13주차 결과보고서

전공: 수학/컴퓨터공학 학년: 3학년 학번: 20181288 이름: 윤성호

1. <알고리즘과 자료구조 설명>

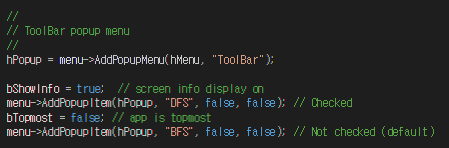
아래의 함수 설명은 12주차 실습과정에서 진행했던 코드에서 dfs, bfs 알고리즘을 활용하여 미로 탈출경로를 찾기 위해서 새롭게 정의하거나, 추가한 코드들을 정리한 것이다.

* Setup 함수



[사진 1]

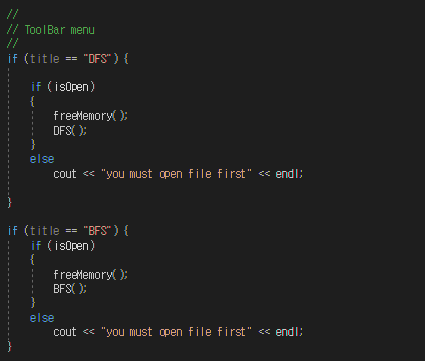
이번 실습에서는 dfs, bfs 알고리즘을 이용한 결과를 화면에 출력하는 것을 요구한다. 그렇게 하기 위해서 dfs, bfs 알고리즘을 동작시켰는지의 여부를 저장하는 변수가 필요하다. 나중에 draw함수에서 이 변수의 값을 조건문에 이용하여 dfs, bfs의 결과를 그릴 지 말지 결정하기 때문이다. [사진 1]은 프로그램이 시작할 때 setup 함수에서 모든 초기화 과정을 진행해주는데, 이 때, dfs함수의 실행부를 판단하는 변수인 isdfs와 bfs함수의 실행여부를 판단하는 변수인 isbfs를 모두 false로 초기화해준다. 함수가 실행되지 않았다면 false를 저장하고, 함수가 실행이 되었다면 true를 저장한다.



[사진 2]

이번 실습에서는 새로운 메뉴바를 하나 더 추가해주었다. 기본적으로 제공된 파일에는 FILE, VIEW, HELP 라는 총 3개의 메뉴바가 있었다. 각각의 메뉴바에는 하위 메뉴들이 존재하며, 이들을 누르면 각각 다른 행동을 수행한다. 메뉴바는 만들기 위해서는 가장 먼저 ofXWinMenu함수를 이용해 menu 객체를 하나 생성해야 한다. 그 이후, menu객체안의 CreateMenuFunction 매서드를 이용하여 appMenuFunction 함수를 사용할 수 있도록 함수를 등록해준다. 마지막으로, menu객체안에 AddPopupMenu 매서드를 불러준다. 그렇게 하면 추가 메뉴바를 생성 할 수 있는 준비가 완료된 것이다. ToolBar라는 메뉴바를 추가해주었는데, 이는 hPopup = menu->AddPopupMenu(hMenu, “ToolBar”); 을 작성함으로서 구현하였다. 이는 ToolBar라는 메뉴바를 추가하라는 코드로, 메뉴바들이 생성되는 코드들을 분석해본 결과 방법을 알 수 있었다. 이 후, ToolBar 메뉴바에 커서를 가져다 대면 DFS, BFS 라는 작은 하위메뉴들이 뜨도록 하였는데, 이는 menu객체의 AddPopupItem 매서드를 사용하면 만들 수 있다. 위에 설명한 과정은 [사진 2]에서 확인할 수 있다.

* appMenuFuncion 함수



[사진 3]

Setup함수에서 ToolBar라는 메뉴바를 새롭게 생성한 후, 하위 메뉴로 DFS, BFS를 생성하였다. 그런데 하위메뉴들을 클릭하면 어떠한 동작을 할지에 대해서는 정해주지 않았었는데, 이는 appMenuFuncition에서 명시해준다. 하위메뉴바를 클릭하게 되면 title 변수에 메뉴바의 이름이 들어가게 된다. 이를 통해 각각의 하위 메뉴바를 클릭하면 해야할 행동을 코드상에 적어줄 수 있게 된다. 먼저, DFS라는 하위메뉴를 클릭하면 미로 파일이 열려 있는지에 따라 다른 행동을 하게 된다. 미로 파일이 열려있지 않으면 “you must open file first” 라는 메세지를 출력해준다. 만약 미로 파일이 열려있다면, 그 이전의 정보를 모두 지워주기 위해 freeMemory 함수를 먼저 불러준다. 이 함수를 이용해 이전의 배열의 값이나 동적할당 된 메모리들을 모두 해제해주고, 새롭게 시작할 수 있는 상태로 되돌아 온다. 그 이후, DFS함수를 호출한다. BFS라는 하위메뉴를 클릭해도 마찬가지로 미로 파일이 열려 있는지에 따라 다른 행동을 하게 된다. 미로 파일이 열려있지 않으면 DFS와 같이 “you must open file first”라는 에러 메시지를 출력하고 다시 새로운 입력을 받을 준비를 한다. 만약 미로 파일이 열려있다면, 그 이전의 정보를 모두 지워주기 위해 freeMemory 함수를 불러준다. 이를 통해 다시 새롭게 시작할 수 있는 모든 준비를 마친 후, BFS 함수를 불러주어 BFS 알고리즘을 통해 미로를 탐색할 수 있도록 한다. [사진 4]를 확인하면 ToolBar의 하위메뉴를 클릭하였을 때 할 행동을 명시한 코드를 나타낸다.

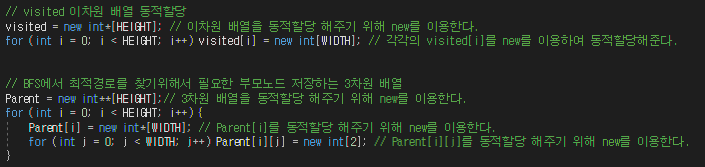
* Draw 함수



[사진 4]

오픈프레임워크에서 화면에 그림을 그리기 위한 함수들이나 과정들은 모두 draw 함수 내부에 적는 것을 암묵적인 규칙으로 한다. draw함수는 매 frame마다 한번 실행되는 함수이고, 화면에 그려주고 싶은 대상들을 그리기 위한 과정들을 코드로 작성한다. 그 중, 이번 실습 및 숙제에서는 dfs와 bfs 알고리즘을 활용하여 미로를 탈출하는 경로를 찾는 것이었고, 찾은 탈출경로를 화면에 그려주는 것이 실습과 숙제의 요구사항이었다. 또한, 탈출경로를 찾기위해 탐색한 경로 또한 화면에 그림으로 그려주어야 했다. 이를 위해서는 draw 함수에서 dfs함수의 실행여부와, bfs함수의 실행여부에 따라 그림을 그릴 지 말지 결정해주기 위해서 앞에서 정의해준 isdfs와 isbfs 변수들을 이용했다. 만약 DFS나 BFS 함수를 실행해주지 않았다면 변수들의 값은 false 일것이고, 따라서, dfs나 bfs 알고리즘을 통해 얻은 경로들을 화면에 그려주지 않을 것이다. 그러나, DFS, BFS 함수를 실행한 경우, 변수들의 값을 true로 바꿔주어 draw함수 내부에서 조건문 안의 내용 또한 실행되도록 하여 탐색경로와 최적 경로를 모두 그리게 할 것이다. 프로그램 상 isdfs, isbfs 변수가 true값이 되기 위해서는 그 이전에 isOpen이 true가 되어있어야 한다. 그러나, 프로그램의 오류를 방지하기 위해 isdfs, isbfs 조건문안에 isOpen의 true, false 여부에 따라 다른 행동을 하도록 코딩하였다. 따라서, 만약 DFS 함수가 실행되었는데 파일이 열리지 않은 상황이라면 에러 메시지를 출력한다. 이는 BFS 함수의 경우에도 마찬가지이다. 만약, 모든 것이 정상적으로 작동하고, DFS, BFS 함수가 호출된 경우에는, 각각 dfsdraw(), bfsdraw() 함수를 추가로 실행한다. 직접 그림을 그리는 과정은 dfsdraw()와 bfsdraw()함수에 코드로 구현되어 있다. [사진 4]는 draw함수에서 isdfs, isbfs 의 값에 따라 최적경로와 탐색경로를 그리는 지의 여부를 나타낸 코드이다.

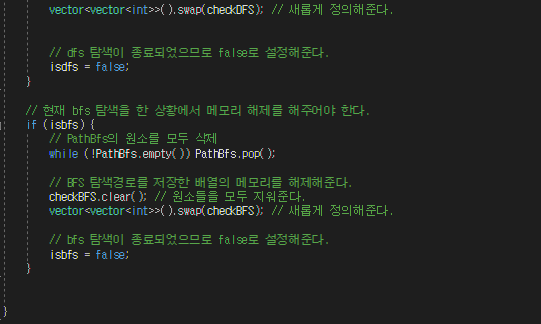
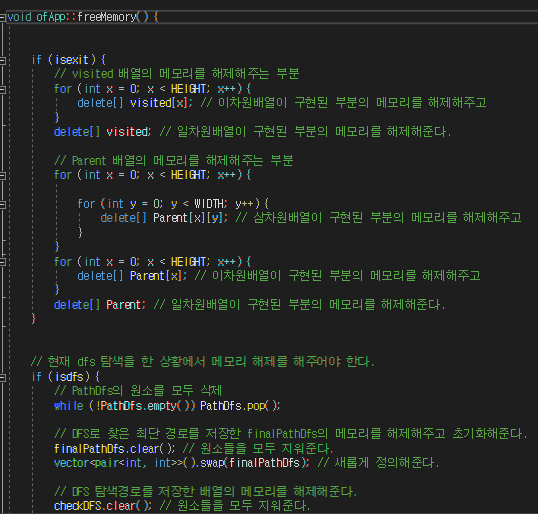
* readFile 함수



[사진 5]

readFile 함수는 .maz 파일 형식으로 주어진 미로 파일을 받아오고, 그래프 자료구조로 변환하여 dfs, bfs 과정을 진행할 수 있도록 하는 과정이다. 이번 실습에서는 dfs, bfs 알고리즘을 이용하여 미로 탈출경로를 찾아야 하기 때문에 이 함수에서 새로운 배열의 동적할당을 진행해주었다. 먼저, visited 라는 이름의 이차원 배열이다. 이 배열은 dfs, bfs 알고리즘을 진행할 때 필수적인 배열이다. 모든 지점의 방문여부를 저장하여, 방문하지 않았다면 경로탐색에 이용하고, 방문하였다면 다시 방문하지 않아서 알고리즘을 성공적으로 마무리하기 위한 도움을 주는 배열이다. 다음으로 Parent 라는 이름의 배열은 삼차원 배열이다. 이 배열은 BFS 함수에서 미로 탈출 경로를 찾기 위해서 쓰이는 배열이다. 알고리즘 특성상 미로 탈출 경로의 길이는 쉽게 알 수 있지만, 미로 탈출 경로는 바로 알 수 없다. 따라서, 탐색을 하며 탈출 경로를 만나게 되면 그 부모의 좌표들을 Parent 배열에 저장해두고, 미로의 탈출 경로를 그릴 때 도착점에서부터 시작하여, 초기 위치에 도착할 때까지 부모위치를 따라간다. 현재 위치를 (x,y) 라고 생각한다면, Parent[x][y][0]은 현재 위치의 부모위치의 x좌표이고, Parent[x][y][1]은 현재 위치의 부모위치의 y좌표이다. [사진 5]는 visited배열과 parent 배열을 동적할당 해주는 코드를 나타낸 것이다.

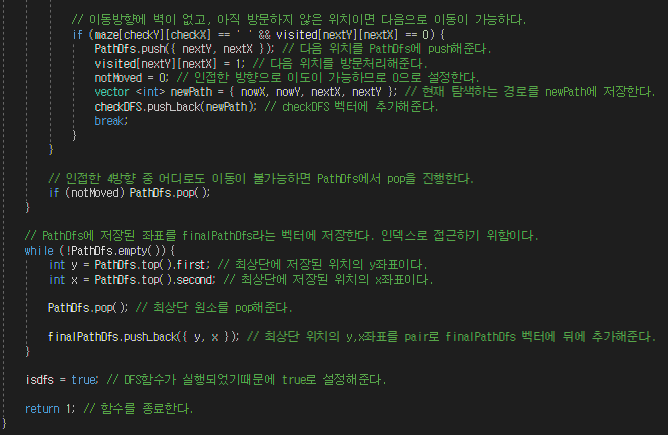
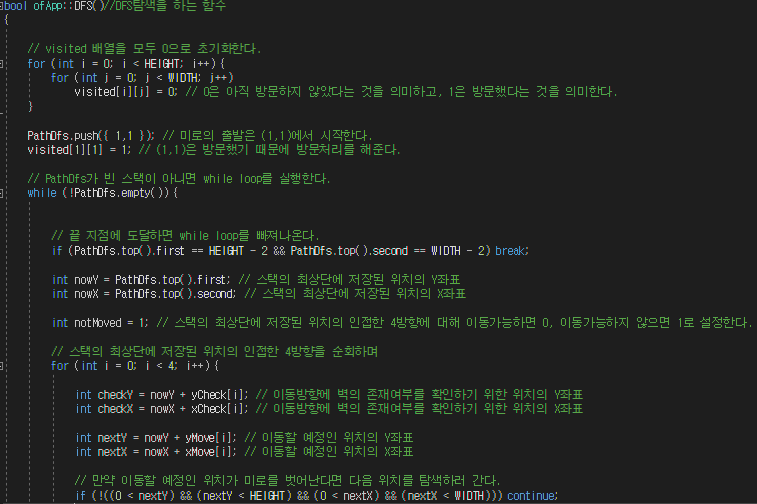
* freeMemory 함수



[사진 6]

freeMemory 함수는 동적할당된 배열이나 새롭게 정의한 자료구조들의 메모리를 해제해주는 기능을 한다. 먼저, isExit이면 해제해야 할 메모리에 대해 설명하도록 하겠다. isExit이 true가 된 상황은 프로그램이 완전히 종료가 된 상태이다. 이 때는 dfs, bfs 알고리즘 때 활용한 visited 배열의 메모리를 해제해준다. 이차원 배열이므로 두 번의 과정을 거쳐서 메모리를 완전히 해제해준다. 또한, bfs 알고리즘을 이용한 미로 탈출 경로를 출력하기 위해 사용한 Parent 배열 또한 해제해준다. 이 배열은 삼차원 배열이기 때문에 세 번의 과정을 거쳐서 메모리를 완전히 해제해준다. 다음으로 isdfs일 때의 메모리 해제이다. DFS 함수를 호출한 후, 이를 종료할 때는 isdfs는 아직 true이다. 이 때 해제해주어야 할 자료구조는 PathDfs와 checkDfs 이다. 이 두 자료구조는 c++ 라이브버리 중 stack과 vector 라이브러리에서 제공하는 stack과 vector 자료구조를 이용한 것이다. 이를 메모리해제 해주는 과정은 먼저, 내부의 원소들을 모두 제거해주고, 그 이후 빈 임시 객체와 swap 해주는 것으로 메모리 해제를 해줄 수 있다. 이 과정을 PathDfs와 checkDFs에 대해 모두 진행한다. 그 이후, 모든 메모리를 해제하였으므로 isdfs 변수를 false로 전환해주어 함수가 실행되기 전 상태로 돌아간다. 마지막으로 isbfs일 때의 메모리 해제이다. BFS 함수를 호출한 후, 이를 종료할 때는 isbfs는 아직 true이다. 이 때 해제해주어야 할 자료구조는 PathBfs와 checkBfs 이다. 이 두 개의 자료구조 또한 DFS 함수와 마찬가지로 각각 stack과 vector 자료구조를 사용하였다. 이를 메모리해제 해주는 과정은 먼저, 내부의 원소들을 모두 제거해주고, 그 이후 빈 임시 객체와 swap 해주는 것으로 메모리 해제를 해줄 수 있다. 이 과정을 PathBfs와 checkBFs에 대해 모두 진행한다. 그 이후, 모든 메모리를 해제하였으므로 isbfs 변수를 false로 전환해주어 함수가 실행되기 전 상태로 돌아간다. [사진 6]은 freeMemory 함수의 전체 코드를 나타낸 것이다.

* DFS 함수



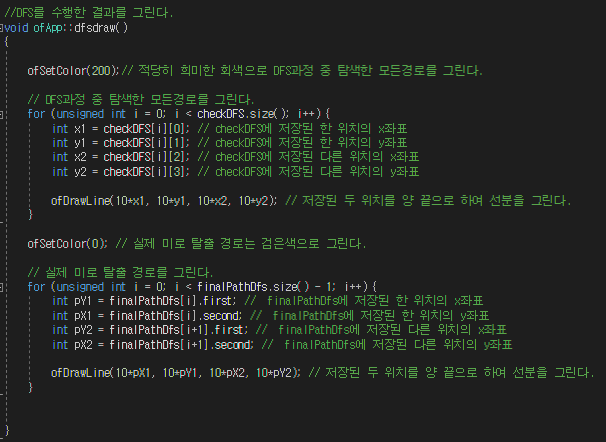
[사진 7]

이번 실습에서 해야하는 가장 주요한 요구사항 중 하나인 DFS 함수의 작성이다. Dfs 알고리즘은 일반적으로 recursive하게 구현한다. 코드가 직관적이기 때문이다. 그러나, 만약 탐색해야 하는 그래프의 크기나 깊이가 매우 깊은 경우에는 스택 오버플로우(stack overflow)가 날 수 있기 때문에 이번 실습에서는 iterative하게 구현하였다. 이렇게 하고, 함수 내부에서 새로운 스택 자료구조를 이용한다면 recursive한 구현에서 이용한 컴퓨터 내부의 stack memory보다 더 컨트롤 하기 쉽기 때문에 더 유연한 프로그래밍이 가능해진다. 가장 먼저, 방문여부를 저장하는 visited 배열을 0으로 초기화한다. 초기상태에는 모든 위치에 아직 방문하지 않았기 때문이다. PathDfs에는 미로 탈출 경로가 저장될 것이고, checkDfs에는 미로 탈출 경로를 찾기위해 탐색한 모든 경로가 저장될 것이다. PathDfs는 stack 라이브러리를 불러와 stack 자료구조를 사용하여 정의하였고, checkDfs는 vector 라이브러리를 불러와 vector 자료구조를 사용하여 정의하였다. 원소의 추가와 삭제가 쉽기 때문에 사용하게 되었다. 가장 먼저, 시작점은 (1,1)이므로 PathDfs에 (1,1)을 추가해준다. 그리고 이 위치를 방문처리 해준다. 그 다음으로는 PathDfs가 비어 있지 않다면 계속 반복을 할 while loop를 만들어주고, PathDfs에 원소가 존재한다면 반복문 안으로 들어온다. 이 후, 바로 PathDfs의 최상단 원소가 끝지점이라면 반복문을 종료하고, 그렇지 않다면 계속해서 dfs 알고리즘을 수행한다. 스택의 최상단 위치에서 그 다음위치로 이동해야 하기 때문에 현재 위치를 스택의 최상단 위치라고 생각한다. 그 이후, 아래쪽, 오른쪽, 위쪽, 왼쪽 순으로 이차원 배열을 탐색하며 미로에 벽이 존재하는 지 여부를 알아낸다. 만약 4방향 중 한 방향으로 이동이 외부로 나가지 않고, 벽이 없어서 이동이 가능하다면 PathDfs에 추가하고, 방문처리를 해주고 checkDfs에도 추가해준다. 이동을 했는지의 여부를 저장하는 notMoved 변수는 0으로 설정해준다. 그리고 만약 4방향 탐색 중 하나 이상의 방향으로 이동이 가능하다면 아무일도 일어나지 않지만 만약 4방향 탐색 중 모두 이동이 불가능하다면 최상단 원소를 제거한다. 그 원소는 탈출경로가 아님이 밝혀진 것이기 때문이다. 현재 탐색위치가 종료위치가 아니면서 4방향중 그 어느 방향도 새로운 길이 아니기 때문에 탈출경로가 될 수 없다. 이러한 과정을 거치면서 끝지점에 도달하면 while loop가 종료되고, PathDfs에는 미로 탈출 경로가, checkDFs에는 탈출 경로를 찾기위해 확인한 경로들이 저장되어 있을 것이다. PathDfs는 stack 자료구조 형태기 때문에 인덱스를 통해 접근이 불가능하다. 따라서 finalPathDfs 라는 vector 자료구조에 미로 탈출경로를 복사한다. 이렇게 된다면 finalPathDfs를 이용하여 미로의 탈출경로를 화면에 그릴 수 있게 된다. 그리고 모든 과정이 마무리 된다면 isdfs 값을 true로 변환하여 DFS 함수가 호출 된 이후의 행동을 할 수 있도록 한다. [사진 7]은 DFS 함수의 코드를 나타낸 것이다.

DFS 함수의 시간복잡도를 생각해보자. 가장 처음 visited 배열을 초기화하는 과정에서는 이중 loop를 돌고 있기 때문에 O(HEIGHT\*WIDTH)만큼 걸린다. 그 이후, dfs 알고리즘을 위한 while loop가 시작한다. 이 후, 현재 위치에 대해서 아래쪽, 오른쪽, 위쪽, 왼쪽 순으로 탐색을 진행하기 때문에 각 위치마다 최대 4번의 탐색이 진행된다. 그리고 이는 인접 리스트 방식의 그래프 탐색에 더 가까운 방식이다. 이러한 탐색과정이 각 위치마다 진행된다. 미로의 가로는 N, 미로의 세로는 M이라고 할 때, 노드의 개수는 N\*M이라고 이야기 할 수 있고, 최악의 상황의 경우 모든 노드에 대해 4방향 탐색을 진행해야 될 수도 있다. 따라서 while loop의 시간복잡도는 O(4\*N\*M) = O(N\*M)이다. 그 이후 isdfs의 값을 변화하는 과정은 상수시간이므로 O(1)이다. 이 모든 과정을 합한다면 O(HEIGHT\*WIDTH + N\*M + 1)인데 HEIGHT = 2\*M + 1 이고, WIDTH = 2\*N + 1이므로 최종적인 시간복잡도는 O((2\*M+1)\*(2\*N+1) + N\*M + 1) = O(N\*M)이다.

DFS 함수의 공간복잡도를 생각해보자. 먼저, 방문여부를 저장할 전역변수인 visited 배열이 사용되므로 여기에는 O(HEIGHT\*WIDTH)가 사용된다. 다음으로 PathDFs와 checkDfs 자료구조에서도 공간복잡도를 생각해주어야 하는데, 먼저, PathDFs는 미로를 탈출하는 경로를 저장하는 stack이고, checkDfs는 미로 탈출경로를 저장하는 vector이다. 만약 미로가 세로줄이 없고 가로줄이 가로 사이즈보다 1 작은 크기로 행마다 왼쪽, 오른쪽으로 붙어있는 상황이 되면 미로의 탈출경로는 미로의 모든 지점을 지나가는 상황이고, 따라서 탐색경로 또한 모든 지점을 지나가는 상황일 것이다. 이 상황에서 미로의 가로를 N, 미로의 세로를 M이라고 한다면 공간복잡도는 O(N\*M)이 될것이다. 마지막으로 finalPathDfs vector 또한 공간복잡도는 O(N\*M)이 될 것이다. 따라서, 이 모든 과정을 합한다면 O(HEIGHT\*WIDTH + N\*M)인데 HEIGHT = 2\*M + 1 이고, WIDTH = 2\*N + 1이므로 최종적인 시간복잡도는 O((2\*M+1)\*(2\*N+1) + N\*M) = O(N\*M)이다.

* dfsdraw 함수



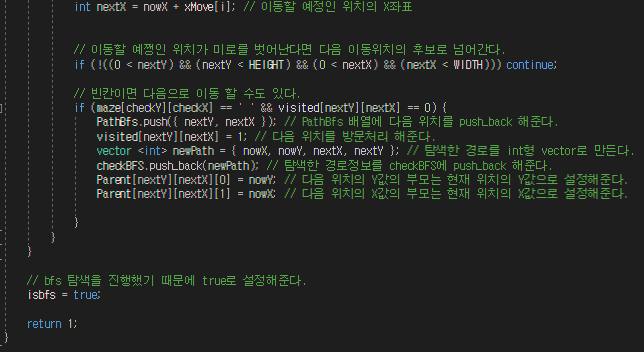
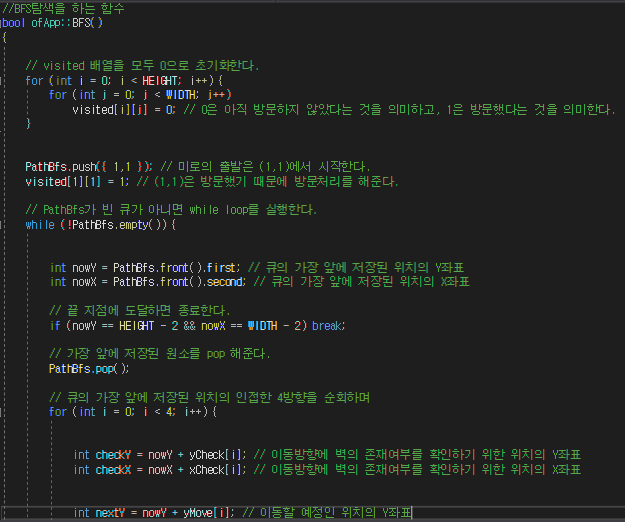
[사진 8]

Dfsdraw 함수는 DFS 함수를 호출하여 얻어낸 미로 탈출경로(finalPathDfs)와 탐색경로(checkDfs)를 화면에 그려준다. 두가지 경로를 구분하기 위해 그려주는 색을 다르게 해야한다. 이는 ofSetColor함수를 이용하면 가능하다. 먼저, 탐색경로는 적당히 희미한 회색으로 그려줄 것이다. checkDfs에는 미로의 탐색경로가 모두 담겨있으므로, 하나의 원소에 양 끝점 정보가 들어있다. 즉, x값 두개, y값 두개 총 4개의 정보가 들어가 있는 것이다. 이 들을 이용하여 ofDrawLine을 이용해 탐색경로를 그려준다. 그릴 때 모든 좌표에 10을 곱해주어 스케일을 조금 키웠다. 같은 방식으로 미로탈출경로도 화면에 그린다. 이는 앞의 탐색경로와 구분하기 위해 검은색으로 그려준다. 그 이후, checkDfs와 같은 방식으로 화면에 그려준다. [사진 8]은 dfsdraw 함수의 코드를 나타낸 것이다.

Dfsdraw의 시간복잡도를 생각해보자. 두 가지 정보를 그려주기 위해 두가지 반복문을 사용하였다. 먼저, 탐색경로를 출력하기 위한 반복문을 살펴보면 chcekDfs의 크기만큼 반복하고, 미로 탈출경로를 출력하기 위한 반복문을 살펴봐도 finalPathDfs의 크기만큼 반복한다. 즉, 시간복잡도는 자료구조의 크기에 비례한다는 것을 알 수 있다. 그런데 DFS 함수에서 알아보았듯이 checkDfs와 finalPathDfs는 최대 O(N\*M)개의 원소를 가질 수 있다고 했다. 즉, 각각의 반복문은 최대 O(N\*M)번 반복한다는 것을 의미한다. 이러한 반복문이 두 번 나타난다. 따라서, 시간복잡도는 O(2\*N\*M) = O(N\*M) 이다.

Dfsdraw의 공간복잡도를 생각해보자. 사용하는 자료구조는 checkDfs와 finalPathDfs라는 전역변수들이다. 이들은 O(N\*M)의 공간복잡도를 가진다. 따라서 공간복잡도는 O(2\*N\*M) = O(N\*M)이다.

* BFS 함수



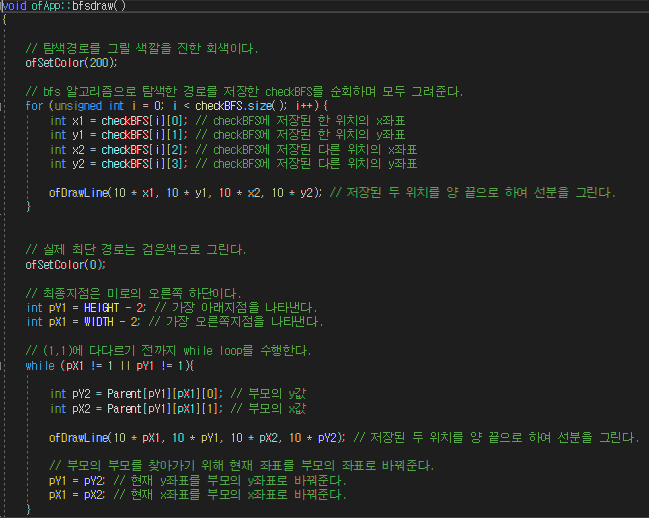
[사진 9]

이번 숙제에서는 실습시간 코드를 확장하여 BFS 함수를 작성하는 것을 요구한다. 가장 먼저, 방문 여부를 저장할 visited 배열을 초기화한다. Bfs 알고리즘을 활용하는 과정에는 PathBfs, checkBfs, Parent 가 필요하다. 먼저, PathBfs는 bfs 알고리즘을 구현하는 데 꼭 필요한 큐 자료구조이다. BFS 함수가 마무리되면 PathBfs에는 크게 의미있는 자료가 남아있는 것은 아니다. checkBfs는 미로 탈출 경로를 찾으면서 탐색했던 경로들을 모두 저장하고 있다. 마지막으로 Parent 배열은 현재 위치의 부모 위치를 저장하는 삼차원 배열이다. 이 세 가지 자료구조를 이용하면 bfs 알고리즘을 사용하여 숙제에서 요구했던 사항들을 모두 충족하여 프로그래밍을 진행 할 수 있게 된다. Visited 배열의 초기화가 종료되었다면, PathBfs에 초기위치를 넣어주고 방문처리를 해준다. 그 이후, while loop에 들어오게 되는데 탈출 조건은 큐가 비어있을 시, 혹은 끝 지점에 도달했을 시이다. 이러한 경우가 아니라면 큐의 가장 앞 원소인 위치를 뽑아서, 그 위치에서 아래쪽, 오른쪽, 위쪽, 왼쪽을 차례로 탐색하며 갈 수 있는 곳이면 큐에 넣고 checkBfs 배열에도 넣고 parent 배열도 업데이트 해주고, 갈 수 없는 곳이면 큐에 넣지 않고 무시한다. 이 과정을 반복하여 끝 지점을 만나게 되면 while loop이 종료되고, 모든 bfs 알고리즘의 과정이 끝났기 때문에 isbfs 변수의 값을 true로 변환해준다. 그렇게 하고 함수를 종료해준다. [사진 9]는 BFS 함수의 코드를 나타낸 것이다.

BFS 함수의 시간복잡도를 생각해보자. 가장 처음 visited 배열을 초기화하는 과정에서는 이중 loop를 돌고 있기 때문에 O(HEIGHT\*WIDTH)만큼 걸린다. 그 이후, bfs 알고리즘을 위한 while loop가 시작한다. 이 후, 현재 위치에 대해서 아래쪽, 오른쪽, 위쪽, 왼쪽 순으로 탐색을 진행하기 때문에 각 위치마다 최대 4번의 탐색이 진행된다. 그리고 이는 인접 리스트 방식의 그래프 탐색에 더 가까운 방식이다. 이러한 탐색과정이 각 위치마다 진행된다. 미로의 가로는 N, 미로의 세로는 M이라고 할 때, 노드의 개수는 N\*M이라고 이야기 할 수 있고, 최악의 상황의 경우 모든 노드에 대해 4방향 탐색을 진행해야 될 수도 있다. 따라서 while loop의 시간복잡도는 O(4\*N\*M) = O(N\*M)이다. 그 이후 isbfs의 값을 변화하는 과정은 상수시간이므로 O(1)이다. 이 모든 과정을 합한다면 O(HEIGHT\*WIDTH + N\*M + 1)인데 HEIGHT = 2\*M + 1 이고, WIDTH = 2\*N + 1이므로 최종적인 시간복잡도는 O((2\*M+1)\*(2\*N+1) + N\*M + 1) = O(N\*M)이다.

BFS 함수의 공간복잡도를 생각해보자. 먼저, 방문여부를 저장할 전역변수인 visited 배열이 사용되므로 여기에는 O(HEIGHT\*WIDTH)가 사용된다. 다음으로 PathBFs와 checkBfs 그리고 Parent 자료구조에서도 공간복잡도를 생각해주어야 하는데, 먼저, PathBFs는 bfs 알고리즘을 구현하기 위한 queue이고, checkBfs는 미로 탈출경로를 저장하는 vector이다. 그리고 Parent 배열은 현재 위치의 부모 위치를 저장하는 삼차원 배열이다. 만약 미로가 세로줄이 없고 가로줄이 가로 사이즈보다 1 작은 크기로 행마다 왼쪽, 오른쪽으로 붙어있는 상황이 되면 미로의 탈출경로는 미로의 모든 지점을 지나가는 상황이고, 따라서 탐색경로 또한 모든 지점을 지나가는 상황일 것이다. 이 상황에서 미로의 가로를 N, 미로의 세로를 M이라고 한다면 공간복잡도는 O(N\*M)이 될것이다. 마지막으로 Parent라는 이름의 삼차원 배열의 공간복잡도는 O(HEIGHT\*WIDTH\*2)이 될 것이다. 따라서, 이 모든 과정을 합한다면 O(HEIGHT\*WIDTH + N\*M + HEIGHT\*WIDTH\*2)인데 HEIGHT = 2\*M + 1 이고, WIDTH = 2\*N + 1이므로 최종적인 시간복잡도는 O((2\*M+1)\*(2\*N+1) + N\*M +((2\*M+1)\*(2\*N+1) \*2) = O(N\*M)이다.

* dfsdraw 함수



[사진 10]

bfsdraw 함수는 BFS 함수를 호출하여 얻어낸 부모위치 정보(Parent)를 이용하여실제 탈출 경로를 화면에 그려주고 탐색경로(checkDfs) 또한 화면에 그려준다. 두가지 경로를 구분하기 위해 그려주는 색을 다르게 해야한다. 이는 ofSetColor함수를 이용하면 가능하다. 먼저, 탐색경로는 적당히 희미한 회색으로 그려줄 것이다. checkBfs에는 미로의 탐색경로가 모두 담겨있으므로, 하나의 원소에 양 끝점 정보가 들어있다. 즉, x값 두개, y값 두개 총 4개의 정보가 들어가 있는 것이다. 이 들을 이용하여 ofDrawLine을 이용해 탐색경로를 그려준다. 그릴 때 모든 좌표에 10을 곱해주어 스케일을 조금 키웠다. 그리고 검은색으로 실제 경로를 그려주어야 한다. 이 과정에서는 도착점부터 시작하여 부모위치를 타고 올라가며 초기위치가 나오면 종료하도록 반복문을 구성하였다. 매번 부모위치를 타고 올라갈 때 마다 화면에 미로 탈출경로를 그려준다. [사진 10]은 bfsdraw함수의 코드를 나타낸 것이다.

bfsdraw의 시간복잡도를 생각해보자. 두 가지 정보를 그려주기 위해 두가지 반복문을 사용하였다. 먼저, 탐색경로를 출력하기 위한 반복문을 살펴보면 chcekBfs의 크기만큼 반복하고, 미로 탈출경로를 출력하기 위한 반복문을 살펴보면 부모노드를 따라 올라간 횟수만큼 반복했을 것이다. 그런데 DFS 함수에서 알아보았듯이 checkDfs는 최대 O(N\*M)개의 원소를 가질 수 있다고 했다. 또한, Parent 배열을 이용해 부모노드를 최대로 따라올라 가기 위해서는 최대 O(N\*M)이 걸린다. 즉, 두개의 반복문은 최대 O(N\*M)번 반복한다는 것을 의미한다. 따라서, 시간복잡도는 O(2\*N\*M) = O(N\*M) 이다.

bfsdraw의 공간복잡도를 생각해보자. 사용하는 자료구조는 PathBfs, checkbfs, parent 라는 전역변수들이다. PathBfs와 checkbfs는 O(N\*M)의 공간복잡도를 가진다. Parent 배열의 공간복잡도는 (N\*M\*2)이다. 따라서 공간복잡도는 O(N\*M + N\*M + N\*M\*2) = O(N\*M)이다.

<실험 전의 생각과 다른 점>

실험 이전에 pseudo code를 여러 번 읽고, 구글링을 통해 dfs, bfs 공부를 어느정도 한 상태였다. 또한, 예비 레포트를 작성하면서 DFS, BFS 알고리즘을 실습에서 어떤 식으로 적용할 지 또한 모두 생각을 많이 한 후 실습에 임했기 때문에 실습 전과 후에 큰 차이는 없는 것 같다. 그러나 실험 전에는 BFS 알고리즘으로 탈출경로를 얻어내는 과정이 잘 와닿지 않았다. 하나의 지점의 부모노드는 여러 개가 될 것 같은데 Parent 배열을 이용하여 탈출경로를 이용하기 위해서는 하나의 위치의 부모위치는 하나라는 전제가 깔려 있어야 하는데, 그것이 코드로 구현하는 것이 전에는 잘 상상이 가지 않았고, dfs와 같은 방식으로 PathDfs와 같이 결과적으로 미로 탈출 경로를 배열로서 얻어낼 수 있을 줄 알았다. 그러나, 실제로 구현해보니 큐를 이용하게 되므로 계속해서 큐의 원소는 빠져나가고 있기 때문에 결과적으로 남겨진 큐에는 미로 탈출 경로가 있을 수 없다는 사실을 알게 되었다.

1. Dfs, bfs 과정을 순조롭게 진행하기 위해서 이차원 배열을 그래프로 해석하여 탐색하기로 생각했다. 이차원 배열은 아래쪽, 오른쪽, 위쪽, 왼쪽으로 이동하면서 순회가 가능하며, 각 위치마다 총 4개의 방향으로 탐색이 가능하다. 이 탐색을 원활하게 해주기 위해서 4개의 방향을 가리키는 배열을 여러 개 정의해놓았다. xCheck, yCheck, xMove, yMove이 그것들이다. xCheck, yCheck는 다음 위치에 벽이 있는 지 여부를 체크하는 것이고, xMove, yMove를 이용하여 벽이 없다면 실제로 이동을 시키는 것이다. 이렇게 구분을 해놓은 이유는 미로가 생성된 방식 때문이다. 미로 파일의 모든 행이 빈칸은 아니다. 벽을 출력하는 행 또한 존재하기 때문이다. 따라서, 벽이 있는지 여부를 검사하는 것은 한칸 이후를 보면 되지만 이동은 이차원 배열상에서 두칸을 이동해야 한칸을 이동한 효과가 난다. 이를 구현하기 위해 xCheck, yCheck를 이용해 체크할 수 있는 범위는 각 축마다 최대 1이고, xMove, yMove를 이용해 이동할 수 있는 범위는 각 축마다 최대 2이다.

이번 실습에서는 iterative dfs 알고리즘을 이용하여 미로 탈출 경로를 찾아보았다. Dfs 에서 가장 중요한 것은 모든 노드들을 중복없이, 빠짐없이 방문할 수 있어야 한다는 점이다. 먼저, 중복없이 방문하기 위해서 visited 배열을 사용하였다. 그리고 빠짐없이 방문할 수 있기 위해서는 모든 위치에서 이동가능성이 있는 위치(아래쪽, 오른쪽, 위쪽, 왼쪽)를 모두 탐색해야 한다. 이를 위해 xCheck, yCheck과 for loop을 이용하였으며, 벽이 없을 시 xMove, yMove를 이용해 다음 칸으로 이동하였다. 이 알고리즘은 모든 과정이 끝나면 PathDfs 라는 스택에 미로 탈출 경로가 저장된다. 이는 iterative dfs 알고리즘의 특성이라고 생각한다. Recursive dfs에서는 실제 탈출 경로를 구하기 위해서는 직접 추가적인 과정을 진행해주어야 했다. 그러나, iterative에서는 그런 과정없이 바로 탈출 경로를 구할 수 있으므로 이런 점에서 굉장히 강력한 알고리즘이라고 생각한다. 그러나, 이러한 장점도 있는 반면에 코드 그 자체는 훨씬 덜 직관적이고, 코드량도 더 많은 것이 문제다.

Bfs 에서 가장 중요한 것은 모든 노드들을 중복없이, 빠짐없이 방문할 수 있어야 한다는 점이다. 먼저, 중복없이 방문하기 위해서 visited 배열을 사용하였다. 그리고 빠짐없이 방문할 수 있기 위해서는 모든 위치에서 이동가능성이 있는 위치(아래쪽, 오른쪽, 위쪽, 왼쪽)를 모두 탐색해야 한다. 이를 위해 xCheck, yCheck과 for loop을 이용하였으며, 벽이 없을 시 xMove, yMove를 이용해 다음 칸으로 이동하였다. 이 알고리즘이 앞서 본 DFS와 다른 점은 끝 지점을 만났을 때, 알고리즘이 종료되면서 pathDfs에는 미로 탈출 경로가 알아서 저장이 되어 있다. 그러나, BFS 함수에서는 그것이 불가능하다. 따라서, Parent 배열을 하나 추가로 두어, 모든 위치의 부모 위치를 저장하고, 탐색이 완료 된 이후에는 가장 끝 지점부터 부모를 타고 올라가며 초기 위치까지 올라간 후 탐색경로 그림을 그려준다. 함수를 구현하기 쉬운 대신 미로 탈출 경로를 알기 위해서는 상대적으로 복잡한 과정이 필요하다.

내가 생성한 자료구조는 일반적인 이차원 배열이다. 이차원 배열의 4방향을 모두 돌며, 이동이 가능하면 그 위치를 스택이나 큐에 넣고, 그렇지 않으면 무시한다. 이러한 과정은 DFS나 BFS나 크게 다른 점은 없을 것 같긴 하다. 그러나, maze project에서 요구하는 바는 미로 탈출 경로를 찾는 것이다. 이게 가장 주요한 요구사항이므로, 이를 충족시키는데는 bfs 보다 dfs가 낫다. 그 이유는 dfs 알고리즘은 함수가 종료되면 PathDfs 자료구조 안에 탈출 경로가 저장이 되어 있으므로 미로 탈출 경로를 얻기 위해 추가적인 처리를 해줄 필요가 없다. 하지만 bfs의 경우에는 bfs를 돌리던 큐를 모든 과정이 끝난 후 살펴보면 아무 의미없는 자료들이 담겨 있는 것을 알 수 있다. 탈출 경로는 하나의 배열에 저장되어 있는 것이 아니라 Parent라는 배열을 잘 순회해서 얻을 수 있도록 프로그래밍을 할 수 밖에 없다. 따라서, 자료구조에 따라서 bfs, dfs 중 더 적합한 알고리즘을 선택하는 것은 쉽지 않지만, 문제의 요구사항에 맞추기 위해서는 DFS 함수를 이용하는 것이 더 좋을 것 같다.