DFS & BFS 완전 정복 노트
(문제 풀이 사고력 중심 정리)

KUCSEPotato

January 28, 2026

Contents

1 이 문서의 목적

이 문서는 DFS(Depth-First Search)와 BFS(Breadth-First Search)를

- 공식처럼 외우지 않고
- 코드만 암기하지 않고
- "왜 이 알고리즘을 써야 하는지"를 스스로 판단할 수 있도록

사고 구조 중심으로 정리한 학습 노트이다.

이 문서를 통해 다음 질문에 명확히 답할 수 있어야 한다.

- 이 문제는 왜 DFS 문제인가?
- 이 문제는 왜 BFS 문제인가?
- visited는 왜 필요한가?
- DFS와 백트래킹은 왜 같은 구조인가?
- BFS는 왜 항상 최단거리를 보장하는가?

2 탐색(Search)이란 무엇인가

DFS와 BFS는 "그래프 알고리즘" 이전에, **탐색(Search)**이라는 개념에서 출발한다.

2.1 탐색의 본질

탐색이란 다음 질문에 답하는 과정이다.

> **가능한 모든 상황 중에서, 우리가 원하는 조건을 만족하는 상황이 존재하는가?**

이를 컴퓨터적으로 표현하면 다음과 같다.

- 상태(State): 현재 상황을 표현하는 정보
- 상태 전이(Transition): 한 상태에서 다른 상태로 이동하는 규칙

2.2 모든 탐색 문제는 그래프로 표현 가능하다

우리가 푸는 대부분의 알고리즘 문제는 형태만 다를 뿐 모두 그래프로 볼 수 있다.

문제 유형	그래프 해석
미로 탐색	정점: 칸, 간선: 상하좌우 이동
순열/조합	정점: 현재까지 선택한 수열
문자열 변환	정점: 현재 문자열
숫자 만들기	정점: 현재 값

즉,

> **DFS/BFS는 "상태 공간 그래프를 어떻게 탐색할 것인가"에 대한 전략이다.**

3 그래프 표현: 인접 리스트

그래프를 코드로 표현하는 가장 대표적인 방식은 인접 리스트이다.

3.1 인접 리스트란?

- key: 정점
- value: 해당 정점에서 이동 가능한 정점들의 리스트

```
graph = {
    1: [2, 3],
    2: [1, 4],
    3: [1, 4],
    4: [2, 3]
}
```

의미:

- 1번 정점 → 2, 3으로 이동 가능
- 2번 정점 → 1, 4로 이동 가능

DFS/BFS는 항상 다음 질문을 반복한다.

현재 정점에서 갈 수 있는 다음 정점은 누구인가?

이 질문에 답해주는 구조가 바로 인접 리스트이다.

4 DFS (Depth-First Search)

4.1 DFS의 핵심 사고

DFS는 다음 문장 하나로 요약된다.

한 방향으로 끝까지 가보고, 더 이상 갈 수 없을 때 되돌아온다.

이때 중요한 개념이 두 가지 있다.

- 들어간다 (go deeper)
- 되돌아온다 (backtrack)

4.2 DFS와 재귀의 관계

DFS는 구조적으로 다음과 완전히 같다.

- 함수 호출 → 들어감
- 함수 종료 → 되돌아옴

그래서 DFS는 재귀로 구현하면 가장 자연스러운 형태가 된다.

4.3 DFS 기본 템플릿

```
def dfs(u):
    visited[u] = True

    for v in graph[u]:
        if not visited[v]:
            dfs(v)
```

이 코드의 의미를 한 줄씩 해석하면 다음과 같다.

- "u에 도착했다"
- "u와 연결된 모든 정점을 하나씩 확인한다"
- "아직 안 간 곳이 있으면 그곳으로 들어간다"

4.4 DFS 실행 흐름 예시

그래프:

$$1 \to 2 \to 4, \quad 1 \to 3$$

DFS(1)의 실행 순서:

1. 1 방문
2. 2 방문
3. 4 방문
4. 되돌아옴
5. 다시 1로 돌아와 3 방문

즉 DFS는 "경로 중심 탐색"이다.

4.5 DFS의 특징

장점

- 모든 경우를 빠짐없이 탐색 가능
- 백트래킹과 구조적으로 동일

단점

- 최단거리 보장 불가
- 깊이가 깊으면 재귀 제한 위험

5 BFS (Breadth-First Search)

5.1 BFS의 핵심 사고

BFS는 다음 문장으로 정의된다.

가까운 상태부터 전부 확인하고, 그 다음 거리로 확장한다.

즉 BFS는 "깊이"가 아니라 "거리(레벨)" 기준 탐색이다.

5.2 왜 큐(Queue)를 사용하는가

BFS의 규칙은 단 하나다.

먼저 발견한 정점을 먼저 처리한다.

이는 FIFO 구조와 완전히 동일하므로 큐를 사용한다.

5.3 BFS 기본 템플릿

```
from collections import deque

def bfs(start):
    q = deque([start])
    visited[start] = True

    while q:
        u = q.popleft()
        for v in graph[u]:
            if not visited[v]:
                visited[v] = True
                q.append(v)
```

5.4 visited를 enqueue 시점에 하는 이유

만약 visited 처리를 늦게 하면:

- 같은 정점이 여러 번 큐에 들어간다
- 메모리 폭발

따라서 BFS에서는

큐에 넣는 순간 방문 처리한다.

이것이 매우 중요하다.

5.5 BFS와 최단거리

모든 간선 비용이 동일할 때:

- BFS는 거리 0
- 그 다음 거리 1
- 그 다음 거리 2

순서대로 탐색한다.
따라서 어떤 정점을 **처음 만난 순간**의 거리가 곧 최단거리이다.

```
dist[v] = dist[u] + 1
```

이 한 줄이 BFS의 핵심이다.

6 DFS vs BFS 선택 기준

구분	DFS	BFS
핵심 개념	경로를 끝까지 탐색	거리(레벨) 기반 탐색
주요 구조	재귀 / 스택	큐
최단거리	보장 X	보장 O
대표 문제	백트래킹, 경우의 수	미로, 최소 이동

7 DFS가 백트래킹이 되는 이유

백트래킹의 본질은 다음 문장이다.

선택 → 내려감 → 실패 시 되돌림

이는 DFS 구조와 완전히 동일하다.

```
path.append(x)
dfs()
path.pop()
```

이 세 줄이 바로 백트래킹의 정체이다.

8 문제 분류 기준 노트: DFS인가? BFS인가?

많은 학생들이 DFS/BFS 문제를 어려워하는 이유는 **코드를 몰라서가 아니라, 문제를 어떻게 분류해야 하는지 모르기 때문**이다.
이 섹션의 목표는 다음 질문에 자동으로 답할 수 있도록 만드는 것이다.

- 이 문제는 DFS 문제인가?
- BFS 문제인가?
- 아니면 둘 다 가능한가?

8.1 가장 먼저 해야 할 질문

문제를 읽자마자 반드시 다음 질문을 스스로에게 던져야 한다.

이 문제에서 내가 탐색해야 하는 "상태(state)"는 무엇인가?

- 위치인가? (좌표)
- 현재 숫자인가?
- 지금까지 만든 문자열인가?
- 선택한 원소들의 집합인가?

상태가 정의되는 순간, 이 문제는 이미 탐색 문제이다.

그 다음 반드시 두 번째 질문을 한다.

이 상태에서 다음 상태로 어떻게 이동할 수 있는가?

이 두 질문이 바로 그래프의

- 정점(Vertex)
- 간선(Edge)

을 정의한다.

8.2 DFS 문제의 특징

다음 특징이 하나라도 보이면 DFS를 우선적으로 의심한다.

- 모든 경우를 출력하라
- 가능한 모든 경우의 수를 탐색하라
- 조합 / 순열 / 부분집합
- 경로가 존재하는지만 판단
- 정답이 여러 개일 수 있음

대표 키워드

- "모든 경우"
- "가능한 방법"
- "전부 탐색"

대표 문제 유형

- N과 M 시리즈
- 백트래킹 문제
- 조합/순열 생성
- 연결 요소 개수 세기

핵심 판단 기준

정답이 하나가 아니라 여러 개이며, 전부 확인해야 한다면 DFS이다.

8.3 BFS 문제의 특징

다음 특징이 보이면 거의 확정적으로 BFS이다.

- 최소 이동 횟수
- 최단 거리
- 가장 빠른 시간
- 최소 연산 횟수

대표 키워드

- "최소"
- "가장 빠른"
- "최단"

대표 문제 유형

- 미로 탐색
- 숨바꼭질
- 토마토
- 벽 부수고 이동하기

핵심 판단 기준

거리, 시간, 횟수가 등장하면 DFS는 즉시 버리고 BFS를 선택한다.

8.4 DFS와 BFS가 모두 가능한 경우

일부 문제는 DFS와 BFS 모두로 풀 수 있다.
예를 들어:

- 단순 연결 여부 확인
- 그래프 순회 출력

하지만 이 경우에도 다음 기준을 따른다.

- 깊이가 중요하면 DFS
- 거리(레벨)가 중요하면 BFS

8.5 실전 판단 5단계 체크리스트

문제를 읽으면 아래 순서대로 생각한다.

1. 상태(state)가 무엇인가?
2. 상태 전이는 무엇인가?
3. 방문한 상태를 다시 방문하면 안 되는가? (visited 필요)
4. "최소"라는 단어가 있는가?
5. 모든 경우를 봐야 하는가?

- 4번이 YES → BFS
- 5번이 YES → DFS

8.6 자주 하는 잘못된 선택

실수 1

"DFS가 더 쉬워 보여서 DFS로 풀자"

→ 틀린 사고이다. 문제의 요구사항이 알고리즘을 결정한다.

실수 2

"BFS가 느릴 것 같아서 DFS로 하자"

→ 최단거리 문제에서 DFS는 정답 자체가 보장되지 않는다.

실수 3

"visited는 그냥 외워서 쓴다"

→ visited는 **상태 공간 그래프에 사이클이 존재하기 때문에 필요한 개념**이다.

8.7 최종 요약 공식

모든 경우 → DFS
최단 거리 → BFS

9 마지막 정리

- DFS/BFS는 그래프 이전에 "상태 탐색 사고"이다.
- DFS는 경로 중심, BFS는 거리 중심이다.
- visited는 사이클을 막기 위한 안전장치이다.
- 최단거리 = BFS
- 모든 경우 생성 = DFS

10 Appendix

BOJ DFS와 BFS 문제 풀이입니다. 최대한 정석적인 코드로 작성하였기 때문에 참고하기에 도움이 될 것입니다.

```
from collections import deque

def DFS(start: int, graph: dict) -> list[int]:
    visited = set()
    result = list()

    def dfs(node: int) -> None:
        visited.add(node)
        result.append(node)

        for val in graph[node]:
            if val not in visited:
                dfs(node=val)
    dfs(start)

    return result

def BFS(start: int, graph: dict) -> list[int]:
    visited = set()
    result = list()
    queue = deque([start])

    visited.add(start)

    while queue:
        val = queue.popleft()
        result.append(val)

        for value in graph[val]:
            if value not in visited:
                visited.add(value)
                queue.append(value)

    return result

def main():
    N, M, V = map(int, input().split())
    graph = {i: [] for i in range(1, N+1)} # adjacency list

    for _ in range(M):
        departure, arrival = map(int, input().split())
        graph[departure].append(arrival)
        graph[arrival].append(departure)

    for i in range(1, N+1):
        graph[i].sort() # O(nlogn)

    dfs = DFS(start=V, graph=graph)
    bfs = BFS(start=V, graph=graph)
```

```
    print(*dfs)
    print(*bfs)

if __name__ == "__main__":
    main()
```