Data Structure Project3

Report

학과 : 컴퓨터 공학과

팀장 : 2013722095 최재은

팀원 : 2013722021 정무영

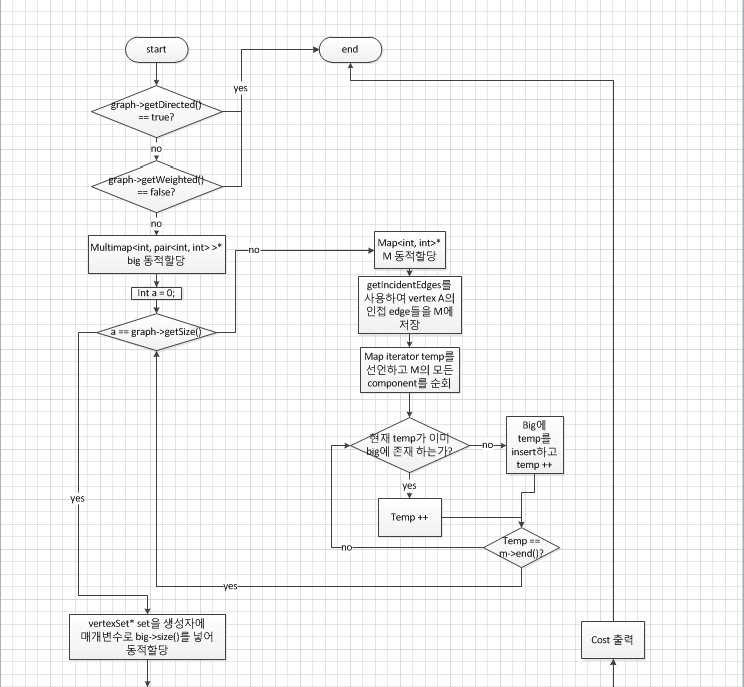
2013722079 김영근

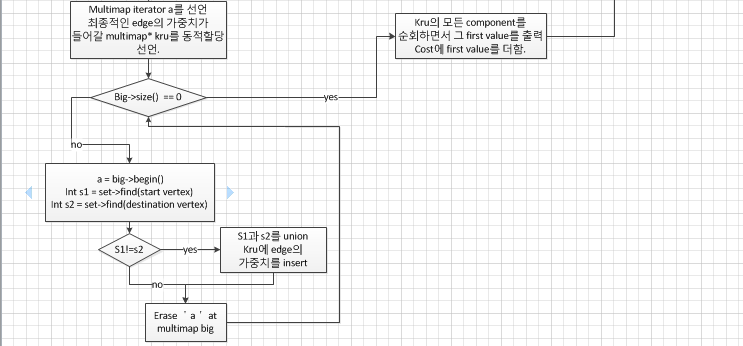
1. Introduction

이번 프로젝트는 배열과 리스트형식의 두 가지 그래프를 이용해서 DFS, DFS\_R(recursive), BFS, Dijkstra, Kruskal알고리즘을 구현하는 것이다. 처음 프로그램을 실행하면 배열이나, 리스트 형식으로 작성된 그래프 정보를 불러온다. 불러온 정보를 바탕으로 프로그램 내부에서 그래프를 형성한다. DFS는 한 개의 vertex를 시작점으로 잡고 하는 깊이 우선 탐색으로 Stack을 사용해서 구현한다. DFS\_R은 Stack대신 재귀 함수를 사용해서 DFS와 같은 결과가 나오도록 구현한다. BFS는 한 개의 vertex를 시작점으로 잡고 하는 넓이 우선 탐색으로 Queue를 사용해서 구현한다. Dijkstra는 한 개의 vertex를 시작점으로 잡고 그 vertex로부터 모든 vertex로의 최단거리와 경로를 구해주는 알고리즘이다. Kruskal은 현재의 그래프에서 MST(Minimum Spanning Tree)를 만들어 내는 알고리즘이다. 위 5개의 알고리즘을 구현할 땐 그래프가 배열기반인지 리스트 기반인지를 구별하지 않고 한 개의 코드로 일반화 해서 구현한다. 또 한 위의 알고리즘을 실행 시키기 위한 전제 조건이 있는데 DFS, BFS는 그래프가 방향성과 가중치가 없어야 하고 Dijkstra는 방향성과 가중치가 모두 있어야 한다. Kruskal은 방향성은 없고 가중치는 있어야한다. 그래프가 제대로 구성 됐는지 확인 하기 위해서 PRINT기능을 사용하면 되는데 그래프의 형식에 따라 출력 형식을 달리 한다.

1. Flowchart
2. BFS, DFS, DFS\_R

Untitled Diagram

2) Kruskal



3)Dijkstra

시작

NO

NO

NO

NO

NO

YES

YES

YES

YES

YES

edge 정보(확인한 edge 다음의)를 받는다.

minheap에서 가장 거리가 짧은(이전 vertex다음 vertex) vertex를 꺼낸다

dist, prev, minheap을 재설정 한다.

s에서부터 직통 거리보다 minheap에서 꺼낸 vertex를 거쳐가는게 더 짧은 가?

해당 edge에 연결 된 vertex가 s에 속하는가?

끝

시작 vertex부터 모든 vertex에 대한 최단 거리를 구하고 그 경로를 역순으로 출력한다.

모든 edge를 살펴 보았나?

minheap에서 꺼낸 vertex의 인접 edge들의 정보를 받는다.

minheap의 끝까지 도달했는가

해당 vertex가 s에 속해 있는가?

각종 변수들 초기화

dijkstra의 실행 조건을 만족 하는가?

1. Algorithm
2. BFS

-queue q 선언, map get[size] 선언, map iterator 선언, bool visited[size] 선언

-for문 사용 size만큼 visited false로 설정

-base값 push, visited[base] true 설정

-while문 사용 !q.empty일 때

-q.front, q.pop, q.front 값을 프린트

-graph->getincidentedges(q.front, get)

-iterator=get[q.front].begin설정

-for문을 사용 iterator가 !get[q.front].end이면 iterator++

-if문 사용 visited[iterator->frist]가 false이면

-q.push iterator->frist, visited[iterator->frist] true로 설정

-while문 탈출 하기 전까지 계속 실행

-탈출 시 return true

1. DFS

-stack s 선언, map get[size] 선언, map iterator 선언, bool visited[size] 선언

-for문 사용 size만큼 visited false로 설정

-base값 push

-while문 사용 !s.empty일 때

-s.top, s.pop,

-if문 사용 만약 visited[s.top]이 false이면 s.top 값을 프린트, visited[s.top] true설정

-graph->getincidentedges(s.top, get)

-iterator=get[s.top].end설정

-for문을 사용 iterator가 !get[s.top].begin이면 iterator--

-if문 사용 visited[iterator->frist]가 false이면

-s.push iterator->frist, visited[iterator->frist] true로 설정

- iterator=get[s.top].begin설정

-if문 사용 visited[iterator->frist]가 false이면

-s.push iterator->frist, visited[iterator->frist] true로 설정

-while문 탈출 하기 전까지 계속 실행

-탈출 시 return true

1. DFS\_R

-map get[size] 선언, map iterator 선언

-visit[base] true선언

-base출력

-graph->getincidentedges(s.top, get)

-iterator=get[base].begin설정

-for문을 사용 iterator가 !get[base].end이면 iterator++

-if문 사용 visit[iterator->first]가 false이면

-DFS\_R recursive

-for문 탈출 시 return true

4) Kruskal

Graph가 방향성을 갖고 있거나, 가중치를 가지고 있지 않다면 false를 반환한다.

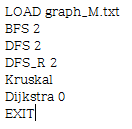
<int, pair<int, int>>\* big을 선언과 함께 동적 할당 해주고, (가중치, 시작vertex, 도착vertex) 반복문을 사용하여 모든 vertex들에 연결된 edge들을 big에 insert한다. 이때 조건문을 사용하여 이미 big map에 존재하는 edge에 관하여는 insert하지 않는다.

vertexSet\* set을 선언하고 그 생성자의 매개변수로 전체 그래프의 edge를 따놓은 big의 size를 입력한다. 이후 반복문을 big의 size가 0이 될 때까지 수행하며, 그 안에서 map iterator a가 big의 begin을 가리키게 한다. Int형 변수 s1과 s2는 각각 a의 second value의 첫 번째 값과 두 번째 값을 받으며, s1 != s2, 즉, 두 vertex간의 edge가 loop(cycle)을 생성하지 않는다면 그 둘을 union하고 최종적으로 kruskal algorithm의 결과 edge들을 담고 있을 kru map에 그 edge의 가중치 값을 넣어준다.

반복문의 제일 끝에서 big의 제일 앞 value를 삭제해준다. 이렇게 kru에 가중치를 기준으로 하여 오름차순으로 edge들을 insert해주고 나면 반복문을 통해서 kru map의 begin부터 순차적으로 출력해준다. 출력될 때마다 그 값들은 int cost에 더해지며 최종적으로 모든 edge의 가중치의 합인 cost를 출력해준다.

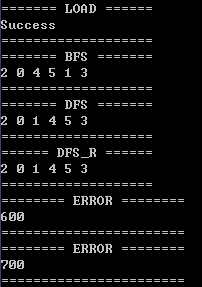
5)Dijkstra

먼저 그래프가 Dijkstra알고리즘을 수행할 수 있는 상태인지 검사를 한다. 방향성과 가중치가 그래프에 있지 않거나 시작 vertex로 지정된 vertex가 그래프 범위내에 있지 않으면 Dijkstra알고리즘을 수행하지 않고 종료한다. 그래프가 Dijkstra조건을 충족하면 각종 변수들 초기화 작업을 한다. 먼저 시작 vertex로부터의 최단거리가 구해진 vertex들을 나타낼 bool형 배열 s를 모두 false로 초기화 한다. 시작 vertex로부터의 거리를 저장한 int형 dist배열을 처음엔 다른 vertex를 거치지 않은 한번에 도달하는 거리를 저장하고 도달 할 수 없는 vertex는 엄청 큰 값으로 지정한다. 마지막으로 해당 vertex로 가는 경로 중 바로 직전 vertex정보를 저장하는 int형 prev배열을 설정한다. 초기화 시에는 시작 vertex와 연결돼 있으면 시작 vertex로 설정하고 그렇지 않으면 -1로 설정한다. 그리고 multimap<int, int>형식으로 s집합으로부터의 s집합에 속하지 않은 vertex의 거리를 minheap형식으로 저장한다. first에 목적지까지의 거리 길이를 저장하고 second에 목적지 vertex의 번호를 저장한다. 그 후 시작 vertex를 s에서 true로 설정하고 dist에선 0으로 prev는 -1로 설정 한다. 사전 설정을 마치고 minheap에서 가장 작은 값들을 꺼내면서 그 값들의 목적지 vertex가 s에 속해 있으면 다음으로 작은 값을 확인하고 vertex가 s에 속해 있지 않거나 minheap의 끝까지 다 체크했을 경우 iterator에 해당 multimap의 정보를 저장한다. 이 때 minheap의 끝까지 확인을 했다면 출력을 해준다. 그렇지 않으면 iterator에 저장된 vertex의 주변 edge정보를 받아서 그edge와 연결된 vertex가 s에 속해 있지 않으면 s로부터 목적지 vertex의 직선거리와 s에서 iterator를 거쳐서 목적지 vertex로 가는 거리를 비교해서 작은 값을 dist에 저장한다. 만약 iterator를 거쳐서 가는 거리가 더 적으면 목적지 vertex의 prev를 iterator로 재설정 해준다. minheap에서는 바뀐 거리 정보를 새로 넣어주고 이전 거리 정보는 삭제를 해준다. 해당 iterator의 주변 edge에 대해서 위의 동작을 전부 시행 했으면 해당 itertor의 vertex는 s에 넣어준다(true로 설정). 그 후 위의 동작을 minheap의 끝까지 탐색했을 때까지 반복한다. 출력 부분은 vertex 0부터 시작해서 마지막 vertex까지 시작 vertex로부터의 최단 거리와 그 경로를 역순으로 출력한다. 최단 거리는 dist배열에 저장된 값을 출력하면 되고 prev는 해당 vertex를 index로 시작해서 prev값을 계속 타고가면서 해당 prev값을 출력해주다가 -1을 만나면 출력하지 않고 출력을 종료하면 역순 출력을 완료 할 수 있다.

1. Result Screen

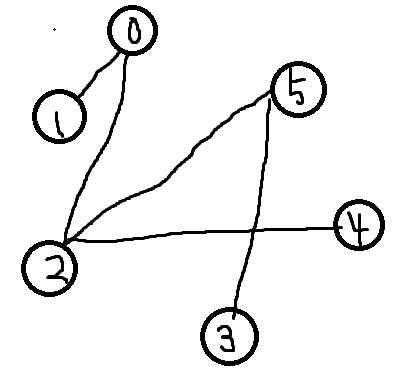
**<사용한command text>**

1. BFS, DFS, DFS\_R

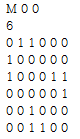


< 도형화 >

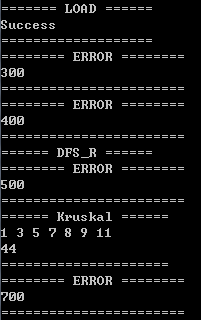
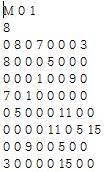
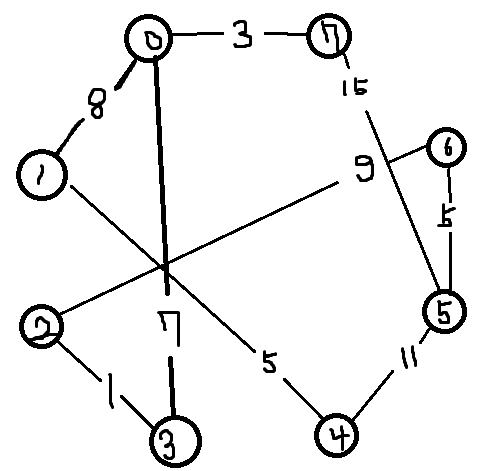
< 도형화 >



<사용한 그래프>



1. Kruskal



< 도형화 >

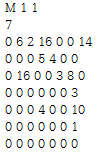
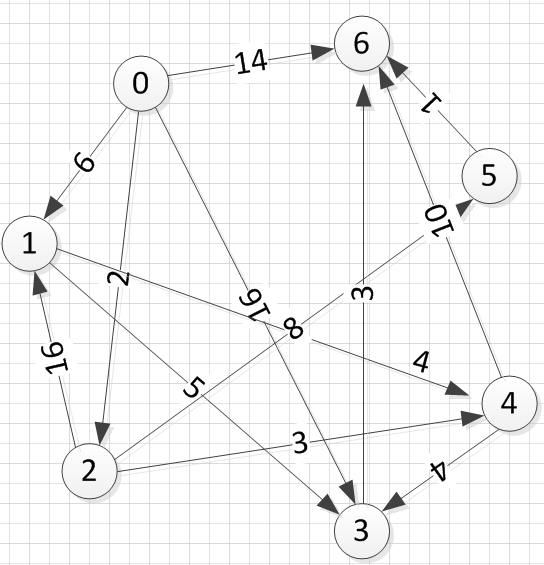
<사용한 그래프>

1. Dijkstra

< 도형화 >

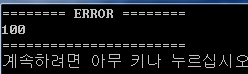
<사용한 그래프>



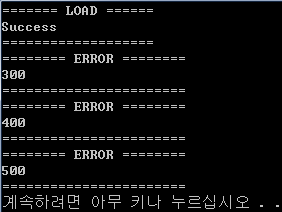


* Error case

1. Data file을 load할 수 없는 경우

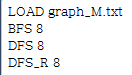


1. ‘시작 vertex를 입력하지 않은 경우’와 ‘입력한 vertex가 해당 그래프에 존재하지 않는 경우’



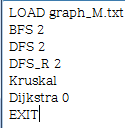
<사용한 command>

<graph의 size>

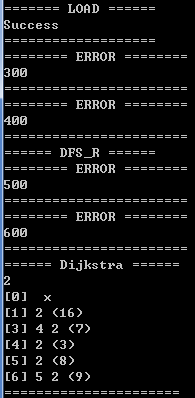


1. graph 의 속성이 해당 search를 수행하기 위한 조건에 맞지 않는 경우

<graph의 속성> <입력한 command>



1. Dijkstra 에서 방문할 수 없는 vertex는 경로를 x로 표시한다.



1. Consideration
2. 최재은

이번 프로젝트를 진행하면서 가장 큰 난관이라면 처음에 그래프를 구현하는 단계이지 않은가 싶다.

사실상 전체 구현에 있어서 가장 중요한 부분이면서 토대가 되어야 할 부분이기 때문에 이를 정상적으로 구현하는 것이 가장 중요하다고 생각되었다. Manager에서 run은 별다르게 특이점도 없었고 어려운 점도 없었다. LOAD에서 4개의 토큰을 사용하여 그래프의 형식, 방향성, 가중치 여부, 그래프의 크기를 받았고, 이에 따라서 각각 다른 생성자를 사용하여 그래프를 구현하도록 설계하였다. BFS나 DFS에서는 별다르게 문제가 없었던 것 같다. 다만 DFS\_R을 구현하는 단계에 있어서 난관이 있었는데, 마지막의 vertex를 출력하지 않거나, 무한 루프를 생성하여 제대로 구현이 되지 않는 부분이었다. 교수님께서 주신 강의자료에는 오버로딩을 통한 재귀함수의 사용방식이 나와있었는데, 추가하려는 것에 귀찮음이 있어 자기 자신을 다시금 호출하는 방식으로 구현하였다. 결국 구현 방식 자체는 동일하다고 본다. 다만 이렇게 구현하다 보니 DFS\_R의 에러를 표시하는 부분에서 ====== DFS\_R ======= 이 나와버린다는 것이 문제였지만 이것은 알고리즘을 이해하고 그것을 직접 구현해보는 단계인 지금에 있어서 그렇게 큰 문제가 될 것 같지 않아 단점으로 두었다.

또한 실행하려는 함수에 따라서 getIncidentEdges함수가 각각 다른 값을 map에 저장하도록 구현하였는데 이 부분에 있어서 불필요한 요소가 다소 첨가되어있는 것 같다. 이번 프로젝트가 시간적으로 더 여유가 있었더라면 하는 마음이 절실했었다. BFS와 DFS함수에서는 제공된 강의자료나 조교님이 설명해주신 바와 같이 각각 queue와 stack를 사용하여 구현하였다. 내부적으로 mark라는 map을 만들어 다음에 방문할 vertex가 이미 방문된 것인지를 확인하는 용도로 사용하였다. Kruskal에서는 multimap을 사용하였는데 이것이 강의자료에서는 minheap을 사용하여 구현하였던 것으로 기억한다. 다만 이 부분을 참고하지 못하고, key값에 따라 자동으로 정렬되는 map의 특성을 응용하여 key를 가중치로 두고 함수를 구현하였다. Disjoint 부분에 있어서는 가히 감탄을 그지 못할 정도의 충격을 먹었다. 배열 방식을 이용해서 그 부모를 찾고 union한다는 것과 그것을 통해 cycle을 생성하는지 여부를 확인할 수 있다는 것이 정말로 신기했다. 솔직히 말하면 보고서를 작성하는 지금 이 순간에도 제대로 이해가 되지 않았다. 코딩을 하면서도 cycle을 어떻게 확인하나…..조건문을 한 6~7개 정도 달아서 주변에 대한 모든 edge와 vertex를 확인해야하나…..하고 걱정하고 있었는데, 그 수고로움이 단 몇 줄만으로 해결된다니….. Dijkstra에서는 그래프의 edge 정보를 matrix type으로 저장한 length 배열과 각 vertex의 distance, visit, path를 담고 있는 info 배열을 만들고 이를 응용하여 함수를 구현하였다. 조금은 코딩방식이 더티코딩이 돼버려서 조금 많이 아쉽지만 애초에 제출주차에 시험이 하루를 제외하고 모두 있는 바람에 3차 프로젝트의 목표를 kruskal까지만 구현하는 것으로 잡았던 것에 비해 비록 몇 개의 입력예제들만 검증해보아서 많은 부분에 예외처리가 필요하겠지만 모든 함수를 구현해 놓았다는 것이 참으로 만족스럽다.

정무영 학생은 나와는 다르게 LOAD에서 4개의 토큰을 사용하지 않고 4개의 int 변수를 두고 사용한 부분만 다르고 나머지는 유사하였다. Method 부분을 보고서는 솔직히 경악했다. 내가 작성한 코드가 얼마나 비효율적이었는지를 조금 깨닫는 순간이었다. 그냥 강의자료에 있는 대로 구현하면 될 것을 굳이 여러 조건을 달고 꼬아서 구현했구나 라는 생각이 들었다…. 조금은 자괴감이 생겨버린 듯 하다…. 코드가 정말 깔끔하다는 생각이 들었다. DFS\_R에 있어서 workhorse, driver를 따로 오버로딩으로 구현하여 만든 것 역시 나의 코드와는 달랐다. Dijkstra에서도 내가 해결하지 않고서 넘어갔던 출력부분에 있어서 이런 간단한 방법을 생각하지 못한 것에 참으로 한탄스럽다.

김영근 학생 역시 주어진 강의자료를 잘 활용하여 기존의 BFS나 DFS를 구현하였다고 생각한다. 또 한번 자괴감이 들었다. 역시 DFS\_R의 구현에서 workhorse와 Driver를 따로 구현하였다. 이쯤 되니 내가 이번 프로젝트는 정말이지 ‘신박한 방법을 썼구나’하는 생각이 든다. Dijkstra를 미처 구현하지 못한 것을 보고 마음이 아팠다….. cmd창에 출력되지 않게 한 것은 어째서인지 모르겠다. 그래도 저번 2차 프로젝트 때 보다 코딩방식이 좀 더 정갈해진 듯한 느낌을 받았다.

1. 정무영

나에 대한 고찰

이번 프로젝트를 진행하면서 계속 느꼈던 점은 ‘교수님의 강의자료와 강의가 없었으면 정말 오래 걸리고 힘들게 구현했겠다.’였다. 강의 자료의 가이드 라인이 없었더라면 처음 접근을 하는 것부터 시작해서 효율적인 코드작성까지 구현하기 힘들거나 아예 구현을 못했을 것이다. 특히 Dijkstra부분은 온라인 강의로도 올라와 있어서 프로젝트를 하면서 다시 강의를 정주행하니 Dijkstra를 짜는데 많은 도움이 됐다. Kruskal을 짤 때도 Find, Union을 배우지 않았더라면 감도 안 잡혔을 텐데 덕분에 아주 쉽게 구현할 수 있었다. 하지만 강의자료를 참고하면서 구현해서 그런지 확실히 코드를 내 것으로 만들지 못해서 퀴즈 때 다시 작성하려 하니 꽤 많은 시간이 필요했다. 이번 프로젝트를 하면서 STL의 유용함을 다시한번 느낄 수 있었다. 처음에 STL을 배운다 했을 땐 괜히 배우는 것만 늘어 나서 귀찮다고 생각했었다. 하지만 프로젝트를 하면서 STL이 아주 유용하게 쓰이는 상황이 자주 있어서 덕분에 코드가 간결해지고 구현도 훨씬 쉬웠다. 교수님의 현안을 다시한번 느낄 수 있었다.

최재은

DFS, BFS에서 if문을 과하게 쓴 느낌이 있다. 그래서 코드가 비교적 지저분해 보인다. 난 강의자료대로 재귀함수로 DFS를 구현하는 과정에서 두개의 함수를 사용했지만 여기 선 한 개의 함수로만 구현 한 것을 보고 ‘그것도 가능하구나’ 라고 느꼈다. Kruskal에서 나는 minheap을 만들 때 함수를 따로 만들어서 했지만 여기 선 Kruskal내부에서 직접 minheap을 만들었다. 여기 선 Kruskal이 형성되지 못하는 그래프의 경우는 예외처리 하지 않은 듯 해 보인다. Dijkstra부분에선 난 minheap을 사용했고 이 코드에선 for문 중첩으로 구현을 한 것 같다. 또한 prev와 s대신 이차원 배열로 해결을 한 거 같은데 강의자료를 안 보고 해서 그런지 전체적으로 5개의 알고리즘 중 내 코드와 가장 괴리감을 보인다.

김영근

DFS, BFS는 if문의 위치만 조금 다를 뿐 내 코드와 거의 유사하다. DFS\_R은 구현하다 만 걸로 보이고 Kruskal과 Dijkstra도 구현하지 않은 것으로 보인다.

3) 김영근

1. 인증샷

