자료구조(Data Structure)

Programming Assignment 6

20161603

신민준

목차

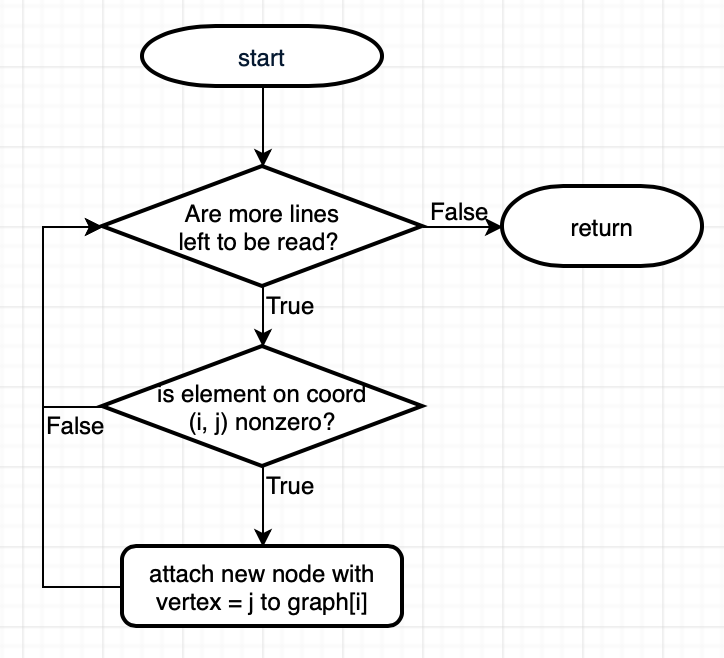
1. 문제1
   1. 프로그램 구조
   2. 세부 설명
   3. 참고 사항
2. 문제2
   1. 프로그램 구조
   2. 세부 설명
   3. 참고 사항
3. 문제 코드
   1. 문제1 코드
   2. 문제2 코드
4. 문제1

문제1으 undirected graph 의 모든 connected component로 이루어진 각 집합을 출력하는 문제입니다. 이 때, 주요한 구현 요소는 다음과 같습니다.

* Adjacency matrix로 입력받은 데이터를 Adjacency List의 형태로 저장하여 사용할 것
* Adjacency List 안의 데이터를 기반으로 DFS를 수행할 것

DFS 알고리즘은 교재에서 소개한 알고리즘과 동일하게 구현하면 되는 것이므로, Adjacency List를 구성하는 과정에 더 집중해 구현할 수 있었습니다.

* 1. 프로그램 구조



Flowchart . Adjacency Matrix to Adjacency List Conversion

* 1. 세부 설명

Adjacency Matrix에서 Adjacency List로 변환하는 방법은 크게 어려울 것 없었습니다. 먼저, 입력받을 의 값을 입력받고, 해당 값으로 적당한 크기의 graph 배열과 visited 배열을 생성했습니다. graph 배열은 Adjacency List representation을 위한 배열이며, visited 배열은 이후 DFS 과정에서 사용됩니다. 해당 변수의 생성과 초기화는 [Code 1]에서 보인 것과 같습니다.

|  |
| --- |
| graph = malloc(**sizeof**(node\_pointer)\*vertex\_cnt);  visited = calloc(vertex\_cnt, **sizeof**(int));  **for**(int i=0 ; i<vertex\_cnt ; i++)  graph[i] = NULL; |

Code

이후, 이중 for 문을 돌면서, row-major order로 Adjacency Matrix를 읽어들이고, 만일 위치에 0이 아닌 값, 즉 edge가 존재한다면, 새로운 node를 생성하고, node의 vertex 값을 j로 설정한 후, 해당 node를 graph[i]에서부터 연결되어있는 Linked List에 연결합니다. 이 과정은 [Code 2]에서 보이고 있습니다.

|  |
| --- |
| **for**(int i=0 ; i<vertex\_cnt ; i++) {  **for**(int j=0 ; j<vertex\_cnt ; j++) {  fscanf(fp, " %d", &tmp);  **if**(tmp) {  **new** = malloc(**sizeof**(**struct** \_node));  **new**->vertex = j;  **new**->link = graph[i];  graph[i] = **new**;  **new** = NULL;  }  }  } |

Code

이 모든 과정이 끝나면, read\_file() 함수는 종료하게 되고, graph[] 배열에 입력받은 그래프가 Adjacency List representation의 방식으로 저장되어지게 됩니다.

이 함수는 사이즈를 가진 Adjacency Matrix의 각 원소를 한번씩 방문하므로, read\_file() 함수의 시간복잡도는 다음과 같습니다.

이후, 생성한 그래프를 가지고 각 vertex에 대한 DFS를 수행해 connected component들을 출력해야 하므로, read\_file() 함수가 종료된 후, [Code 3]처럼 출력과정을 구현했습니다.

|  |
| --- |
| int main(int argc, **const** char\* argv[]) {  read\_file();  **if**(!(ofp = fopen(OUTPUT\_FILENAME, "w"))) {  fprintf(stderr, "error: writing to file failed**\n**");  exit(1);  }  **for**(int i=0 ; i<vertex\_cnt ; i++) {  **if**(!visited[i]) {  dfs(i);  fprintf(ofp, "**\n**");  }  }  fclose(ofp);  ... |

Code

이 과정에서 사용되는 DFS 수행 함수인 dfs()는 다음과 같이 recursive하게 구현했습니다.

|  |
| --- |
| */\*\**  *\* Executes depth-first search starting from a vertex*  *\* @param vertex The vertex number to start dfs from*  *\*/*  void dfs(int vertex) {  fprintf(ofp, "%d", vertex);  visited[vertex] = 1;  **for**(node\_pointer ptr = graph[vertex] ; ptr ; ptr = ptr->link) {  **if**(!visited[ptr->vertex]) {  fprintf(ofp, " ");  dfs(ptr->vertex);  }  }  } |

Code

이 때, Adjacency List representation을 사용한 DFS의 시간복잡도는 다음과 같습니다.

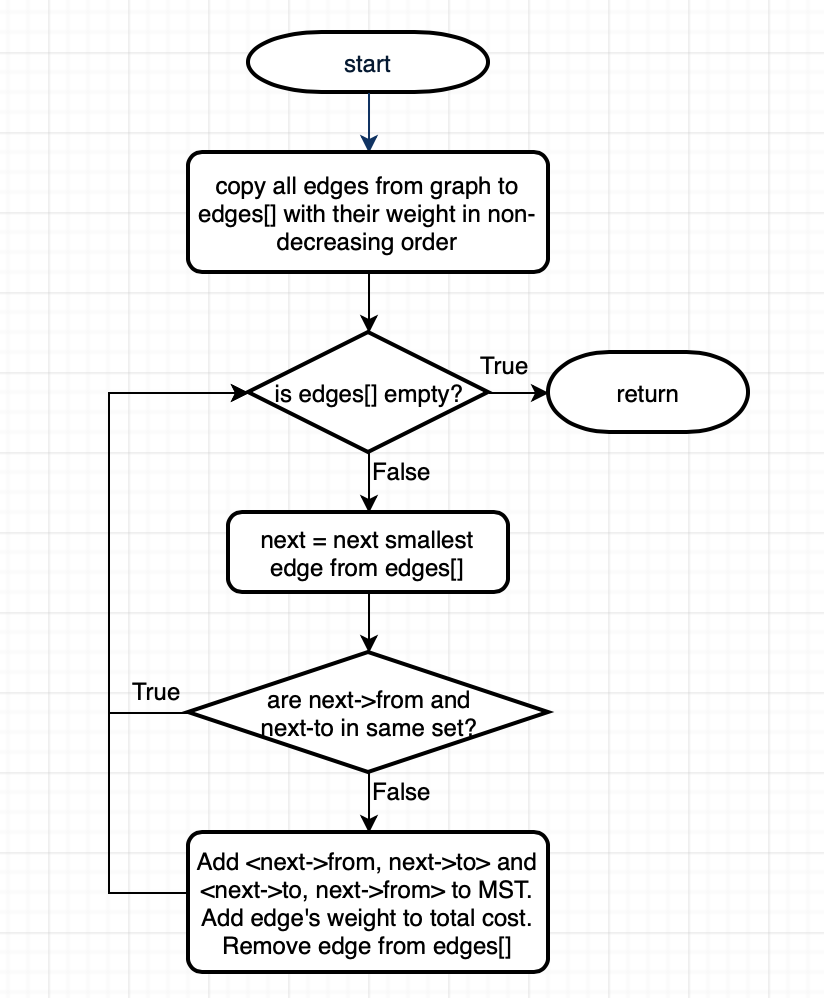
따라서, 그래프의 edge 수는 vertex 수의 제곱보다 클 수 없으므로, 이 프로그램의 전체 시간복잡도는 다음과 같습니다.

* 1. 참고 사항
* 같은 데이터에 대해 수없이 다른 자료구조로 나타낼 수 있다는 것은 알고 있었지만, 서로 다른 자료구조로 전환하는 일이 빈번히 일어난다는 것을 느낄 수 있었습니다. 따라서, 데이터를 다른 형태로 전환하는 과정을 원활히 하기 위해선 여러 가지 형태의 자료구조에 대해 익숙해져야 함을 배웠습니다.
* 이 문제에서는 Adjacency List를 요구했기에 해당 자료구조로 DFS를 처리했지만, 이 과정은 Adjacency Matrix로도 크게 불편함 없이 구현이 가능합니다. 이 경우, DFS 과정의 시간 복잡도는 가 됩니다.

1. 문제2

문제2는 교재에서 소개한 Kruskal’s Algorithm을 사용해 Minimum cost Spanning Tree를 구성하는 문제입니다. 이 문제를 해결하기 위해, 교재의 Chapter 5에서 소개한 union-find function을 활용해 각 vertices의 집합을 구성했습니다.

* 1. 프로그램 구조



Flowchart 2 – create\_mst() function

* 1. 세부 설명

Linked List와 Array를 모두 사용해 구현했습니다. Kruskal 알고리즘에 따르면, 입력받은 graph에서 모든 edge들을 나타내는 집합을 새로 만들어주어야 했기에, [Code 7]과 같이 새로운 구조체를 선언해 주어, edge들의 방향과 각 edge들이 지닌 가중치, weight를 모두 나타낼 수 있는 새로운 구조체를 사용할 수 있게 구현했습니다.

|  |
| --- |
| **struct** edge {  int from;  int to;  int weight;  };  **typedef** **struct** \_node\* node\_pointer;  **struct** \_node {  int vertex;  int weight;  node\_pointer link;  };  node\_pointer\* graph;  node\_pointer\* mst; |

Code

이후, 해당 구조체의 배열인 edges[]를 선언해 그래프의 모든 edge를 포함하도록 구현했고, 모든 edge를 포함시킨 이후, 각 edge들을 그들의 weight에 따라 오름차순으로 정렬시켰습니다. 이 과정은 [Code 8]에, 그리고 정렬에 사용된 sort\_edges() 함수의 구현은 [Code 9]에서 보이고 있습니다.

|  |
| --- |
| edges = malloc(**sizeof**(**struct** edge\*)\*edge\_cnt);  **for**(int i=0 ; i<vertex\_cnt ; i++) {  **for**(node\_pointer ptr = graph[i] ; ptr ; ptr = ptr->link) {  edges[edge\_idx] = malloc(**sizeof**(**struct** edge));  edges[edge\_idx]->from = i;  edges[edge\_idx]->to = ptr->vertex;  edges[edge\_idx]->weight = ptr->weight;  edge\_idx++;  }  }  sort\_edges(edges, edge\_cnt); |

Code

|  |
| --- |
| */\*\**  *\* Sort edges in non-decreasing order.*  *\* @param edges Set of edges to sort*  *\* @param n Number of edges in set*  *\*/*  void sort\_edges(**struct** edge\*\* edges, int n) {  **struct** edge\* tmp;  **for**(int i=0 ; i<n-1 ; i++) {  **for**(int j=i ; j<n ; j++) {  **if**(edges[i]->weight > edges[j]->weight) {  tmp = edges[j];  edges[j] = edges[i];  edges[i] = tmp;  }  }  }  } |

Code

각 edge들을 추가할 때 마다, 현재 그래프에서 cycle이 발생하는지 여부를 확인해야 합니다. 이 과정을 구현하기 위해 parent[] 배열을 생성하고, 각 노드들에 대해 root인 경우 -1을, 특정 vertex를 부모로 가진 경우 해당 vertex 값을 원소의 값으로 두도록 집합을 구성했습니다.

|  |
| --- |
| parent = malloc(**sizeof**(int)\*vertex\_cnt);  **for**(int i=0 ; i<vertex\_cnt ; i++) {  parent[i] = -1;  } |

Code

이 때, 각 집합의 root 노드를 찾는 함수를 sfind()로, 그리고 두 집합을 하나로 union하는 함수는 sunion()으로 구현해 사용했습니다. 이는 [Code 11]에서 볼 수 있습니다.

|  |
| --- |
| */\*\**  *\* Find function of union-find*  *\* @param parent Array holding information on sets*  *\* @param i Index to find set for*  *\* @return Found set*  *\*/*  int sfind(int parent[], int i) {  **while**(parent[i] >= 0)  i=parent[i];  **return** i;  }  */\*\**  *\* Union function of union-find*  *\* @param parent Array holding info on sets*  *\* @param i Index to union*  *\* @param j Index to union*  *\*/*  void sunion(int parent[], int i, int j) {  parent[i] = j;  } |

Code

이후, weight의 순서대로 edges[]의 모든 원소들을 읽어가면서, 각 edge의 from과 to에 해당하는 vertex의 집합이 다르다면, 두 vertex를 연결했을 때 cycle이 발생하지 않으므로, 이 edge를 새로운 mst[] 트리에 추가시킵니다. 이 때, 이 트리는 non-directional하므로, 반대 방향의 edge 또한 추가시켜줍니다.

|  |
| --- |
| **while**(edge\_idx < edge\_cnt) {  **struct** edge\* next = edges[edge\_idx++];  x = sfind(parent, next->from);  y = sfind(parent, next->to);  **if**(x != y) {  node\_pointer **new**;  **new** = malloc(**sizeof**(**struct** \_node));  **new**->link = mst[next->from];  **new**->vertex = next->to;  **new**->weight = next->weight;  mst[next->from] = **new**;  *// since the matrix is non-directional, add an edge with reversed direction.*  **new** = malloc(**sizeof**(**struct** \_node));  **new**->link = mst[next->to];  **new**->vertex = next->from;  **new**->weight = next->weight;  mst[next->to] = **new**;  sunion(parent, x, y);  cost += next->weight;  }  } |

Code

Edge의 정렬을 하는 알고리즘으로 selection sort를 사용했기 때문에, MST를 구하는 함수 create\_mst()의 시간복잡도는 일 때, 입니다.

create\_mst() 함수가 생성한 MST는 global variable mst[] 배열에 Adjacency List 형태로 저장되게 됩니다. 이에 대해 [Code 11]과 같이 DFS를 수행하면 원하는 출력을 받을 수 있습니다.

|  |
| --- |
| visited = calloc(vertex\_cnt, **sizeof**(int));  **for**(int i=0 ; i<vertex\_cnt ; i++) {  **if**(!visited[i]) {  dfs(i);  fprintf(ofp, "**\n**");  }  }  fprintf(ofp, "%d**\n**", cost); |

Code

* 1. 참고 사항
* MST의 생성 알고리즘은 알고리즘 중간에 정렬 과정이 들어가 있기 때문에, 어떤 sorting algorithm을 사용했는지에 따라 알고리즘의 시간복잡도가 바뀝니다. 제 구현 방식에서는 간단하게 의 selection sort 알고리즘을 사용했지만, 만약 heap sort나 quick sort와 같이 더 작은 time complexity를 가진 sorting algorithm을 사용한다면, Kruskal’s algorithm의 시간복잡도는 다음과 같습니다.

1. 문제 코드
   1. 문제1 코드

|  |
| --- |
| #define INPUT\_FILENAME "input.txt"  #define OUTPUT\_FILENAME "output.txt"  #include <stdio.h>  #include <stdlib.h>  **typedef** **struct** \_node\* node\_pointer;  **struct** \_node {  int vertex;  node\_pointer link;  };  node\_pointer\* graph;  int\* visited;  int vertex\_cnt;  FILE\* ofp;  */\*\**  *\* Reads data from input file*  *\*/*  void read\_file(void) {  FILE\* fp;  int tmp;  node\_pointer **new**;  **if**(!(fp = fopen(INPUT\_FILENAME, "r"))) {  fprintf(stderr, "error: fopen failed**\n**");  exit(1);  }  fscanf(fp, " %d", &vertex\_cnt);  graph = malloc(**sizeof**(node\_pointer)\*vertex\_cnt);  visited = calloc(vertex\_cnt, **sizeof**(int));  **for**(int i=0 ; i<vertex\_cnt ; i++)  graph[i] = NULL;  **for**(int i=0 ; i<vertex\_cnt ; i++) {  **for**(int j=0 ; j<vertex\_cnt ; j++) {  fscanf(fp, " %d", &tmp);  **if**(tmp) {  **new** = malloc(**sizeof**(**struct** \_node));  **new**->vertex = j;  **new**->link = graph[i];  graph[i] = **new**;  **new** = NULL;  }  }  }  fclose(fp);  }  */\*\**  *\* Executes depth-first search starting from a vertex*  *\* @param vertex The vertex number to start dfs from*  *\*/*  void dfs(int vertex) {  fprintf(ofp, "%d", vertex);  visited[vertex] = 1;  **for**(node\_pointer ptr = graph[vertex] ; ptr ; ptr = ptr->link) {  **if**(!visited[ptr->vertex]) {  fprintf(ofp, " ");  dfs(ptr->vertex);  }  }  }  int main(int argc, **const** char\* argv[]) {  read\_file();  **if**(!(ofp = fopen(OUTPUT\_FILENAME, "w"))) {  fprintf(stderr, "error: writing to file failed**\n**");  exit(1);  }  **for**(int i=0 ; i<vertex\_cnt ; i++) {  **if**(!visited[i]) {  dfs(i);  fprintf(ofp, "**\n**");  }  }  fclose(ofp);  **for**(int i=0 ; i<vertex\_cnt ; i++) {  **for**(node\_pointer ptr=graph[i] ; ptr ; ptr=ptr->link) {  free(ptr);  }  }  free(graph);  free(visited);  **return** 0;  } |

* 1. 문제2 코드

|  |
| --- |
| #define INPUT\_FILENAME "input.txt"  #define OUTPUT\_FILENAME "output.txt"  #include <stdio.h>  #include <stdlib.h>  **struct** edge {  int from;  int to;  int weight;  };  **typedef** **struct** \_node\* node\_pointer;  **struct** \_node {  int vertex;  int weight;  node\_pointer link;  };  node\_pointer\* graph;  node\_pointer\* mst;  int vertex\_cnt;  int edge\_cnt;  int\* visited;  int cost = 0;  FILE\* ofp;  */\*\**  *\* Reads data from input file*  *\*/*  void readfile(void) {  FILE\* fp;  node\_pointer **new**;  int weight;  **if**(!(fp = fopen(INPUT\_FILENAME, "r"))) {  fprintf(stderr, "error: fopen failed to read**\n**");  exit(1);  }  fscanf(fp, " %d", &vertex\_cnt);  graph = calloc(vertex\_cnt, **sizeof**(node\_pointer));  visited = calloc(vertex\_cnt, **sizeof**(int));  *//reading upper triangle only*  edge\_cnt = 0;  **for**(int i=0 ; i<vertex\_cnt ; i++) {  **for**(int j=0 ; j<vertex\_cnt ; j++) {  fscanf(fp, " %d", &weight);  **if**(weight != -1) {  **if**(i > j) **continue**;  edge\_cnt++;  **new** = malloc(**sizeof**(**struct** \_node));  **new**->vertex = j;  **new**->weight = weight;  **new**->link = graph[i];  graph[i] = **new**;  **new** = NULL;  }  }  }  fclose(fp);  }  */\*\**  *\* Sort edges in non-decreasing order.*  *\* @param edges Set of edges to sort*  *\* @param n Number of edges in set*  *\*/*  void sort\_edges(**struct** edge\*\* edges, int n) {  **struct** edge\* tmp;  **for**(int i=0 ; i<n-1 ; i++) {  **for**(int j=i ; j<n ; j++) {  **if**(edges[i]->weight > edges[j]->weight) {  tmp = edges[j];  edges[j] = edges[i];  edges[i] = tmp;  }  }  }  }  */\*\**  *\* Find function of union-find*  *\* @param parent Array holding information on sets*  *\* @param i Index to find set for*  *\* @return Found set*  *\*/*  int sfind(int parent[], int i) {  **while**(parent[i] >= 0)  i=parent[i];  **return** i;  }  */\*\**  *\* Union function of union-find*  *\* @param parent Array holding info on sets*  *\* @param i Index to union*  *\* @param j Index to union*  *\*/*  void sunion(int parent[], int i, int j) {  parent[i] = j;  }  */\*\**  *\* Create minimum spanning tree.*  *\*/*  void create\_mst(void) {  **struct** edge\*\* edges;  int x, y, edge\_idx = 0;  int\* parent;    edges = malloc(**sizeof**(**struct** edge\*)\*edge\_cnt);  **for**(int i=0 ; i<vertex\_cnt ; i++) {  **for**(node\_pointer ptr = graph[i] ; ptr ; ptr = ptr->link) {  edges[edge\_idx] = malloc(**sizeof**(**struct** edge));  edges[edge\_idx]->from = i;  edges[edge\_idx]->to = ptr->vertex;  edges[edge\_idx]->weight = ptr->weight;  edge\_idx++;  }  }  sort\_edges(edges, edge\_cnt);  mst = calloc(vertex\_cnt, **sizeof**(node\_pointer\*));  parent = malloc(**sizeof**(int)\*vertex\_cnt);  **for**(int i=0 ; i<vertex\_cnt ; i++) {  parent[i] = -1;  }  edge\_idx = 0;  **while**(edge\_idx < edge\_cnt) {  **struct** edge\* next = edges[edge\_idx++];  x = sfind(parent, next->from);  y = sfind(parent, next->to);  **if**(x != y) {  node\_pointer **new**;  **new** = malloc(**sizeof**(**struct** \_node));  **new**->link = mst[next->from];  **new**->vertex = next->to;  **new**->weight = next->weight;  mst[next->from] = **new**;  *// since the matrix is non-directional, add an edge with reverse dir.*  **new** = malloc(**sizeof**(**struct** \_node));  **new**->link = mst[next->to];  **new**->vertex = next->from;  **new**->weight = next->weight;  mst[next->to] = **new**;  sunion(parent, x, y);  cost += next->weight;  }  }  **for**(int i=0 ; i<edge\_cnt ; i++) {  free(edges[i]);  }  free(edges);  free(parent);  }  */\*\**  *\* Executes depth-first search starting from a vertex*  *\* @param vertex The vertex number to start dfs from*  *\*/*  void dfs(int vertex) {  fprintf(ofp, "%d", vertex);  visited[vertex] = 1;  **for**(node\_pointer ptr = mst[vertex] ; ptr ; ptr = ptr->link) {  **if**(!visited[ptr->vertex]) {  fprintf(ofp, " ");  dfs(ptr->vertex);  }  }  }  int main(int argc, **const** char\* argv[]) {  readfile();  create\_mst();  **if**(!(ofp = fopen(OUTPUT\_FILENAME, "w"))) {  fprintf(stderr, "error: fopen cannot write to file**\n**");  exit(1);  }  visited = calloc(vertex\_cnt, **sizeof**(int));  **for**(int i=0 ; i<vertex\_cnt ; i++) {  **if**(!visited[i]) {  dfs(i);  fprintf(ofp, "**\n**");  }  }  fprintf(ofp, "%d**\n**", cost);  fclose(ofp);  **for**(int i=0 ; i<vertex\_cnt ; i++) {  free(graph[i]);  free(mst[i]);  }  free(graph);  free(mst);  **return** 0;  } |