11주차 결과보고서

전공: 수학과,컴퓨터공학과 학년: 3학년 학번: 20181256 이름: 김도현

1-1.

Eller’s 알고리즘을 이용해 완전 미로를 구현하기 위해 사용한 알고리즘 및 자료 구조를 코드 사진을 통해 설명하겠다.

텍스트이(가) 표시된 사진

자동 생성된 설명

우선 따로 .h 헤더파일을 이용하지 않고 .c 파일로만 알고리즘을 구현 했기 때문에 변수 선언을 먼저 해주었다. 우선 입출력에 사용할 <stdio.h>, 동적 메모리 할당을 해주는 <stdlib.h>을 #inlcude 했다. WIDTH는 미로의 가로 길이, HEIGHT 는 미로의 세로 길이로 이 길이는 scanf를 통해 받아왔다. 그리고 나서 FILE \*fp 로 파일을 불러올 것을 선언하고 fopen 을 통해 “wt’ 즉 maze.maz 새 파일을 만들고 거기에 내용을 입력 한다. 이후 미로는 random 하게 side벽이나 under 벽을 지워야 하는데 실제 rand 만 가지고 사용하면 rand에 이용되는 seed 값이 한 번 선언되면 고정되기 때문에 항상 같은 미로만 생성이 된다. 이를 방지하기 위해 <time.h>를 include 선언해주고 srand함수를 통해 time을 seed 값으로 가져와 실행시마다 난수 값이 다르게 나오도록 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

다음 과정은 미로를 생성하기 전의 변수 선언 및 미로 생성에 필요한 초기화 과정이다. 우선 가장 윗 줄은 WIDTH 개수 만큼 ‘+-‘를 출력하여 실습 예시와 같은 모양이 나오도록 한다. 미로의 가장 오른쪽 위 모서리는 ‘+’로 끝나야 하기 때문에 이 역시 출력하고 “\n” 으로 다음 행으로 넘겨 이제 미로의 1행을 받을 수 있도록 한다. 이후 동적할당을 통해 maze 배열, side배열, under 배열을 각 각 WIDTH,WIDTH-1,WIDTH 만큼의 크기로 받는다. Side배열은 결국 방과 방 사이의 벽이기 때문에 개수는 WIDTH-1개가 필요하기 때문에 최소한의 메모리만 동적 할당 받는다. 이때 maze 배열은 한 행에 들어있는 방들이 속해 있는 집합을 저장하는 배열이다. Under 배열은 maze 배열과 비교를 통해 수직 경로를 통해 만들지 말지 결정하는 배열로 이 역시 집합이 저장된다. Side 배열은 maze배열의 방마다 집합을 통해 비교한 후 side벽을 만들지 말지를 저장하는 배열로 집합이 아닌 ‘0’,’1’ 로만 구성된 배열이다. 우선 첫 행이 시작되어야 하는데 maze배열에 0부터 WIDTH-1까지, 각 index 마다 각 index 자체의 숫자를 저장해 서로 다른 집합을 초기에 형성하도록 한다. 이후 set\_number 변수 역시 수직경로 생성시에 새로운 집합에 저장할 수 있도록 하는 변수로 WIDTH-1을 저장하였다. Set\_number 사용 전에 set\_number++ 을 해주어서 새로운 집합의 번호는 WIDTH로 이는 현재의 집합에 없는 숫자이다. 다음은 under\_flag 변수로 수직경로 생성과정에서 현재 집합 중 수직 경로가 있는지 없는지 판단하는 flag이다. Eller’s 알고리즘은 같은 집합 당 최소한 1개의 수직경로를 가지고 있어야 한다. 이 변수를 이용하여 해당 집합에 대하여 수직경로가 없다면 그 이전 방의 수직 경로를 뚫어서 수직 경로를 하나 만들 수 있게 할 것이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

HEIGHT만큼 반복문을 돌면서 maze에 집합을 만들면서 print로 출력해 나간다. 처음에 왼쪽 사이드벽 “|” 를 출력하고 이후 side 배열을 모두 1로 초기화 시켜준다. 0은 벽이 없다는 뜻으로 이는 앞으로의 과정에 따라 결정 할 것이다. (1은 방 사이에 벽이 존재한다는 것이다.) 반목문을 WIDTH만큼 돌면서 side벽을 뚫을지 말지 결정한다. 결국 side벽은 WIDTH-1 만큼 존재하기 때문에 인덱스 오류가 나지 않도록 i<WIDTH-1 일 때만 다음 과정을 시행했다. 만약 maze[i]!=maze[i+1] 즉 인접한 두 방 사이의 집합이 다를 경우 rand()%2, 50프로 확률로 벽을 지우거나 j=HEIGHT-1 즉 가장 마지막 행을 입력할 반복문에 해당한다면 무조건 벽을 지워야 한다. (만약 두 방 사이의 집합이 같다면 불완전 미로, 즉 한 목적지에 대하여 두개 이상의 경로가 생길 수 있기 때문에 지워서는 안된다.) 즉 이 경우 side[i]를 0으로 설정한다. 이때 벽을 뚫으면 같은 집합에 속해지게 되는데 집합이 더 작은 쪽으로 속할 수 있도록 maze[i+1], maze[i]의 크기 비교를 통해 maze[i+1]이 더 크다면 현재 maze[i+1] 과 같은 집합을 모두 찾아 이를 maze[i]로 바꿔준다. Maze[i]가 더 크다면 maze[i] 와 같은 집합을 모두 찾아 이를 maze[i+1]로 바꿔준다. 이렇게 side[i]가 0이면 side 벽 없이 스페이스바만 출력해주고 side[i]==1 이라면 side 벽인 “|” 를 출력해준다. For문이 모두 끝나면 한 행의 가장 마지막 열의 테두리 벽을 출력하고 엔터를 입력하여 (“|\n” ) 다음 수직경로에 해당하는 under 벽을 그릴 수 있게 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

다음은 수직경로를 탐색하여 under 벽을 생성하는 코드이다. 우선 반복문을 WIDTH만큼 도는데 index 오류가 나지 않도록 i==0일 때와 아닐 때와 구분 지었다. 우선 i==0일 때 maze[i]==maze[i+1] 즉 인접한 두 방의 집합이 같은지 비교하고 같다면 rand 함수를 이용해 50프로 확률로 수직경로를 생성한다. 이때 under\_flag=1로 둔다. 그리고 under[i]=maze[i]로 두어 under 배열의 i 번째에 위의 maze i와 같은 집합임을 알리고 수직경로가 생성된 것이다. 만약 50프로 확률로 수직경로가 생성되지 않았다면 set\_number를 1 추가하고 under[i] 에는 이를 넣어주어 새로운 집합을 만든다. 이때 under\_flag=0으로 아직 maze[i] 가 속해있는 집합의 수직경로가 생성되지 않았다. 만약 처음부터 인접한 두 방 사이의 집합이 다르다면 무조건 maze[0] 에는 수직경로가 생성되어야 한다. Under[i]=maze[i]으로 두고 이제 새로운 집합에 대한 flag가 필요하기 때문에 under\_flag를 0으로 재설정한다. 이후 i>0일 때 역시 i==0일 때와 비슷하게 실행된다. 인접한 왼쪽 방과 오른쪽 방의 집합이 같다면rand()%2, 50 프로 확률로 수직경로를 생성하여 Under\_flag를 1로 두어 이 집합에 수직경로가 생성되었음을 알리고 under[i]도 maze[i]로 둔다. 그렇지 않다면 set\_number에 1을 더해 이를 under[i]의 새로운 집합으로 쓰이도록 한다. 만약 인접합 왼쪽방과 오른쪽 방의 집합이 다르다면 under\_flag 를 이용하여 이전 집합에 수직경로가 생겼는지 없는지 알 수 있다. 만약 under\_flag가 0이라면 수직경로가 없다는 의미로 이전 집합의 가장 마지막 방, 즉 under[i-1] 에 maze[i-1] 값을 넣어 같은 집합으로 만들어서 그 집합의 마지막 방에 수직경로를 생성한다. 이후 다음 집합의 수직경로 판별을 계속해서 해 나가면 된다. 마찬가지로 rand() 를 이용하여 50프로 확률로 해당 집합의 수직경로를 만들거나(under[i]=maze[i) set\_number에 1을 추가하여 under[i]에 set\_number를 넣어주어 새로운 집합을 형성한다. 이때 i=0과 나머지의 가장 큰 차이는 maze[i-1]==maze[i] 즉 현재 기준으로 왼쪽방과 비교하면서 under 벽 생성여부를 결정하였다. 이렇게 구분한 이유는 만약 i가 0이라면 maze[0-1] 즉 maze[-1]이 되기 때문에 index 오류가 난다.

텍스트이(가) 표시된 사진

자동 생성된 설명

여기서 주의해야 할 점은 가장 마지막 줄은 수직경로가 아닌 밑면의 테두리만 만들어야 하기 때문에 행 기준 HEIGHT-1 개의 under 벽을 만든다. 이 말은 가장 마지막 행의 방들에 수직경로 역시 계산이 되어 있지만 실제로는 이미 마지막 행이기 때문에 큰 필요는 없다는 말이다. 미로의 테두리는 모두 꽉 채워져 있어야 한다.

나머지의 경우 j는 HEIGHT-1보다 작기 때문에 if(j<HEIGHT-1) 문에 모두 충족한다. For문을 WIDTH 만큼 돌면서 under[i]==maze[i] 가 같다면 즉 위의 방과 아래의 방 집합이 같다면 수직경로를 만들어 “+ “ 로 출력하고 다르다면 “+-“를 출력하여 위의 방과 아래 방의 집합이 달라 수직경로가 없음을 알린다. 이후 maze[i]에 under[i]를 다시 저장하여 다음 행의 maze 배열을 만들어준다. 이 과정을 통해서 우리는 2차원 배열이 아닌 1차원 배열로도 eller’s 알고리즘을 구현할 수 있다. 새로 생성된 maze 배열을 통해 다음 행에서 다시 side 벽을 뚫을지 말지 결정할 수 있도록 한다. 이후 가장 마지막에는 “+\n” 를 출력하여 테두리를 만들어 주고 엔터 를 쳐 다음 행이 계산되고 출력될 수 있도록 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

여기서는 가장 밑면의 테두리를 “+-’로 그려주고 가장 오른쪽 끝 역시 “+” 로 미로의 끝을 그려서 마무리 한다. 이때 동적 할당 받은 배열인 maze, under, side배열을 다시 메모리 free 시켜준다. 1차원 배열들이기 때문에 단순히 free 시켜주면 된다.

최종적으로 요약하자면 한 행 씩 양 옆 방을 비교하여 같은 집합이 아니라면 random하게 side 벽을 생성해주고 이후 수직 경로인 under 벽은 같은 집합에는 최소 1개의 수직 경로를 무조건 만들어 주어야 한다. 남은 수직 경로는 random 하게 뚫어준다. 그리고 가장 마지막 행은 인접하는 방이 서로 같은 집합이 아니라면 그냥 side 벽을 다 뚫어준다. 이 과정을 하나라도 빠트리면 더 이상 eller’s 알고리즘이 아니며 이는 어떤 경우에서는 완전 미로를 생성하지 못한다. (우리는 모든 경우의 수에 대해 완전미로를 생성해야 한다.) 특히 cycle이 나와 한 목적지에 대해 2개 이상의 경로가 나오는 것을 확인할 수 있었다. 이를 해결하기 위해서는 위의 사진 속의 코드들처럼 이런 조건들을 모두 만족하게 코드를 짜야 한다.

해당 코드는 한 줄 씩 출력하는데 사용되는 반복문 O(WIDTH)가 있고 하나의 행씩 순차적으로 한 열 씩 확인하여 한 번에 미로의 벽 생성여부를 결정하면서 이를 바로 출력해주는 반복문이 있다. 이때 주의해야 할 점이 side 벽 생성 여부에서 벽을 뚫을 때 인접한 배열의 집합 중 크기가 더 작은 집합으로 바꿔줘야 한다. 이 경우 다시 for문을 통해 maze 배열에서 WIDTH 만큼 해당 집합과 같은 것이 있는지 탐색하는 과정이 필요하다. 즉 시간 복잡도는 O(HEIGHT\*WIDTH\*WIDTH) 가 된다. 그리고 공간 복잡도는 수직경로 생성 여부를 알리는 under\_flag, 새로운 집합을 부여하는 변수인 set\_number 나 반복문에 필요한 변수들이 있고 O(1), maze, side, under 1차원 배열은 동적 할당을 받아야 하기 때문에 최종적인 공간 복잡도는 O(WIDTH) 이다.

1-2.

사실 실험 전 이 알고리즘 구현 방법에 대해 그림을 그려가며 구체화시키고 예비보고서를 작성 했었다. 그래서 대부분 내 생각과 일치하였다. 특히 어떻게 하면 메모리를 작게 사용할 수 있을까 에 대해 고민했었다. 이는 작은 크기의 미로를 생성할 때는 큰 문제가 없겠지만 결국 2차원 배열로 미로를 만드는 것과 1차원 배열로 미로를 만드는 것은 크기가 큰 미로를 만들 때 공간적, 시간적 차이가 날 것으로 예상하고 공간, 시간 복잡도를 줄이기 위해 노력했다. 또한 under 배열에 bool type 처럼 0,1 을 저장하여 그냥 수직경로가 있는지 없는지 정도만 판단하거나 아니면 그 위의 방 집합인 maze 배열 index 의 집합을 부여 할지에 대해 둘 다 예비보고서에 작성했었다. 결과적으로 더 쉽게 구현할 수 있는 것은 0,1 이 아닌 실제 집합을 비교하여 수직경로를 만들 때는 수직경로가 생성되면 under[i]와 maze[i] 같도록 하고, 아니면 under[i]에 새로운 집합을 부여하는 방법으로 알고리즘 및 자료구조를 구현하였다.

하지만 시간 복잡도 측면에서 틀린 점이 있었다. 당연히 한 행 씩 반복문을 돌면서 해당 행의 열들을 비교해가며 미로를 출력하기 때문에 O(WIDTH\*HEIGHT) 일 것을 예상했지만 side 벽을 만들 때마다 인접한 벽 말고도 만약 같은 집합이 있다면 그 해당 집합들을 더 작은 집합들로 모두 동일하게 수정해줘야 한다는 점이다. 즉 내부에 이를 탐색하는 반복문이 더 필요하였다. 그래도 미로를 생성하는데 일반적인 2차원 배열을 사용하지 않고 1차원 배열만을 이용하여 Eller’s 알고리즘을 구현했다는 점에서 공간 복잡도 측면에서는 효율적인 것 같고 실험 전 고민을 많이 한 점이 실제 코드 구현에 도움이 많이 되었다.