PROJECT 3

과목명	데이터구조실습
담당교수	이형근 교수님
학과	컴퓨터정보공학부
학년	2학년
학번	2019202009
이름	서여지
제출일	2020. 12. 04(금)

KWANGWOON UNIVERSITY

1. Introduction

이번 과제는 mapdata.txt의 도로 정보를 읽어 가게 사이의 최단 경로를 구하는 프로그램을 구현하는 것이다. 프로그램은 BFS, Dijkstra, Bellmanford, Floyd 알고리즘을 이용한다. 최단 경로에 해당하는 vertex를 정렬한 뒤, 라빈카프 공국의 압축 규칙에 맞도록 압축하여 출력한다. command.txt에서 명령어를 읽어 동작하고, log.txt파일에 결과를 출력한다. 프로그램이 수행하는 명령어는 다음과 같다.

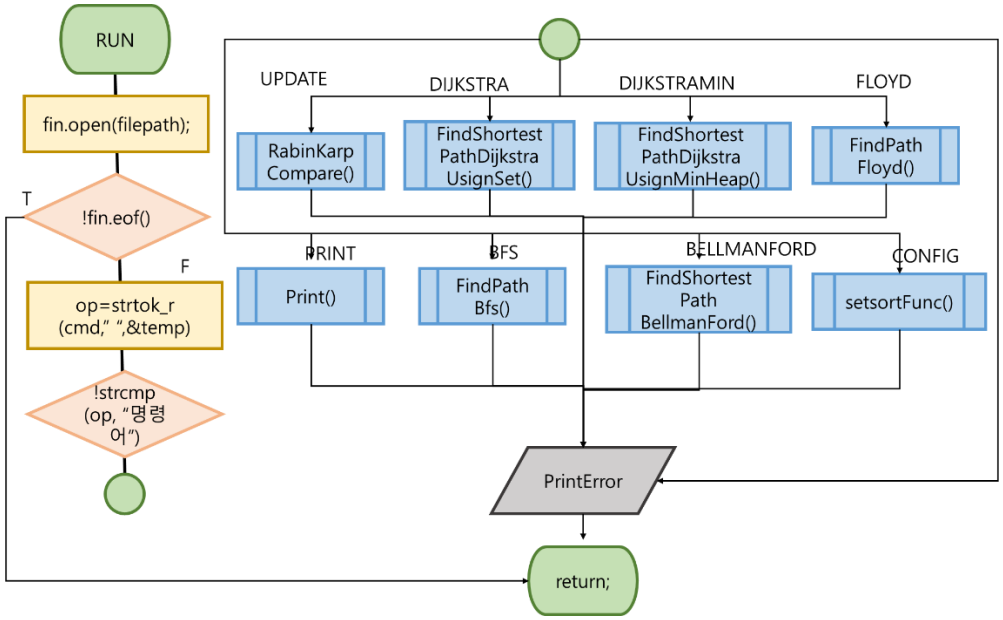
LOAD	mapdata.txt의 정보를 읽어 그래프를 생성한다.
UPDATE	라빈카프 공국의 규칙에 따라 가게 사이의 간선비를 조정한다.
PRINT	현재 그래프 정보를 Matrix 형태로 출력한다.
BFS	BFS를 수행하여 주어진 두 vertex 사이의 최단경로를 찾는다. 이때 Weight는 고려하지 않는다.
DIJKSTRA	set을 이용하여 구현한 다익스트라 알고리즘으로 주어진 두 vertex 사이의 최단경로를 찾는다.
DIJKSTRAMIN	MinHeap을 이용하여 구현한 다익스트라 알고리즘으로 주어진 두 vertex 사이의 최단경로를 찾는다.
BELLMANFORD	Bellmanford 알고리즘을 이용하여 주어진 두 vertex사이 최단경로를 찾는다.
FLOYD	Floyd 알고리즘을 이용하여 모든 vertex간의 최단경로를 찾는다.
CONFIG	사용하는 정렬 알고리즘을 변경한다.

LOAD 명령어를 통해 mapdata.txt의 내용을 그래프로 구성한다. 그래프는 vertex와 edge의 연결리스트 형태이다. vertex는 vertex의 key, 연결된 edge의 수와 첫 번째 edge의 포인터, 연결된 vertex의 포인터를 갖는다. edge는 종점의 key, 가중치, 다음에 연결된 edge의 포인터 정보를 갖는다. 그래프는 vertex 연결리스트의 head node의 포인터와 연결된 vertex의 수를 갖고, 추가로 각 vertex의 이름을 저장한 nameArry를 갖는다.

2. Flowchart

A. Manager.run()

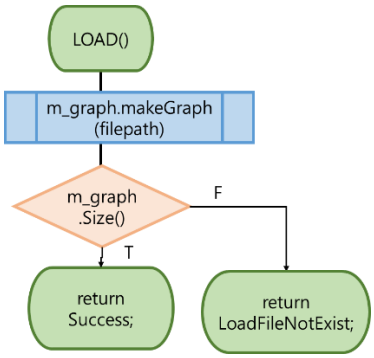
프로그램의 전체 실행을 다룬다. command.txt 에서 명령어를 읽어 해당하는 함수로 연결한다.



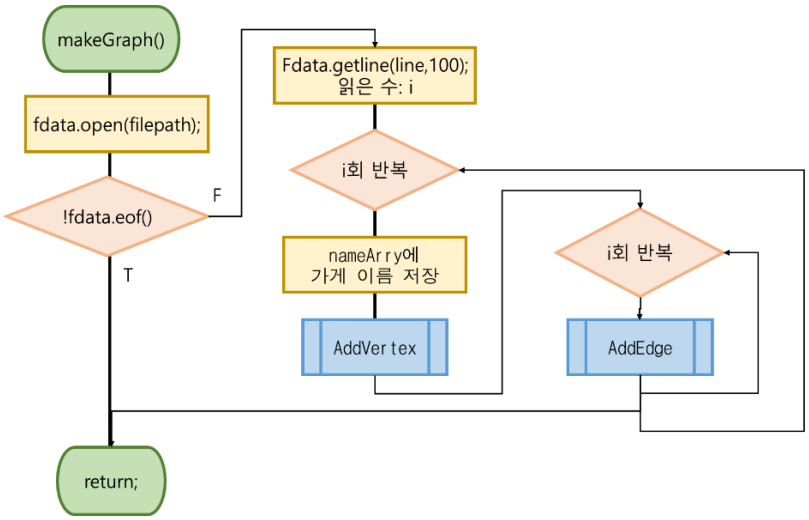
B. LOAD

mapdata.txt 의 정보를 이용해 그래프를 구성한다.

i. LOAD()



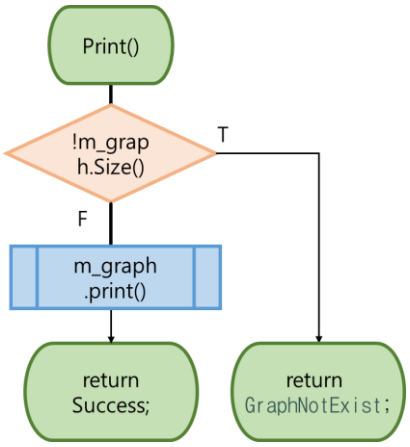
ii. makeGraph()



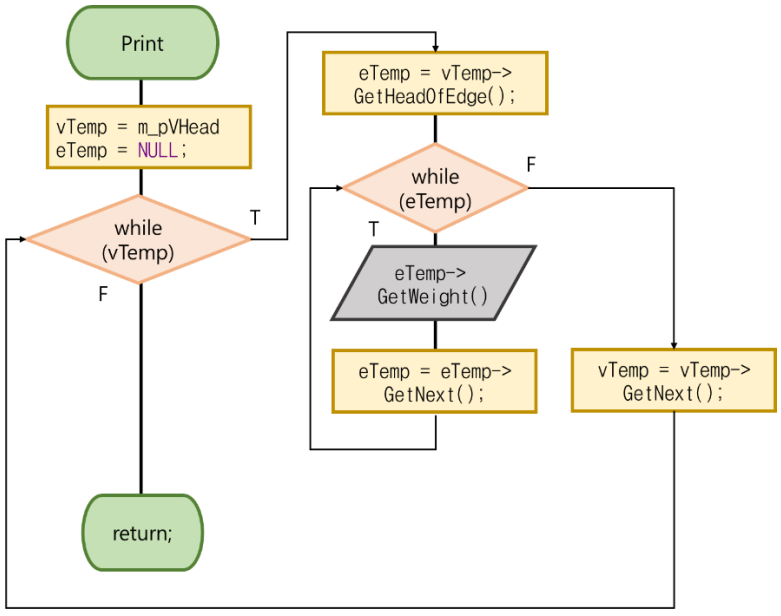
C. PRINT

그래프의 정보를 log.txt 에 모두 출력한다.

i. Manager::Print()



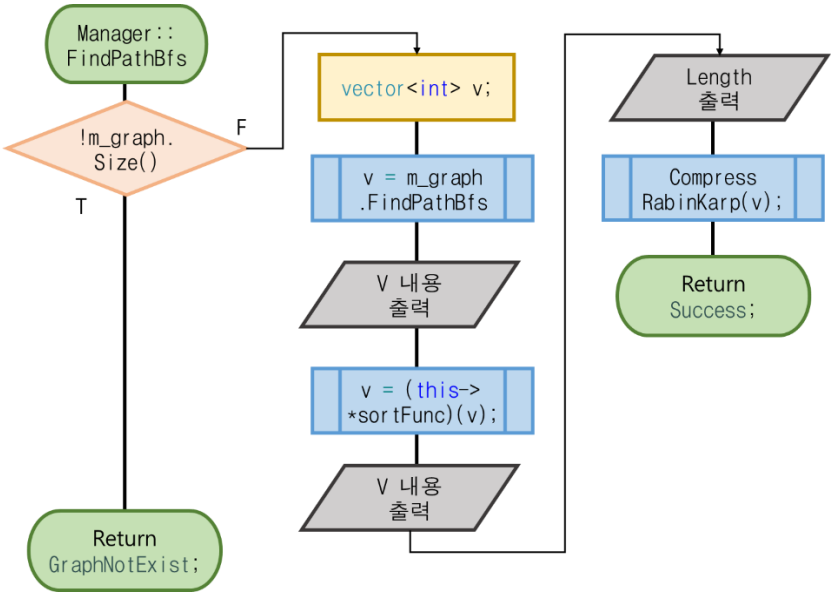
ii. Graph::Print



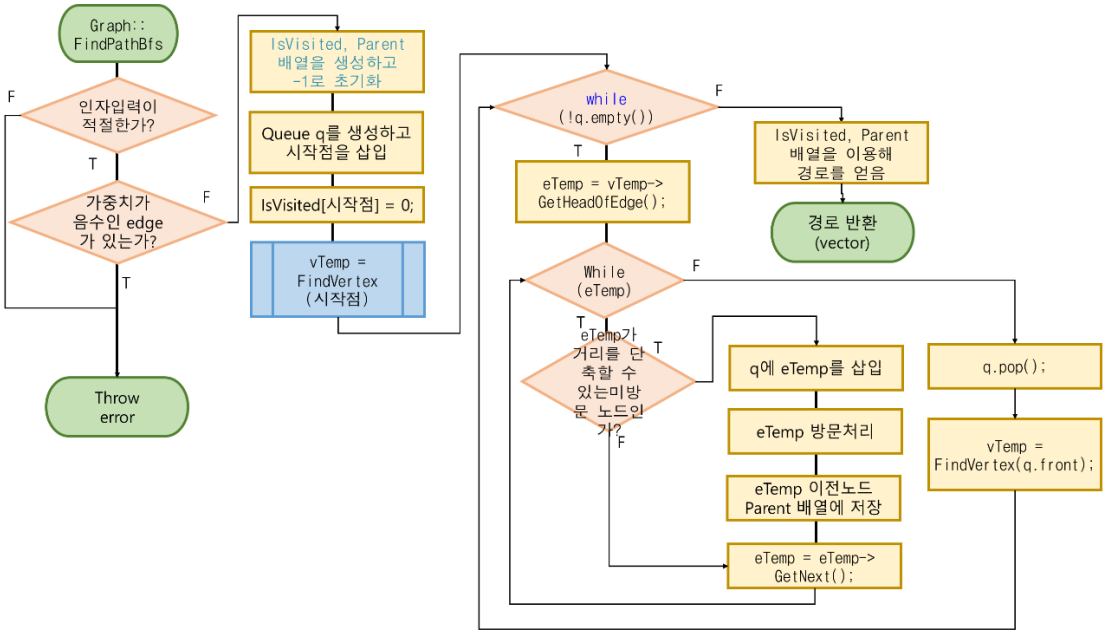
D. BFS

인접한 정점을 차례로 방문하는 BFS 를 이용하여 경로를 찾는다. 이때 가중치는 고려하지 않는다. BFS 를 비롯한 경로를 출력하는 동작은 FindPathBfs 와 유사한 구조를 갖는다. 각각 해당하는 알고리즘을 통해 경로를 얻고, 출력한 뒤, 정렬 후 다시 출력, 압축 후 출력하는 구조이다.

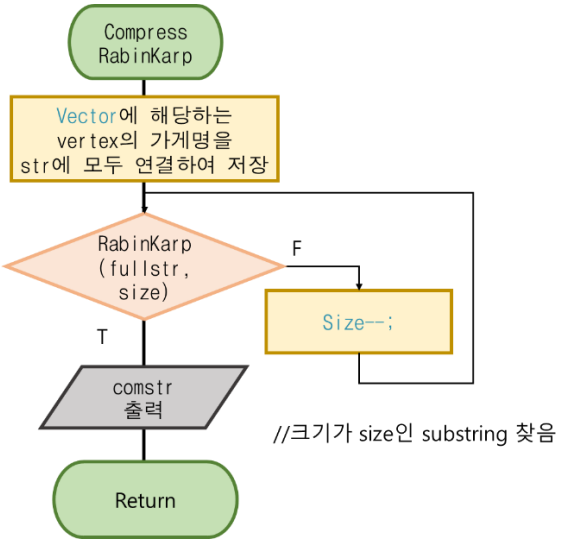
i. Manager::FindPathBfs



ii. Graph::FindPathBfs



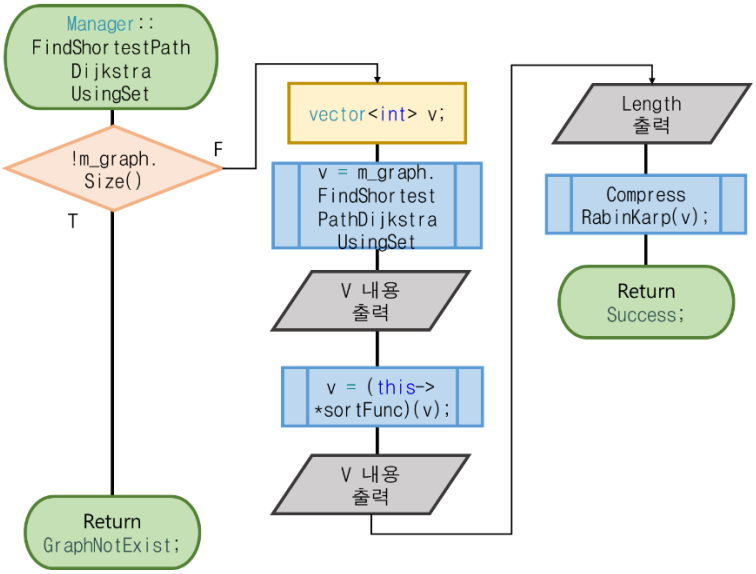
iii. CompressRabinKarp



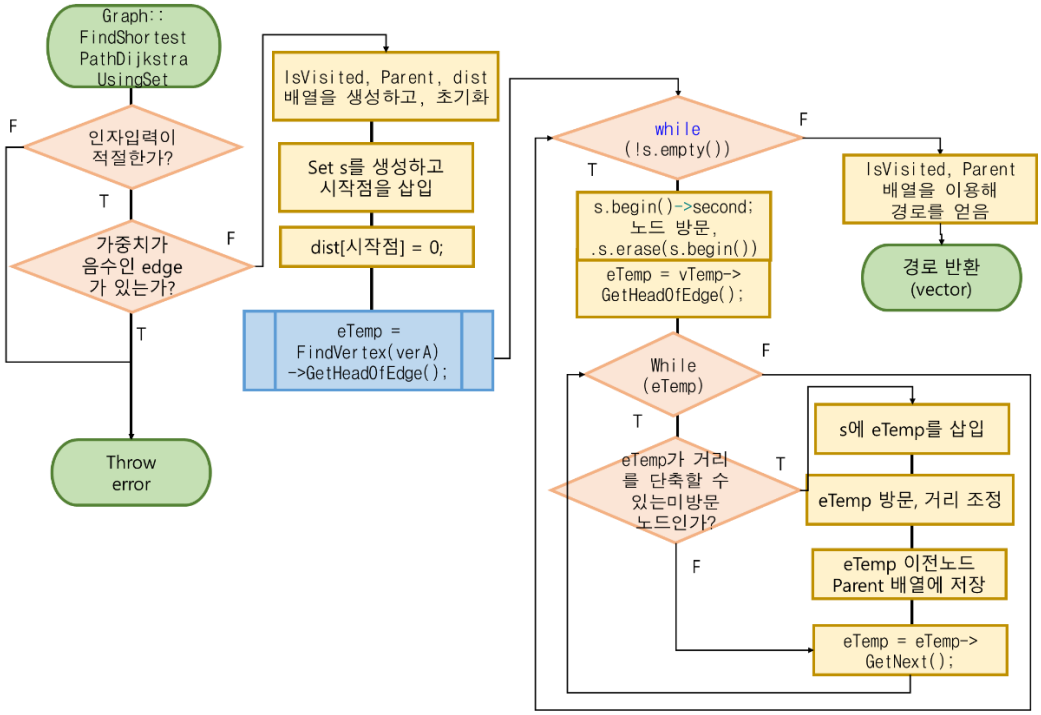
E. DIJKSTRA

set 을 이용하여 구현한 다익스트라 알고리즘을 통해 인자로 주어진 정점 사이 최단거리를 찾는다.

i. Manager::FindShortestPathDijkstraUsingSet



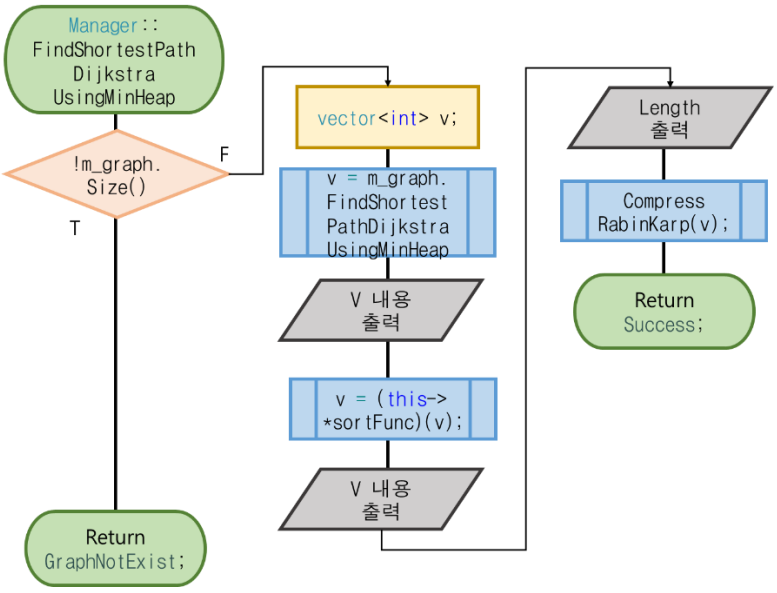
ii. Graph:: FindShortestPathDijkstraUsingSet



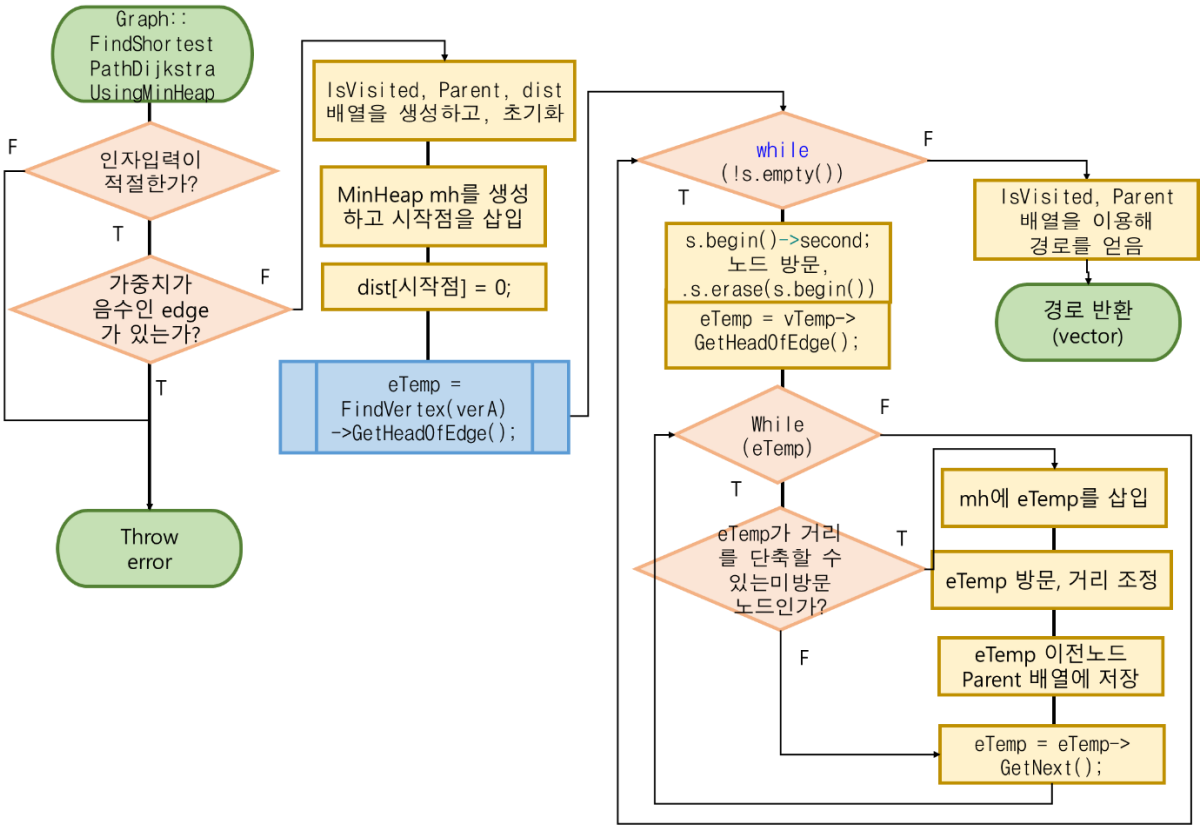
F. DIJKSTRAMIN

minheap 을 이용하여 구현한 다익스트라 알고리즘을 통해 인자로 주어진 정점 사이 최단거리를 찾는다.

i. Manager::FindShortestPathDijkstraUsingSet



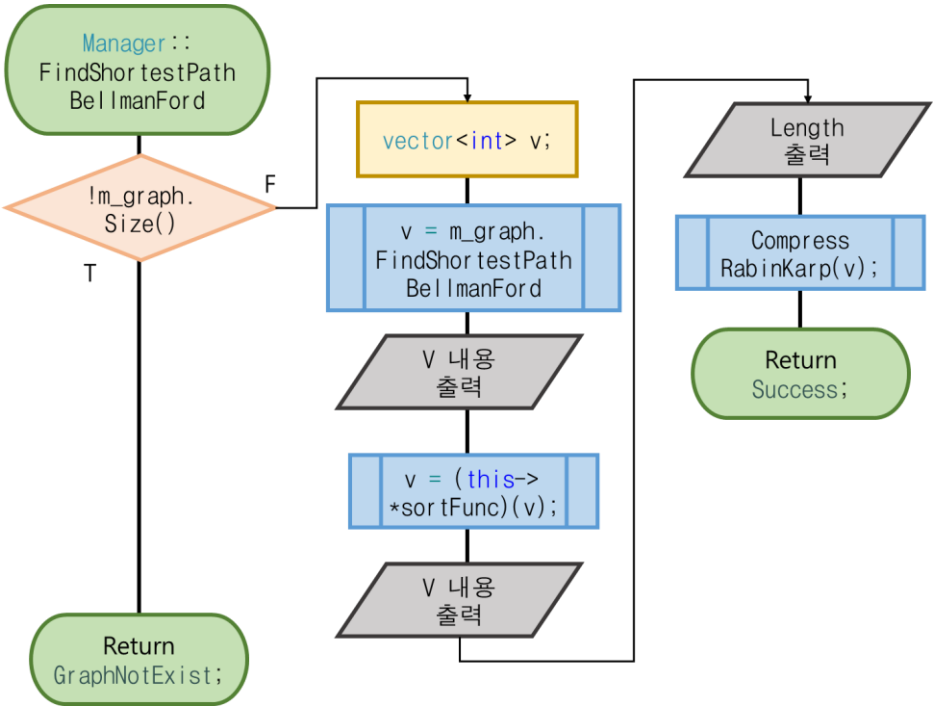
ii. Graph::FindShortestPathDijkstraUsingSet



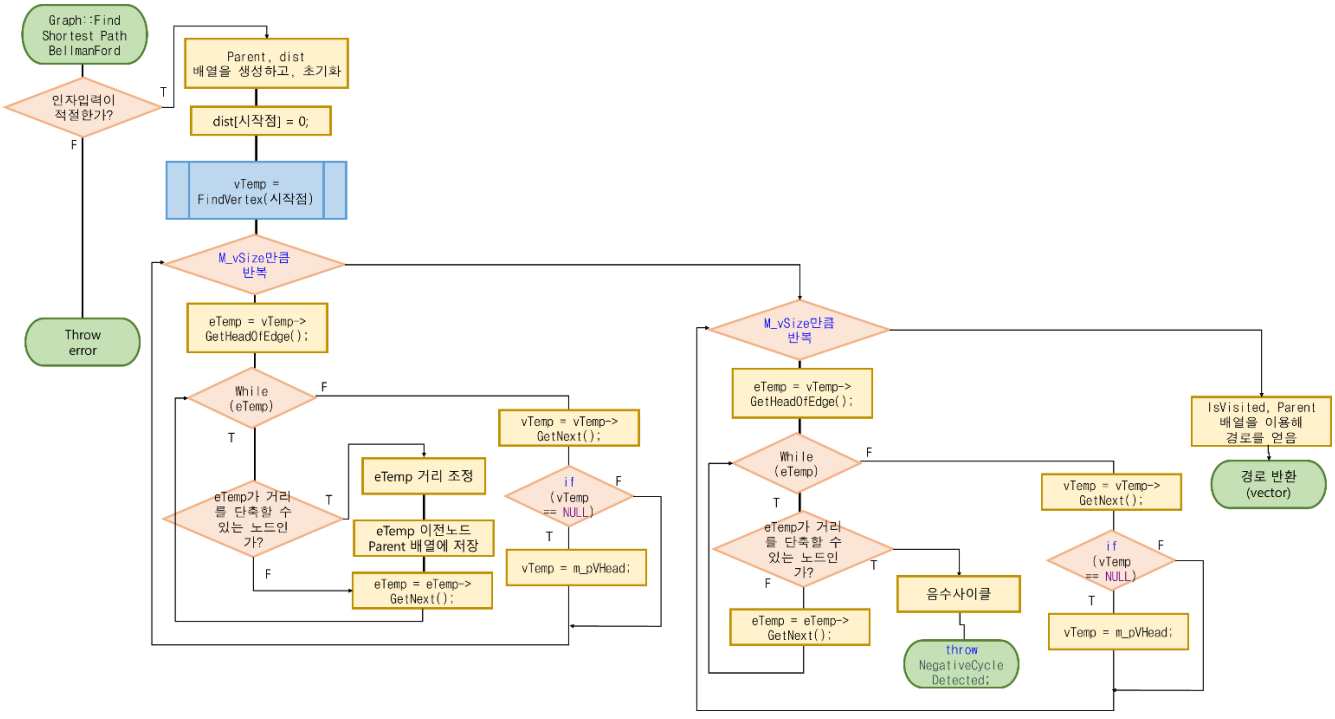
G. BELLMANFORD

벨만포드 알고리즘을 통해 인자로 주어진 정점 사이 최단거리를 찾는다. 거리를 얻는 계산을 완료한 뒤, 다시 한 번 같은 과정을 거쳐 값이 변하면 음수사이클을 갖는 것으로 판단한다.

i. Manager::FindShortestPathBellmanFord

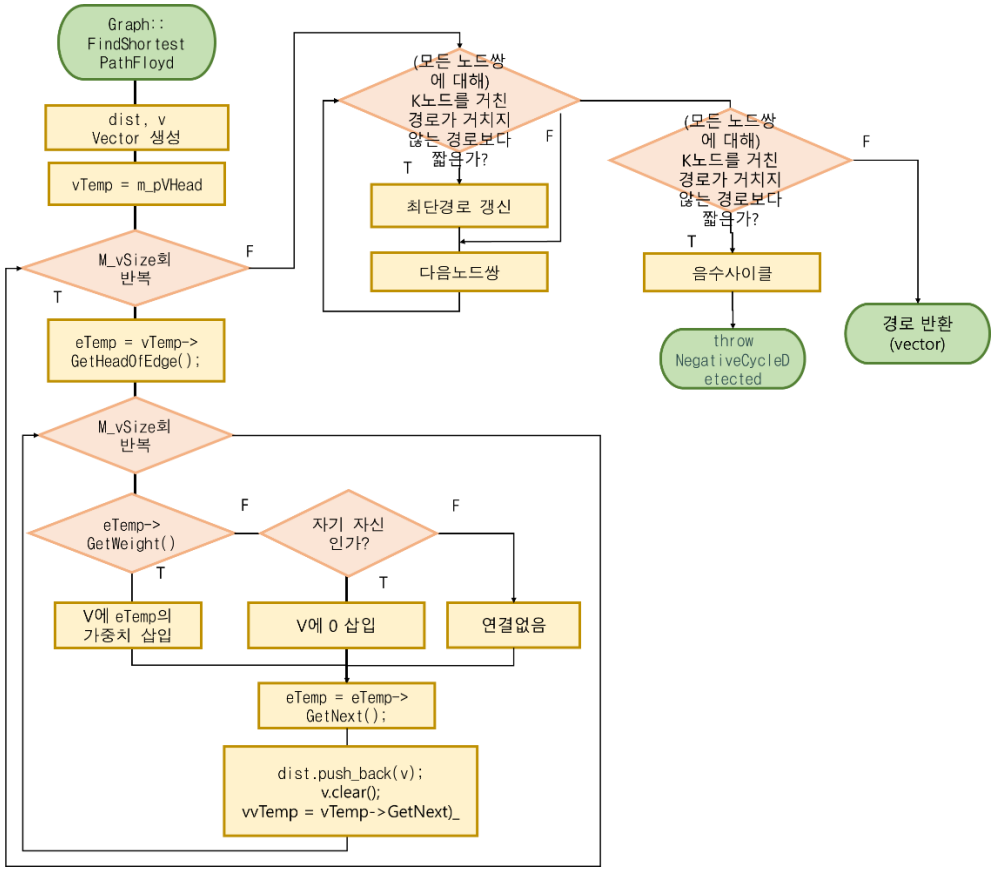


ii. Graph:: FindShortestPathBellmanFord



H. FLOYD

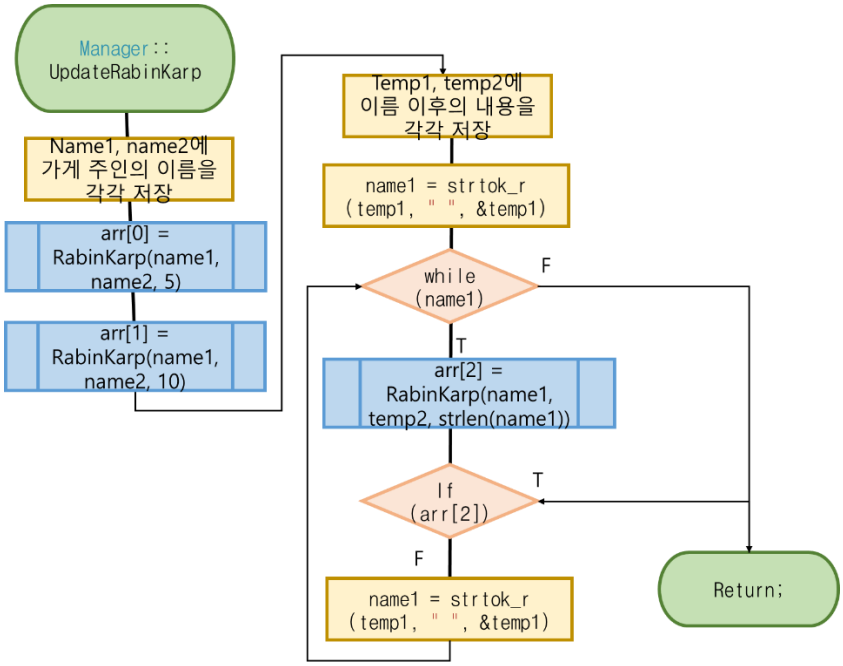
플로이드 알고리즘을 통해 그래프의 모든 vertex 쌍에 대한 최단거리를 계산한다. 벨만포드 알고리즘에서와 같이 마지막에 계산 과정을 한 번 더 진행하여 값이 변하면 음수사이클이 있는 것으로 판단한다.



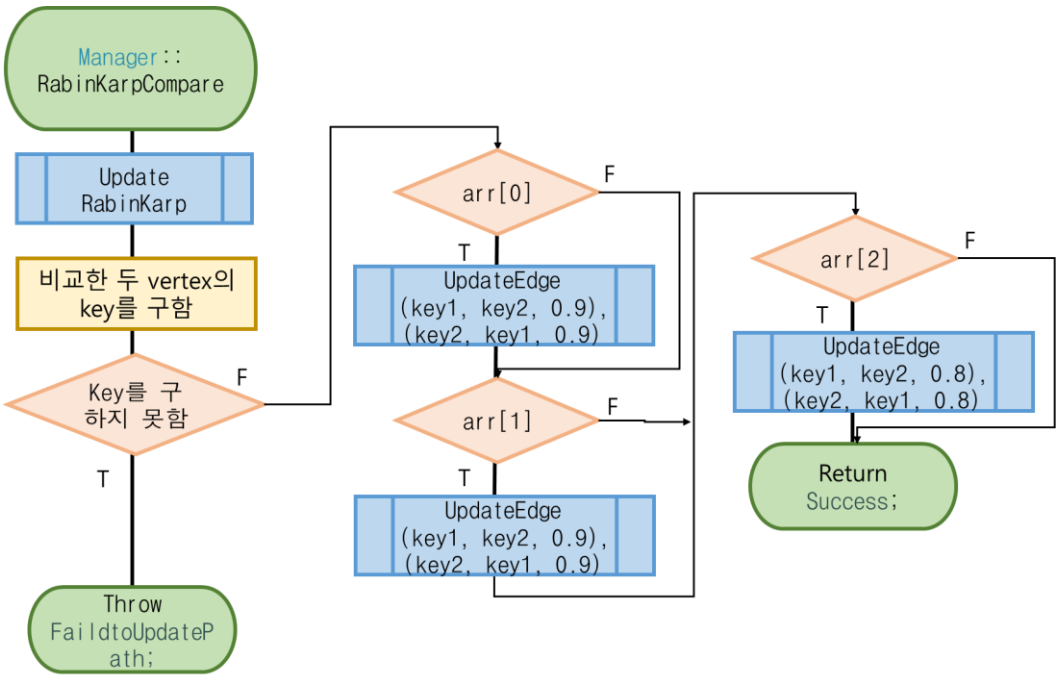
I. UPDATE

세 가지 규칙을 순서대로 라빈카프 알고리즘을 이용하여 확인한 후, 가중치를 계산하여 update 한다.

i. updateRabinKarp



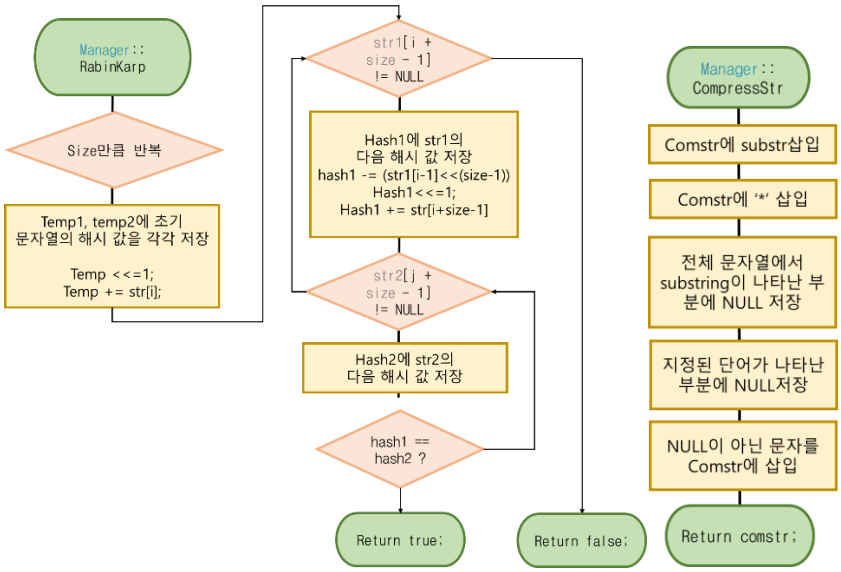
ii. RabinKarpCompare



J. RABINKARP

부분문자열의 해시 값을 이용하여 문자열을 비교하는 라빈카프 알고리즘이다. 입출력형식에 따라 오버로딩 되었으나 구조는 유사하므로 한가지 경우에 대해 표현하였다.

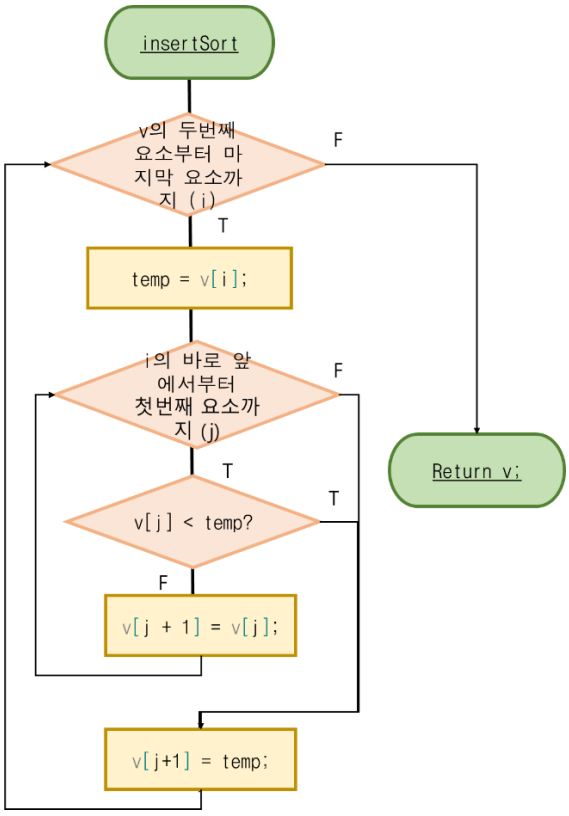
i. RabinKarp



K. SORT

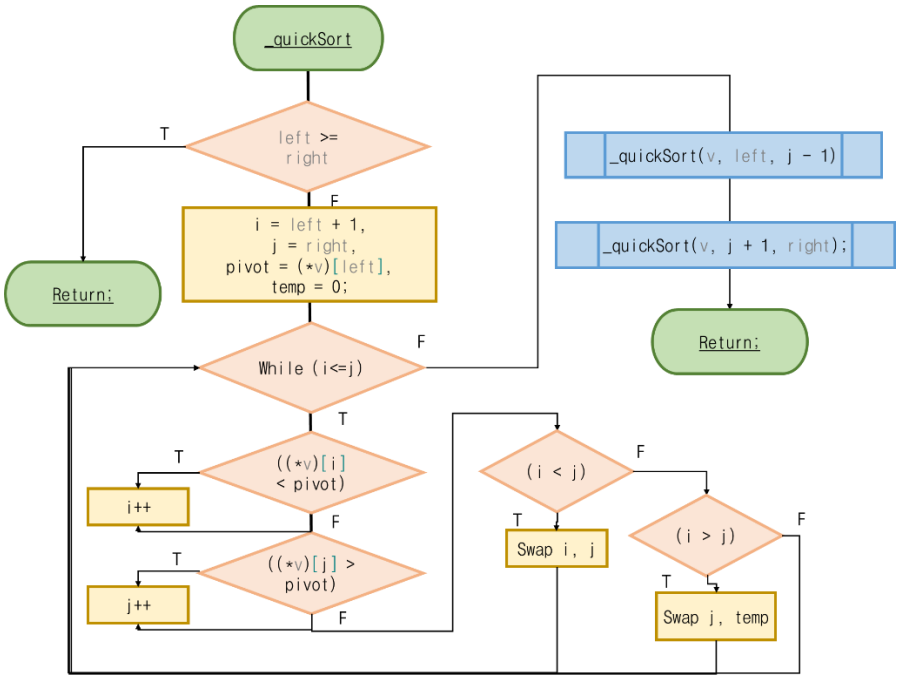
i. insert

특정 값의 앞에 적절한 위치를 찾아 삽입하는 방법으로 정렬하는 삽입정렬이다.



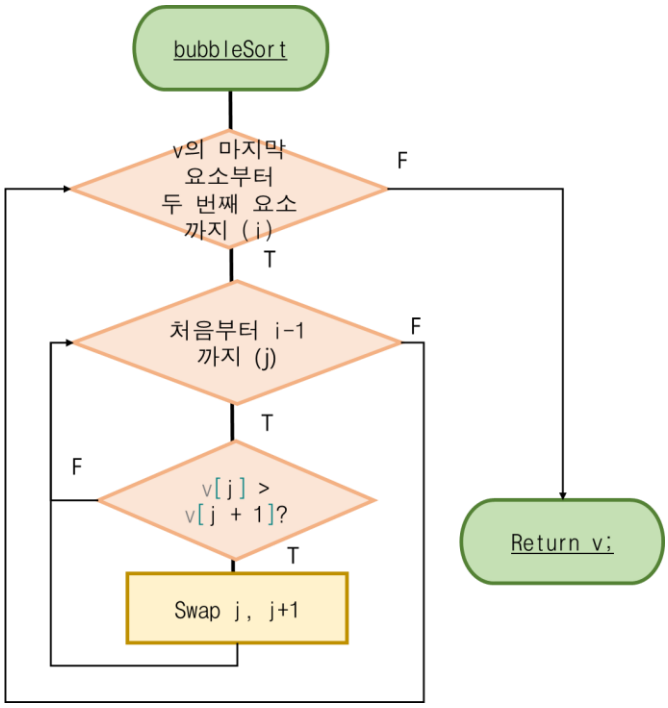
ii. quick

배열의 양 끝에서 pivot 에 대한 상대적인 크기를 확인하여 정렬하는 퀵정렬이다.



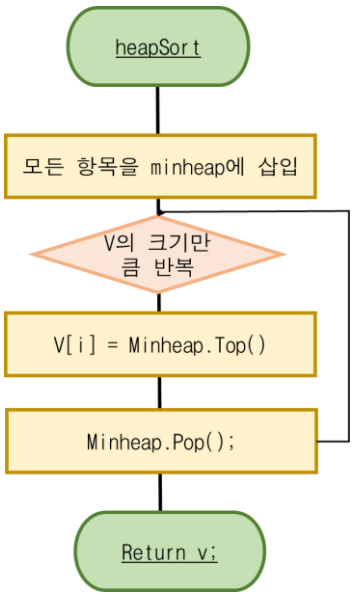
iii. bubble

인접한 요소의 크기를 비교하여 서로 교환하며 정렬하는 버블정렬이다.



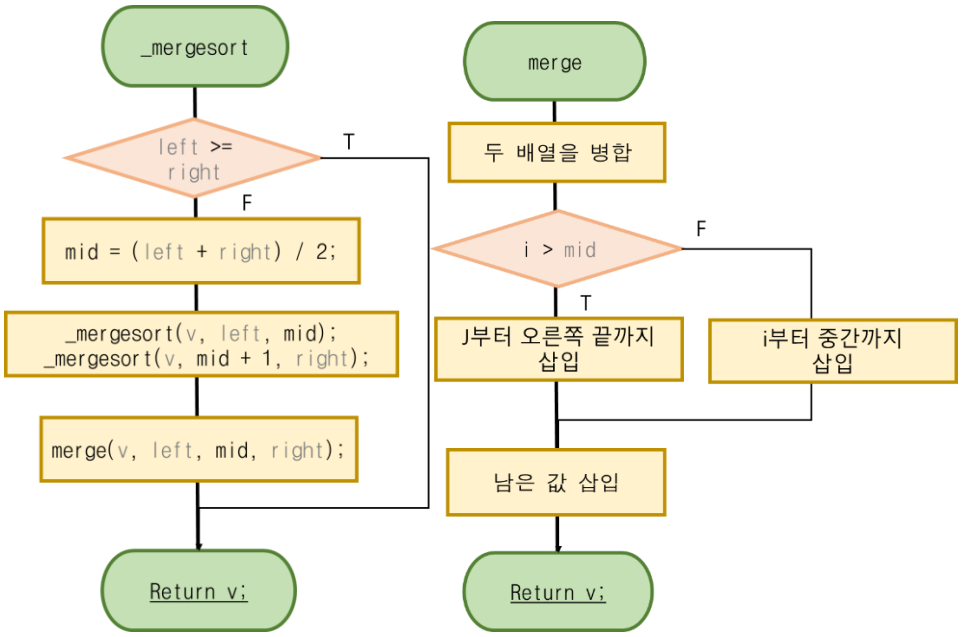
iv. heap

minheap 을 이용하여 정렬하는 힙정렬이다. 사용한 heap 은 다익스트라 알고리즘에 이용되었던 것이므로 실제 구현시에 pair 에 같은 값을 두 번 넣어 구현하게되었다.



v. merge

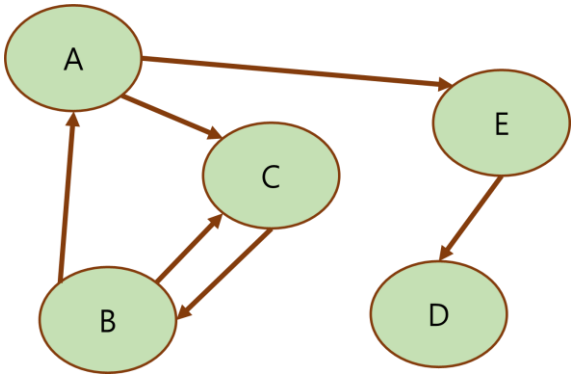
배열을 하나의 요소만을 갖는 작은 배열로 나누어 병합하는 과정을 통해 정렬하는 병합정렬이다.



3. Algorithm

A. BFS

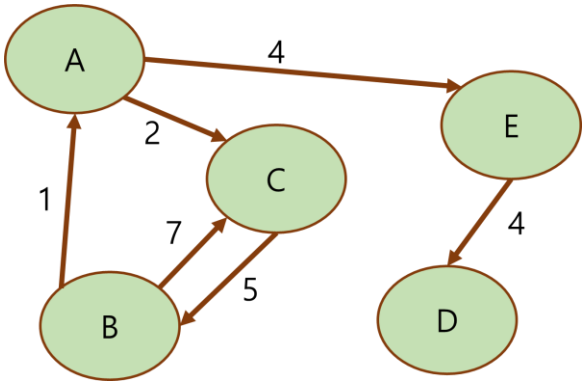
너비 우선 탐색은 자신과 인접한 정점을 먼저 방문한다. 이러한 특징을 이용하면 BFS 를 queue 를 이용하여 구현할 수 있다. 첫 번째 노드를 queue 에 삽입하고, 이후에 queue 에서 노드를 제거할 때는 해당 노드와 연결된 모든 노드를 queue 에 삽입한다. 이것을 queue 에 더 이상 노드가 남아있지 않을 때까지 반복하는 과정이 너비 우선 탐색이다. 탐색을 통해서 경로를 알아내기 위해 parent 배열을 추가로 사용하였다. 해당 노드를 방문하기 전에 위치한 노드를 parent 배열에 저장하여 탐색순서를 확인할 수 있도록 하였다. 이 방법은 다른 알고리즘을 이용한 과정에서도 동일하게 이용되었다.



위의 그래프에서 A 에서 BFS 를 진행한다면, queue 에 A 를 넣어 시작하게 된다. 이미 방문한 노드를 확인하기 위해 IsVisited 배열을 이용하여 queue 에 노드를 삽입할 때 방문했음을 표시한다. queue 의 최상위에 있는 a 노드와 연결된 다른 노드인 c, e 노드를 queue 에 삽입하고, a 노드를 삭제한다. 이때 최단경로를 찾기 위해 생성한 parent 배열의 c, e 에는 a 를 삽입한다. 다시 queue 의 최상위에 있는 c 가 연결된 노드 b 를 찾아 queue 에 삽입한다. 다음 queue 의 최상위 노드는 e 노드이고, e 노드를 제거하며 d 노드를 삽입한다. 모든 노드의 방문이 완료되었지만 아직 queue 에 노드가 남아있으므로 연산이 종료되지 않는다. 이후 b 노드와 d 노드를 삭제한다. b 노드가 연결된 a,c 노드는 이미 방문했으므로 queue 에 삽입되지 않는다.

B. 다익스트라

다익스트라 알고리즘은 가중치를 가진 그래프에서, 가장 비용이 적은 경로를 선택하는 알고리즘이다. 탐색을 진행하는 과정에서 당장 다음 노드로 이동할 때 가장 비용이 적은 경로를 선택하므로 그래프를 전체적으로 고려한 뒤 구한 최단경로보다 비효율적인 결과를 얻을 수도 있다. 다익스트라 알고리즘은 시작 노드에서 갈 수 있는 가장 가까운 노드로 이동하고, 각 노드에 대한 최단거리를 갱신한다. 시작점에서 가장 가까운 노드까지의 거리와, 도착한 노드에서 다시 가장 가까운 노드까지의 거리를 더한 결과가 기존의 비용보다 적다면 새로운 최단거리를 얻을 수 있다.



위의 그래프를 다익스트라 알고리즘을 이용하여 A 부터 최단경로를 찾아보기로 한다. a 노드는 2 개의 노드와 연결된다. 최단경로가 저장된 배열은 다음과 같다.

0	-	2	4	-

가까운 노드인 c 를 방문한 뒤, 최단경로 배열을 갱신한다. c 와 연결된 노드는 b 이고, 해당 가중치인 5 와 c 의 비용인 2 를 더한 값 7 은 무한대보다 작은 수 이므로 값이 갱신된다.

0	7	2	4	-

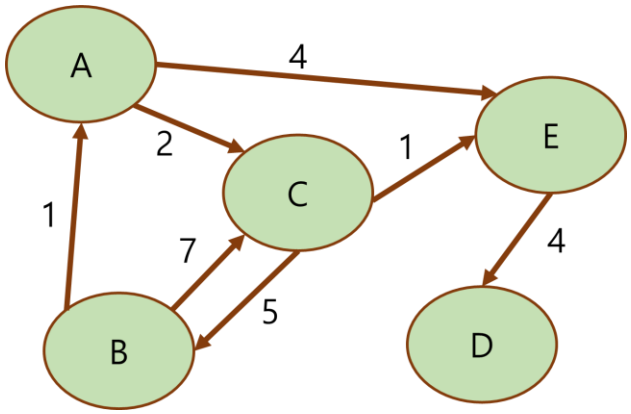
아직 방문하지 않은 노드 중 다음으로 가까운 노드인 e 를 방문한다. e 는 d 와 연결되어 있고 e 를 거쳐 d 로 이동하는 최단 거리는 8 이므로 무한대보다 작은 수이다. 따라서 값이 다시 갱신된다.

0	7	2	4	8

미방문 노드는 b 와 d 이고, b 의 값이 더 작으므로 b 를 방문한다. b 가 연결된 노드인 a 와 c 는 이미 더 작은 값을 가지고 있어 값이 갱신되지 않는다. 마지막 노드인 d 를 방문하면, 더 이상 연결된 노드가 없기 때문에 값이 갱신되지 않는다. 모든 노드에 방문했으므로 알고리즘이 종료된다.

C. 벨만포드

벨만포드 알고리즘은 다익스트라 알고리즘과 달리 그래프를 전체적으로 고려하여 최단경로를 구한다. 또한 벨만포드 알고리즘은 가중치가 음수인 경우에 음수사이클을 갖지 않는다면 최단경로를 구할 수 있다. 사이클이 음수인 경우 계산을 반복하면 계속해서 비용이 적어지므로 최단경로를 구할 수 없다. 벨만포드 알고리즘은 알고리즘은 a 에서 c 로 이동할 때 a 에서 c 로 직접 가는 비용보다, a 와 b 사이 최소비용과 b 와 c 사이의 최소비용을 더한 것을 합했을 때의 비용이 더 적다면 그 경로를 a 와 c 사이 최단경로로 갱신한다.



위의 그래프를 벨만포드 알고리즘을 이용하여 a 에서 d 까지 최단경로를 찾는다.

	a	b	c	d	e
a	0	-	2	-	4
b	0	-	2	-	4
c	0	7	2	-	3
d	0	7	2	-	3
e	o	7	2	7	3

D. 플로이드

플로이드 알고리즘은 그래프를 구성하는 서로 연결된 모든 노드 쌍에 대한 최단거리를 계산한다는 점에서 다른 알고리즘과 차이가 있다. 벨만포드 알고리즘과 동일하게 음수 가중치를 가져도 최단거리를 구할 수 있다. 두 알고리즘에서 음수 사이클을 확인하는 방법은 계산을 완료한 뒤에 한 번 더 계산했을 때 값이 변화하면, 음수 사이클을 거쳐서 최단경로의 비용이 갱신된 것이므로 이것을 통해 확인할 수 있다.

E. 라빈카프

라빈카프 알고리즘은 문자열을 비교하는 알고리즘이다. 이때 문자열을 구성하는 문자를 직접 비교하는 것이 아닌 해시값을 구하여 이용한다. 이번 프로젝트에서 해시값은 2 의 지수로 나타나는 문자열의 각 자리에 해당 문자의 아스키코드 값을 곱한 값을 해시로 이용하였다. 해시를 이용하여 문자열을 비교하는 것의 장점은 계산이 간단하고 빠르다는 것이다. 해시 값을 자리수와 아스키코드의 곱으로 정했기 때문에 전체 문자열에서 부분 문자열을 구할 때 전체 문자가 한 칸씩 이동하는 상황에서 간단하게 새로운 해시값을 얻을 수 있었다.

“welcome“ 문자열에서 “elc”를 찾는 경우를 살펴보면 다음과 같다. 찾으려는 문자열의 길이가 3 이므로 welcome 의 앞부분 3 글자의 해시를 구한다.

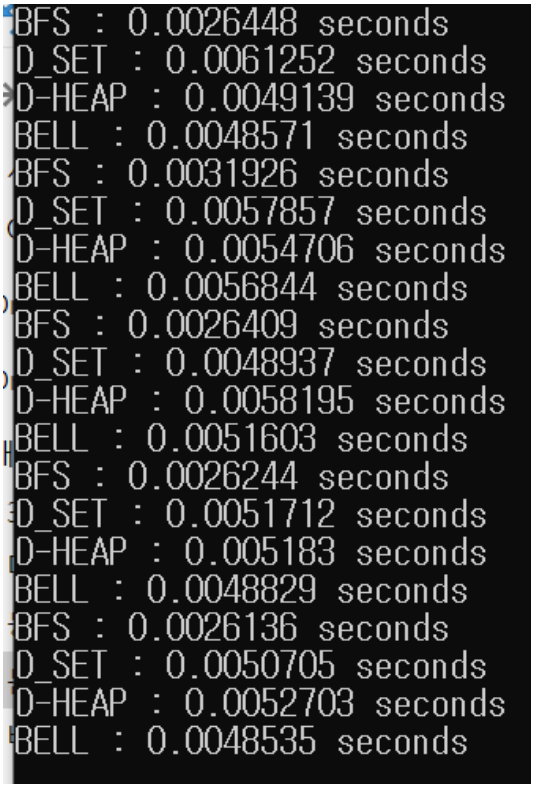
w	e	l	c	o	m	e
e	l	c				

w: 119, e:101, c:99 이므로 119*4 + 101 * 2 + 99 = 777 이고, e: 101, l: 108, c:99 이므로 101*4 + 108*2 + 99 = 701 이다. 두 해시 값이 일치하지 않으므로 “wel”come 에서 한 글자 이동하여 w”elc”ome 을 비교한다.

w	e	l	c	o	m	e
	e	l	c			

wel 의 해시값인 777 에서 w 의 아스키코드 값인 119 와 자리 값인 2^2 를 곱한 476 을 빼고, 남은 값에 2 를 곱한 뒤 새로운 글자인 c 의 아스키 코드 값 99 를 더한다. (777 – 119*4)*2 + 99 = 701 이다. 해시 값이 일치하는 문자열을 확인한 결과 찾으려던 문자열 “elc”를 발견했음을 확인할 수 있다.

F. 정렬알고리즘의 성능차이



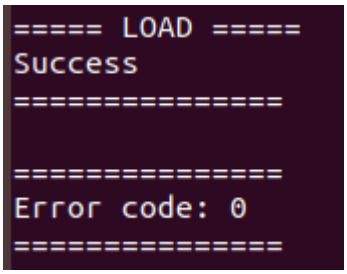
(seconds)	BFS 4 5	DIJKSTRA 4 5	DIJSTRAMIN 4 5	BELLMANFORD 4 5
quick	0.0026448	0.0061252	0.0049139	0.0048571
insert	0.0031926	0.0057857	0.0054706	0.0056844
merge	0.0026409	0.0048937	0.0058195	0.0051603
heap	0.0026244	0.0051712	0.005183	0.0048829
bubble	0.0026136	0.0050705	0.0052703	0.0048535

각 정렬알고리즘에 대한 각 경로 알고리즘의 결과는 위와 같이 나타났다. 경로 알고리즘의 경우 BFS 가 다른 알고리즘과 큰 차이로 빠른 속도를 보였다. 이것은 BFS 의 경우 가중치를 고려하지 않기 때문에 나타난 결과로 생각할 수 있다. set 을 이용한 다익스트라 알고리즘의 경우 퀵정렬이 가장 느리고 병합정렬이 가장 빠른 반면 heap 을 이용한 다익스트라 알고리즘은 퀵정렬이 가장 빠르고 병합정렬이 가장 느린 정 반대의 결과가 나타났다. 힙정렬과 퀵정렬은 set 을 이용한 다익스트라 알고리즘의 경우를 제외하고 대체로 빠른 편으로 나타났고, 가장 느릴것이라고 생각했던 버블정렬의 속도가 의외로 빠른 편으로 나타난 것을 확인할 수 있다. 그러나 몇 가지 경우를 제외하면 알고리즘 사이의 분명한 차이점을 확인하기엔 부적합하다. 또한 나타난 결과는 이해하기 어려웠다. 이것은 실험에 이용한 정보의 크기가 작은 것을 원인으로 생각할 수 있다. 충분히 큰 자료를 여러 개

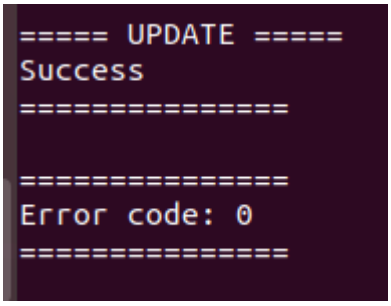
이용하여 실험하면 알고리즘 별 소요시간의 차이와 그 원인을 좀 더 선명하게 확인할 수 있을 것이다.

4. Result Screen

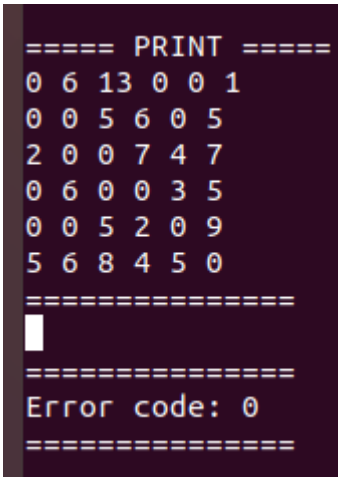
A. LOAD



B. UPDATE



C. PRINT



D. BFS

```
===== BFS =====
shortest path: 0 1 3
sorted nodes: 0 1 3
path length: 2
Course: Denny's *Bread ChoCar shop Labuajiea's Yoga class
===============

===============
Error code: 0
===============
```

E. DIJKSTRA

```
===== DIJKSTRA =====
shortest path: 5 3
sorted nodes: 3 5
path length: 4
Course: Labu*ajiea's Yoga class on's Computer Academy
===============

==============
Error code: 0
===============
```

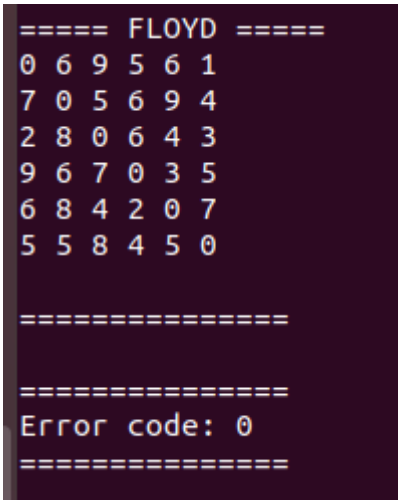
F. DIJKSTRAMIN

```
===== DIJKSTRAMIN =====
shortest path: 4 3 5
sorted nodes: 3 4 5
path length: 7
Course:  class *Labuajiea's YogaGrandy's KickingJeawon's Computer Aca
===============

===============
Error code: 0
===============
```

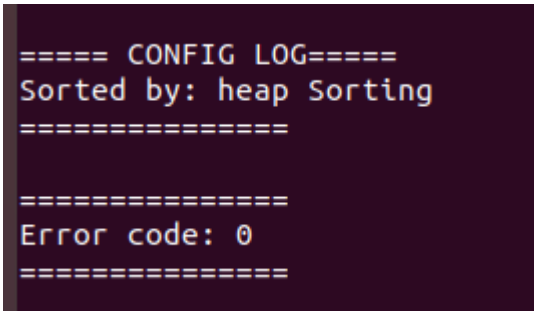
G. BELLMANFORD

```
===== BELLMANFORD =====
shortest path: 1 2 4
sorted nodes: 1 2 4
path length: 9
Course:  Kicking *Chodenny's Car shop Granddey's GoGrandy'sClass
===============

===============
Error code: 0
===============
```

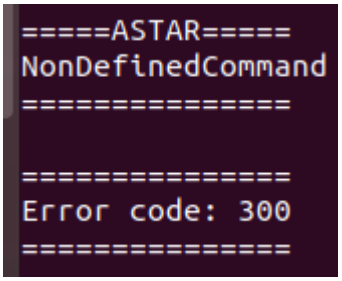
H. FLOYD



I. CONFIG



J. NONDEFINED



5. Consideration

작성한 라빈카프 알고리즘을 확인하는 과정에서 해시의 충돌이 발생하여 문제를 겪었다. 비교하려는 문자열의 길이가 충분히 긴 경우 해시의 충돌이 일어날 확률이 적기 때문에 프로젝트에서는 예외처리를 하지 않았지만, 짧은 문자열의 경우 충돌이 발생하여 비정상적인 결과가 나타날 수 있었다. 이 경우 <df labu>와 <ga clas>의 해시 값이 모두 8189 인 것을 직접 계산하여 확인하였다.

2 차 프로젝트는 클래스의 구조를 파악하는 것에서 어려움을 겪었다면 이번 프로젝트에서는 다양한 알고리즘의 원리를 이해하는 것이 주요 문제였다. 그래프 그림을 보며 직접 알고리즘을 적용하는 것은 어렵지 않게 느껴졌지만, 그것을 코드로 옮기는 과정에서 예상치 못한 문제가 자주 발생했다. 예를 들어 처음 BFS 를 이용하여 경로를 찾는 과정에서 순회를 성공적으로 종료했지만 그것으로 경로는 알아낼 수 없었다. 수업내용과 관련 자료를 찾아보며 추가적인 배열을 이용하여 문제를 해결하는 방법을 찾을 수 있었다.

6. Reference

vector, https://en.cppreference.com/w/cpp/container/vector

set, http://www.cplusplus.com/reference/set/set/

알고리즘 시각화,https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

BFS, 다익스트라, 4 차 산업혁명 시대의 이산수학, 김대수 외, 생능출판

다익스트라 알고리즘, https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

라빈카프 알고리즘 https://www.acmicpc.net/problem/3033

함수포인터 https://docs.microsoft.com/ko-kr/cpp/error-messages/compiler-errors-1/compiler-error-c2064?view=msvc-160

Milková, Eva. "BFS Tree and xy Shortest Paths Tree." Proceedings of International Conference on Applied Computer Science. 2010.

병합정렬 https://gmlwjd9405.github.io/2018/05/08/algorithm-merge-sort.html

chrono, https://jacking.tistory.com/988

이기훈교수님, 데이터구조설계 강의자료, 광운대학교 컴퓨터정보공학과, 2020

이형근교수님, 데이터구조실습 강의자료, 광운대학교 컴퓨터정보공학과, 2020