안녕하세요, 동계 대학생 S/W 알고리즘 특강의 다섯번째 시간인 오늘은 그래프 탐색에 대해 다루어보도록 하겠습니다.

1. 기초 강의

동영상 강의 컨텐츠 확인 > 4. 그래프

Link : https://swexpertacademy.com/main/learn/course/subjectDetail.do?courseId=CONTENTS_REVIEW&subjectId=AYVXaW_aQSYDFARs

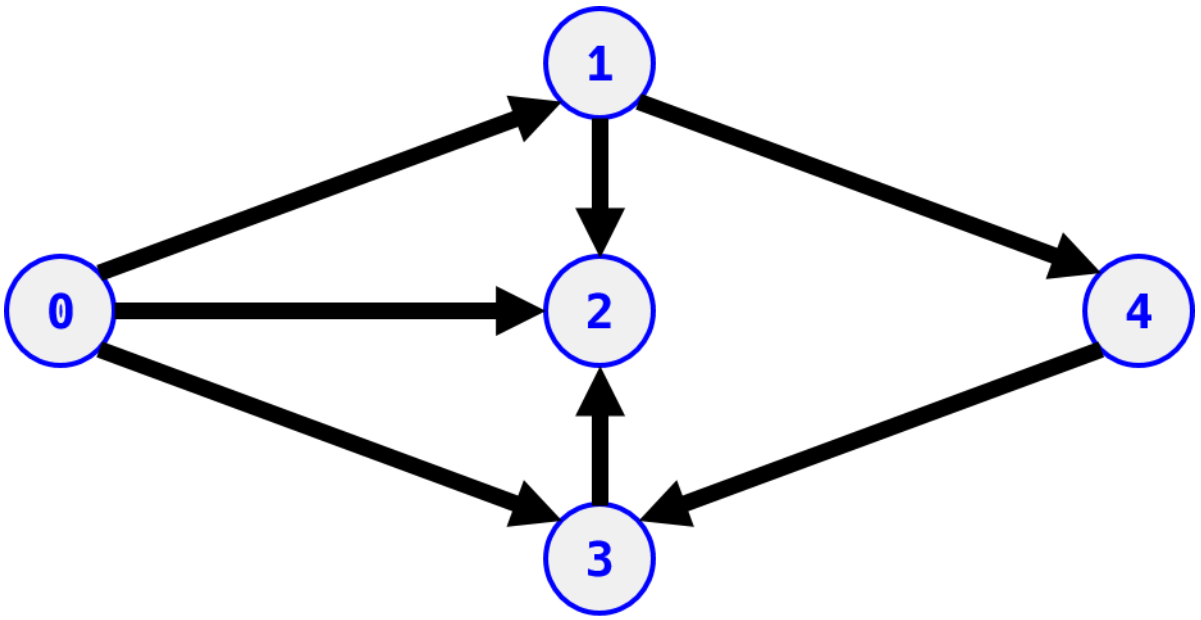
※ 출석은 강의 수강 내역으로 확인합니다.

간단한 구현 연습 문제가 준비되어 있습니다. 동계 대학생 S/W 알고리즘 특강 - 기본 문제에서 아래 2문제를 풀어주세요.

1. 기초 DFS 연습
2. 기초 BFS 연습

2. 그래프 구현

그래프를 표현하는 방법은 인접 행렬과 인접 리스트가 있습니다. 여기서는 아래의 그래프를 다양한 방법의 인접 리스트로 구현해보고 각자의 장단점을 알아보겠습니다.



정점의 개수는 n, 간선의 정보는 단방향간선을 std::pair로 담은 std::vector로 주어진다고 하겠습니다.

```
constexpr int n = 5;
const std::vector<std::pair<int, int>> edges = {{0, 1}, {0, 2}, {0, 3}, {1, 2}, {1, 4}, {3, 2}, {4, 3}};
```

2.1 2차원 std::vector

구현량이 적어서 가장 많이 쓰이는 방식입니다. 어떤 정점과 인접한 모든 정점을 std::vector같은 동적 배열에 저장합니다.

```
std::vector<std::vector<int>> graph(n);
for (const auto& e : edges) {
    graph[e.first].emplace_back(e.second);
}
```

u	graph[u]
0	1 2 3
1	2 4
2	
3	2
4	3

◆ 장점

구현이 매우 쉽습니다.

간선으로 이어진 정점 정보가 하나의 배열에 담기므로, 배열에서 할 수 있는 모든 작업(정렬, 삭제 등)을 자유롭게 할 수 있습니다.

◆ 단점

std::vector에 정점을 push하는 데 오버헤드가 생깁니다.

여러 개의 테스트케이스를 처리할 경우, 메모리에 빈 공간이 많이 생깁니다.

메모리상에 각각의 배열이 파편화되어 저장되므로 캐시 효율이 떨어집니다.

2.2 연결 리스트

std::vector를 대신하는 방법입니다. 정점마다 인접한 정점의 개수는 미리 알지 못하지만, 총 간선의 개수는 알고 있으므로 사용할 메모리만 정확하게 미리 할당할 수 있습니다.

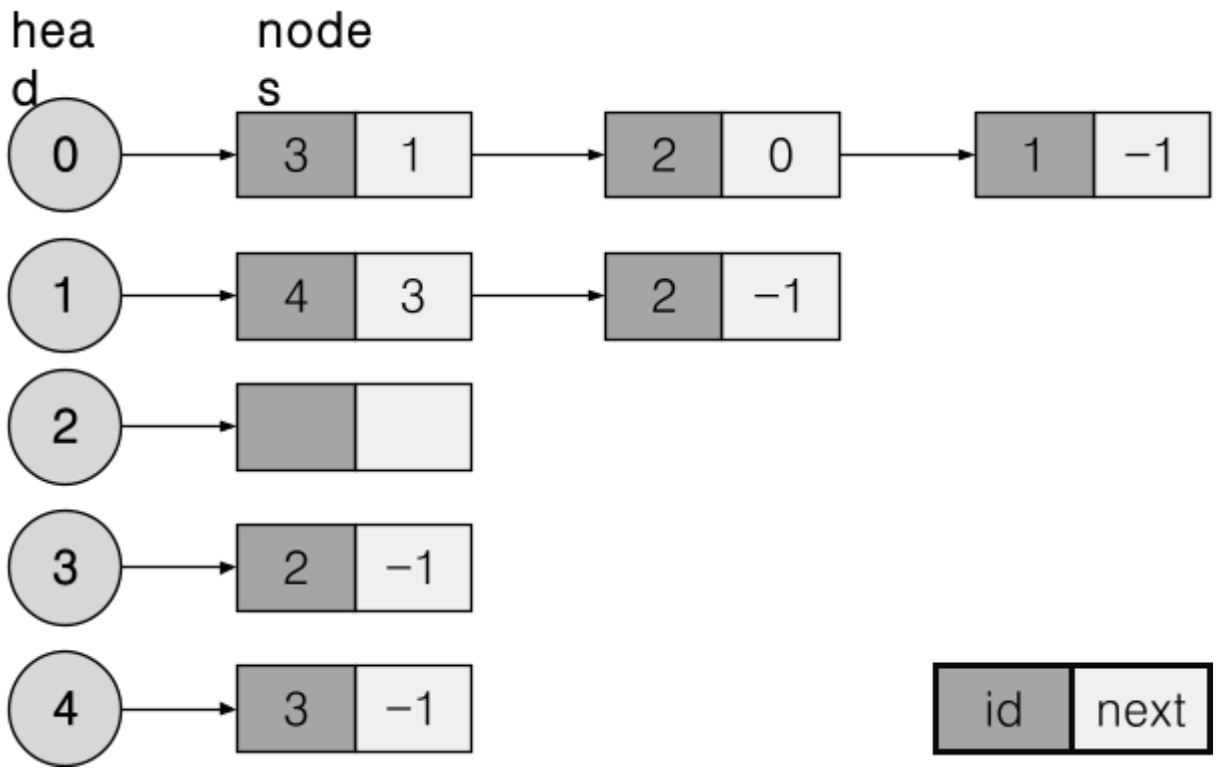
```
struct LinkedListNode {
	int id;
	int next;
};
std::vector<LinkedListNode> nodes(edges.size());
std::vector<int> head(n, -1);
for (int i = 0; i < static_cast<int>(edges.size()); ++i) {
	nodes[i].id = edges[i].second;
	nodes[i].next = head[edges[i].first];
	head[edges[i].first] = i;
}
```

nodes :

i	0	1	2	3	4	5	6
id	1	2	3	2	4	2	3
next	-1	0	1	-1	3	-1	-1

head :

i	0	1	2	3	4
head[i]	2	4	-1	5	6



◆ 장점

메모리를 효율적으로 사용합니다. 특히 여러 개의 테스트케이스를 처리할 때 차이가 큽니다.

◆ 단점

가독성이 떨어집니다.

정점 정보를 연결 리스트에 저장하므로 정렬하거나 삭제하는 데 어려움이 있습니다.

연결 리스트 순회는 기본적으로 배열보다 느립니다.

2.3 1차원 배열

2차원으로 놓인 여러 개의 정점 리스트를 일렬로 잇는 방법입니다. 정점마다 인접한 정점의 개수를 미리 전처리로 구해야 하므로, 실시간으로 간선이 추가/삭제되는 문제를 풀 수 없습니다.

```
std::vector<int> outdegree(n);

std::vector<int> prefix(n + 1);

std::vector<int> vertices(edges.size());

for (const auto& e : edges) {
```

```
        ++outdegree[e.first];
}
std::partial_sum(outdegree.begin(), outdegree.end(), std::next(prefix.begin()));
for (const auto& e : edges) {
        vertices[prefix[e.first] + --outdegree[e.first]] = e.second;
}
```

	0	1	2	3	4
out	3	2	0	1	1

	0	1	2	3	4	
pref	0	3	5	5	6	7

	0	1	2	3	4	5	6
vtx	3	2	1	4	2	2	3

◆ 장점

캐시 효율이 가장 뛰어납니다.

◆ 단점

가독성이 떨어집니다.

크기가 다른 세 개의 독립된 배열을 사용하므로 실수할 여지가 많습니다.

실시간으로 간선을 추가/삭제 할 수 없습니다.

2.4 Test Code

```
#include <iostream>
#include <numeric>
```

```
#include <utility>
#include <vector>

void f0(int n, std::vector<std::pair<int, int>> edges) {
	// data
	std::vector<std::vector<int>> graph(n);
	// build
	for (const auto& e : edges) {
		graph[e.first].emplace_back(e.second);
	}
	// iterate
	std::cout << "using vector\n";
	for (int u = 0; u < n; ++u) {
		std::cout << u << " ->";
		for (const auto& v : graph[u]) {
			std::cout << ' ' << v;
		}
		std::cout << '\n';
	}
	std::cout << '\n';
}

void f1(int n, std::vector<std::pair<int, int>> edges) {
	// data
	struct LinkedListNode {
		int id;
		int next;
```

```
        };
        std::vector<LinkedListNode> nodes(edges.size());
        std::vector<int> head(n, -1);
        // build
        for (int i = 0; i < static_cast<int>(edges.size()); ++i) {
                nodes[i].id = edges[i].second;
                nodes[i].next = head[edges[i].first];
                head[edges[i].first] = i;
        }
        // iterate
        std::cout << "using linked list\n";
        for (int u = 0; u < n; ++u) {
                std::cout << u << " ->";
                for (int i = head[u]; i != -1; i = nodes[i].next) {
                        std::cout << ' ' << nodes[i].id;
                }
                std::cout << '\n';
        }
        std::cout << '\n';
}

void f2(int n, std::vector<std::pair<int, int>> edges) {
        // data
        std::vector<int> outdegree(n);
        std::vector<int> prefix(n + 1);
        std::vector<int> vertices(edges.size());
        // build
```

```
	for (const auto& e : edges) {
		++outdegree[e.first];
	}
	std::partial_sum(outdegree.begin(), outdegree.end(), std::next(prefix.begin()));
	for (const auto& e : edges) {
		vertices[prefix[e.first] + --outdegree[e.first]] = e.second;
	}
	// iterate
	std::cout << "using prefix sum\n";
	for (int u = 0; u < n; ++u) {
		std::cout << u << " ->";
		for (int i = prefix[u]; i < prefix[u + 1]; ++i) {
			std::cout << ' ' << vertices[i];
		}
		std::cout << '\n';
	}
	std::cout << '\n';
}

int main() {
	constexpr int n = 5;
	const std::vector<std::pair<int, int>> edges =
		{{0, 1}, {0, 2}, {0, 3}, {1, 2}, {1, 4}, {3, 2}, {4, 3}};

	f0(n, edges);
	f1(n, edges);
	f2(n, edges);
```

}

3. 기본 문제

그래프 탐색 기본 문제입니다.

- 프로세서 연결하기
- 파핑파핑 지뢰 찾기
- 영준이의 진짜 BFS
- 하나로
- 고속도로 건설 2