**Maze-2,3 결과보고서**

전공: 수학 학년: 4 학번: 20161255 이름: 장원태

1. **실습 및 숙제로 작성한 프로그램의 알고리즘과 자료구조를 요약하여 기술한다. 완성한 알고리즘의 시간 및 공간 복잡도를 보이고 실험 전에 생각한 방법과 어떻게 다른지 아울러 기술한다.**

우선, 파일을 읽어오는 코드는 다음과 같다. (정상적으로 파일을 읽은 경우 기준이다.)

|  |
| --- |
| while (1) { // width 측정  file.get(tempchar);  if (tempchar == '\n') break;  //printf("%c", tempchar);  if (tempchar == '-') WIDTH++;  }  int space\_check = 0;  while (1) { // height 측정  if (file.eof()) break;  file.get(tempchar);  //printf("%c", tempchar);  if (tempchar == '\n') {  space\_check++;  if (space\_check == 2) {  HEIGHT++;  space\_check = 0;  }  }  }  if (++space\_check == 2) HEIGHT++; // 엔터로 끝나지 않는 경우를 생각함  file.close();  // 메모리 할당  cell = new vertex \* [HEIGHT];  for (i = 0; i < HEIGHT; i++) {  cell[i] = new vertex[WIDTH];  }  visited = new int\* [HEIGHT];  for (i = 0; i < HEIGHT; i++) {  visited[i] = new int[WIDTH];  for (j = 0; j < WIDTH; j++)  visited[i][j] = 0;  }    // 자료구조 저장  file.open(fileName);  while (1) {  file.get(tempchar);  if (tempchar == '\n') break;  }  for (i = 0; i < HEIGHT; i++) {  file.get(tempchar); // 맨왼쪽 벽 읽음  for (j = 0; j < WIDTH; j++) {  cell[i][j].row = i;  cell[i][j].col = j;  file.get(tempchar); // 방 읽음  file.get(tempchar); // 방 오른쪽 벽 읽음  if (tempchar == '|') cell[i][j].right = 0;  else if (tempchar == ' ') cell[i][j].right = 1;  }  file.get(tempchar); // 엔터 읽음  if (tempchar == '\r') file.get(tempchar);  file.get(tempchar); // 맨왼쪽 벽 읽음  for (j = 0; j < WIDTH; j++) {  file.get(tempchar); // 방과 아랫방 사이 읽음  if (tempchar == '-') cell[i][j].down = 0;  else if (tempchar == ' ') cell[i][j].down = 1;  file.get(tempchar); // 방 모서리 읽음(+)  //printf("\n%d %d", cell[i][j].right, cell[i][j].down);  }  file.get(tempchar); // 엔터 읽음  if (tempchar == '\r') file.get(tempchar);  //printf("\n-------------------");  }  file.close(); |

미로의 너비와 높이를 나타내는 변수인 WIDTH, HEIGHT은 전역 변수로 각각 0으로 초기화되어 있다. 파일은 maze 1주차 실습 때 그린 미로의 형태로 되어 있다. 우선 맨 위 벽을 나타내는 첫 줄을 읽으며 WIDTH를 측정한다. 미로의 너비는 첫 줄에서 “-“의 개수, 즉 한 줄에 있는 방의 개수와 같으므로, “-“를 읽을 때마다 WIDTH에 1씩 더해준다.

다음으로 HEIGHT을 측정하는데, HEIGHT은 파일의 끝까지 전부 읽어 측정한다. 두 번째 줄부터 파일을 읽게 되며, 줄바꿈이 두 번 일어나면, 방을 나타내는 행과, 그 아랫방 사이의 경계를 나타내는 행을 읽은 것이므로, 그 때마다 HEIGHT에 1씩 더해준다. 줄바꿈이 두 번 일어나는 것은 space\_check라는 변수를 설정해 그 변수값이 2가 되면 HEIGHT를 1 더하고 변수값을 다시 초기화하는 과정을 거친다. 파일의 끝이 엔터로 끝나지 않는 경우를 고려하여, 마지막에는 space\_check이 1이어도 HEIGHT에 1을 더해준다. 그 후, 파일을 닫아준다.

그 후 다시 파일을 열어, 이번에는 각 방에 대한 정보를 자료구조에 저장한다. 데이터를 저장하는 자료구조는 이차원 배열로, cell이라는 이름을 가진다. 각 정보는 구조체 형태(vertex)로, 각 방의 행 좌표(row)와 열 좌표(col), 그리고 오른쪽으로 갈 수 있는지 여부(right)와 아래쪽으로 갈 수 있는지 여부(down)를 의미하는 변수들로 구성되어 있다. 그리고, BFS 알고리즘을 위하여, 미로의 탐색 과정에서 각 점에 도착했을 때 어느 방향으로부터 왔는지를 의미하는 enumeration type의 came\_from 변수가 있다. 이 변수는 BFS 알고리즘에만 사용되며, 초기화되어 있지 않다.

이 자료구조에 데이터를 저장하기 시작한다. 첫째 줄은 아무 것도 저장하지 않고 넘어간다. 둘째 줄부터 두 줄씩 읽어가며 끝까지 읽는다. 방에 해당하는 부분을 읽을 때마다 그 방의 행 좌표와 열 좌표를 cell에 저장한다. 그 후, 그 오른쪽 벽을 읽는다. 그 벽이 스페이스바로 뚫려있으면, cell에서 해당 방의 right 값을 1로 저장하여, 오른쪽으로 이동할 수 있음을 알려준다. 반대로 ‘|’로 막혀있으면, right 값을 0으로 저장하여 이동할 수 없음을 알려준다. 각 줄의 끝에 다다를 때, ‘\r’이라는 알 수 없는 문자가 읽히는 것이 확인되어, 그 문자를 읽게 되면 한 문자를 더 읽어 줄바꿈을 정상적으로 읽을 수 있도록 했다. 그 다음 줄에서는 앞서 탐색한 한 줄의 방들과 그 아랫방 사이의 경계를 읽게 되며, 그 경계가 스페이스바로 뚫려있으면, cell에서 그 위 방의 down 값을 1로 저장하여, 아랫쪽으로 이동할 수 있음을 알려준다. 반대로 ‘-‘로 막혀있으면, down 값을 0으로 저장하여, 아랫쪽으로 이동할 수 없음을 알려준다. 이 과정을 모두 거치면 cell에 came\_from을 제외한 모든 정보가 저장된다.

이제 미로를 윈도우 상에 그릴 수 있게 하는 과정이 필요하다. 해당 코드는 다음과 같다.

|  |
| --- |
| for (i = 0; i < HEIGHT; i++) {  if (i == 0) // 맨 윗줄 그리기  ofDrawLine(startx, starty, startx + 40.0f \* WIDTH, starty);  for (j = 0; j < WIDTH; j++) {  if (j == 0) { // 맨 왼쪽 벽 그리기  linex = startx;  liney = starty + 40.0f \* (float)i;  ofDrawLine(linex, liney, linex, liney + 40.0f);  }  if (cell[i][j].right == 0) { // 해당 칸의 오른쪽 벽 그리기  linex = startx + 40.0f \* ((float)j + 1.0f);  liney = starty + 40.0f \* (float)i;  ofDrawLine(linex, liney, linex, liney + 40.0f);  }  if (cell[i][j].down == 0) { // 해당 칸의 아랫쪽 벽 그리기  linex = startx + 40.0f \* (float)j;  liney = starty + 40.0f \* ((float)i + 1.0f);  ofDrawLine(linex, liney, linex + 40.0f, liney);  }  }  } |

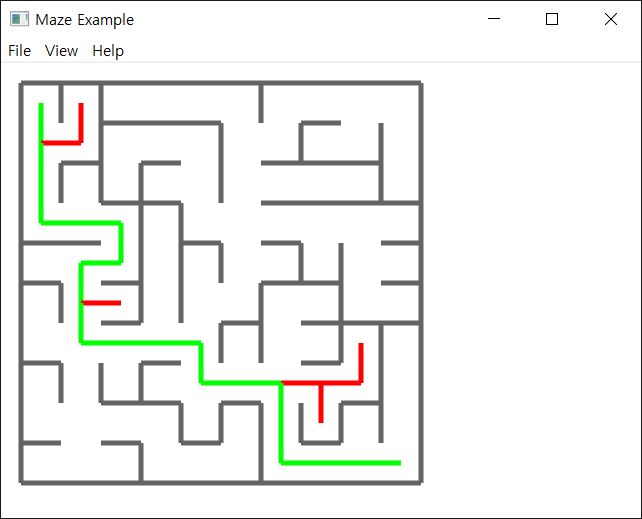
행 별로 왼쪽부터 오른쪽까지 cell 값을 활용하여 그림을 그린다. 여기서 startx와 starty는 각각 미로의 왼쪽 위 꼭지점이 그려질 x좌표와 y좌표를 뜻하며, 이번 실습에서는 (20, 20)을 왼쪽 위 꼭지점으로 설정했다. 그리고, 인접한 방과 방 사이의 길은 모두 40만큼의 길이를 가지도록 했다. 맨 윗줄과 맨 왼쪽 벽은 항상 그려주도록 하며, cell 값을 참조하여 각 방의 오른쪽이 막혀 있다면 그 쪽에 직선을 그리고, 아랫쪽이 막혀 있다면 그 쪽에 직선을 그려주었다(직선의 두께는 5.0이다). 이렇게 하면 방과 방 사이의 벽들 뿐 아니라 미로의 오른쪽 외곽선과 아랫쪽 외곽선도 cell에서 막혀있음을 확인할 수 있으므로 자동적으로 이들을 그려주게 된다.

다음으로 DFS 함수는 (0,0)에서 (width-1, height-1) 방까지의 경로를 탐색함과 동시에 그 경로를 그려주는 함수이다. 이 함수는 draw() 함수에서 호출되며, 그 코드는 다음과 같다.

|  |
| --- |
| bool ofApp::DFS()//DFS탐색을 하는 함수  {  vertex current;  float linex, liney, backx, backy;  //TO DO  //DFS탐색을 하는 함수 ( 3주차)  push(cell[0][0]);  visited[0][0] = 1;  current = mystack[top];  ofSetLineWidth(5.0f);  while (!stack\_isempty()) {  if (current.row == HEIGHT - 1 && current.col == WIDTH - 1)  return true;  ofSetColor(ofColor::green);  linex = startx + 20.0f + ((float)current.col \* 40.0f);  liney = starty + 20.0f + ((float)current.row \* 40.0f);  if (current.right && !visited[current.row][current.col + 1]) { // 오른쪽으로 이동 가능  current = cell[current.row][current.col + 1];  push(current);  visited[current.row][current.col] = 1;  ofDrawLine(linex, liney, linex + 40.0f, liney);  }  else if (current.down && !visited[current.row + 1][current.col]) { // 아랫쪽으로 이동 가능  current = cell[current.row + 1][current.col];  push(current);  visited[current.row][current.col] = 1;  ofDrawLine(linex, liney, linex, liney + 40.0f);  }  else if (current.col > 0 && cell[current.row][current.col - 1].right && !visited[current.row][current.col - 1]) { // 왼쪽으로 이동 가능  current = cell[current.row][current.col - 1];  push(current);  visited[current.row][current.col] = 1;  ofDrawLine(linex, liney, linex - 40.0f, liney);  }  else if (current.row > 0 && cell[current.row - 1][current.col].down && !visited[current.row - 1][current.col]) { // 윗쪽으로 이동 가능  current = cell[current.row - 1][current.col];  push(current);  visited[current.row][current.col] = 1;  ofDrawLine(linex, liney, linex, liney - 40.0f);  }  else {  ofSetColor(ofColor::red);  pop();  if (stack\_isempty()) break;  current = mystack[top];  backx = startx + 20.0f + ((float)current.col \* 40.0f);  backy = starty + 20.0f + ((float)current.row \* 40.0f);  ofDrawLine(linex, liney, backx, backy);  }  }  return false;  } |

DFS 알고리즘은 stack 자료구조를 이용하며, 오른쪽-아래-왼쪽-위쪽 순서대로 이동할 수 있는지를 확인하며 이동할 수 있으면 현재 방의 정보를 stack에 push한다. 그 후 이동하다가 막히면 이전 방으로 돌아가게 되며, 이 때 stack에서 pop함을 통해 그를 구현하게 된다. 그리고 가다가 막히는 길은 되돌아간다는 특성을 이용하여, DFS 함수에서 그 경로까지 같이 그리게 설정했다. 그리고 일련의 방문 후에는 방문한 방을 visited 배열을 통해 표시해주어야 한다. 방문하지 않은 방에 대해서 0으로 초기화되어 있으며, 방문한 방에 대해서는 1로 값을 바꾼다.

(0,0) 방을 시작점으로 하며, 그 시작점을 stack에 push하고 시작한다. 오른쪽-아래-왼쪽-위쪽 순서대로 이동할 수 있는지 탐색하며, 이동 가능하면 그 다음 방을 stack에 push하고, 그 두 방을 초록색으로 이어준다. 만약 사방 어떤 곳도 갈 수 없는 상황이 된다면, stack에서 pop을 진행하여, 그 이전 방으로 이동한다. 그리고, 그 두 방을 빨간색으로 이어주어 그 길이 틀렸음을 표시해준다. Stack이 빌 때까지 이 과정을 반복하여, 목적지인 (width-1, height-1) 방에 도착하면 true를 return하며 끝낸다. 이 과정을 거치면 미로가 정상적으로 그려지며, 결과는 다음과 같이 나타난다.

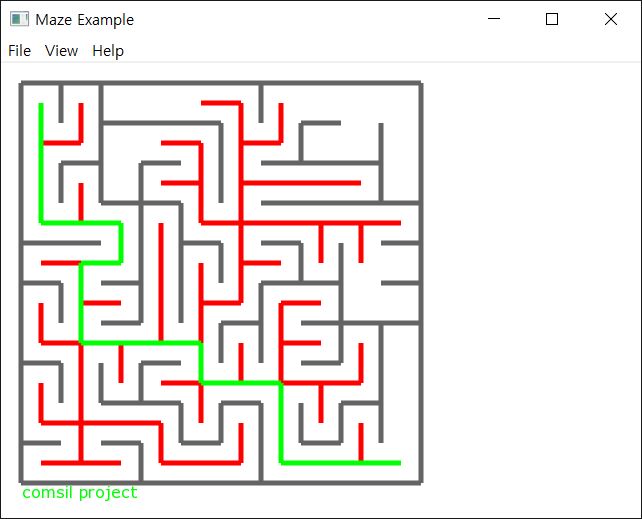


DFS 함수를 구현함에 있어서 조심해야 할 점은, 프로그램은 그림을 지속적으로 그려주는 것이나 stack과 visited 함수는 DFS 함수를 계속 호출함에 따라 계속 바뀐다. 그러므로, 매번 그릴 때마다 탐색을 다시 하게 하는 방식을 취하기 위해 stack을 비우고 visited 함수를 0으로 초기화해주는 과정을 draw() 함수 내에서 취해주어야 한다.

이어서 BFS 함수도 DFS 함수와 비슷한 방식으로, 탐색과 동시에 경로를 윈도우 상에 그려준다. 코드는 다음과 같다.

|  |
| --- |
| bool ofApp::BFS()  {  //TO DO  //BFS를 수행한 결과를 그린다. (3주차 내용)  vertex current;  float linex, liney;  addq(cell[0][0]);  visited[0][0] = 1;  ofSetLineWidth(5.0f);  while (!queue\_isempty()) {  current = deleteq();  ofSetColor(ofColor::red);  linex = startx + 20.0f + ((float)current.col \* 40.0f);  liney = starty + 20.0f + ((float)current.row \* 40.0f);  if (current.row == HEIGHT - 1 && current.col == WIDTH - 1) {  ofSetColor(ofColor::green);  while (current.row != 0 || current.col != 0) { // 찾은 루트 되감아가며 초록색으로 표시  switch (cell[current.row][current.col].came\_from) {  case left\_dir:  ofDrawLine(linex, liney, linex - 40.0f, liney);  current.col--;  break;  case up\_dir:  ofDrawLine(linex, liney, linex, liney - 40.0f);  current.row--;  break;  case right\_dir:  ofDrawLine(linex, liney, linex + 40.0f, liney);  current.col++;  break;  case down\_dir:  ofDrawLine(linex, liney, linex, liney + 40.0f);  current.row++;  break;  }  linex = startx + 20.0f + ((float)current.col \* 40.0f);  liney = starty + 20.0f + ((float)current.row \* 40.0f);  }  return true;  }  if (current.right && !visited[current.row][current.col + 1]) { // 오른쪽으로 갈 수 있는 경우  cell[current.row][current.col + 1].came\_from = left\_dir;  addq(cell[current.row][current.col + 1]);  visited[current.row][current.col + 1] = 1;  ofDrawLine(linex, liney, linex + 40.0f, liney);  }  if (current.down && !visited[current.row + 1][current.col]) { // 아랫쪽으로 갈 수 있는 경우  cell[current.row + 1][current.col].came\_from = up\_dir;  addq(cell[current.row + 1][current.col]);  visited[current.row + 1][current.col] = 1;  ofDrawLine(linex, liney, linex, liney + 40.0f);  }  if (current.col > 0 && cell[current.row][current.col - 1].right && !visited[current.row][current.col - 1]) { // 왼쪽으로 갈 수 있는 경우  cell[current.row][current.col - 1].came\_from = right\_dir;  addq(cell[current.row][current.col - 1]);  visited[current.row][current.col - 1] = 1;  ofDrawLine(linex, liney, linex - 40.0f, liney);  }  if (current.row > 0 && cell[current.row - 1][current.col].down && !visited[current.row - 1][current.col]) { // 윗쪽으로 갈 수 있는 경우  cell[current.row - 1][current.col].came\_from = down\_dir;  addq(cell[current.row - 1][current.col]);  visited[current.row - 1][current.col] = 1;  ofDrawLine(linex, liney, linex, liney - 40.0f);  }  }  return false;  } |

BFS 알고리즘은 queue 자료구조를 사용하며, queue에 저장한 다음 방의 좌표 순서대로 경로를 찾는다. 이 알고리즘에서도 시작점인 (0,0) 방을 queue에 push해주며 시작한다. 그 후 while문을 들어가, queue에서 pop을 진행하여 저장된 다음 좌표를 받아낸다. 마찬가지로 오른쪽-아래쪽-왼쪽-위쪽 순서대로 탐색하되, 이번에는 DFS와 다르게 해당 방에서 갈 수 있는 모든 방향을 저장해야 하므로 else if 대신 if를 사용하여 모든 경우에 대해 비교할 수 있도록 한다. 해당 방향으로 갈 수 있으면, 후술할 “올바른 경로 그리기”를 위해 came\_from 변수를 설정해준다. Came\_from 변수는 이동할 방 기준으로 어느 방향에서부터 이동한 것인지를 표시한다. 그리고 queue에 이동할 방을 push해주고, 그 경로를 빨간색으로 표시한다. 이 과정을 queue가 빌 때까지 반복하며, 목적지에 도착하면 마무리해야 한다. 이 때, 우리는 모든 경로를 빨간색으로 칠한 상태로, 어느 경로가 올바른 경로인지 아직 알 수 없다. 그래서, 목적지를 기준으로 아까 저장했던 came\_from 변수를 참조하여 시작점까지 경로를 역추적한다. 이는 하나의 방으로 올 때 그 방향은 여러 개가 될 수 없다는 perfect maze의 특성을 이용한 것이다. 목적지에서부터 이동한 방향의 반대로 추적하며 그 경로를 초록색으로 덮어 씌운다. 그러면 시작점까지 초록색으로 이어지며, 그것이 올바른 경로가 되는 것이다. 그 결과는 다음과 같다.



BFS 함수도 마찬가지로 매 프레임마다 그려주어야 하므로, visited와 queue를 초기화해주는 과정을 취해주어야 한다.

이 프로그램의 시간 복잡도를 고려해보자. 우선 파일을 읽는 함수에서 HEIGHT\*WIDTH만큼 2번 파일을 읽는다(벽을 고려하면 그의 2배 정도 읽지만, 시간 복잡도 측정 특성 상 이와 같이 표시한다). 그리고 미로를 그리는 함수 역시 HEIGHT\*WIDTH만큼 진행한다. DFS 함수는 미로의 상태에 따라 다르지만, 최악의 경우 HEIGHT\*WIDTH만큼의 방을 모두 탐색한다. 이는 BFS 함수도 마찬가지이다. 그러므로, 전체적인 알고리즘의 시간 복잡도는 O(HEIGHT\*WIDTH)가 된다.

다음으로 공간 복잡도의 경우, 우리가 이용한 가장 큰 자료구조는 이차원 배열(cell과 visited)이다. 그 중 cell 배열 안에는 구조체를 저장하기는 하지만, 그 구조체의 구성 요소의 개수는 일정하다. 그리고 배열 자체의 크기는 WIDTH\*HEIGHT이 된다. 그러므로, 전체적인 알고리즘의 공간 복잡도는 O(HEIGHT\*WIDTH)가 된다.

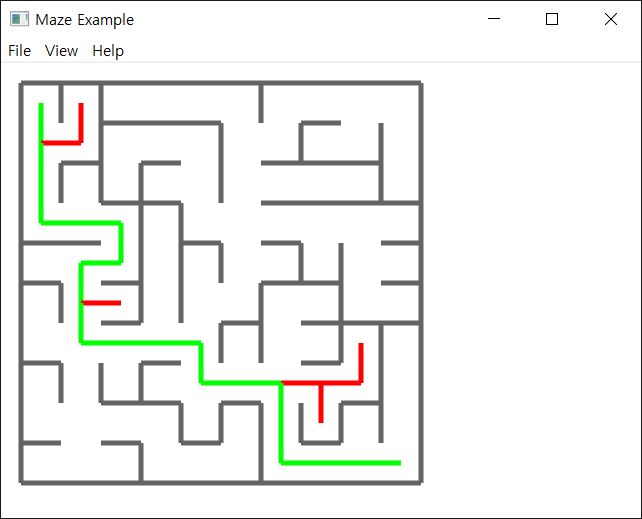
실험 전에는 cell이라는 이차원 배열에 각 방의 행 좌표와 열 좌표, 오른쪽으로 이동할 수 있는지 여부, 그리고 아래쪽으로 이동할 수 있는지 여부를 저장하면 된다고 생각했다. 이 생각은 DFS 알고리즘까지는 충분히 구현할 수 있었지만, BFS 알고리즘의 경우 “올바른 경로”를 그리는 데에 있어서 그것만으로는 부족했다. 그래서, BFS 알고리즘에 한정해서 came\_from이라는 변수가 추가로 필요했다. 그리고, 그림을 그리는 데에 있어서 stack와 queue를 이용하는 것은 맞았으나, 매 프레임마다 그림을 그리기 위해서는 stack과 queue를 초기화해주어야 한다는 사실을 모르고 있어, 첫 시도 때는 경로가 한 프레임 동안 그려졌다가 그 후에는 시작점과 끝 점이 빨간색으로 직선으로 쭉 이어져 있는 상황이 발생했었다. 그래서, 매 프레임마다 stack과 queue, 그리고 visited 배열까지 모두 초기화해주는 과정을 추가했다.

1. **자신이 설계한 프로그램을 실행하여 보고 DFS, BFS 알고리즘을 서로 비교한다. 각각의 알고리즘은 어떤 장단점을 가지고 있는지, 자신의 자료구조에는 어떤 알고리즘이 더 적합한지 등에 대해 관찰하고 설명한다.**

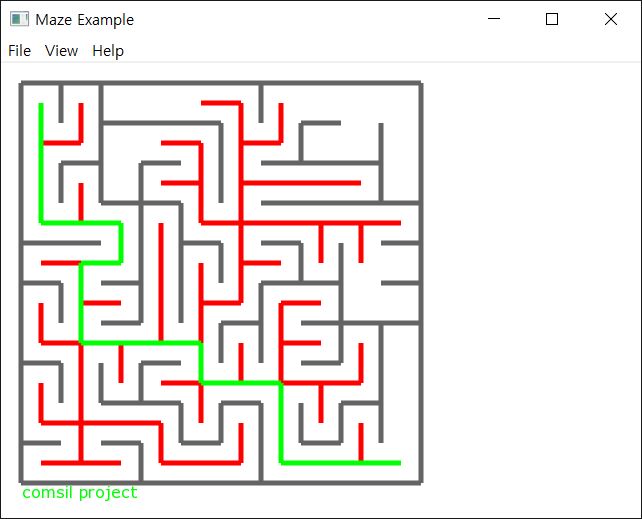
미로를 탐색하는 데에 있어서 DFS 알고리즘은 하나의 경로로 빠르게 이동할 수 있다는 장점이 있다. 그리고, 방을 차례차례 이동하기 때문에 보기 좀 더 직관적이다. 하지만, 만약 미로 내에서 한 번 길을 잘못 들게 되면, 그 쪽의 방들을 모두 탐색할 때까지 샅샅이 탐색할 수도 있다. 그리고, 이번 실습은 perfect maze를 이용한 실습이었지만, 만약 입력한 미로가 imperfect maze였다면 이 알고리즘은 최단 경로를 찾지 못하는 경우도 생긴다.

다음으로 BFS 알고리즘은 여러 경로를 한 번에 이동해보며 최단 경로를 찾을 수 있다는 장점이 있다. 즉 여러 가지 경우를 한 번에 시도해볼 수 있다는 것이며, 이에 따라 미로 내에서 한 경로에서 길을 잘못 들더라도 다른 경로에서 길을 찾을 수 있으므로 큰 문제가 되지 않는다. 또한, 마찬가지 이유로 미로가 imperfect maze여도 최단 경로를 찾을 수 있다. 하지만, queue를 이용해 탐색하는 과정에서 방을 이리저리 이동하기 때문에, DFS 알고리즘에 비해 직관성이 떨어지게 된다. 뿐만 아니라, 단순한 경로를 가지는 미로여도 너무 많은 경로를 탐색하게 되어 그 속도가 조금 더 느려질 확률이 높다.

이번 실습에서의 DFS 알고리즘과 BFS 알고리즘의 결과는 다음과 같이 나타났다.



<DFS 알고리즘>



<BFS 알고리즘>

이번 실습에서 사용한 자료구조는 이차원 배열이었는데, DFS 알고리즘에 비해 BFS 알고리즘이 눈에 띄게 많은 경로를 탐색한 것이 보였다. 그러나 위의 예시는 먼저 탐색한 방향인 오른쪽과 아래쪽으로 이동한 것이 대부분인 단순한 경로이기 때문에, DFS 알고리즘에서 막다른 길을 선택한 경우 이차원 배열의 인덱스를 하나하나 따라감을 따라 되돌아가야 한다는 점을 추가적으로 생각할 필요가 있어 보인다. 그리고, DFS 알고리즘으로는 imperfect maze에서의 최단 경로를 찾기 어려워 보이기도 한다. 그러므로, 매 탐색마다 많은 방들을 탐색함에도 불구하고 조금 더 확실한 길을 찾을 수 있는 BFS 알고리즘이 더 적합한 알고리즘이라고 생각된다.