Universidad Tecnológica de Pereira

CAPÍTULO COMPLETO: GRAFOS Y LABERINTOS
CURSO DE ESTRUCTURA DE DATOS - Código IS304 - Grupo 01

Programa de Ingeniería de Sistemas y Computación
Profesor Hugo Humberto Morales Peña
Lunes 10 de Diciembre de 2021

Grafos no dirigidos sin pesos en las aristas:

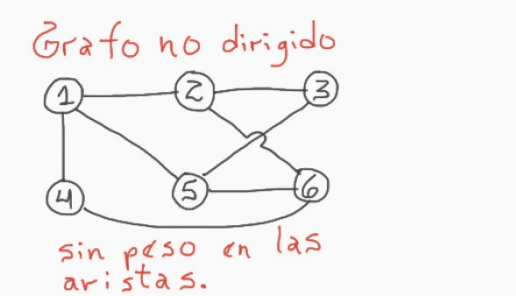


Figura 1: Ejemplo de grafo no dirigido sin peso en las aristas

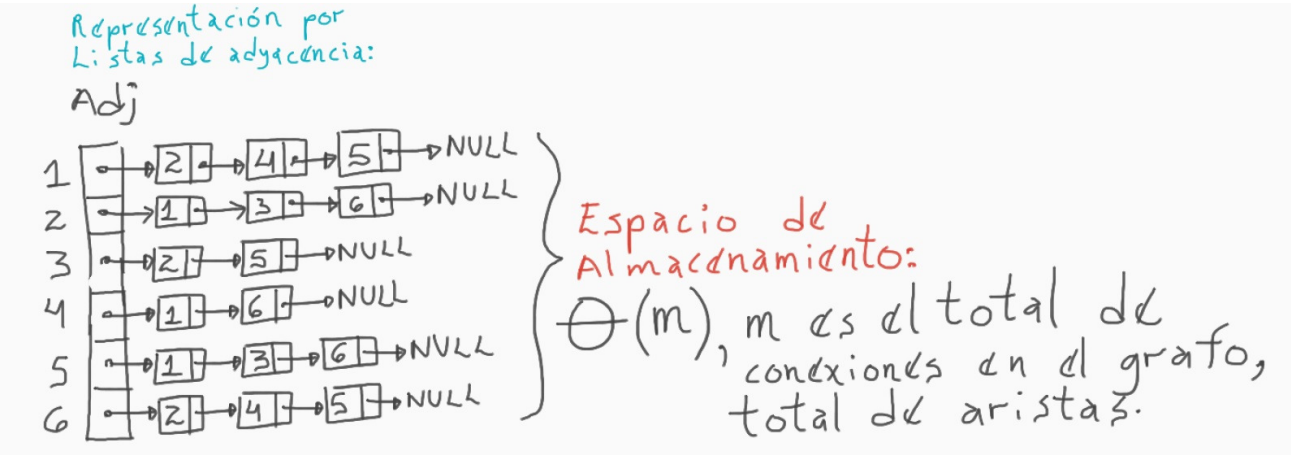


Figura 2: Representación por listas de adyacencia del grafo no dirigido sin peso en las aristas

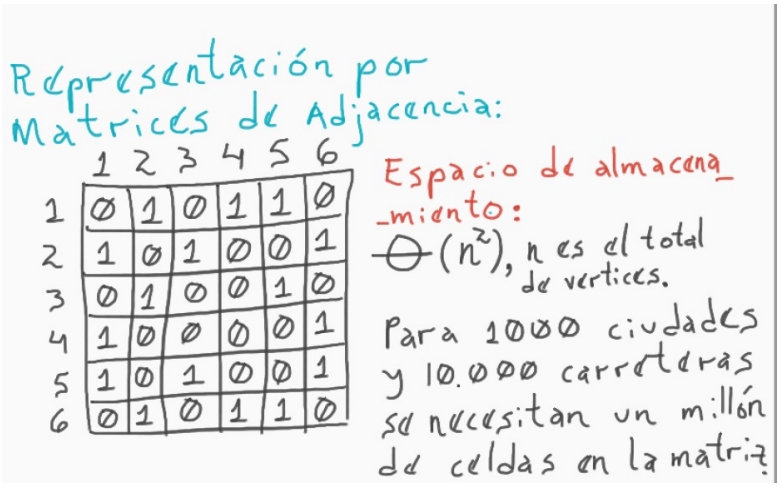


Figura 3: Representación por matices de adyacencia del grafo no dirigido sin peso en las aristas

Grafos no dirigidos con pesos en las aristas:

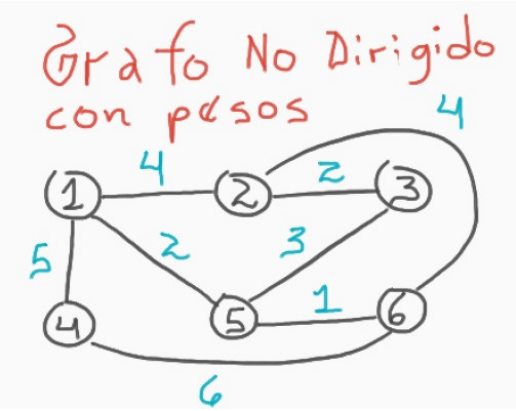


Figura 4: Ejemplo de grafo no dirigido con pesos en las aristas

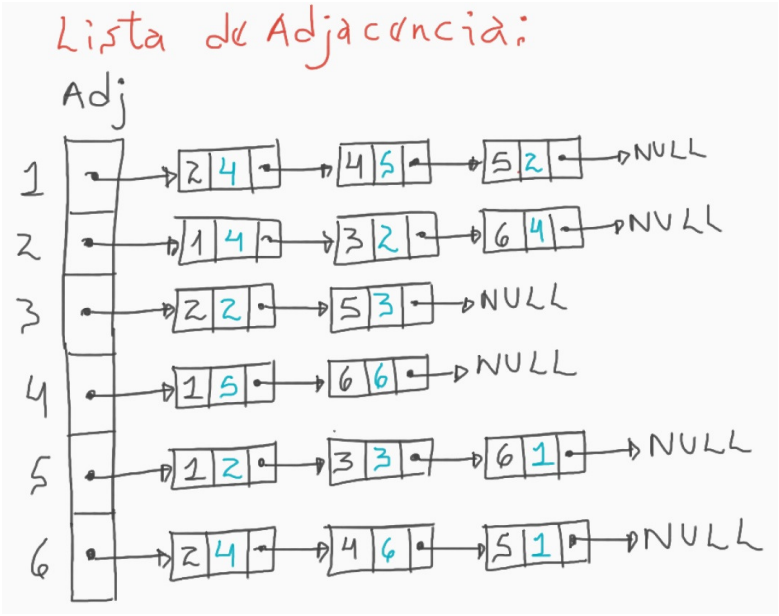


Figura 5: Representación por listas de adyacencia del grafo no dirigido con pesos en las aristas

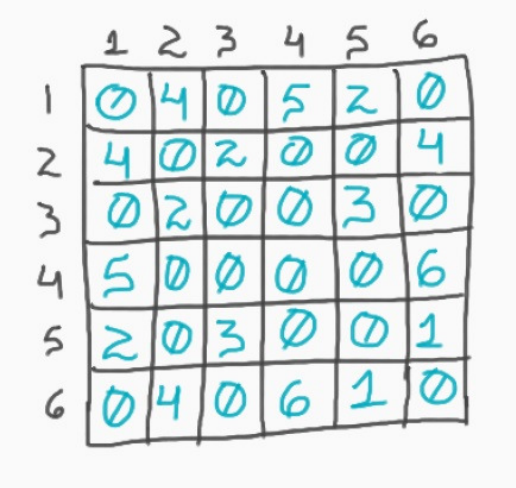


Figura 6: Representación por matices de adyacencia del grafo no dirigido con pesos en las aristas

Implementación de Grafos por medio de listas de adyacencia

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXV 1005


struct edge
{
    int vertex;
/*  int weight;  */
    struct edge *next;
};
```

```c
struct graph
{
    int n_vertex;
    int m_edges;
    struct edge *Adj[MAXV];
};


struct graph *ReadGraph(int vertexes, int edges)
{
    int idVertex, idEdge, u, v;
    struct graph *G;
    struct edge *tempEdge;

    G = (struct graph *) malloc(sizeof(struct graph));
    G->n_vertex = vertexes;
    G->m_edges = edges;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        G->Adj[idVertex] = NULL;

    for(idEdge = 1; idEdge <= G->m_edges; idEdge++)
    {
        scanf("%d %d", &u, &v);
        tempEdge = (struct edge *) malloc(sizeof(struct edge));
        tempEdge->vertex = v;
        tempEdge->next = G->Adj[u];
        G->Adj[u] = tempEdge;

        if(u != v)
        {
            tempEdge = (struct edge *) malloc(sizeof(struct edge));
            tempEdge->vertex = u;
            tempEdge->next = G->Adj[v];
            G->Adj[v] = tempEdge;
        }
    }
    return G;
}


void PrintGraph(struct graph *G)
{
    int idVertex;
    struct edge *tempEdge;

    if(G != NULL)
    {
        printf("Representation for graph's adjacent lists: \n");
        for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        {
            printf("[%d]: ", idVertex);
            tempEdge = G->Adj[idVertex];
            while(tempEdge != NULL)
            {
                printf(" -> %d", tempEdge->vertex);
                tempEdge = tempEdge->next;
            }
            printf(" -> NULL\n");
        }
    }
    else
        printf("Empty Graph.\n");
}
```

```c
struct graph *DeleteGraph(struct graph *G)
{
    int idVertex;
    struct edge *ActualEdge, *NextEdge;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        ActualEdge = G->Adj[idVertex];
        while(ActualEdge != NULL)
        {
            NextEdge = ActualEdge->next;
            free(ActualEdge);
            ActualEdge = NextEdge;
        }
    }
    free(G);
    G = NULL;
    return G;
}


int main()
{
    int vertexes, edges;
    struct graph *G;

    while(scanf("%d %d", &vertexes, &edges) != EOF)
    {
        G = ReadGraph(vertexes, edges);
        PrintGraph(G);
        G = DeleteGraph(G);
        PrintGraph(G);
    }
    return 0;
}
```

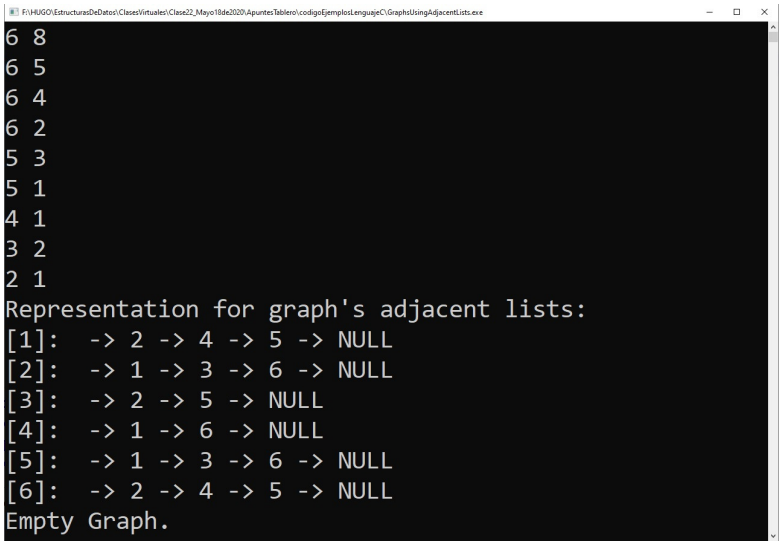


Figura 7: Salida del programa que representa grafos por medio de listas de adyacencia.

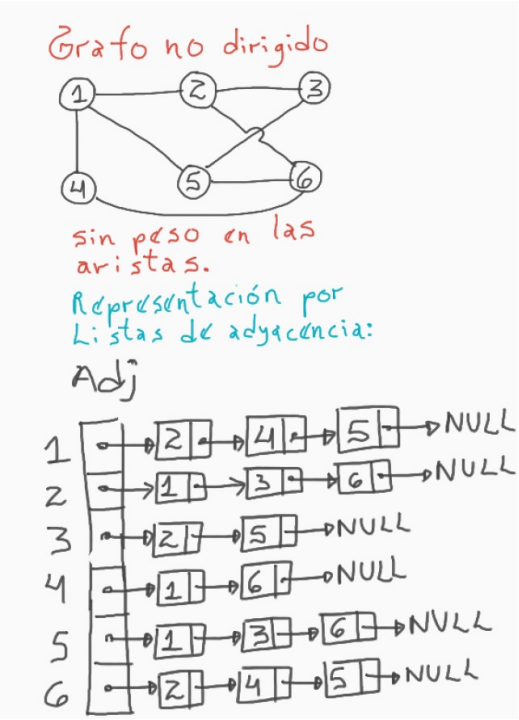


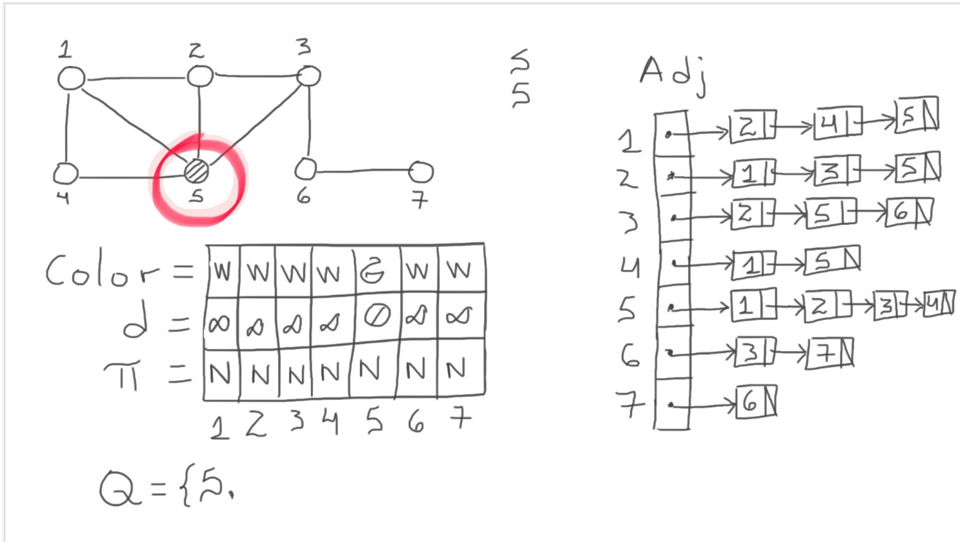
Figura 8: Grafo original que se utilizó en la salida del programa anterior.

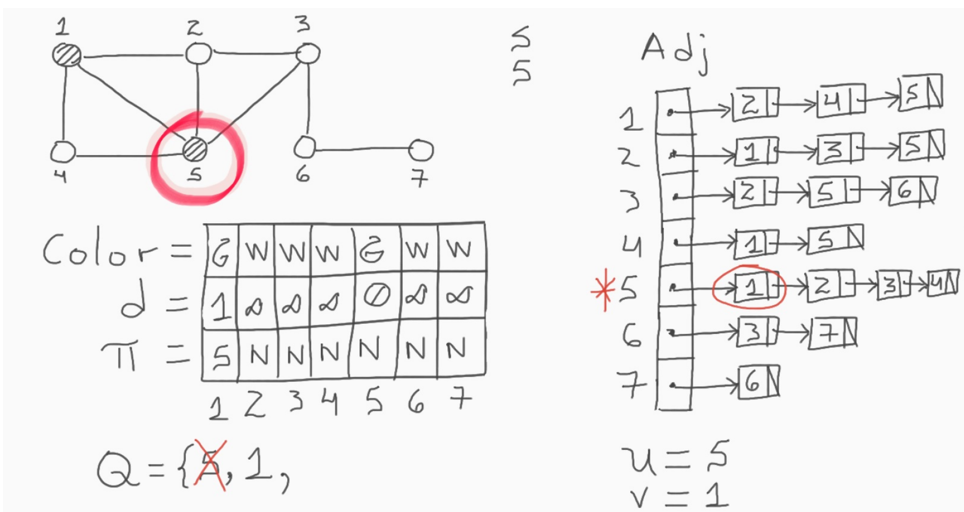
Búsqueda Primero en Amplitud

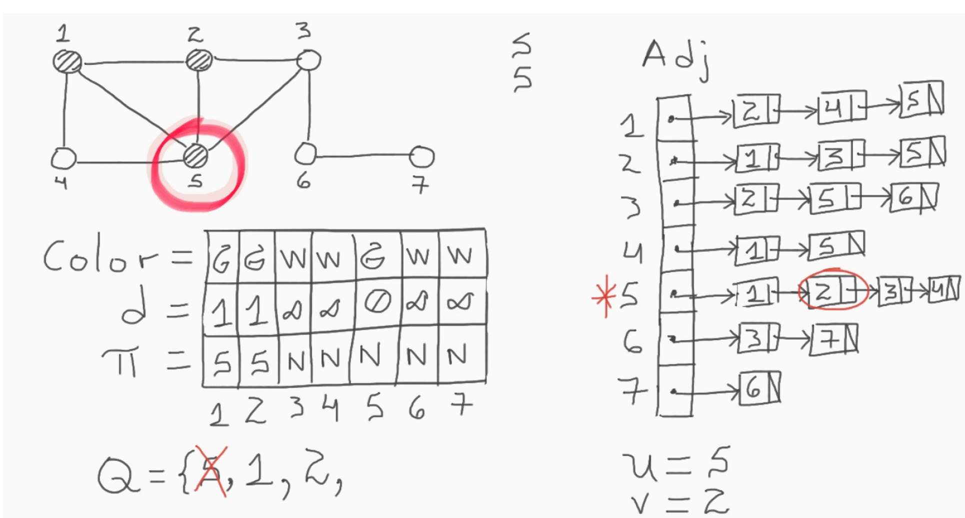
Algoritmo Breadth-First Search

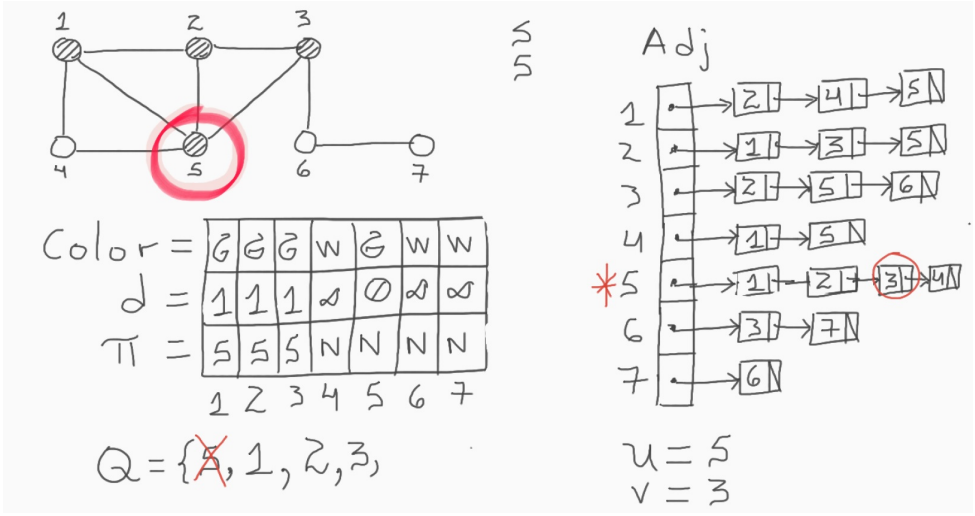
function BFS(G, s)

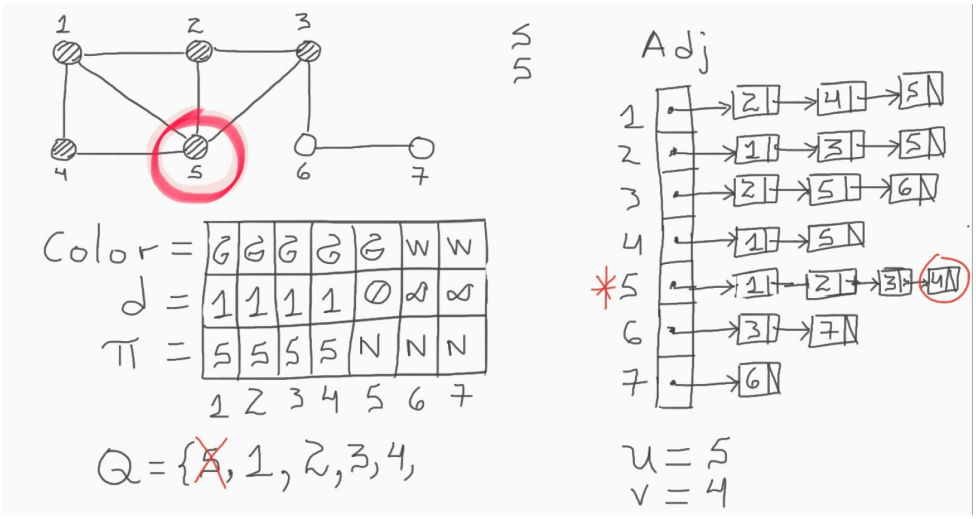
```
1   for each vertex u ∈ V[G] − {s} do
2       color[u] ← WHITE
3       d[u] ← ∞
4       π[u] ← NIL
5   color[s] ← GRAY
6   d[s] ← 0
7   π[s] ← NIL
8   Q ← { }
9   ENQUEUE( Q, s )
10  while Q ≠ { } do
11      u ← DEQUEUE( Q )
12      for each vertex v ∈ Adj[u] do
13          if color[v] == WHITE then
14              color[v] ← GRAY
15              d[v] ← d[u] + 1
16              π[v] ← u
17              ENQUEUE( Q, v )
18      color[u] ← BLACK
```

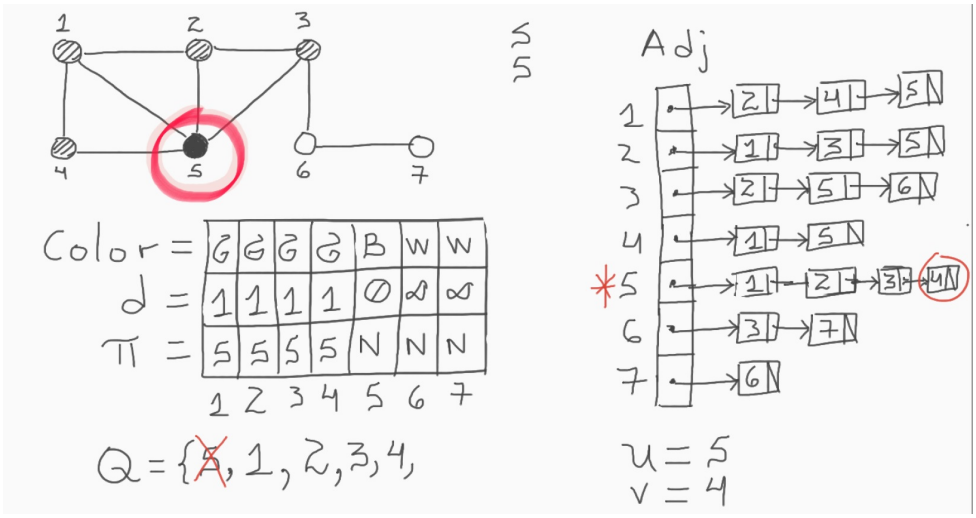

1 2 3 4 5 6 7
s 5
Adj
1 → 2 → 4 → 5 N
2 → 1 → 3 → 5 N
3 → 2 → 5 → 6 N
4 → 1 → 5 N
5 → 1 → 2 → 3 → 4 N
6 → 3 → 7 N
7 → 6 N
Color = W W W W G W W
d = ∞ ∞ ∞ ∞ ∅ ∞ ∞
π = N N N N N N N
1 2 3 4 5 6 7
Q = {5,

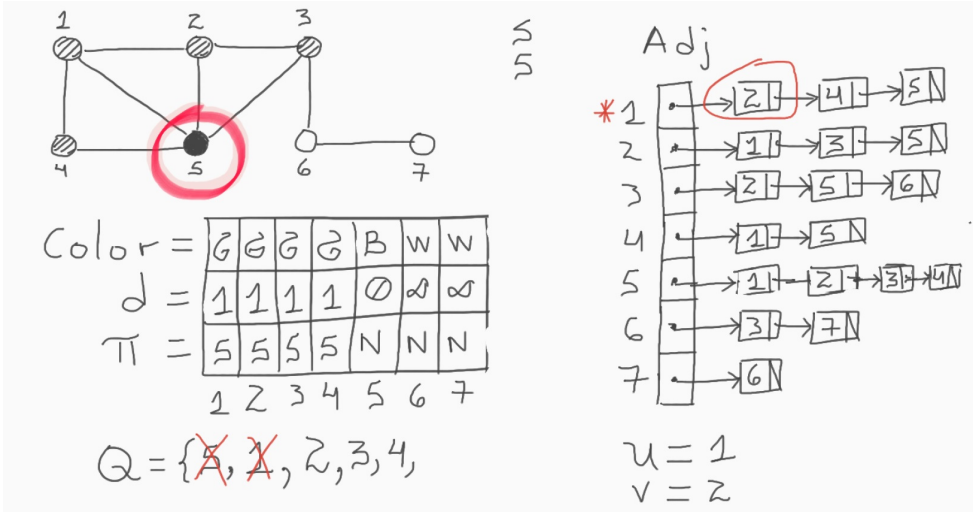

1 2 3 4 5 6 7
s 5
Adj
1 → 2 → 4 → 5 N
2 → 1 → 3 → 5 N
3 → 2 → 5 → 6 N
4 → 1 → 5 N
*5 → 1 → 2 → 3 → 4 N
6 → 3 → 7 N
7 → 6 N
Color = G W W W G W W
d = 1 ∞ ∞ ∞ ∅ ∞ ∞
π = 5 N N N N N N
1 2 3 4 5 6 7
Q = {5, 1,
u = 5
v = 1

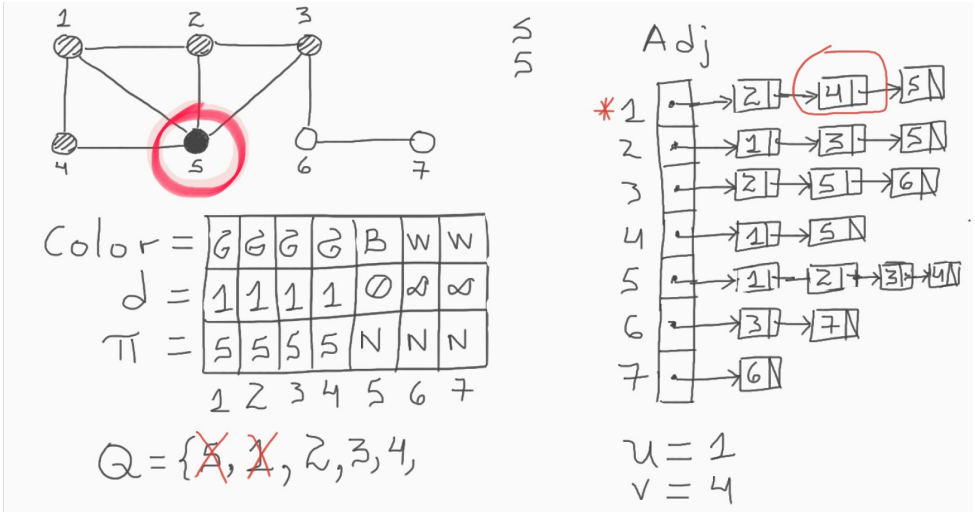

1 2 3 4 5 6 7
s 5
Adj
1 → 2 → 4 → 5 N
2 → 1 → 3 → 5 N
3 → 2 → 5 → 6 N
4 → 1 → 5 N
*5 → 1 → 2 → 3 → 4 N
6 → 3 → 7 N
7 → 6 N
Color = G G W W G W W
d = 1 1 ∞ ∞ ∅ ∞ ∞
π = 5 5 N N N N N
1 2 3 4 5 6 7
Q = {5, 1, 2,
u = 5
v = 2

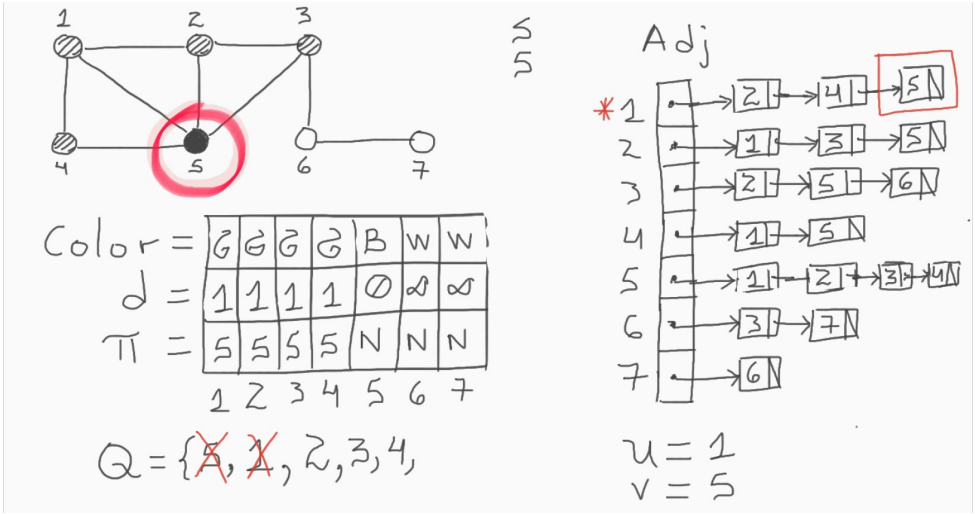

1 2 3 4 5 6 7
Color = G G G W G W W
d = 1 1 1 ∞ ∅ ∞ ∞
π = 5 5 5 N N N N
1 2 3 4 5 6 7
Q = {5, 1, 2, 3,
Adj
1 → 2 → 4 → 5N
2 → 1 → 3 → 5N
3 → 2 → 5 → 6N
4 → 1 → 5N
*5 → 1 → 2 → 3 → 4N
6 → 3 → 7N
7 → 6N
u = 5
v = 3

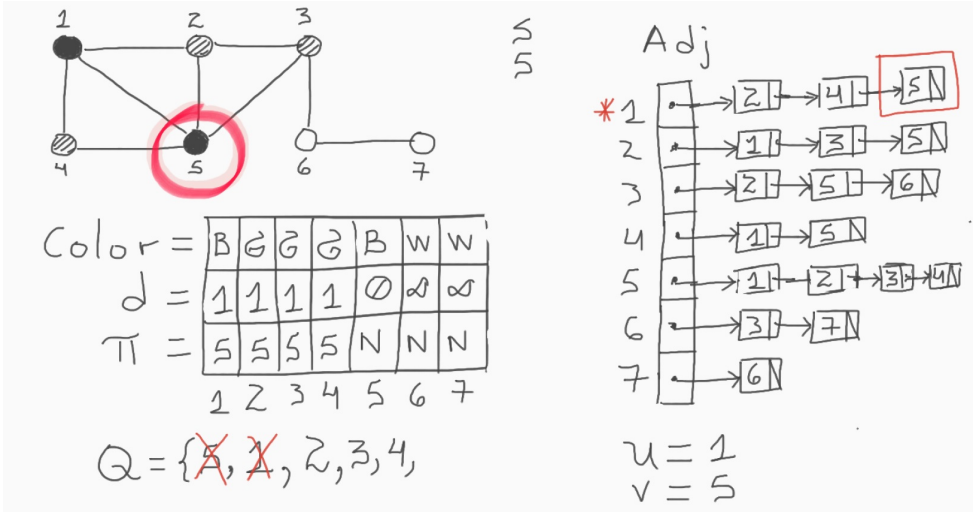

1 2 3 4 5 6 7
Color = G G G G G W W
d = 1 1 1 1 ∅ ∞ ∞
π = 5 5 5 5 N N N
1 2 3 4 5 6 7
Q = {5, 1, 2, 3, 4,
Adj
1 → 2 → 4 → 5N
2 → 1 → 3 → 5N
3 → 2 → 5 → 6N
4 → 1 → 5N
*5 → 1 → 2 → 3 → 4N
6 → 3 → 7N
7 → 6N
u = 5
v = 4

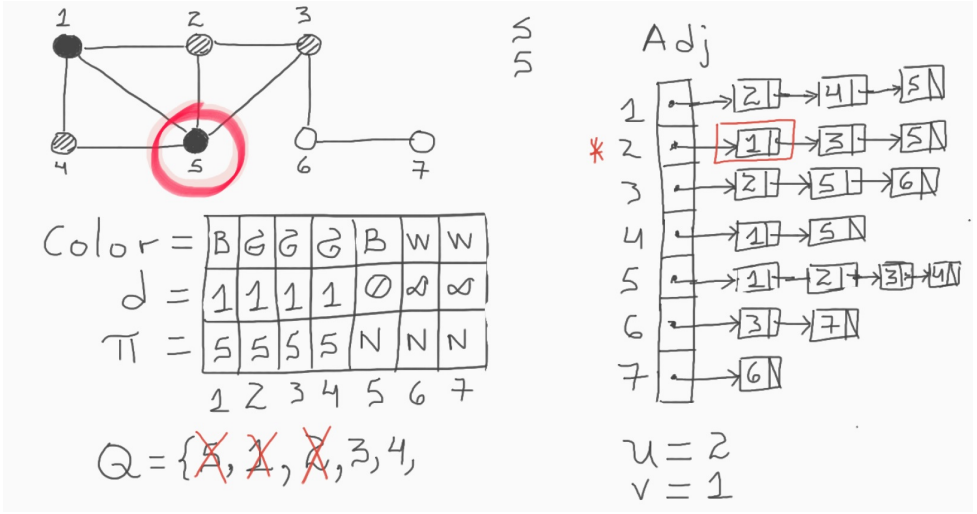

1 2 3 4 5 6 7
Color = G G G G B W W
d = 1 1 1 1 ∅ ∞ ∞
π = 5 5 5 5 N N N
1 2 3 4 5 6 7
Q = {5, 1, 2, 3, 4,
Adj
1 → 2 → 4 → 5N
2 → 1 → 3 → 5N
3 → 2 → 5 → 6N
4 → 1 → 5N
*5 → 1 → 2 → 3 → 4N
6 → 3 → 7N
7 → 6N
u = 5
v = 4

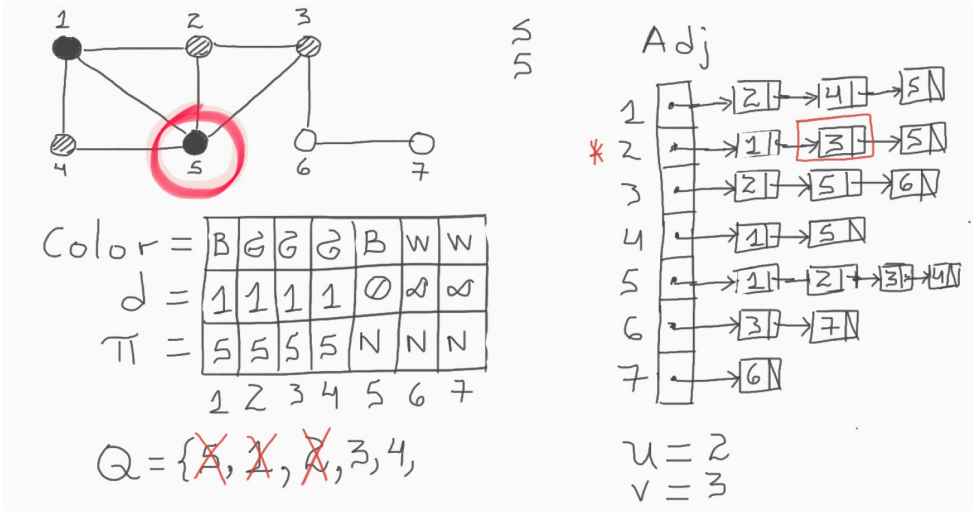

1 2 3 4 5 6 7
Color = G G G G B W W
d = 1 1 1 1 ∅ ∞ ∞
π = 5 5 5 5 N N N
1 2 3 4 5 6 7
Q = {5, 1, 2, 3, 4,
S S
Adj
*1 → 2 → 4 → 5
2 → 1 → 3 → 5
3 → 2 → 5 → 6
4 → 1 → 5
5 → 1 → 2 → 3 → 4
6 → 3 → 7
7 → 6
u = 1
v = 2

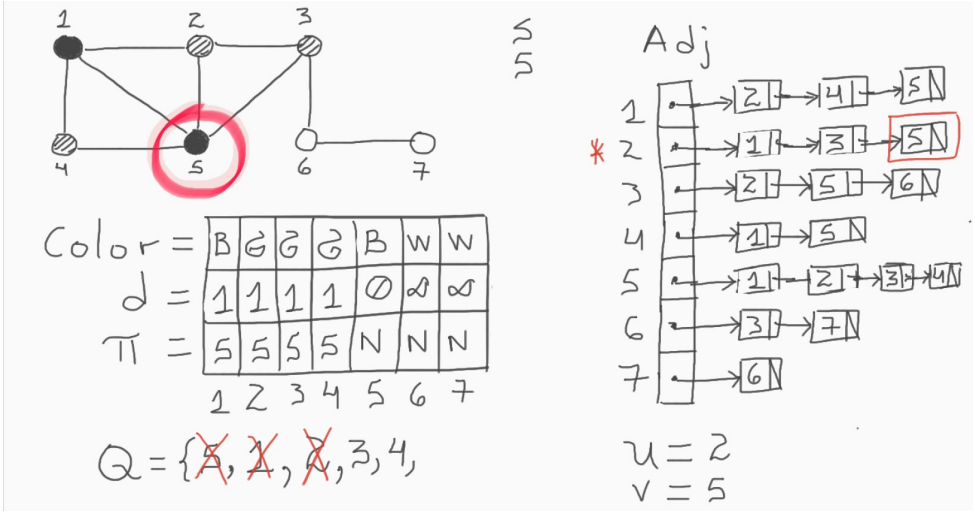

1 2 3 4 5 6 7
Color = G G G G B W W
d = 1 1 1 1 ∅ ∞ ∞
π = 5 5 5 5 N N N
1 2 3 4 5 6 7
Q = {5, 1, 2, 3, 4,
S S
Adj
*1 → 2 → 4 → 5
2 → 1 → 3 → 5
3 → 2 → 5 → 6
4 → 1 → 5
5 → 1 → 2 → 3 → 4
6 → 3 → 7
7 → 6
u = 1
v = 4

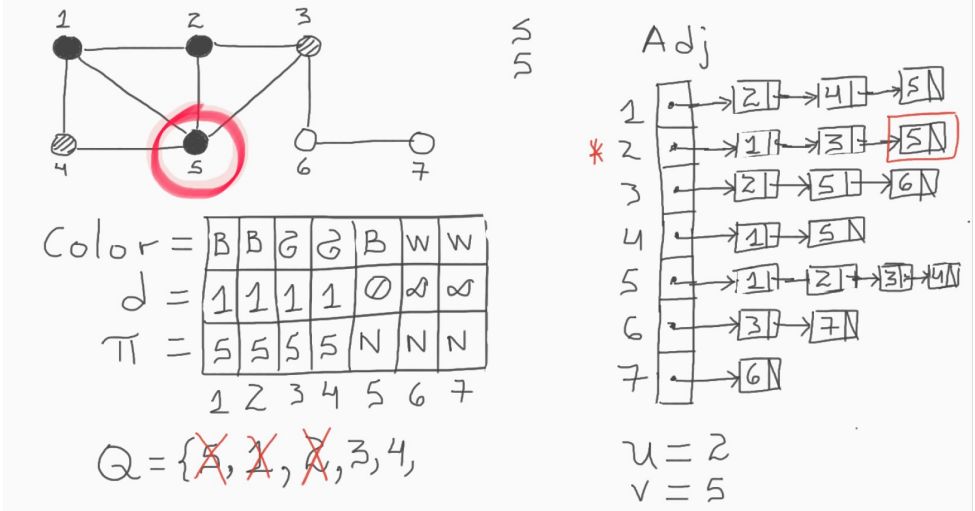

1 2 3 4 5 6 7
Color = G G G G B W W
d = 1 1 1 1 ∅ ∞ ∞
π = 5 5 5 5 N N N
1 2 3 4 5 6 7
Q = {5, 1, 2, 3, 4,
S S
Adj
*1 → 2 → 4 → 5
2 → 1 → 3 → 5
3 → 2 → 5 → 6
4 → 1 → 5
5 → 1 → 2 → 3 → 4
6 → 3 → 7
7 → 6
u = 1
v = 5

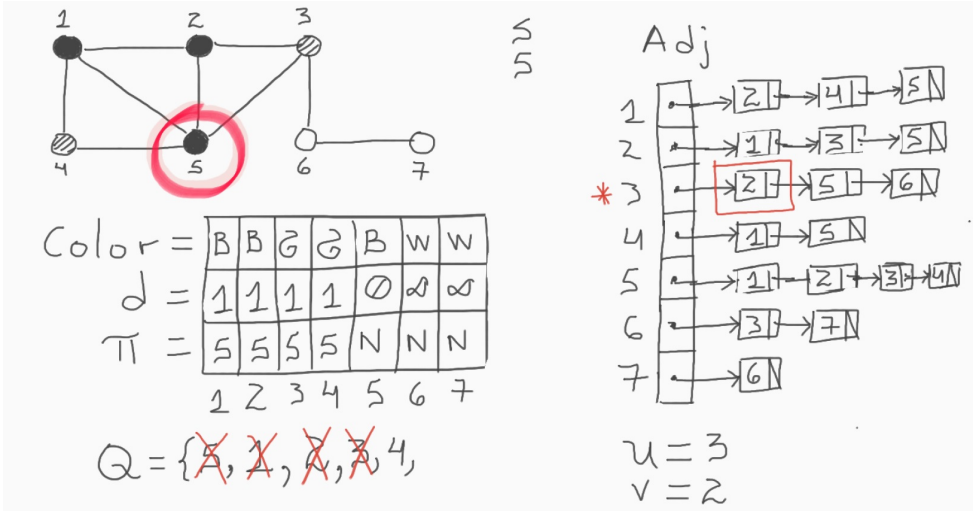

1 2 3 4 5 6 7
Color = B G G G B W W
d = 1 1 1 1 ∞ ∞ ∞
π = 5 5 5 5 N N N
1 2 3 4 5 6 7
Q = {5, 1, 2, 3, 4,
s
s
Adj
1 → 2 → 4 → 5N
2 → 1 → 3 → 5N
3 → 2 → 5 → 6N
4 → 1 → 5N
5 → 1 → 2 → 3 → 4N
6 → 3 → 7N
7 → 6N
u = 1
v = 5

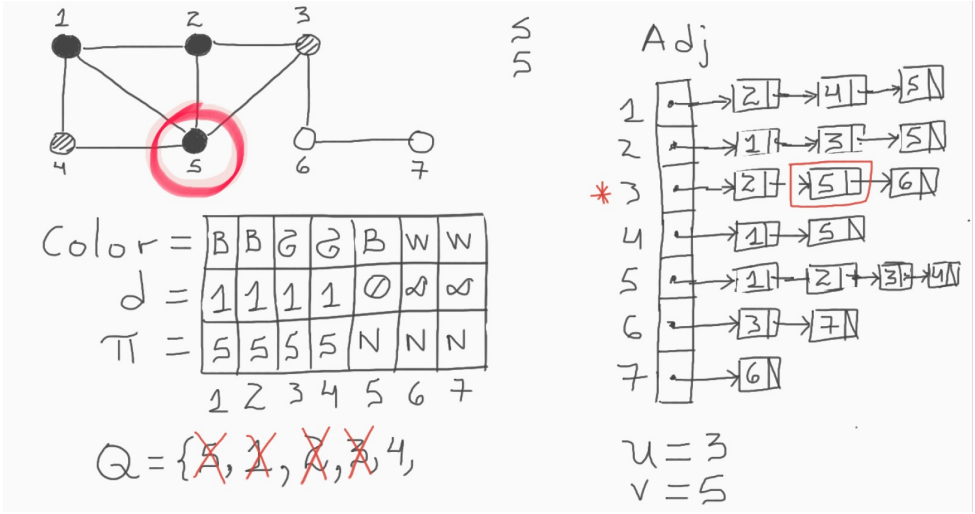

1 2 3 4 5 6 7
Color = B G G G B W W
d = 1 1 1 1 ∞ ∞ ∞
π = 5 5 5 5 N N N
1 2 3 4 5 6 7
Q = {5, 1, 2, 3, 4,
s
s
Adj
1 → 2 → 4 → 5N
2 → 1 → 3 → 5N
3 → 2 → 5 → 6N
4 → 1 → 5N
5 → 1 → 2 → 3 → 4N
6 → 3 → 7N
7 → 6N
u = 2
v = 1

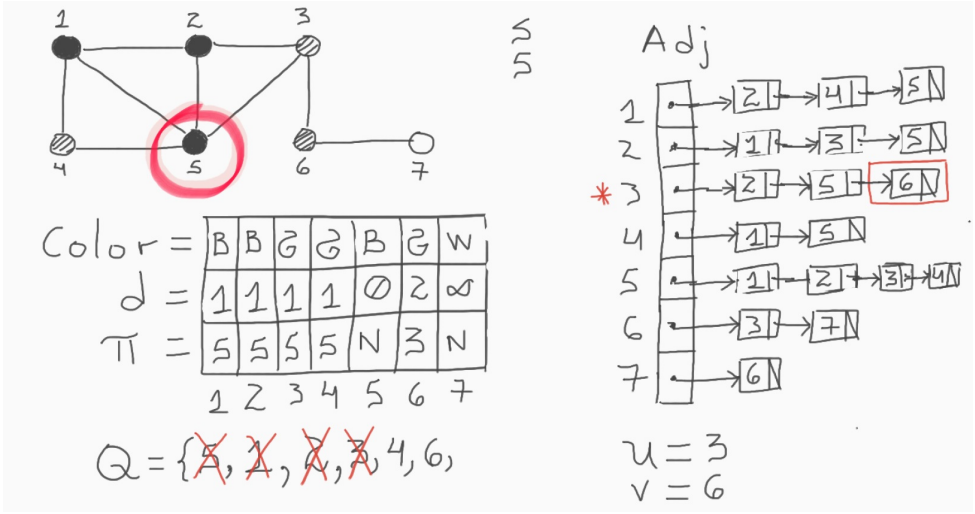

1 2 3 4 5 6 7
Color = B G G G B W W
d = 1 1 1 1 ∞ ∞ ∞
π = 5 5 5 5 N N N
1 2 3 4 5 6 7
Q = {5, 1, 2, 3, 4,
s
s
Adj
1 → 2 → 4 → 5N
2 → 1 → 3 → 5N
3 → 2 → 5 → 6N
4 → 1 → 5N
5 → 1 → 2 → 3 → 4N
6 → 3 → 7N
7 → 6N
u = 2
v = 3

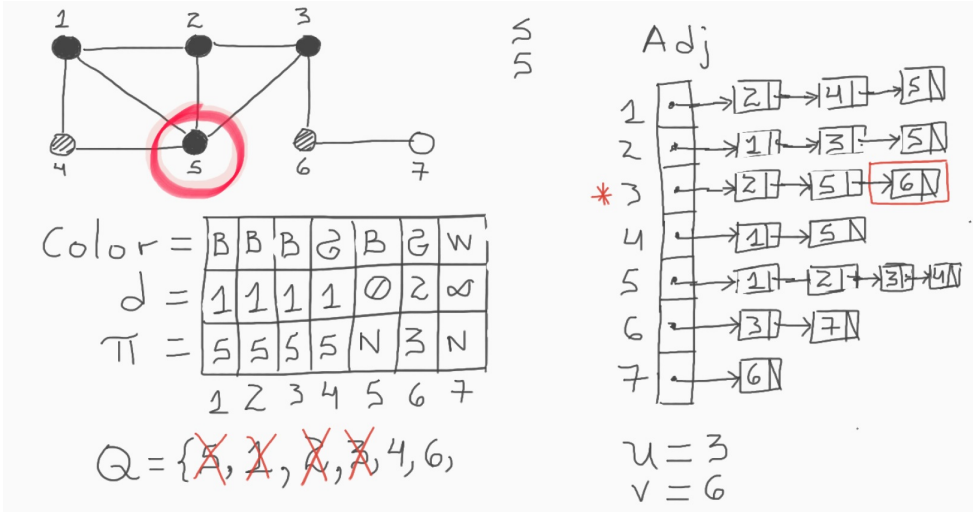

1 2 3 4 5 6 7
Color = B G G G B W W
d = 1 1 1 1 ∞ ∞ ∞
π = 5 5 5 5 N N N
1 2 3 4 5 6 7
Q = {5, 2, 4, 3, 6,
u = 2
v = 5
Adj
1 → 2 → 4 → 5N
2 → 1 → 3 → 5N
3 → 2 → 5 → 6N
4 → 1 → 5N
5 → 1 → 2 → 3 → 4N
6 → 3 → 7N
7 → 6N

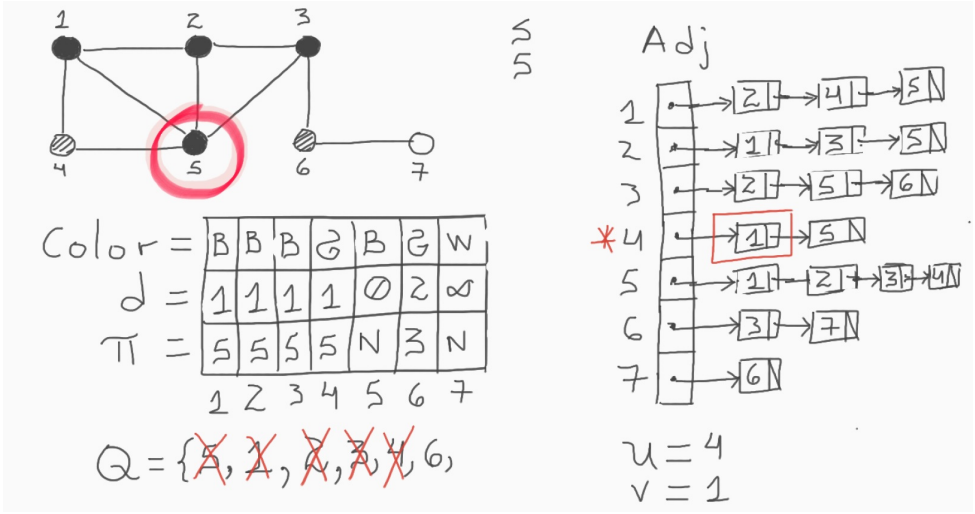

1 2 3 4 5 6 7
Color = B B G G B W W
d = 1 1 1 1 ∞ ∞ ∞
π = 5 5 5 5 N N N
1 2 3 4 5 6 7
Q = {5, 2, 4, 3, 6,
u = 2
v = 5
Adj
1 → 2 → 4 → 5N
2 → 1 → 3 → 5N
3 → 2 → 5 → 6N
4 → 1 → 5N
5 → 1 → 2 → 3 → 4N
6 → 3 → 7N
7 → 6N

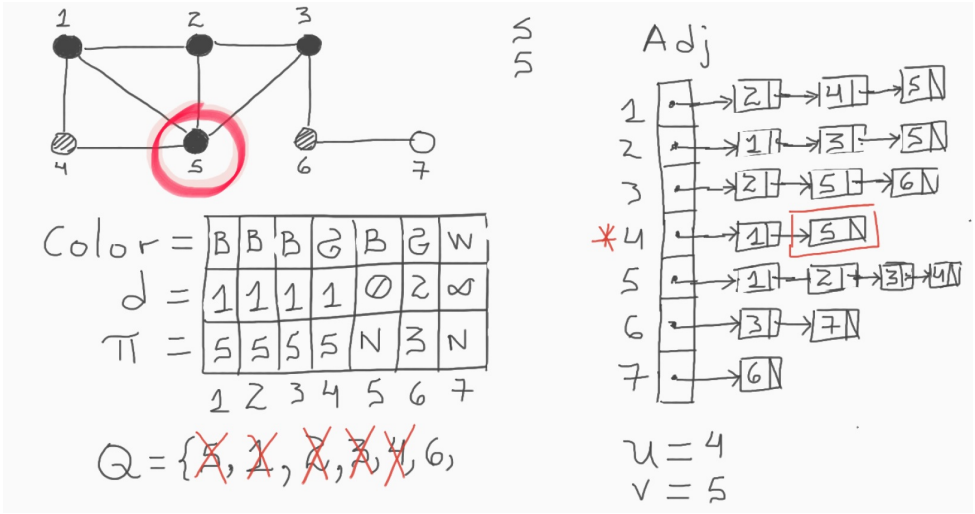

1 2 3 4 5 6 7
Color = B B G G B W W
d = 1 1 1 1 ∞ ∞ ∞
π = 5 5 5 5 N N N
1 2 3 4 5 6 7
Q = {5, 2, 4, 3, 6,
u = 3
v = 2
Adj
1 → 2 → 4 → 5N
2 → 1 → 3 → 5N
3 → 2 → 5 → 6N
4 → 1 → 5N
5 → 1 → 2 → 3 → 4N
6 → 3 → 7N
7 → 6N

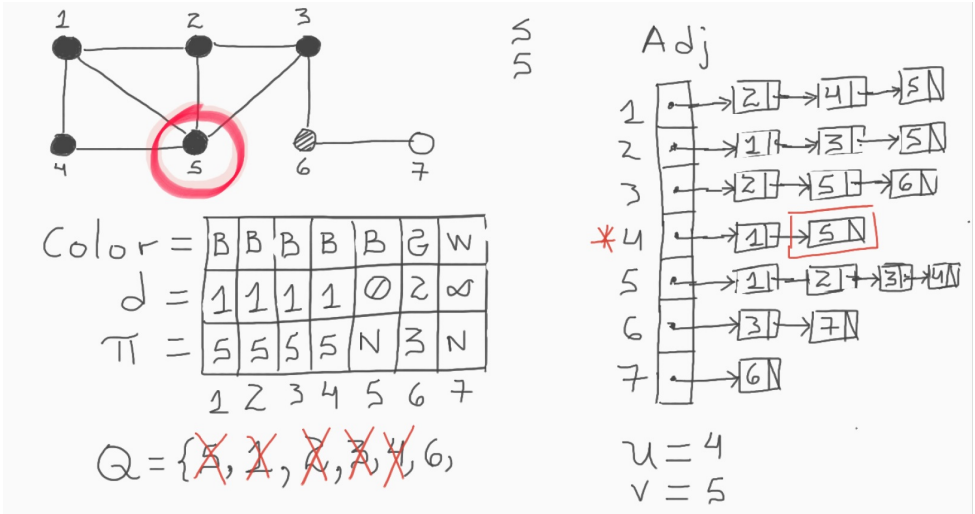

1 2 3 4 5 6 7
Color = B B G G B W W
d = 1 1 1 1 ∞ ∞ ∞
π = 5 5 5 5 N N N
1 2 3 4 5 6 7
Q = {5, 1, 4, 3, 2, 4,
Adj
1 → 2 → 4 → 5 N
2 → 1 → 3 → 5 N
* 3 → 2 → 5 → 6 N
4 → 1 → 5 N
5 → 1 → 2 → 3 → 4 N
6 → 3 → 7 N
7 → 6 N
u = 3
v = 5

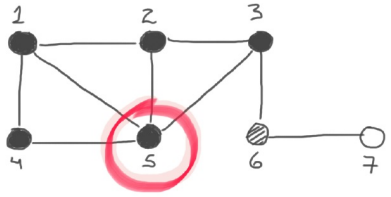

1 2 3 4 5 6 7
Color = B B G G B G W
d = 1 1 1 1 ∞ 2 ∞
π = 5 5 5 5 N 3 N
1 2 3 4 5 6 7
Q = {5, 1, 4, 3, 2, 4, 6,
Adj
1 → 2 → 4 → 5 N
2 → 1 → 3 → 5 N
* 3 → 2 → 5 → 6 N
4 → 1 → 5 N
5 → 1 → 2 → 3 → 4 N
6 → 3 → 7 N
7 → 6 N
u = 3
v = 6


1 2 3 4 5 6 7
Color = B B B G B G W
d = 1 1 1 1 ∞ 2 ∞
π = 5 5 5 5 N 3 N
1 2 3 4 5 6 7
Q = {5, 1, 4, 3, 2, 4, 6,
Adj
1 → 2 → 4 → 5 N
2 → 1 → 3 → 5 N
* 3 → 2 → 5 → 6 N
4 → 1 → 5 N
5 → 1 → 2 → 3 → 4 N
6 → 3 → 7 N
7 → 6 N
u = 3
v = 6


1 2 3
4 5 6 7
Color = B B B G B G W
d = 1 1 1 1 ∅ 2 ∞
π = 5 5 5 5 N 3 N
1 2 3 4 5 6 7
Q = {5, 1, 4, 3, 4, 6,
Adj
1 → 2 → 4 → 5N
2 → 1 → 3 → 5N
3 → 2 → 5 → 6N
*4 → 1 → 5N
5 → 1 → 2 → 3 → 4N
6 → 3 → 7N
7 → 6N
u = 4
v = 1


1 2 3
4 5 6 7
Color = B B B G B G W
d = 1 1 1 1 ∅ 2 ∞
π = 5 5 5 5 N 3 N
1 2 3 4 5 6 7
Q = {5, 1, 4, 3, 4, 6,
Adj
1 → 2 → 4 → 5N
2 → 1 → 3 → 5N
3 → 2 → 5 → 6N
*4 → 1 → 5N
5 → 1 → 2 → 3 → 4N
6 → 3 → 7N
7 → 6N
u = 4
v = 5


1 2 3
4 5 6 7
Color = B B B B B G W
d = 1 1 1 1 ∅ 2 ∞
π = 5 5 5 5 N 3 N
1 2 3 4 5 6 7
Q = {5, 1, 4, 3, 4, 6,
Adj
1 → 2 → 4 → 5N
2 → 1 → 3 → 5N
3 → 2 → 5 → 6N
*4 → 1 → 5N
5 → 1 → 2 → 3 → 4N
6 → 3 → 7N
7 → 6N
u = 4
v = 5

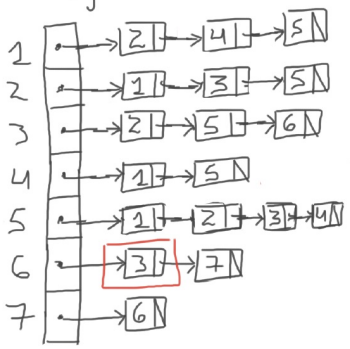


	1	2	3	4	5	6	7
Color =	B	B	B	B	B	G	W
d =	1	1	1	1	∅	2	∞
π =	5	5	5	5	N	3	N

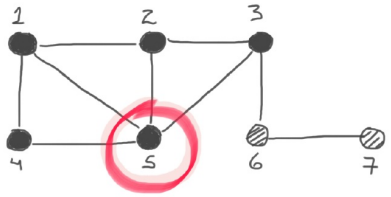
Q = {~~5~~, ~~1~~, ~~2~~, ~~3~~, ~~4~~, 6,

S
S

Adj



u = 6
v = 3

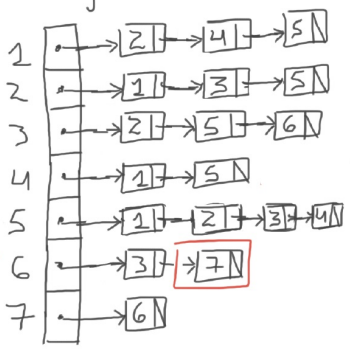


	1	2	3	4	5	6	7
Color =	B	B	B	B	B	G	G
d =	1	1	1	1	∅	2	3
π =	5	5	5	5	N	3	6

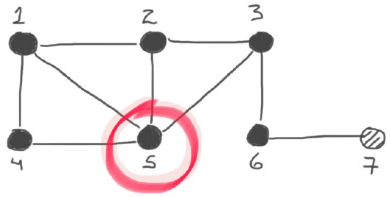
Q = {~~5~~, ~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~6~~, 7

S
S

Adj



u = 6
v = 7

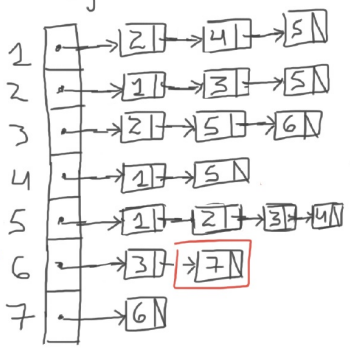


	1	2	3	4	5	6	7
Color =	B	B	B	B	B	B	G
d =	1	1	1	1	∅	2	3
π =	5	5	5	5	N	3	6

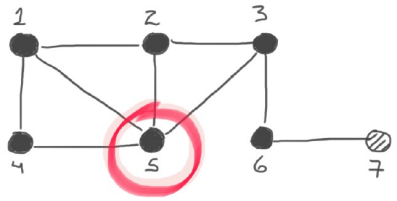
Q = {~~5~~, ~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~6~~, 7

S
S

Adj



u = 6
v = 7



	1	2	3	4	5	6	7
Color =	B	B	B	B	B	B	G
d =	1	1	1	1	∞	2	3
π =	5	5	5	5	N	3	6

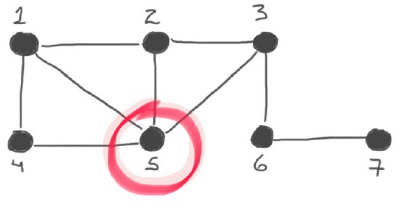
$Q = \{$ ~~5~~, ~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~6~~, ~~7~~ $\}$

S
S

Adj

1 → 2 → 4 → 5 N
2 → 1 → 3 → 5 N
3 → 2 → 5 → 6 N
4 → 1 → 5 N
5 → 1 → 2 → 3 → 4 N
6 → 3 → 7 N
* 7 → 6 N

u = 7
v = 6



	1	2	3	4	5	6	7
Color =	B	B	B	B	B	B	B
d =	1	1	1	1	∞	2	3
π =	5	5	5	5	N	3	6

$Q = \{$ ~~5~~, ~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~6~~, ~~7~~ $\}$

S
S

Adj

1 → 2 → 4 → 5 N
2 → 1 → 3 → 5 N
3 → 2 → 5 → 6 N
4 → 1 → 5 N
5 → 1 → 2 → 3 → 4 N
6 → 3 → 7 N
* 7 → 6 N

u = 7

Implementación de la Búsqueda en Amplitud sobre Grafos representados por medio de listas de adyacencia

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define WHITE 2
#define GRAY 1
#define BLACK 0
#define NIL -1
#define myInfinite 2147483647
#define MAXV 1005

struct edge
{
    int vertex;
/*  int weight;  */
    struct edge *next;
};

struct graph
{
    int n_vertex;
    int m_edges;
    struct edge *Adj[MAXV];
};

struct graph *ReadGraph(int vertexes, int edges)
{
    int idVertex, idEdge, u, v;
    struct graph *G;
    struct edge *tempEdge;

    G = (struct graph *) malloc(sizeof(struct graph));
    G->n_vertex = vertexes;
    G->m_edges = edges;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        G->Adj[idVertex] = NULL;

    for(idEdge = 1; idEdge <= G->m_edges; idEdge++)
    {
        scanf("%d %d", &u, &v);
        tempEdge = (struct edge *) malloc(sizeof(struct edge));
        tempEdge->vertex = v;
        tempEdge->next = G->Adj[u];
        G->Adj[u] = tempEdge;

        if(u != v)
        {
            tempEdge = (struct edge *) malloc(sizeof(struct edge));
            tempEdge->vertex = u;
            tempEdge->next = G->Adj[v];
            G->Adj[v] = tempEdge;
        }
    }
    return G;
}

void PrintGraph(struct graph *G)
{
    int idVertex;
    struct edge *tempEdge;

    if(G != NULL)
    {
        printf("Representation for graph's adjacent lists: \n");
        for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
```

```
        {
            printf("[%d]: ", idVertex);
            tempEdge = G->Adj[idVertex];
            while(tempEdge != NULL)
            {
                printf(" -> %d", tempEdge->vertex);
                tempEdge = tempEdge->next;
            }
            printf(" -> NULL\n");
        }
    }
    else
        printf("Empty Graph.\n");
}

struct graph *DeleteGraph(struct graph *G)
{
    int idVertex;
    struct edge *ActualEdge, *NextEdge;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        ActualEdge = G->Adj[idVertex];
        while(ActualEdge != NULL)
        {
            NextEdge = ActualEdge->next;
            free(ActualEdge);
            ActualEdge = NextEdge;
        }
    }
    free(G);
    G = NULL;
    return G;
}

void BFS(struct graph *G, int s, int color[], int d[], int pi[])
{
    int u, v, Q[MAXV], idHead = 1, idTail = 1;
    struct edge *tempEdge;

    for(u = 1; u <= G->n_vertex; u++)
    {
        color[u] = WHITE;
        d[u] = myInfinite;
        pi[u] = NIL;
    }

    color[s] = GRAY;
    d[s] = 0;
    pi[s] = NIL;
    Q[idTail] = s;
    idTail++;

    while(idHead < idTail)
    {
        u = Q[idHead];
        idHead++;
        tempEdge = G->Adj[u];
        while(tempEdge != NULL)
        {
            v = tempEdge->vertex;
            if(color[v] == WHITE)
            {
                color[v] = GRAY;
                d[v] = d[u] + 1;
                pi[v] = u;
                Q[idTail] = v;
                idTail++;
```

```
            }
            tempEdge = tempEdge->next;
        }
        color[u] = BLACK;
    }
}

void solver(struct graph *G, int s)
{
    int color[MAXV], d[MAXV], pi[MAXV], idVertex;

    BFS(G, s, color, d, pi);

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        if(color[idVertex] == WHITE)
            printf("color[%d] = WHITE\n", idVertex);
        if(color[idVertex] == GRAY)
            printf("color[%d] = GRAY\n", idVertex);
        if(color[idVertex] == BLACK)
            printf("color[%d] = BLACK\n", idVertex);
    }
    printf("\n");

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        printf("d[%d] = %d\n", idVertex, d[idVertex]);
    printf("\n");

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        if(pi[idVertex] == NIL)
            printf("pi[%d] = NIL\n", idVertex);
        else
            printf("pi[%d] = %d\n", idVertex, pi[idVertex]);
    }
}

int main()
{
    int vertexes, edges;
    struct graph *G;

    while(scanf("%d %d", &vertexes, &edges) != EOF)
    {
        G = ReadGraph(vertexes, edges);
/*      PrintGraph(G);     */
        solver(G, 5);
        G = DeleteGraph(G);
    }
    return 0;
}
```

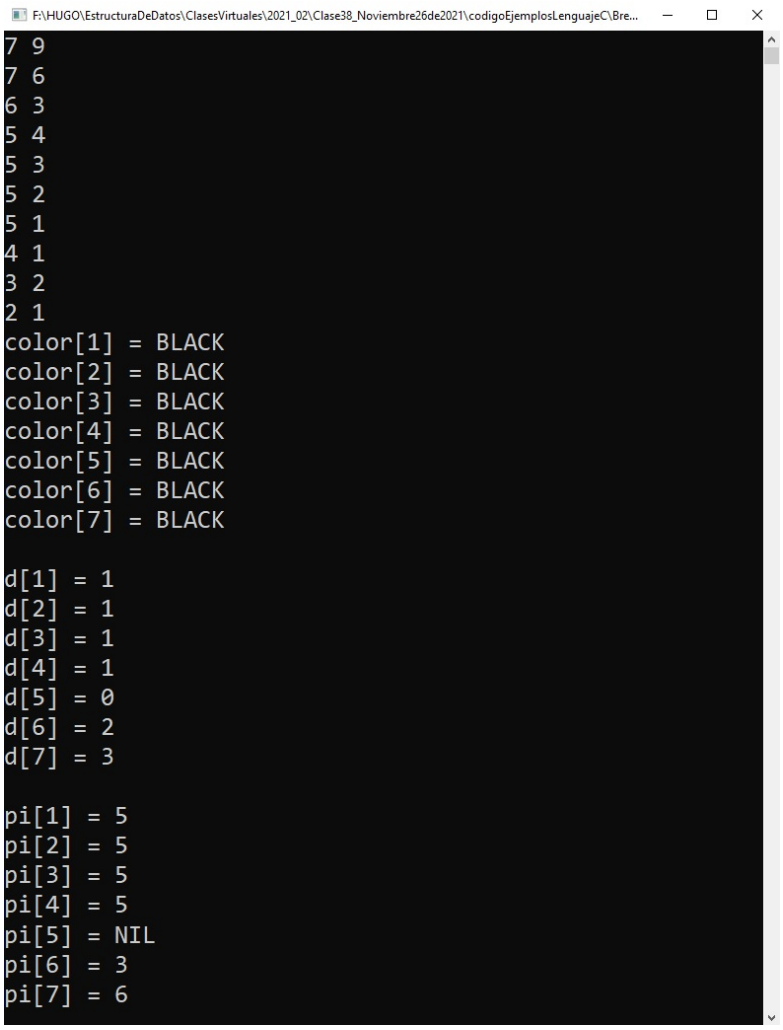


Figura 9: Salida del programa que realiza la Búsqueda Primero en Amplitud.

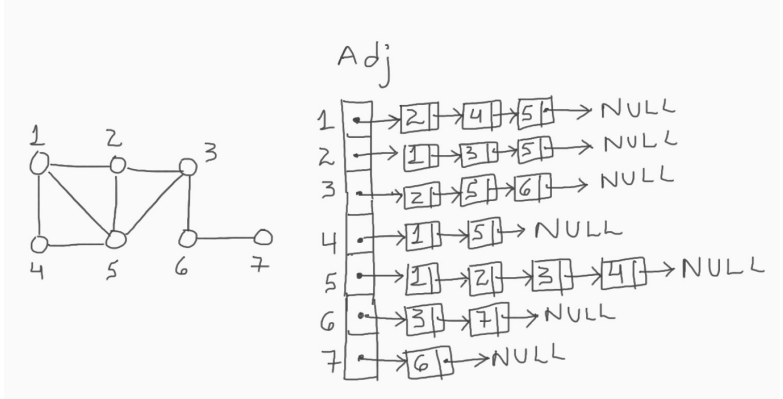


Figura 10: Grafo original que se utilizó para correr el programa de la Búsqueda Primero en Amplitud utilizando el 5 como vértice fuente.

Programming Challenge: "UVa - 11463 - Commandos"

A group of commandos were assigned a critical task. They are to destroy an enemy head quarter.

The enemy head quarter consists of several buildings and the buildings are connected by roads. The commandos must visit each building and place a bomb at the base of each building. They start their mission at the base of a particular building and from there they disseminate to reach each building. The commandos must use the available roads to travel between buildings. Any of them can visit one building after another, but they must all gather at a common place when their task in done.

In this problem, you will be given the description of different enemy headquarters. Your job is to determine the minimum time needed to complete the mission. Each commando takes exactly one unit of time to move between buildings.

You may assume that the time required to place a bomb is negligible. Each commando can carry unlimited number of bombs and there is an unlimited supply of commando troops for the mission.

Input specification:

The first line of input contains a number $T < 50$, where T denotes the number of test cases.

Each case describes one head quarter scenario. The first line of each case starts with a positive integer $N \leq 100$, where N denotes the number of buildings in the head quarter. The next line contains a positive integer R, where R is the number of roads connecting two buildings. Each of the next R lines contain two distinct numbers, $0 \leq u, v < N$, this means there is a road connecting building u to building v. The buildings are numbered from 0 to $N - 1$. The last line of each case contains two integers $0 \leq s, d < N$. Where s denotes the building from where the mission starts and d denotes the building where they must meet.

You may assume that two buildings will be directly connected by at most one road. The input will be such that, it will be possible to go from any building to another by using one or more roads.

Output specification:

For each case of input, there will be one line of output. It will contain the case number followed by the minimum time required to complete the mission. Look at the sample output for exact formatting.

Example input:

```
2
4
3
0 1
2 1
1 3
0 3
2
1
0 1
1 0
```

Example output:

```
Case 1: 4
Case 2: 1
```

Búsqueda Primero en Profundidad

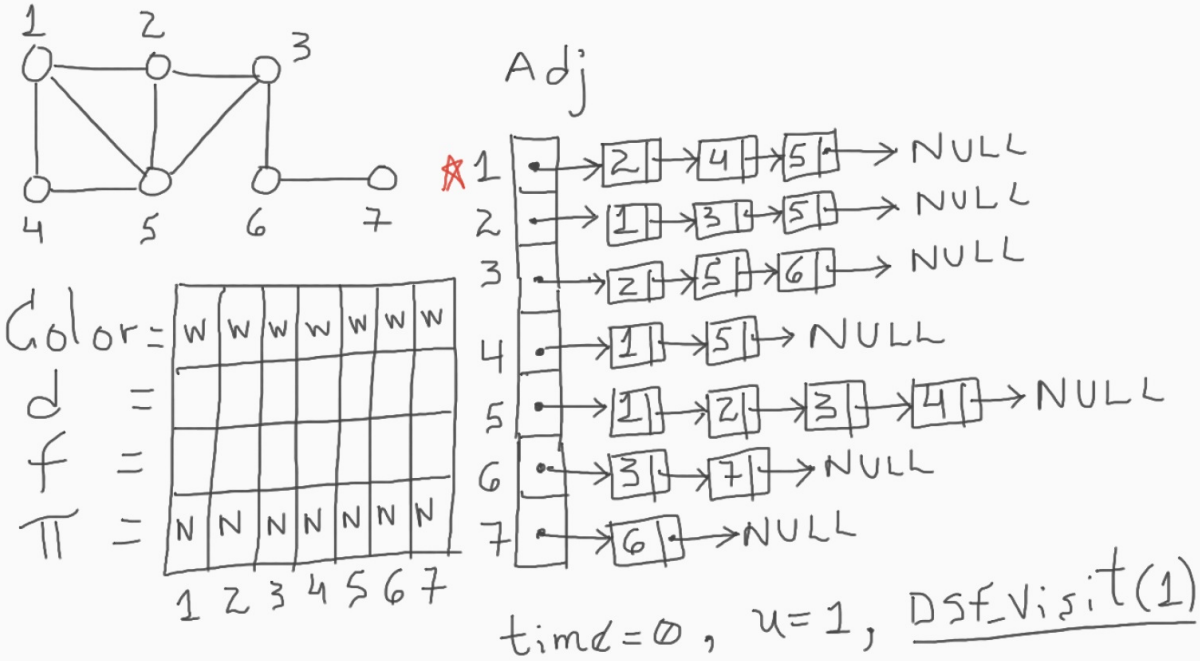
Algoritmo Depth-First Search

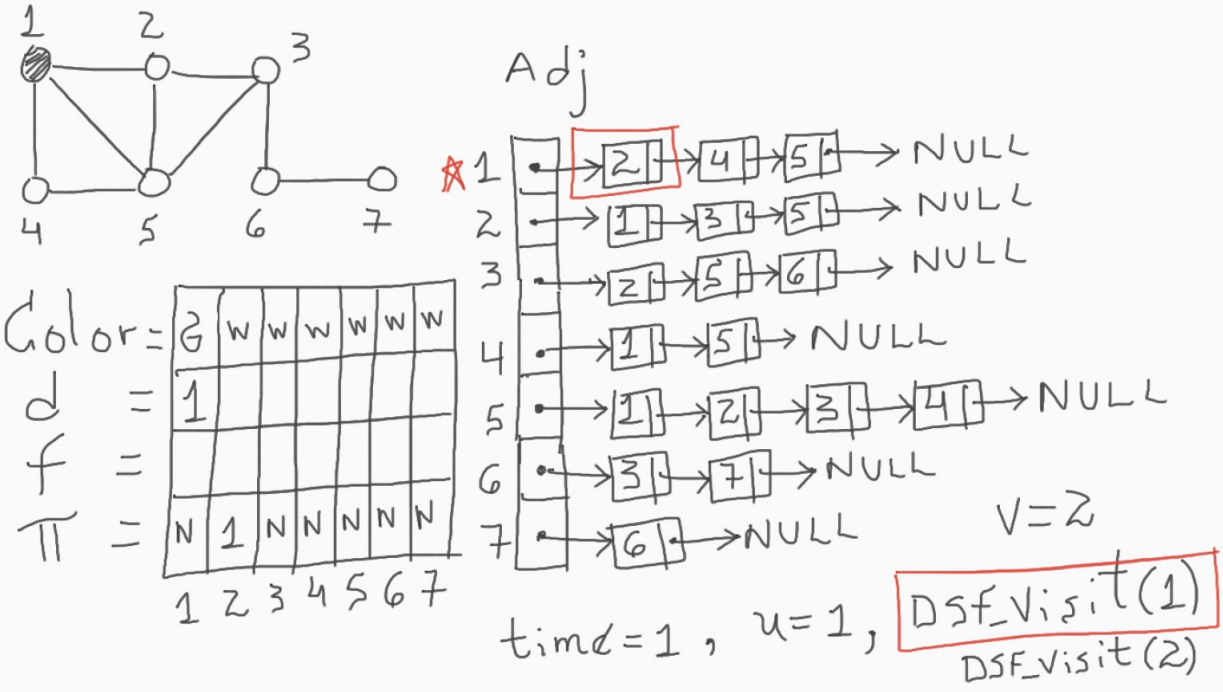
function `DFS`(G)

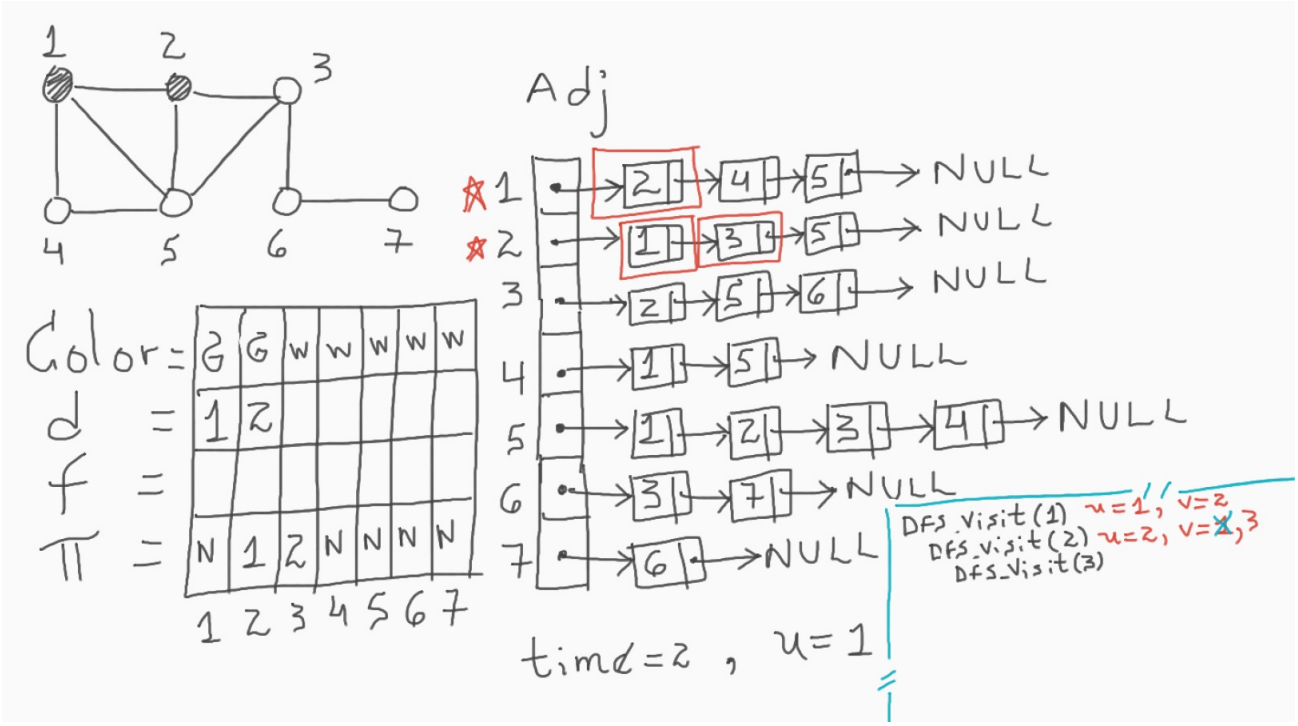
```
1   for each vertex u ∈ V[G] do
2       color[u] ← WHITE
3       π[u] ← NIL
4   time ← 0
5   for each vertex u ∈ V[G] do
6       if color[u] == WHITE then
7           DFS-Visit( G, u )
```

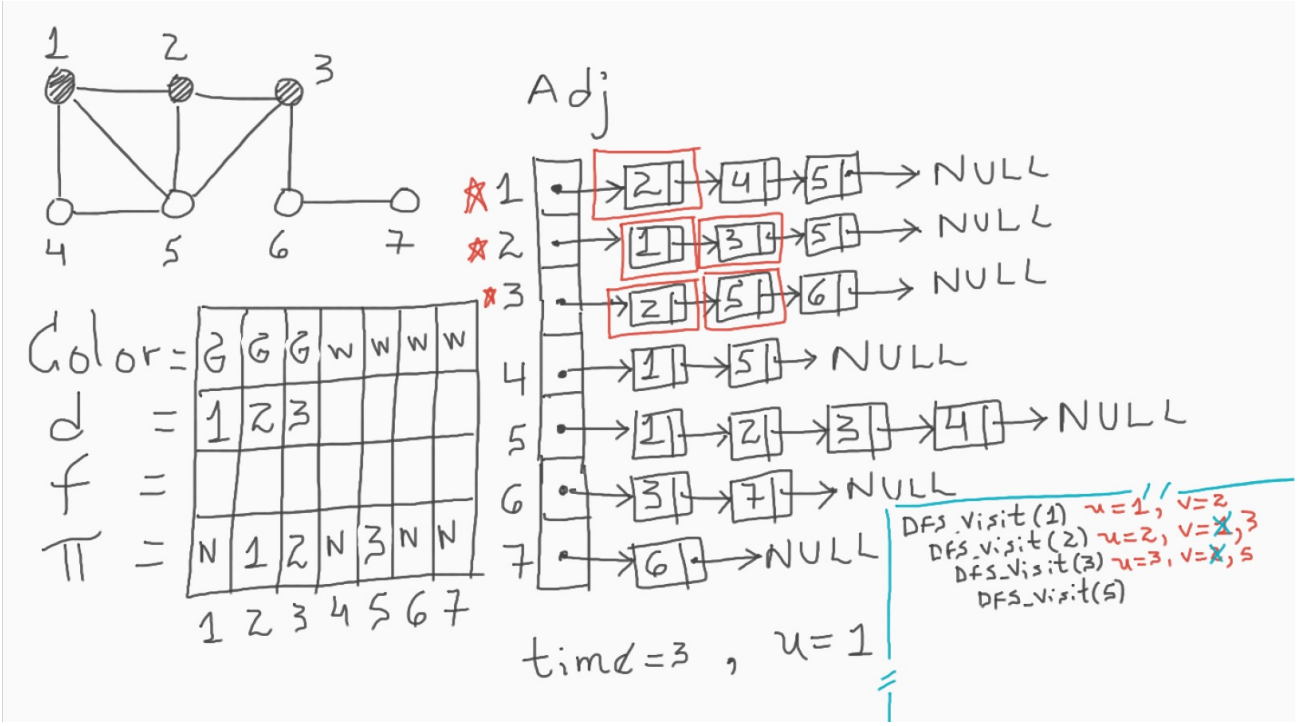
function `DFS-Visit`(G, u)

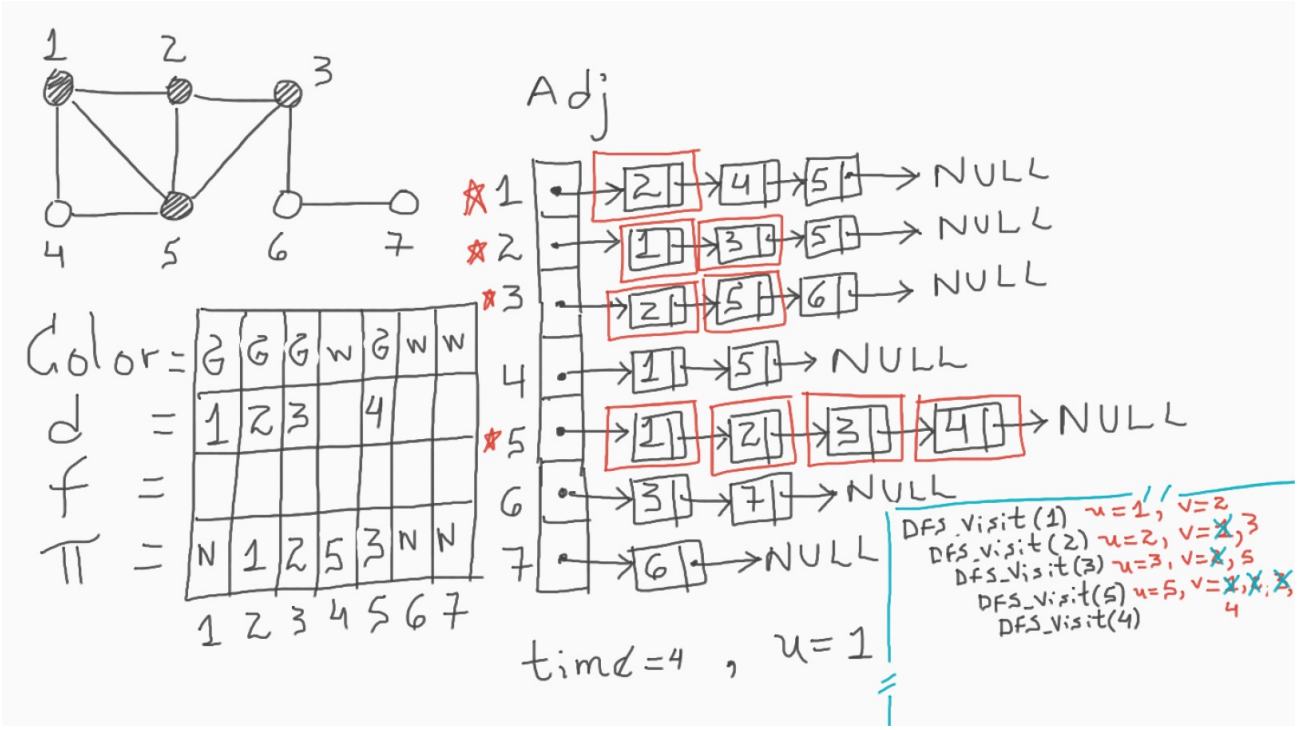
```
1   color[u] ← GRAY              ▷ white vertex u has just been discovered
2   time ← time + 1
3   d[u] ← time
4   for each vertex v ∈ Adj[u] do    ▷ explore edge (u, v)
5       if color[v] == WHITE then
6           π[v] ← u
7           DFS-Visit( G, v )
8   color[u] ← BLACK             ▷ blacken u; it is finished
9   time ← time + 1
10  f[u] ← time
```

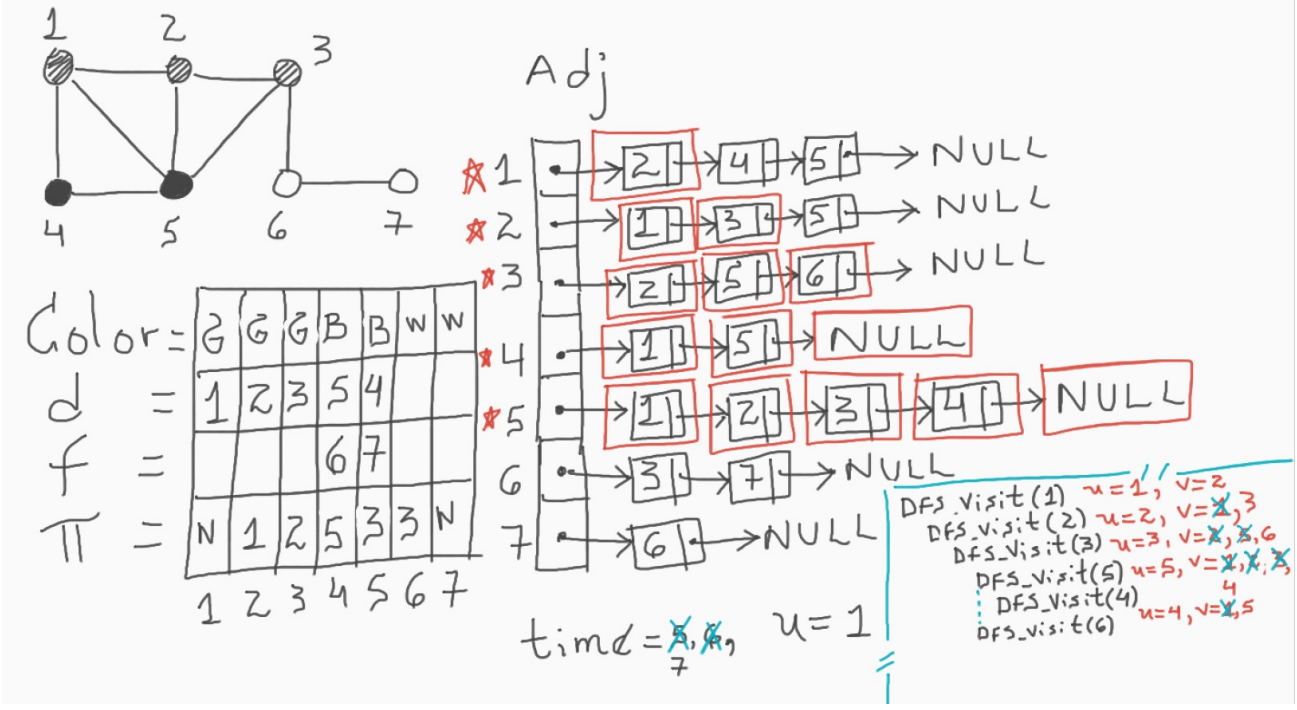

1
2
3
4
5
6
7
Adj
1 → 2 → 4 → 5 → NULL
2 → 1 → 3 → 5 → NULL
3 → 2 → 5 → 6 → NULL
4 → 1 → 5 → NULL
5 → 1 → 2 → 3 → 4 → NULL
6 → 3 → 7 → NULL
7 → 6 → NULL
Color = W W W W W W W
d =
f =
π = N N N N N N N
1 2 3 4 5 6 7
time = 0 , u = 1 , DSF_Visit(1)

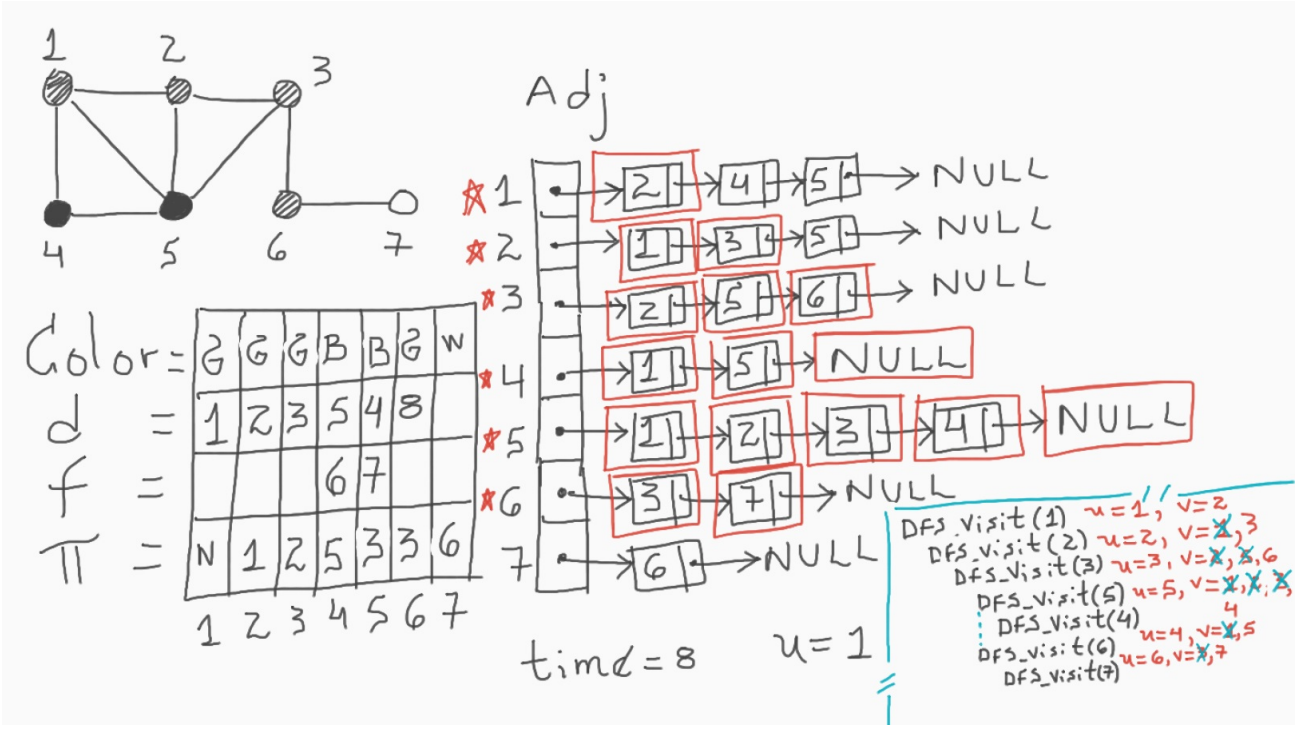

1
2
3
4
5
6
7
Adj
1 → 2 → 4 → 5 → NULL
2 → 1 → 3 → 5 → NULL
3 → 2 → 5 → 6 → NULL
4 → 1 → 5 → NULL
5 → 1 → 2 → 3 → 4 → NULL
6 → 3 → 7 → NULL
7 → 6 → NULL
Color = G W W W W W W
d = 1
f =
π = N 1 N N N N N
1 2 3 4 5 6 7
V = 2
time = 1 , u = 1 , DSF_Visit(1)
DSF_Visit(2)

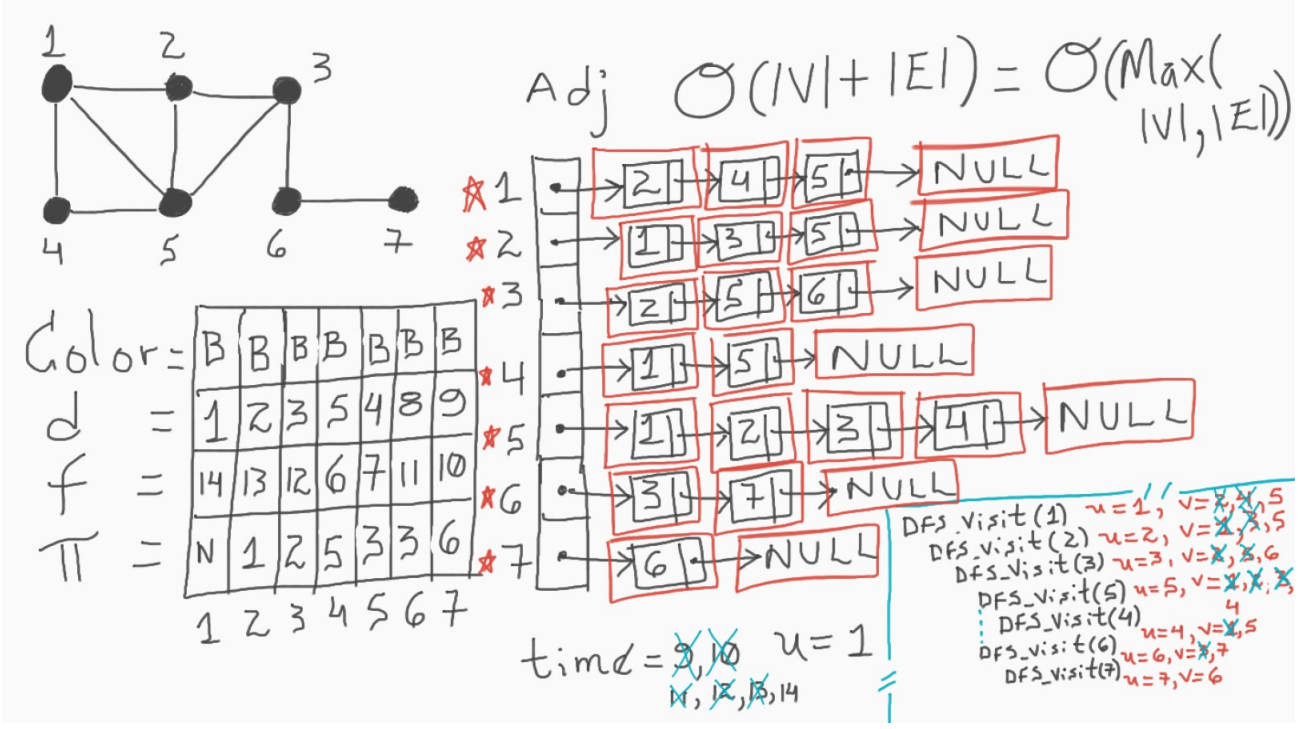

1 2 3 4 5 6 7
Adj
1 → 2 → 4 → 5 → NULL
2 → 1 → 3 → 5 → NULL
3 → 2 → 5 → 6 → NULL
4 → 1 → 5 → NULL
5 → 1 → 2 → 3 → 4 → NULL
6 → 3 → 7 → NULL
7 → 6 → NULL
Color = G G W W W W W
d = 1 2
f =
π = N 1 2 N N N N
1 2 3 4 5 6 7
time = 2 , u = 1
DFS_Visit(1) u=1, v=2
DFS_Visit(2) u=2, v=X, 3
DFS_Visit(3)

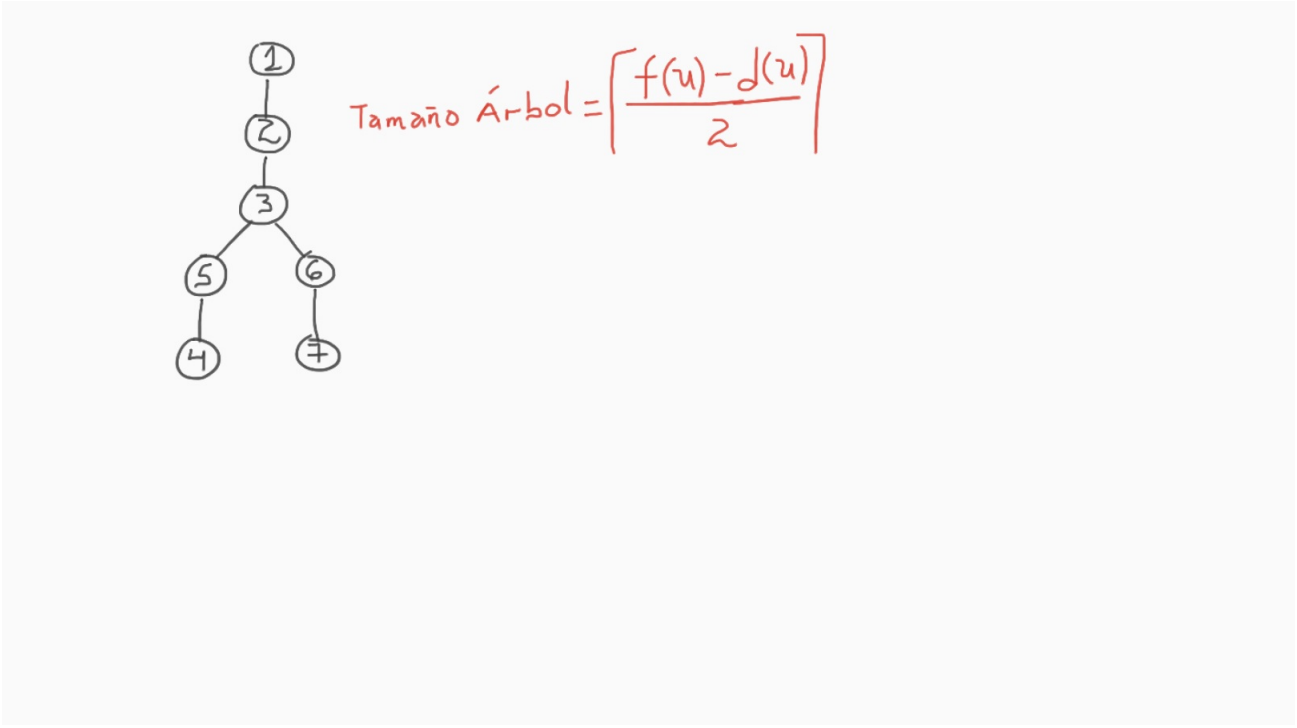

1 2 3 4 5 6 7
Adj
1 → 2 → 4 → 5 → NULL
2 → 1 → 3 → 5 → NULL
3 → 2 → 5 → 6 → NULL
4 → 1 → 5 → NULL
5 → 1 → 2 → 3 → 4 → NULL
6 → 3 → 7 → NULL
7 → 6 → NULL
Color = G G G W W W W
d = 1 2 3
f =
π = N 1 2 N 3 N N
1 2 3 4 5 6 7
time = 3 , u = 1
DFS_Visit(1) u=1, v=2
DFS_Visit(2) u=2, v=X, 3
DFS_Visit(3) u=3, v=X, 5
DFS_Visit(5)


1 2 3 4 5 6 7
Adj
1 → 2 → 4 → 5 → NULL
2 → 1 → 3 → 5 → NULL
3 → 2 → 5 → 6 → NULL
4 → 1 → 5 → NULL
5 → 1 → 2 → 3 → 4 → NULL
6 → 3 → 7 → NULL
7 → 6 → NULL
Color = G G G W G W W
d = 1 2 3 4
f =
π = N 1 2 5 3 N N
1 2 3 4 5 6 7
time = 4 , u = 1
DFS_Visit(1) u=1, v=2
DFS_Visit(2) u=2, v=3
DFS_Visit(3) u=3, v=5
DFS_Visit(5) u=5, v=4
DFS_Visit(4)


1 2 3 4 5 6 7
Adj
1 → 2 → 4 → 5 → NULL
2 → 1 → 3 → 5 → NULL
3 → 2 → 5 → 6 → NULL
4 → 1 → 5 → NULL
5 → 1 → 2 → 3 → 4 → NULL
6 → 3 → 7 → NULL
7 → 6 → NULL
Color = G G G B B W W
d = 1 2 3 5 4
f = 6 7
π = N 1 2 5 3 3 N
1 2 3 4 5 6 7
time = 7, u = 1
DFS_Visit(1) u=1, v=2
DFS_Visit(2) u=2, v=3
DFS_Visit(3) u=3, v=5
DFS_Visit(5) u=5, v=4
DFS_Visit(4) u=4, v=6
DFS_Visit(6)


Adj
Color = G G G B B G W
d = 1 2 3 5 4 8
f = 6 7
π = N 1 2 5 3 3 6
1 2 3 4 5 6 7
time = 8
u = 1
NULL
DFS_Visit(1) u=1, v=2
DFS_Visit(2) u=2, v=3
DFS_Visit(3) u=3, v=6
DFS_Visit(5) u=5, v=4
DFS_Visit(4) u=4, v=5
DFS_Visit(6) u=6, v=7
DFS_Visit(7)


Adj O(|V|+|E|) = O(Max(|V|,|E|))
Color = B B B B B B B
d = 1 2 3 5 4 8 9
f = 14 13 12 6 7 11 10
π = N 1 2 5 3 3 6
1 2 3 4 5 6 7
time = 8,10 / 11, 12, 13, 14
u = 1
NULL
DFS_Visit(1) u=1, v=2
DFS_Visit(2) u=2, v=3
DFS_Visit(3) u=3, v=6
DFS_Visit(5) u=5, v=4
DFS_Visit(4) u=4, v=5
DFS_Visit(6) u=6, v=7
DFS_Visit(7) u=7, v=6



Implementación de la Búsqueda Primero en Profundidad sobre Grafos representados por medio de listas de adyacencia

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define WHITE 2
#define GRAY 1
#define BLACK 0
#define NIL -1
#define MAXV 1005

struct edge
{
    int vertex;
/*  int weight;  */
    struct edge *next;
};

struct graph
{
    int n_vertex;
    int m_edges;
    struct edge *Adj[MAXV];
};

int time = 0;

struct graph *ReadGraph(int vertexes, int edges)
{
    int idVertex, idEdge, u, v;
    struct graph *G;
    struct edge *tempEdge;

    G = (struct graph *) malloc(sizeof(struct graph));
    G->n_vertex = vertexes;
```

```
    G->m_edges = edges;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        G->Adj[idVertex] = NULL;

    for(idEdge = 1; idEdge <= G->m_edges; idEdge++)
    {
        scanf("%d %d", &u, &v);
        tempEdge = (struct edge *) malloc(sizeof(struct edge));
        tempEdge->vertex = v;
        tempEdge->next = G->Adj[u];
        G->Adj[u] = tempEdge;

        if(u != v)
        {
            tempEdge = (struct edge *) malloc(sizeof(struct edge));
            tempEdge->vertex = u;
            tempEdge->next = G->Adj[v];
            G->Adj[v] = tempEdge;
        }
    }
    return G;
}

void PrintGraph(struct graph *G)
{
    int idVertex;
    struct edge *tempEdge;

    if(G != NULL)
    {
        printf("Representation for graph's adjacent lists: \n");
        for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        {
            printf("[%d]: ", idVertex);
            tempEdge = G->Adj[idVertex];
            while(tempEdge != NULL)
            {
                printf(" -> %d", tempEdge->vertex);
                tempEdge = tempEdge->next;
            }
            printf(" -> NULL\n");
        }
    }
    else
        printf("Empty Graph.\n");
}

struct graph *DeleteGraph(struct graph *G)
{
    int idVertex;
    struct edge *ActualEdge, *NextEdge;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        ActualEdge = G->Adj[idVertex];
        while(ActualEdge != NULL)
        {
            NextEdge = ActualEdge->next;
            free(ActualEdge);
            ActualEdge = NextEdge;
        }
    }
    free(G);
    G = NULL;
    return G;
}
```

```
void DFS_Visit(struct graph *G, int u, int color[], int d[], int f[], int pi[])
{
    int v;
    struct edge *tempEdge;
    color[u] = GRAY;
    time++;
    d[u] = time;

    tempEdge = G->Adj[u];
    while(tempEdge != NULL)
    {
        v = tempEdge->vertex;
        if(color[v] == WHITE)
        {
            pi[v] = u;
            DFS_Visit(G, v, color, d, f, pi);
        }
        tempEdge = tempEdge->next;
    }
    color[u] = BLACK;
    time++;
    f[u] = time;
}

void DFS(struct graph *G, int color[], int d[], int f[], int pi[])
{
    int u;

    for(u = 1; u <= G->n_vertex; u++)
    {
        color[u] = WHITE;
        pi[u] = NIL;
    }

    time = 0;

    for(u = 1; u <= G->n_vertex; u++)
    {
        if(color[u] == WHITE)
            DFS_Visit(G, u, color, d, f, pi);
    }
}

void solver(struct graph *G)
{
    int color[MAXV], d[MAXV], f[MAXV], pi[MAXV], idVertex;

    DFS(G, color, d, f, pi);

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        if(color[idVertex] == WHITE)
            printf("color[%d] = WHITE\n", idVertex);
        if(color[idVertex] == GRAY)
            printf("color[%d] = GRAY\n", idVertex);
        if(color[idVertex] == BLACK)
            printf("color[%d] = BLACK\n", idVertex);
    }
    printf("\n");

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        printf("d[%d] = %d\n", idVertex, d[idVertex]);
    printf("\n");

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        printf("f[%d] = %d\n", idVertex, f[idVertex]);
    printf("\n");
```

```
    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        if(pi[idVertex] == NIL)
            printf("pi[%d] = NIL\n", idVertex);
        else
            printf("pi[%d] = %d\n", idVertex, pi[idVertex]);
    }
}

int main()
{
    int vertexes, edges;
    struct graph *G;

    while(scanf("%d %d", &vertexes, &edges) != EOF)
    {
        G = ReadGraph(vertexes, edges);
/*      PrintGraph(G);      */
        solver(G);
        G = DeleteGraph(G);
    }
    return 0;
}
```

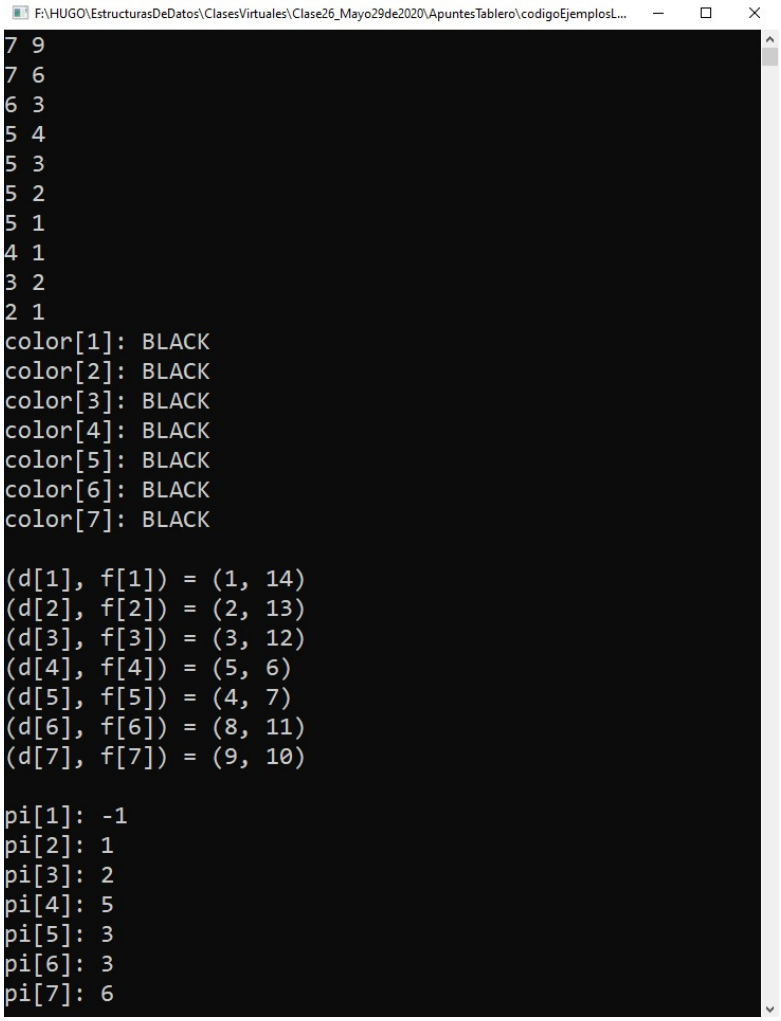


Figura 11: Salida del programa que realiza la Búsqueda Primero en Profundidad.

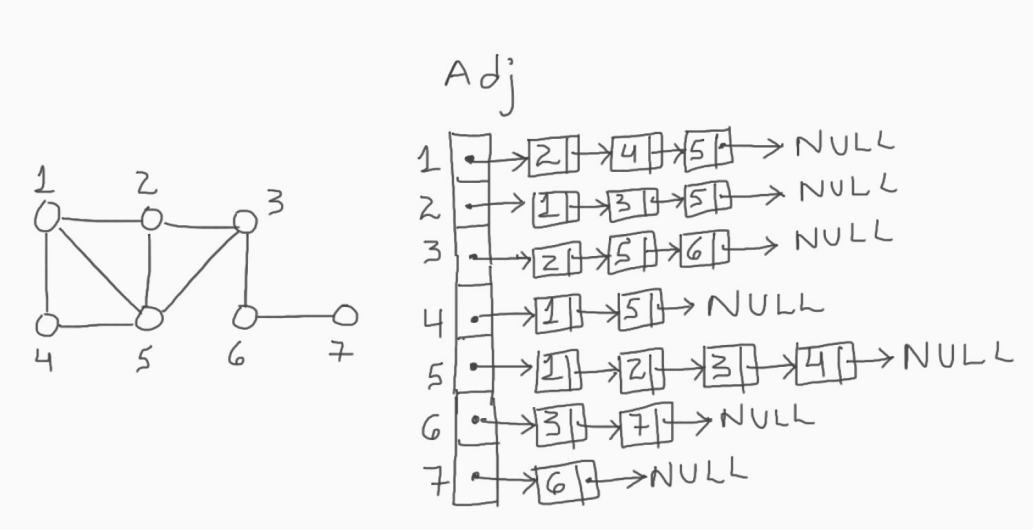


Figura 12: Grafo original que se utilizó para correr el programa de la Búsqueda Primero en Profundidad.

Programming Challenge: "Toby and the Graph"[1]

Toby has an undirected graph where not necessarily all vertices are connected to each other. As a result, we could have some disjoint sets of interconnected vertices. Toby is a curious and smart dog, so he is wondering about the following question: how many new sets can we get after joining **any two** initial sets?

Input specification:

The first line contains a single integer T denoting the number of test cases. Each case in the first line contains two integers n ($1 \leq n \leq 10^4$) and m ($0 \leq m \leq n-1$), the number of vertices and the number of edges respectively. The next m lines contains two integers separated by a single space a, b ($1 \leq a \leq b \leq n$) meaning that vertex a is connected to vertex b. Each vertex is enumerated from 1 to n.

Output specification:

Print the answer in a separate line.

Example input:

```
2
7 4
2 3
4 5
5 6
4 6
1 0
```

Example output:

```
6
0
```

[1] https://www.hackerrank.com/contests/utp-open-2018/challenges/toby-and-the-graph

Solución del reto "Toby and the Graph" utilizando Búsqueda Primero en Profundidad sobre el grafo

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define WHITE 2
#define GRAY 1
#define BLACK 0
#define NIL -1
#define MAXV 10005

struct edge
{
    int vertex;
/*  int weight;  */
    struct edge *next;
};

struct graph
{
    int n_vertex;
    int m_edges;
    struct edge *Adj[MAXV];
};

int time = 0;

struct graph *ReadGraph(int vertexes, int edges)
{
    int idVertex, idEdge, u, v;
    struct graph *G;
    struct edge *tempEdge;

    G = (struct graph *) malloc(sizeof(struct graph));
    G->n_vertex = vertexes;
    G->m_edges = edges;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        G->Adj[idVertex] = NULL;

    for(idEdge = 1; idEdge <= G->m_edges; idEdge++)
    {
        scanf("%d %d", &u, &v);
        tempEdge = (struct edge *) malloc(sizeof(struct edge));
        tempEdge->vertex = v;
        tempEdge->next = G->Adj[u];
        G->Adj[u] = tempEdge;

        if(u != v)
        {
            tempEdge = (struct edge *) malloc(sizeof(struct edge));
            tempEdge->vertex = u;
            tempEdge->next = G->Adj[v];
            G->Adj[v] = tempEdge;
        }
    }

    return G;
}

void PrintGraph(struct graph *G)
{
    int idVertex;
    struct edge *tempEdge;

    if(G != NULL)
    {
```

```
        printf("Representation for graph's adjacent lists: \n");
        for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        {
            printf("[%d]: ", idVertex);
            tempEdge = G->Adj[idVertex];
            while(tempEdge != NULL)
            {
                printf(" -> %d", tempEdge->vertex);
                tempEdge = tempEdge->next;
            }
            printf(" -> NULL\n");
        }
    }
    else
        printf("Empty Graph.\n");
}

struct graph *DeleteGraph(struct graph *G)
{
    int idVertex;
    struct edge *ActualEdge, *NextEdge;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        ActualEdge = G->Adj[idVertex];
        while(ActualEdge != NULL)
        {
            NextEdge = ActualEdge->next;
            free(ActualEdge);
            ActualEdge = NextEdge;
        }
    }

    free(G);
    G = NULL;

    return G;
}

void DFS_Visit(struct graph *G, int u, int color[], int d[], int f[], int pi[])
{
    int v;
    struct edge *tempEdge;
    color[u] = GRAY;
    time++;
    d[u] = time;

    tempEdge = G->Adj[u];
    while(tempEdge != NULL)
    {
        v = tempEdge->vertex;
        if(color[v] == WHITE)
        {
            pi[v] = u;
            DFS_Visit(G, v, color, d, f, pi);
        }
        tempEdge = tempEdge->next;
    }
    color[u] = BLACK;
    time++;
    f[u] = time;
}

void DFS(struct graph *G, int color[], int d[], int f[], int pi[])
{
    int u;

    for(u = 1; u <= G->n_vertex; u++)
```

```
    {
        color[u] = WHITE;
        pi[u] = NIL;
    }

    time = 0;

    for(u = 1; u <= G->n_vertex; u++)
    {
        if(color[u] == WHITE)
            DFS_Visit(G, u, color, d, f, pi);
    }
}

int solver(struct graph *G)
{
    int color[MAXV], d[MAXV], f[MAXV], pi[MAXV], idVertex;
    int totalTrees = 0, result;

    DFS(G, color, d, f, pi);

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        if(pi[idVertex] == NIL)
            totalTrees++;

    result = ((totalTrees - 1) * totalTrees) / 2;

    return result;
}

int main()
{
    int totalCases, idCase, n, m;
    struct graph *G;

    scanf("%d", &totalCases);

    for(idCase = 1; idCase <= totalCases; idCase++)
    {
        scanf("%d %d", &n, &m);
        G = ReadGraph(n, m);
        printf("%d\n", solver(G));
        G = DeleteGraph(G);
    }

    return 0;
}
```

Figura 13: Salida del programa para el reto “Toby and the Graph”.

Implementación de la Búsqueda Primero en Amplitud sobre Laberintos

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXH 105
#define MAXW 105
#define WHITE 2
#define GRAY 1
#define BLACK 0
#define myInfinite 2147483647
#define NIL -1

struct cell
{
    int coord_x;
    int coord_y;
};

struct cell ReadMaze(char Maze[][MAXW], int W, int H)
{
    char line[MAXW];
    int idRow, idColumn;
    struct cell source;

    for(idRow=1; idRow<=H; idRow++)
    {
        scanf("%s", line);
        for(idColumn=1; idColumn<=W; idColumn++)
        {
            Maze[idRow][idColumn] = line[idColumn - 1];
            if(line[idColumn - 1] == '@')
            {
                source.coord_x = idColumn;
                source.coord_y = idRow;
                Maze[idRow][idColumn] = '.';
            }
        }
    }

    return source;
}


void PrintMaze(char Maze[][MAXW], int W, int H)
{
    int idRow, idColumn;

    printf("\nThe original maze without source position:\n\n");
    for(idRow=1; idRow<=H; idRow++)
    {
        for(idColumn=1; idColumn <= W; idColumn++)
            printf("%c", Maze[idRow][idColumn]);
        printf("\n");
    }
    printf("\n");
}

void initializeMovements(struct cell movements[])
{
    movements[0].coord_x = 0;
    movements[0].coord_y = 0;
    movements[1].coord_x = 0;
    movements[1].coord_y = -1;
    movements[2].coord_x = 0;
    movements[2].coord_y = 1;
    movements[3].coord_x = 1;
    movements[3].coord_y = 0;
```

```
    movements[4].coord_x = -1;
    movements[4].coord_y = 0;
}

void BFS_Maze(char Maze[][MAXW], int W, int H, struct cell s,
              int color[][MAXW], int d[][MAXW], struct cell pi[][MAXW])
{
    int idRow, idColumn, idHead = 1, idTail = 1, idMovement;
    struct cell NilFather, Q[(H * W) + 5], u, v, movements[5];

    initializeMovements(movements);
    NilFather.coord_x = NIL;
    NilFather.coord_y = NIL;

    for(idRow = 1; idRow <= H; idRow++)
    {
        for(idColumn = 1; idColumn <= W; idColumn++)
        {
            color[idRow][idColumn] = WHITE;
            d[idRow][idColumn] = myInfinite;
            pi[idRow][idColumn] = NilFather;
        }
    }

    color[s.coord_y][s.coord_x] = GRAY;
    d[s.coord_y][s.coord_x] = 0;
    pi[s.coord_y][s.coord_x] = NilFather;
    Q[idTail] = s;
    idTail++;

    while(idHead < idTail)
    {
        u = Q[idHead];
        idHead++;
        for(idMovement = 1; idMovement <= 4; idMovement++)
        {
            v.coord_x = u.coord_x + movements[idMovement].coord_x;
            v.coord_y = u.coord_y + movements[idMovement].coord_y;

            if((v.coord_x >= 1 && v.coord_x <= W) &&
               (v.coord_y >= 1 && v.coord_y <= H) &&
               (Maze[v.coord_y][v.coord_x] == '.') &&
               (color[v.coord_y][v.coord_x] == WHITE))
            {
                color[v.coord_y][v.coord_x] = GRAY;
                d[v.coord_y][v.coord_x] = d[u.coord_y][u.coord_x] + 1;
                pi[v.coord_y][v.coord_x] = u;
                Q[idTail] = v;
                idTail++;
            }
        }
        color[u.coord_y][u.coord_x] = BLACK;
    }
}

void solver(char Maze[][MAXW], int W, int H, struct cell source)
{
    int color[MAXH][MAXW], d[MAXH][MAXW], idRow, idColumn;
    struct cell pi[MAXH][MAXW];

    BFS_Maze(Maze, W, H, source, color, d, pi);

    printf("Matrix of colors:\n\n");
    for(idRow=1; idRow<=H; idRow++)
    {
        for(idColumn=1; idColumn <= W; idColumn++)
        {
            if(color[idRow][idColumn] == WHITE)
```

```
                printf(" W");
            if(color[idRow][idColumn] == GRAY)
                printf(" G");
            if(color[idRow][idColumn] == BLACK)
                printf(" B");
        }
        printf("\n");
    }
    printf("\n");

    printf("Matrix of distances:\n\n");
    for(idRow=1; idRow<=H; idRow++)
    {
        for(idColumn=1; idColumn<=W; idColumn++)
        {
            if(d[idRow][idColumn] == myInfinite)
                printf(" IN");
            else
            {
                if(d[idRow][idColumn] < 10)
                    printf("  %d", d[idRow][idColumn]);
                else
                    printf(" %d", d[idRow][idColumn]);
            }
        }
        printf("\n");
    }
    printf("\n");

    printf("Matrix of fathers:\n\n");
    for(idRow=1; idRow<=H; idRow++)
    {
        for(idColumn=1; idColumn<=W; idColumn++)
        {
            if(pi[idRow][idColumn].coord_x == NIL)
                printf(" [ -1, -1]");
            else
            {
                if(pi[idRow][idColumn].coord_x < 10)
                    printf(" [  %d,", pi[idRow][idColumn].coord_x);
                else
                    printf(" [ %d,", pi[idRow][idColumn].coord_x);
                if(pi[idRow][idColumn].coord_y < 10)
                    printf("  %d]", pi[idRow][idColumn].coord_y);
                else
                    printf(" %d]", pi[idRow][idColumn].coord_y);
            }
        }
        printf("\n");
    }
    printf("\n");
}

int main()
{
    char Maze[MAXH][MAXW];
    int T, W, H, idCase;
    struct cell source;

    scanf("%d", &T);
    for(idCase=1; idCase<=T; idCase++)
    {
        scanf("%d %d", &W, &H);
        source = ReadMaze(Maze, W, H);
        PrintMaze(Maze, W, H);
        solver(Maze, W, H, source);
    }
```

```
    return 0;
}
```

Figura 14: Primera parte de la salida del programa que realiza la Búsqueda Primero en Amplitud Profundidad.

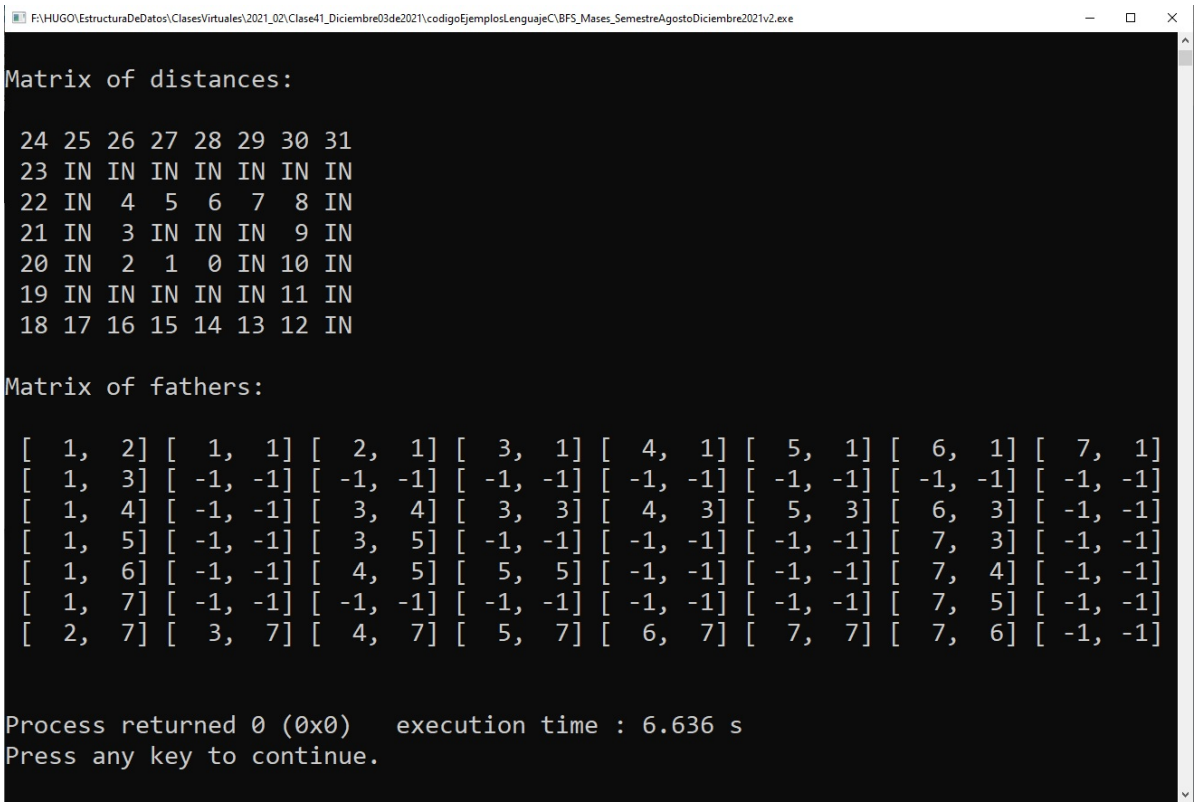


Figura 15: Segunda parte de la salida del programa que realiza la Búsqueda Primero en Amplitud Profundidad.

Reto de programación: "El Príncipe Juanman"[2]

Una vez hubo un gran rey llamado Juribeto. El tuvo un hijo llamado Juanman. Por alguna razón inexplicable (generada por algunos hechos false/true, black/white o cero/one) el rey tenía en riesgo la gobernabilidad en su reino. La corte del rey Juribeto siempre acuso a su hijo Juanman de conspirar en contra de él y ser culpable de buscar la caída del rey para para queda como el nuevo rey.

Gracias al amor a su hijo el rey Juribeto decide que su hijo no debe ser decapitado en plaza publica para generar escarmiento, en lugar de ello, el príncipe debe ser desterrado del reino, se debe ir a un nuevo lugar. El príncipe Juanman tuvo que seguir la voluntad de su padre si quería conservar su vida. En el camino el príncipe encontró que el lugar tiene una combinación de tierra y agua. Debido a que el no sabe nadar, el únicamente fue capaz de moverse sobre la tierra. El no conoce cuántos lugares pueden alcanzar en su nuevo destino. Así, que el necesita de su ayuda.

Por facilidad, usted puede considerar que el lugar es una cuadrícula (una matriz) de celdas. Una celda puede contener tierra o puede contener agua. En cada momento el príncipe puede moverse a una nueva celda desde su posición actual si ellas comparten un lado.

Ahora escriba un programa para encontrar el número de celdas (unidades de tierra) que el príncipe puede alcanzar incluyendo la celda en la cual se encuentra como punto de partida.

Especificación de la entrada:

La entrada comienza con un número entero positivo T ($1 \leq T \leq 10$), denotando el número de casos de prueba.

[2] https://www.hackerrank.com/contests/utp-open-2018/challenges/el-principe-juanman

Cada caso comienza con una linea que contiene dos enteros positivos W H (Wide, High, $3 \leq W, H \leq 1000$); El caso de prueba continua con H líneas, cada una conteniendo W caracteres. Cada caracter representa el estatus de la celda como sigue:

- '`.`' - tierra,
- '`#`' - agua,
- '`@`' - posición inicial del príncipe (aparece exactamente una vez por caso de prueba).

Especificación de la salida:

Para cada caso de prueba, imprima el número del caso y el número de celdas que el príncipe puede alcanzar desde su posición inicial (incluyendo ésta). Para mayor claridad con el formato de salida mirar los ejemplos a continuación.

Ejemplo de entrada:

```
4
6 9
....#.
.....#
......
......
......
......
......
#@...#
.#..#.
11 9
.#.........
.#.#######.
.#.#.....#.
.#.#.###.#.
.#.#..@#.#.
.#.#####.#.
.#.......#.
.#########.
...........
11 6
..#..#..#..
..#..#..#..
..#..#..###
..#..#..#@.
..#..#..#..
..#..#..#..
7 7
..#.#..
..#.#..
###.###
...@...
###.###
..#.#..
..#.#..
```

Ejemplo de salida:

```
Case 1: 45
Case 2: 59
Case 3: 6
Case 4: 13
```

Solución del reto: "El Príncipe Juanman" utilizando Búsqueda Primero en Amplitud sobre laberintos

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXH 1005
#define MAXW 1005
#define WHITE 2
#define GRAY 1
#define BLACK 0
#define myInfinite 2147483647
#define NIL -1

struct cell
{
    int coord_x;
    int coord_y;
};

struct cell ReadMaze(char Maze[][MAXW], int W, int H)
{
    char line[MAXW];
    int idRow, idColumn;
    struct cell source;

    for(idRow=1; idRow<=H; idRow++)
    {
        scanf("%s", line);
        for(idColumn=1; idColumn<=W; idColumn++)
        {
            Maze[idRow][idColumn] = line[idColumn - 1];
            if(line[idColumn - 1] == '@')
            {
                source.coord_x = idColumn;
                source.coord_y = idRow;
                Maze[idRow][idColumn] = '.';
            }
        }
    }

    return source;
}

void PrintMaze(char Maze[][MAXW], int W, int H)
{
    int idRow, idColumn;

    printf("\nThe original maze without source position:\n\n");
    for(idRow=1; idRow<=H; idRow++)
    {
        for(idColumn=1; idColumn <= W; idColumn++)
            printf("%c", Maze[idRow][idColumn]);
        printf("\n");
    }
    printf("\n");
}
```

```
void initializeMovements(struct cell movements[])
{
    movements[0].coord_x = 0;
    movements[0].coord_y = 0;
    movements[1].coord_x = 0;
    movements[1].coord_y = -1;
    movements[2].coord_x = 0;
    movements[2].coord_y = 1;
    movements[3].coord_x = 1;
    movements[3].coord_y = 0;
    movements[4].coord_x = -1;
    movements[4].coord_y = 0;
}

void BFS_Maze(char Maze[][MAXW], int W, int H, struct cell s,
              int color[][MAXW], int d[][MAXW], struct cell pi[][MAXW])
{
    int idRow, idColumn, idHead = 1, idTail = 1, idMovement;
    struct cell NilFather, Q[(H * W) + 5], u, v, movements[5];

    initializeMovements(movements);
    NilFather.coord_x = NIL;
    NilFather.coord_y = NIL;

    for(idRow = 1; idRow <= H; idRow++)
    {
        for(idColumn = 1; idColumn <= W; idColumn++)
        {
            color[idRow][idColumn] = WHITE;
            d[idRow][idColumn] = myInfinite;
            pi[idRow][idColumn] = NilFather;
        }
    }

    color[s.coord_y][s.coord_x] = GRAY;
    d[s.coord_y][s.coord_x] = 0;
    pi[s.coord_y][s.coord_x] = NilFather;
    Q[idTail] = s;
    idTail++;

    while(idHead < idTail)
    {
        u = Q[idHead];
        idHead++;
        for(idMovement = 1; idMovement <= 4; idMovement++)
        {
            v.coord_x = u.coord_x + movements[idMovement].coord_x;
            v.coord_y = u.coord_y + movements[idMovement].coord_y;

            if((v.coord_x >= 1 && v.coord_x <= W) &&
               (v.coord_y >= 1 && v.coord_y <= H) &&
               (Maze[v.coord_y][v.coord_x] == '.') &&
               (color[v.coord_y][v.coord_x] == WHITE))
            {
                color[v.coord_y][v.coord_x] = GRAY;
                d[v.coord_y][v.coord_x] = d[u.coord_y][u.coord_x] + 1;
                pi[v.coord_y][v.coord_x] = u;
                Q[idTail] = v;
                idTail++;
            }
        }
        color[u.coord_y][u.coord_x] = BLACK;
    }
}

int solver(char Maze[][MAXW], int W, int H, struct cell source)
{
    int color[MAXH][MAXW], d[MAXH][MAXW];
```

```
    int idRow, idColumn, result = 0;
    struct cell pi[MAXH][MAXW];

    BFS_Maze(Maze, W, H, source, color, d, pi);

    for(idRow=1; idRow<=H; idRow++)
        for(idColumn=1; idColumn <= W; idColumn++)
            if(color[idRow][idColumn] == BLACK)
                result++;

    return result;
}

int main()
{
    char Maze[MAXH][MAXW];
    int T, W, H, idCase;
    struct cell source;

    scanf("%d", &T);
    for(idCase=1; idCase<=T; idCase++)
    {
        scanf("%d %d", &W, &H);
        source = ReadMaze(Maze, W, H);
        printf("Case %d: %d\n", idCase, solver(Maze, W, H, source));
    }

    return 0;
}
```

Algoritmo de Prim

El algoritmo de Prim opera muy parecido al algoritmo de Dijkstra para encontrar los caminos más cortos en un grafo. El árbol de abarcamiento minimal empieza en un vértice raíz arbitrario s y crece hasta que el árbol abarca todos los vértices en V. En cada paso, una arista ligera es añadida; por lo tanto, cuando el algoritmo finaliza, las aristas forman un árbol de abarcamiento mínimo. Esta estrategia es voraz debido a que el árbol es aumentado en cada paso con la arista que contribuye la mínima cantidad posible al peso del árbol.

En el pseudocódigo siguiente, el grafo conectado G y la raíz s del árbol de abarcamiento mínimo que está por crecer son entradas del algoritmo. Durante la ejecución del algoritmo, todos los vértices que no están en el árbol residen en una cola de prioridad mínima Q basada en un campo de d. Para cada vértice v, $d[v]$ es el peso mínimo de cualquier arista que conecta a v con un vértice en el árbol; por convención, $d[v] = \infty$ si no hay esta arista. El campo $\pi[v]$ nombra el padre de v en el árbol.

Pseudocódigo del algoritmo de Prim

function `PRIM`(G, w, s)

1 **for** each vertex $u \in V[G]$ **do**

2 $d[u] \leftarrow \infty$

3 $\pi[u] \leftarrow$ `NIL`

4 $d[s] \leftarrow 0$

5 $Q \leftarrow V[G]$

6 **while** $Q \neq \{\ \}$ **do**

7 $u \leftarrow$ `EXTRACT-MIN`(Q)

8 **for** each vertex $v \in Adj[u]$ **do**

9 **if** $v \in Q$ **and** $d[v] > w(u, v)$ **then**

10 $d[v] \leftarrow w(u, v)$

11 $\pi[v] \leftarrow u$

Implementación del Algoritmo de Prim

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXV 1005
#define myInfinite 2147483647
#define NIL -1
#define TRUE 1
#define FALSE 0

struct edge
{
    int vertex;
    int weight;
    struct edge *next;
};

struct graph
{
    int n_vertex;
```

```
    int m_edges;
    struct edge *Adj[MAXV];
};


struct graph *ReadGraph(int vertexes, int edges)
{
    int idVertex, idEdge, u, v, w;
    struct graph *G;
    struct edge *tempEdge;

    G = (struct graph *) malloc(sizeof(struct graph));
    G->n_vertex = vertexes;
    G->m_edges = edges;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        G->Adj[idVertex] = NULL;

    for(idEdge = 1; idEdge <= G->m_edges; idEdge++)
    {
        scanf("%d %d %d", &u, &v, &w);
        tempEdge = (struct edge *) malloc(sizeof(struct edge));
        tempEdge->vertex = v;
        tempEdge->weight = w;
        tempEdge->next = G->Adj[u];
        G->Adj[u] = tempEdge;

        if(u != v)
        {
            tempEdge = (struct edge *) malloc(sizeof(struct edge));
            tempEdge->vertex = u;
            tempEdge->weight = w;
            tempEdge->next = G->Adj[v];
            G->Adj[v] = tempEdge;
        }
    }
    return G;
}


void PrintGraph(struct graph *G)
{
    int idVertex;
    struct edge *tempEdge;

    if(G != NULL)
    {
        printf("\n Representation for graph's adjacent lists: \n\n");
        for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        {
            printf("[%d]: ", idVertex);
            tempEdge = G->Adj[idVertex];
            while(tempEdge != NULL)
            {
                printf(" -> (%d, %d)", tempEdge->vertex, tempEdge->weight);
                tempEdge = tempEdge->next;
            }
            printf(" -> NULL\n");
        }
    }
    else
        printf("\n Empty Graph.\n");
}


struct graph *DeleteGraph(struct graph *G)
{
    int idVertex;
```

```
    struct edge *ActualEdge, *NextEdge;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        ActualEdge = G->Adj[idVertex];
        while(ActualEdge != NULL)
        {
            NextEdge = ActualEdge->next;
            free(ActualEdge);
            ActualEdge = NextEdge;
        }
    }
    free(G);
    G = NULL;
    return G;
}


int ExtractMin(int d[], int inQueue[], int n)
{
    int idVertex, minimum = myInfinite;
    int idMinimum = myInfinite;

    for(idVertex = 1; idVertex <= n; idVertex++)
    {
        if(inQueue[idVertex] == TRUE)
        {
            if(d[idVertex] < minimum)
            {
                minimum = d[idVertex];
                idMinimum = idVertex;
            }
        }
    }
    if(idMinimum != myInfinite)
        inQueue[idMinimum] = FALSE;

    return idMinimum;
}


void Prim(struct graph *G, int d[], int pi[], int s)
{
    int idVertex, u, v, w, Q[MAXV], inQueue[MAXV];
    int totalElementsInQueue = G->n_vertex;
    struct edge *tempEdge;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        d[idVertex] = myInfinite;
        pi[idVertex] = NIL;
        inQueue[idVertex] = TRUE;
    }
    d[s] = 0;

    while(totalElementsInQueue > 0)
    {
        u = ExtractMin(d, inQueue, G->n_vertex);

        if(u == myInfinite)
            break;

        totalElementsInQueue--;
        tempEdge = G->Adj[u];
        while(tempEdge != NULL)
        {
            v = tempEdge->vertex;
            w = tempEdge->weight;
```

```
            if((inQueue[v] == TRUE) && (d[v] > w))
            {
                d[v] = w;
                pi[v] = u;
            }
            tempEdge = tempEdge->next;
        }
    }
}


void solver(struct graph *G)
{
    int d[MAXV], pi[MAXV], idVertex, weightMST = 0;
    Prim(G, d, pi, 9);

    printf("\n");
    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        printf("d[%d]: %d\n", idVertex, d[idVertex]);
        weightMST += d[idVertex];
    }
    printf("\n");
    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        printf("pi[%d]: %d\n", idVertex, pi[idVertex]);
    printf("\n");
    printf("Total weight of the minimum spanning tree: %d\n", weightMST);
}


int main()
{
    int vertexes, edges;
    struct graph *G;

    while(scanf("%d %d", &vertexes, &edges) != EOF)
    {
        G = ReadGraph(vertexes, edges);
        PrintGraph(G);
        solver(G);
        G = DeleteGraph(G);
        PrintGraph(G);
    }
    return 0;

/*
6 8
1 2
1 4
1 5
2 3
2 6
3 5
4 6
5 6

6 8
2 1
4 1
5 1
3 2
6 2
5 3
6 4
6 5

6 8
6 5
```

```
6  4
6  2
5  3
5  1
4  1
3  2
2  1
*/
}
```

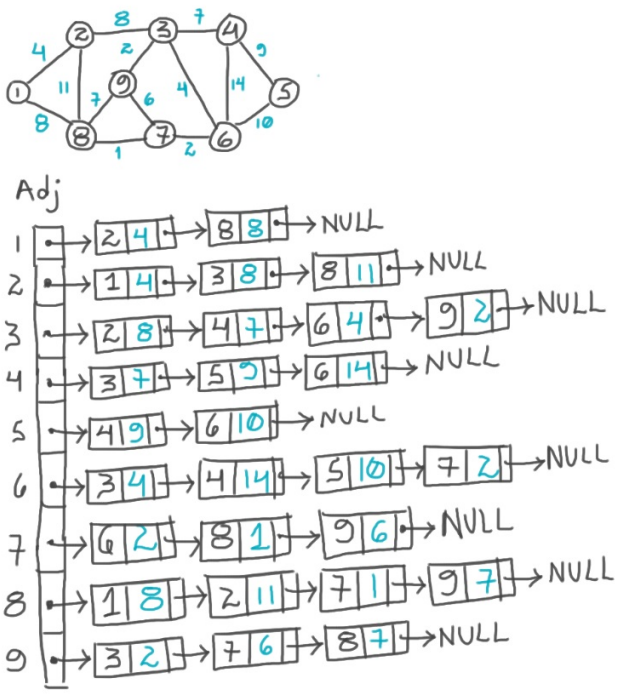


Figura 16: Grafo original que se utilizó para correr el programa del Algoritmo de Prim.

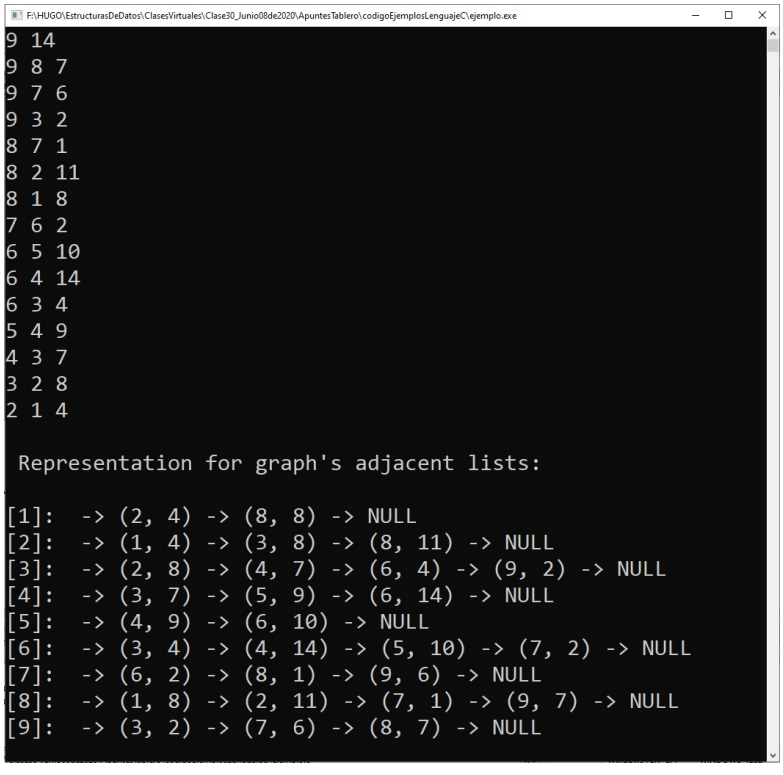


Figura 17: Parte superior de la salida del programa del Algoritmo de Prim.

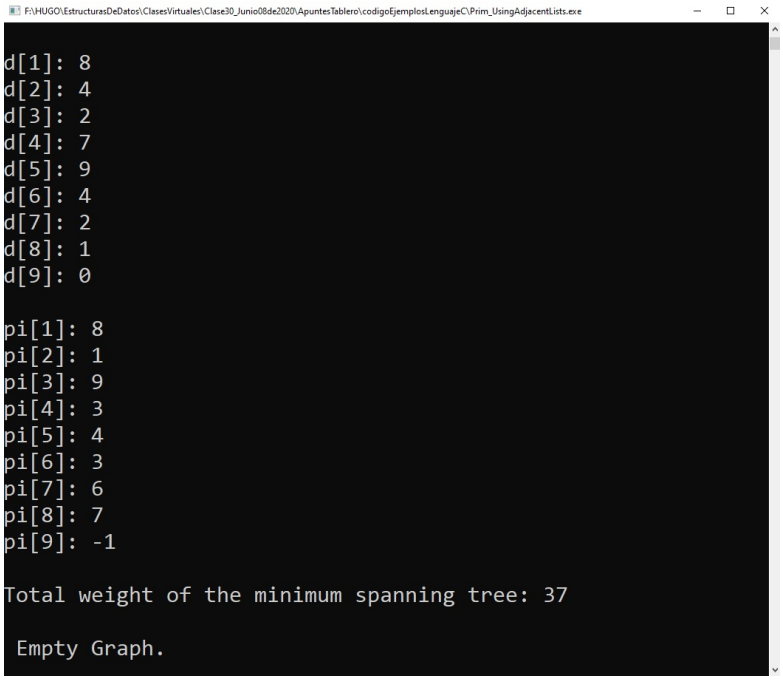
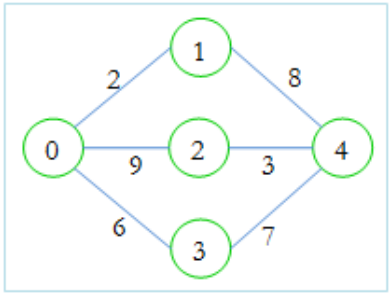


Figura 18: Parte inferior de la salida del programa del Algoritmo de Prim.

Programming Challenge: "Light OJ 1002 - Country Roads"

I am going to my home. There are many cities and many bi-directional roads between them. The cities are numbered from 0 to $n-1$ and each road has a cost. There are m roads. You are given the number of my city t where I belong. Now from each city you have to find the minimum cost to go to my city. The cost is defined by the cost of the maximum road you have used to go to my city.



For example, in the above picture, if we want to go from 0 to 4, then we can choose

1. 0 - 1 - 4 which costs 8, as 8 (1 - 4) is the maximum road we used
2. 0 - 2 - 4 which costs 9, as 9 (0 - 2) is the maximum road we used
3. 0 - 3 - 4 which costs 7, as 7 (3 - 4) is the maximum road we used

So, our result is 7, as we can use 0 - 3 - 4.

Input specification:

Input starts with an integer T (≤ 20), denoting the number of test cases.

Each case starts with a blank line and two integers n ($1 \leq n \leq 500$) and m ($0 \leq m \leq 16000$). The next m lines, each will contain three integers u, v, w ($0 \leq u, v < n,\ u \neq v,\ 1 \leq w \leq 20000$) indicating that there is a road between u and v with cost w. Then there will be a single integer t ($0 \leq t < n$). There can be multiple roads between two cities.

Output specification:

For each case, print the case number first. Then for all the cities (from 0 to $n-1$) you have to print the cost. If there is no such path, print 'Impossible'.

Example input:

```
2

5 6
0 1 5
0 1 4
2 1 3
3 0 7
3 4 6
3 1 8
1

5 4
0 1 5
0 1 4
2 1 3
3 4 7
1
```

Example output:

```
Case 1:
4
0
3
7
7
Case 2:
4
0
3
Impossible
Impossible
```

Solución del reto "Light OJ 1002 - Country Roads" utilizando el Algoritmo de Prim sobre el grafo

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXV 505
```

```c
#define myInfinite 2147483647
#define NIL -1
#define TRUE 1
#define FALSE 0

struct edge
{
    int vertex;
    int weight;
    struct edge *next;
};

struct graph
{
    int n_vertex;
    int m_edges;
    struct edge *Adj[MAXV];
};


struct graph *ReadGraph(int vertexes, int edges)
{
    int idVertex, idEdge, u, v, w;
    struct graph *G;
    struct edge *tempEdge;

    G = (struct graph *) malloc(sizeof(struct graph));
    G->n_vertex = vertexes;
    G->m_edges = edges;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        G->Adj[idVertex] = NULL;

    for(idEdge = 1; idEdge <= G->m_edges; idEdge++)
    {
        scanf("%d %d %d", &u, &v, &w);
        u++;
        v++;
        tempEdge = (struct edge *) malloc(sizeof(struct edge));
        tempEdge->vertex = v;
        tempEdge->weight = w;
        tempEdge->next = G->Adj[u];
        G->Adj[u] = tempEdge;

        if(u != v)
        {
            tempEdge = (struct edge *) malloc(sizeof(struct edge));
            tempEdge->vertex = u;
            tempEdge->weight = w;
            tempEdge->next = G->Adj[v];
            G->Adj[v] = tempEdge;
        }
    }
    return G;
}


struct graph *DeleteGraph(struct graph *G)
{
    int idVertex;
    struct edge *ActualEdge, *NextEdge;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        ActualEdge = G->Adj[idVertex];
        while(ActualEdge != NULL)
        {
            NextEdge = ActualEdge->next;
```

```
            free(ActualEdge);
            ActualEdge = NextEdge;
        }
    }
    free(G);
    G = NULL;
    return G;
}


int ExtractMin(int d[], int inQueue[], int n)
{
    int idVertex, minimum = myInfinite;
    int idMinimum = myInfinite;
    for(idVertex = 1; idVertex <= n; idVertex++)
    {
        if(inQueue[idVertex] == TRUE)
        {
            if(d[idVertex] < minimum)
            {
                minimum = d[idVertex];
                idMinimum = idVertex;
            }
        }
    }
    if(idMinimum != myInfinite)
        inQueue[idMinimum] = FALSE;

    return idMinimum;
}


void Prim(struct graph *G, int d[], int pi[], int s)
{
    int idVertex, u, v, w, Q[MAXV], inQueue[MAXV];
    int totalElementsInQueue = G->n_vertex;
    struct edge *tempEdge;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        d[idVertex] = myInfinite;
        pi[idVertex] = NIL;
        inQueue[idVertex] = TRUE;
    }
    d[s] = 0;

    while(totalElementsInQueue > 0)
    {
        u = ExtractMin(d, inQueue, G->n_vertex);

        if(u == myInfinite)
            break;

        totalElementsInQueue--;
        tempEdge = G->Adj[u];
        while(tempEdge != NULL)
        {
            v = tempEdge->vertex;
            w = tempEdge->weight;
            if((inQueue[v] == TRUE) && (d[v] > w))
            {
                d[v] = w;
                pi[v] = u;
            }
            tempEdge = tempEdge->next;
        }
    }
}
```

```
void solver(struct graph *G, int s)
{
    int d[MAXV], pi[MAXV], idVertex, vertex, maxWeightEdge;
    Prim(G, d, pi, s);

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        vertex = idVertex;
        maxWeightEdge = d[vertex];
        while((vertex != s) && (vertex != NIL))
        {
            if(d[vertex] > maxWeightEdge)
                maxWeightEdge = d[vertex];
            vertex = pi[vertex];
        }

        if(maxWeightEdge == myInfinite)
            printf("Impossible\n");
        else
            printf("%d\n", maxWeightEdge);
    }
}


int main()
{
    int vertexes, edges, totalCases, idCase, t;
    struct graph *G;

    scanf("%d", &totalCases);

    for(idCase = 1; idCase <= totalCases; idCase++)
    {
        scanf("%d %d", &vertexes, &edges);
        G = ReadGraph(vertexes, edges);
        scanf("%d", &t);
        t++;
        printf("Case %d:\n", idCase);
        solver(G, t);
        G = DeleteGraph(G);
    }
    return 0;
}
```

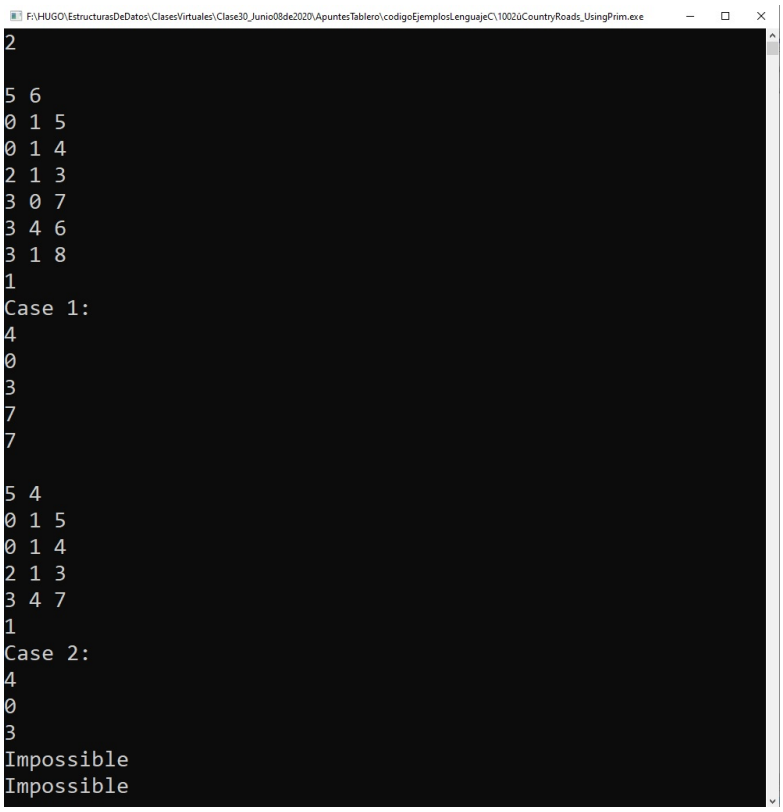


Figura 19: Salida del programa para el reto "Light OJ 1002 - Country Roads".

Algoritmo de Dijkstra

El algoritmo de Dijkstra soluciona el problema del camino más corto desde una sola fuente en un grafo con pesos $G = (V,\ E)$ para el caso en el que todos los pesos de las aristas son no negativos. Se asume que $w(u,\ v) \geq 0$ para cada arista $(u,\ v) \in E$.

En la siguiente pseudo-código, se usa una cola de mínima prioridad Q con respecto al valor de la distancia (d) de cada uno de los vértices.

Algoritmo de Dijkstra

function DIJKSTRA(G, w, s)

1 **for** each vertex $u \in V[G]$ **do**

2 $d[u] \leftarrow \infty$

3 $\pi[u] \leftarrow$ NIL

4 $d[s] \leftarrow 0$

5 $Q \leftarrow V[G]$

6 **while** $Q \neq \{\ \}$ **do**

7 $u \leftarrow$EXTRACT-MIN(Q)

8 **for** each vertex $v \in Adj[u]$ **do**

9 **if** $d[v] > d[u] + w(u, v)$ **then**

10 $d[v] \leftarrow d[u] + w(u, v)$

11 $\pi[v] \leftarrow u$

Implementación del Algoritmo de Dijkstra

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXV 1005
#define myInfinite 2147483647
#define NIL -1
#define TRUE 1
#define FALSE 0


struct edge
{
    int vertex;
    int weight;
    struct edge *next;
};



struct graph
{
    int n_vertex;
    int m_edges;
    struct edge *Adj[MAXV];
};
```

```
struct graph *ReadGraph(int vertexes, int edges)
{
    int idVertex, idEdge, u, v, w;
    struct graph *G;
    struct edge *tempEdge;

    G = (struct graph *) malloc(sizeof(struct graph));
    G->n_vertex = vertexes;
    G->m_edges = edges;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        G->Adj[idVertex] = NULL;

    for(idEdge = 1; idEdge <= G->m_edges; idEdge++)
    {
        scanf("%d %d %d", &u, &v, &w);
        tempEdge = (struct edge *) malloc(sizeof(struct edge));
        tempEdge->vertex = v;
        tempEdge->weight = w;
        tempEdge->next = G->Adj[u];
        G->Adj[u] = tempEdge;

        if(u != v)
        {
            tempEdge = (struct edge *) malloc(sizeof(struct edge));
            tempEdge->vertex = u;
            tempEdge->weight = w;
            tempEdge->next = G->Adj[v];
            G->Adj[v] = tempEdge;
        }
    }
    return G;
}


void PrintGraph(struct graph *G)
{
    int idVertex;
    struct edge *tempEdge;

    if(G != NULL)
    {
        printf("\n Representation for graph's adjacent lists: \n\n");
        for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        {
            printf("[%d]: ", idVertex);
            tempEdge = G->Adj[idVertex];
            while(tempEdge != NULL)
            {
                printf(" -> (%d, %d)", tempEdge->vertex, tempEdge->weight);
                tempEdge = tempEdge->next;
            }
            printf(" -> NULL\n");
        }
    }
    else
        printf("\n Empty Graph.\n");
}


struct graph *DeleteGraph(struct graph *G)
{
    int idVertex;
    struct edge *ActualEdge, *NextEdge;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
```

```
        ActualEdge = G->Adj[idVertex];
        while(ActualEdge != NULL)
        {
            NextEdge = ActualEdge->next;
            free(ActualEdge);
            ActualEdge = NextEdge;
        }
    }
    free(G);
    G = NULL;
    return G;
}


int ExtractMin(int d[], int inQueue[], int n)
{
    int idVertex, minimum = myInfinite;
    int idMinimum = myInfinite;
    for(idVertex = 1; idVertex <= n; idVertex++)
    {
        if(inQueue[idVertex] == TRUE)
        {
            if(d[idVertex] < minimum)
            {
                minimum = d[idVertex];
                idMinimum = idVertex;
            }
        }
    }
    if(idMinimum != myInfinite)
        inQueue[idMinimum] = FALSE;

    return idMinimum;
}


void Dijkstra(struct graph *G, int d[], int pi[], int s)
{
    int idVertex, u, v, w, Q[MAXV], inQueue[MAXV];
    int totalElementsInQueue = G->n_vertex;
    struct edge *tempEdge;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        d[idVertex] = myInfinite;
        pi[idVertex] = NIL;
        inQueue[idVertex] = TRUE;
    }

    d[s] = 0;

    while(totalElementsInQueue > 0)
    {
        u = ExtractMin(d, inQueue, G->n_vertex);

        if (u == myInfinite || d[u] == myInfinite)
            break;

        totalElementsInQueue--;
        tempEdge = G->Adj[u];

        while(tempEdge != NULL)
        {
            v = tempEdge->vertex;
            w = tempEdge->weight;
            if(d[v] > d[u] + w)
            {
                d[v] = d[u] + w;
```

```
                pi[v] = u;
            }
            tempEdge = tempEdge->next;
        }
    }
}


void solver(struct graph *G)
{
    int d[MAXV], pi[MAXV], idVertex;
    Dijkstra(G, d, pi, 1);

    printf("\n");
    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        printf("d[%d]: %d\n", idVertex, d[idVertex]);

    printf("\n");
    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        printf("pi[%d]: %d\n", idVertex, pi[idVertex]);

    printf("\n");
}


int main()
{
    int vertexes, edges;
    struct graph *G;

    while(scanf("%d %d", &vertexes, &edges) != EOF)
    {
        G = ReadGraph(vertexes, edges);
        PrintGraph(G);
        solver(G);
        G = DeleteGraph(G);
        PrintGraph(G);
    }
    return 0;
}
```

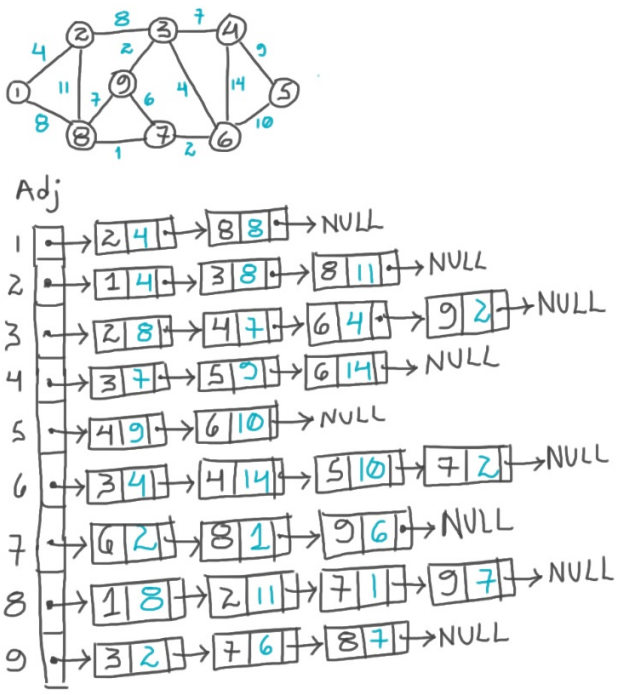


Figura 20: Grafo original que se utilizó para correr el programa del Algoritmo de Dijkstra.

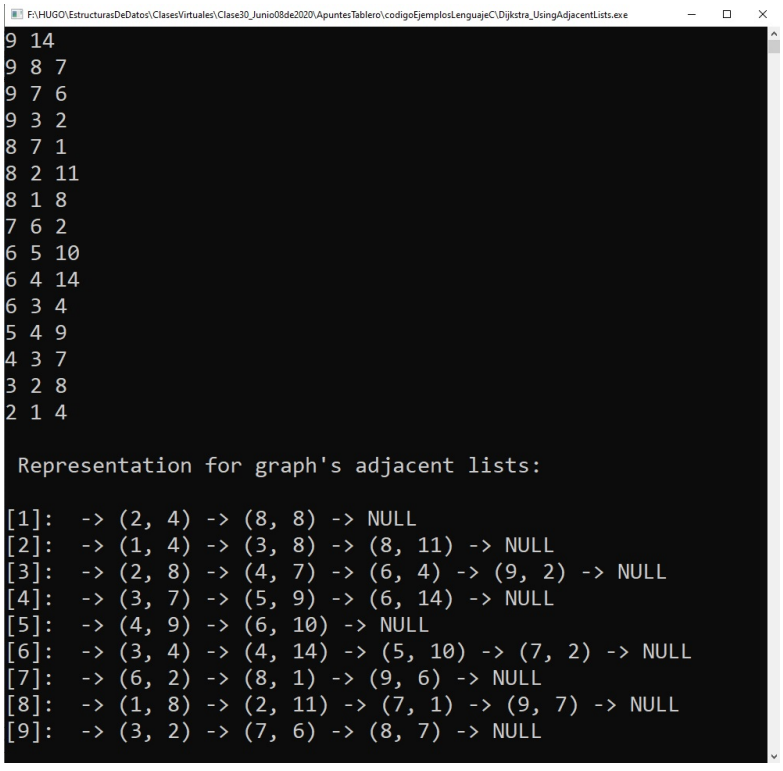


Figura 21: Parte superior de la salida del programa del Algoritmo de Dijkstra.

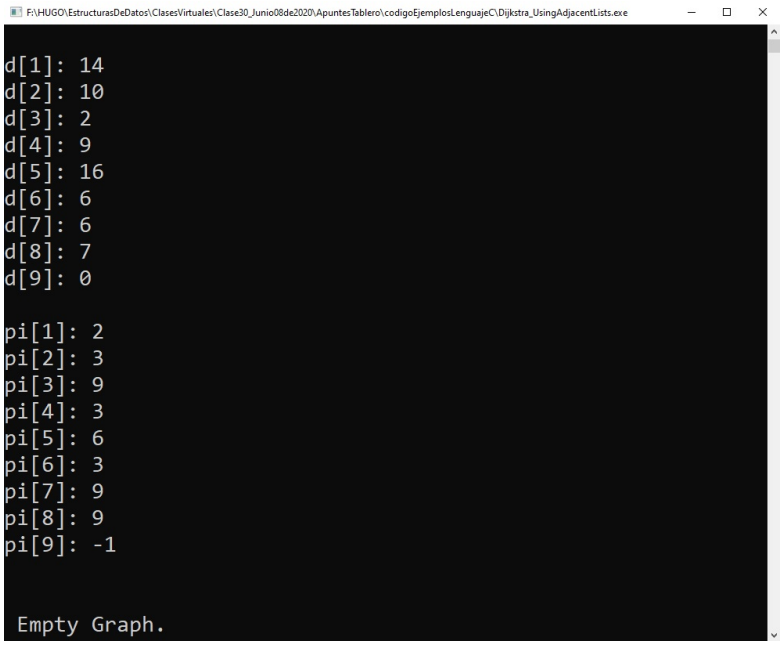


Figura 22: Parte inferior de la salida del programa del Algoritmo de Dijkstra.

Programming Challenge: "UVa - 11463 - Commandos"

A group of commandos were assigned a critical task. They are to destroy an enemy head quarter.

The enemy head quarter consists of several buildings and the buildings are connected by roads. The commandos must visit each building and place a bomb at the base of each building. They start their mission at the base of a particular building and from there they disseminate to reach each building. The commandos must use the available roads to travel between buildings. Any of them can visit one building after another, but they must all gather at a common place when their task in done.

In this problem, you will be given the description of different enemy headquarters. Your job is to determine the minimum time needed to complete the mission. Each commando takes exactly one unit of time to move between buildings.

You may assume that the time required to place a bomb is negligible. Each commando can carry unlimited number of bombs and there is an unlimited supply of commando troops for the mission.

Input specification:

The first line of input contains a number $T < 50$, where T denotes the number of test cases.

Each case describes one head quarter scenario. The first line of each case starts with a positive integer $N \leq 100$, where N denotes the number of buildings in the head quarter. The next line contains a positive integer R, where R is the number of roads connecting two buildings. Each of the next R lines contain two distinct numbers, $0 \leq u, v < N$, this means there is a road connecting building u to building v. The buildings are numbered from 0 to $N - 1$. The last line of each case contains two integers $0 \leq s, d < N$. Where s denotes the building from where the mission starts and d denotes the building where they must meet.

You may assume that two buildings will be directly connected by at most one road. The input will be such that, it will be possible to go from any building to another by using one or more roads.

Output specification:

For each case of input, there will be one line of output. It will contain the case number followed by the minimum time required to complete the mission. Look at the sample output for exact formatting.

Example input:

```
2
4
3
0 1
2 1
1 3
0 3
2
1
0 1
1 0
```

Example output:

```
Case 1: 4
Case 2: 1
```

Solución del reto "UVa - 11463 - Commandos" utilizando el Algoritmo de Dijkstra sobre el grafo

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXV 105
#define NIL -1
#define TRUE 1
#define FALSE 0
#define myInfinite 2147483647

struct edge
{
    int vertex;
    int weight;
    struct edge *next;
};

struct graph
{
    int n_vertex;
    int m_edges;
    struct edge *Adj[MAXV];
};

struct graph *ReadGraph(int vertexes, int edges)
{
    int idVertex, idEdge, u, v;
    struct graph *G;
    struct edge *tempEdge;

    G = (struct graph *) malloc(sizeof(struct graph));
    G->n_vertex = vertexes;
    G->m_edges = edges;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
        G->Adj[idVertex] = NULL;
```

```
    for(idEdge = 1; idEdge <= G->m_edges; idEdge++)
    {
        scanf("%d %d", &u, &v);
        u++;
        v++;
        tempEdge = (struct edge *) malloc(sizeof(struct edge));
        tempEdge->vertex = v;
        tempEdge->weight = 1;
        tempEdge->next = G->Adj[u];
        G->Adj[u] = tempEdge;

        if(u != v)
        {
            tempEdge = (struct edge *) malloc(sizeof(struct edge));
            tempEdge->vertex = u;
            tempEdge->weight = 1;
            tempEdge->next = G->Adj[v];
            G->Adj[v] = tempEdge;
        }
    }
    return G;
}


struct graph *DeleteGraph(struct graph *G)
{
    int idVertex;
    struct edge *ActualEdge, *NextEdge;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        ActualEdge = G->Adj[idVertex];
        while(ActualEdge != NULL)
        {
            NextEdge = ActualEdge->next;
            free(ActualEdge);
            ActualEdge = NextEdge;
        }
    }
    free(G);
    G = NULL;
    return G;
}


int ExtractMin(int d[], int inQueue[], int n)
{
    int idVertex, minimum = myInfinite;
    int idMinimum = myInfinite;
    for(idVertex = 1; idVertex <= n; idVertex++)
    {
        if(inQueue[idVertex] == TRUE)
        {
            if(d[idVertex] < minimum)
            {
                minimum = d[idVertex];
                idMinimum = idVertex;
            }
        }
    }
    if(idMinimum != myInfinite)
        inQueue[idMinimum] = FALSE;

    return idMinimum;
}
```

```
void Dijkstra(struct graph *G, int d[], int pi[], int s)
{
    int idVertex, u, v, w, Q[MAXV], inQueue[MAXV];
    int totalElementsInQueue = G->n_vertex;
    struct edge *tempEdge;

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        d[idVertex] = myInfinite;
        pi[idVertex] = NIL;
        inQueue[idVertex] = TRUE;
    }
    d[s] = 0;

    while(totalElementsInQueue > 0)
    {
        u = ExtractMin(d, inQueue, G->n_vertex);

        if (u == myInfinite || d[u] == myInfinite)
            break;

        totalElementsInQueue--;
        tempEdge = G->Adj[u];
        while(tempEdge != NULL)
        {
            v = tempEdge->vertex;
            w = tempEdge->weight;
            if(d[v] > d[u] + w)
            {
                d[v] = d[u] + w;
                pi[v] = u;
            }
            tempEdge = tempEdge->next;
        }
    }
}


int solver(struct graph *G, int s, int d)
{
    int dS[MAXV], piS[MAXV], dD[MAXV], piD[MAXV];
    int idVertex, result = 0;

    Dijkstra(G, dS, piS, s);
    Dijkstra(G, dD, piD, d);

    for(idVertex = 1; idVertex <= G->n_vertex; idVertex++)
    {
        if((dS[idVertex] != myInfinite) &&
           (dD[idVertex] != myInfinite) &&
           (dS[idVertex] + dD[idVertex]) > result)
            result = dS[idVertex] + dD[idVertex];
    }
    return result;
}


int main()
{
    int totalCases, idCase, vertexes, edges, s, d;
    struct graph *G;

    scanf("%d", &totalCases);
    for(idCase = 1; idCase <= totalCases; idCase++)
    {
        scanf("%d %d", &vertexes, &edges);
        G = ReadGraph(vertexes, edges);
        scanf("%d %d", &s, &d);
```

```
            s++;
            d++;
            printf("Case %d: %d\n", idCase, solver(G, s, d));
            G = DeleteGraph(G);
        }
        return 0;
}
```

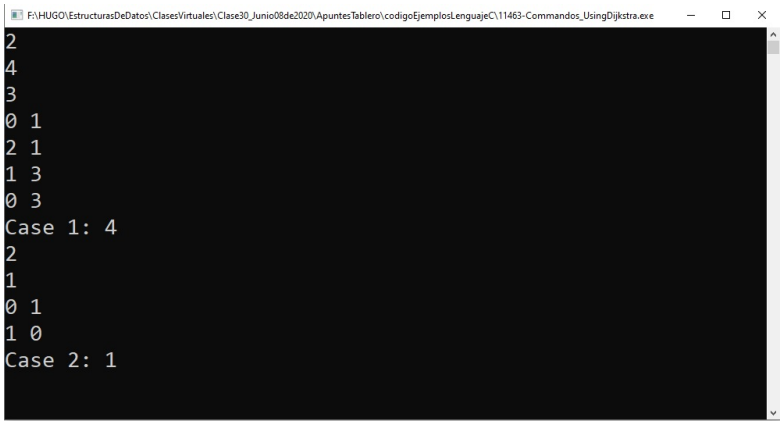


Figura 23: Salida del programa para el reto “UVa - 11463 - Commandos” utilizando el Algoritmo de Dijkstra.