maze file into directed graph: **LoadMaze**
  2. Check digraph for symmetry (same as checking maze for consistency): **CheckSymmetry**
  3. Find path from start to goal: **Path\_DFS**
* The following image shows the DFS solution of the previously generated maze:



**Hints - Visualization / demo**

* The executable area51/mazemaster.x is useful in analysing small maze files while the maze programs are under development. The [mazebuilder-olson](http://www.cs.fsu.edu/~lacher/courses/notes/mazebuilder-olson.html) and [mazebuilder-brown](http://www.cs.fsu.edu/~lacher/courses/notes/mazebuilder-brown.html) javascripts (written by former students) are useful in creating small mazes for testing.
* There are two issues in setting up the maze\_demo. The first is making sure you have exectutables for the supporting infrastructure. It is likely that your linprog executables ranmaze2w.x and solvemaze.x execute correctly on the web server. It is also likely that printmaze.x does not. You can rebuild printmaze.x on the web server with this line:
* cc -O2 -g mazeprint.c -o printmaze.x -lm

(This is a line out of the makefile.)

* The second issue is managing the permissions and ownership of files. Note that technically apache is the executor of the programs activated by the "submit" button. So the data files need to be created for the first time by apache, their owner. This in turn requires that the directory have public write permission just while the files are created for the first time. After that, the dangerous access flags can be changed to 711 on the directory. Inside the directory, data files need to be publicly readable and executable files need to be publicly executable. In neither case do they need public write permission.

When first populating your maze\_demo directory the permissions should look like this:

-rw-r--r-- 1 [user] [group] 2565 Aug 13 19:35 index.php

-rwxr-xr-x 1 [user] [group] 1212 Aug 13 21:51 mazegen.sh

-rwxr-xr-x 1 [user] [group] 25997 Aug 13 14:53 printmaze.x

-rwxr-xr-x 1 [user] [group] 32592 Aug 13 14:53 ranmaze2w.x

-rwxr-xr-x 1 [user] [group] 52656 Aug 13 14:53 solvemaze.x

with the containing directory set to

drwxrwxrwx 4 [user] [group] 4096 Aug 13 21:51 maze\_demo

Then execute by clicking to see:

-rw-r--r-- 1 [user] [group] 2565 Aug 13 19:35 index.php

-rw-r--r-- 1 apache apache 12229 Aug 13 23:47 maze.dfs.png

-rwxr-xr-x 1 [user] [group] 1212 Aug 13 21:51 mazegen.sh

-rw-r--r-- 1 apache apache 7100 Aug 13 23:47 maze.png

-rwxr-xr-x 1 [user] [group] 25997 Aug 13 14:53 printmaze.x

-rwxr-xr-x 1 [user] [group] 32592 Aug 13 14:53 ranmaze2w.x

-rwxr-xr-x 1 [user] [group] 52656 Aug 13 14:53 solvemaze.x

-rw-r--r-- 1 apache apache 21388 Aug 13 23:47 \_TEMPMAZE.1

-rw-r--r-- 1 apache apache 23513 Aug 13 23:47 \_TEMPMAZE.1.dfs

-rw-r--r-- 1 apache apache 25895 Aug 13 23:47 \_TEMPMAZE.dfs.ps

-rw-r--r-- 1 apache apache 23314 Aug 13 23:47 \_TEMPMAZE.ps

and finally change permissions on the parent directory to:

drwx--x--x 4 [user] [group] 4096 Aug 13 21:51 maze\_demo

In the above, [user] stands for your CS-username and [group] your group - most likely "majors", but whatever it is leave it be. Once apache has been permitted to create (and own) the new files, they can be updated "on-click" with safer permissions on the parent directory.

* The index.php as distributed emits diagnostic information from mazegen.sh which can be helpful while getting all the moving parts to work together in the demo. (This shows up at the bottom of the display, underneath the description.) You may hide that output by commenting out the last <?php ... ?> block:
* <!-- <-- begins comment
* <?php
* echo '========= Diagnostic output from mazegen.sh =========  
  ';
* echo "<pre>$output</pre>";
* ?>
* --> <-- ends comment

However, please continue to display the project discription and credit for the class.