2017 SCSC Workshop

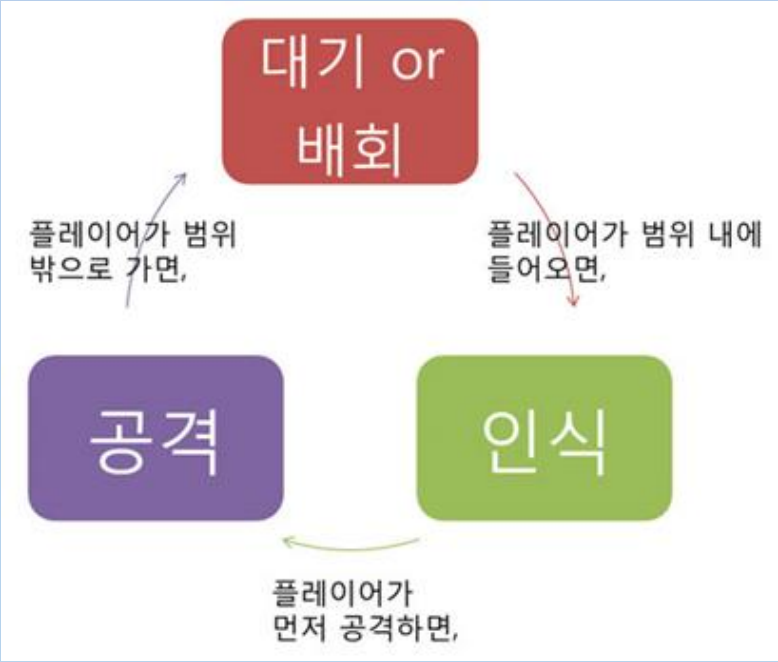
Sogang univ.
System modeling & Optimization Lab
Sangdurk Han

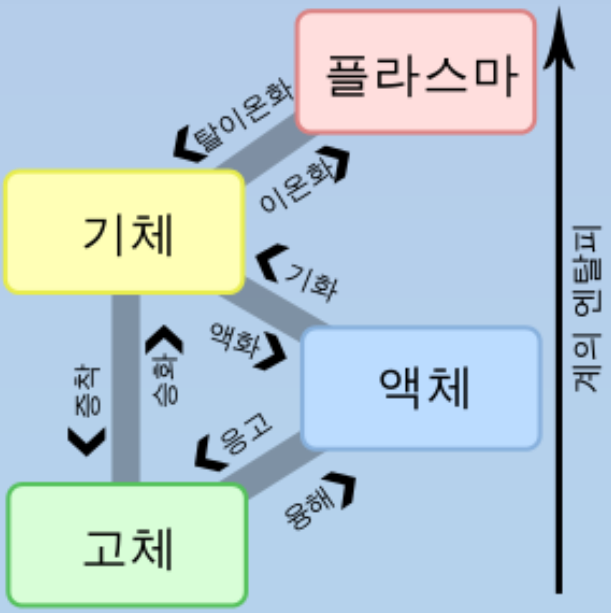
Contents

- Graph
- DFS (Depth – First – Search)
- BFS (Breadth – First – Search)

Search

State


대기 or
배회
플레이어가 범위
밖으로 가면,
플레이어가 범위 내에
들어오면,
공격
인식
플레이어가
먼저 공격하면,


플라스마
기체
액체
고체
탈이온화
이온화
기화
액화
응고
승화
승화
계의 엔탈피

State

- 하나의 경우

Ex) 현재 퍼즐의 모양

비밀번호 (1234)

집에서 학교까지의 거리

Search

- 모든 경우의 수를 탐색 (Backtracking)

Ex) 퍼즐을 최소 횟수로 완성해라.

비밀번호 풀기.

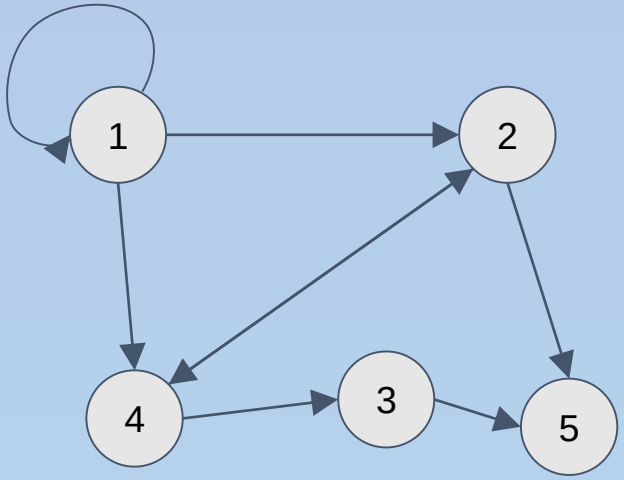
신촌에서 강남역까지 최단 경로는?

Graph

- Graph를 이용하여 표현

경우의 수를 정점(vertex)

그 관계를 간선(edge)



Ex) 정점-역, 간선-선로

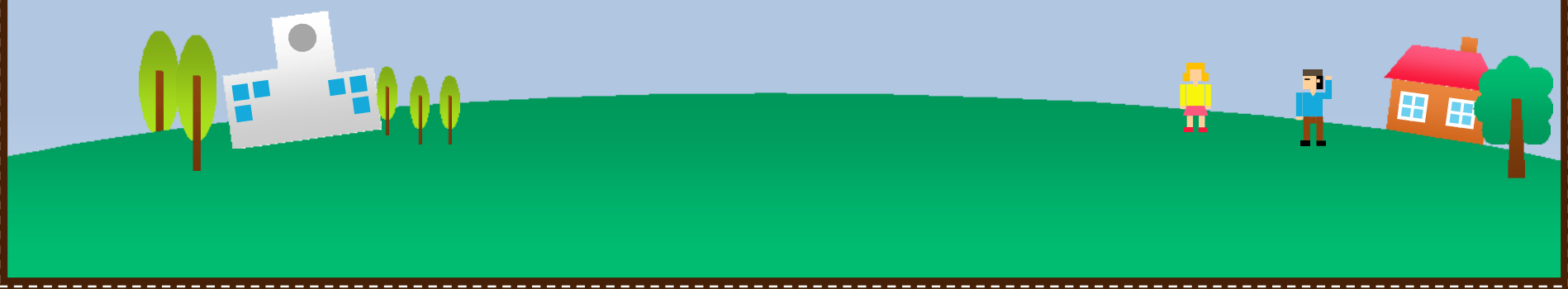
정점-캐릭터 상태, 간선-캐릭터 행동

Graph

상태를 그래프로 나타내보면…

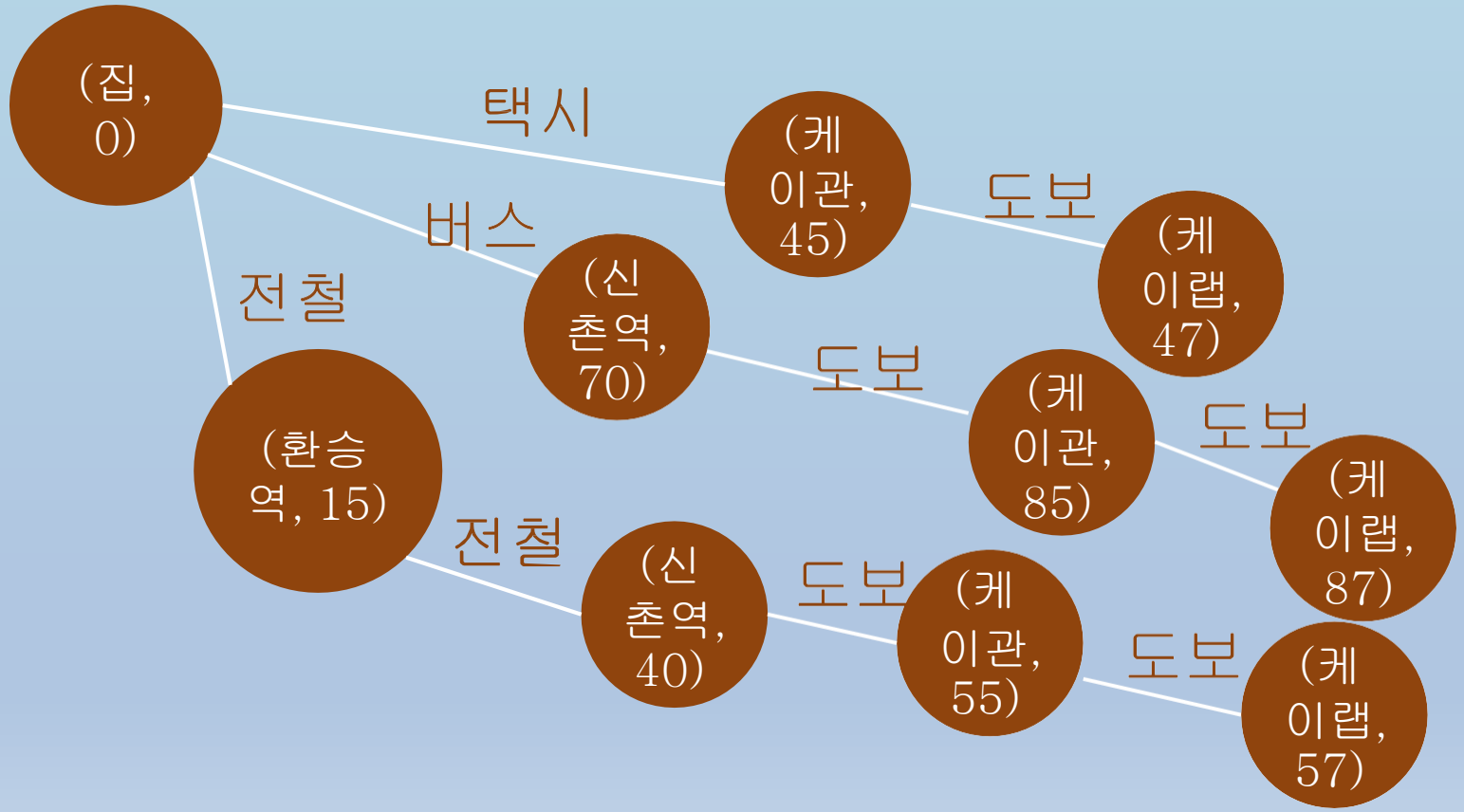
상태를 정점으로,
그 상태들간의 관계를 간선으로!

케이랩까지 가장 빨리 도착하는 상태를 탐색하는 문제
상태를 간단히 (위치, 시간(분)) 으로 나타내봅시다.



Graph

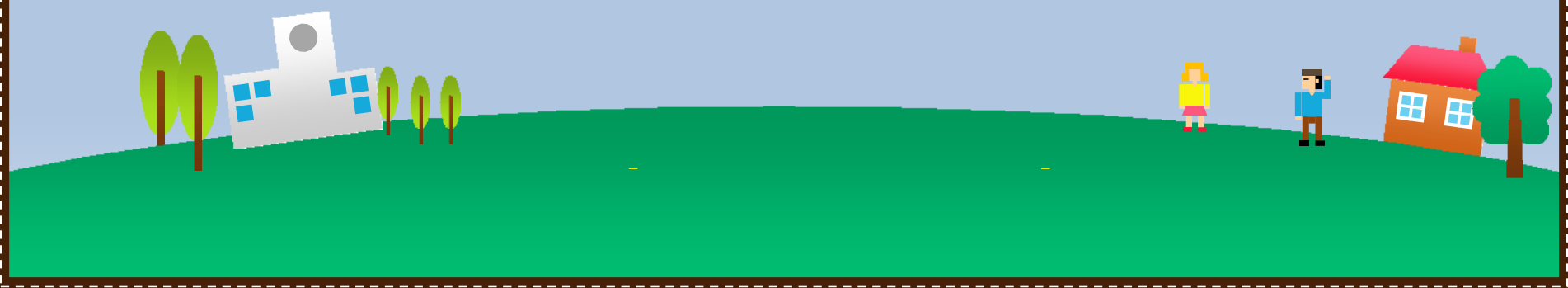
상태를 그래프로 나타내보면…



DFS

Depth – First – Search

깊이 – 우선 – 탐색



DFS

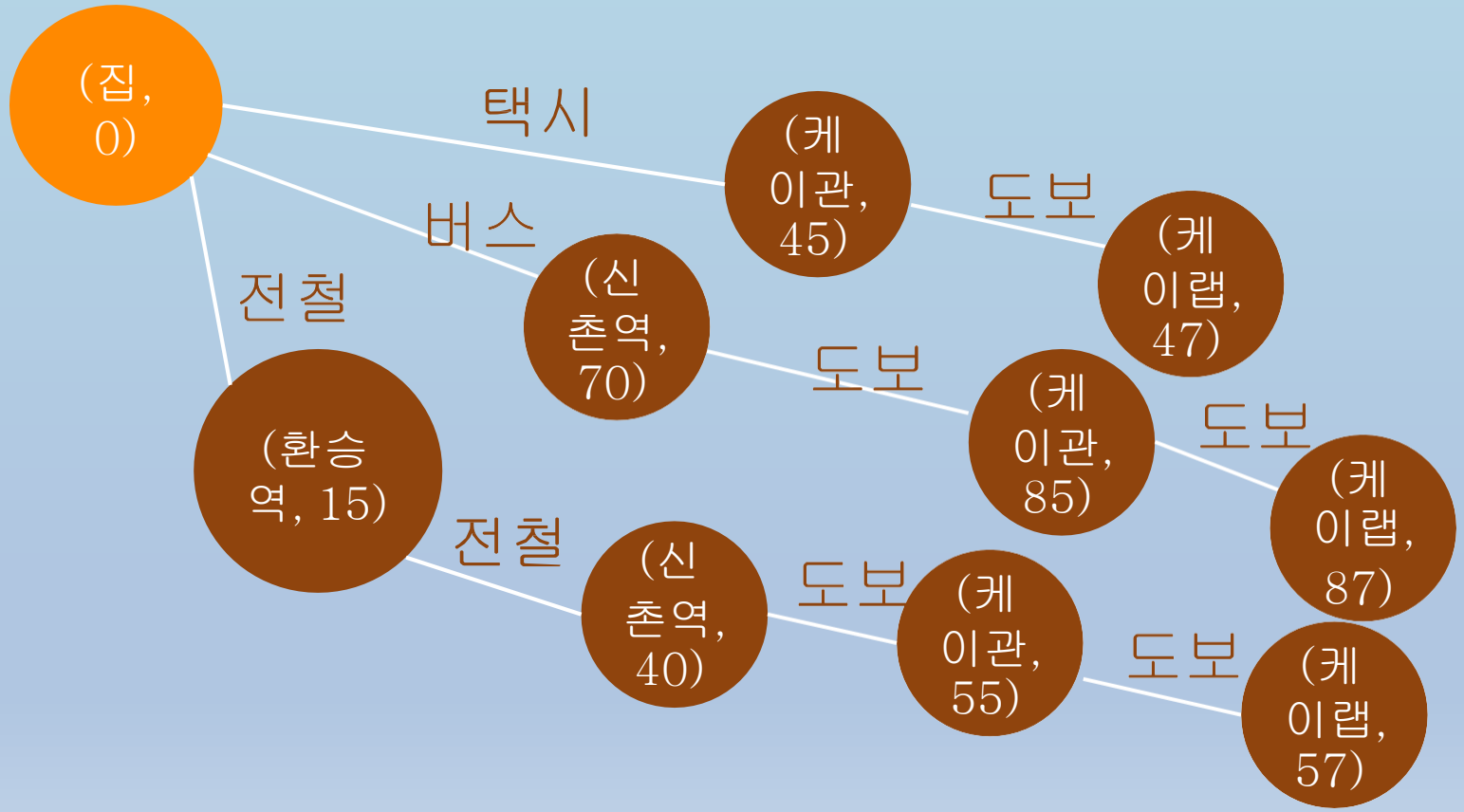
간단히 말하자면…

갈 수 있을때까지
가보고, 다시 돌아와서
다른 길로 또 가보고, …

stack

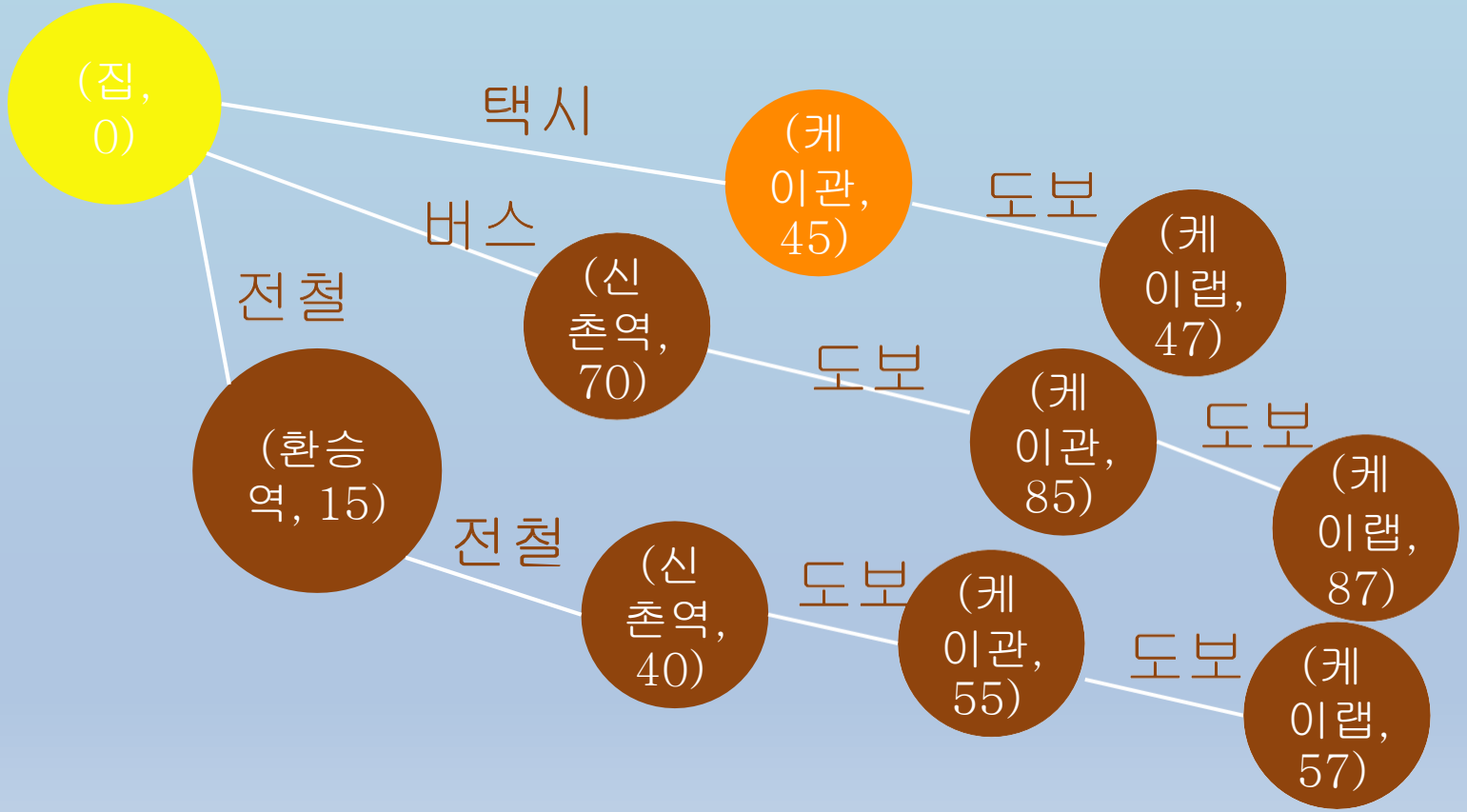
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



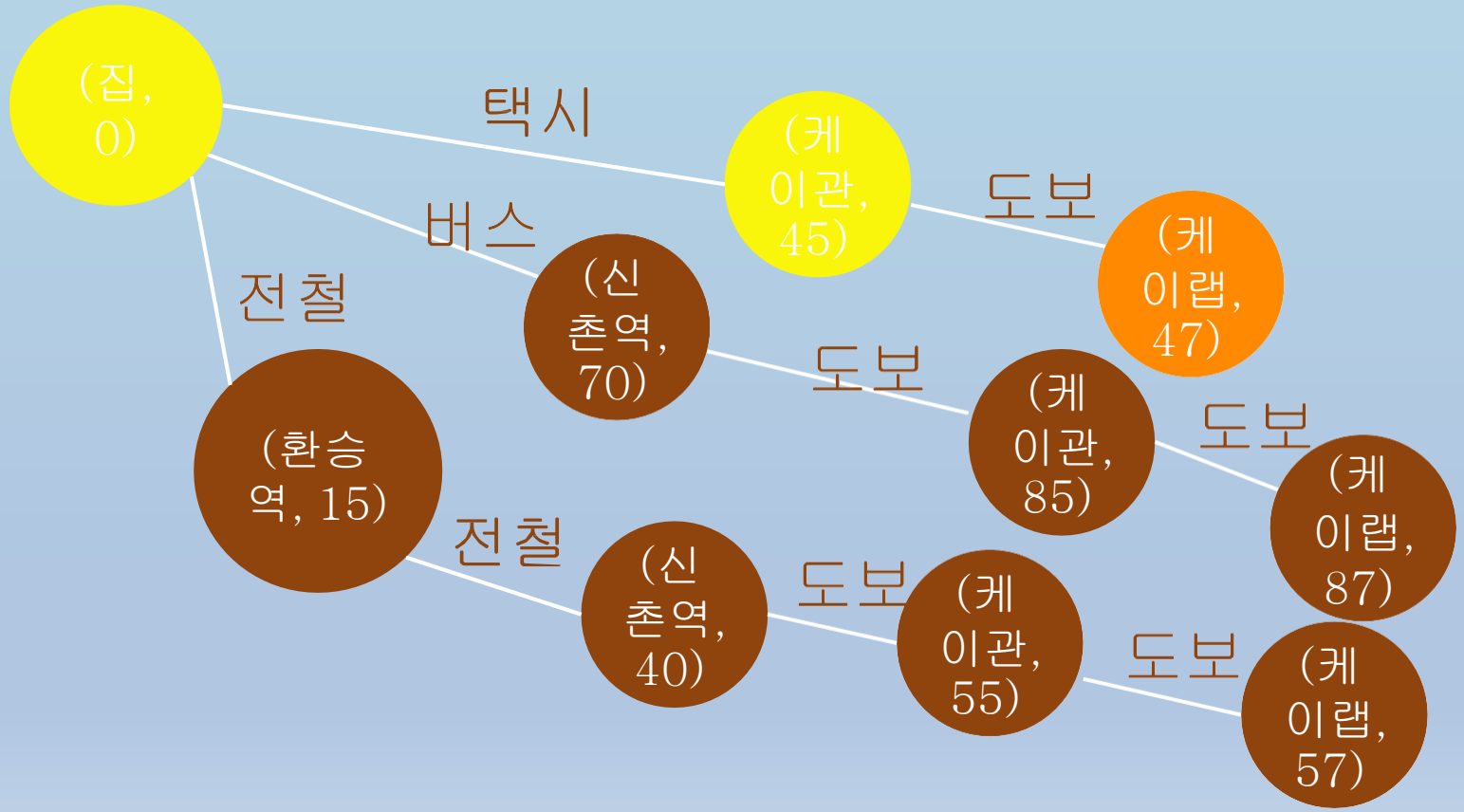
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



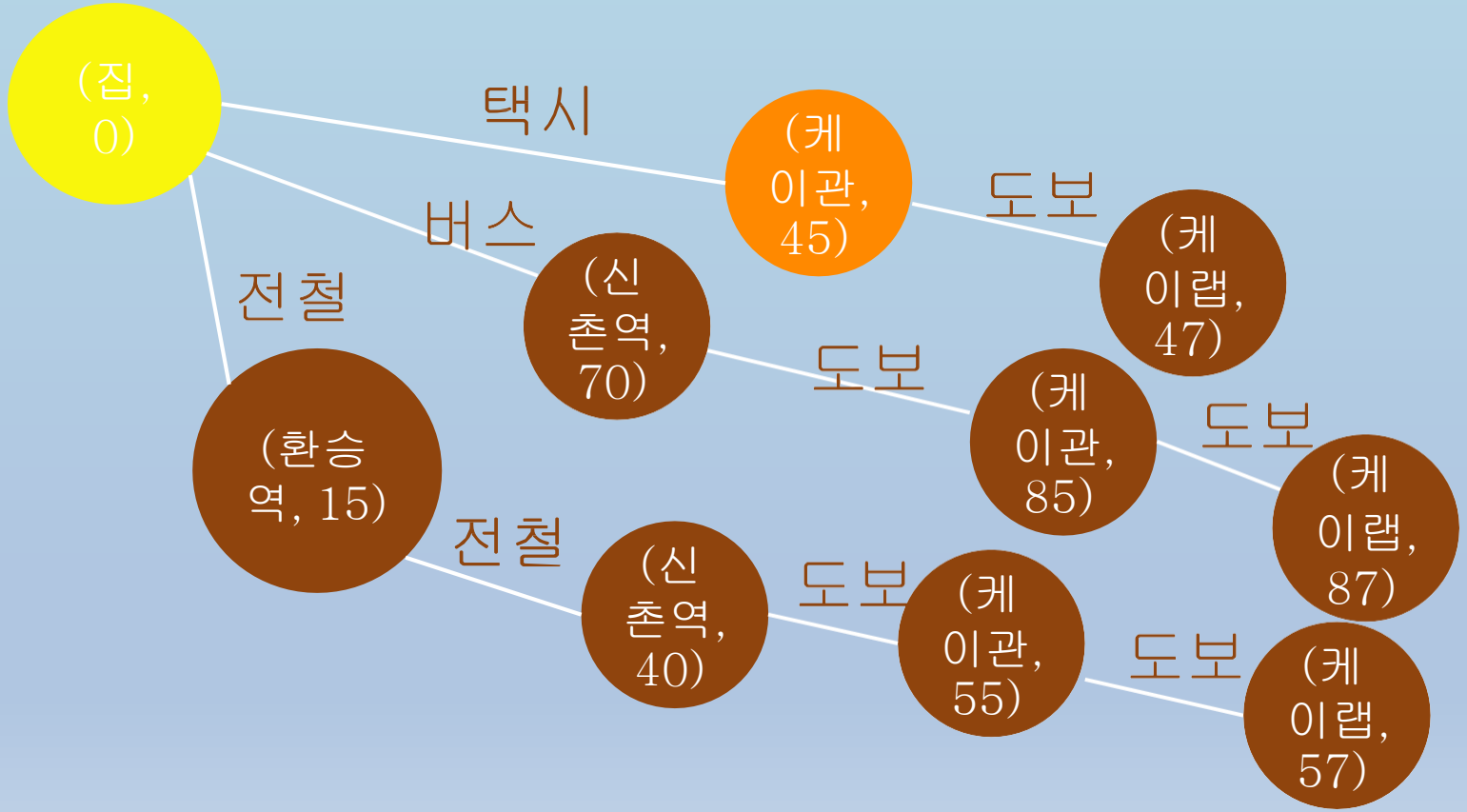
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



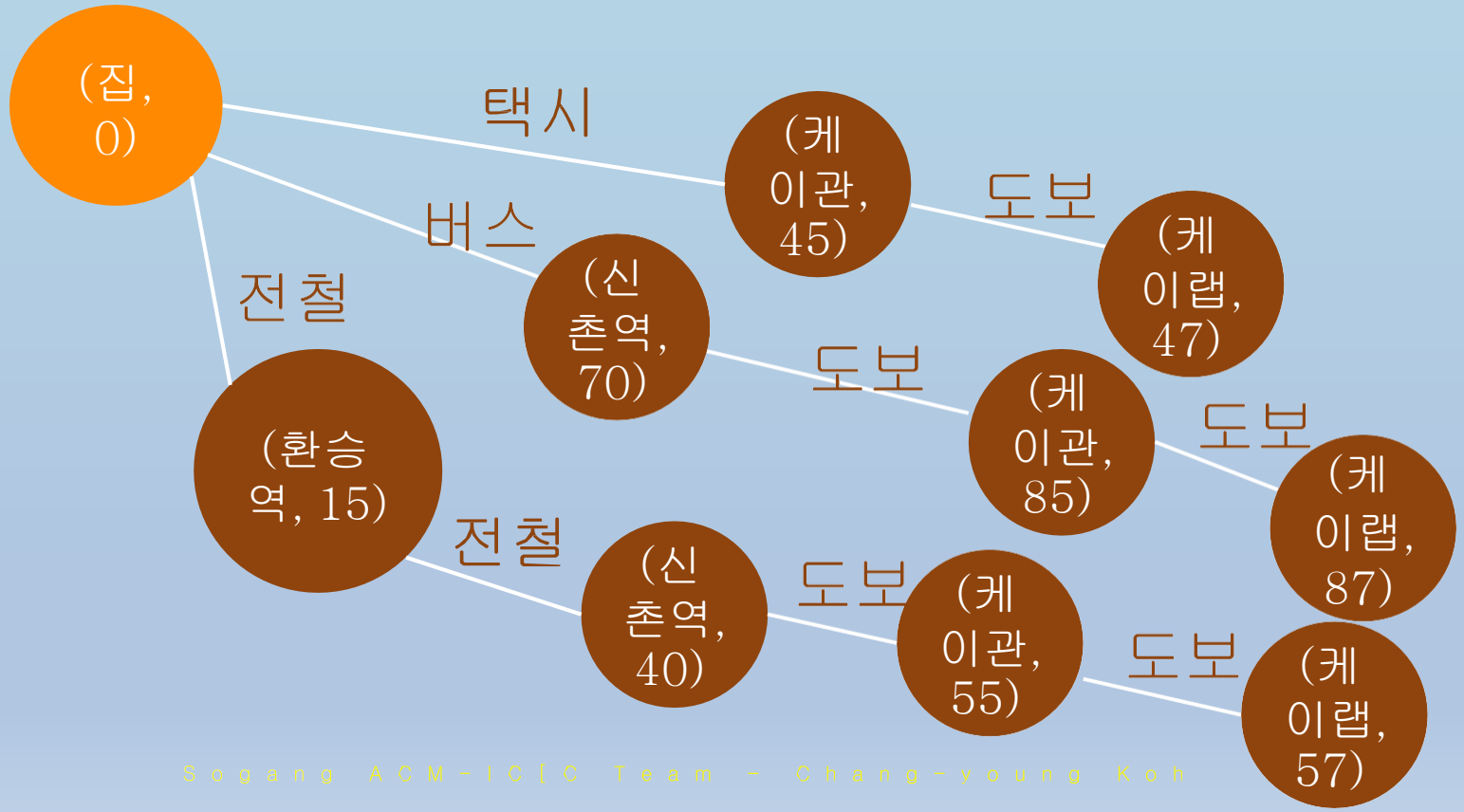
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



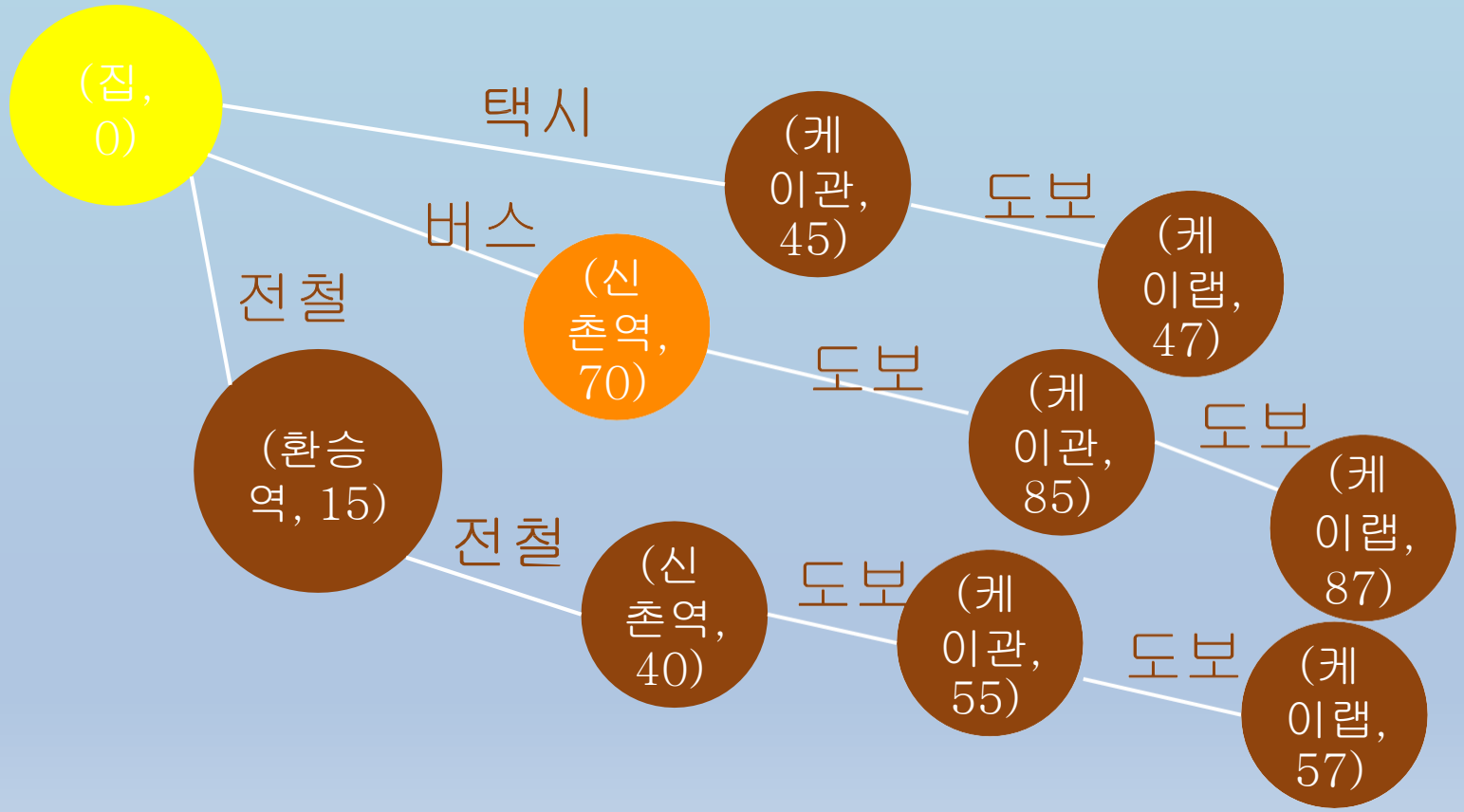
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



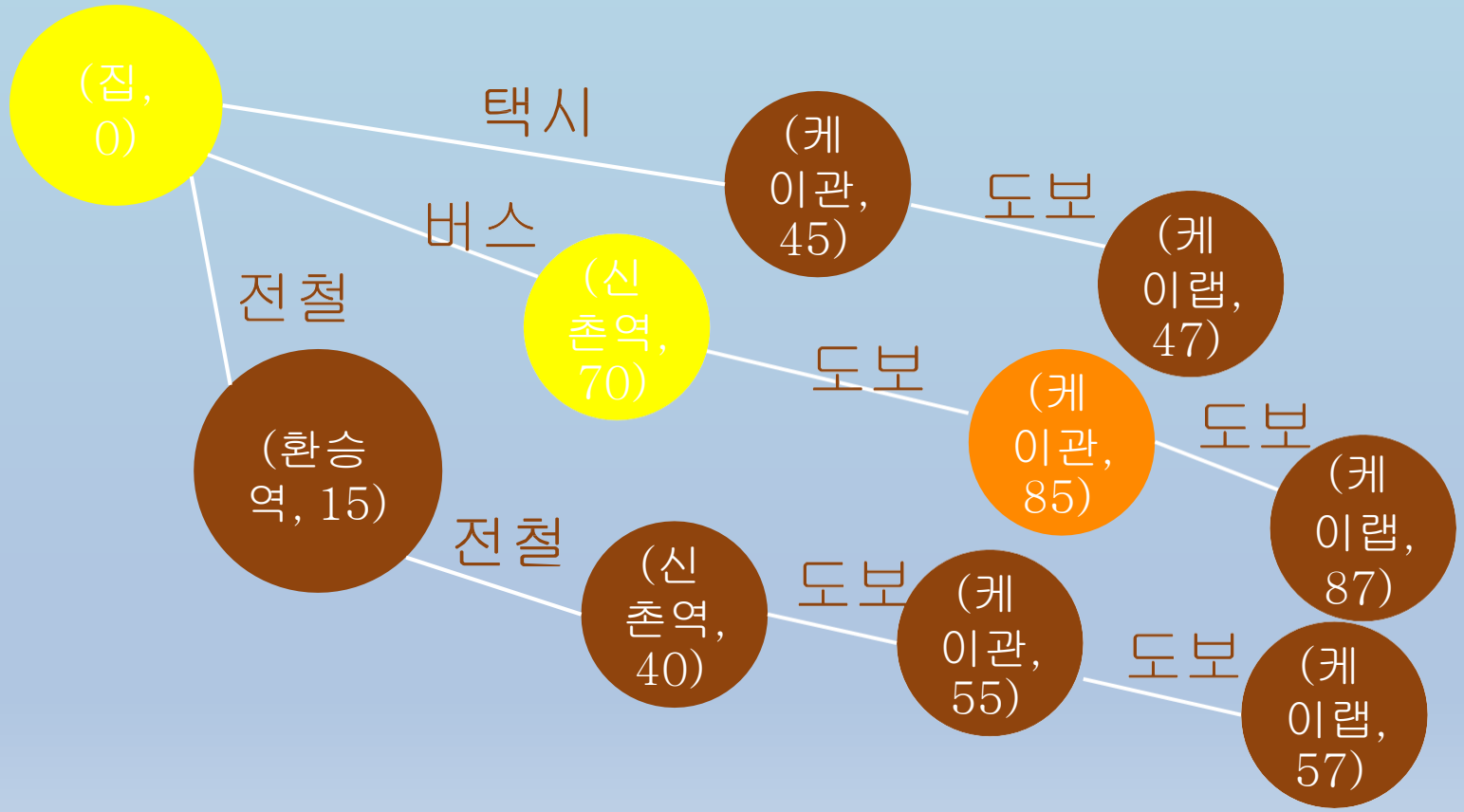
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



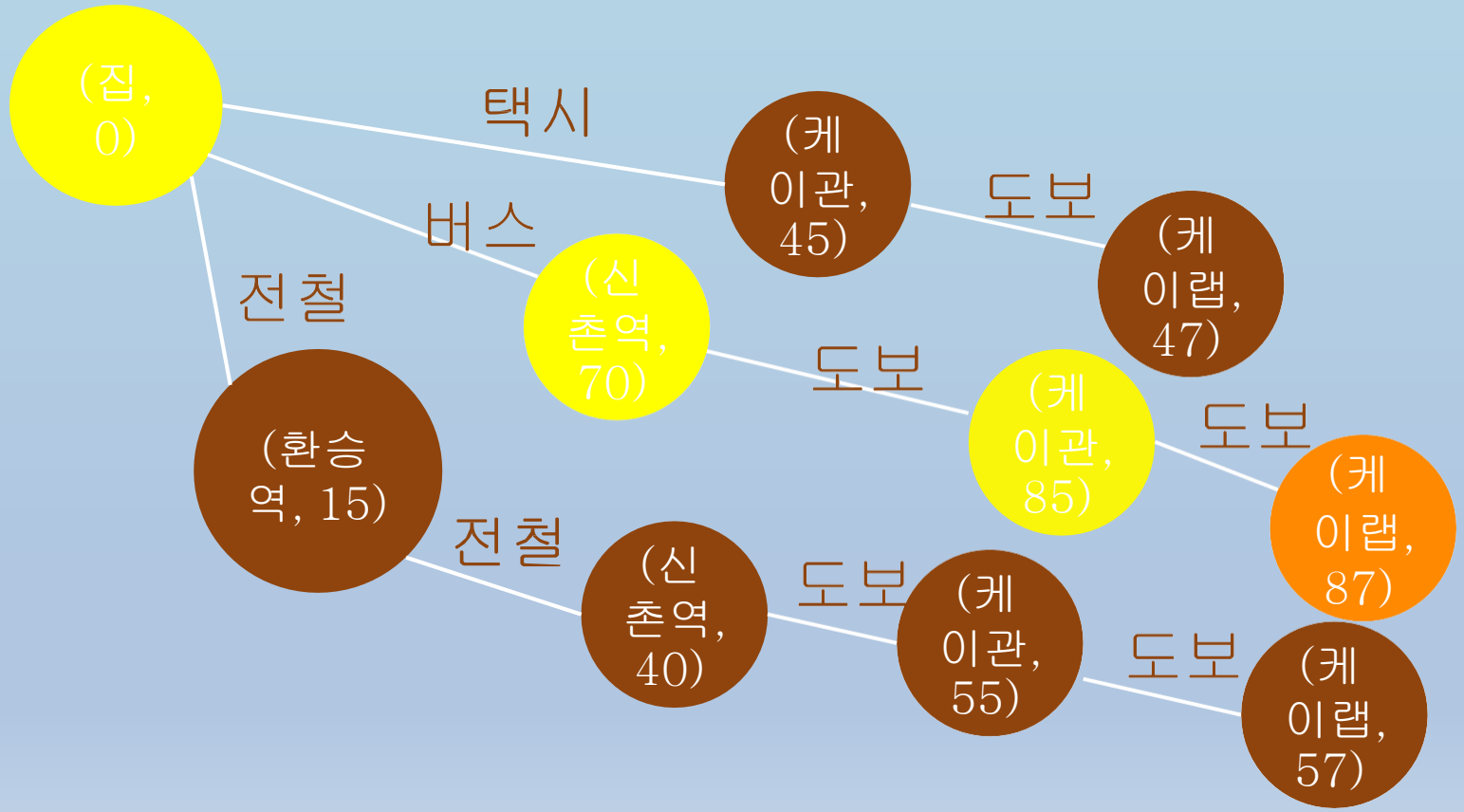
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



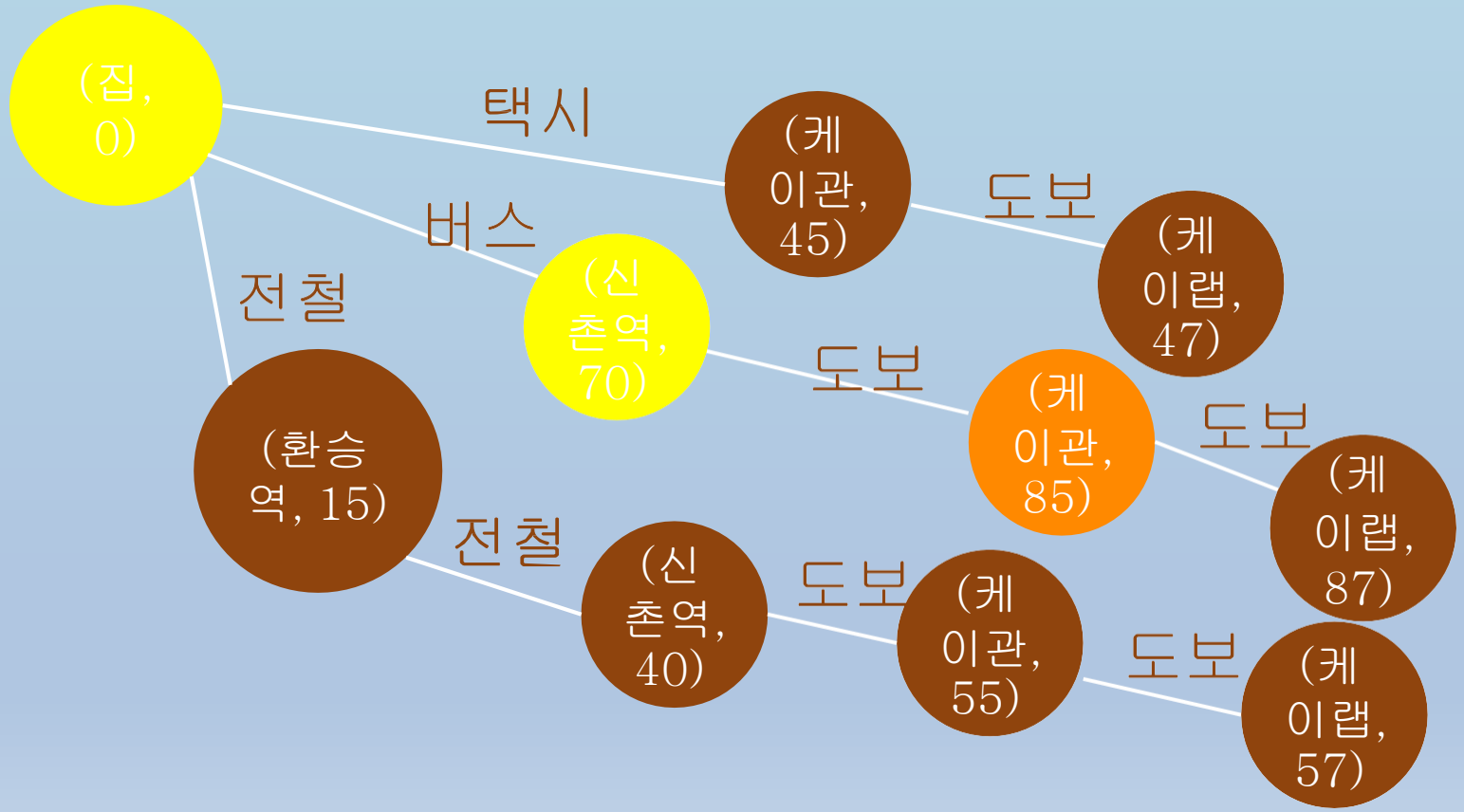
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



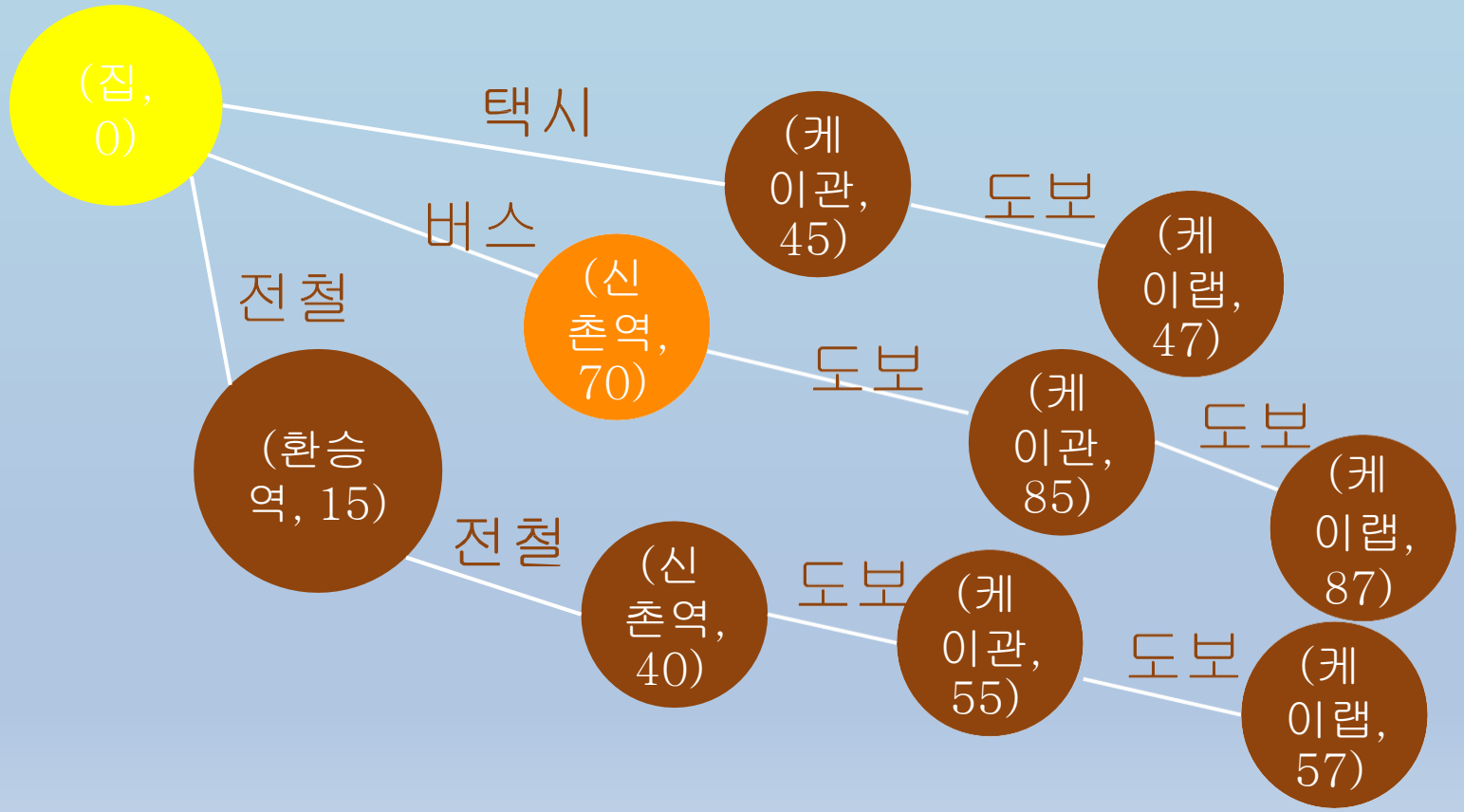
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



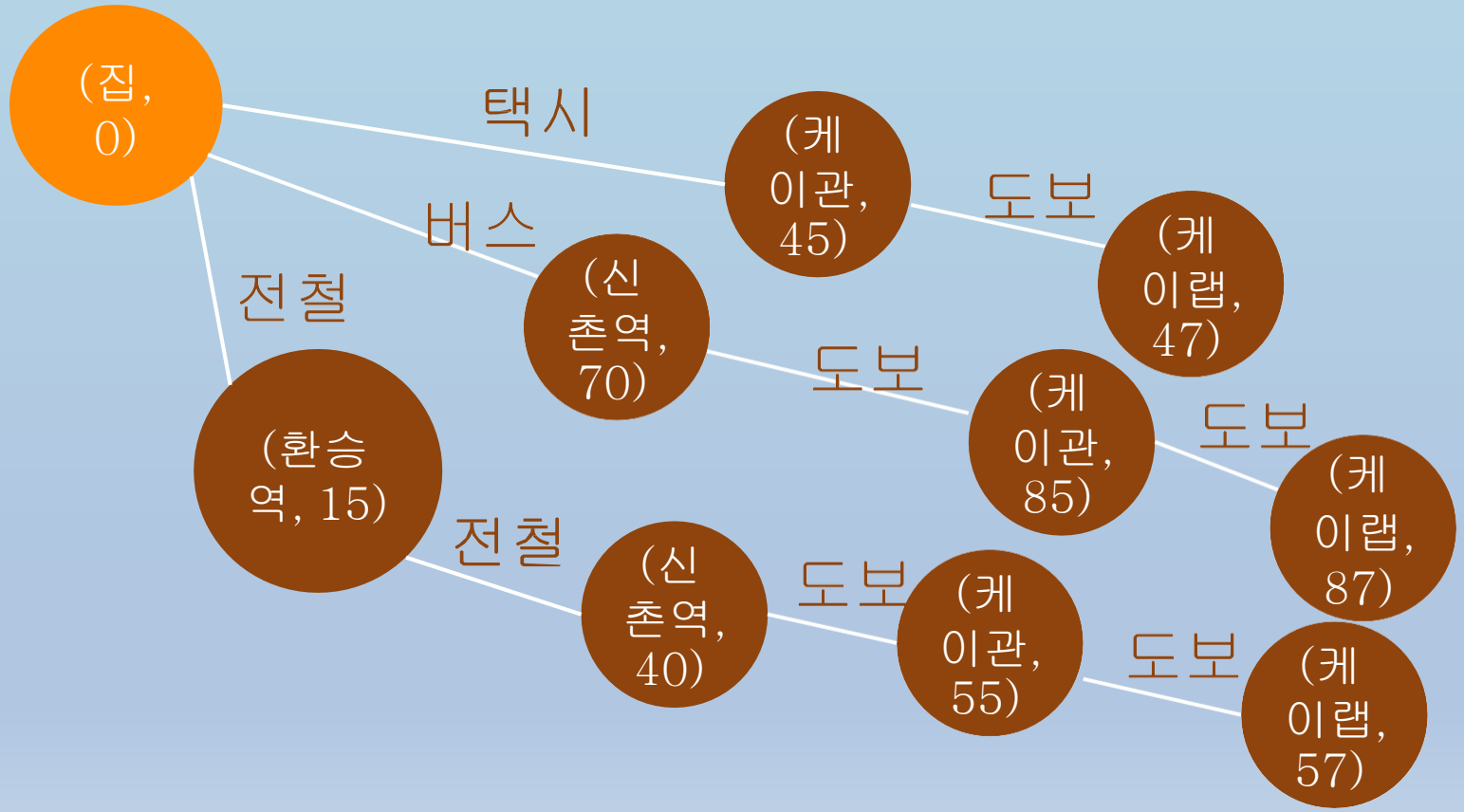
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



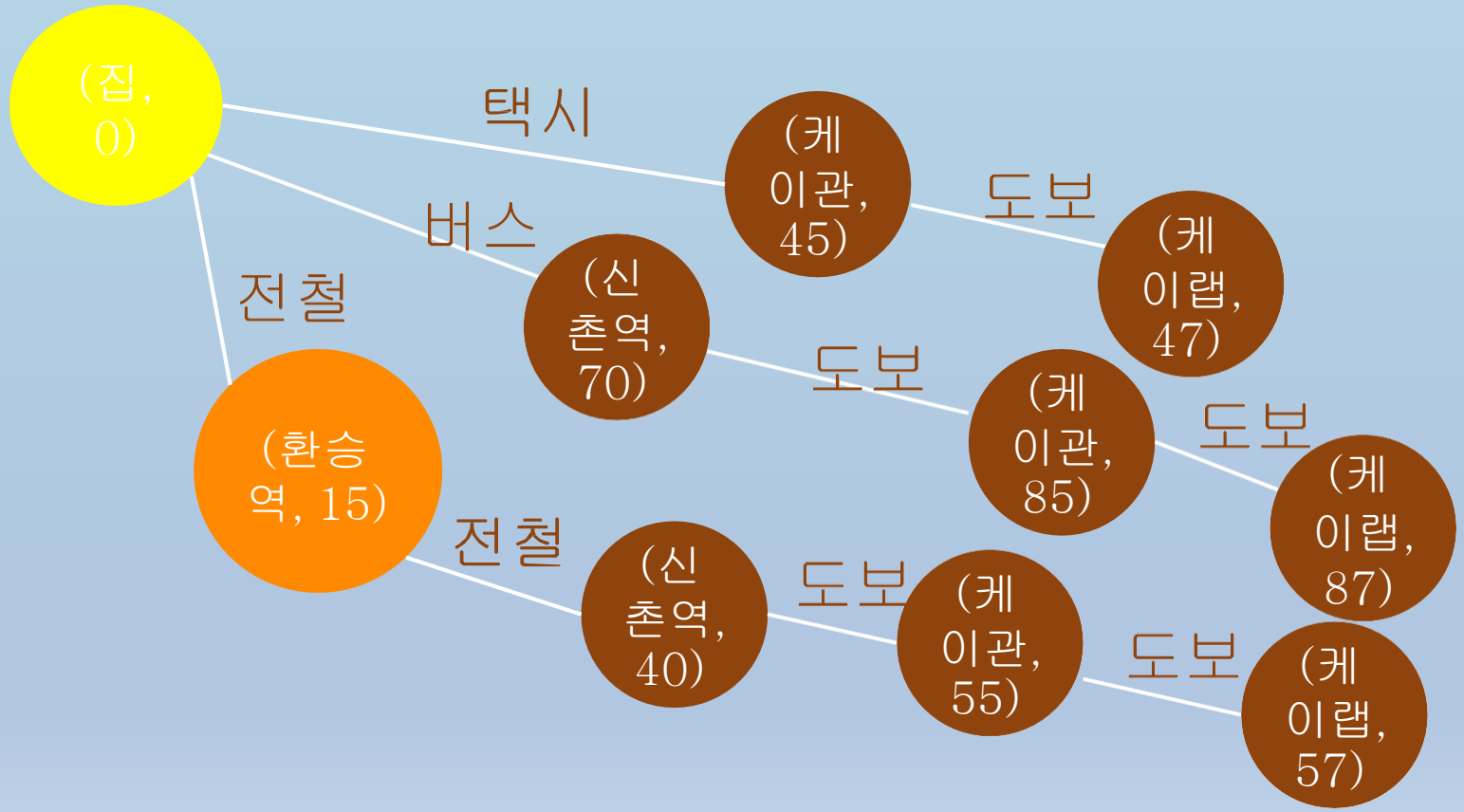
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



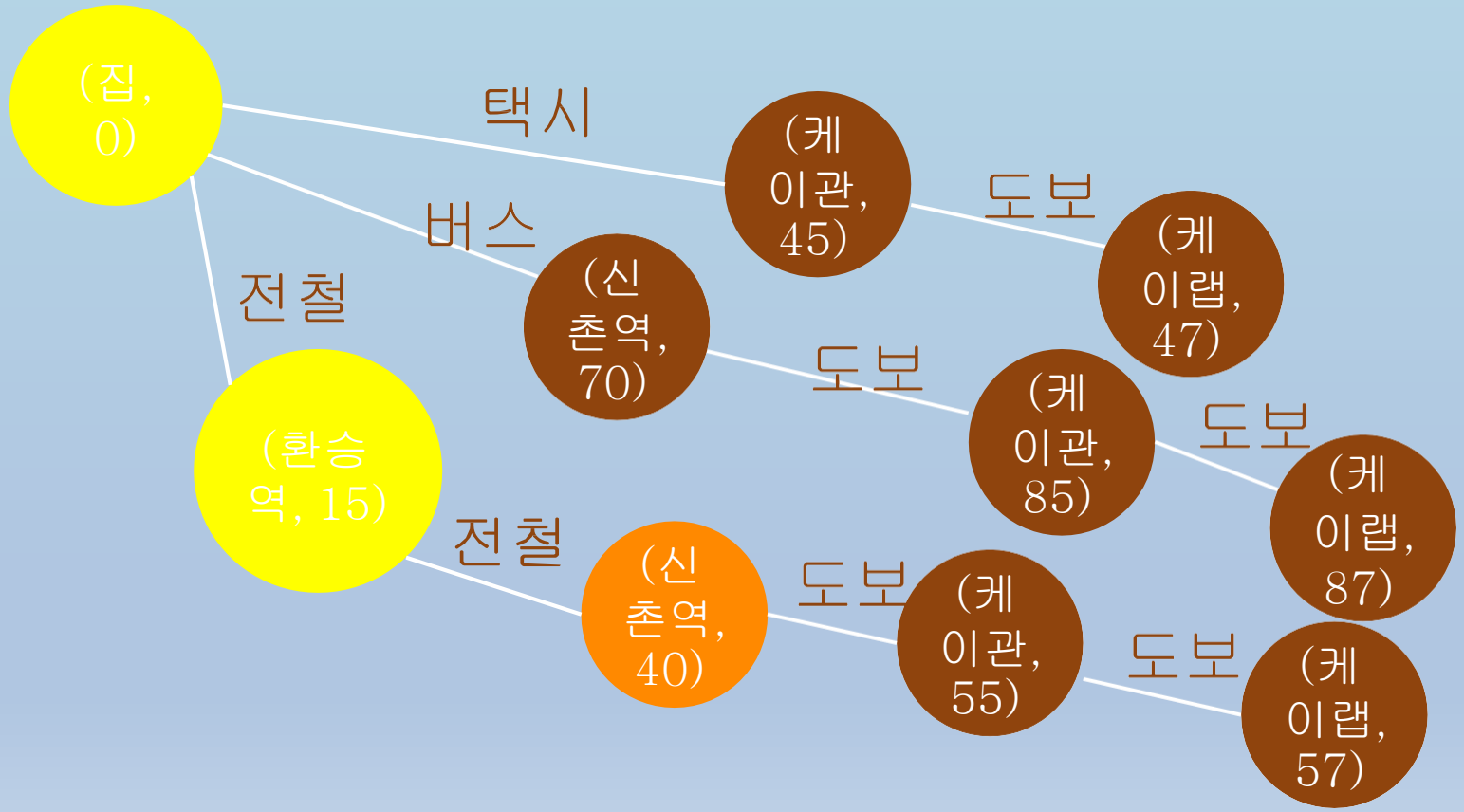
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



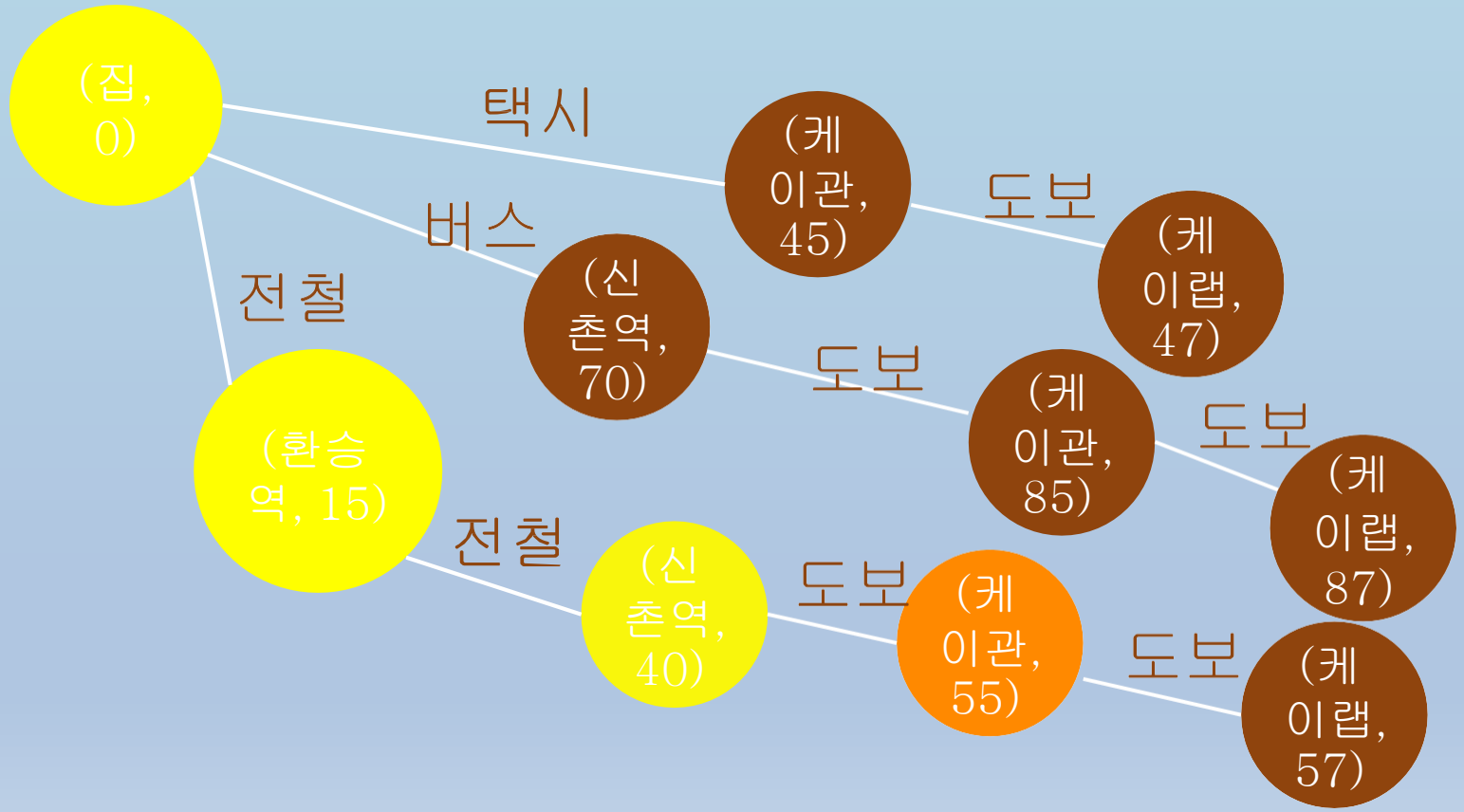
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



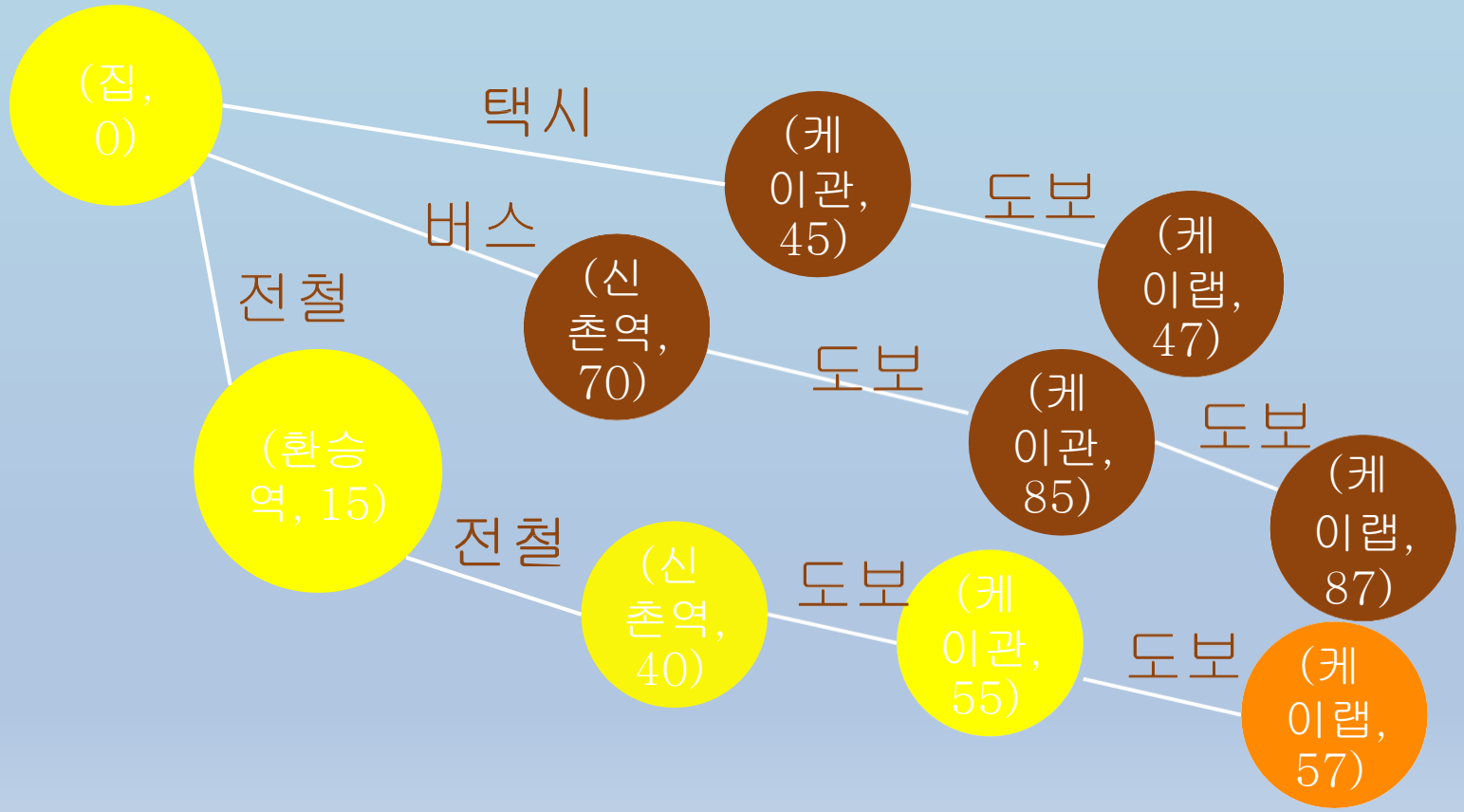
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



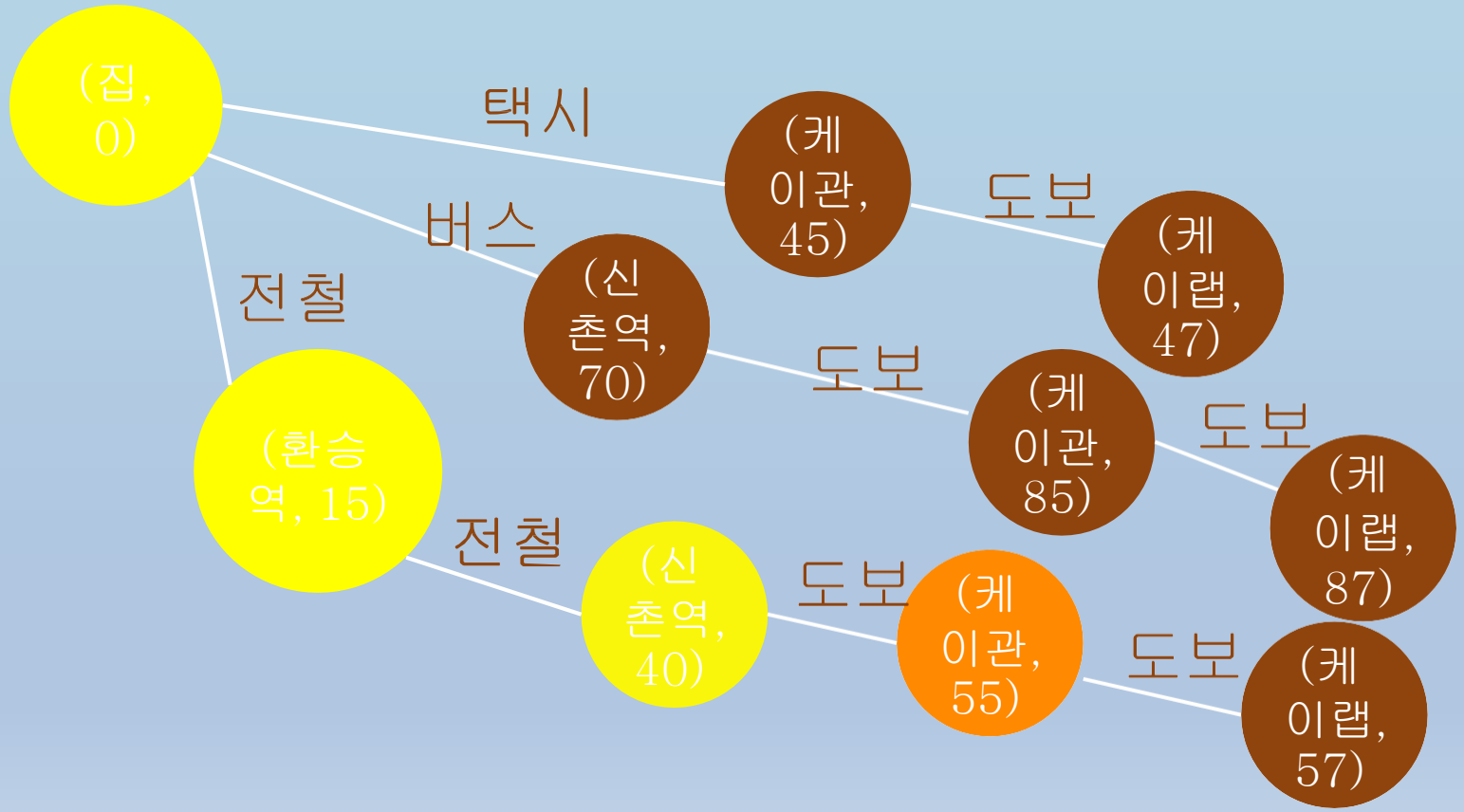
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



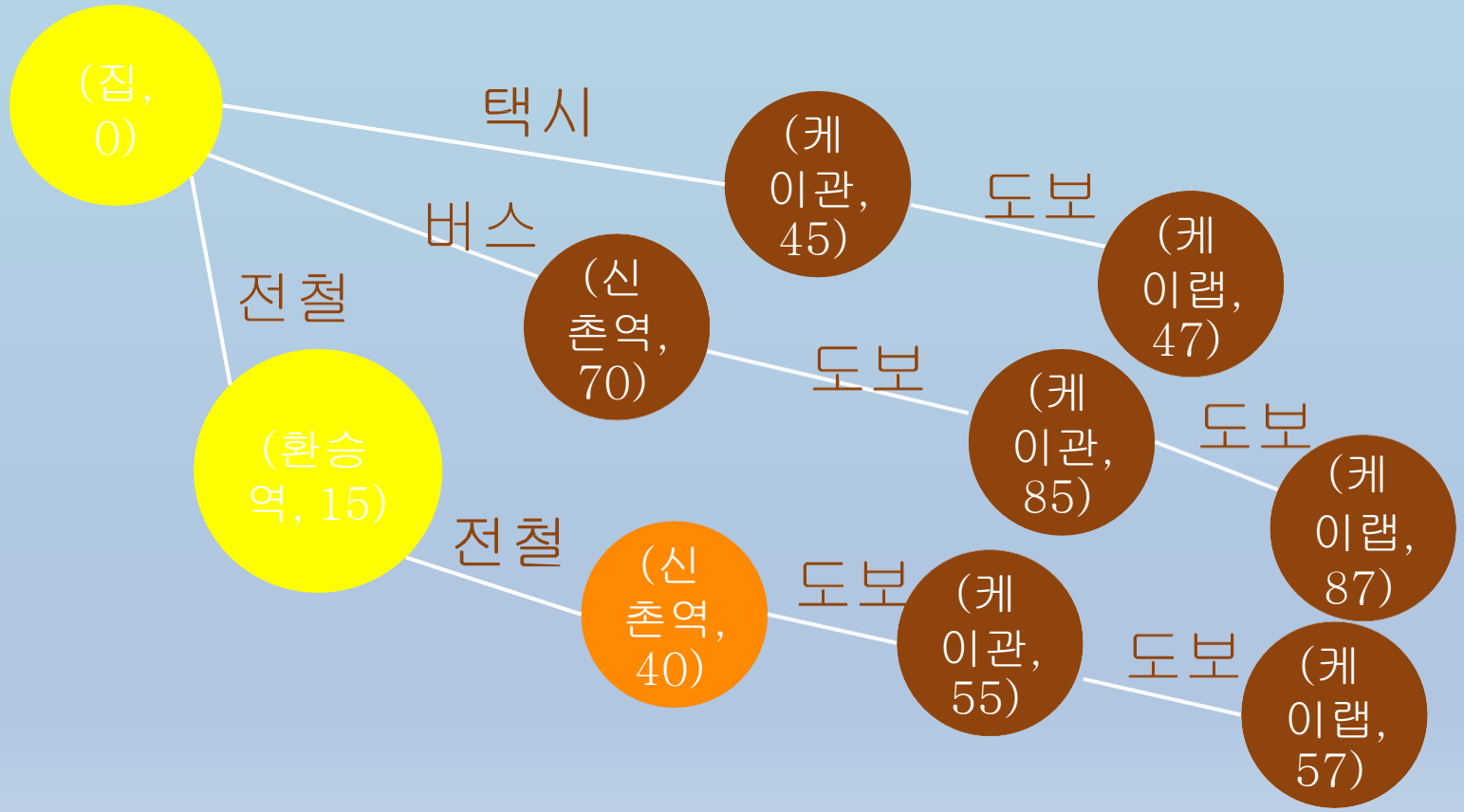
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



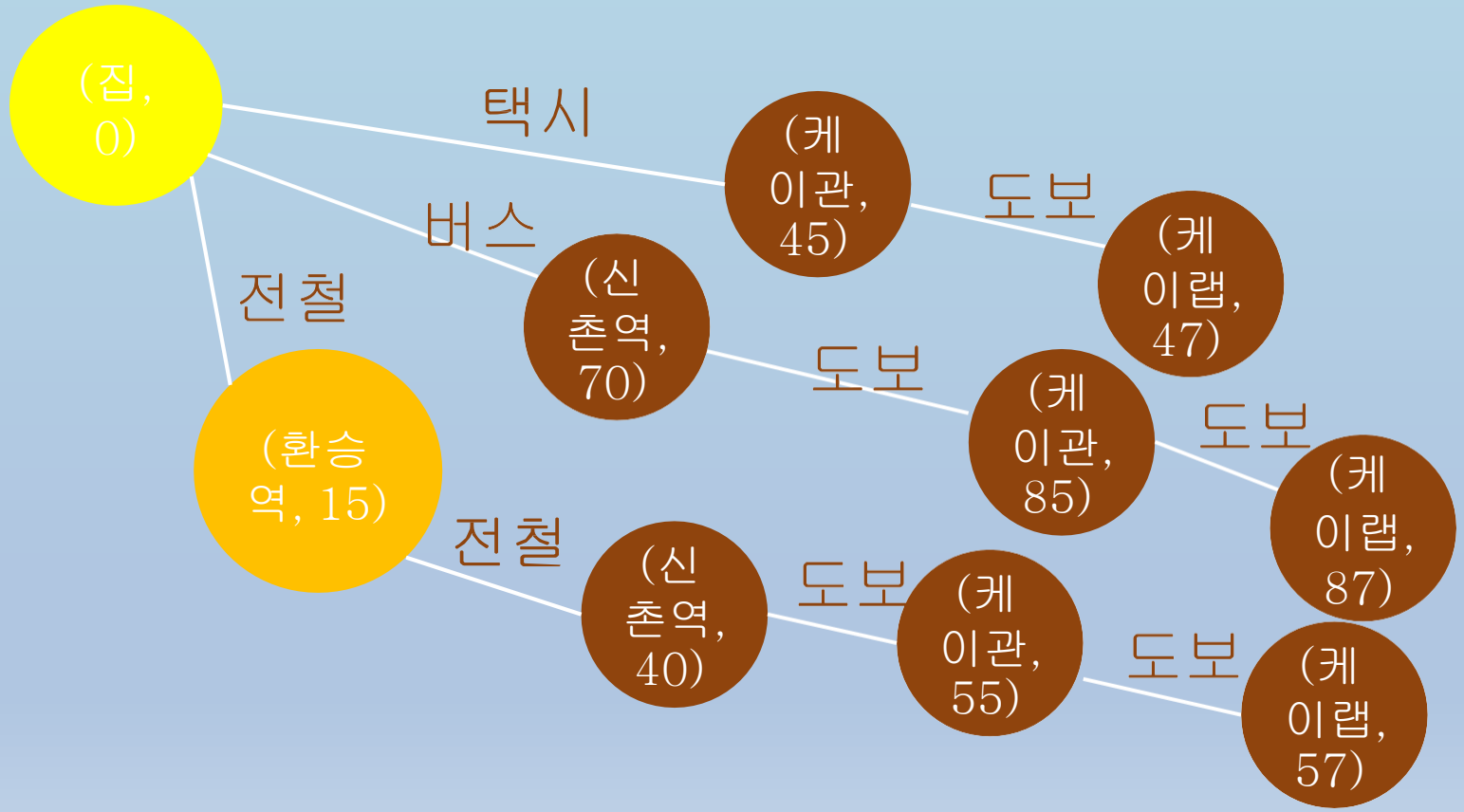
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



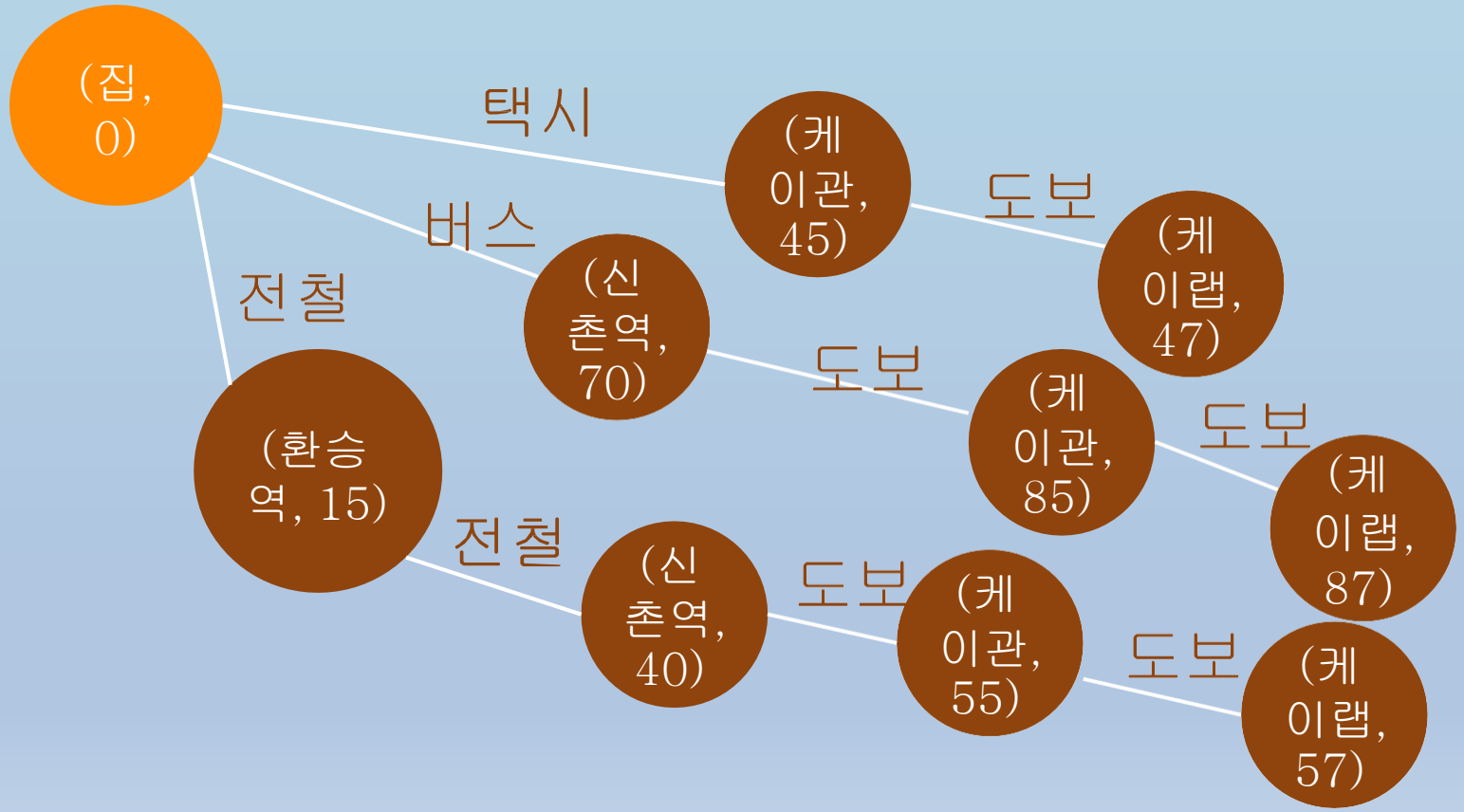
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



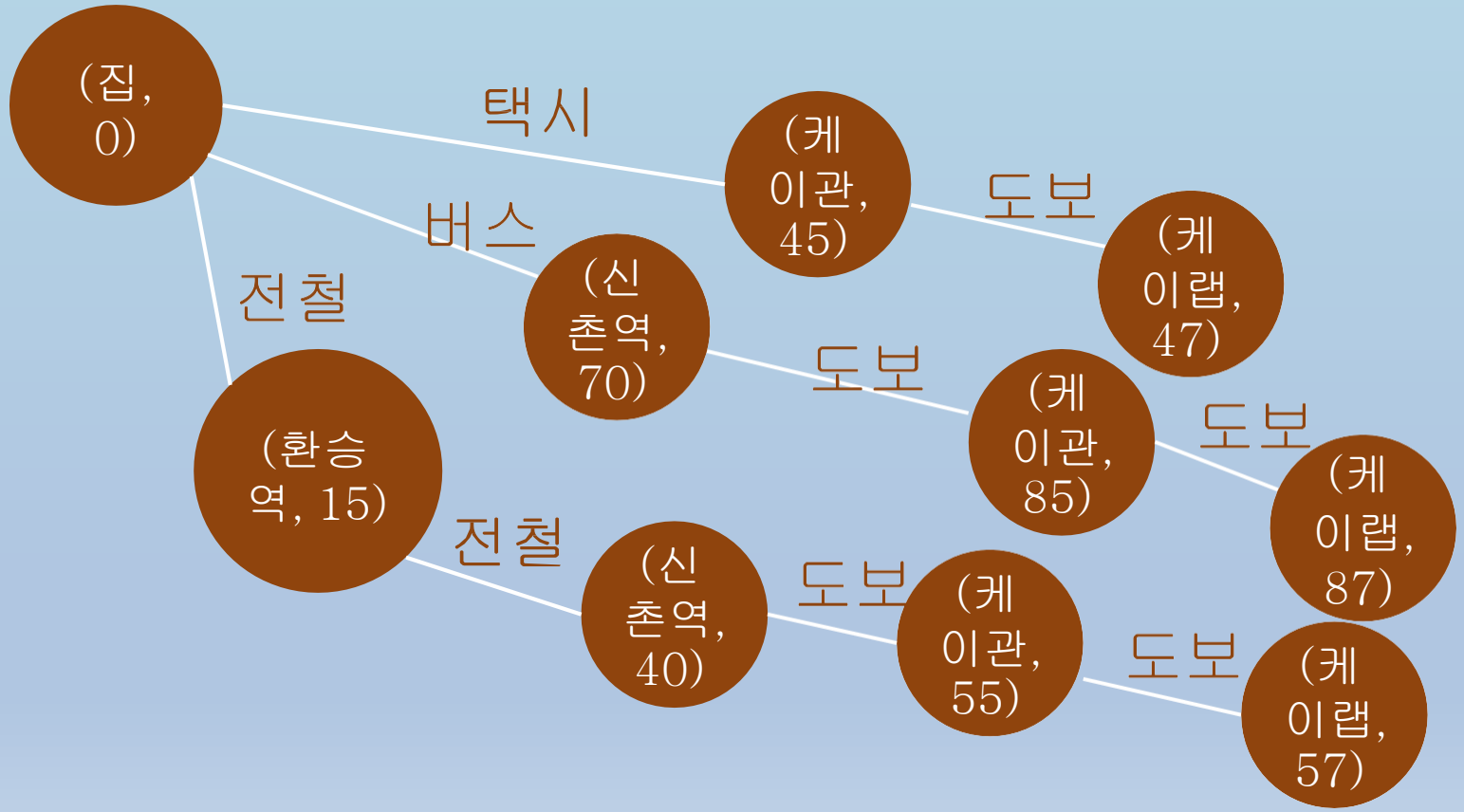
DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



DFS

주황색 : 현재 상태,
노란색 : 스택에 쌓인 상태.



BFS

Breadth – First – Search

너비 – 우선 – 탐색

DFS

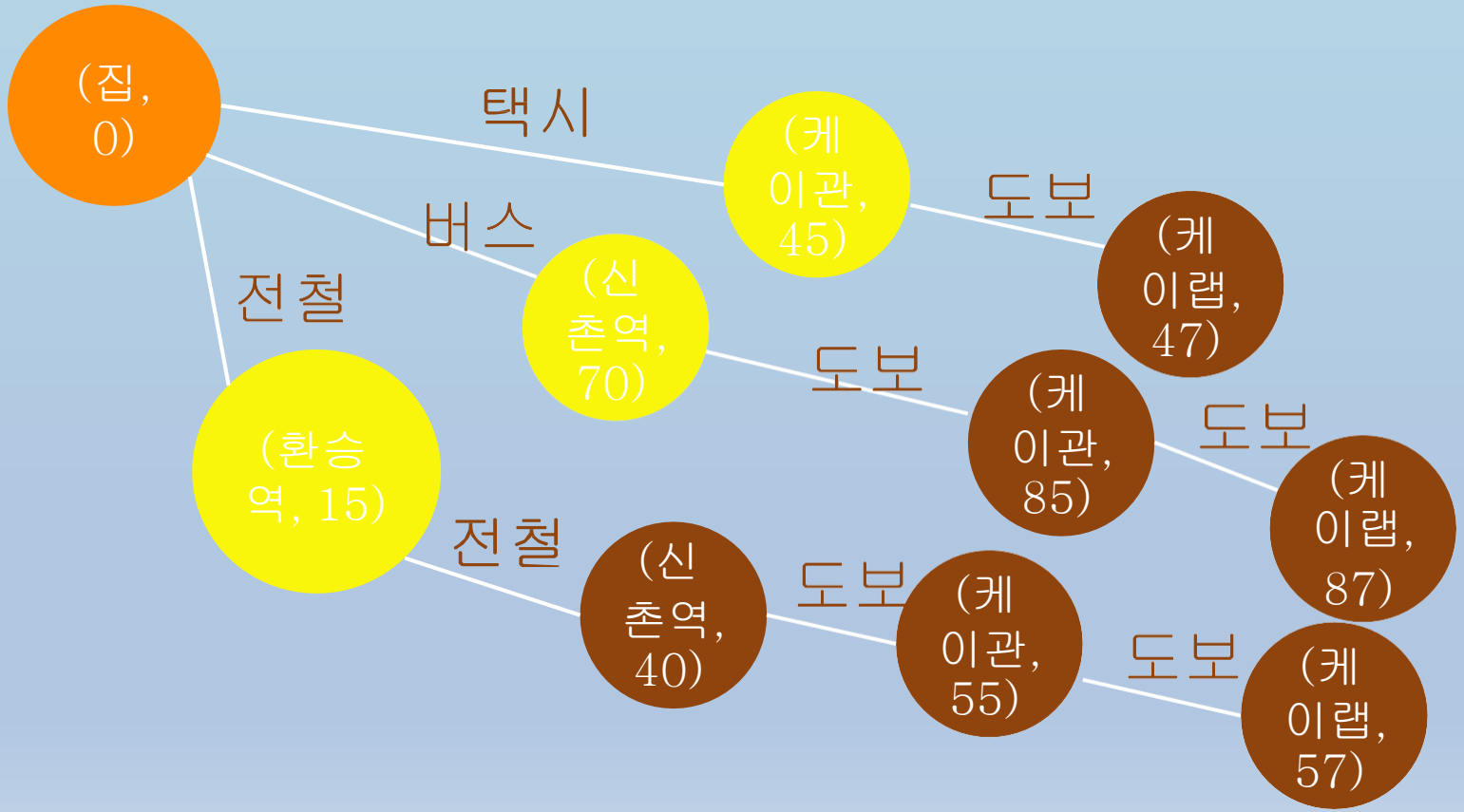
간단히 말하자면…

내가 여러 명으로 복사되어
동시에 여러 길을 가보는것!

Queue

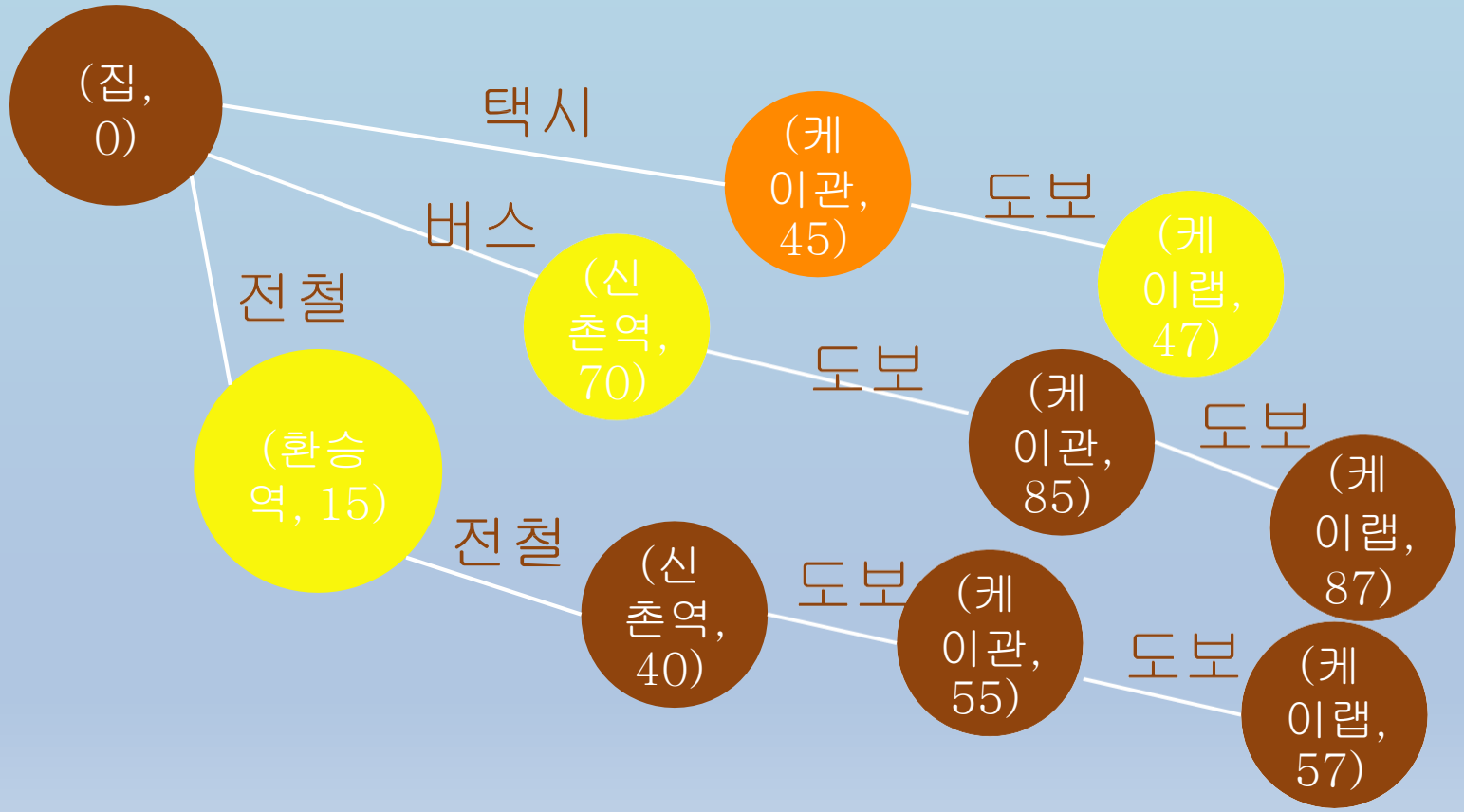
BFS

주황색 : 현재 상태,
노란색 : 큐에 쌓인 상태.



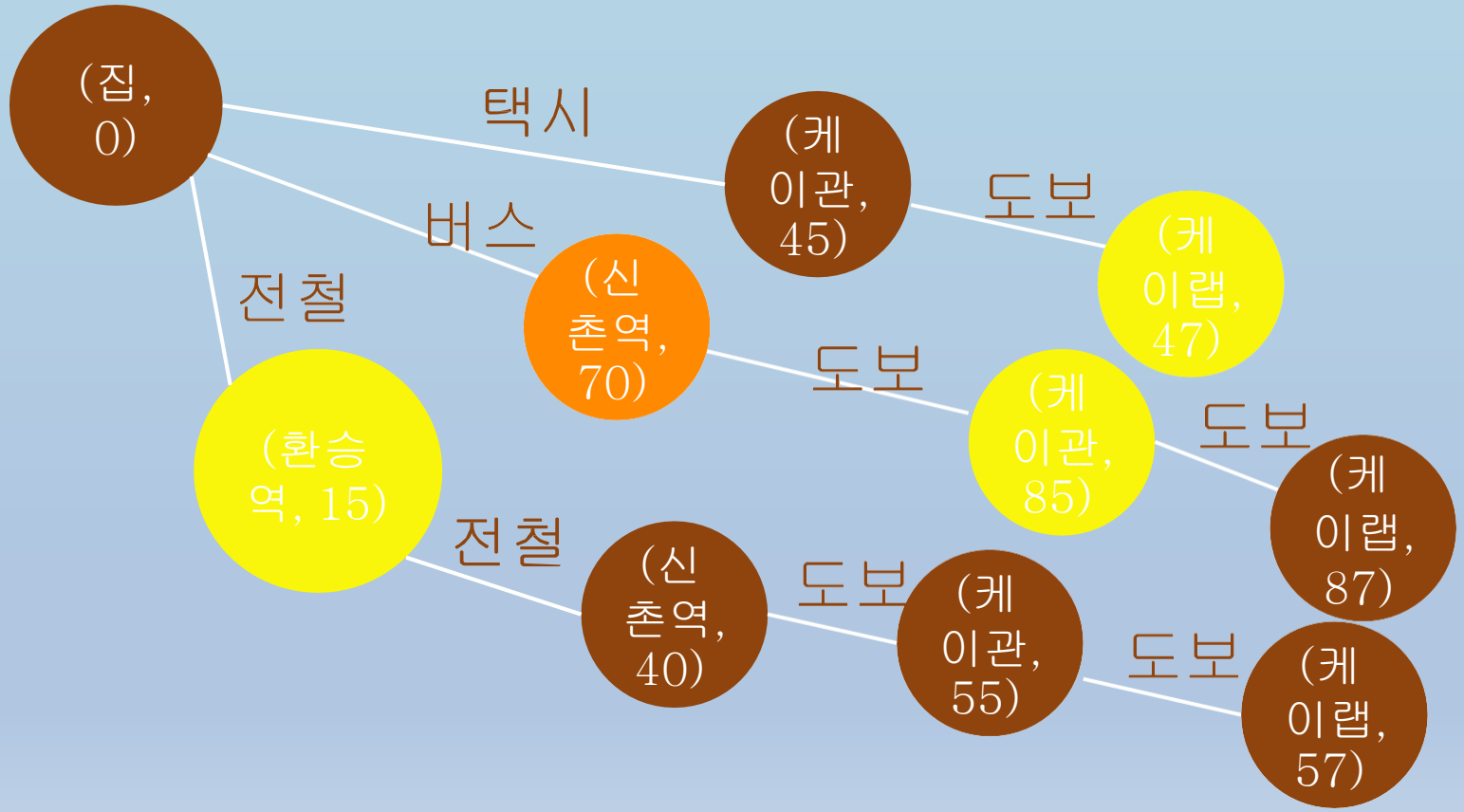
BFS

주황색 : 현재 상태,
노란색 : 큐에 쌓인 상태.



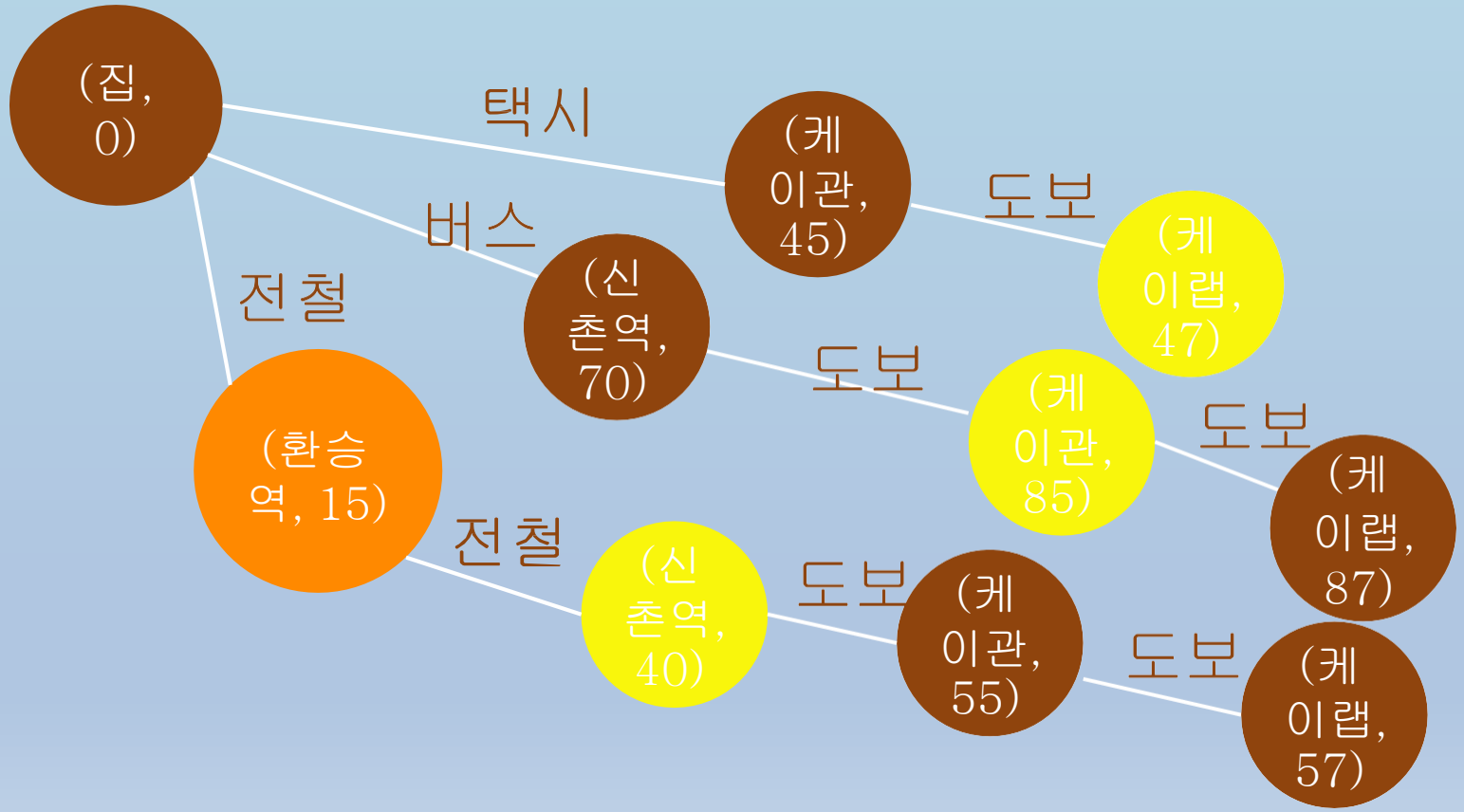
BFS

주황색 : 현재 상태,
노란색 : 큐에 쌓인 상태.



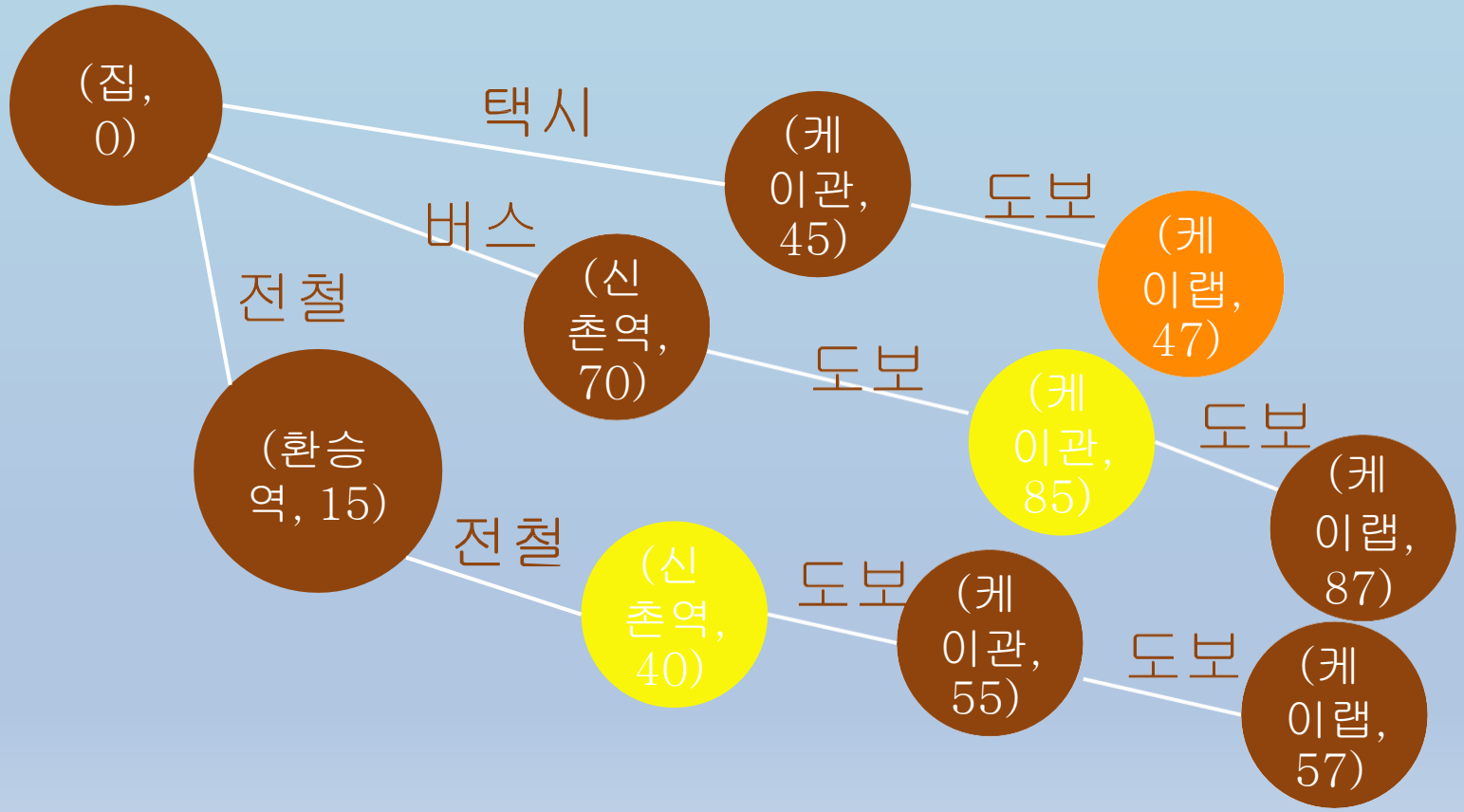
BFS

주황색 : 현재 상태,
노란색 : 큐에 쌓인 상태.



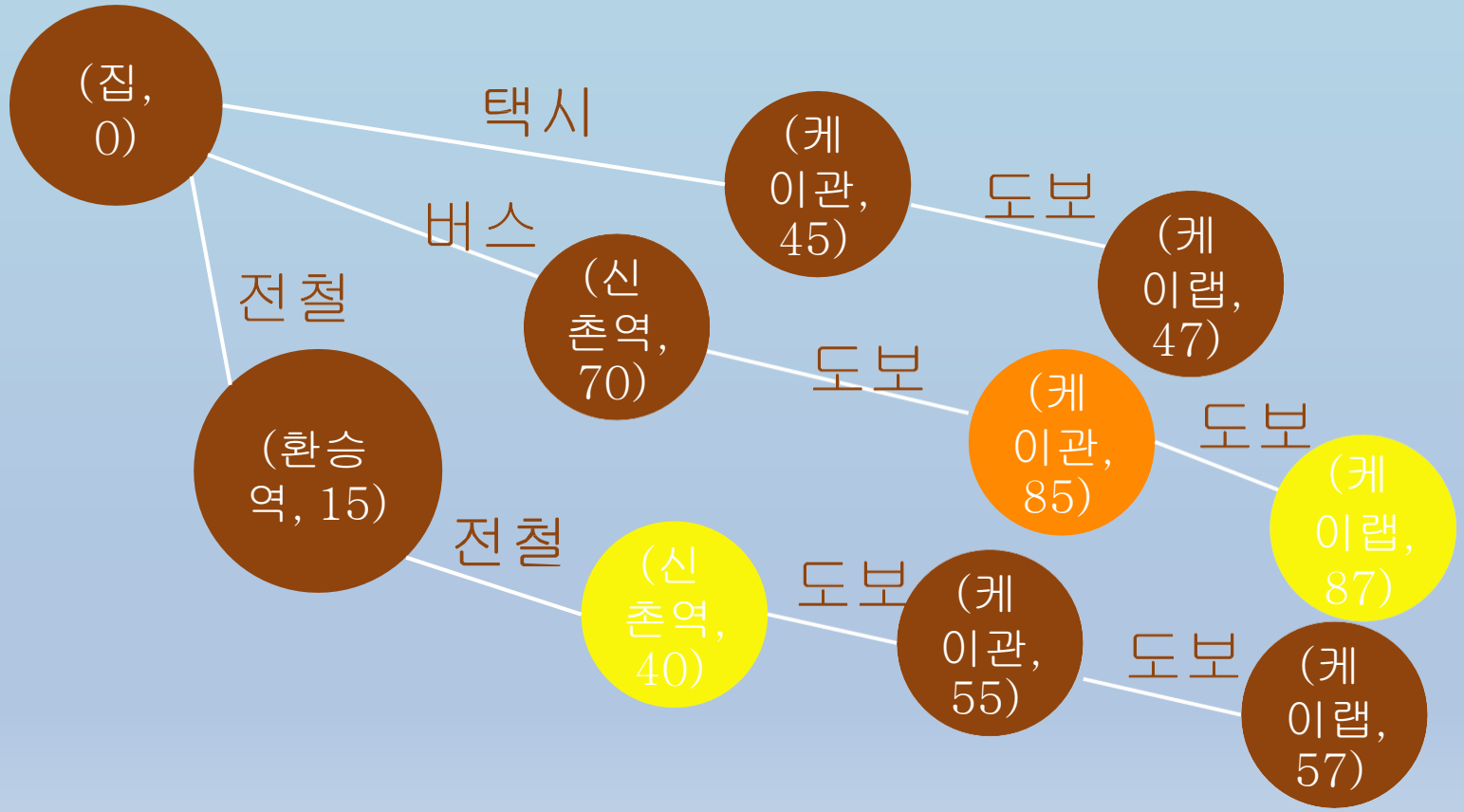
BFS

주황색 : 현재 상태,
노란색 : 큐에 쌓인 상태.



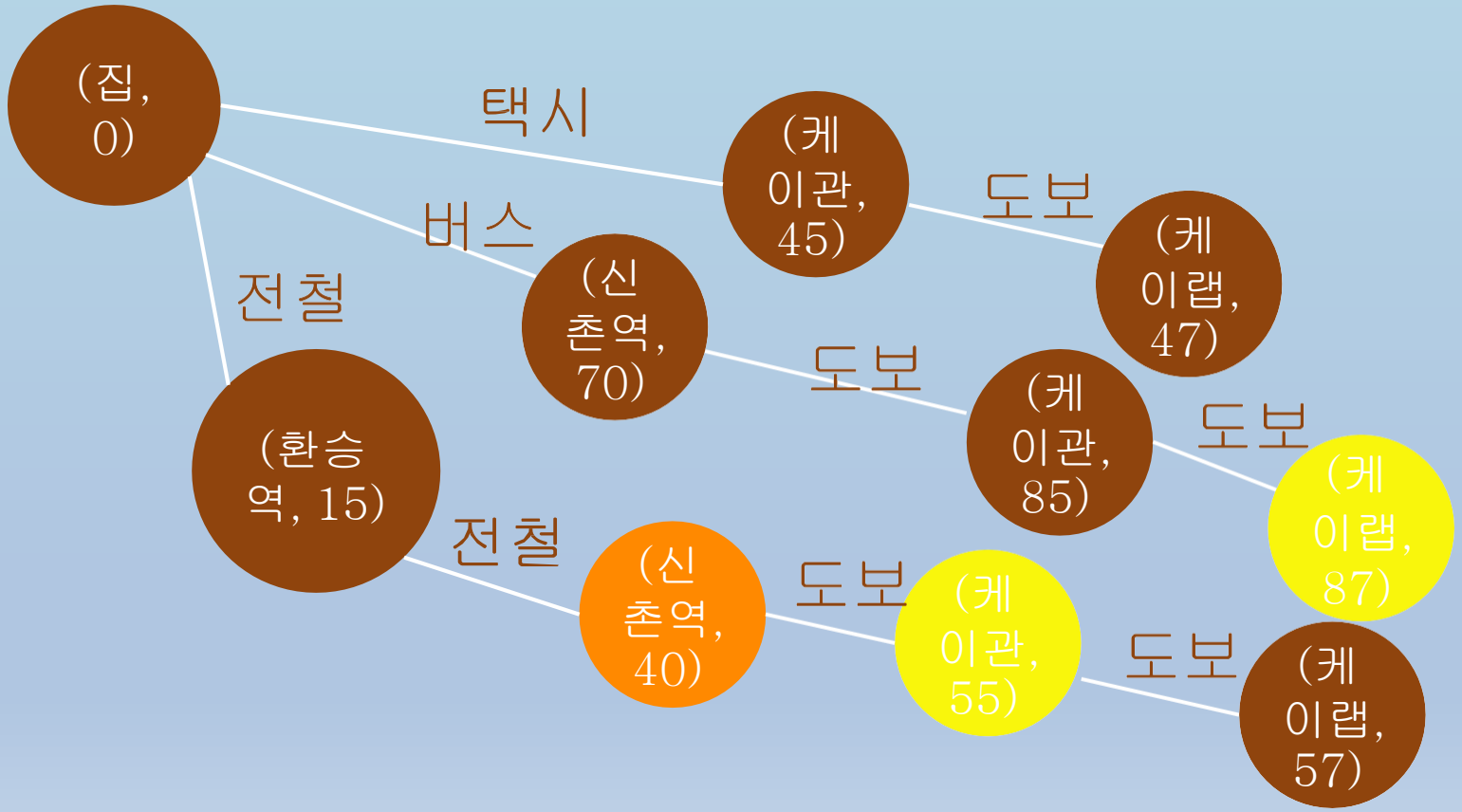
BFS

주황색 : 현재 상태,
노란색 : 큐에 쌓인 상태.



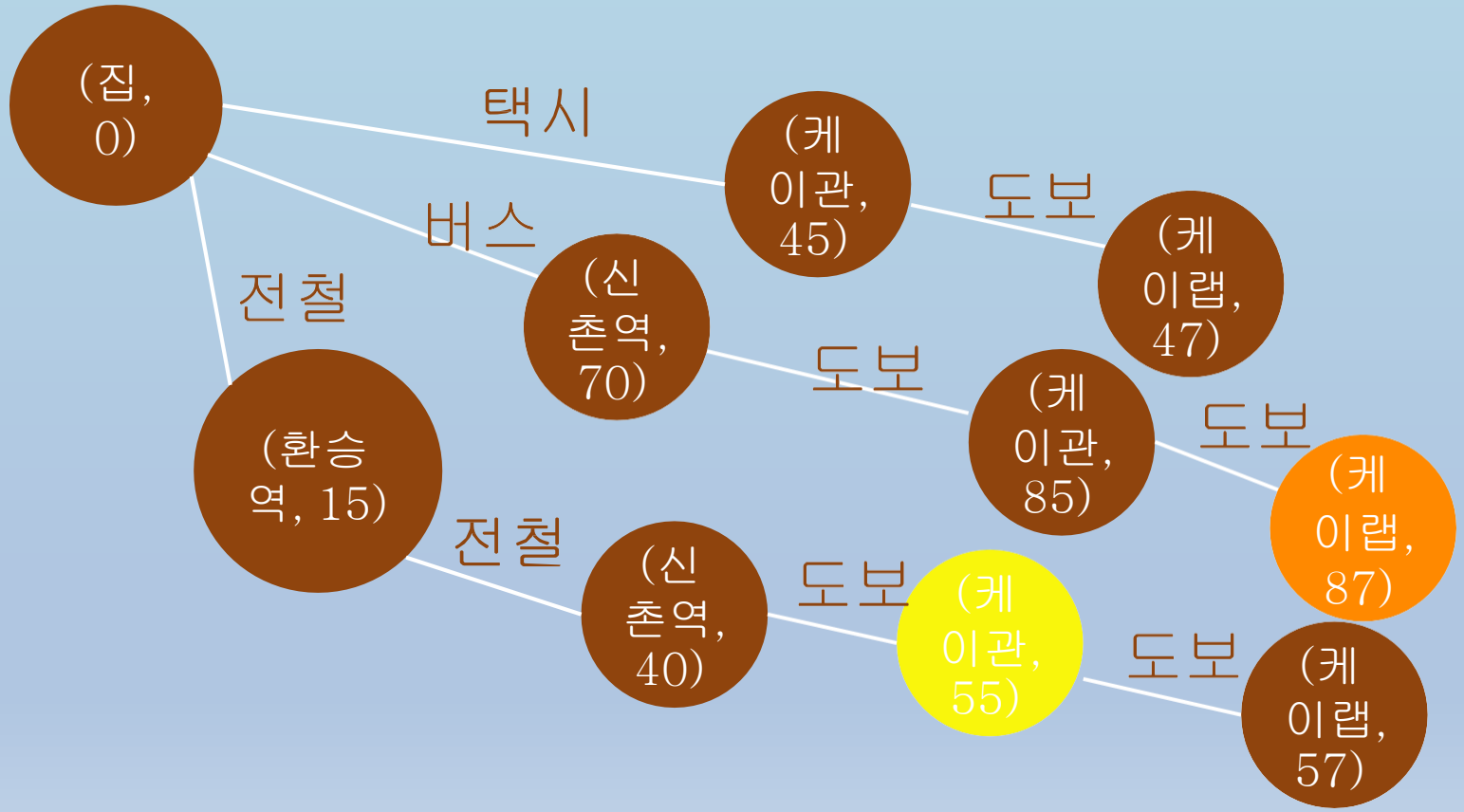
BFS

주황색 : 현재 상태,
노란색 : 큐에 쌓인 상태.



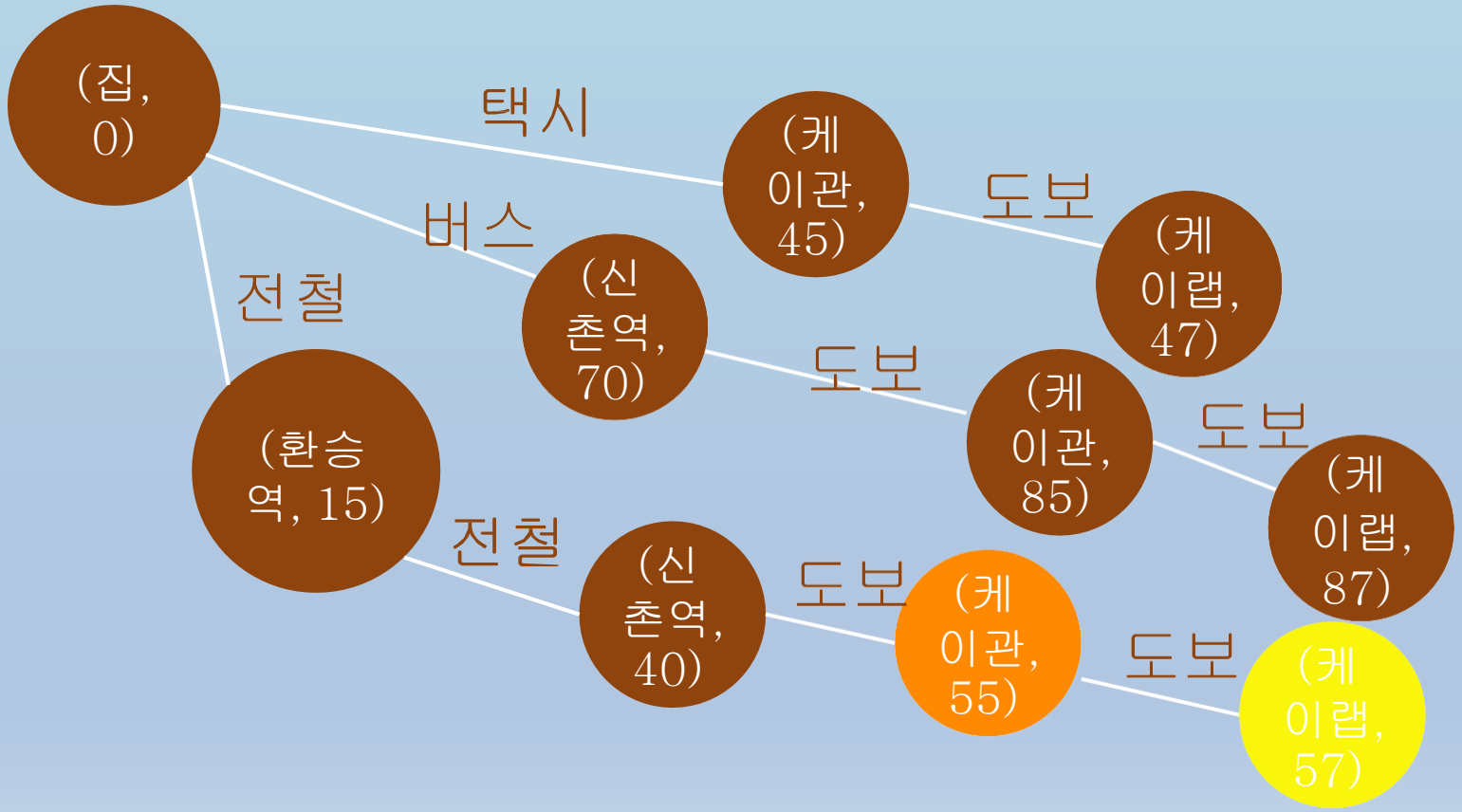
BFS

주황색 : 현재 상태,
노란색 : 큐에 쌓인 상태.



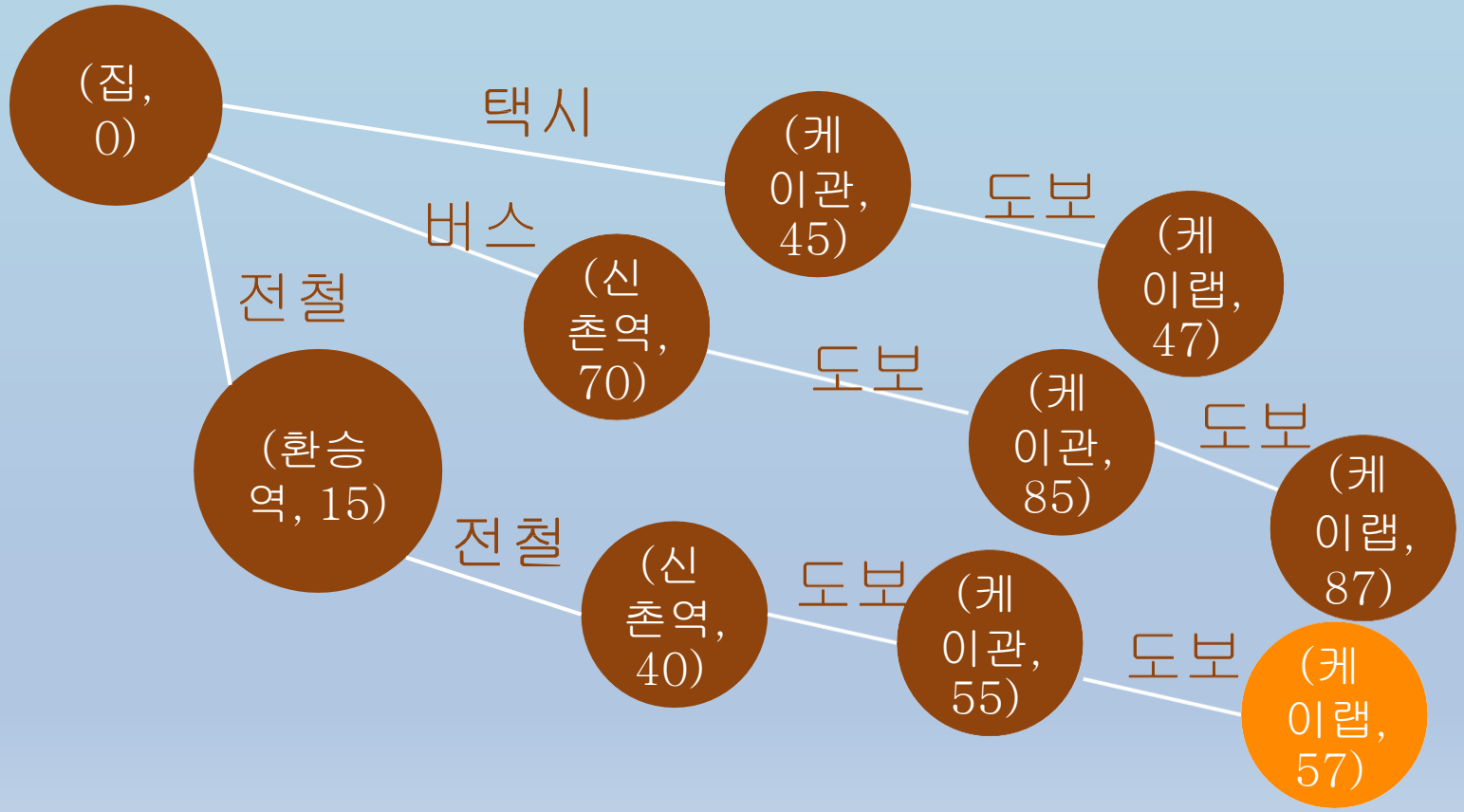
BFS

주황색 : 현재 상태,
노란색 : 큐에 쌓인 상태.



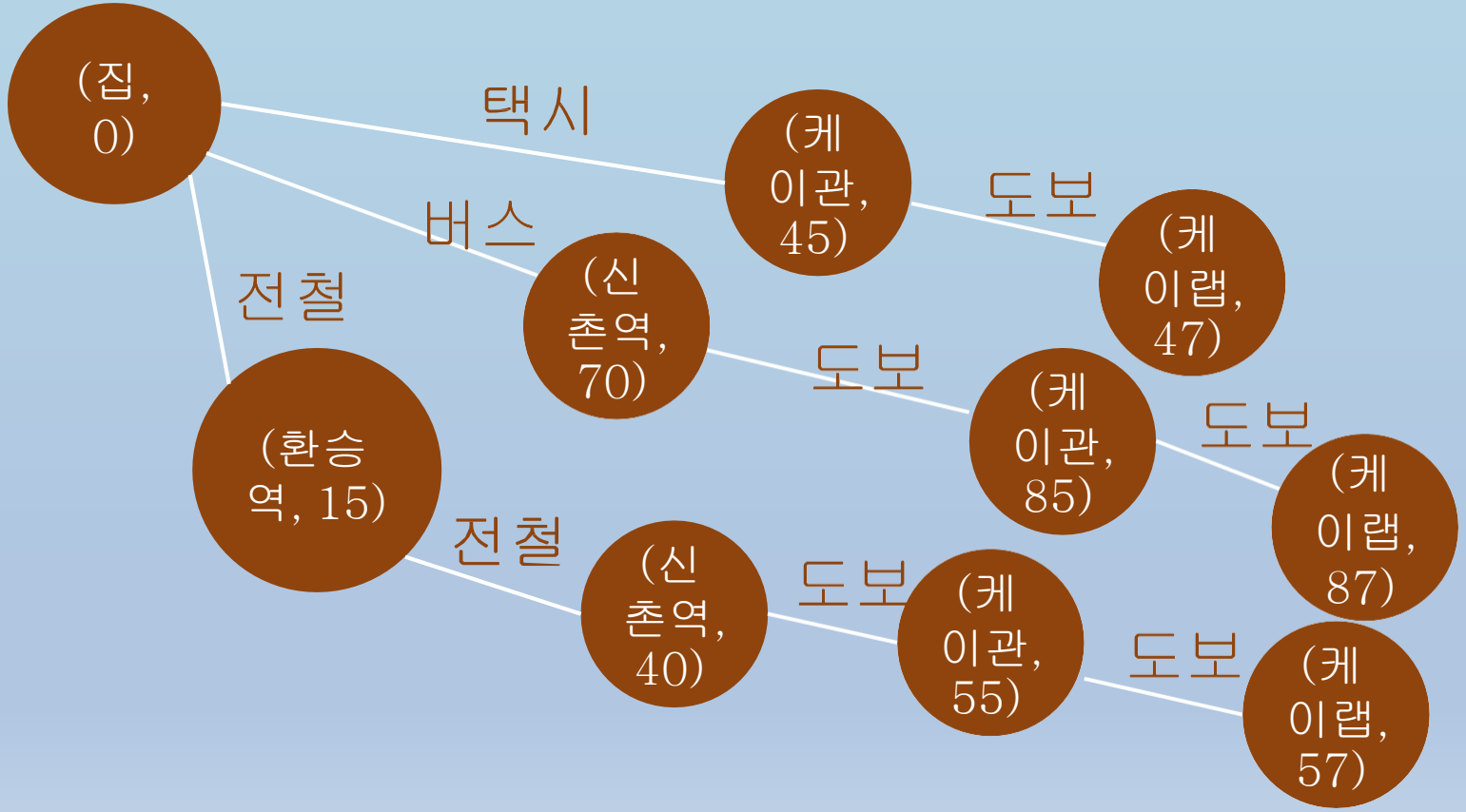
BFS

주황색 : 현재 상태,
노란색 : 큐에 쌓인 상태.



BFS

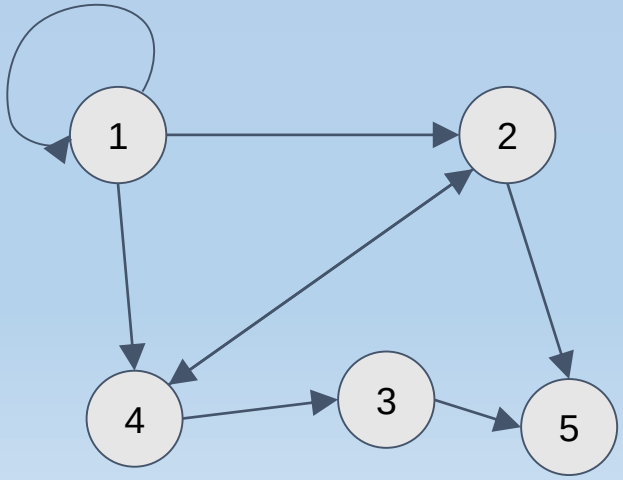
주황색 : 현재 상태,
노란색 : 큐에 쌓인 상태.



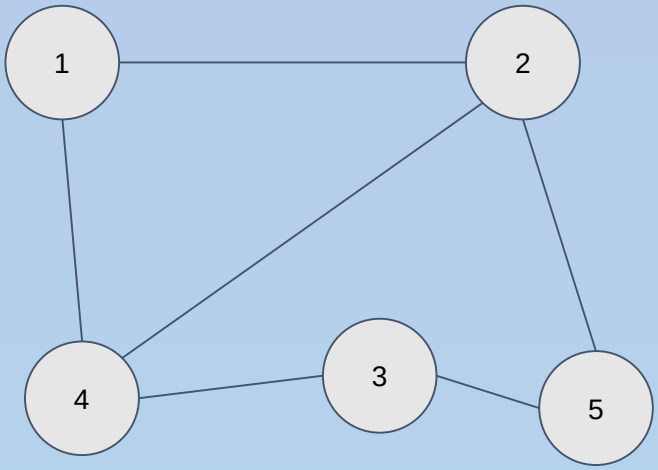
Implementation

Graph 표현

- 1. 인접 행렬 (Adjacency Matrix)
- 2. 인접 리스트 (Adjacency List)

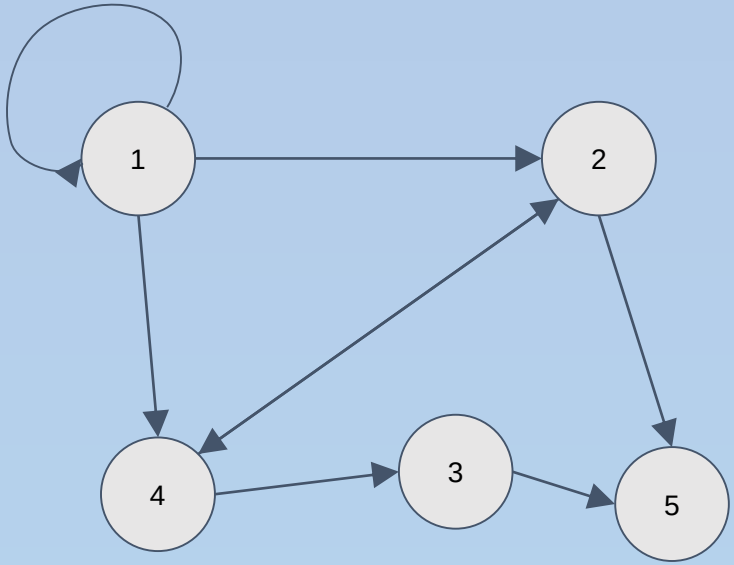


Graph의 표현 - (1) 인접 행렬



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	1	1
3	0	0	0	1	1
4	1	1	1	0	0
5	0	1	1	0	0

Graph의 표현 - (1) 인접 행렬

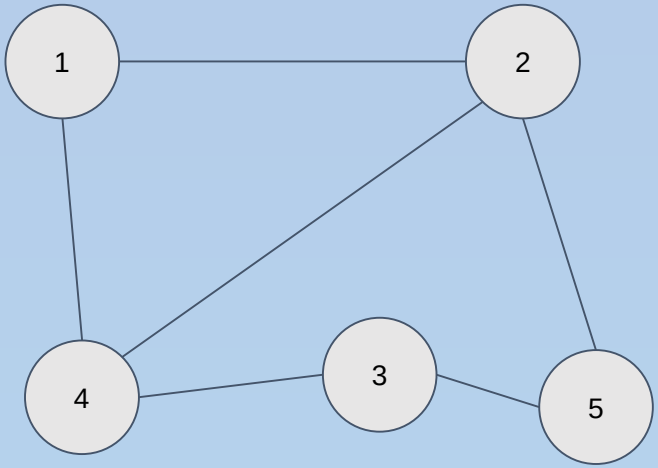


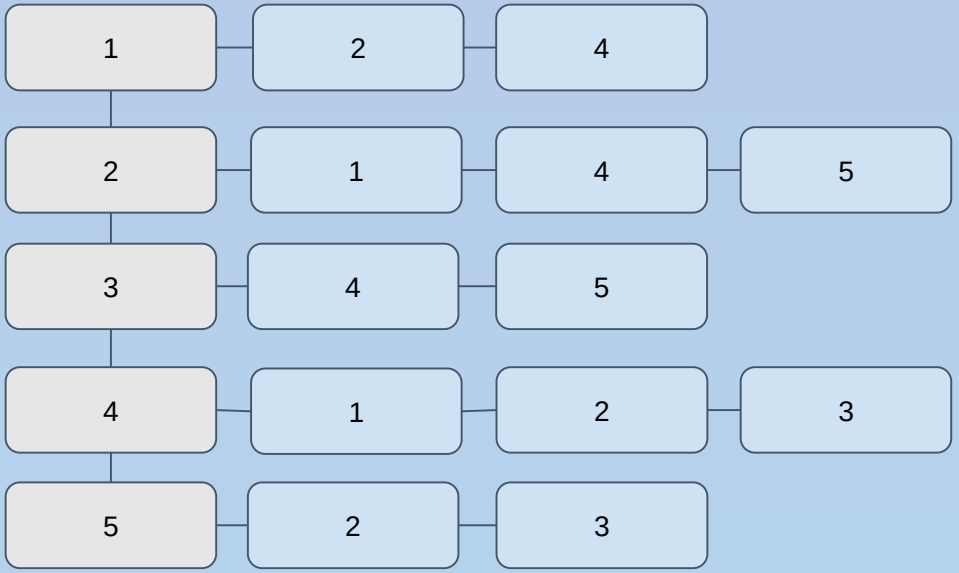
	1	2	3	4	5
1	1	1	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	0	1	1	0	0
5	0	0	0	0	0

인접 행렬

```cpp
#include <stdio.h>
#include <vector>

using namespace std;

const int N = 101; // number of vertex
int e[N][N];

int main(){
    int n, m; // number of vertex, edge
    scanf("%d %d", &n, &m);
    for(int i=0; i<m; i++){
        int a, b;
        scanf("%d %d", &a, &b); // edge a to b
        e[a][b] = 1;
        // e[b][a] = 1; 양방향 일 경우
    }
}
```

Graph의 표현 - (2) 인접 리스트





인접 리스트

```cpp
#include <stdio.h>
#include <vector>

using namespace std;

const int N = 101; // number of vertex
vector <int> e[N];

int main(){
    int n, m; // number of vertex, edge
    scanf("%d %d", &n, &m);
    for(int i=0; i<m; i++){
        int a, b;
        scanf("%d %d", &a, &b); // edge a to b
        e[a].push_back(b);
        // e[b].push_back(a); 양방향 일 경우
    }
}
```

DFS

```
int visit[N];
void dfs(int cur){
    visit[cur] = 1;
    for(int i=0; i<e[i].size(); i++){
        int next = e[cur][i];
        if( !visit[next] )
            dfs( next );
    }
}
```

main()

```
for(int i=1; i<=n; i++)
    visit[i] = 0;

for(int i=1; i<=n; i++)
    if( !visit[i] )
        dfs( i );
```

BFS

```
void bfs(int start){
    queue <int> q;
    q.push( start );
    visit[start] = 1; // 이 위치 반드시 기억 1

    while( !q.empty() ){
        int cur = q.front(); q.pop();

        for(int i=0; i<e[i].size(); i++){
            int next = e[cur][i];
            if( !visit[next] ){
                visit[next] = 1;
                q.push( next ); // 이 위치 반드시 기억 2
            }
        }
    }
}
```

main()

```
for(int i=1; i<=n; i++)
    visit[i] = 0;
for(int i=1; i<=n; i++)
    if( !visit[i] )
        bfs( i );
```

다양한 State

```cpp
#include <stdio.h>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

const int N = 101; // number of vertex

typedef pair <int, int> ii;
vector <ii> e[N];

int main(){
    int n, m; // number of vertex, edge
    scanf("%d %d", &n, &m);

    for(int i=0; i<m; i++){
        int a, b, cost;
        scanf("%d %d %d", &a, &b, &cost); // edge a to b
        e[a].push_back(ii(b, cost));
        // e[b].push_back(a); 양방향 일 경우
    }
}
```

```cpp
#include <stdio.h>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

const int N = 101; // number of vertex

typedef pair <int, string> is;
vector <is> e[N];

int main(){
    int n, m; // number of vertex, edge
    scanf("%d %d", &n, &m);
    char buf[100];
    for(int i=0; i<m; i++){
        int a, b, cost;
        scanf("%d %d %s", &a, &b, buf); // edge a to b
        e[a].push_back(ii(b, buf));
        // e[b].push_back(a); 양방향 일 경우
    }
}
```

반드시 관계를 저장할 필요는 없음

```
void dfs(int cur, int depth, int cost){
    if( ~~ ) // 조건 1
        dfs( next1, depth1, cost1 );
    if( ~~ ) // 조건 2
        dfs( next2, depth2, cost2 );
    if( ~~ ) // 조건 3
        dfs( next3, depth3, cost3 );
}
```

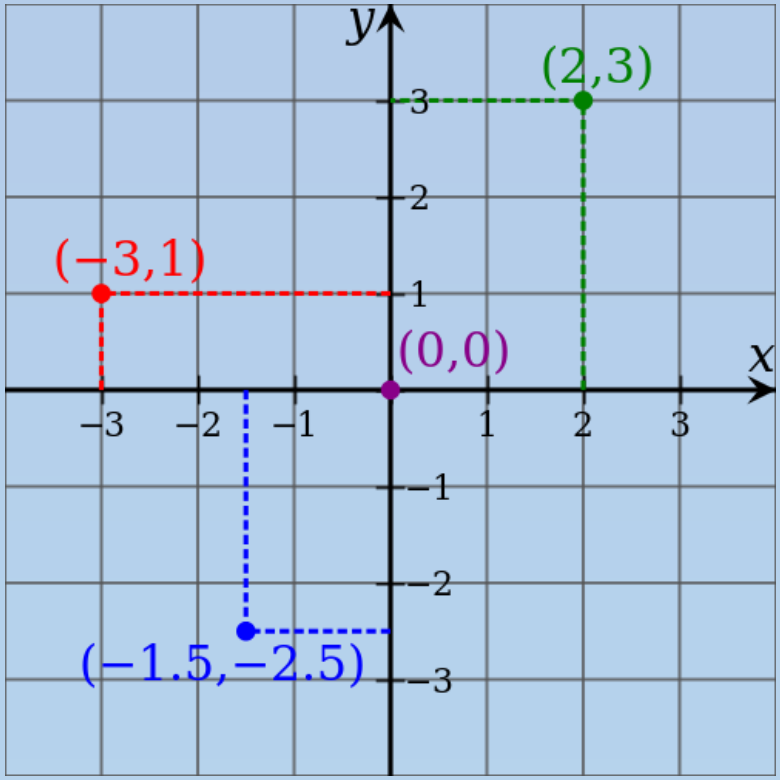
Recursive를 이용하여
모든 경우의 수 탐색
(Backtracking)

반드시 관계를 저장할 필요는 없음

```
void dfs(int cur, int depth, int cost){
    if( ~~ ) // 조건 1
        dfs( next1, depth1, cost1 );
    if( ~~ ) // 조건 2
        dfs( next2, depth2, cost2 );
    if( ~~ ) // 조건 3
        dfs( next3, depth3, cost3 );
}
```

Recursive를 이용하여
모든 경우의 수 탐색
(Backtracking)

반드시 관계를 저장할 필요는 없음




```
int dx[] = {1, -1, 0, 0};
int dy[] = {0, 0, 1, -1};

void dfs(int x, int y){
    v[x][y] = 1;
    for(int k=0; k<4; k++){
        int nx = x + dx[k];
        int ny = y + dy[k];

        // 좌표 범위가 넘어가는지와 방문했는지
        if( check(nx, ny) && !v[nx][ny] ){
            v[nx][ny] = 1;
            dfs( nx, ny );
        }
    }
}
```

Q&A