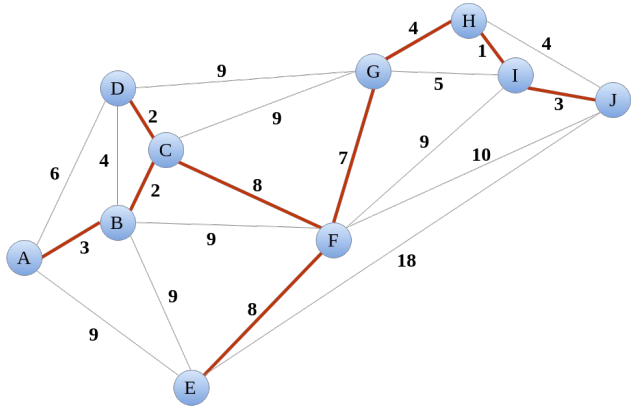
Maze Generator

Jacques Goosen

How It Works

Recursive Backtracking

- Mark current node as visited
- Pick random neighbor
- Remove wall between nodes
- Mark neighbor as current node
- If no neighbors, backtrace with stack



Code

```
12  int main(int argc, char* argv[]){
13      time_t t;
14      srand((unsigned)time(&t));
15      width = atoi(argv[1]);
16      height = atoi(argv[2]);
17      maze_t maze = {NULL,0,0};
18      maze.w = width;
19      maze.h = height;
20      allocateMaze(&maze);
21      printMaze(&maze);
22      designMaze(&maze);
23      printMaze(&maze);
24      return 0;
25  }
```

Code

```
1   typedef struct maze_t{ // maze matrix struct
2       struct cell_t **matrix;
3       int w; // size of maze
4       int h;
5   }maze_t;
```

```
7   typedef struct cell_t{ // maze cell
8       int x,y; // coordinate of cell
9       int visited;
10      int north,east,south,west; // walls
11      //char type;
12  }cell_t;
```

```
14  typedef struct stack_t{
15      cell_t* cell;
16      int top;
17  }stack_t;
```

Code

```
63  void designMaze(maze_t* maze){
64      curx = 0; cury = 0;
65      int randNeighbor;
66      stack_t* stack = (struct stack_t*)malloc(sizeof(struct stack_t));
67      stack->cell = (cell_t*)malloc(sizeof(cell_t)*maze->w*maze->h);
68      stack->top = 0;
69      stack->cell[stack->top].x = maze->matrix[cury][curx].x;
70      stack->cell[stack->top].y = maze->matrix[cury][curx].y;
71
72      maze->matrix[cury][curx].visited = 1;
73
74      while(stack->top > -1){
75          if((randNeighbor = neighbor(maze)) != -1){
76              removeEdge(maze,randNeighbor);
77              maze->matrix[cury][curx].visited = 1;
78              stack->top++;
79              stack->cell[stack->top] = maze->matrix[cury][curx];
80              printMaze(maze);
81          }
82          else{
83              stack->top--;
84              curx = stack->cell[stack->top].x;
85              cury = stack->cell[stack->top].y;
86          }
87      }
88  }
```

Code

```
27  int neighbor(maze_t* maze){
28      int result[4];
29      int hasNeighbors = 0;
30      int out;
31      memset(result, 0, sizeof(result));
32      if((maze->matrix[cury][curx].y > 0) && maze->matrix[cury][curx].north &&
33          !maze->matrix[cury-1][curx].visited){ // Possible north
34          result[0] = 1;
35          hasNeighbors = 1;
36      }
37      if((maze->matrix[cury][curx].x < width-1) && maze->matrix[cury][curx].east &&
38          !maze->matrix[cury][curx+1].visited){ // Possible east
39          result[1] = 1;
40          hasNeighbors = 1;
41      }
42      if((maze->matrix[cury][curx].y < height-1) && maze->matrix[cury][curx].south &&
43          !maze->matrix[cury+1][curx].visited){ // Possible south
44          result[2] = 1;
45          hasNeighbors = 1;
46      }
47      if((maze->matrix[cury][curx].x > 0) && maze->matrix[cury][curx].west &&
48          !maze->matrix[cury][curx-1].visited){ // Possible west
49          result[3] = 1;
50          hasNeighbors = 1;
51      }
52
53      if(hasNeighbors){
54          while(1){
55              if(result[out = (rand() % (sizeof(result)/sizeof(result[0])))]){
56                  return out;
57              }
58          }
59      }
60      else return -1;
```

Next Steps

- Add a maze solver to complement the generator
- Convert output to use SDL or other graphics library
- Import mazes from images/files

Demo