11주차 결과보고서

전공 : 경영학과 학년 : 4학년 학번 : 20190808 이름 : 방지혁

1. 실험시간에 작성한 프로그램의 알고리즘과 자료구조를 요약하여 기술하시오. 완성한 알고리즘의 시간 및 공간 복잡도를 보이고 실험 전에 생각한 방법과 어떻게 다른지 아울러 기술하시오

본 실험에서는 eller's algorithm을 사용했다. 알고리즘 동작 과정에 대해 설명하겠다.

알고리즘 설명

첫 단계로는 미로의 첫 번째 행을 초기화한다. 만약 입력으로 들어온 미로의 너비가 N이라면 N개의 서로 다른 집합을 만드는 것이다.

두 번째 단계로는 가로 방향으로 연결한다. 맨 좌측의 cell부터 인접한 두 cell이 서로 다른 집합에 속하면, rand함수를 돌려 50%의 확률로 벽을 제거하여 두 집합의 cell을 합치거나 벽을 유지한다. 그러나 같은 집합이라면 벽을 무조건 유지해서 순환경로를 방지하여 완전미로가 될 수 있도록 한다.

세 번째 단계로는 수직 방향에 대한 처리를 한다. 각 행에 존재하는 각 집합에 대하여 해당 집합에 속해 있는 최소 하나 이상의 cell이 아래 행으로 연결될 수 있도록 해야한다. 이를 위해 처음에 만나는 cell은 무조건 아래로 연결되고 그 이후의 cell들은 두 번째 단계처럼 Rand 함수를 돌려 연결 여부가 정해지도록 한다. 이를 통해 고립되는 집합이 없도록 보장하는 것이다.

네번째 단계로는 세 번째 단계에서 아래 행의 연결되지 않은 cell들에게 새로운 집합에 속하도록 번호를 부여한다. 이렇게 계속 반복한다

마지막 행에서는 인접한 cell들 중에서 서로 다른 집합에 속한다면 무조건 연결한다.

자료구조 설명

1) 집합 정보 배열

Int *sets = (int *)malloc(N * sizeof(int));

현재 행의 셀이 속한 집합 번호를 저장하는 자료구조이다. 크기는 N으로 각 행마다 재사용될 수

있도록 하였다.

2) 세로벽 배열

```
int **right_walls = (int **)malloc(M * sizeof(int*));
for (int i = 0; i < M; i++) {
        right_walls[i] = (int*)malloc(N * sizeof(int));
}
```

한 행의 각 cell에 대하여 오른쪽 벽 존재 유무를 저장하는 2차원 배열이다. 크기는 M * N으로 Walls[row][col]은 (row, col)과 (row, col + 1)에 위치한 cell 들 사이의 세로벽을 나타낸다.

3) 가로벽 배열

```
int **up_down_walls = (int **)malloc(M * sizeof(int*));
for (int i = 0; i < M; i++) {
        up_down_walls[i] = (int*)malloc(N * sizeof(int));
}
```

각 셀에 대해 아래쪽 벽이 있는지 여부를 저장하는 2차원 배열이다. 크기는 마찬가지로 M*N이며 up_down_walls[row][col]은 (row, col)과 (row + 1, col)에 위치한 cell 들 사이의 가로벽을 나타낸다.

4) 집합 연결 여부 체크 배열

```
int *set_connected = (int *)malloc(set_num * sizeof(int));
```

각 집합에 대해 다음 행으로 최소 1개이상의 셀이 연결되었는지 확인하기 위한 1차원 배열이다. 크기는 set_num으로 지금까지 생성한 최대 집합 번호값이다. 완전 미로를 만들기 위해서 필수적이다.

다음 장에 이어서

시간 복잡도 분석

1) 각 행에서 세로벽 연결 및 집합 처리

```
for (int col = 0; col < N - 1; col++) {
        // 집합 번호가 다르다면
        if (sets[col] != sets[col + 1]) {
          // 벽을 제거할지 말지 임의로 선택
          right_walls[row][col] = rand() % 2;
          if (right_walls[row][col] == 0) {
            // 어 벽이 제거되었네? 그럼 두 집합을 합쳐야지
            int old_set = sets[col + 1];
            int new_set = sets[col];
            for (int k = 0; k < N; k++) {
              if (sets[k] == old_set) {
                sets[k] = new_set;
              }
            }
          }
        }
        // 집합 번호가 같다면
        else {
          right_walls[row][col] = 1;   // 같은 집합이면 벽 세우기
        }
      }
```

첨부한 루프의 외부 루프가 O(N)이고, 내부루프도 O(N)이다 그러기에 O(N^2)이라고 볼 수 있다.

2) 가로 벽 및 연결 처리

```
if (row < M - 1) {
        int* set_connected = (int*)malloc(set_num * sizeof(int));
        for (int i = 0; i < set_num; i++) set_connected[i] = 0;
        for (int col = 0; col < N; col++) {
          // 아래로 연결할지 말지 임의로 선택
          int which_set = sets[col];
          if (set_connected[which_set] == 0) {
            // 아직 이 집합은 아래로 연결된 적이 없음
            up_down_walls[row][col] = 0;   // 무조건 아래로 연결
            set_connected[which_set] = 1; // 이 집합은 아래로 연결됨
          }
          else {
            up_down_walls[row][col] = rand() % 2;
          }
        }

        for (int col = 0; col < N; col++) {
          if (up_down_walls[row][col] == 1) {
```

```
            // 아래로 연결하지 않는다면 새로운 집합 번호 부여
            sets[col] = set_num++;
          }
        }
        free(set_connected);
    }
```

For loop 2개 모두 O(N)이기에 최종적으로 O(N)이라고 볼 수 있다.

전체 시간 복잡도는 M(가장 외부 loop)*(세로 연결 + 가로 연결)이기에 O(M*N^2)이라고 볼 수 있다.

공간 복잡도 분석

앞서 설명한 자료구조들에 대해 이야기하자면 sets 배열은 O(N), right_walls 및 up_down_walls 배열은 O(MN), set_connected 배열은 O(Set_num)이기에 다 더하면 O(MN)의 공간복잡도를 가진다는 결론을 최종적으로 도출 할 수 있다.

예비보고서와의 차이점

사실 인생이 급한 막학기생으로서 미리 코드를 작성하고 예비보고서를 작성했기에 그다지 차이점이 없습니다. 대신 kruskal과 비교해보겠습니다. Kruskal의 경우 우리가 자료구조 시간에 배운 최소신장 트리(MST) 알고리즘을 사용합니다. 모든 가능한 벽을 리스트로 만들어 랜덤하게 섞습니다. 벽을 하나씩 검사하여, 다른 집합이면 벽을 제거하여 병합하고 하나의 집합이 될 때까지 반복하는 것입니다. 이를 위해 Union-find 연산이 필요합니다. 벽섞기의 경우 O(MNlogMN)의 시간복잡도가 예상되며 union find 연산의 경우 O(MN)일 것입니다. 최종적으로 O(MN logMN)이 나올 것입니다. 공간복잡도의 경우 벽 리스트의 크기가 O(MN), Union-find 연산을 위한 배열도 이와 같기에 O(MN)일 것입니다. 이와 달리 우리가 실험에서 구현한 알고리즘은 한 행씩 미로를 구현하는 것입니다. 앞서 언급했듯 전체 시간 복잡도는 O(M*N^2), 공간복잡도는 O(MN)입니다.