

DFS & BFS 정리 노트 (개인 학습용)

KUCSEPotato

January 18, 2026

Contents

1	이 문서의 목표	2
2	탐색(Search)이란 무엇인가	2
2.1	탐색의 관점: 상태 공간(State Space)	2
2.2	그래프(Graph)로 보면 더 단순해진다	2
3	DFS(깊이 우선 탐색)	2
3.1	직관	2
3.2	DFS가 자연스럽게 재귀로 구현되는 이유	3
3.2.1	가장 작은 DFS 예시	3
3.3	DFS 구현 템플릿 (재귀)	3
3.4	DFS 구현 템플릿 (스택)	3
3.5	DFS의 장점과 단점	4
4	BFS(너비 우선 탐색)	4
4.1	직관	4
4.2	BFS가 큐(Queue)로 구현되는 이유	4
4.3	BFS 구현 템플릿	4
4.4	BFS는 왜 최단거리를 보장하는가?	5
5	DFS vs BFS 비교	5
6	visited가 왜 필요한가	5
7	DFS가 백트래킹으로 이어지는 이유	5
7.1	백트래킹의 핵심	6
8	자주 터지는 실수 체크리스트	6
9	어떤 문제에서 무엇을 선택할까	6
9.1	DFS를 우선 고려하는 경우	6
9.2	BFS를 우선 고려하는 경우	6
10	연습 문제 추천 (난이도 순서)	7
11	마무리 정리	7

1 이 문서의 목표

이 문서는 DFS(Depth-First Search)와 BFS(Breadth-First Search)를 그냥 공식처럼 외우지 않고, 실제로 문제 풀이에 적용할 수 있도록 개념과 구현 패턴을 정리한 개인 학습 노트이다.

특히 다음 질문에 답하는 것이 목표이다.

- DFS/BFS는 왜 등장했는가?
- DFS/BFS는 무엇을 해결하는가?
- DFS/BFS는 어떤 상황에서 쓰는가?
- DFS/BFS를 구현할 때 항상 헷갈리는 포인트는 무엇인가?

2 탐색(Search)이란 무엇인가

2.1 탐색의 관점: 상태 공간(State Space)

DFS/BFS는 결국 “탐색”이다. 탐색이란 가능한 상태(상황, 경우)를 이동하며 조사하는 과정이다.

- 상태(state): 현재 상황을 표현하는 정보(예: 현재 위치, 현재까지 만든 문자열, 남은 선택지 등)
- 상태 전이(transition): 한 상태에서 다른 상태로 이동하는 규칙(예: 상하좌우 이동, 숫자 하나 선택하기 등)

2.2 그래프(Graph)로 보면 더 단순해진다

DFS/BFS는 보통 그래프 탐색으로 소개된다. 하지만 실제로는 어떤 문제든 “상태”를 정의할 수 있으면 그래프로 볼 수 있다.

- 정점(Vertex): 상태
 - 간선(Edge): 상태 전이
- 예를 들어,
- 미로 문제: 정점=격자 칸, 간선=인접 칸 이동
 - 백트래킹 문제(N과 M): 정점=현재까지 만든 수열, 간선=다음 숫자 선택
 - 문자열 변환: 정점=현재 문자열, 간선=가능한 연산 적용

3 DFS(깊이 우선 탐색)

3.1 직관

DFS는 다음과 같이 생각하면 된다.

한 방향으로 갈 수 있을 때까지 계속 가보고, 더 이상 못 가면 되돌아온다.

즉,

- 현재 상태에서 가능한 다음 상태를 하나 선택한다.
- 그 상태로 “들어간다” (재귀 호출 또는 스택 push).
- 막히면(더 갈 곳이 없으면) “되돌아온다” (재귀 return 또는 스택 pop).

3.2 DFS가 자연스럽게 재귀로 구현되는 이유

DFS는 “들어감”과 “되돌아옴”이 반복된다. 이 구조는 함수 호출 구조(콜 스택)와 거의 동일하기 때문에 재귀로 구현하면 코드가 자연스럽게 나온다.

3.2.1 가장 작은 DFS 예시

예시 문제: 0에서 시작해서 3에 도달하는 모든 방법 한 번에 +1 또는 +2만 할 수 있다고 하자.

```
def dfs(x):
    if x == 3:
        print("Dfs is done!")
        return
    if x > 3:
        return

    dfs(x + 1)
    dfs(x + 2)
```

핵심: DFS는 “상태(x)”를 기준으로 다음 상태를 계속 호출한다.

3.3 DFS 구현 템플릿 (재귀)

그래프 탐색(인접 리스트) 기준의 대표 템플릿:

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

- `visited[u]`: 정점 `u`를 이미 방문했는지 여부
- 그래프가 사이클을 가질 수 있으므로 `visited`는 사실상 필수이다.

3.4 DFS 구현 템플릿 (스택)

재귀를 쓰지 않고 직접 스택으로도 구현 가능하다.

```
def dfs_iter(start):
    stack = [start]
    visited[start] = True

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

3.5 DFS의 장점과 단점

장점

- 구현이 간단(특히 재귀)
- 경로를 “끝까지” 탐색하는 데 적합
- 백트래킹(모든 경우 생성)과 구조적으로 잘 맞음

단점

- 최단거리(최소 이동 횟수)를 보장하지 않음
- 재귀 깊이가 깊어질 수 있음(파이썬은 재귀 제한/스택 오버플로우 주의)

4 BFS(너비 우선 탐색)

4.1 직관

BFS는 다음과 같이 생각하면 된다.

가까운 것부터 전부 보고, 그 다음 레벨(거리)로 넘어간다.

즉, BFS는 “거리 0”, “거리 1”, “거리 2” 처럼 **레벨(층)** 단위로 확장된다.

4.2 BFS가 큐(Queue)로 구현되는 이유

BFS의 본질은 “먼저 발견한 것을 먼저 처리”하는 것이다. 이 구조는 FIFO(First-In First-Out) 큐와 동일하다.

4.3 BFS 구현 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)
```

4.4 BFS는 왜 최단거리를 보장하는가?

가중치가 모두 동일(예: 한 번 이동 비용이 1)한 그래프에서, BFS는 “거리 1짜리 정점들”을 모두 처리한 뒤에 “거리 2짜리 정점들”을 처리한다.

따라서 어떤 정점을 **처음 방문한 순간의 거리**가 그 정점까지의 최단거리이다.

이를 코드로 표현하면 다음과 같이 거리 배열을 둔다.

```
from collections import deque

def bfs_distance(start):
    q = deque([start])
    dist = [-1] * (n + 1)
    dist[start] = 0

    while q:
        u = q.popleft()
        for v in graph[u]:
            if dist[v] == -1:
                dist[v] = dist[u] + 1
                q.append(v)
    return dist
```

5 DFS vs BFS 비교

구분	DFS	BFS
핵심 전략	한 방향으로 깊게 탐색, 막히면 되돌아옴	가까운 것부터 레벨 단위로 확장
구현 자료구조	재귀(콜스택) 또는 스택(Stack)	큐(Queue)
최단거리 보장	일반적으로 보장하지 않음	(가중치 동일 시) 최단거리 보장
주요 사용처	백트래킹, 연결 요소 탐색, 경로 존재 여부	최단거리, 최소 이동 횟수, 레벨 탐색
메모리 사용	상황에 따라 적거나 큼(깊이에 비례)	최악의 경우 큼(한 레벨 폭에 비례)

6 visited가 왜 필요한가

그래프에 사이클이 있으면, visited 없이 탐색하면 무한 루프에 빠질 수 있다.

- DFS: $A \rightarrow B \rightarrow C \rightarrow A \rightarrow \dots$
- BFS: 큐에 같은 정점이 계속 들어감

따라서 “이미 방문한 상태를 다시 방문하지 않는다”는 규칙이 필요하다.

7 DFS가 백트래킹으로 이어지는 이유

많은 “모든 경우 출력” 문제는 DFS의 변형이다.

7.1 백트래킹의 핵심

선택을 하고(상태 변경), 다음으로 내려가며, 필요하면 선택을 되돌린다(상태 복원).

대표적인 코드 구조:

```
path = []
used = [False] * (N + 1)

def backtrack():
    if len(path) == M:
        print(path)
        return

    for x in range(1, N + 1):
        if not used[x]:
            used[x] = True
            path.append(x)

            backtrack()

            path.pop()
            used[x] = False
```

중요: DFS + 상태 복원(pop, used reset)이 백트래킹이다.

8 자주 터지는 실수 체크리스트

- DFS 재귀에서 종료 조건(base case)이 명확한가?
- visited 처리를 “언제” 하는가? (push할 때 / pop할 때)
- BFS에서 방문 처리를 늦게 하면(큐에서 꺼낼 때) 중복 삽입이 발생할 수 있다.
- 2차원 격자 BFS에서 범위 체크($0 \leq nx < R$)를 했는가?
- 백트래킹에서 상태 복원(pop(), visited reset)을 빠뜨리지 않았는가?

9 어떤 문제에서 무엇을 선택할까

9.1 DFS를 우선 고려하는 경우

- 모든 경우를 생성/출력해야 한다 (순열, 조합, 부분집합)
- 연결 요소(connected component) 개수를 세고 싶다
- 경로가 “존재하는지”만 중요하고 최단은 필요 없다

9.2 BFS를 우선 고려하는 경우

- 최단거리/최소 이동 횟수 문제
- 시작점에서 각 노드까지의 거리(레벨)를 구해야 한다
- 미로에서 최소 이동

10 연습 문제 추천 (난이도 순서)

DFS/BFS 개념을 정착시키려면, 다음과 같은 순서가 자연스럽다.

1. 단순 DFS: 트리 순회 (pre/in/post)
2. DFS + visited: 연결 요소 개수
3. BFS: 최단거리(격자 미로)
4. 백트래킹: N과 M (순열/조합)

11 마무리 정리

- DFS/BFS는 “그래프 알고리즘”이기 전에 “상태 탐색 사고”이다.
- DFS는 깊게 들어가고, BFS는 레벨 단위로 확장한다.
- visited는 사이클을 막기 위한 핵심 장치이다.
- 백트래킹은 DFS + 상태 복원이다.