**//Stack-----------------------------------------------------**

typedef struct{

int data[100];

int n;

}Stack;

void make\_null(Stack \*S){

S->n = 0;

}

void push(Stack \*S, int x){

S->data[S->n] = x;

S->n++;

}

int top(Stack \*S){

S->n--;

return S->data[S->n];

}

int empty(Stack S){

return S.n == 0;

}

**//DFSre---------------------------------------------------**

int mark[100], parent[100];

void DFS(Graph G, int u, int p){

if(mark[u] == 1) return;

mark[u] = 1;

parent[u] = p;

List lst = neighbors(G, u);

int i;

for(i = 0; i < lst.size; i++)

{

int y = lst.data[i];

//*Kiem tra chu trinh*

// if(mark[y] == 1 && parent[u] != y)

// {

// cycle = 1;

// return;

// }

if(mark[y] == 0)

DFS(G, y, u);

}

}

**//Queue---------------------------------------------------**

typedef struct{

int data[100];

int front, rear;

}Queue;

void make\_null(Queue \*Q){

Q->front = 0;

Q->rear = -1;

}

void push(Queue \*Q, int x){

Q->rear++;

Q->data[Q->rear] = x;

}

int top(Queue \*Q){

Q->front++;

return Q->data[Q->front-1];

}

int empty(Queue Q){

return Q.front > Q.rear;

}

-------------------------------------------------------------

int main(){

int n, m, u, v, i;

scanf("%d%d", &n, &m);

Graph G;

G.n = n;

for(i = 1; i <= m; i++)

{

scanf("%d%d", &u, &v);

G.A[u][v] = 1;

G.A[v][u] = 1;

}

for(i = 1; i <= n; i++)

{

mark[i] = 0;

parent[i] = 0;

}

for(i = 1; i <= n; i++)

if(mark[i] == 0)

DFS(G, i, 0);

for(i = 1; i <= n; i++)

printf("%d %d\n", i, parent[i]);

return 0;

}

**//BFS--------------------------------------------------------------**

int mark[100], parent[100];

void BFS(Graph G, int u){

int i;

Queue Q;

make\_null(&Q);

push(&Q, u);

while(!empty(Q))

{

int x = top(&Q);

if(mark[x] == 1)

continue;

mark[x] = 1;

List lst = neighbors(G, x);

for(i = 0; i < lst.size; i++)

{

int y = lst.data[i];

if(mark[y] == 0 && parent[y] == 0)

{

push(&Q, y);

parent[y] = x;

}

}

}

}

**//DFS--------------------------------------------------------------**

int mark[100], parent[100];

void DFS(Graph G, int u){

Stack S;

make\_null(&S);

push(&S, u);

while(!empty(S))

{

int x = top(&S);

if(mark[x] == 1)

continue;

mark[x] = 1;

List lst = neighbors(G, x);

int i;

for(i = 0; i < lst.size; i++)

{

int y = lst.data[i];

if(mark[y] == 0)

{

push(&S, y);

parent[y] = x;

}

}

}

}

---------------------------------------------------------------------

int main(){

int n, m, u, v, i;

scanf("%d%d", &n, &m);

Graph G;

G.n = n;

for(i = 1; i <= m; i++)

{

scanf("%d%d", &u, &v);

G.A[u][v] = 1;

G.A[v][u] = 1;

}

for(i = 1; i <= n; i++)

{

mark[i] = 0;

parent[i] = 0;

}

for(i = 1; i <= n; i++)

if(mark[i] == 0)

BFS(G, i);

for(i = 1; i <= n; i++)

printf("%d %d\n", i, parent[i]);

return 0;

}

**--------------------------------------------------------------**

int main(){

int n, m, u, v, i;

scanf("%d%d", &n, &m);

Graph G;

G.n = n;

for(i = 1; i <= m; i++)

{

scanf("%d%d", &u, &v);

G.A[u][v] = 1;

G.A[v][u] = 1;

}

for(i = 1; i <= n; i++)

{

mark[i] = 0;

parent[i] = 0;

}

for(i = 1; i <= n; i++)

DFS(G, i);

for(i = 1; i <= n; i++)

printf("%d %d\n", i, parent[i]);

return 0;

}

**//Bigraph--------------------------------------------------------------**

**//Chú ý: Khởi tạo mark của các đỉnh = -1 trong hàm main**

int mark[100], parent[100], check;

void DFS(Graph G, int u, int p){

if(mark[u] == -1)

{

mark[u] = p;

List lst = neighbors(G, u);

int i;

for(i = 0; i < lst.size; i++)

{

int y = lst.data[i];

DFS(G, y, !p);

}

}

else if(mark[u] != p)

check = 0;

}

**//Liên thông mạnh----------------------------------------------------**

int num[100], min\_num[100], mark[100], idx = 1, connect = 0;

Stack S;

void strong\_connect(Graph G, int x){

num[x] = min\_num[x] = idx; idx++;

push(&S, x);

mark[x] = 1;

List lst = neighbors(G, x);

int i;

for(i = 0; i < lst.size; i++)

{

int y = lst.data[i];

if(num[y] < 0)

{

strong\_connect(G, y);

min\_num[x] = min(min\_num[x], min\_num[y]);

}

else if(mark[y])

min\_num[x] = min(min\_num[x], num[y]);

}

if(num[x] == min\_num[x])

{

connect++;

int w;

do

{

w = top(&S);

mark[w] = 0;

}

while(w != x);

}

}

**//Moore-Dijkstra-----------------------------------------------------**

int mark[100], pi[100];

int choose(Graph G){

int min = 9999, idx = 0, i;

for(i = 1; i <= G.n; i++)

{

if(pi[i] < min && mark[i] == 0)

{

min = pi[i];

idx = i;

}

}

return idx;

}

void dijkstra(Graph G, int u){

int i;

for(i = 1; i <= G.n; i++)

{

mark[i] = 0;

pi[i] = 9999;

}

pi[u] = 0;

for(i = 1; i <= G.n; i++)

{

int x = choose(G);

mark[x] = 1;

List lst = neighbors(G, x);

int j;

for(j = 1; j <= lst.size; j++)

{

int y = lst.data[j-1];

if(G.A[i][y] > 0 && mark[y] == 0)

pi[y] = min(pi[y], G.A[i][y] + pi[i]);

}

}

}

*//In ra số đường đi ngắn nhất*

void dijkstra(Graph G, int u){

int i;

for(i = 1; i <= G.n; i++)

{

mark[i] = 0;

pi[i] = 9999;

cnt[i] = 1;

}

pi[u] = 0;

for(i = 1; i <= G.n; i++)

{

int x = choose(G);

mark[x] = 1;

List lst = neighbors(G, x);

int j;

for(j = 1; j <= lst.size; j++)

{

int y = lst.data[j-1];

if(G.A[i][y] > 0 && mark[y] == 0)

{

if(pi[y] > pi[i] + G.A[i][y])

{

pi[y] = pi[i] + G.A[i][y];

cnt[y] = cnt[i];

}

else if(pi[y] == pi[i] + G.A[i][y])

cnt[y] += cnt[i];

}

}

}

}

**//Bellman\_ford-----------------------------------------------------**

typedef struct{

int x, y, w;

}Edge;

typedef struct{

Edge A[100];

int n, m;

}Graph;

void init\_graph(Graph\* G, int n){

G->n = n;

G->m = 0;

}

void add\_edge(Graph \*G, int x, int y, int w){

G->A[G->m].x = x;

G->A[G->m].y = y;

G->A[G->m].w = w;

G->m++;

}

int pi[100], p[100];

void bellman\_ford(Graph G, int u){

int i, j;

for(i = 1; i <= G.n; i++)

{

pi[i] = 99999;

p[i] = -1;

}

pi[u] = 0;

for(i = 1; i < G.n; i++)

{

for(j = 0; j < G.m; j++)

{

int u = G.A[j].x;

int v = G.A[j].y;

int w = G.A[j].w;

if(pi[v] > pi[u] + w)

{

pi[v] = pi[u] + w;

p[v] = u;

}

}

}

for(i = 0; i < G.m; i++)

{

int u = G.Edges[i].x;

int v = G.Edges[i].y;

int w = G.Edges[i].w;

if(pi[v] > pi[u] + w)

{

cycle = 1;

break;

}

}

}

**//Xếp hạng đồ thị-----------------------------------------------------**

void copy\_list(List \*S1, List \*S2){

int i, x;

S1->size = 0;

for(i = 1; i <= S2->size; i++)

{

x = S2->data[i - 1];

S1->data[S1->size] = x;

S1->size++;

}

}

List L1, L2;

int rank[100];

void topo\_sort(Graph G){

L1.size = 0;

int i, j, d[100];

for(i = 1; i <= G.n; i++)

d[i] = degree(G, i);

for(i = 1; i <= G.n; i++)

if(d[i] == 0)

{

L1.data[L1.size] = i;

L1.size++;

}

int k = 0;

while(L1.size > 0)

{

L2.size = 0;

for(j = 0; j < L1.size; j++)

{

int x = L1.data[j];

rank[x] = k;

List lst = neighbors(G, x);

for(i = 0; i < lst.size; i++)

{

int y = lst.data[i];

d[y]--;

if(d[y] == 0)

{

L2.data[L2.size] = y;

L2.size++;

}

}

}

copy\_list(&L1, &L2);

k++;

}

}

**//Tổ chức thi công----------------------------------------------------**

void add\_edge(Graph\* G, int x, int y){

G->A[x][y] = 1;

G->A[y][x] = -1;

}

List L1, L2;

void neighbors(Graph G, int x){

int y;

L1.size = 0;

L2.size = 0;

for(y = 1; y <= G.n; y++)

{

if(G.A[x][y] == 1)

{

L1.data[L1.size] = y;

L1.size++;

}

else if(G.A[x][y] == -1)

{

L2.data[L2.size] = y;

L2.size++;

}

}

}

int min(int a, int b){

return a < b? a:b;

}

int max(int a, int b){

return a > b? a:b;

}

int t[100], T[100], d[100];

void cal(Graph G){

int i, j;

for(i = 1; i <= G.n; i++)

{

neighbors(G, i);

for(j = 0; j < L1.size; j++)

{

int y = L1.data[j];

t[y] = max(t[y], t[i] + d[i]);

}

}

for(i = G.n; i >= 1; i--)

{

T[G.n] = t[G.n];

neighbors(G, i);

for(j = 0; j < L2.size; j++)

{

int y = L2.data[j];

T[y] = min(T[y], T[i] - d[y]);

}

}

}

-------------------------------------------------------------------------

int main(){

Graph G;

int n, u, v;

scanf("%d", &n);

G.n = n+2;

for(u = 1; u <= n; u++)

{

scanf("%d", &d[u]);

do{

scanf("%d", &v);

if(v > 0) add\_edge(&G, v, u);

}

while(v > 0);

}

add\_edge(&G, n+1, 1);

add\_edge(&G, n, n+2);

int i;

for(i = 1; i <= n; i++)

T[i] = 99999;

cal(G);

printf("%d\n", t[n+2]);

for(i = 1; i <= G.n; i++)

{

printf("%d-%d\n", t[i], T[i]);

}

}

**//Kruskal--------------------------------------------------------------**

typedef struct{

int x, y, w;

}Edge;

typedef struct{

Edge A[100];

int n, m;

}Graph;

void init\_graph(Graph\* G, int n){

G->m = 0;

G->n = n;

}

void add\_edge(Graph \*G, int x, int y, int w){

G->A[G->m].x = x;

G->A[G->m].y = y;

G->A[G->m].w = w;

G->m++;

}

void sort(Graph \*G){

int i, j;

Edge t;

for(i = 0; i < G->m; i++)

for(j = i+1; j < G->m; j++)

if(G->A[i].w > G->A[j].w)

{

t = G->A[i];

G->A[i] = G->A[j];

G->A[j] = t;

}

}

…

**//Kruskal--------------------------------------------------------------**

…

int parent[100];

int find\_root(int u){

if (parent[u] == u)

return u;

return find\_root(parent[u]);

}

int kruskal(Graph \*G, Graph \*T){

int i;

sort(G);

init\_graph(T, G->n);

for(i = 1; i <= G->n; i++)

parent[i] = i;

int sum = 0;

for(i = 0; i < G->m; i++)

{

int u = G->A[i].x;

int v = G->A[i].y;

int w = G->A[i].w;

int u1 = find\_root(u);

int v1 = find\_root(v);

if(u1 != v1)

{

if(u < v) add\_edge(T, u, v, w);

else add\_edge(T, v, u, w);

parent[v1] = u1;

sum += w;

}

}

return sum;}

------------------------------------------------------------------------

int main(){

Graph G;

int n, m, u, v, w, i;

scanf("%d%d", &n, &m);

init\_graph(&G, n);

for(i = 0; i < m; i++)

{

scanf("%d%d%d", &u, &v, &w);

add\_edge(&G, u, v, w);

}

Graph T;

int sum = kruskal(&G, &T);

printf("%d\n", sum);

for(i = 0; i < T.m; i++)

printf("%d %d %d\n", T.A[i].x, T.A[i].y, T.A[i].w);

return 0;

}

// **Prim----------------------------------------------------------------**

int distanceFrom(int u, List L, Graph G){

int i, min\_dist = 99999, min = -1;

for(i = 0; i < L.size; i++)

{

int v = L.data[i];

if(G.A[u][v] != 0 && G.A[u][v] < min\_dist)

{

min\_dist = G.A[u][v];

min = v;

}

}

return min;

}

int mark[100], s = 0;

Edge E[100];

int prim(Graph G, Graph \*T){

int i, u, k = 0;

init\_graph(T, G.n);

List L;

L.size = 0;

for(i = 1; i < G.n; i++)

mark[i] = 0;

push(&L, 1);

mark[1] = 1;

for(i = 1; i < G.n; i++)

{

int min\_dist = 99999, min\_u, min\_v;

for(u = 1; u <= G.n; u++)

if(mark[u] == 0)

{

int v = distanceFrom(u, L, G);

if(v != -1 && G.A[u][v] < min\_dist)

{

min\_dist = G.A[u][v];

min\_u = u;

min\_v = v;

E[s].x = v;

E[s].y = u;

E[s].w = min\_dist;

s++;

}

}

push(&L, min\_u);

mark[min\_u] = 1;

add\_edge(T, min\_u, min\_v, min\_dist);

k += min\_dist;

}

return k;

}

// **Prim----------------------------------------------------------------**

int main(){

Graph G;

int n, m, u, v, w, i;

scanf("%d%d", &n, &m);

init\_graph(&G, n);

for(i = 0; i < m; i++)

{

scanf("%d%d%d", &u, &v, &w);

add\_edge(&G, u, v, w);

}

Graph T;

int k = prim(G, &T);

printf("%d\n", k);

for(i = 0; i < s; i++)

printf("%d %d %d\n", E[i].x, E[i].y, E[i].w);

return 0;

}

// **Ford-Fulkerson----------------------------------------------------**

typedef struct{

int C[100][100];

int F[100][100];

int n;

}Graph;

typedef struct{

int dir;

int sigma;

int pre;

}Label;

void init\_flow(Graph \*G){

int u, v;

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

for(v = 1; v <= G->n; v++)

G->F[u][v] = 0;

}

int min(int a, int b){

return a<b? a:b;

}

// **Ford-Fulkerson----------------------------------------------------**

Label labels[100];

int ford\_fullkerson(Graph G, int s, int t){

init\_flow(&G);

int u, v;

int sum\_flow = 0;

Queue Q;

do

{

for(u = 1; u <= G.n; u++)

labels[u].dir = 0;

labels[s].dir = 1;

labels[s].pre = s;

labels[s].sigma = 99999;

make\_null(&Q);

push(&Q, s);

int found = 0;

while(!empty(Q))

{

int u = top(&Q);

for(v = 1; v <= G.n; v++)

{

if(labels[v].dir == 0 && G.C[u][v] != 0 && G.F[u][v] < G.C[u][v])

{

labels[v].dir = 1;

labels[v].pre = u;

labels[v].sigma = min(labels[u].sigma, G.C[u][v] - G.F[u][v]);

push(&Q, v);

}

// if(labels[v].dir == 0 && G.C[v][u] != 0 && G.F[v][u] > 0)

// {

// labels[v].dir = -1;

// labels[v].pre = u;

// labels[v].sigma = min(labels[u].sigma, G.F[u][v]);

// push(&Q, v);

// }

}

if(labels[t].dir != 0)

{

found = 1;

break;

}

}

if(found == 1)

{

int x = t;

int sigma = labels[t].sigma;

sum\_flow += sigma;

while(x != s)

{

int u = labels[x].pre;

if (labels[x].dir==+1)

G->F[ labels[x].pre ][ x ]+=sigma;

if (labels[x].dir==-1)

G->F[ x ][ labels[x].pre ]-=sigma;

// **Ford-Fulkerson----------------------------------------------------**

x = u;

}

} else break;

}while(1);

return sum\_flow;

}

int main(){

Graph G;

int n, m, u, v, c ,e;

scanf("%d%d", &n, &m);

G.n = n;

for (e = 0; e < m; e++) {

scanf("%d%d%d", &u, &v, &c);

G.C[u][v] = c;

}

int max\_flow = ford\_fullkerson(G,1,n);

printf ("Max flow: %d \n",max\_flow);

printf("X0: ");

for ( e = 1; e <= G.n; e++) {

if (labels[e].dir != 0)

printf("%d ",e);

}

printf("\nY0: ");

for (e = 1; e <= G.n; e++) {

if(labels[e].dir == 0)

printf("%d ",e);

}

return 0;

}