Data Structure Lab. Project #3

제출일자: 2017년 12월 08일 (금)

학 과: 컴퓨터공학과

담당교수: 이기훈 교수님

실습분반: 화요일 3,4

학 번: 2013722004

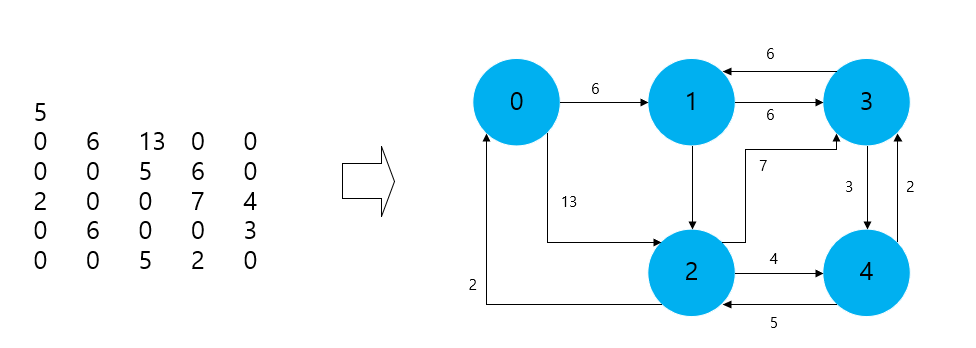
성 명: 최민기

**Introduction**

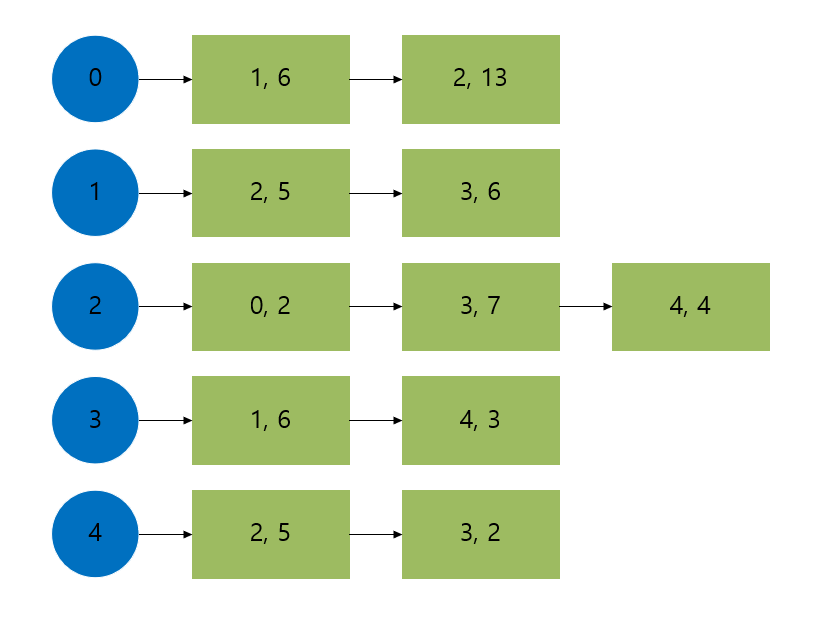
**미로 맵 데이터**

미로 맵 데이터 프로그램은 방향성과 가중치를 가지고 있는 그래프 정보를 Matrix 형태로 미로의 맵 데이터를 저장한 Text 파일을 LOAD 명령어를 통해 정보를 가져와 Graph 클래스에 저장한다. 미로 맵 데이터 텍스트 파일의 첫 번째 줄에는 미로의 크기가 저장되어 있고, 다음 줄에 미로의 맵 데이터가 저장되어 있다. 미로 맵 데이터의 행과 열은 각각 Edge의 시작 Vertex와 끝 Vertex의 Weight를 의미한다. 모든 Vertex 값은 0 이상의 정수이며, Vertex와 Vertex가 연결되어 있지 않다면 Weight는 0이다. 반대로 연결되어 있는 경우 Weight는 음수와 양수를 가질 수 있다.

**Graph**



위의 그림은 미로 맵 데이터가 저장되어 있는 텍스트 파일과 해당 텍스트 파일의 연결 예시이다. 첫번째 줄의 미로의 크기가 저장되어 있고, 순서대로 각 Vertex에 연결된 Edge들의 Weight들이 저장되어 있다.



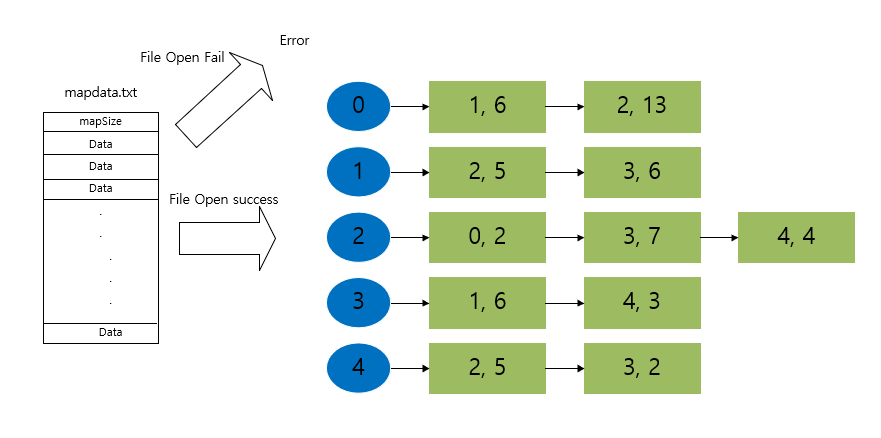
위의 그림은 Graph 클래스 내부에 존재하는 Linked List 연결 예시이다. 각 Vertex별로 연결되어 있는 Edge들의 Vertex와 Weight가 Linked-List와 오름차순 형태로 연결되어 있다.

**Program Command & Function**

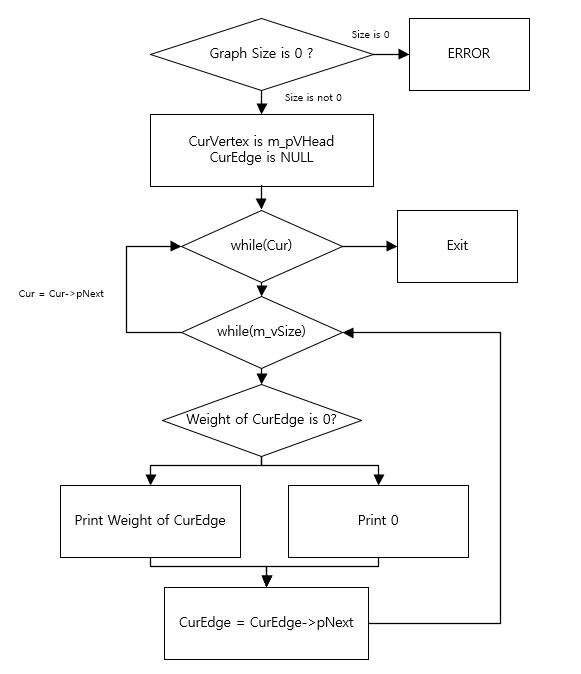
|  |  |
| --- | --- |
| **Command** | **Function** |
| RUN | Command.txt 파일을 읽어서 Text내에 존재하는 명령어들을 읽어와 해당하는 command를 호출한다. |
| LOAD | 맵 데이터가 저장되어 있는 Text file을 읽어 맵 데이터 정보를 불러와 그래프를 구성한다. 텍스트 파일이 존재하지 않을 경우, 해당하는 error code를 출력한다. |
| PRINT | 미로 맵 데이터를 읽어 미로 Map을 출력한다. Matrix형태로 미로 Map을 출력한다. 만약 미로 Map데이터가 존재하지 않을 경우 해당하는 error code를 출력한다. |
| DFS | DFS 명령어는 인자를 2개 가지고 실행 된다. 첫 번째 인자는 start Vertex에 해당하고, 두 번째 인자는 End Vertex에 해당한다. Start Vertex를 기준으로 DFS 알고리즘을 수행하여 End Vertex까지의 경로와 거리를 구한다. 만약 인자 개수가 2개 보다 적거나, Graph가 구성되어 있지 않거나, 미로 map 데이터에 음수가 존재하거나, 인자로 들어온 Vertex들이 존재하지 않을 때 각각 해당하는 error code를 출력한다. |
| DIJKSTRA | DIJKSTRA 명령어는 인자를 2개 가지고 실행 된다. 첫 번째 인자는 start Vertex에 해당하고, 두 번째 인자는 End Vertex에 해당한다. Start Vertex를 기준으로 DIJKSTRA 알고리즘을 수행하여 End Vertex까지의 경로와 거리를 구한다. 만약 인자 개수가 2개 보다 적거나, Graph가 구성되어 있지 않거나, 미로 map 데이터에 음수가 존재하거나, 인자로 들어온 Vertex들이 존재하지 않을 때 각각 해당하는 error code를 출력한다. |
| DIJKSTRAMIN | DIJKSTRAMIN 명령어는 인자를 2개 가지고 실행 된다. 첫 번째 인자는 start Vertex에 해당하고, 두 번째 인자는 End Vertex에 해당한다. Start Vertex를 기준으로 DIJKSTRAMIN 알고리즘을 수행하여 End Vertex까지의 경로와 거리를 구한다. 만약 인자 개수가 2개 보다 적거나, Graph가 구성되어 있지 않거나, 미로 map 데이터에 음수가 존재하거나, 인자로 들어온 Vertex들이 존재하지 않을 때 각각 해당하는 error code를 출력한다. |
| BELLMANFORD | BELLMANFORD 명령어는 인자를 2개 가지고 실행 된다. 첫 번째 인자는 start Vertex에 해당하고, 두 번째 인자는 End Vertex에 해당한다. Start Vertex를 기준으로 BELLMANFORD 알고리즘을 수행하여 End Vertex까지의 경로와 거리를 구한다. 만약 인자 개수가 2개 보다 적거나, Graph가 구성되어 있지 않거나, 인자로 들어온 Vertex들이 존재하지 않을 때 각각 해당하는 error code를 출력한다. |

**Flow Chart**

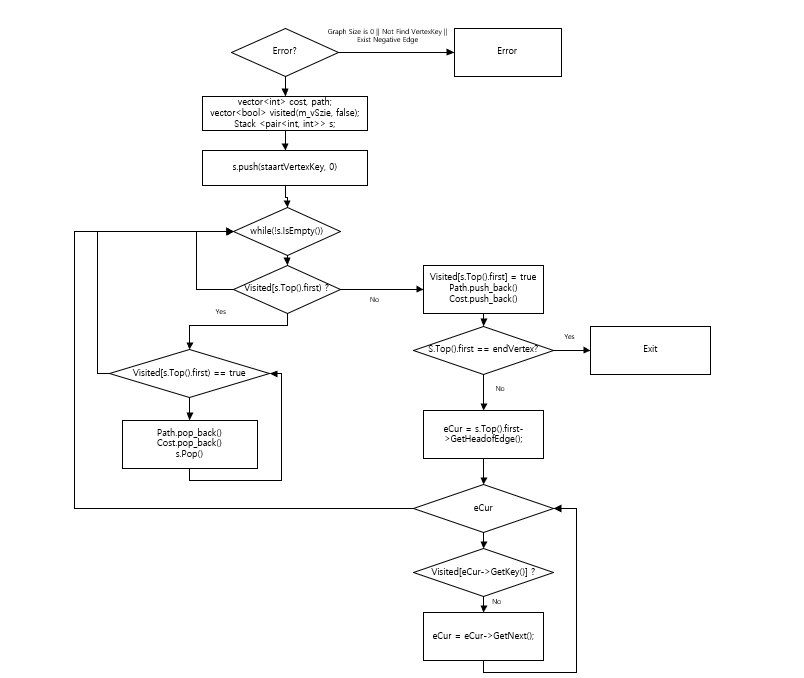
**LOAD**



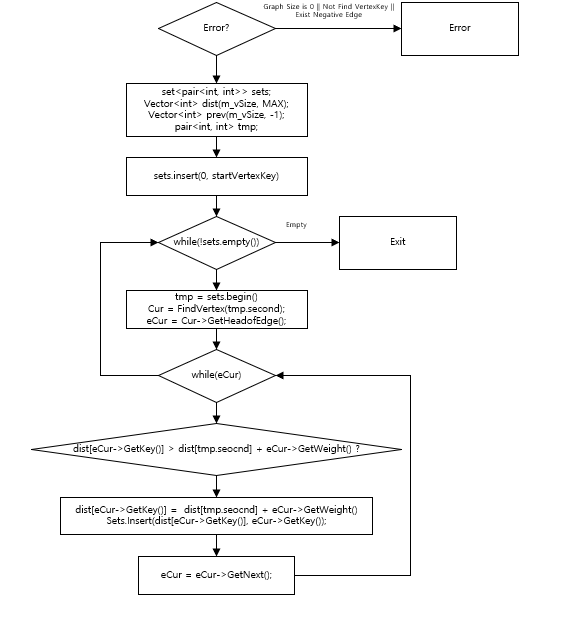
**PRINT**



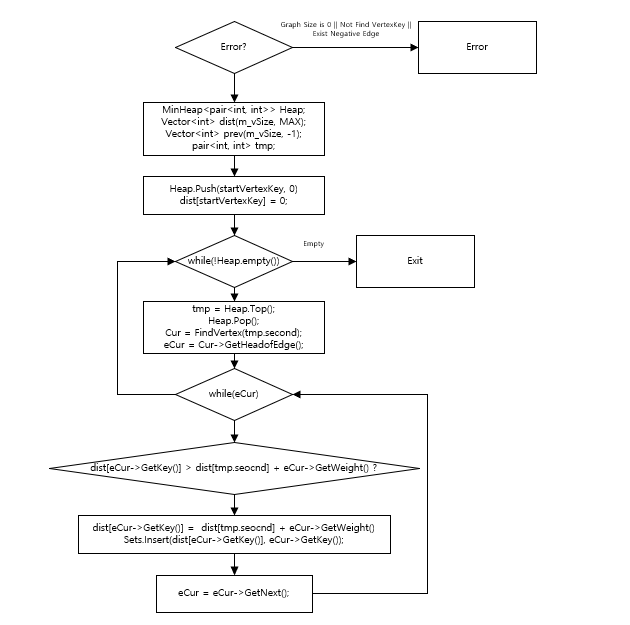
**DFS**



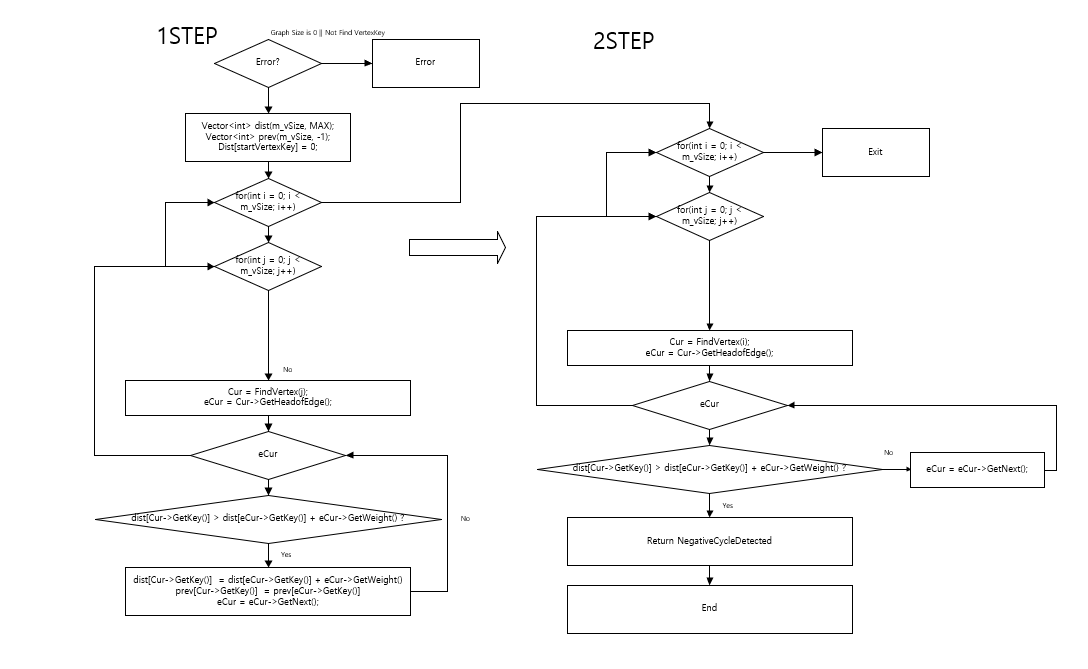
**DIJKSTRA**



**DIJKSTRAMIN**



**BELLMANFORD**



**Algorithm**

1. **Run**

command.txt 파일을 불러와 fin.getline()을 통해 차례대로 내용을 받아온다. 받아 온 인자를 cpy\_arg에 저장하여, strtok()을 사용해 인자 개수를 먼저 계산한다. 이후 strtok()을 사용해 token을 분리해서 Command에 따라서 각 기능을 수행하는 명령어 함수를 호출한다. 만약 인자개수가 적을 때 Error를 출력해야 한다면 아까 계산했던 인자 개수를 사용해서 Error를 출력한다.

1. **LOAD**

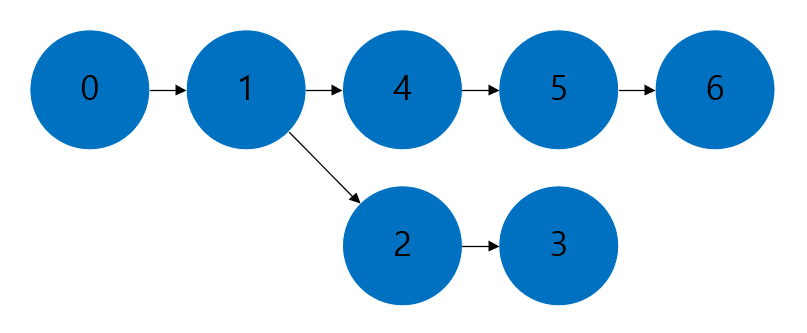
미로 정보가 들어있는 mapdata.txt 파일을 읽어온다. 만약 파일 Open 실패 시, Error를 return한다. File Open 성공 시, file의 끝일 때까지 while문을 수행하면서 getline()을 통해 차례대로 Edge들의 정보를 받아온다. map data에 첫 줄에 있는 map Size만큼 for문을 수행한다. for문에서 사용하는 변수는 0부터 map Size만큼 증가하면서 수행한다. 해당 변수는 0부터 map Size만큼 증가하기 때문에 Edge의 Key 인자로 사용되고, 처음의 int형 변수를 두어서 맨 첫 줄을 제외하고, getline()으로 한 줄을 받아올 때 마다, Count를 증가시켜줘서 Start Vertex의 인자로 사용된다. for문을 수행하면서 Vertex가 Graph에 존재하지 않을 경우, Vertex를 추가하고, Weight가 0이 아닐 경우, AddEdge 함수를 사용해 Edge를 추가한다.

1. **PRINT**

먼저 Graph의 Size가 0인 경우, Error를 출력한다. Size가 0이 아닐 경우, Print() 함수를 호출한다. 먼저 CurVertex를 m\_pVHead를 가리키게 하고, CurEdge는 NULL을 가리킨다. CurVertex가 NULL이 될 때까지, while문을 수행한다. while문 안에서는 먼저 Cur->GetHeadofEdge() 함수로 eCur이 그것을 가리키게 한다. 그리고 변수 i를 두어서 0부터 변수를 증가시키면서 m\_vSize 보다 작으면 for문을 수행한다. for문 안에서 CurEdge가 NULL이면 Edge가 없는 것이기 때문에 모두 0을 출력한다. 또한 CurEdge가 존재하지만, CurEdge->GetKey()가 i가 아니라면 0을 출력하게 했다. 만약 i가 CurEdge->GetKey()와 같다면 Matrix형태로 출력해야 하므로 출력하는 부분을 찾았으므로 eCur->GetWeight()를 출력한다.

1. **DFS**

먼저 Stack에 pair를 사용하여, first에는 Vertex Key를, Second에는 Weight를 넣어준다. 그리고 Path를 저장할 Vector와 거리를 저장할 Vector와 방문을 했는지 안 했는지 여부를 판단 할 Vector를 사용한다. Stack에 시작 Vertex와 0값을 처음에 넣고 시작한다. 그리고 Stack이 빌 때까지, while문을 수행한다. While문 내부에서는 visit Vector를 사용해 스택의 Top의 first(Vertex Key)를 아직 방문하지 않았을 경우 true로 바꿔주고, Path Vector와 Cost Vector에 스택의 Top의 first와 second를 각각 저장한다. 그리고 스택의 Top의 first(Vertex Key)가 끝 Vertex라면 while문을 탈출한다. 그게 아니라면 s.Top().first(Vertex Key)에 연결된 모든 Edge들을 탐색 해, 만약 해당 Vertex에 방문하지 않았다면 Stack에 Push한다. Push되지 않은 경우와 길이 막힌 경우 해당 Vertex의 visit이 아직 True이기 때문에 방문하지 않은 곳까지 찾기 위해 while문을 수행하여 Path와 Cost와 Stack에서 Pop해주면서 다시 경로를 찾게 했다. 경로는 Path에 저장되어 있고, Path 뒤에 Cost Vector를 사용해 총 거리를 계산해서 뒤에 Push해주었다. 아래 예시를 보면 이해하기 편리하다.



Map Data가 다음과 같이 연결되어 있고, 각 Weight는 1이라 하고 DFS 0 3을 진행하면 다음과 같이 진행된다.

**Initailize**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Stack | | | | (0, 0) | | | |
| Path | | | | (0) | | | |
| Cost | | | | (0) | | | |
| Visited | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| bool | F | F | F | F | F | F | F |

**1STEP**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Stack | | | | (0, 0), (1, 1) | | | |
| Path | | | | (0), (1) | | | |
| Cost | | | | (0), (1) | | | |
| Visited | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| bool | F | T | F | F | F | F | F |

**2STEP**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Stack | | | | (0, 0), (1, 1), (2, 1), (4, 1) | | | |
| Path | | | | (0), (1), (4) | | | |
| Cost | | | | (0), (1), (1) | | | |
| Visited | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| bool | F | F | F | F | T | F | F |

**3STEP**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Stack | | | | (0, 0), (1, 1), (2, 1), (4, 1), (5, 1) | | | |
| Path | | | | (0), (1), (4), (5) | | | |
| Cost | | | | (0), (1), (1), (1) | | | |
| Visited | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| bool | F | F | F | F | T | T | F |
|  |  |  |  |  |  |  |  |

**4STEP**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Stack | | | | (0, 0), (1, 1), (2, 1), (4, 1), (5, 1), (6, 1) | | | |
| Path | | | | (0), (1), (4), (5), (6) | | | |
| Cost | | | | (0), (1), (1), (1), (1) | | | |
| Visited | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| bool | F | F | F | F | T | T | T |

추가 할 Edge가 없어서 s.Top()의 visited를 확인했을 때, 아직 True이기 때문에 false를 찾을 때 까지 Path와 Cost Vector, Stack에서 Pop을 해준다.

**5STEP**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Stack | | | | (0, 0), (1, 1), (2, 1) | | | |
| Path | | | | (0), (1) | | | |
| Cost | | | | (0), (1) | | | |
| Visited | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| bool | F | F | T | F | T | T | T |

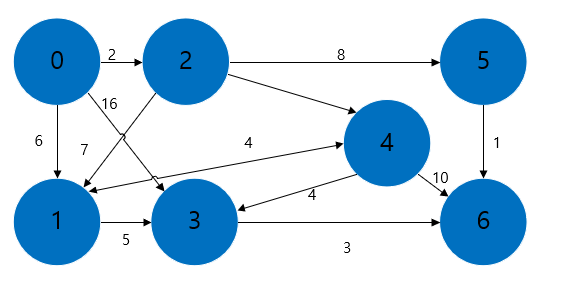
**6STEP**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Stack | | | | (0, 0), (1, 1), (2, 1), (3, 1) | | | |
| Path | | | | (0), (1), (2), (3) | | | |
| Cost | | | | (0), (1), (1), (1) | | | |
| Visited | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| bool | F | F | T | T | T | T | T |

End Vertex를 찾았기 때문에 while문을 탈출한다. 따라서 각 Path와 Total Cost를 계산해서 Path를 return하고 끝난다.

1. **DIJKSTRA**

STL set을 사용하여 구현한다. 먼저 dist(m\_vSzie, INT\_MAX)를 사용해서 거리를 모두 MAX값으로 초기화 한다. 그리고 prev(m\_vSize, -1)를 사용해서 어느 경로에서 왔는지에 대한 정보를 나타내는 prev vector를 초기화 해준다. 먼저, set에 pair로 0과 시작 Vertex를 넣어준다. Set이 빌 때까지 while문을 수행한다. set에는 기본적으로 grater와 less를 안 쓰면 pair형태로 삽입했을 때, first값 기준으로 작은 것부터 정렬되어 삽입된다. 따라서, 빈 pair형 tmp에 set의 begin()을 저장해놓고, set에서 begin()을 erase한다. tmp에서 Vertex Key를 가져와 연결된 Edge들을 모두 탐색한다. 연결된 Edge의 dist보다 tmp에서 가져온 Vertex Key에 Edge의 Weight를 더한 값이 더 작다면 tmp에서 가져온 Vertex Key에 대한 거리를 작은 값으로 초기화 해준다. Distance 정보가 수정되었기 때문에 다시 set에 해당 Distance와 Vertex Key를 Insert한다. 예시를 표현하면 다음과 같다.



다익스트라 알고리즘을 사용하여 Dist가 최단 경로로 바뀌는 예시이다. Start Vertex는 0이다.

**Initialization**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Dist | INT\_MAX | INT\_MAX | INT\_MAX | INT\_MAX | INT\_MAX | INT\_MAX | INT\_MAX |
| Visit | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| path | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

**1STEP**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Dist | 0 | INT\_MAX | INT\_MAX | INT\_MAX | INT\_MAX | INT\_MAX | INT\_MAX |
| Visit | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| path | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

먼저, start Vertex에서 Dist와 Visit을 바꿔준다.

**2STEP**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Dist | 0 | 6 | 2 | 16 | INT\_MAX | INT\_MAX | INT\_MAX |
| Visit | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| path | -1 | 0 | 0 | 0 | -1 | -1 | -1 |

Start Vertex에 연결된 Edge들의 Dist와 Path를 Update해준다.

**3STEP**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Dist | 0 | 6 | 2 | 16 | 5 | 10 | INT\_MAX |
| Visit | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| path | -1 | 0 | 0 | 0 | 2 | 2 | -1 |

거리가 가장 짧은 Edge에 대해서 Update 과정이 일어난다. Set에서 거리에 대해서 오름차순으로 정렬되어 있기 때문에 begin()에 대해서 알고리즘을 수행한다.

**4STEP**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Dist | 0 | 6 | 2 | 9 | 5 | 10 | 15 |
| Visit | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| path | -1 | 0 | 0 | 4 | 2 | 2 | 4 |

거리가 가장 짧은 Edge에 대해서 Update 과정이 일어난다. 이 과정을 set에 begin()을 보면서 진행하면서 모든 Vertex에 방문할 때까지 진행한다. 찾으려고 하는 End Vertex에 대해서 최단 거리는 End Vertex에 Path Vector를 따라가면 얻을 수 있다.

1. **DIJKSTRAMIN**

Min Heap을 사용하여 구현한다. 위에서 설명했던 DIJKSTRA 알고리즘을 그대로 사용하며, STL set 대신 Min Heap을 사용하여 구현 했다. Min Heap에서는 거리에 대해서 Min Heap구조를 갖게 했다. Set에서처럼 Distance를 기준으로 Min Heap을 사용해서 낮은 것부터 정렬했기 때문에 Heap에 쌓이는 것뿐이지, 진행되는 알고리즘은 동일하다.

1. **Min Heap**

**생성자**

먼저 Min Heap에서 생성자로 Min Heap을 처음 구성하면, Min Heap내부에서 m\_vec (Vector)를 가지고 운영하는데, m\_vec[0]에 INT\_MAX값이 들어가서 0번은 사용하지 않는다.

|  |
| --- |
| INT\_MAX |

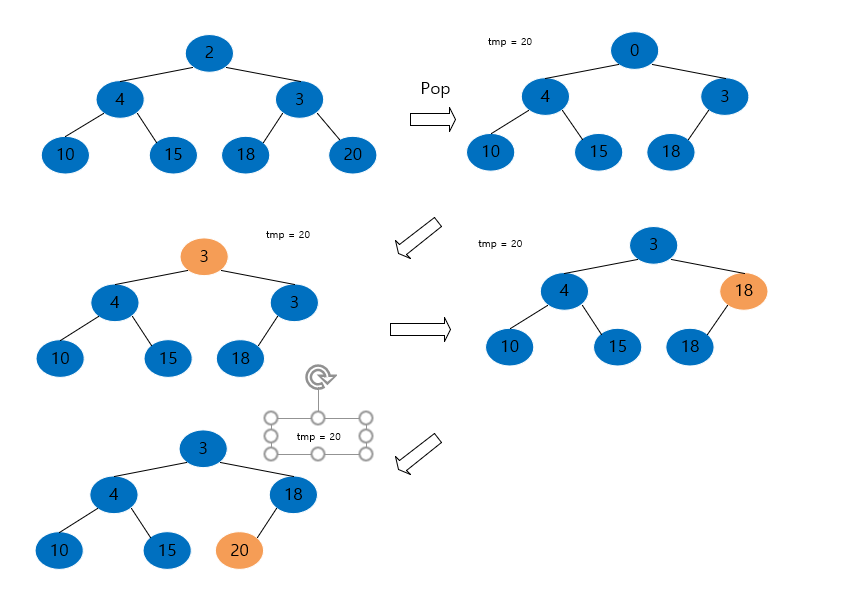
**소멸자**

m\_vec.clear()를 호출해서 vector의 원소 값들을 다 지워준다.

**Push**

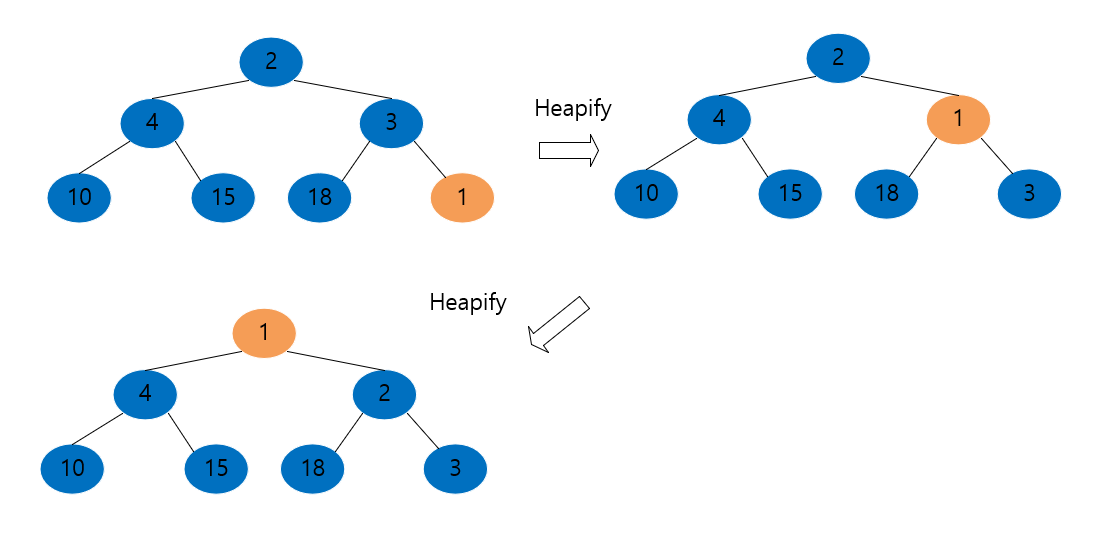
Push 과정이 일어나면 먼저 m\_vec.push\_back(Key, Value)를 사용하여 먼저 Vector 뒤에 추가한다. Min Heap구조를 이뤄야 하기 때문에 해당 삽입된 위치에 부모의 Index를 가지고 Heapify함수를 호출한다. 만약 Push했을 때, m\_vec.size() -1 == 1이라면 Heapify를 호출하지 않아도 상관 없다. 왜냐하면 처음 Push과정에서는 다시 정렬할 필요가 없기 때문이다.

**Pop**



먼저 m\_vec.size()가 1이면, 원소가 하나도 없기 때문에 그냥 return한다. m\_vec.size()가 2라면 원소가 하나밖에 없기 때문에 그냥 m\_vec.pop\_back()을 진행해서 지워준다. 그게 아닌 경우 삭제하고 정렬이 필요하다. 먼저 m\_vec[1]에 원소 값들을 초기화 시켜준다. 그리고 temp값에 m\_vec에 마지막 원소를 저장한다. 그리고 vector에서 pop\_back()으로 마지막 원소를 지운다. Cur이 1(root)를 가리키게 하고, Child는 2(자식)을 가리키게 한다. 그리고 while문을 수행한다. while문은 Child <= m\_vec.size() -1만큼 수행한다. 이 마지막 값이 들어갈 위치를 찾는 알고리즘이 사용되는데, Min Heap구조이기 때문에 child와 child +1과 비교해서 작은 쪽으로 Child를 위치시킨다. temp값이 들어가는 위치를 찾으면 break하고, 아닐 경우 child는 Up되고, Cur 포인터는 내려가면서 수행한다. while문을 탈출해서 temp가 들어가야 하는 위치에 삽입한다.

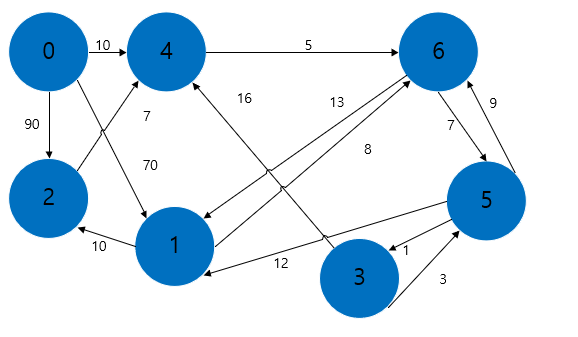
**Heapify**



Heapify함수에서는 Min Heap구조로 만들어주는 역할을 한다. 먼저 Cur은 받아온 인덱스를 가리키고, Left는 Cur \* 2, Right는 Cur \* 2 + 1을 가리킨다. 먼저 Right Child가 있는 경우와 없는 경우로 나눠진다. Right Child가 없는 경우 Left Child와 Cur을 비교해서 Left가 작은 경우 Left와 Cur을 Swap하고 끝낸다. Right Child가 존재하는 경우 Left Child와 Right Child와 비교해서 작은 값을 Cur과 Swap해준다. Cur Index가 1이면 root까지 탐색하면서 recursive하게 진행된 것이기 때문에 return한다. 아닌 경우 index값을 2로 나눠서 부모의 index를 가지고 다시 Heapify를 recursive하게 호출한다. 트리구조에서 Push된 부분의 부모와 자식들을 보면서 작은 값으로 변경시켜 주면서 위로 쭉쭉 올라가는 형태이다.

1. **BELLMANFORD**

먼저 최단 BELLMANFORD 알고리즘은 distance와 이전 경로를 저장하는 path Vector를 가지고 진행한다. BELLMANFORD 알고리즘은 처음 Start Vertex Key에 대한 Distance 값을 0으로 초기화 한다. 그리고 전체적인 알고리즘은 Vertex 개수인 m\_vSize만큼 진행한다. 그 안에서는 m\_vSize만큼 한번 더 진행한다. 이 안쪽에서는 0번 Vertex부터 m\_vSize만큼 하나씩 증가시키면서 각 Vertex에 대해서 연결된 Edge들에 거리 정보를 Update를 진행한다.



Start Vertex는 0이라고 가정한다.

**Initialization**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Dist | 0 | INT\_MAX | INT\_MAX | INT\_MAX | INT\_MAX | INT\_MAX | INT\_MAX |
| path | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

먼저 start Vertex Key에 대해서 Distance를 0으로 만들어 주어 Initialization을 한다.

**STEP 1**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Dist | INT\_MAX | 70 | 90 | INT\_MAX | 10 | INT\_MAX | INT\_MAX |
| path | -1 | 0 | 0 | -1 | 0 | -1 | -1 |

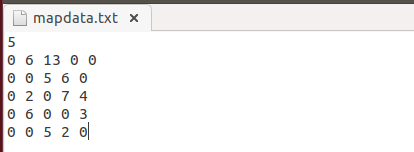
0에 연결된 Edge들의 Vertex들에 대해서 거리가 변경된 것을 확인할 수 있다. 0번 Vertex에 대해서 진행했으므로 1, 2, 3, … 6 번 Vertex에 대해서 수행한다. 이 수행 횟수를 Vertex Size만큼 진행하면 최단 경로를 얻을 수 있다.

**STEP 2**

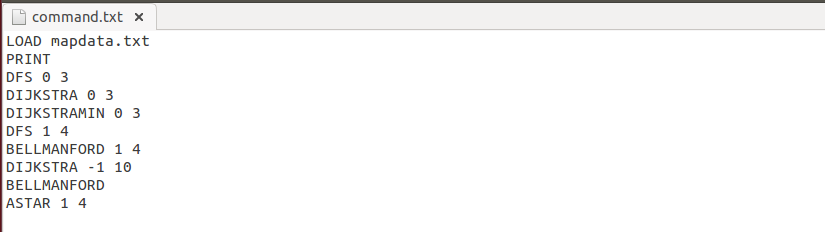
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Dist | INT\_MAX | 70 | 80 | INT\_MAX | 10 | INT\_MAX | 78 |
| path | -1 | 0 | 1 | -1 | 0 | -1 | 1 |

이 과정을 계속 반복하면서 진행한다. 그리고 이 알고리즘을 한번 더 적용해서 Dist가 Update 되면 Negative Cycle이 존재하는 것이다.

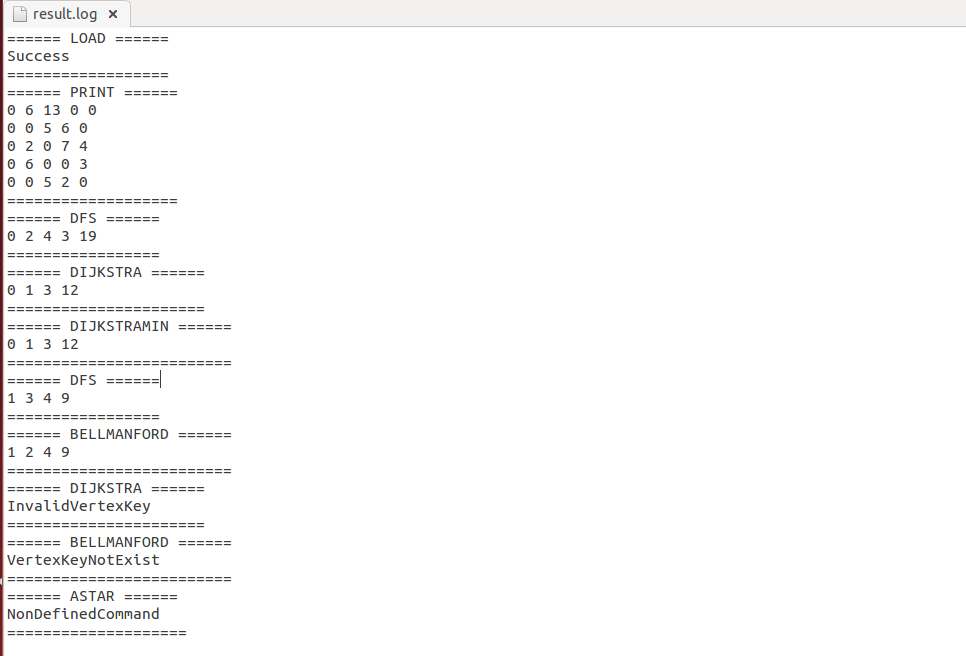
**Result Screen**



map data에 정보가 들어있는 Text 파일의 캡쳐 화면이다.



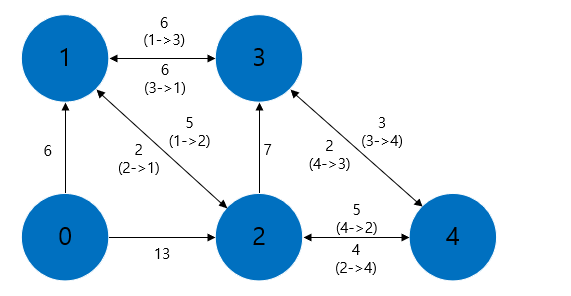
Command 정보가 들어있는 Text 파일의 캡쳐 화면이다.

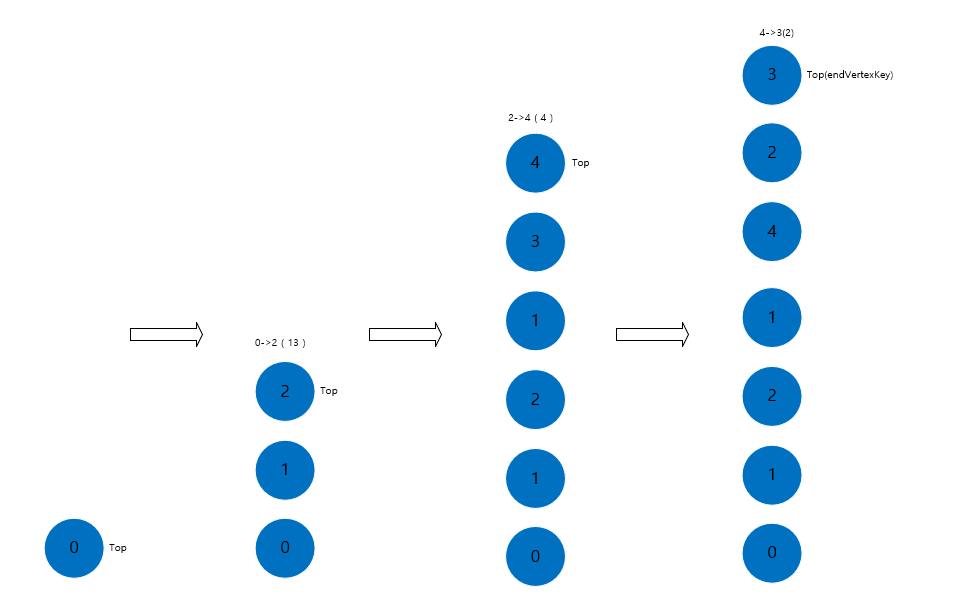


**(LOAD mapdata.txt)** mapdata.txt가 존재하기 때문에 LOAD 성공 메세지를 출력한 것을 확인할 수 있다.

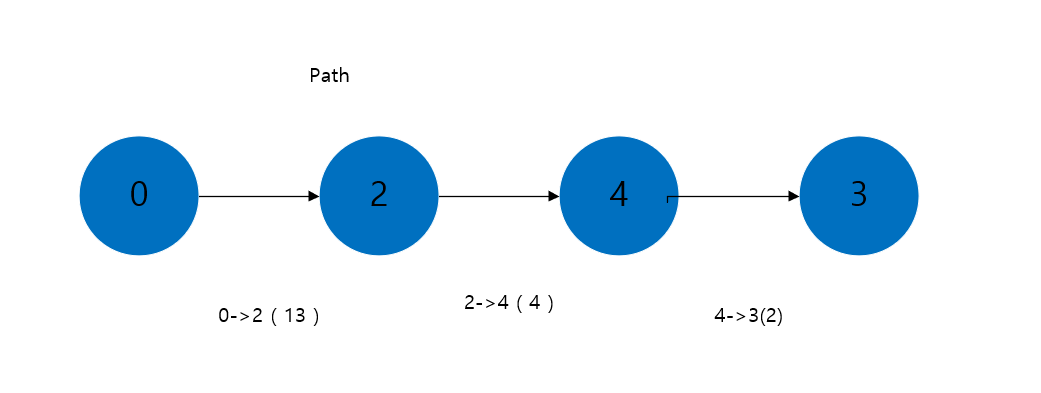
**(PRINT)** mapdata.txt에 존재하는 map data가 그대로 출력된 것을 확인할 수 있다.

**(DFS 0 3)** 인자 개수가 부족하지 않고, 각 인자로 들어온 Vertex Key가 존재하므로 DFS 알고리즘을 수행한다. Graph구성 연결은 그림으로 그리면 다음과 같다.

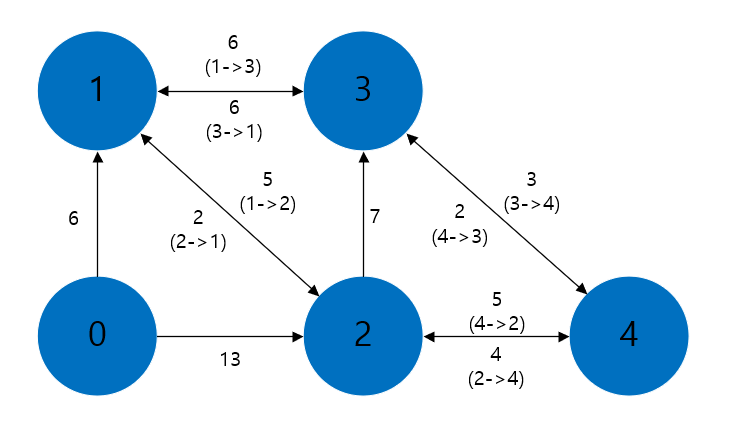




먼저 구현한 알고리즘 대로 진행되는 Flow이다. 길이 막혀 있을 경우 Pop을 하는데, 이 경우에는 각 Top에서 연결된 Edge들을 Stack에 Push하고, 또 Top에 대해서 다시 Edge들을 Stack에 Push하면서 진행하는데 길 막힘 없이 End Vertex Key를 찾았기 때문에 알고리즘이 종료된다. Stack에 pair로 Vertex와 Edge로 가는 Weight를 넣어줬기 때문에 모두 더한 값은 19인 것을 확인할 수 있다. 따라서 다음과 같은 Path와 Distance를 얻을 수 있다.



**(DIJKSTRA 0 3)** 인자 개수가 부족하지 않고, 각 인자로 들어온 Vertex Key가 존재하므로 DFS 알고리즘을 수행한다. Graph구성 연결은 그림으로 그리면 다음과 같다.



먼저 DIJKSTRA 알고리즘이 진행되는 Flow는 다음과 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| Distance | 0 | MAX | MAX | MAX | MAX |
| Path | -1 | -1 | -1 | -1 | -1 |
| Visit | F | F | F | F | F |

먼저 Start Vertex Key에 대해서 Distance를 초기화한다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| Distance | 0 | 6 | 13 | MAX | MAX |
| Path | -1 | 0 | 0 | -1 | -1 |
| Visit | T | F | F | F | F |

그 다음 Start Vertex Key에 대해서 연결된 Edge들의 Distance를 Update한다. 가장 짧은 거리를 가진 Vertex는 1번 Vertex이기 때문에 1번 Vertex를 기준으로 다시 진행한다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| Distance | 0 | 6 | 11 | 12 | MAX |
| Path | -1 | 0 | 1 | 1 | -1 |
| Visit | T | T | F | F | F |

그 다음 짧은 거리를 가진 2번 Vertex를 기준으로 다시 진행한다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| Distance | 0 | 6 | 11 | 12 | 15 |
| Path | -1 | 0 | 1 | 1 | 2 |
| Visit | T | T | T | F | F |

그 다음 방문하지 않은 가장 짧은 Vertex 3번을 기준으로 다시 진행한다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| Distance | 0 | 6 | 11 | 12 | 15 |
| Path | -1 | 0 | 1 | 1 | 2 |
| Visit | T | T | T | T | F |

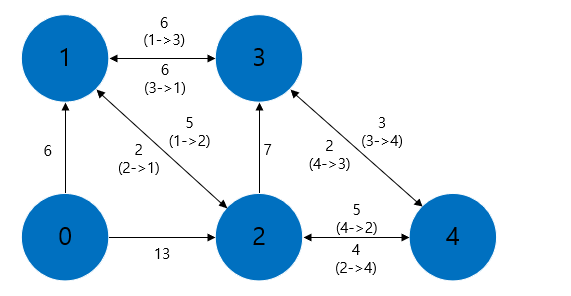
마지막 Vertex 4번을 기준으로 다시 진행한다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| Distance | 0 | 6 | 11 | 12 | 15 |
| Path | -1 | 0 | 1 | 1 | 2 |
| Visit | T | T | T | T | T |

모든 Vertex에 대해 방문이 완료되었으므로 업데이트 된 정보는 다음과 같다. Start Vertex Key는 0이고, End Vertex Key는 3이기 때문에 Path를 따라가보면 3번을 참조했을 때 1, 1번을 참조했을 때 0이므로 Path는 0 -> 1 -> 3 -> 2이다. Distance 정보는 그냥 해당 End Vertex의 Distance를 출력하면 되므로 12이다. 결과화면에서 나온 것과 동일한 것을 알 수 있다.

**(DIJKSTRAMIN 0 3)** 위의 DIJKSTRA 알고리즘과 동일하나, Min Heap을 사용한 것만 다르다. 구성된 알고리즘은 동일하고 같은 결과를 출력한 것을 확인할 수 있다.

**(BELLMANFORD 1 4)** Graph 구성은 다음과 같다.



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Vertex | 0 | 1 | 2 | 3 | 4 |
| dist | MAX | 0 | MAX | MAX | MAX |
| Path | -1 | -1 | -1 | -1 | -1 |

먼저 start Vertex에 대해서 distance를 0으로 초기화한다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Vertex | 0 | 1 | 2 | 3 | 4 |
| dist | MAX | 0 | 5 | 6 | MAX |
| Path | -1 | -1 | 0 | 0 | -1 |

0번 Vertex부터 차례대로 진행하는데, 1번을 제외한 모든 값은 Max이기 때문이 1번 Vertex부터 업데이트가 진행 된다.

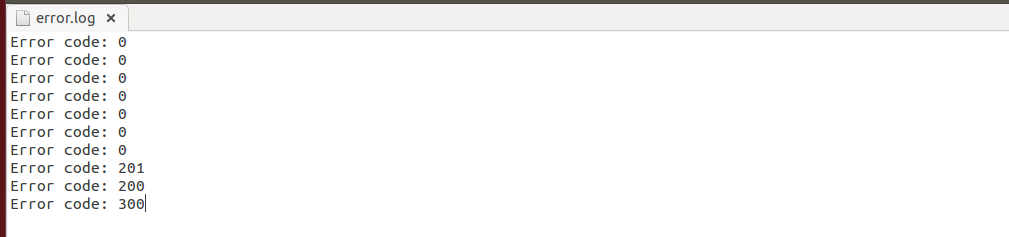
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Vertex | 0 | 1 | 2 | 3 | 4 |
| dist | MAX | 7 | 5 | 6 | 9 |
| Path | -1 | 2 | 0 | 0 | 2 |

다음은 2번 Vertex에 대해서 진행된다. 2번의 연결된 Edge 1, 3, 4에 대해서 진행하는데 원래 3번 Vertex의 Distance 값이 더 작기 때문에 Update는 되지 않는다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Vertex | 0 | 1 | 2 | 3 | 4 |
| dist | MAX | 7 | 5 | 6 | 9 |
| Path | -1 | 2 | 0 | 0 | 2 |

다음은 4번 Vertex에 대해서 진행된다. 4번에 연결된 Edge 2, 3이 모두 현재 가장 짧은 Distance를 갖고 있기 때문에 Update 되지 않는다. 순차적으로 이 알고리즘을 Vertex 수만큼 진행한다. Start Vertex Key가 1이고, End Vertex Key가 4이기 때문에 4번 Vertex의 Path를 추적하면 1, 2, 4, 9를 얻을 수 있고, Distance는 4번 End Vertex Key에 대해서 출력한다. 따라서 Distance는 9다. 결과 화면에 맞게 출력된 것을 확인할 수 있다.

**(ETC)** 나머지 명령어들의 수행은 위의 알고리즘과 동일하고, 예외에 대한 출력을 확인할 수 있다.



먼저 error.log에 대한 결과 화면이다. 명령어 성공 시, Error Code 0을 출력하고, 나머지는 해당 Error Code에 맞게 결과를 error.log에 출력한 것을 확인할 수 있다.

**Consideration & Conclusion**

이번 프로젝트는 Graph에 대해서 경로와 최단경로를 구하는 알고리즘을 사용해서 Path와 Distance를 구하는 것이었다. 설계 수업 때 어느정도 이해해서 그렇게 어렵지 않을 것이라고 생각했지만, 실제로 Code를 작성하고, 경로를 찾다가 예외를 처리하는 부분에 있어서 어려웠다. 먼저 DFS 알고리즘에서 길 막힌 것에 대해서 처리하는 과정이 너무 어려웠다. 실습 강의자료대로 인접한 Edge들 중 방문하지 않은 것에 대해 수를 넣으려고 했지만 이게 맘처럼 되지 않았다. 따라서 방문하고 Pop처리 하지 않고, 모든 정보들을 Push하고 길이 막혀 돌아가야 할 때, Pop을 하면서 Path와 Distance를 지워주는 방식으로 해결했다. 그리고 DIJKSTRA 알고리즘에서는 Visit table을 사용하지 않고, STL Set의 개념으로 해결했다. 그리고 Min Heap으로 구현한 DIJKSTRA도 그냥 Min Heap을 구현하고 Set처럼 똑같이 사용하면 돼서 어렵지 않았다. 그러나 Min Heap을 구현하는데 있어서 Heapify 함수를 이해하기 어려웠다. 이 부분에 있어서는 Push하는 경우에 삽입 된 위치에 부모를 Index로 해서 바꿔주고, 부모, 왼쪽자식, 오른쪽 자식 3개의 Node를 Grouping해서 recursive하게 Root까지 확인하면서 Swap처리로 해결했다. Heapify함수를 사용하면서 Vector 형태로 되어 있어서 배열내에서도 완전히 오름차순으로 정렬하고 싶었지만, 쉽지 않았다. 따라서 Min Heap구조가 성립하도록 했다. DS 수업을 수강하면서 STL도 많이 사용해보고, 많은 자료구조도 직접 구현해보고 해서 실력이 많이 향상 되었다고 느꼈다.