2020-2

Data Structure Project #3

학번: 2019202045

이름: 박수연

담당교수: 이형근 교수님

제출일자: 2020-10-09

Data Structure Lab. Project #3 Report

A. Introduction

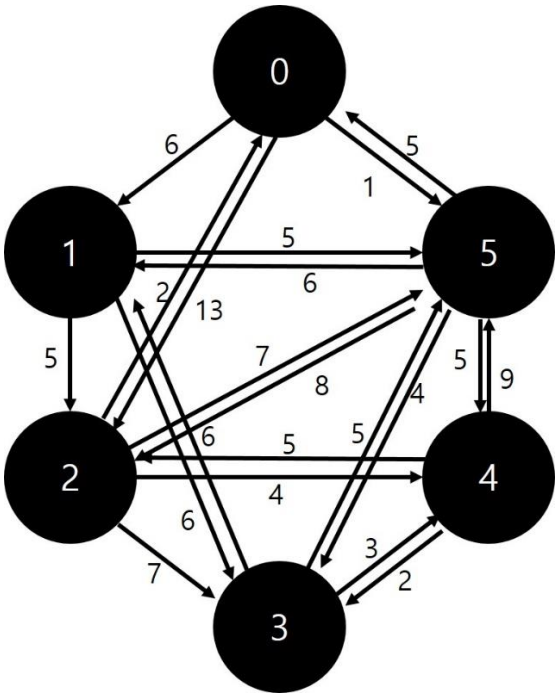
1. 제목

그래프를 통한 최단 경로 탐색 프로그램

2. 프로그램 설명

방향성과 가중치를 갖고 있는 Matrix 형태의 상점 거리 데이터 파일을 바탕으로 그래프를 생성하여 해당 그래프를 통한 최단 경로 탐색을 시행하는 프로그램을 작성한다. 그래프 생성 후 각 라빈카프 알고리즘을 통해 상점명과 상점의 주인명의 문자를 비교해 일부 조건에 따라 그래프 가중치의 값을 변경 하고 이후 최소 비용 경로를 BFS, 다익스트라, 벨만포드, 플로이드 알고리즘을 통해 도출한다. 해당 프로그램은 최단 경로 및 그 최단 거리의 key값과 정렬(다섯 종류 중 설정)된 key값, 경로 상점의 압축된 문자열을 출력한다. Weight가 음수일 경우, 다익스트라는 에러를 출력, 벨만포드에서는 음수 사이클이 발생할 경우 에러를 출력하고 그렇지 않을 경우 결과를 출력한다.

[structure implement - 프로젝트 최단 경로 구성 예시]

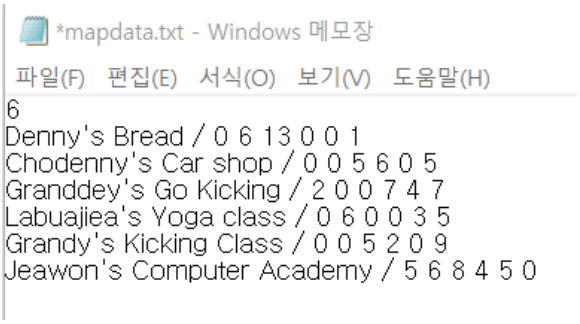


1) 가게 정보 데이터 - Linked List로 인접 리스트 생성

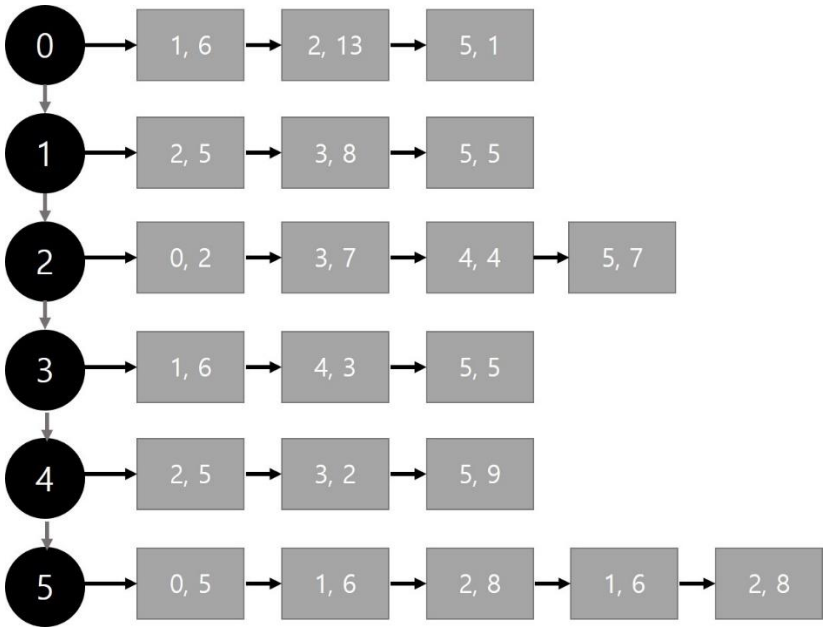
방향성 및 가중치를 갖고 있는 그래프 정보 데이터를 읽어 가게 간 거리 데이터를 Linked List의 형태로 생성한다. 거리 데이터의 첫 줄은 상점의 개수가, 이후 행과 열에 각각 Edge의 Start Vertex와 End Vertex의 Weight를 의미한다. 각 행의 문자열은 상점의 이름이다.

Graph Class로 인접 리스트를, Vertex와 Edge Class를 이용하여 인접 리스트 내 Vertex(Start Vertex의 key)와 Edge(end Vertex의 key, 가중치)를 구현하며, 해당 맵 데이터를 순차적으로 읽어 Linked List의 가장 끝에 연결하여 Vertex 오름차순으로 저장한다.

[mapdata.txt]



[mapdata.txt로 도출한 인접 리스트]



2) 가중치 Update

그래프 최단경로 탐색 전 형성된 그래프 내 거리 가중치 정보를 라빈카프 알고리즘을 통한 문자열 비교 (대소문자 구분x) 를 통해 다음과 같은 규칙에 의해 수정한다.

1. 상점 가게 주인의 이름이 5글자 이상 동일한 가게는 간선 비용이 10% 감소 (소수점 올림)

2. 상점 가게 주인의 이름이 10글자 이상 동일한 가게는 1의 규칙 적용 후 간선 비용이 10% 추가 감소 (소수점 올림)

3. 가게 이름이 한 단어 이상 같은 가게는 간선 비용이 20% 감소 (소수점 올림)

3) 그래프 연산

BFS : Start Vertex와 End Vertex를 입력 받아 큐를 통한 BFS 탐색을 통해 가중치를 고려하지 않은 최단 경로와 가중치를 고려한 거리를 도출한다.

DIJKSTRA : Start Vertex와 End Vertex를 입력 받아 set을 통한 DIJKSTRA 탐색을 통해 가중치를 고려한 최단 경로와 거리를 도출한다.

DIJSTRAMIN : : Start Vertex와 End Vertex를 입력 받아 min Heap을 통한 DIJKSTRA 탐색을 통해 가중치를 고려한 최단 경로와 거리를 도출한다.

BELLMANFORD : Start Vertex와 End Vertex를 입력 받아 벨만 포드 탐색을 통해 가중치를 고려한 최단 경로와 거리를 도출한다. 음수인 가중치가 있는 경우에도 사용 가능하며, 음수 사이클이 발생한 경우에는 에러를 출력한다.

FLOYD : 입력 인자 없이 모든 쌍에 대하여 최단 경로 행렬을 도출한다. 음수인 가중치가 있는 경우에도 사용 가능하며, 음수 사이클이 발생한 경우에는 에러를 출력한다.

4) 정렬 연산

최단 경로로 방문한 노드를 정렬하여 다시 출력한다.

Quick Sort, Insert Sort, Merge Sort, Heap Sort, Bubble Sort의 다섯 종류의 연산이 가능하며, CONFIG 명령어를 통해 어떤 Sort 방식을 사용할 것인지 설정할 수 있다. 기본값은 Quick Sort로 한다.

5) Function

명령어가 저장된 command.txt로부터 명령을 불러와 순차적으로 동작하며 '//'처리가 된 명령어는 무시한다.

LOAD : "mapdata.txt" 텍스트 파일로부터 데이터를 불러와, graph를 형성한다. 텍스트 파일이 존재하지 않을 시 에러코드를 출력한다.

UPDATE : 형성된 graph에 대하여 라빈카프 규칙에 따라 간선 비용 업데이트를 진행한다.

PRINT : 도로 정보 데이터를 읽어 Matrix 형태로 출력한다. 도로 정보 데이터가 존재하지 않을 시 에러코드를 출력한다.

BFS : StartVertex와 EndVertex 사이의 최단 경로를 BFS 알고리즘을 통해 도출한다. 입력한 Vertex가 그래프에 존재하지 않거나 인자가 부족한 경우 또는 음수인 가중치가 있는 경우 에러코드를 출력한다

DIJKSTRA / DIJSTRAMIN : StartVertex와 EndVertex 사이의 최단 경로를 DIJKSTRA 알고리즘을 통해 도출한다. 입력한 Vertex가 그래프에 존재하지 않거나 인자가 부족한 경우 또는 음수인 가중치가 있는 경우 에러코드를 출력한다

BELLMANFORD : StartVertex와 EndVertex 사이의 최단 경로를 벨만포드 알고리즘을 통해 도출한다. 음수인 가중치가 있는 경우에도 동작한다. 입력한 Vertex가 그래프에 존재하지 않거나 인자가 부족한 경우 또는 음수인 사이클이 발생한 경우 에러코드를 출력한다

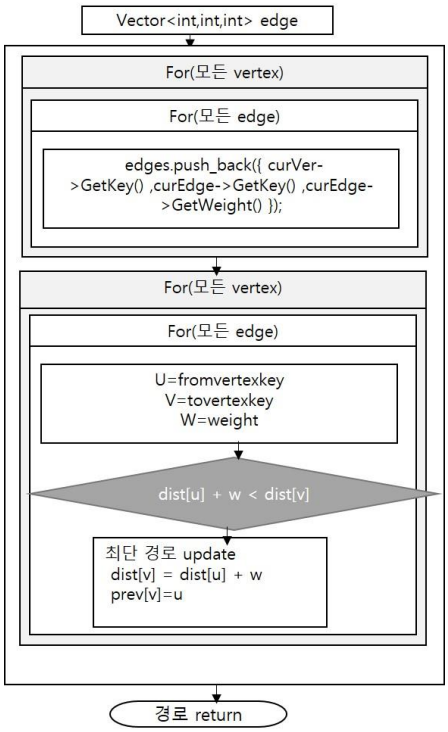
FLOYD : 모든 도시의 쌍에 대하여 해당 도시 간의 필요한 비용의 최소 거리를 행렬 형태로 도출한다.

CONFIG : 정렬 알고리즘을 선택한다.

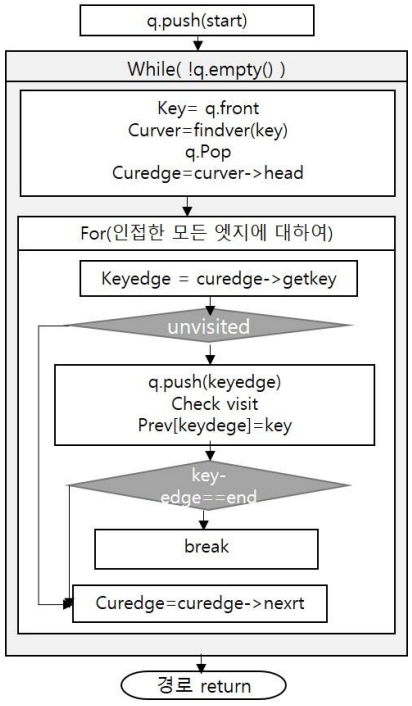
B. Flow chart

1) 최단 경로

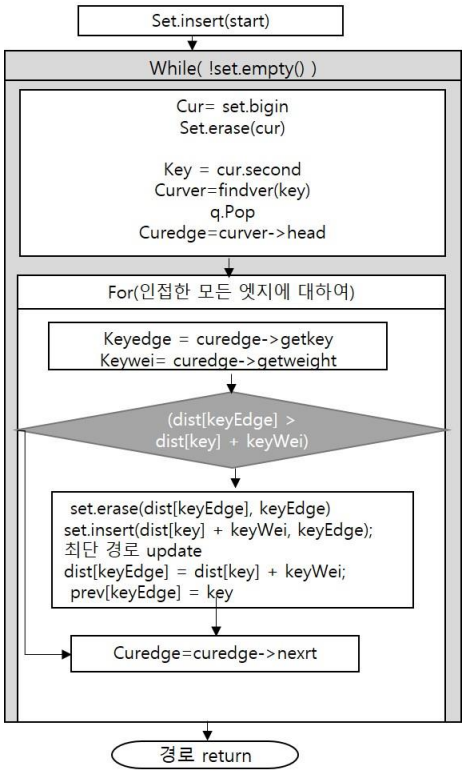
BELLMANFORD



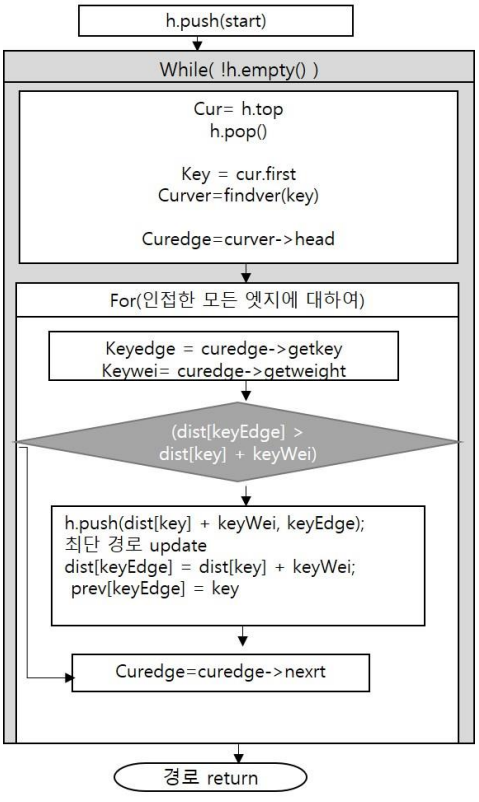
BFS



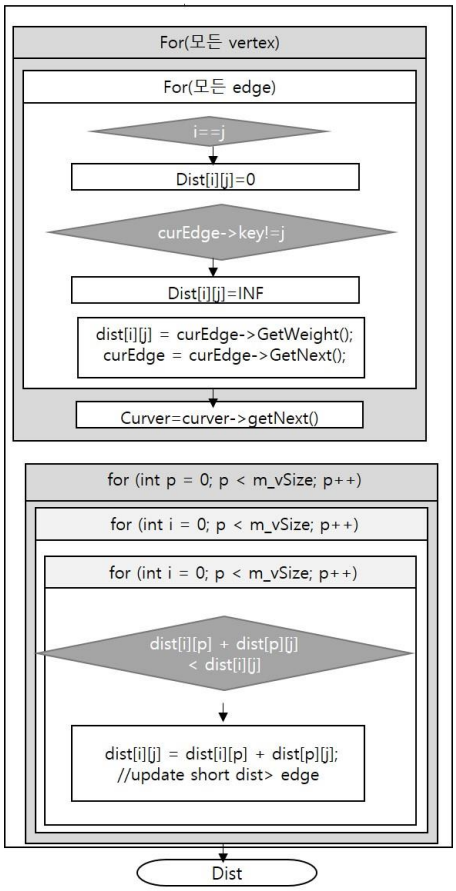
DIJKSTRA



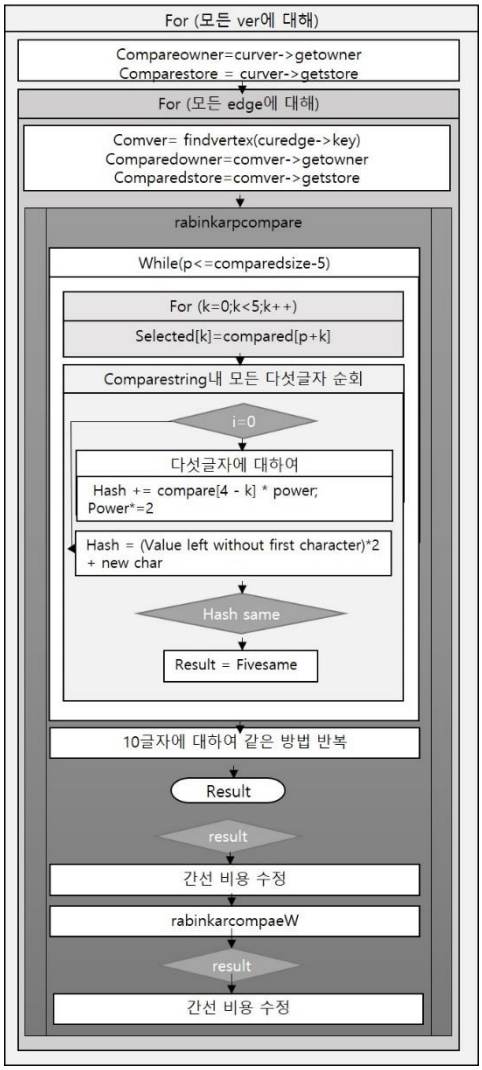
DIJKSTRAMIN



FLOYD

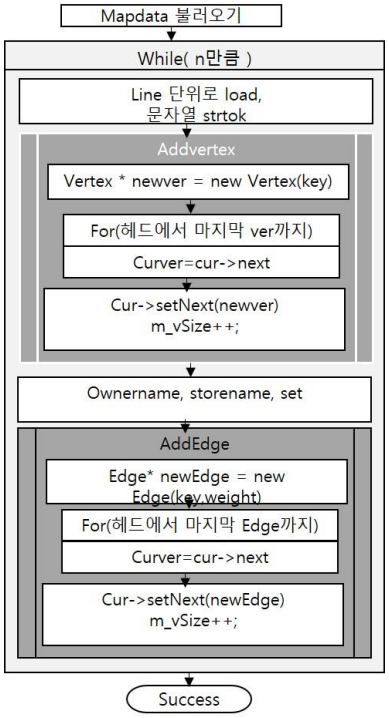


UPDATE



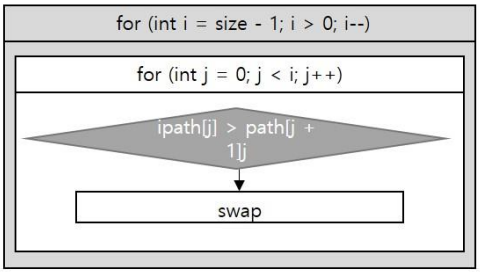
2) 명령어

LOAD

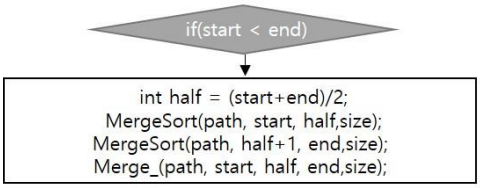


3) 정렬

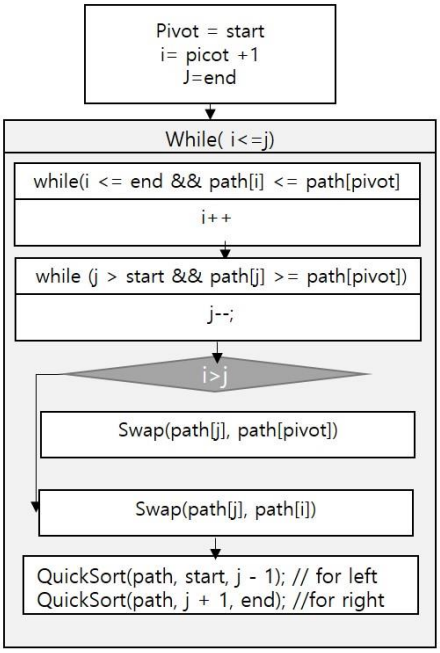
Bubble



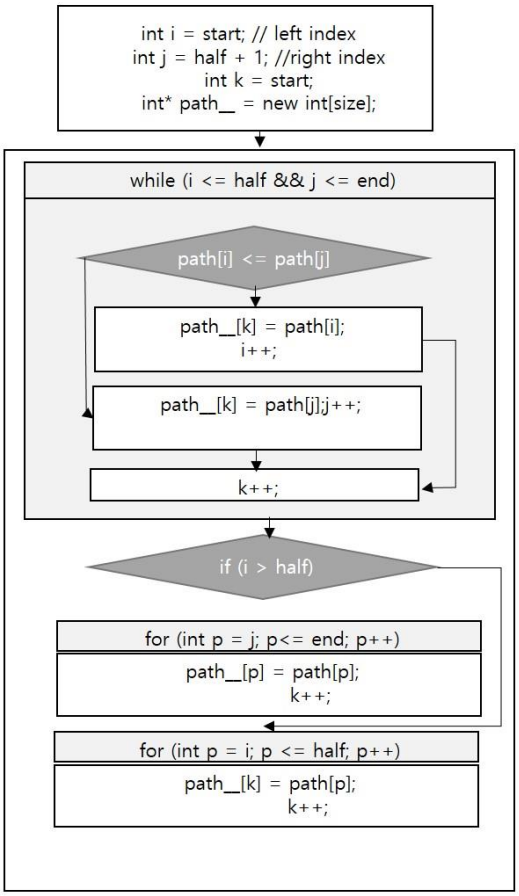
MergeSort



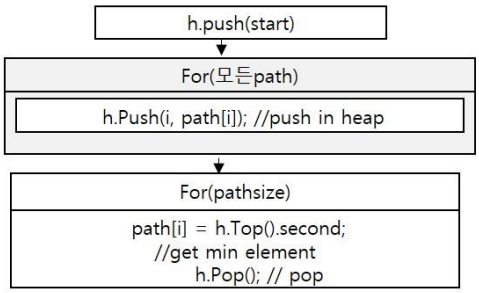
Quick



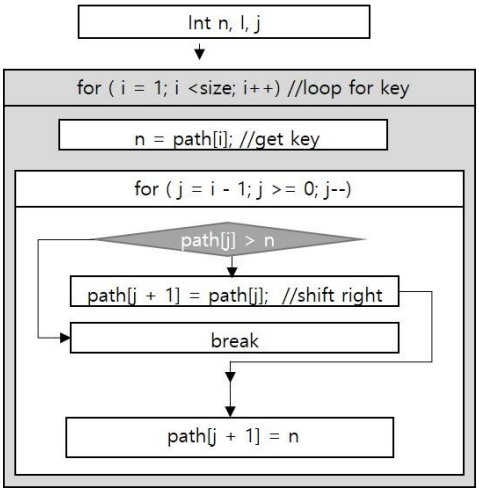
Merge_



Heap



Insert



C. 알고리즘

1) 경로 탐색 알고리즘

BFS

- Graph::FindPathBfs(int startVertexKey, int endVertexKey)

해당 프로젝트에서 가중치를 고려하지 않고, 해당 Vertex까지 도달하는데 거치는 점의 수를 기준으로 최단 경로를 도출하였다.

경로를 저장하기 위해 배열 prev, BFS 연산을 위한 큐 q, 방문 여부를 위한 bool 배열 check를 사용한다. 초기 거리 정보는 INF로 초기화한다.

q의 첫 번째 key값의 Vertex로부터, 현재 정점에서 인접한 모든 점에 대하여 탐색을 수행하는 방식이다. 만약 해당 점을 방문하지 않았다면 해당 key 값을 q에 push후 방문 했다고 체크하는 방식으로 q가 비거나, endVertexKey에 해당하는 Vertex에 도달할 때까지 반복 탐색을 수행한다. 정점을 방문할 때, parent key값을 prev에 저장함으로 경로를 저장할 수 있다.

반복 탐색을 완료한 후 endVertexKey로부터 prev 배열을 거슬러 올라가 이를 vector<int> path에 저장해 return 한다.

DIJKSTRA

- Graph::FindShortestPathDijkstraUsingSet(int startVertexKey, int endVertexKey)

경로를 저장하기 위한 배열 prev, DIJKSTRA 연산을 위한 set, 거리 정보 저장을 위한 vector dist를 선언해 사용한다. 초기 거리 정보는 INF로 초기화한다.

set의 첫 번째 요소로부터, 현재 위치에서 인접한 모든 점에 대하여 탐색을 수행하는 방식이다. 만약 (해당 점으로의 최단거리) 가 (현재 위치의 최단거리 + 해당 점의 가중치) 보다 크면 해당 점을 거쳐서 갈 경우가 거리가 기존 거리보다 더 짧은 것이기 때문에 최단 거리를 갱신하고, 전자를 set에서 제외 , 후자를 set에 추가한다. set이 빌 때까지 해당 연산을 반복하며 값을 갱신할 때 parent key값을 prev에 저장함으로 경로를 저장할 수 있다.

반복 탐색을 완료한 후 endVertexKey로부터 prev 배열을 거슬러 올라가 이를 vector<int> path에 저장해 return 한다.

DIJKSTRAMIN

- Graph::FindShortestPathDijkstraUsingMinHeap(int startVertexKey, int endVertexKey)

경로를 저장하기 위한 배열 prev, DIJKSTRA 연산을 위한 Minheap h, 거리 정보 저장을 위한 vector dist를 선언해 사용한다.

h의 첫 번째 요소로부터, 현재 위치에서 인접한 모든 점에 대하여 탐색을 수행하는 방식이다. 만약 (해당 점으로의 최단거리) 가 (현재 위치의 최단거리 + 해당 점의 가중치)

보다 크면 해당 점을 거쳐서 갈 경우가 거리가 기존 거리보다 더 짧은 것이기 때문에 최단 거리를 갱신하고, h에 후자를 push한다. 힙이 빌 때까지 해당 연산을 반복하며 값을 갱신할 때 parent key값을 prev에 저장함으로 경로를 저장할 수 있다.
반복 탐색을 완료한 후 endVertexKey로부터 prev 배열을 거슬러 올라가 이를 vector<int> path에 저장해 return 한다.

BELLMANFORD

- Graph::FindShortestPathBellmanFord(int startVertexKey, int endVertexKey)

벨만포드 연산을 쉽게 하기 위하여, 그래프의 거리 정보 리스트를 모든 vertex, edge에 대하여 순차적으로 접근하며 tuple<int,int,int>={startvertex key, endvertex key, weight} 형태로 저장한다. 경로를 저장하기 위한 배열 prev, 거리 정보 저장을 위한 vector dist를 선언해 사용한다. 초기 거리 정보는 INF로 초기화한다.
최대 (edge수 -1)번의 반복을 통해 모든 Edges 데이터를 불러와 만약 해당 edge로의 거리가 INF가 아니고, (해당 점으로의 최단 거리) 가 (소스 점의 최단 거리 + 해당 점의 가중치) 보다 크면 해당 점을 거쳐서 갈 경우의 거리가 기존 거리보다 더 짧은 것이기 때문에 최단 거리 정보를 갱신한다. 값을 갱신할 때 parent key값을 prev에 저장함으로 경로를 저장할 수 있다.
이후 다시 같은 반복문을 수행해 값의 갱신이 또 필요하다고 인식되면 음의 사이클이 존재하는 것이기 때문에 빈 vector를 선언해 error로 리턴한다.
반복 탐색을 완료한 후 endVertexKey로부터 prev 배열을 거슬러 올라가 이를 vector<int> path에 저장해 return 한다.

FLOYD

- Graph::FindShortestPathFloyd()

FLOYD 연산을 쉽게 수행하기 위해 인접 리스트로부터, 각 거리 값을 저장하는 인접 행렬을 이중 vector를 이용하여 저장해 사용한다.

모든 행과 열에 대해 반복을 수행하여, (해당 점으로의 최단 거리) 가 (소스 점의 최단 거리 + 해당 점의 가중치) 보다 크면 해당 점을 거쳐서 갈 경우의 거리가 기존 거리보다 더 짧은 것이기 때문에 최단 거리 정보를 갱신한다.

값이 변경된 dist를 return 한다.

2) 정렬 연산

Quick Sort

pivot을 기준으로 분할해, 왼쪽에서 오른쪽 방향으로 pivot 보다 큰 값을 찾고 오른쪽에선 왼쪽방향으로 pivot보다 작은 값을 찾아, 왼쪽 탐색 인덱스(i)와 오른쪽 탐색 인덱스(j) 값을 비교해 교차된 상황이라면 j번째 값과 pivot째 값을 교환하고, 그렇지 않은 상황이라면 i번째 와 j번째 값을 교환한다. 이 과정을 왼쪽 탐색 인덱스와 오른쪽 탐색 인덱스가 만날 때까지 반복하며, 왼쪽 분할에 대하여 다시 Quick Sort를 실행 및 오른쪽 분할에 대하여 다시 Quick Sort를 수행하는 것을 재귀적으로 반복한다.

Insert Sort

key를 기준으로 key보다 왼쪽에 있는 요소들을 탐색하여 key의 위치를 찾아 삽입하는 정렬방법이다.

key는 1번째 값부터 시작하며, key의 앞 요소들을 탐색하여 만약 앞 요소가 key보다 크면 해당 요소를 오른쪽으로 한 칸 미는 것을 반복하며 제대로 정렬된 위치에 도달하면 반복을 중단하고, 그 자리에 key값을 저장한다. key값은 1번째 값부터 오른쪽으로 한 칸씩 이동하여 이를 N-1번 반복하게 된다.

Heap Sort

최소 heap 이진 트리를 구성하여 정렬하는 방법으로, 해당 프로젝트에서 이미 minheap을 헤더파일로 구현해놓았기 때문에 이를 이용하여 입력 받은 배열을 heap에 모두 push 후 min 값을 차례로 불러오는 방식으로 정렬을 수행하였다.

MinHeap에서는, 첫번째 요소에서 시작하여 해당 요소의 parent 와 child에 대하여 만약 chlid가 parent보다 클 시 올바르지 않은 정렬이므로 값의 swap을 수행하는 것을 root에 도달할 때까지 반복하는 연산을, 마지막 요소까지 반복 수행하는 방식으로 Heapify를 수행한다.

Merge Sort

배열을 두 개의 균등한 크기로 분할하고 분할된 배열을 정렬 후, 정렬된 부분 배열을 다시 합하면서 전체가 정렬되게 하는 방법이다. 해당 함수에서는 정렬되는 값의 저장을 위해 path__ 배열을 새로 선언해 사용한다.

먼저 중간 위치를 계산하여 배열을 분할하고, 왼쪽 리스트에 대하여 해당 함수를 재귀적으로 호출, 오른쪽 리스트에 대하여 해당 함수를 재귀적으로 호출, 부분 배열을 합병하는 함수(merge)를 호출함으로서 분할된 배열들이 정렬되며 합쳐질 수 있도록 한다.

merge 함수에서는 왼쪽 인덱스(i)와 오른쪽 인덱스(j)를 따로 선언하여, 분할된 배열을 합병하면서 값의 비교를 거듭해 올바르게 정렬된 값을 path_에 저장한다. 이후 남아있는 값들을 일괄적으로 복사하고 path_의 값을 다시 path에 복사해 원본 배열이 정렬될 수 있도록 하였다.

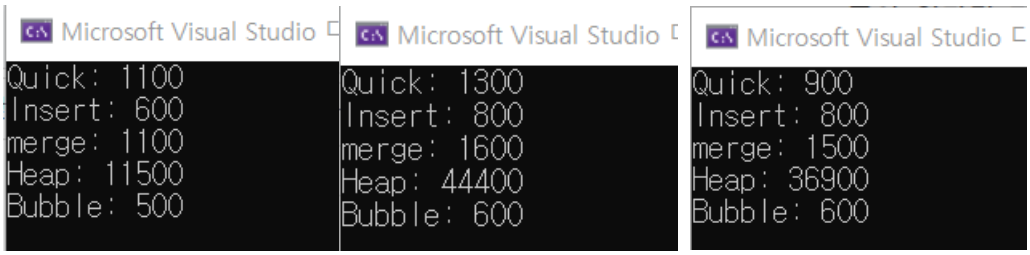
Bubble Sort

앞에서부터 마지막까지 차례로 오른쪽으로 이동하며 두 개의 값들을 순서대로 비교해가면서, 값이 올바르게 정렬되어 있지 않은 경우 값들을 교체한다. 해당 연산을 N-1번 반복하면 정렬이 완료된다.

정렬 알고리즘 성능 비교

Sort	Time1 (경로 4개)	Time2 (경로 3개)	Time3 (경로 3개)
Quick	1100	1300	900
Insert	600	800	800
Merge	1100	1600	1500
Heap	11500	44400	36900
Bubble	500	600	600

결과화면



다음 결과를 통해 해당 프로젝트의 mapdata 예시와 같이 정렬해야 하는 데이터의 개수가 적을 때, Bubble Sort, Insert Sort, Quick Sort, Merge Sort, Heap Sort의 순서로 빠른 결과를 도출할 수 있음을 확인 할 수 있었다.

해당 프로젝트에서, heap 정렬의 구현에 직접 구현한 minheap을 불러와 사용했기 때문에 소모 시간이 더 오래 걸린 것이 아닌가 추정된다.

알고리즘 별 시간 평균 시간복잡도인 Quick, Heap, Merge : O(nlogn), Bubble, Insert : $O(n^2)$의 이론 값과 같은 결과는 도출해내지 못했다.

라빈 카프 알고리즘

RabinKarpCompare(const char* CompareString, const char* ComparedString)

해당 조건의 비교를 수행하기 위해서, . 인자로 받은 문자열은 수정할 수 없으므로 문자열을 compare과 compared에 복사하고, 이를 모두 대문자 취급한다. 두 문자열의 size를 compareSize, comparedSize에 저장한다. 5문자 보다 작은 문자열이 있다면, none을 반환한다.

해당 조건에서는 다섯문자와 열문자를 기준으로 비교를 제안하고 있으므로, compared 문자열에서 앞에서부터 차례로 비교할 다섯문자를 선택하는 것을 반복하여 라빈카프 알고리즘을 수행한다. 선택된 문자열과 비교 문자열의 비교를 시작하는데 먼저 처음에는 각 문자열의 해쉬 값을 계산하고 그 이후의 루프에서는, 첫 글자의 해쉬 값을 빼고 남은 값에 2를 곱하고, 새로운 한 글자의 해쉬 값을 더해 수정을 수행한다. 두 해쉬 값이 같을 경우 해쉬 값은 같고, 문자열은 다른 경우를 고려하여 문자열의 모든 요소를 비교해 만약 다른 문자가 있을 경우 None의 결과를, 그렇지 않을 경우 Fivesame의 결과를 도출한다.

이후 열 문자의 비교도 같은 방법을 수행한다. 다섯 문자만 같을 경우 FiveSame을, 열 문자까지 같을 경우 TenSame을 리턴한다.

RabinKarpCompareW(const char* CompareString, const char* ComparedString)

간선 비용 규칙의 3번째 규칙인 단어별 비교를 위해 해당 함수를 사용한다. 인자로 받은 문자열은 수정할 수 없으므로 값을 다른 문자열에 복사하고, 이를 모두 대문자 취급한다.

문자 별 비교를 위해 첫 번째 문자열 내 단어와 두 번째 문자열 내 단어를 순서대로 모두 순회해 값을 비교하도록 한다. strtok를 통한 단어 단위 문자 도출을 위해 비교할 문자를 word1, word2에 저장, 공백 단위로 자르고 남은 부분을 nextword1, nextword2에 저장해서 계속해서 공백 단위로 문자를 자를 수 있게 하였다.

위의 라빈카프 비교 알고리즘을 word1, word2 단위의 이중 loop를 반복하며 진행하여 같은 단어가 있을 시 반복을 멈추고 Wordsame을 리턴한다.

D. 실행결과

1) test case1

Command.txt	mapdata.txt
LOAD mapdata.txt UPDATE PRINT CONFIG -sort bubble BFS 0 3 DIJKSTRA 0 3 DIJKSTRAMIN 0 3 BELLMANFORD 0 3 FLOYD	6 Denny's Bread / 0 6 13 0 0 1 Chodenny's Car shop / 0 0 5 6 0 5 Granddey's Go Kicking / 2 0 0 7 4 7 Labuajiea's Yoga class / 0 6 0 0 3 5 Grandy's Kicking Class / 0 0 5 2 0 9 Jeawon's Computer Academy / 5 6 8 4 5 0

Data.txt

```
====== LOAD ======
Success
==================
================
Error code: 0
================
====== UPDATE ======
Success
==================
================
Error code: 0
================
====== PRINT =======
0 6 13 0 0 1
0 0 5 6 0 5
2 0 0 7 4 7
0 6 0 0 3 5
0 0 4 2 0 9
5 6 8 4 5 0
======================
================
Error code: 0
================
====== CONFIG LOG ======
Sorted by : bubble Sorting
========================
================
Error code: 0
================
===== BFS =====
order: 0 1 3
sorted nodes: 0 1 3
path length: 12
Course :
======================
================
Error code: 0
================
================
===== DIJKKSTRA =====
order: 0 5 3
sorted nodes: 0 3 5
path length: 5
Course :
======================
================
Error code: 0
================
===== DIJKKSTRAMIN =====
order: 0 5 3
sorted nodes: 0 3 5
path length: 5
Course :
======================
================
Error code: 0
================
===== BELLMANFORD =====
order: 0 5 3
sorted nodes: 0 3 5
path length: 5
Course :
======================
================
Error code: 0
================
====== FLOYD ======
0 6 9 5 6 1
7 0 5 6 9 5
2 8 0 6 4 3
9 6 7 0 3 5
6 8 4 2 0 7
5 6 8 4 5 0

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

2) test case2

Command.txt	mapdata.txt
DIJKSTRA 2 5 CONFIG -albet bubble LOAD mapdata.txt UPDATE PRINT CONFIG -sort bubble BFS 0 3 DIJKSTRA 0 3 DIJKSTRAMIN 0 3 BELLMANFORD 0 3 FLOYD AA //PRINT BFS 0	6 Denoodenny's Bread shop / 0 26 13 21 20 1 Choooodenny's Car Go shop / 50 0 15 6 51 5 Granddey's Go Kicking / 2 34 0 7 44 7 Labuajiea's Yoga class / 0 62 0 0 3 5 Grandy's Kicking Class / 0 10 52 24 0 9 Jeawon's Computer GO Academy / 5 654 8 4 56 0

Data.txt

```
================
Error code: 202
================
```

로드된 데이터 없음

```
================
Error code: 3
================
```

올바르지 않은 CONFIG 설정

```
====== LOAD ======
Success
==================
================
Error code: 0
================
====== UPDATE ======
Success
==================
================
Error code: 0
================
====== PRINT =======
0 20 13 21 20 1
36 0 15 6 51 4
2 28 0 7 40 6
0 62 0 0 3 5
0 10 38 20 0 9
5 524 8 4 56 0
======================
================
Error code: 0
================
====== CONFIG LOG ======
Sorted by : bubble Sorting
========================
================
Error code: 0
----------------
```

```
===== BFS =====
order: 0 3
sorted nodes: 0 3
path length: 21
Course :
======================
================
Error code: 0
================
===== DIJKKSTRA =====
order: 0 5 3
sorted nodes: 0 3 5
path length: 5
Course :
======================
================
Error code: 0
================
===== DIJKKSTRAMIN =====
order: 0 5 3
sorted nodes: 0 3 5
path length: 5
Course :
======================
================
Error code: 0
================
===== BELLMANFORD =====
order: 0 5 3
sorted nodes: 0 3 5
path length: 5
Course :
======================
================
Error code: 0
================
```

간선 비용이 수정된

올바른 결과 값

```
====== FLOYD ======
0 18 9 5 8 1
9 0 12 6 9 4
2 20 0 7 10 3
10 13 13 0 3 5
14 10 17 13 0 9
5 17 8 4 7 0

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

```
================
================
Error code: 300
================
================
Error code: 200
================
```

잘못된 명령어

인자 값 부족

E. 고찰

이번 프로젝트에서는 거리정보를 저장하는 링크드 리스트의 구현 및 이를 기반으로 한 경로 탐색 프로그램의 설계와 라빈카프 알고리즘의 구현을 수행하였다. 구현해야 할 부분이 많아 모든 예외 상황을 적용시키는 것이 쉽지 않았던 프로젝트 였다.

1) 라빈카프 알고리즘의 구현에서의 고찰

3번째 규칙인 단어 간의 라빈카프 비교를 구현하는데 여러 가지 문제점이 있었다. 두 문자열문자간의 모든 단어의 쌍을 비교하기 위하여, 각 문자열 내 단어를 첫 단어부터 마지막 단어까지 모든 쌍을 이중 반복문을 통해 비교하고자 했으나 이를 구현하기 쉽지 않았다. 문자열을 단어 단위로 나누는데 strtok를 사용하였는데, strtok 함수는 첫번째 인자로 문자열의 주소값이 들어오면 그 위치로부터 분리를 시도하고, NULL이 들어오면 이전에 분리한 문자열의 분리지점 바로 다음 주소 값을 기억해 두었다 그 위치로부터 분리를 시도하는 특성을 갖고 있다. 때문에 두 문자열을 동시에 strtok를 통해 다음 단어로 넘어가게 하는 것이 불가능했다. 이를 해결하기 위해 문자열을 자를 때 그 뒷부분을 널 단위로 분리해 또 다른 변수에 저장해 다음 strtok의 인자로 사용하는 방법이다. 이 방법을 수행하면 strtok를 실행하는 동시에 널문자가 사라지기 때문에 해당 작업의 반복을 위해서는 마지막에 널문자를 삽입해야 했다. 두번째로 고안한 방법은 처음부터 여러 개의 문자열을 선언해놓고, 처음 문자열을 받을 때 모든 단어를 다른 문자열로 저장하는 방법이다. 그러나 이 방법을 수행하면 준비된 문자열의 개수보다 단어의 개수가 많을 때 문제가 발생하며, 메모리의 낭비가 소모된다고 생각되어 꽤나 복잡하게 코드가 설계되었지만 첫번째 방법을 사용했다. 더 손쉽게 공백 단위로 문자열을 활용할 방법이 없는지 고안해보아야겠다.

또한 해당 라빈카프 알고리즘에서는 해쉬 값은 같고 문자열이 다른 경우를 고려해 해쉬 값이 같더라도 다시 한번 문자열의 비교를 for문을 통해 수행해 주었다.

이번 프로젝트에서는 다섯문자와 열문자 단위로 문자열의 비교가 필요했는데, 라빈카프 알고리즘을 통해서 이를 간단하게 해결하는 방법으로 비교되는 문자열 내 다섯 글자, 열 글자를 처음부터 끝까지 한 칸씩 밀어가며 반복 선택해 이 패턴이 비교하는 문자열 내에 존재하는지 비교하는 것으로 이 문제를 해결했다. 다만 다섯 문자와 열 문자를 비교하는 코드가 동일한데 각각의 경우를 따로 작성하는 것이 낭비가 되었다고 판단되어, 만약 함수의 인자 값으로 몇 글자를 비교할지 설정할 수 있었다면 더 좋았을 것 같다.

2) Sort

Heap Sort의 경우 Min 이진 트리를 활용하여 숫자를 정렬하는 방법이다. 해당 프로젝트에서는 다익스트라 알고리즘의 구현에서 MinHeap.h을 구현해 사용하였기 때문에 코드의 재활용도를 높이고자 Minheap의 힙 객체를 선언해 힙 내에 숫자를 모두 push 하여 정렬된

heap의 min 값을 차례로 불러오는 방식으로 정렬을 수행하였는데, 이 방법으로 코드를 설계한 결과 Heap sort의 시간 결과값이 오래 걸리게 측정된 것이 아닌가 하는 생각이 든다. 이론 상으로 배운 Sort의 시간 복잡도와 나의 코드 결과 값이 차이가 있었는데 정확한 원인은 알아낼 수 없었다. 해당 프로젝트 내의 정렬 데이터의 경우 그 수가 너무 적기도 하고, 또한 단순히 코드의 실행만을 수행하는 것이 아니라 각 소트 별로 함수를 선언하여 함수에 인자를 전달하는 과정 등의 연산이 함께 수행되는데, 각 함수 별로 전달되는 인자 값의 량 등의 조건이 다르게 설정되어 결과에 오차가 생긴 것 일수도 있겠다고 판단하였다.

3) 경로탐색

해당 프로젝트를 통해 자료구조 수업에서 배운 여러가지 알고리즘을 직접 구현할 수 있었다. 다익스트라 알고리즘의 경우 우선순위 큐를 이용하여 구현하는 방법에 대해서만 알고 있었기 때문에 STL set을 이용하여 구현하는 방법에 대해서 고민이 필요했다. STL set을 이용해 이를 구현해보니, set을 사용하면 큐와 유사하게 동작하면서 값을 갱신하는 것이 가능하기 때문에 오히려 더 편하게 구현할 수 있음을 깨달았다.

또한 BELLMANFORD 연산의 경우 음수 값인 weight가 있어도 원활하게 작동하는 것이 알고리즘의 핵심 기능인데, 결과를 확인해보니 특정 음수 값들이 weight로 저장되어 있는 경우 어디선가 오류가 발생함을 확인되었지만 어디서 문제가 발생한지 수정하지 못하고 프로젝틀 제출하게 되었다. 다시 해당 알고리즘을 재점검할 필요가 있음을 인지하고 프로젝트에서 못다한 부분을 더 보충하는 시간을 가져야겠다.

4) 리눅스 상의 컴파일 오류

윈도우 환경에서는 제대로 작동되는 코드가 리눅스 환경에서 컴파일 오류가 발생하는 경우가 종종 있었다. 대부분의 경우 헤더파일의 호환 문제 였기 때문에, 리눅스 환경에서 코드를 컴파일 할 때 헤더 파일을 꼼꼼히 체크하여 문제가 발생하지 않도록 주의해야겠다. 또한 윈도우 환경에서는 오류로 나타나지 않는, segment fault의 경우 대부분 메모리 해제가 제대로 이루어지지 않아 발생하는 문제였다. 동적 할당을 통해 메모리를 사용한 경우 꼭 제대로 이를 해제함을 유의해야 겠다.