**Maze – 1 결과보고서**

전공: 수학 학년: 4 학번: 20161255 이름: 장원태

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

우선 자료구조는 실험 전에 생각한 바와 같이 이차원 배열을 사용했다. 이 이차원 배열(int\*\* cell)은 각 방이 어느 집합에 속하는지 그 숫자를 저장하는 배열이다. 또한, 방과 방 사이의 벽은 그 cell값을 기준으로 뚫기는 하지만, 오른쪽으로 벽을 뚫을 때는 원래부터 그 값이 같을 때가 아니라 새로 벽을 뚫게 되었을 경우에만 해당되도록 하여 사이클(cycle) 생성을 방지하였다. 이에 따라 각 줄에 대해 가로로 각 벽을 뚫을 것인지 말 것인지를 저장하는 일차원 배열인 break\_wall을 선언했다. 각 배열은 GetInfo 함수(width와 height을 입력받는 함수)에서 입력받은 width와 height 값에 따라 동적으로 할당되게 된다. 두 배열 모두 초기값은 0으로 설정해주었다.

|  |
| --- |
| cell = (int\*\*)malloc(sizeof(int\*) \*height);  for (i = 0; i < height; i++) {  cell[i] = (int\*)malloc(sizeof(int) \* width);  for (j = 0; j < width; j++)  cell[i][j] = 0;  }  break\_wall = (int\*)malloc(sizeof(int) \* (width - 1));  for (i = 0; i < width - 1; i++) {  break\_wall[i] = 0;  } |

알고리즘의 경우는 수업 자료에 나온 Eller’s algorithm을 따랐다. 각 집합을 다르게 선언하기 위해, 각 방에 새로 num값을 배치하는 동시에 1을 더해주었다. 우선, 첫째 줄은 모든 방을 서로 다른 집합에 속하도록 설정했다.

|  |
| --- |
| // first line  for (i = 0; i < width; i++) {  cell[0][i] = num++;  } |

이제 이 첫 번째 줄을 포함한 모든 줄에 대해, 오른쪽으로 벽을 뚫고 아랫쪽으로 벽을 뚫는 작업을 수행한다. 우선, 오른쪽으로 벽을 뚫는 과정은 아래와 같다.

|  |
| --- |
| // loop for every lines  for (i = 0; i < height; i++) {  for (j = 0; j < width; j++) { // going to right  if (j < width - 1) {  if (cell[i][j] != cell[i][j + 1]) {  if (rand() % 2) {  previous\_value = max(cell[i][j], cell[i][j+1]);  cell[i][j] = min(cell[i][j], cell[i][j + 1]);  cell[i][j + 1] = cell[i][j];  break\_wall[j] = 1;  for (m=0; m<=i; m++){  for (n=0; n<width; n++){  if (cell[m][n] == previous\_value) cell[m][n] = cell[i][j];  }  }  }  else if (!cell[i][j]) cell[i][j] = num++;  }  }  else if (!cell[i][j]) cell[i][j] = num++;  } |

(오른쪽으로 뚫는 과정과 아랫쪽으로 뚫는 과정을 나누어 서술하므로, 아직 가장 바깥 for문은 닫히지 않은 상태이다)

각 방별로, 그 방과 다음 방을 뚫을지 말지를 결정한다. 해당 방과 다음 방이 서로 다른 cell 값을 가지고 있다면 뚫을지 말지를 결정할 수 있다. 이는 난수를 2로 나눈 나머지가 1이 되면(즉, 난수가 홀수이면) 벽을 뚫어주도록 설정함을 통해, 정확히 50%의 확률로 뚫어주도록 설정했다. 이 때, 벽을 뚫어주게 된다면, 두 cell 값 중 작은 값을 기준으로 두 cell 값을 통일시켜주며, break\_wall 값은 1로 설정해주어 “벽을 뚫는다”는 신호를 준다. 이 때, 해당 칸보다 앞서 탐색했던 모든 칸들의 값 중, 현재 상황에서 사라진(즉, 두 cell 값 중 더 큰) 값이 있을 수 있으며, 그 값들을 모두 현재 값으로 바꾸어줘야 한다. 이는 미로의 cycle 생성을 방지하기 위함이다. 그러므로, 별도의 for문을 수행하여 그러한 값들을 현재 cell 값으로 바꿔준다. 이 과정에서, 첫째 줄을 제외한 나머지 줄에서 벽을 뚫어주지 않은 상황이거나, 다음 칸이 존재하지 않는 마지막 칸이 초기화되어 있지 않은 상황이라면, 그 cell 값들을 별도로 초기화해준다.

그런데 만약 해당 방과 다음 방이 서로 같은 cell 값을 이미 가지고 있다면, 어딘가에서 이미 벽이 뚫려 그 두 방은 어떻게든 경로로 이어지게 된다. 그 상황에서 벽을 뚫게 되면 미로에 cycle이 생겨버리므로, 이들은 같은 cell 값을 가지게 되더라도 break\_wall 값을 1로 설정하지 않고 0으로 둔다.

다음으로 아래로 뚫는 과정 및 마무리 과정은 다음과 같다.

|  |
| --- |
| if (i < height - 1) {// going down  j = 0;  while (j < width) {  do {  if (rand() % 2) {  cell[i + 1][j] = cell[i][j];  vertical\_flag = 1;  }  else cell[i + 1][j] = num++;  j++;  } while (cell[i][j-1] == cell[i][j]);  if (!vertical\_flag) {  do {  j--;  } while (j > 0 && cell[i][j] == cell[i][j - 1]);  }  else vertical\_flag = 0;  }  }  else { // for the final line  for (j = 0; j < width - 1; j++) {  if (cell[i][j] != cell[i][j + 1]) {  previous\_value = max(cell[i][j], cell[i][j+1]);  cell[i][j] = min(cell[i][j], cell[i][j + 1]);  cell[i][j + 1] = cell[i][j];  break\_wall[j] = 1;  for (m=0; m<=i; m++){  if (cell[m][n] == previous\_value) cell[m][n] == cell[i][j];  }  }  }  }    print\_maze\_line(i);  for (j = 0; j < width - 1; j++)  break\_wall[j] = 0;  } |

우선 해당 줄의 모든 방을 가로로 돌며, 50% 확률로 아래를 뚫어주도록 한다. 이 때, 각 집합마다 적어도 한 방은 아래로 뚫려 있어야 한다. 그래서, 아래로 뚫었다는 것을 의미하는 vertical\_flag를 설정하여, 아래로 뚫은 방이 적어도 하나 존재하는지를 판단하게 한다. 아래로 뚫게 된다면, 그 아랫방의 cell 값을 현재 방의 cell 값과 동일하게 해주고, vertical\_flag를 1로 set해준다. 뚫지 않았다면, 아랫방을 num값으로 초기화해준다. 같은 집합에 속하는 방을 쭉 돌다가, 다음 방이 다른 집합에 속하는 방까지 오면, 거기까지 아래로 뚫린 방이 있는지 vertical\_flag를 통해 확인한다. 만약 vertical\_flag가 set되어있지 않다면, 아래로 뚫린 방이 하나도 없으므로 다시 과정을 수행해야 한다. 그러므로, 이전 방이 다른 집합에 속하는 방을 만나기 전까지 방을 왼쪽으로 쭉 이동한 후 같은 과정을 반복해준다.

만약 마지막 줄에 도착하게 되고, 그 줄에 대해 오른쪽으로 뚫는 과정까지 모두 마쳤다면, 여기서는 아래로 더 뚫을 수가 없다. 대신, 마지막 줄에서는 서로 다른 집합에 속한 방들 사이의 모든 벽을 제거하여 모든 방이 같은 집합에 속하도록 한다. 이 때도 역시, 현재 방과 다음 방이 이미 같은 집합에 속해있다면 벽을 다시 뚫지 않도록 주의한다. 만약 현재 방과 다음 방의 cell 값이 다르다면, 이번에는 무조건 벽을 뚫어주도록 한다. 이 경우도 역시 이전까지 지나온 방들 중 지금 바뀌어 사라진 값을 가지고 있는 방들에 대해서는 그 값을 바꿔주는 과정을 수행하여 cycle 생성을 방지한다.

각 줄마다의 과정을 마치면, 그 줄에 대해 먼저 미로를 출력한다. 이는 각 줄에 대해서 오른쪽으로 뚫은 벽, 아랫쪽으로 뚫은 벽 순서대로 출력을 하게 된다. 그 후, break\_wall 배열을 모두 0으로 다시 초기화해준 뒤 loop를 돌게 된다. 모든 loop가 종료되면, 동적 할당했던 cell과 break\_wall을 모두 free해주고, file을 닫고 프로그램을 종료한다.

각 줄에 대한 미로 출력 함수는 다음과 같다.

|  |
| --- |
| void print\_maze\_line(int i) {  int j;  if (i == 0) { // the very top  for (j = 0; j < width; j++) {  printf("+-");  fprintf(file, "+-");  }  printf("+\n");  fprintf(file, "+\n");  }  for (j = 0; j < width; j++) { // the room and the next room  if (j == 0){ printf("|"); fprintf(file, "|");}  else if (!break\_wall[j-1]) {printf("|"); fprintf(file, "|");}  else {printf(" "); fprintf(file, " ");}  printf(" "); fprintf(file, " ");  }  printf("|\n"); fprintf(file, "|\n");  if (i == height - 1) {  for (j = 0; j < width; j++) {  printf("+-"); fprintf(file, "+-");  }  printf("+\n"); fprintf(file, "+\n");  }  else { // the room and the bottom room  for (j = 0; j < width; j++) {  printf("+"); fprintf(file, "+");  if (cell[i][j] != cell[i + 1][j]){ printf("-"); fprintf(file, "-");}  else {printf(" "); fprintf(file, " ");}  }  printf("+\n"); fprintf(file, "+\n");  }  } |

모든 출력은 콘솔 상으로도 그 출력을 확인할 수 있도록 printf와 fprintf를 모두 사용했다. 우선, 첫 번째 줄에 대한 출력일 경우, 맨 위는 width만큼 가로벽이 있으므로 별도로 출력해준다. 그리고, 각 줄에 대해 다음 방과 벽이 있는지 없는지를 break\_wall 배열 값을 통하여 결정하며, 벽이 있으면 “|”를, 없으면 “ “를 출력한다. 비슷한 맥락으로 아랫방과 현재 방 사이도 뚫려 있는지를 파악하되, 이 경우는 함수 호출 전에 아랫방을 뚫으며 cell값을 같은 값으로 초기화해 주었으므로 cell값을 비교하여 같으면 “ “를, 다르면 “-“를 출력하도록 한다. 만약 마지막 줄이면 아랫방이 없으므로, 이 때는 맨 첫 줄의 가로줄을 출력하듯이 맨 끝 줄의 가로줄을 출력해주도록 한다.

이 과정의 경우, cell의 모든 방에 대하여 탐색을 하고, 그 안에서 또 앞선 방에 대해 탐색을 하므로, (단, 이 경우 행은 1부터 width까지 다양함) 시간 복잡도는 O(WIDTH\*WIDTH\*HEIGHT)이 된다. 공간 복잡도의 경우는 이차원 배열 하나와 일차원 배열 하나를 사용하므로, O(WIDTH\*HEIGHT)이 된다.