13주차 예비보고서

전공: 경영학과 학년: 3학년 학번: 20190963 이름: 한다현

**1.**

DFS

DFS 알고리즘은 while문을 사용해 stack이 비어 있는지 확인하고, stack이 비어 있지 않은 경우에 계속 반복한다. while 문 내에서는 먼저 stack의 top이 목적지와 동일한지 검사한다. 동일하다면 while 반복문을 종료하고, 동일하지 않다면 stack의 top에 해당하는 vector의 사이즈가 0이 아닌지 검사한다. vector의 사이즈가 0이 아니면, 즉, 탐색할 노드가 남아있다면 for loop을 실행한다. 이 for loop에서는 i가 0부터 stack의 top에 해당하는 vector의 사이즈 전까지 i를 1씩 증가시키며 반복한다. 이 반복문 내에서 아직 방문하지 않은 노드가 있는지 검사하고, 방문하지 않은 노드를 stack에 추가시키는 방식으로 반복문을 실행한다. 따라서 DFS 알고리즘은 stack이 비어 있지 않은 동안 while문을 반복하고, vector의 사이즈만큼 for 문을 반복하는 이중 반복문이기 때문에 이 알고리즘의 시간 복잡도는 O(n^2)이다.

BFS

BFS 알고리즘은 queue를 사용하는데, while 문을 사용해 queue가 비어 있는지 확인하고, 비어 있지 않다면 반복문 내의 코드를 반복한다. while 문 내에서는 queue의 front가 target과 일치하는지 확인하고, 일치하면 while문을 종료하고, 일치하지 않으면 다음 코드를 실행한다. 현재 노드를 pop한 후에 이 노드와 인접한 노드를 찾는다. 이 때 k가 0부터 현재 노드에 해당하는 vector의 사이즈 전까지 k를 1씩 증가시키며 for 문을 실행한다. 이 과정을 통해 인접한 노드를 찾은 후 그 노드를 방문한 적이 있는지 확인하고, 그렇지 않다면 queue에 추가한다. 따라서 BFS 알고리즘은 queue가 비어 있지 않은 동안 while문을 반복하고, vector의 사이즈만큼 for 문을 반복하는 이중 반복문이기 때문에 이 알고리즘의 시간 복잡도 역시 O(n^2)이다.

**2.**

DFS

DFS 알고리즘에서는 stack을 사용하고, visited 배열과 방문한 모든 경로를 저장하는 path 배열, 최적 경로를 저장하는 best 배열을 사용한다. 시작점에 해당하는 노드 v를 stack에 추가하고, v의 visited를 1로 변경한다. stack이 비어 있지 않을 때까지 반복하는 while 반복문을 실행하고, stack의 top이 목적지와 일치하면 반복문을 종료한다. stack의 top이 목적지가 아니면, stack의 top에 해당하는 graph vector의 사이즈가 0인지 확인한다. 0이 아니면 for loop을 통해 stack의 top에 해당하는 노드 u와 인접한 노드를 검사한다. 이 노드들 중에서 방문한 적이 없는 노드를 stack에 추가하고, 이 노드를 path에 추가한다. 이런식으로 반복을 하게 되면 방문한 경로를 path 배열에 저장할 수 있다. stack의 top이 목적지와 일치할 때 while문을 종료하고, stack을 top부터 empty까지 반복하며 stack에 저장된 노드를 반환한다. 이 때 반환한 노드를 best 배열에 저장하면 DFS 알고리즘을 통한 최적 경로를 찾을 수 있다. DFS 알고리즘을 iterative한 방법으로 구현하기 위한 방법은 위의 설명한 방법이다. 먼저 stack의 top에 해당하는 graph vector의 사이즈가 0이 아니라면, for loop을 사용해 해당 노드와 인접한 노드를 모두 찾을 수 있고, 그 중 방문한 적이 없는 노드를 stack에 추가한다. 이렇게 추가된 노드는 stack의 top이 되고, 다시 해당 노드의 graph vector 사이즈를 검사하는 방식으로 반복하면 재귀 방식이 아닌 iterative 방식으로 DFS 알고리즘을 구현할 수 있다.

BFS

BFS 알고리즘은 DFS 알고리즘과 달리 queue를 사용한다. 먼저 queue를 초기화하고, 시작 노드를 queue에 추가한다. 이 후 while 문을 사용해 queue가 비어 있지 않은 동안 반복한다. while 문 내에서는 먼저 queue의 back이 목적지와 동일한지 검사하고, 동일하다면 반복문을 종료하고, 동일하지 않다면 queue의 front를 반환한다. 이렇게 반환한 노드와 인접한 노드를 모두 검사하는데, 이 때 방문한 적이 없는 인접한 노드의 visited 값을 1로 수정한다. 그리고 그 노드들을 queue에 추가하는 과정을 반복한다. 이 과정에서 현재 노드를 w, 이와 인접한 노드를 u라고 할 때, u의 parent를 w라고 저장해야 한다. 목적지에 도착했을 때, 그 노드로부터 노드의 parent를 따라가는 식으로 최적 경로를 도출할 수 있기 때문이다.