SWIFT
Data Structure - Graph

Bill Kim(김정훈) | ibillkim@gmail.com

목차

Graph

그래프(Graph)는 비선형(Non-Linear) 자료구조로서 노드(Node)와 간선(Edge)로 구성된 형태를 가진 네트워크 모델의 자료구조입니다.

객체(Node) 간의 관계를 선(Edge)으로 연결하여 표현하는 형태로서 각 정점 간에는 부모-자식 관계가 없습니다.

간선에 방향과 가중치를 두어서 다양한 분야에서 활용가능한 자료구조 중 하나입니다.

최단 경로 찾기 및 저렴한 항공편 검색 등 다양한 분야에서 그래프를 활용할 수 있습니다.

본 강의에서는 간략한 그래프의 용어와 개념에 대해서 설명을 합니다. 좀 더 깊은 개념 및 그래프 탐색 등의 내용은 다른 강의에서 별도로 설명할 예정입니다.

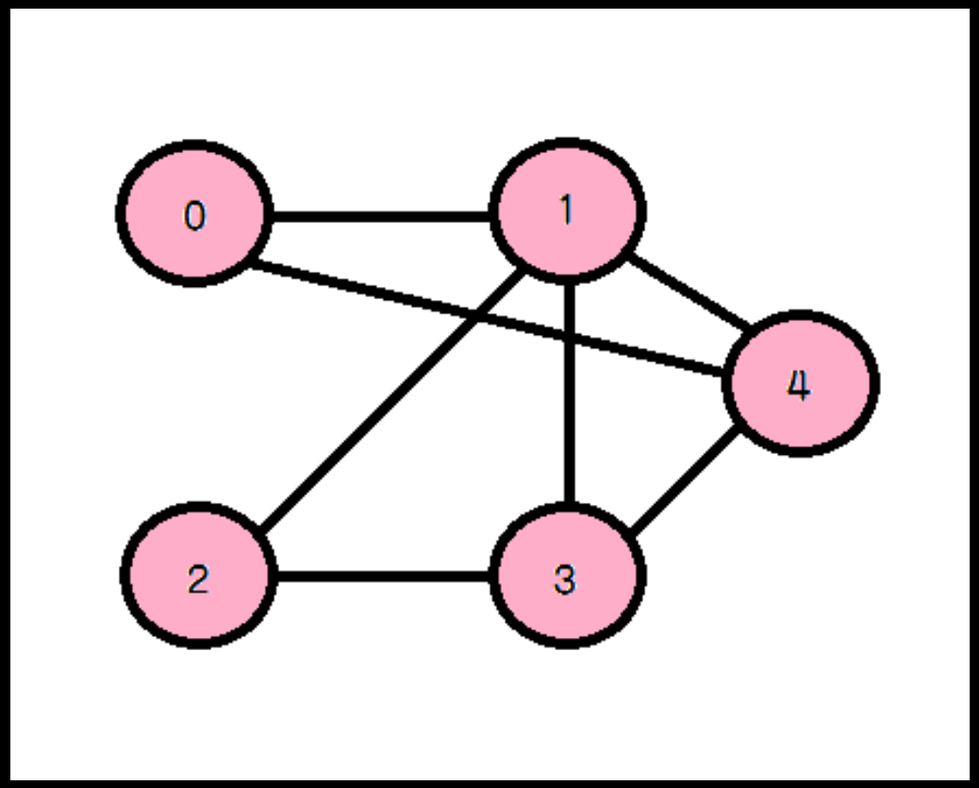
Graph

그래프(Graph)에서 사용하는 기본적인 용어들은 아래와 같습니다.

1. Vertex(Node, 정점) : 정점으로서 데이터가 저장되는 기본 객체
2. Edge(간선) : 간선으로서 정점을 연결하는 선
3. Adjacent(인접) : 한 정점에서 간선을 한번에 갈 수 있으면 해당 정점들은 인접하다고 할 수 있습니다.
4. Degree(차수) : 한 정점이 가지고 있는 간선(Edge)의 수
5. 진입 차수(In-degree) : 방향 그래프에서 외부에서 오는 간선의 수
6. 진출 차수(Out-degree) : 방향 그래프에서 외부로 향하는 간선의 수
7. 경로 길이(Path Length) : 경로를 구성하는데 사용된 간선의 수
8. 단순 경로(Simple Path) : 경로 중에서 반복되는 정점이 없는 경우
9. 사이클(Cycle) : 단순 경로의 시작 정점과 종료 정점이 동일한 경우

Graph

앞서 설명한 용어들을 예제로 살펴보면 아래와 같습니다.



V(Vertex) = { 0, 1, 2, 3, 4 }
E(Edges) = { 01, 12, 23, 34, 04, 14, 13 }
1 노드의 Degree = 4
3 노드의 인접 노드 = 1, 2, 4

Graph

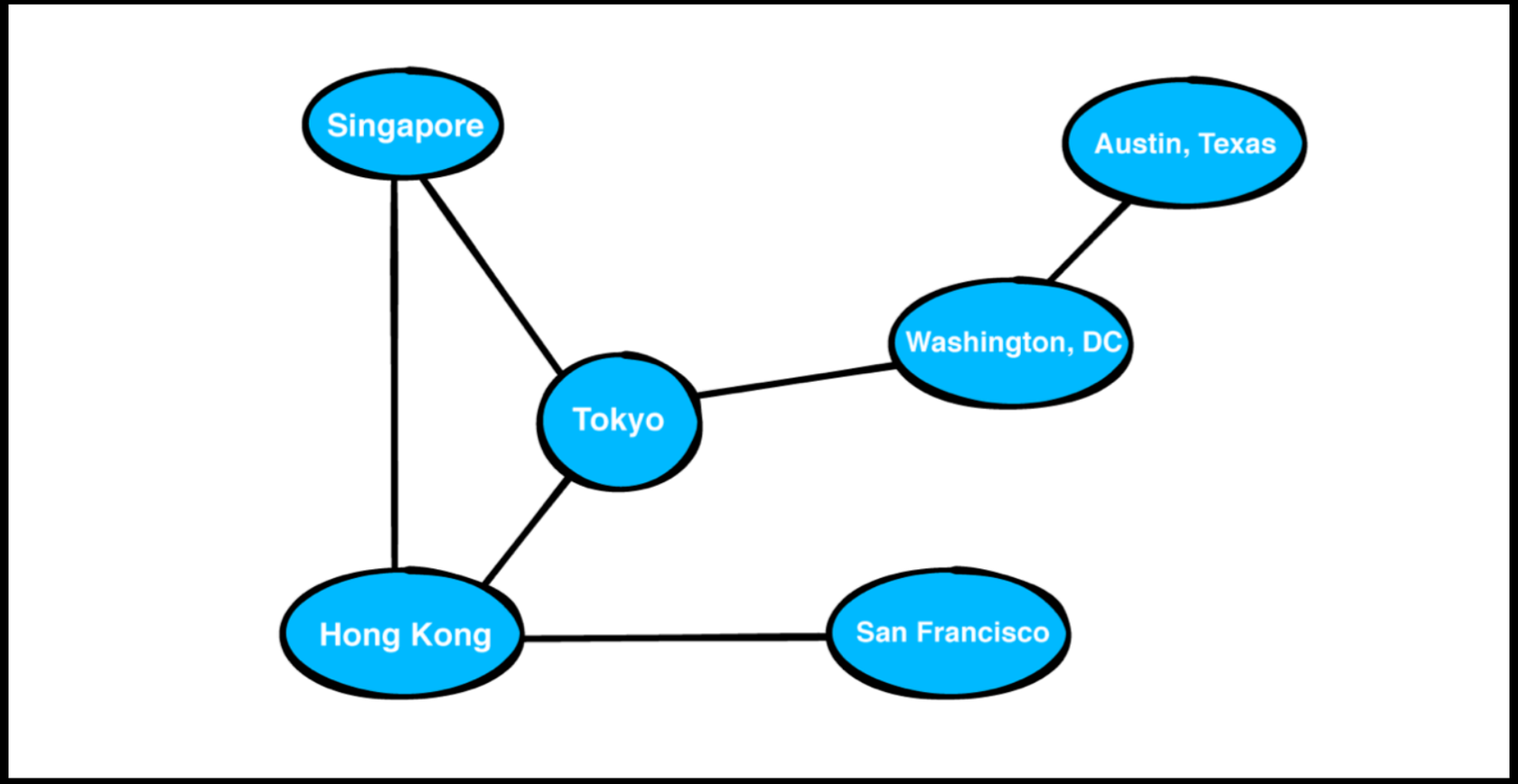
그래프(Graph)와 트리(Tree) 의 차이점을 살펴보면 아래와 같습니다.

	그래프	트리
정의	노드(node)와 그 노드를 연결하는 간선(edge)을 하나로 모아 놓은 자료 구조	그래프의 한 종류 DAG(Directed Acyclic Graph, 방향성이 있는 비순환 그래프)의 한 종류
방향성	방향 그래프(Directed), 무방향 그래프(Undirected) 모두 존재	방향 그래프(Directed Graph)
사이클	사이클(Cycle) 가능, 자체 간선(self-loop)도 가능, 순환 그래프(Cyclic), 비순환 그래프(Acyclic) 모두 존재	사이클(Cycle) 불가능, 자체 간선(self-loop)도 불가능, 비순환 그래프(Acyclic Graph)
루트 노드	루트 노드의 개념이 없음	한 개의 루트 노드만이 존재, 모든 자식 노드는 한 개의 부모 노드 만을 가짐
부모-자식	부모-자식의 개념이 없음	부모-자식 관계 top-bottom 또는 bottom-top으로 이루어짐
모델	네트워크 모델	계층 모델
순회	DFS, BFS	DFS, BFS안의 Pre-, In-, Post-order
간선의 수	그래프에 따라 간선의 수가 다름, 간선이 없을 수도 있음	노드가 N인 트리는 항상 N-1의 간선을 가짐
경로	-	임의의 두 노드 간의 경로는 유일
예시 및 종류	지도, 지하철 노선도의 최단 경로, 전기 회로의 소자들, 도로(교차점과 일방 통행길), 선수 과목	이진 트리, 이진 탐색 트리, 균형 트리(AVL 트리, red-black 트리), 이진 힙(최대힙, 최소힙) 등

Types

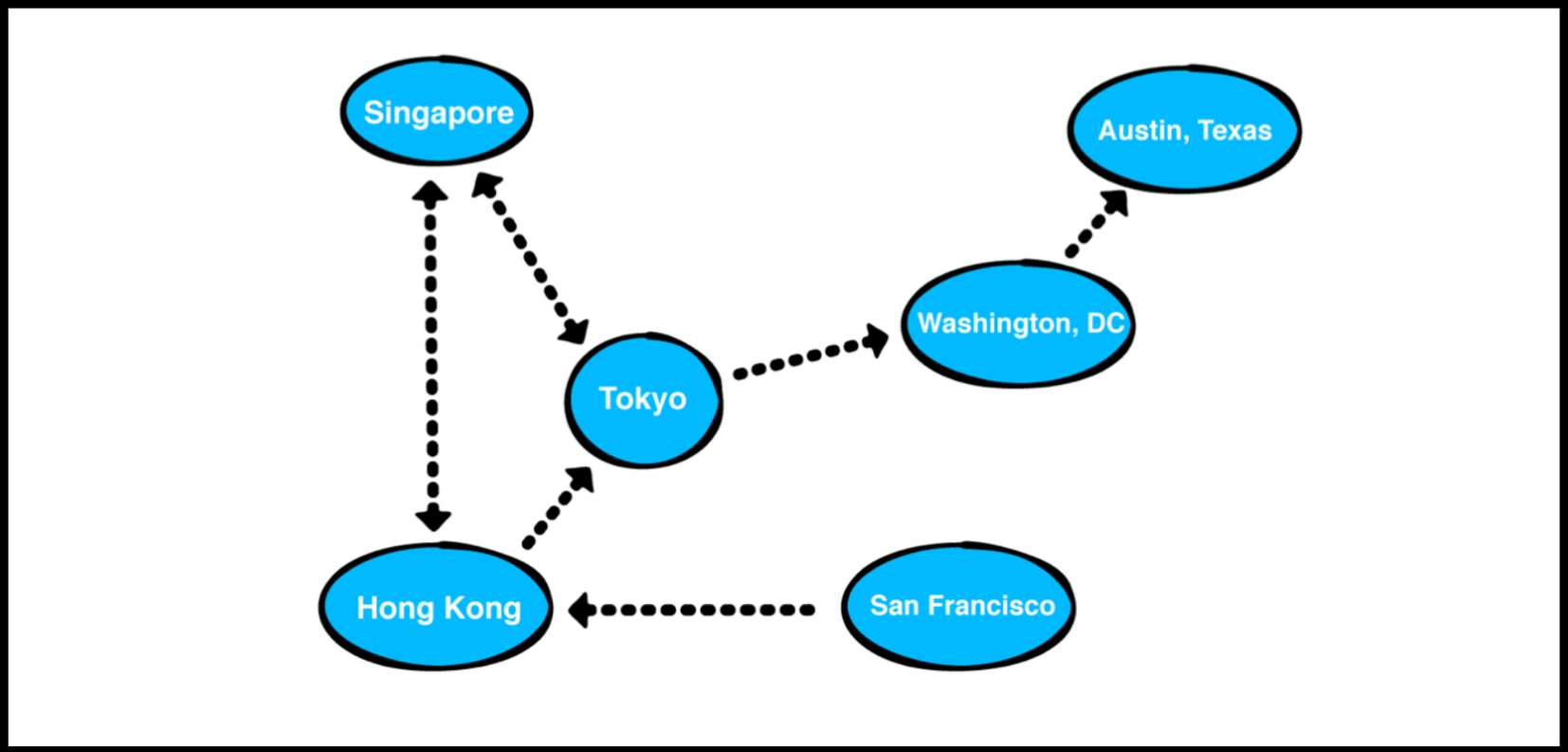
그래프(Graph)의 종류는 크게 다음과 같습니다.

무방향 그래프 : 모든 간선이 양방향인 그래프, 가중치는 양방향 모두 적용



Types

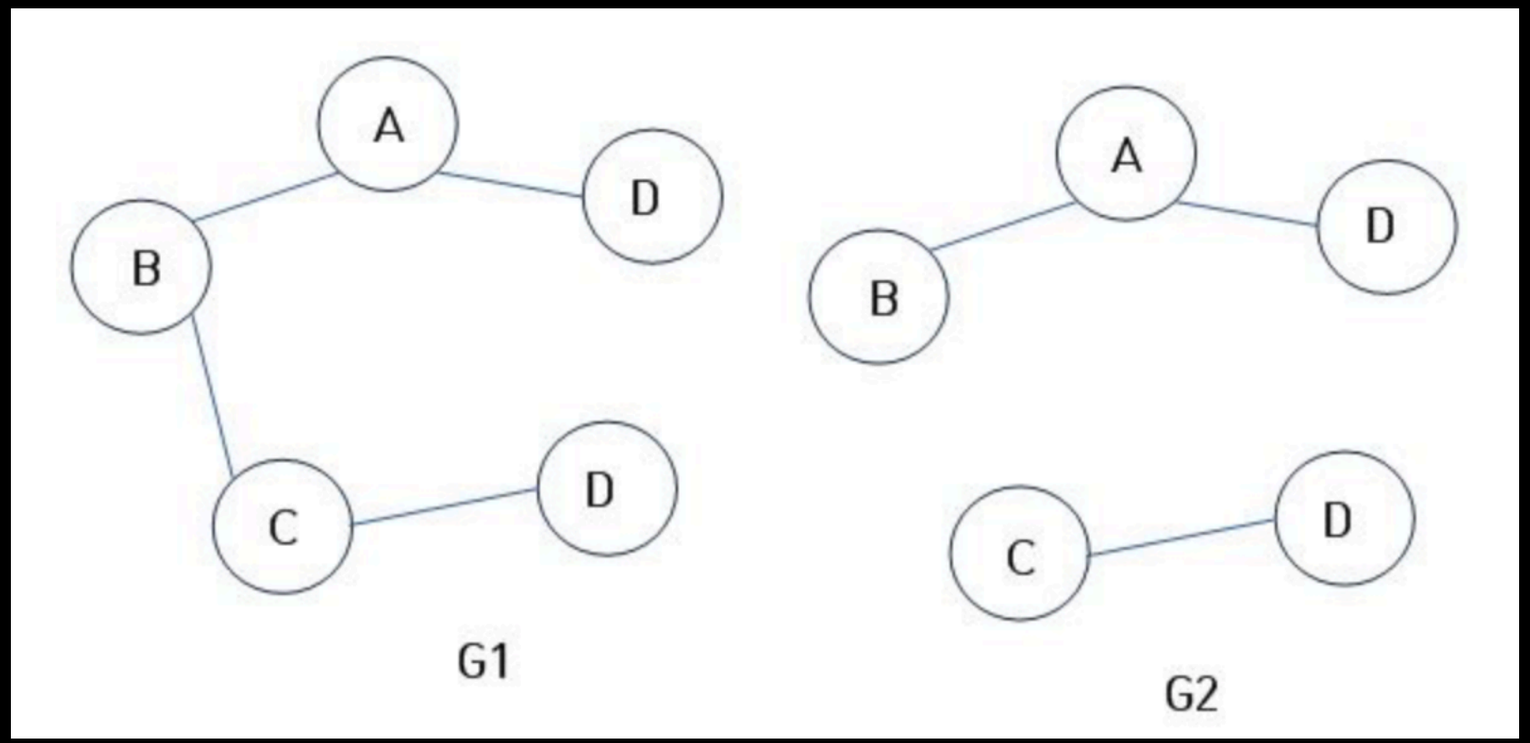
방향 그래프 : 간선들이 방향을 가진 그래프, 방향은 단방향, 양방향이 가능하며 탐색 시 정해진 방향으로만 탐색



Types

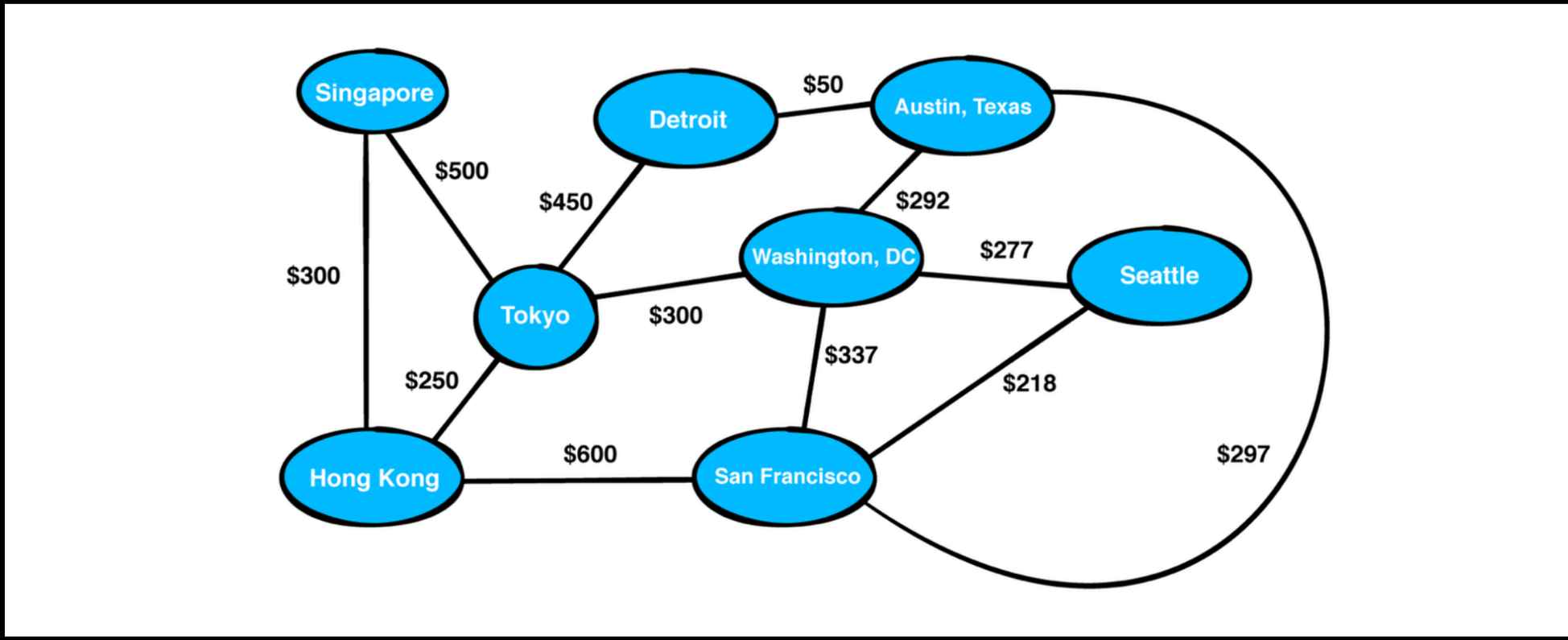
연결 그래프 : 무방향 그래프에서 모든 정점 쌍에 대해서 경로가 존재하는 그래프

비연결 그래프 : 무방향 그래프에서 특정 정점 쌍에 사이에 경로가 존재하지 않는 그래프



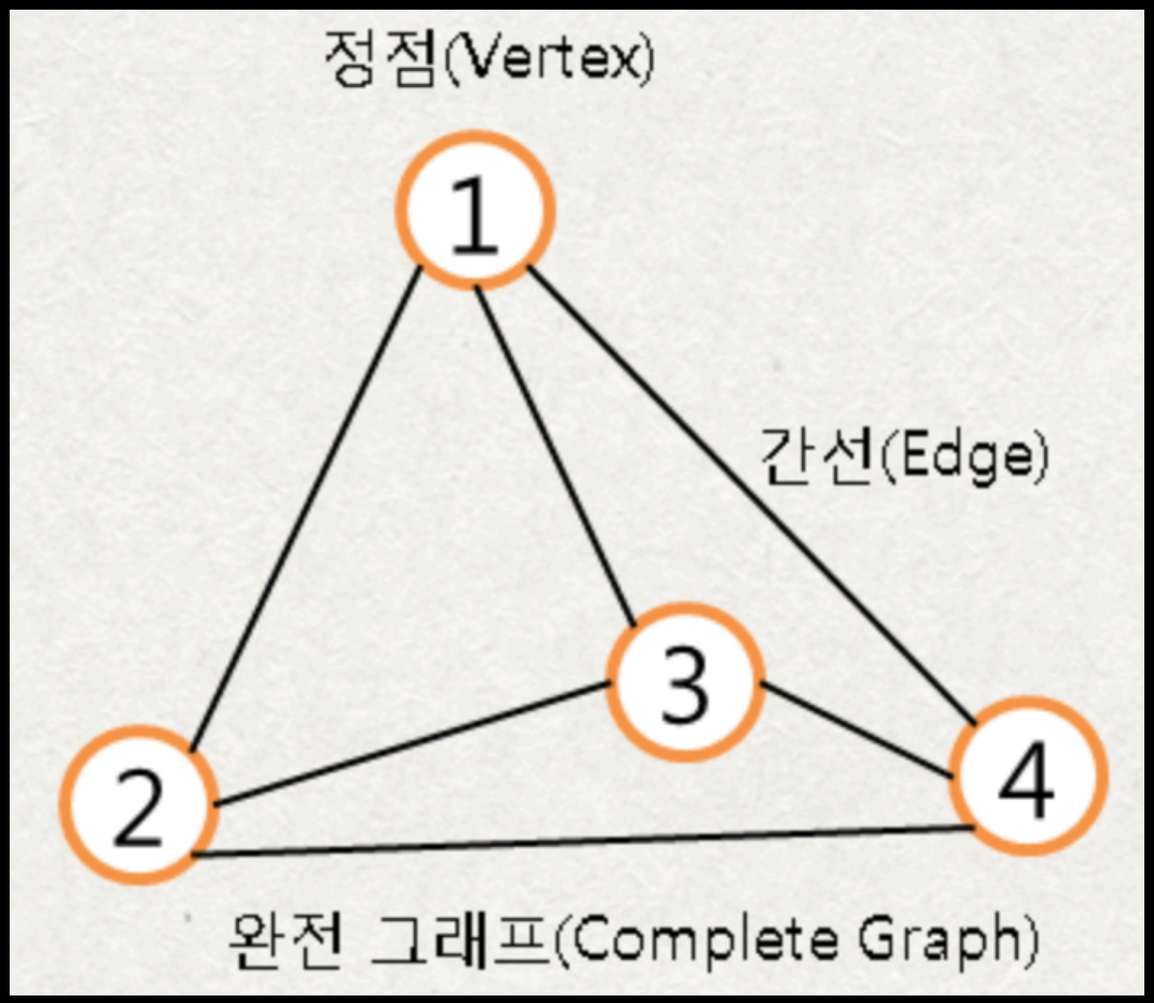
Types

가중치 그래프 : 간선들이 각각 가중치를 가진 그래프, 가중치는 다양한 기준을 활용하여 설정이 가능하며 가중치에 따라서 원하는 경로로 순회가 가능합니다.



Types

완전 그래프 : 모든 정점들이 인접한 상태인 그래프, n개의 정점이 있다면 모든 정점이 n-1개의 간선을 가지는 그래프



Representation

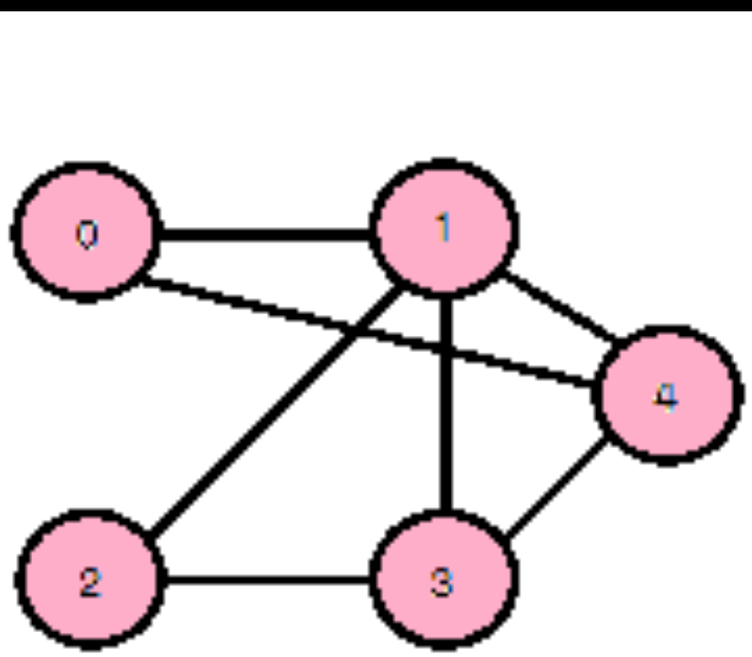
Graph를 코드로 구현(표현)하는 방법으로는 크게 두가지 방식이 있습니다.

- 인접 행렬(AdjacencyMatrix) :
2차원 배열로 정점과 간선을 표현하는 방식

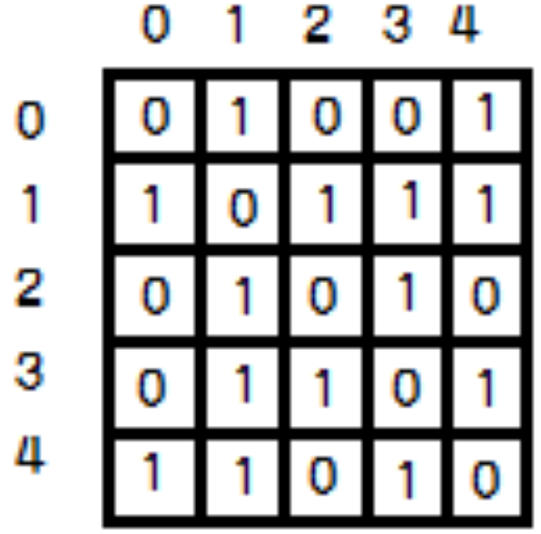
- 인접 리스트(AdjacencyList) :
각 정점(Vertex)의 리스트는 헤더 노드를 가지고 헤더 노드에 계속 정점을 연결하는 방식

Representation

인접 행렬(AdjacencyMatrix)



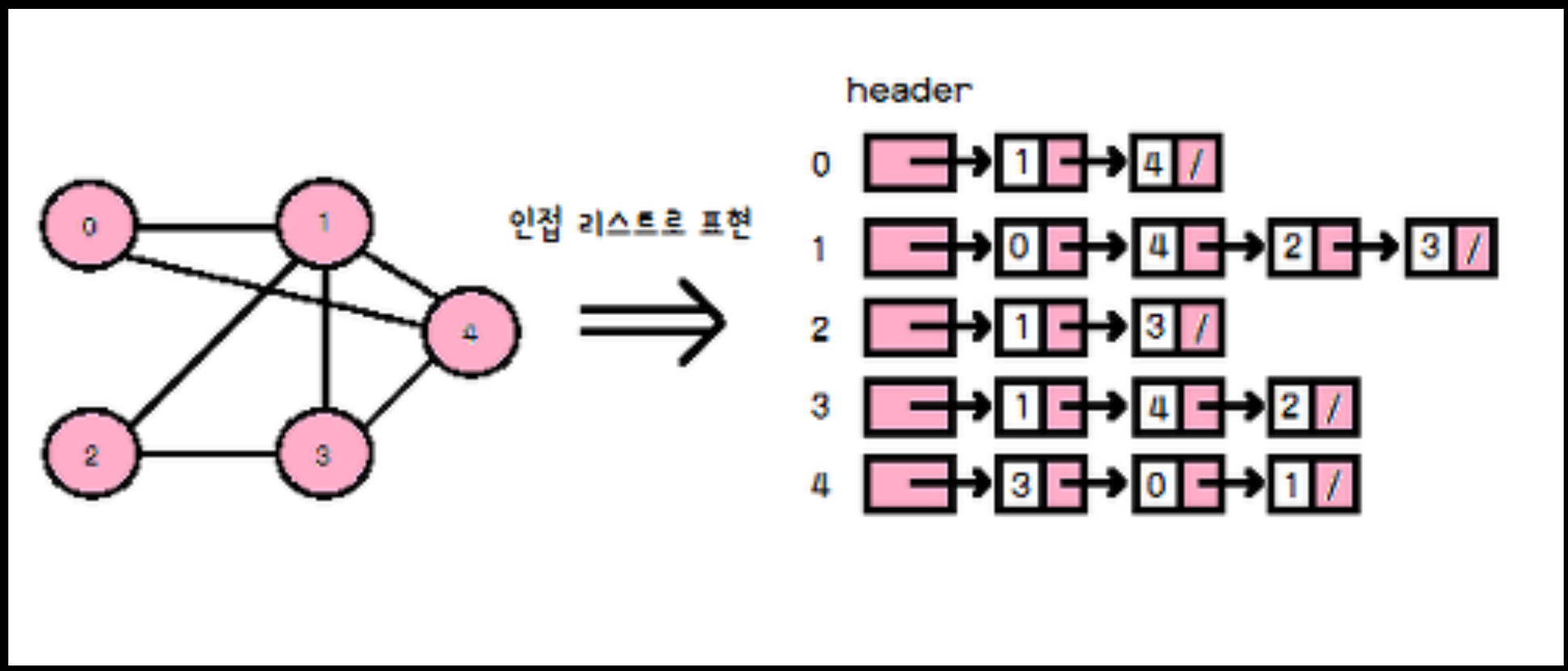
인접 행렬로 표현



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

Representation

인접 리스트(AdjacencyList)



Implementation

Swift를 활용하여 가장 기본적인 Graph를 구현해보겠습니다.
우선 필요한 객체와 메소드는 아래와 같습니다.

필요한 객체

- 정점(Vertex) 객체
- 간선(Edge) 객체
- 그래프(Graph) 추상 객체
- 인접 행렬 및 인접 리스트 객체

그래프 기본 메소드

- createVertex : 정점을 추가하는 함수
- addDirectedEdge : 방향을 가진 간선을 추가하는 함수
- addUndirectedEdge : 방향이 없는 간선을 추가하는 함수
- weightFrom : 간선 사이에 가중치를 추가하는 함수
- edgeFrom : 간선 리스트를 가져오는 함수

Implementation

```
public struct Vertex<T>: Equatable where T: Hashable {
    public var data: T
    public let index: Int
}

extension Vertex: CustomStringConvertible {
    public var description: String {
        return "\(index): \(data)"
    }
}

extension Vertex: Hashable {
    public func hash(into hasher: inout Hasher) {
        hasher.combine(data)
        hasher.combine(index)
    }
}
```

Implementation

```
public struct Edge<T>: Equatable where T: Hashable {
    public let from: Vertex<T>
    public let to: Vertex<T>

    public let weight: Double?
}

extension Edge: CustomStringConvertible {
    public var description: String {
        guard let unwrappedWeight = weight else {
          return "\(from.description) -> \(to.description)"
        }

        return "\(from.description) -(\(unwrappedWeight))-> \(to.description)"
    }
}

extension Edge: Hashable {
    public func hash(into hasher: inout Hasher) {
        hasher.combine(from.description)
        hasher.combine(to.description)
        hasher.combine(weight)
    }
}
```

Implementation

```
public func ==<T>(lhs: Vertex<T>, rhs: Vertex<T>) -> Bool {
    guard lhs.index == rhs.index else {
        return false
    }

    guard lhs.data == rhs.data else {
        return false
    }

    return true
}

public func == <T>(lhs: Edge<T>, rhs: Edge<T>) -> Bool {
    guard lhs.from == rhs.from else {
        return false
    }

    guard lhs.to == rhs.to else {
        return false
    }

    guard lhs.weight == rhs.weight else {
        return false
    }

    return true
}
```

Implementation

```
open class AbstractGraph<T>: CustomStringConvertible where T: Hashable {
    public required init() {}

    public required init(fromGraph graph: AbstractGraph<T>) {
        for edge in graph.edges {
            let from = createVertex(edge.from.data)
            let to = createVertex(edge.to.data)

            addDirectedEdge(from, to: to, withWeight: edge.weight)
        }
    }

    open func createVertex(_ data: T) -> Vertex<T> {
        fatalError("abstract function called")
    }

    open func addDirectedEdge(_ from: Vertex<T>, to: Vertex<T>, withWeight weight:
Double?) {
        fatalError("abstract function called")
    }

    open func addUndirectedEdge(_ vertices: (Vertex<T>, Vertex<T>), withWeight weight:
Double?) {
        fatalError("abstract function called")
    }

    ....
}
```

Implementation

```
open class AbstractGraph<T>: CustomStringConvertible where T: Hashable {

    ....

    open func weightFrom(_ sourceVertex: Vertex<T>,
                         to destinationVertex: Vertex<T>) -> Double? {
        fatalError("abstract function called")
    }

    open func edgesFrom(_ sourceVertex: Vertex<T>) -> [Edge<T>] {
        fatalError("abstract function called")
    }

    open var description: String {
        fatalError("abstract property accessed")
    }

    open var vertices: [Vertex<T>] {
        fatalError("abstract property accessed")
    }

    open var edges: [Edge<T>] {
        fatalError("abstract property accessed")
    }
}
```

Implementation

```
let graphList = AdjacencyListGraph<Int>()

var v1 = graphList.createVertex(1)
var v2 = graphList.createVertex(2)
var v3 = graphList.createVertex(3)
var v4 = graphList.createVertex(4)
var v5 = graphList.createVertex(5)

graphList.addDirectedEdge(v1, to: v2, withWeight: 1.0)
graphList.addDirectedEdge(v2, to: v3, withWeight: 1.0)
graphList.addDirectedEdge(v3, to: v4, withWeight: 4.5)
graphList.addDirectedEdge(v4, to: v1, withWeight: 2.8)
graphList.addDirectedEdge(v2, to: v5, withWeight: 3.2)

// Returns the weight of the edge from v1 to v2 (1.0)
graphList.weightFrom(v1, to: v2)

// Returns the weight of the edge from v1 to v3 (nil, since there is not an edge)
graphList.weightFrom(v1, to: v3)

// Returns the weight of the edge from v3 to v4 (4.5)
graphList.weightFrom(v3, to: v4)

// Returns the weight of the edge from v4 to v1 (2.8)
graphList.weightFrom(v4, to: v1)

print(graphList)
// 1 -> [(2: 1.0)]
// 2 -> [(3: 1.0), (5: 3.2)]
// 3 -> [(4: 4.5)]
// 4 -> [(1: 2.8)]

print(graphList.edgesFrom(v2))
// [1: 2 -(1.0)-> 2: 3, 1: 2 -(3.2)-> 4: 5]
```

Implementation

```
let graphMatrix = AdjacencyMatrixGraph<Int>()

v1 = graphMatrix.createVertex(1)
v2 = graphMatrix.createVertex(2)
v3 = graphMatrix.createVertex(3)
v4 = graphMatrix.createVertex(4)
v5 = graphMatrix.createVertex(5)

graphMatrix.addDirectedEdge(v1, to: v2, withWeight: 1.0)
graphMatrix.addDirectedEdge(v2, to: v3, withWeight: 1.0)
graphMatrix.addDirectedEdge(v3, to: v4, withWeight: 4.5)
graphMatrix.addDirectedEdge(v4, to: v1, withWeight: 2.8)
graphMatrix.addDirectedEdge(v2, to: v5, withWeight: 3.2)

// Returns the weight of the edge from v1 to v2 (1.0)
graphMatrix.weightFrom(v1, to: v2)

// Returns the weight of the edge from v1 to v3 (nil, since there is not an edge)
graphMatrix.weightFrom(v1, to: v3)

// Returns the weight of the edge from v3 to v4 (4.5)
graphMatrix.weightFrom(v3, to: v4)

// Returns the weight of the edge from v4 to v1 (2.8)
graphMatrix.weightFrom(v4, to: v1)

print(graphMatrix)
//  ø   1.0   ø    ø    ø
//  ø    ø   1.0   ø   3.2
//  ø    ø    ø   4.5   ø
// 2.8   ø    ø    ø    ø
//  ø    ø    ø    ø    ø

print(graphMatrix.edgesFrom(v2))
// [1: 2 -(1.0)-> 2: 3, 1: 2 -(3.2)-> 4: 5]
```

References

[1] Graph : https://ehdrjsdlzzzz.github.io/2019/01/21/Graph/

[2] [Swift Algorithm Club 번역] 그래프 (Graph) : https://oaksong.github.io/2018/04/08/swift-algorithm-club-ko-graph/

[3] Swift로 그래프 탐색 알고리즘을 실전 문제에 적용해보기 - BFS 편 : https://wlaxhrl.tistory.com/89

[4] Graph 자료구조 : https://yagom.net/forums/topic/graph-자료구조-2/

[5] [Data Structure] 그래프 순회, 탐색(DFS) - 자료 구조 : https://palpit.tistory.com/898

References

[6] DFS (Depth-First Search) BFS (Breadth-First Search) 개념 : https://hucet.tistory.com/83

[7] 자료구조 그래프(Graph) : https://yeolco.tistory.com/66

[8] 자료구조 :: 그래프(1) "정의, 그래프의 구현" : http://egloos.zum.com/printf/v/755618

[9] [알고리즘] BFS & DFS : https://hyesunzzang.tistory.com/186

[10] 자료구조 #4 그래프(Graph) : https://nextcube.tistory.com/190?category=459354

Thank you!