

Maze Generator

CS 241

Data Organization using C

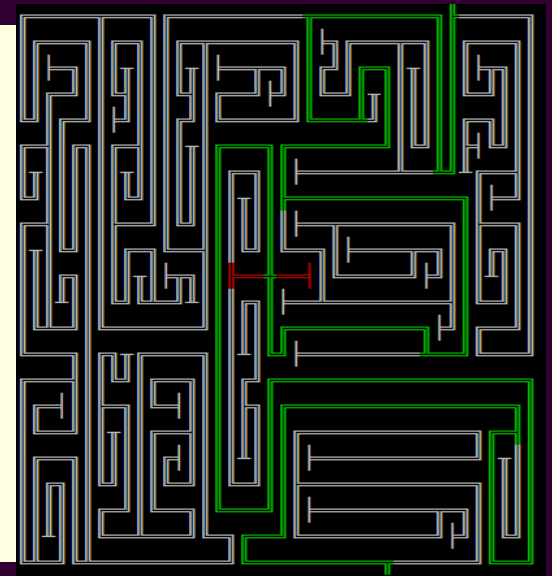
Instructor: Joel Castellanos

e-mail: joel@unm.edu

Web: <http://cs.unm.edu/~joel/>

Office: Farris Engineering Center

Room 2110



The Maze Project: Required Skills

- Control flow and character I/O: Chapters 1-4
- Header files: 4.5
- Variable scope: 4.4 and 4.6
- Recursion: 4.10
- Pointers: 5.1 through 5.6
- Multi-dimensional Arrays: 5.7, 5.8, 5.9
- Dynamic Memory Allocation: `malloc()`, `free()`
- valgrind

Maze Project Milestones

- Basic Recursive Maze
 - Draw with Pipe Characters.
 - Tested with function calls from given `mazeTest.c`
- Improved Recursive Maze
 - Draw with Pipe Characters.
 - Tested with function calls from given `mazeTest.c`
- Polished Maze with extra credit enhancements.
 - Write bitmap binary image file.
 - One-on-one: Demo, Code Review, & "How Would You?"

Maze Requirements (Milestone I)

1. Your generator must create rectangular mazes of any specified width and height (≥ 3 and within the computer's memory limitations).
2. The mazes must be random: A pair of mazes of the same, sufficiently large size, must have a low probability of being identical.
3. Each maze must have exactly two openings along its outer edge: one in a random location along the top. The other in a random location along the bottom.

Maze Requirements (Milestone I)

4. Each maze must have exactly one path that connects the two outer edge openings.
5. Every cell in each maze must be reachable from any other cell in the maze.
6. Each maze must contain dead ends which, on average, increase in number, get longer, have more turns, and more branches as maze size increases.

Maze Requirements (Milestone 1)

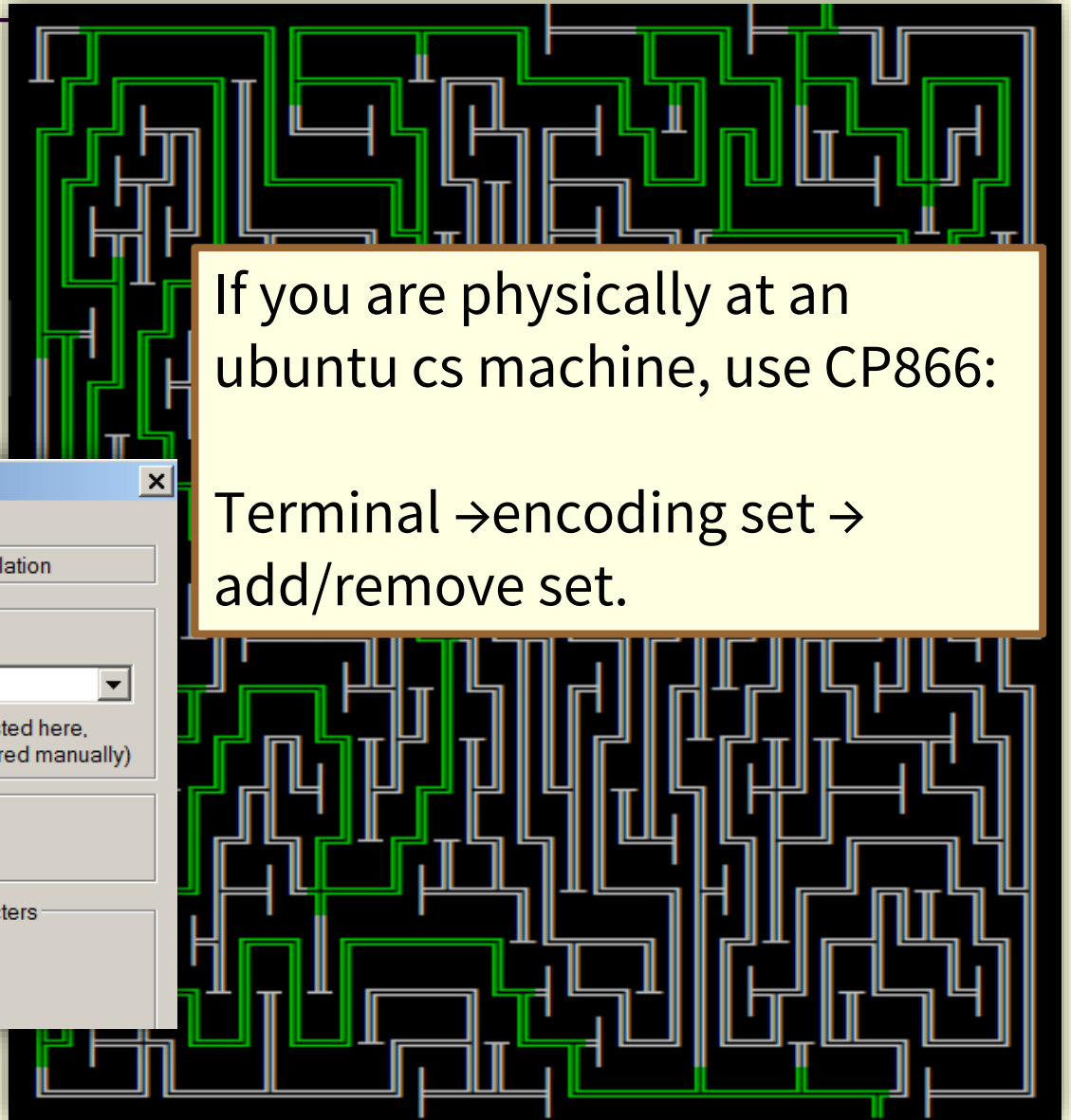
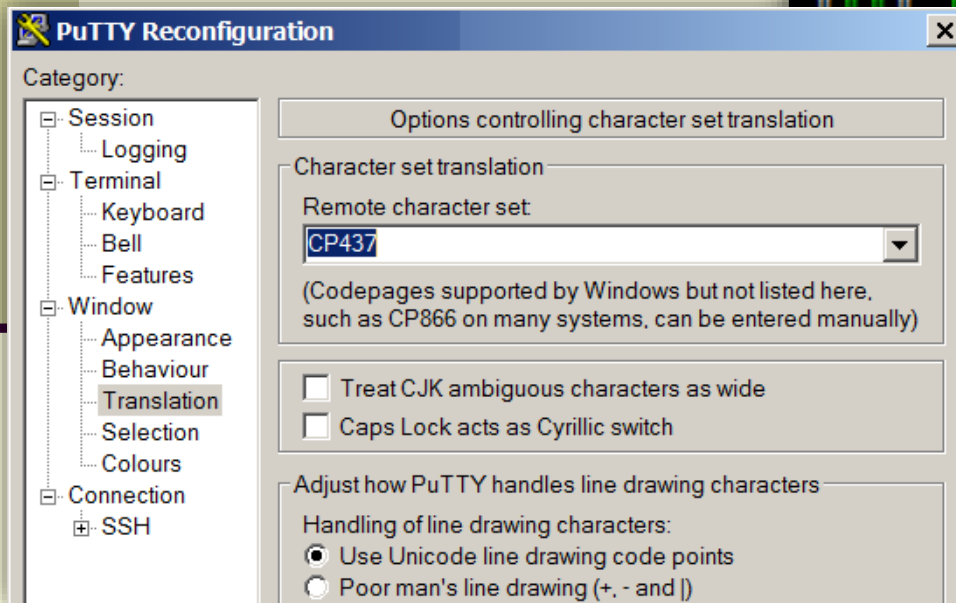
7. Each maze must use dynamically allocated memory (`malloc` & `free`) and must not contain memory leaks (checked using `valgrind`).
8. Your file, `mazegen.c`, must implement each of the functions defined in the header file `mazegen.h`.
9. The functions defined in `mazegen.h` must run correctly when called from `mazetest.c`. This file contains `main(...)`.
 - Hint: to get started, comment out all but the first 1 or 2 tests in `mazetest.c`

Maze Requirements (Milestone 1)

In milestone 1 and 2, each maze must be drawn using the pipe characters of the CP437 character set.

If you are physically at an ubuntu cs machine, use CP866:

Terminal → encoding set → add/remove set.



CP437 Extended ASCII Pipe Characters

ASCII	Sym	Bits
219	█	0000
208	⌚	0001
198	ƒ	0010
200	℔	0011
210	π	0100
186	∥	0101
201	ƒ	0110
204	ƒ	0111

ASCII	Sym	Bits
181	ƒ	1000
188	⌚	1001
205	=	1010
202	⌚	1011
187	ƒ	1100
185	ƒ	1101
203	π	1110
206	ƒ	1111

Required Interface: mazegen.h

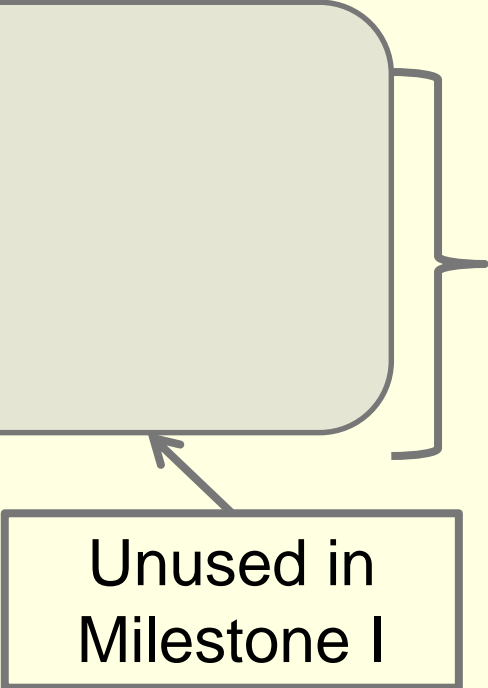
```
int mazeGenerate(int width, int height,  
int wayPointX, int wayPointY,  
int wayPointAlleyLength,  
double wayPointDirectionPercent,  
double straightProbability,  
int printAlgorithmSteps)
```

```
void mazeSolve(void);
```

```
void mazePrint(void);
```

```
void mazeFree(void);
```

Unused in
Milestone I

A diagram consisting of a rounded rectangular callout box with a grey background and a black border. This box encloses the entire signature of the mazeGenerate function. To the right of this callout box is a large right-facing curly bracket. Below the callout box is a rectangular box with a black border containing the text 'Unused in Milestone I'. An arrow points from the top-left corner of this box to the bottom-right corner of the callout box.

Useful Constants: mazegen.h

```
#define NORTH 1 //0000 0001
#define EAST 2 //0000 0010
#define SOUTH 4 //0000 0100
#define WEST 8 //0000 1000
#define TOTAL_DIRECTIONS 4
#define ALL_DIRECTIONS 0x0F //0000 1111
#define NO_DIRECTIONS 0 //0000 0000

//Useful when solving the maze.
#define GOAL 16 //0001 0000
#define VISITED 32 //0010 0000

#define SPECIAL 64 //0100 0000
```

mazeGenerate(...)

```
int mazeGenerate(int width, int height,  
int wayPointX, int wayPointY,  
int wayPointAlleyLength,  
double wayPointDirectionPercent,  
double straightProbability,  
int printAlgorithmSteps)
```

- Allocates memory and generates a random maze of the given width and height.
- Returns FALSE if all is well.
- Returns TRUE and displays an error message to **stdout** (NOT **stderr**).

mazeFree(void)

```
extern void mazeFree(void);
```

- Frees memory used by the maze.
- Must be called (by you) at the start of every call to `mazeGenerate(...)` to free all memory used to create the last maze.
- Must be called (by `mazetest.c`) before program exits.
- `valgrind` will be used to test for memory leaks.

mazeSolve(void)

Must solve the maze.

How you solve the maze and how you represent the solution internally is not specified.

If a maze is printed before `mazeSolve()` is called, then the output must not show the solution,

If a maze is printed after `mazeSolve()` is called, then the output must show the solution in green pipes.

mazePrint(void)

Prints:

```
printf("\n\n");  
printf("=====\n");  
printf("Maze(%d x %d): (%d, %d)\n",  
      width, height, wayPointX, wayPointY);  
printf("=====\n");
```

The Maze - using the pipe characters:

- The waypoint must be red.
- The solution path, if **mazeSolve()** was called, must be green.
- The other pipes must be white.

mazetest.c: Structure and Use

```
#include <stdio.h> //look in standard library
#include "mazegen.h" //look in current directory.
```

```
int main(void)
{
    // Test Cases:

    // Release memory
    mazeFree();

    // Program finished properly
    return FALSE;
}
```

Build maze executable:
Create a directory with
 mazeTest.c
 mazegen.h
 mazegen.c
Compile:
 gcc *.c

mazetest.c: Required Output

```
mazeGenerate(3,3, 2,2,0, 1.0, 0.0, FALSE);
```

```
//When mazePrint() is called after mazeGenerate(),  
// must print maze using monochrome pipes characters.
```

```
mazePrint();
```

```
mazeSolve();
```

```
//When mazePrint() is called after mazeSolve():
```

```
// Must print the most recent maze.
```

```
// Must draw pipes that are not part of the solution in one  
// color and pipes that lie on the solution in green.
```

```
// Must draw pipe at waypoint in red.
```

```
mazePrint();
```


Setting Text Color

- Given in mazeset.c is a helper function for setting the text color displayed by `printf`.

```
//#define TEXTCOLOR_BLACK    30
//#define TEXTCOLOR_RED      31
//#define TEXTCOLOR_GREEN    32
//#define TEXTCOLOR_YELLOW   33
//#define TEXTCOLOR_BLUE     34
//#define TEXTCOLOR_MAGENTA  35
//#define TEXTCOLOR_CYAN     36
//#define TEXTCOLOR_WHITE    37
void textcolor(int color)
{
    printf("%c[%d;%d;%dm", 0x1B, 0, color, 40);
}
```

Grading Rubric (Milestone I, 50 Points)

20 Points: TEST 1: Basic random mazes.

This includes, correct formatting, correct pipe characters, correct coloring before and after call to `mazeSolve`, and maze meeting requirements.

10 Points: TEST 2: No memory leaks (`valgrind`), does not use much memory for a given maze size. Does not need expanded stack or heap space.

10 Points: TEST 3: Super stress test.

10 Points: TEST 4: Error cases.

Note: if your code gets less than 15 points on Test 1, then your code auto fails all of test 2 and 3.

Grading Penalties

[-20 points]: Does not use malloc/free to dynamically reserve space the maze. For example:

```
unsigned char maze[1000][1000];
```

[-10 points]: Code does not follow CS-241 standard. This includes comments, indenting, variable naming, repeated code, etc.

Recursive Algorithm: `carvemaze(x, y, dir)`

- 1) Call `carvemaze` with the starting location and the direction it came from.
- 2) `carvemaze` must add an opening in cell (x,y) in direction, `dir`.
- 3) `carvemaze` must also add an opening of the opposite direction in the location it came from.
- 4) `carvemaze` must determine how many directions are currently open.
 - a) If there are no open directions, it must return.
 - b) If there is one or more open directions, it must call `carvemaze` with the `x` and `y` value of a cell in a randomly selected open direction.
 - c) When that call returns, go back to start of step 4.

Recursive Algorithm: debugging

- Create a small grid and print the state of the maze at the start of each call to `carvemaze()`.
- For testing, use `srand(seed)` with a fixed seed value, so you can regenerate the same maze until you get it working.

4	6	9		
7	9			
3	12			
4	5			
3	9			

π	∩	∟	■	■
∩	∟	■	■	■
∟	∩	■	■	■
∩	∥	■	■	■
∟	∟	■	■	■

Dealing with the Maze Edges

- One way to avoid out-of-bounds errors at the maze edges is to protect each check of the cell to the north with an `if (y-1 > 0)`. Similarly, protect checks in each of the other directions.
- I recommend a different way to avoid out-of-bounds errors at the maze edges:

Add an extra row and column on each edge with all values = 15.

With this structure, you can look in that direction, without going out-of-bounds, yet never break a wall in that direction. Also, never render those extra boarder cells.

	x y-1			
x-1 y	x y	x+1 y		
	x y+1			

3×3 maze

Path to Success #1: Paper First!!

- Get some graph paper!
- Draw a small, maze (5×4) and work manually through the process you are trying to code.
- When you do this, write in each cell, as the maze develops, the number your algorithm should store in each cell.
- Later, when you have code, print the direction your program chooses at each step. Then figure out on your graph paper what value your algorithm should write in each cell. Verify that your program writes the same value in each cell that you calculate on paper.

Path to Success #2: Fixed Size Array

Start by using:

```
static unsigned char maze[1000][1000];  
void mazeFree(void) {} //Does nothing.
```

After you get this working change to

```
static unsigned char **maze;  
and then use malloc() and free().
```


Recursive Algorithm: Heap verses Stack

Before you start coding, think about what variables are needed.

- Which of these variables should be arguments in the recursive call and why?
- Which variables not passed as arguments need to be automatic variables (stored on the stack where each level of the recursion has its own version of the variable).
- Which variables need to be on the heap so that all levels of the recursion see the same memory for the variable. This is needed to have one level of recursion see changes made by other levels.
- Which value(s) need to be returned by the recursion?
- Should the maze project use *tail recursion*?

How to Increasing Stack Size


NOTE: If you are running out of stack space on the maze project, then you **ARE DOING SOMETHING WRONG.**

- User limits - limit the use of system-wide resources.
- `ulimit -s <maximum stack size in bytes>`
- `ulimit -s 32768`
- `ulimit` is a *Bash* shell command.
- `man ulimit` for more information.

How might these structures be useful?

```
const int DIRECTION_LIST[] = {NORTH, EAST, SOUTH, WEST};  
const int DIRECTION_DX[]   = {  0,   1,   0,  -1};  
const int DIRECTION_DY[]   = { -1,   0,   1,   0};
```

```
const unsigned char pipeList[] =  
{  
    219, 208, 198, 200, 210, 186, 201, 204,  
    181, 188, 205, 202, 187, 185, 203, 206  
};
```



Is there a particular order to the elements of this array?

How to Find and Fix: ****Error**** Stack Overflow

- Does the problem only happen on mazes larger than any in the test file? If so, what might be the issue?
- Does the problem only happen when you generate and solve multiple mazes?
 - Does problem happen always on a second maze?
 - Does problem only happen after a bunch of large mazes?
 - How do last two questions effect where to look for errors?
- Does the problem happen when *carving* OR when *solving* OR when carving only *after a solve*?
- If problem happens with just a single maze, then test with the smallest maze that results in the same problem (3x3? 3x4?, ...).

Maze II: wayPointX, wayPointY

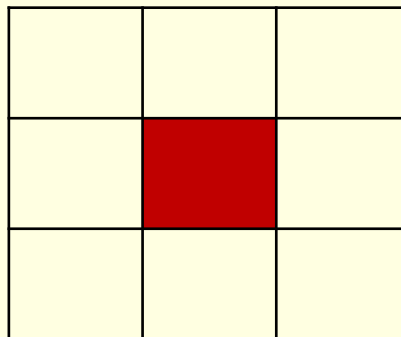
The maze solution must pass through (wayPointX, wayPointY).

```
mazeGenerate(3, 3, 2, 2, 0, 1.0, 0.0, TRUE);
```

```
mazeGenerate(20, 11, 18, 5, 4, 1.0, 0.0, TRUE);
```

The waypoints are indexed with the first row and column as 1.

For example, a 3,3 maze with a waypoint at location 2,2 would have its waypoint in the center:



Waypoint Algorithm

There are a number of ways to implement this. The one suggested below also works when adding `wayPointAlleyLength`.

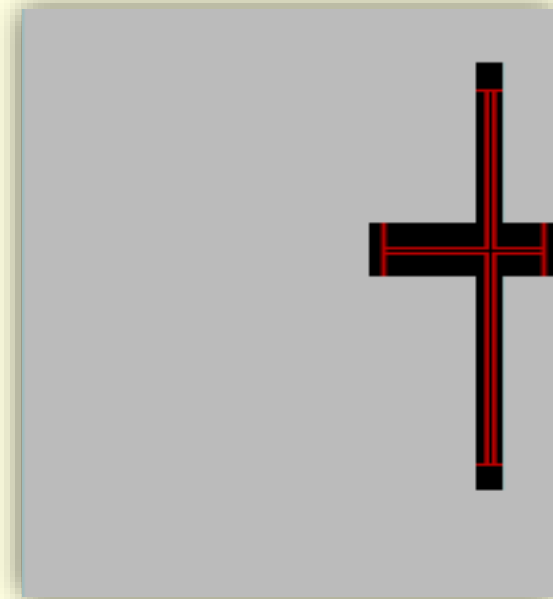
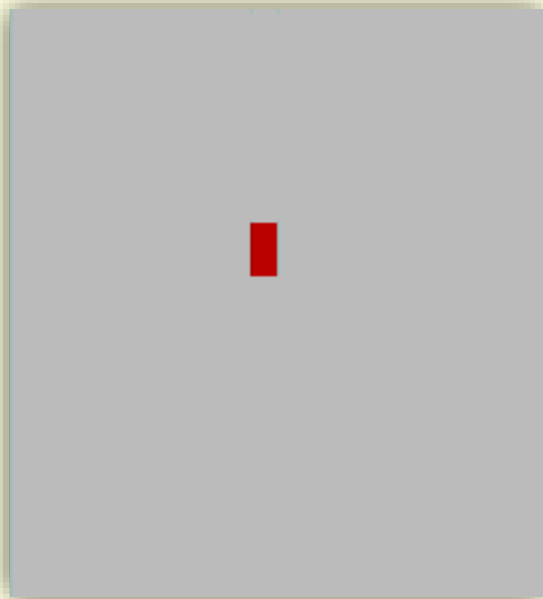
- 1) Start carving at the waypoint.
- 2) When you reach the top or bottom edge near one of corners, carve a hole. DO NOT RETURN. Set a global variable and start a new recursion within the recursion only in a random direction from the waypoint.
- 3) Carve from the waypoint in the usual way until you reach the top or bottom edge near the same or different corner.

Note: This will sometimes fail. When carving the first path from the waypoint, the path might completely block off the other waypoint directions from all edges. In this case, set an error flag to true, print a message, and print the maze partly completed. When the error flag is set, Each recursive call should return without doing any other carving.

Maze II: wayPointAlleyLength

Before starting the maze carving recursion, carve a set of allies that start at the waypoint and go straight in each direction for wayPointAlleyLength cells. *Stop when you hit an edge!*

```
mazeGenerate(3,3, 2,2,0, 1.0, 0.0, TRUE);  
mazeGenerate(20,11,18,5,4, 1.0, 0.0, TRUE);
```



Maze II: `wayPointDirectionPercent`

The `wayPointDirectionPercent` specifies the maximum number of cells to carve from the end of a single waypoint alley ***before starting a new recursion*** at the end of a different randomly selected waypoint alley.

When the max number of cells is reached, start a new recursion at each of the alley endpoints that are still open.

For example, if a 20x11 maze has a `wayPointDirectionPercent` = 0.2 (20%), then the maximum cells to carve from a single waypoint before starting another is:

$$20 \times 11 \times 0.2 = 44 \text{ cells.}$$

This cell count includes all branches made from the start of ***the most recent*** waypoint alley.

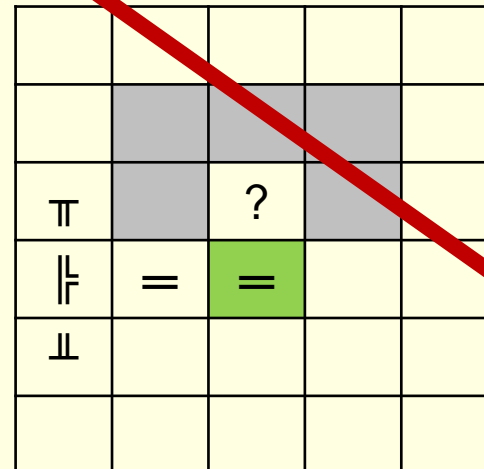
When a new waypoint recursion is started, DO NOT RETURN from the current recursion. After that new recursion returns, your maze carving still needs to pick up where it left off 

Maze II: leaveGap

After carving the allies, when carving new cells, *always choose a carve direction that connects to a cell neither next to a carved cell nor next to an edge* in **forward five directions** – if and only if such a direction exists.

For example, in the figure below, if carving NORTH from the green cell, then the forward 5 cells are shown in gray.

After carving the **SKIP** current step, all directions lead either to a used cell or a cell next to a used cell.



Maze II: `straightProbability`

If the most recent carve direction is d , *and* if continuing in that direction would not break any of the other constraints (no cycles, leave gap and max direction percentage)...

Then, before choosing a random direction, the algorithm must continue to carve in direction d with percentage equal to `straightProbability`.

A `straightProbability` of 0.0, was used in milestone 1.

A `straightProbability` of 1.0 makes degenerate mazes.

Maze II: `printAlgorithmSteps`

When `printAlgorithmSteps` is true, then print the maze as it is (with uncarved blocks shown as the solid fill character) after each step in the algorithm. In particular, print:

- 1) At the start of each new alley recursion.
- 2) At the return of each alley recursion

Draw alleys in red.

After `mazeSolve`, show solution in green, and alley not on solution in red.

Maze II: Grading Rubric

[30 Points]: Pass 10 unknown tests (3 points each).

[10 Points]: Uses dramatic memory allocation with no memory leak.

[-10 points]: Code does not follow CS-241 standard. This includes comments, indenting, variable naming, repeated code, etc.

Maze III: Bitmap and Extra Credit

- Milestone III of the maze project adds only one requirement:
- Replace the pipe characters with tiles (no smaller than 8x8 and no larger than 16x16). Output the maze (when `mazePrint()` is called) as a single bitmap image.
- Grading is during a one-on-one Demo, Code Review, & "How Would You? Some of these will be done during lab class others can be scheduled outside of class.
- There are no special color requirements for Maze III (you are not required to display a solution or color waypoint ...).
- Full points for Maze III can still be earned if the ally length or other parameters are not working correctly (but no seg faults or other crashing: make sure you always draw a random maze).

Maze III: Extra Credit

You can earn extra credit for many different creative things you might choose to do with maze III. Example:

- Use isometric sprites so the maze has a 3D appearance.
- Do actual 3D rendering with lighting and surface effects using OpenGL.
- Invent or implement from some other source different parameters than waypoint direction percent, or allies or straight probability that makes more interesting mazes.
- Make hex mazes (where each cell has 6 directions).
- Maybe experiment with cluster mazes: (i.e. size 15x15 maze areas each connected to two others via a single path).