图论

- 图论
  - 最短路
    - SPFA
    - Dijkstra
  - LCA
    - 倍增
    - Tarjan
  - 树的重心
  - 树链剖分
  - 虚树
  - 点分治
  - k 短路
  - Tarjan
    - 圆方树
  - 欧拉回路
  - 网络流
    - Dinic
    - EK
    - ISAP
    - 最高标号预流推进

最短路

SPFA

```cpp
// C++ Version
struct edge {
  int v, w;
};

vector<edge> e[maxn];
int dis[maxn], cnt[maxn], vis[maxn];
queue<int> q;

bool spfa(int n, int s) {
  memset(dis, 63, sizeof(dis));
  dis[s] = 0, vis[s] = 1;
  q.push(s);
  while (!q.empty()) {
    int u = q.front();
    q.pop(), vis[u] = 0;
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w;
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        cnt[v] = cnt[u] + 1;  // 记录最短路经过的边数
        if (cnt[v] >= n) return false;
        // 在不经过负环的情况下，最短路至多经过 n - 1 条边
        // 因此如果经过了多于 n 条边，一定说明经过了负环
        if (!vis[v]) q.push(v), vis[v] = 1;
      }
    }
  }
  return true;
}
```

Dijkstra

```cpp
// C++ Version
struct edge {
  int v, w;
};

vector<edge> e[maxn];
int dis[maxn], vis[maxn];

void dijkstra(int n, int s) {
  memset(dis, 63, sizeof(dis));
  dis[s] = 0;
  for (int i = 1; i <= n; i++) {
    int u = 0, mind = 0x3f3f3f3f;
    for (int j = 1; j <= n; j++)
      if (!vis[j] && dis[j] < mind) u = j, mind = dis[j];
    vis[u] = true;
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w;
      if (dis[v] > dis[u] + w) dis[v] = dis[u] + w;
    }
  }
}
```

LCA

倍增

```
#include <cstdio>
#define N 500005
struct Edge { int to, next; } edges[N << 1];
int head[N], tot, f[N][21], deep[N];
void add(int x, int y) { edges[++tot] = (Edge) { y, head[x] }, head[x] = tot; }
void dfs(int rt, int fa) {
        f[rt][0] = fa;
        deep[rt] = deep[fa] + 1;
        for (int i = 1; (1 << i) <= deep[rt]; i++)
                f[rt][i] = f[f[rt][i - 1]][i - 1];
        for (int x = head[rt]; x; x = edges[x].next)
                if (edges[x].to != fa) dfs(edges[x].to, rt);
}
int lca(int a, int b) {
        if (deep[a] < deep[b]) a ^= b, b ^= a, a ^= b;
        int len = deep[a] - deep[b], k = 0;
        while (len) {
                if (len & 1) a = f[a][k];
                k++, len >>= 1;
        }
        if (a == b) return a;
        for (int i = 20; i >= 0; i--)
                if (f[a][i] != f[b][i])
                        a = f[a][i], b = f[b][i];
        return f[a][0];
}
int main() {
        int n, m, s, a, b;
        scanf("%d%d%d", &n, &m, &s);
        for (int i = 1; i < n; i++)
                scanf("%d%d", &a, &b), add(a, b), add(b, a);
        dfs(s, 0);
        for (int i = 1; i <= m; i++)
                scanf("%d%d", &a, &b), printf("%d\n", lca(a, b));
        return 0;
}
```

Tarjan

```
#include <algorithm>
#include <iostream>
using namespace std;

class Edge {
 public:
  int toVertex, fromVertex;
  int next;
  int LCA;
  Edge() : toVertex(-1), fromVertex(-1), next(-1), LCA(-1){};
  Edge(int u, int v, int n) : fromVertex(u), toVertex(v), next(n), LCA(-1){};
};

const int MAX = 100;
int head[MAX], queryHead[MAX];
Edge edge[MAX], queryEdge[MAX];
int parent[MAX], visited[MAX];
int vertexCount, edgeCount, queryCount;

void init() {
  for (int i = 0; i <= vertexCount; i++) {
    parent[i] = i;
  }
}

int find(int x) {
  if (parent[x] == x) {
    return x;
  } else {
    return find(parent[x]);
  }
}

void tarjan(int u) {
  parent[u] = u;
  visited[u] = 1;

  for (int i = head[u]; i != -1; i = edge[i].next) {
    Edge& e = edge[i];
    if (!visited[e.toVertex]) {
      tarjan(e.toVertex);
      parent[e.toVertex] = u;
    }
  }

  for (int i = queryHead[u]; i != -1; i = queryEdge[i].next) {
    Edge& e = queryEdge[i];
    if (visited[e.toVertex]) {
```

```
      queryEdge[i ^ 1].LCA = e.LCA = find(e.toVertex);
    }
  }
}

int main() {
  memset(head, 0xff, sizeof(head));
  memset(queryHead, 0xff, sizeof(queryHead));

  cin >> vertexCount >> edgeCount >> queryCount;
  int count = 0;
  for (int i = 0; i < edgeCount; i++) {
    int start = 0, end = 0;
    cin >> start >> end;

    edge[count] = Edge(start, end, head[start]);
    head[start] = count;
    count++;

    edge[count] = Edge(end, start, head[end]);
    head[end] = count;
    count++;
  }

  count = 0;
  for (int i = 0; i < queryCount; i++) {
    int start = 0, end = 0;
    cin >> start >> end;

    queryEdge[count] = Edge(start, end, queryHead[start]);
    queryHead[start] = count;
    count++;

    queryEdge[count] = Edge(end, start, queryHead[end]);
    queryHead[end] = count;
    count++;
  }

  init();
  tarjan(1);

  for (int i = 0; i < queryCount; i++) {
    Edge& e = queryEdge[i * 2];
    cout << "(" << e.fromVertex << "," << e.toVertex << ") " << e.LCA << endl;
  }

  return 0;
}
```

树的重心

```
// 这份代码默认节点编号从 1 开始，即 i ∈ [1,n]
int size[MAXN],  // 这个节点的“大小”（所有子树上节点数 + 该节点）
    weight[MAXN],  // 这个节点的“重量”
    centroid[2];   // 用于记录树的重心（存的是节点编号）

void GetCentroid(int cur, int fa) {  // cur 表示当前节点 (current)
  size[cur] = 1;
  weight[cur] = 0;
  for (int i = head[cur]; i != -1; i = e[i].nxt) {
    if (e[i].to != fa) {  // e[i].to 表示这条有向边所通向的节点。
      GetCentroid(e[i].to, cur);
      size[cur] += size[e[i].to];
      weight[cur] = max(weight[cur], size[e[i].to]);
    }
  }
  weight[cur] = max(weight[cur], n - size[cur]);
  if (weight[cur] <= n / 2) {  // 依照树的重心的定义统计
    centroid[centroid[0] != 0] = cur;
  }
}
```

树链剖分

```
#include <cstdio>
#define N 100005
#define mid (l + r >> 1)
#define gc() (p1 == p2 ? (p2 = buf + fread(p1 = buf, 1, 1 << 20, stdin), p1 == p2 ? EOF
#define read() ({ register int x = 0, f = 1; register char c = gc(); while(c < '0' || c
char buf[1 << 20], *p1, *p2;
int head[N], tot, f[N], dep[N], len[N], son[N], top[N], size[N], n, m, r, P, val[N], df
struct Edge { int v, next; } e[N << 1];
void add(int x, int y) { e[++tot] = (Edge) { y, head[x] }, head[x] = tot; }
void dfs(int u) {
        len[u] = 1, size[u] = 1;
        for (int x = head[u]; x; x = e[x].next)
                if (e[x].v != f[u]) {
                        f[e[x].v] = u, dep[e[x].v] = dep[u] + 1, dfs(e[x].v);
                        size[u] += size[e[x].v];
                        if (len[u] < len[e[x].v] + 1) len[u] = len[e[x].v] + 1;
                        if (len[son[u]] < len[e[x].v]) son[u] = e[x].v;
                }
}
void join(int u, int t) {
        top[u] = t, dfn[u] = ++ts, pos[ts] = u;
        if (son[u]) join(son[u], t);
        for (int x = head[u]; x; x = e[x].next)
                if (e[x].v != f[u] && e[x].v != son[u]) join(e[x].v, e[x].v);
}
void build(int p, int l, int r) {
        if (l == r) return tr[p] = val[pos[l]], void();
        build(p << 1, l, mid), build(p << 1 | 1, mid + 1, r);
        tr[p] = (tr[p << 1] + tr[p << 1 | 1]) % P;
}
void add(int p, int l, int r, int s, int t, int v) {
        if (s <= l && r <= t) return (tr[p] += 1ll * (r - l + 1) * v % P) %= P, lz[p] +=
        if (lz[p]) {
                (tr[p << 1] += 1ll * lz[p] * (mid - l + 1) % P) %= P, (tr[p << 1 | 1] +=
                (lz[p << 1] += lz[p]) %= P, (lz[p << 1 | 1] += lz[p]) %= P, lz[p] = 0;
        }
        if (s <= mid) add(p << 1, l, mid, s, t, v);
        if (t > mid) add(p << 1 | 1, mid + 1, r, s, t, v);
        tr[p] = (tr[p << 1] + tr[p << 1 | 1]) % P;
}
void addpath(int a, int b, int z) {
        while(top[a] != top[b])
                if (dep[top[a]] < dep[top[b]]) add(1, 1, n, dfn[top[b]], dfn[b], z), b =
                else add(1, 1, n, dfn[top[a]], dfn[a], z), a = f[top[a]];
        dep[a] < dep[b] ? add(1, 1, n, dfn[a], dfn[b], z) : add(1, 1, n, dfn[b], dfn[a]
}
int ask(int p, int l, int r, int s, int t, int ans = 0) {
```

```
        if (s <= l && r <= t) return tr[p];
        if (lz[p]) {
                (tr[p << 1] += 1ll * lz[p] * (mid - l + 1) % P) %= P, (tr[p << 1 | 1] +=
                (lz[p << 1] += lz[p]) %= P, (lz[p << 1 | 1] += lz[p]) %= P, lz[p] = 0;
        }
        if (s <= mid) ans = ask(p << 1, l, mid, s, t);
        if (t > mid) (ans += ask(p << 1 | 1, mid + 1, r, s, t)) %= P;
        return ans;
}
int askpath(int a, int b, int ans = 0) {
        while(top[a] != top[b])
                if (dep[top[a]] < dep[top[b]]) (ans += ask(1, 1, n, dfn[top[b]], dfn[b]
                else (ans += ask(1, 1, n, dfn[top[a]], dfn[a])) %= P, a = f[top[a]];
        (ans += dep[a] < dep[b] ? ask(1, 1, n, dfn[a], dfn[b]) : ask(1, 1, n, dfn[b], d
        return ans;
}
void print(int p, int l, int r) {
        if (l == r) return printf("%d ", tr[p]), void();
        if (lz[p]) {
                (tr[p << 1] += 1ll * lz[p] * (mid - l + 1) % p) %= P, (tr[p << 1 | 1] +=
                (lz[p << 1] += lz[p]) %= P, (lz[p << 1 | 1] += lz[p]) %= P, lz[p] = 0;
        }
        print(p << 1, l, mid), print(p << 1 | 1, mid + 1, r);
}
int main() {
        n = read(), m = read(), r = read(), P = read();
        for (int i = 1; i <= n; i++) val[i] = read();
        for (int i = 1, x, y; i < n; i++) x = read(), y = read(), add(x, y), add(y, x);
        dfs(r), join(r, r), build(1, 1, n);
        for (int op, x, y, z; m; m--) {
                op = read(), x = read();
                if (op == 1) y = read(), z = read(), addpath(x, y, z);
                else if (op == 2) y = read(), printf("%d\n", askpath(x, y));
                else if (op == 3) z = read(), add(1, 1, n, dfn[x], dfn[x] + size[x] - 1
                else printf("%d\n", ask(1, 1, n, dfn[x], dfn[x] + size[x] - 1));
        }
        return 0;
}
```

虚树

```
inline bool cmp(const int x, const int y) { return id[x] < id[y]; }

void build() {
  sort(h + 1, h + k + 1, cmp);
  sta[top = 1] = 1, g.sz = 0, g.head[1] = -1;
  // 1 号节点入栈，清空 1 号节点对应的邻接表，设置邻接表边数为 1
  for (int i = 1, l; i <= k; ++i)
    if (h[i] != 1) {
      // 如果 1 号节点是关键节点就不要重复添加
      l = lca(h[i], sta[top]);
      // 计算当前节点与栈顶节点的 LCA
      if (l != sta[top]) {
        // 如果 LCA 和栈顶元素不同，则说明当前节点不再当前栈所存的链上
        while (id[l] < id[sta[top - 1]])
          // 当次大节点的 Dfs 序大于 LCA 的 Dfs 序
          g.push(sta[top - 1], sta[top]), top--;
        // 把与当前节点所在的链不重合的链连接掉并且弹出
        if (id[l] > id[sta[top - 1]])
          // 如果 LCA 不等于次大节点（这里的大于其实和不等于没有区别）
          g.head[l] = -1, g.push(l, sta[top]), sta[top] = l;
        // 说明 LCA 是第一次入栈，清空其邻接表，连边后弹出栈顶元素，并将 LCA
        // 入栈
        else
          g.push(l, sta[top--]);
        // 说明 LCA 就是次大节点，直接弹出栈顶元素
      }
      g.head[h[i]] = -1, sta[++top] = h[i];
      // 当前节点必然是第一次入栈，清空邻接表并入栈
    }
  for (int i = 1; i < top; ++i)
    g.push(sta[i], sta[i + 1]);  // 剩余的最后一条链连接一下
  return;
}
```

点分治

```
#include <bits/stdc++.h>
#define N 10005
#define gc() (p1 == p2 ? (p2 = buf + fread(p1 = buf, 1, 1 << 20, stdin), p1 == p2 ? EOF
#define read() ({ register int x = 0; register char c = gc(); while(c < '0' || c > '9')
using namespace std;
char buf[1 << 20];
int n, m, head[N], tot = 1, size[N], maxsize[N], root, k, cnt, query[101], ok[101];
bool vis[N << 1], flag;
struct Edge { int v, w, next; } e[N << 1];
struct Node { int d, b; } h[N];
bool cmp(Node a, Node b) { return a.d < b.d; }
void add(int x, int y, int w) { e[++tot] = (Edge) { y, w, head[x] }, head[x] = tot; }
void getroot(int u, int f, int tsize) {
        size[u] = 1, maxsize[u] = 0;
        for (int x = head[u]; x; x = e[x].next)
                if (e[x].v != f && !vis[x]) {
                        getroot(e[x].v, u, tsize);
                        size[u] += size[e[x].v];
                        maxsize[u] = max(maxsize[u], size[e[x].v]);
                }
        maxsize[u] = max(maxsize[u], tsize - size[u]);
        if (maxsize[root] > maxsize[u]) root = u;
}
void dfs(int u, int fa, int d, int bel) {
        h[++cnt].d = d, h[cnt].b = bel;
        for (int x = head[u]; x; x = e[x].next)
                if (e[x].v != fa && !vis[x]) dfs(e[x].v, u, d + e[x].w, bel);
}
void divide(int u, int f) {
        h[1].d = h[1].b = 0;
        cnt = 1;
        for (register int x = head[u]; x; x = e[x].next) if (!vis[x]) dfs(e[x].v, u, e[
        sort(h + 1, h + cnt + 1, cmp);
        for (register int i = 1; i <= m; i++) {
                const int k = query[i];
                if (!ok[i])
                        for (register int l = 1, r = cnt; l <= cnt; l++) {
                                while(h[l].d + h[r].d > k && r >= 1) r--;
                                if (h[l].d + h[r].d == k && h[l].b != h[r].b) {
                                        ok[i] = 1;
                                        break;
                                }
                        }
        }
        for (int x = head[u]; x; x = e[x].next) {
                if (e[x].v != f && !vis[x]) {
                        vis[x] = vis[x ^ 1] = 1;
```

```
                        maxsize[root = 0] = INT_MAX, getroot(e[x].v, 0, size[e[x].v]);
                        divide(root, u);
                }
        }
}
int main() {
    register char *p1, *p2;
        n = read(), m = read();
        for (register int i = 1, u, v, w; i < n; i++)
                u = read(), v = read(), add(u, v, w = read()), add(v, u, w);
        maxsize[root = 0] = INT_MAX, getroot(1, 0, n);
        for (register int i = 1; i <= m; i++) query[i] = read();
        divide(root, 0);
        for (register int i = 1; i <= m; i++) puts(ok[i] ? "AYE" : "NAY");
}
```

k 短路

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;
const int maxn = 5010;
const int maxm = 400010;
const int inf = 2e9;
int n, m, s, t, k, u, v, ww, H[maxn], cnt[maxn];
int cur, h[maxn], nxt[maxm], p[maxm], w[maxm];
int cur1, h1[maxn], nxt1[maxm], p1[maxm], w1[maxm];
bool tf[maxn];

void add_edge(int x, int y, double z) {
  cur++;
  nxt[cur] = h[x];
  h[x] = cur;
  p[cur] = y;
  w[cur] = z;
}

void add_edge1(int x, int y, double z) {
  cur1++;
  nxt1[cur1] = h1[x];
  h1[x] = cur1;
  p1[cur1] = y;
  w1[cur1] = z;
}

struct node {
  int x, v;

  bool operator<(node a) const { return v + H[x] > a.v + H[a.x]; }
};

priority_queue<node> q;

struct node2 {
  int x, v;

  bool operator<(node2 a) const { return v > a.v; }
} x;

priority_queue<node2> Q;

int main() {
  scanf("%d%d%d%d%d", &n, &m, &s, &t, &k);
```

```
  while (m--) {
    scanf("%d%d%d", &u, &v, &ww);
    add_edge(u, v, ww);
    add_edge1(v, u, ww);
  }
  for (int i = 1; i <= n; i++) H[i] = inf;
  Q.push({t, 0});
  while (!Q.empty()) {
    x = Q.top();
    Q.pop();
    if (tf[x.x]) continue;
    tf[x.x] = true;
    H[x.x] = x.v;
    for (int j = h1[x.x]; j; j = nxt1[j]) Q.push({p1[j], x.v + w1[j]});
  }
  q.push({s, 0});
  while (!q.empty()) {
    node x = q.top();
    q.pop();
    cnt[x.x]++;
    if (x.x == t && cnt[x.x] == k) {
      printf("%d\n", x.v);
      return 0;
    }
    if (cnt[x.x] > k) continue;
    for (int j = h[x.x]; j; j = nxt[j]) q.push({p[j], x.v + w[j]});
  }
  printf("-1\n");
  return 0;
}
```

Tarjan

```cpp
// C++ Version
int dfn[N], low[N], dfncnt, s[N], in_stack[N], tp;
int scc[N], sc;  // 结点 i 所在 SCC 的编号
int sz[N];       // 强连通 i 的大小

void tarjan(int u) {
  low[u] = dfn[u] = ++dfncnt, s[++tp] = u, in_stack[u] = 1;
  for (int i = h[u]; i; i = e[i].nex) {
    const int &v = e[i].t;
    if (!dfn[v]) {
      tarjan(v);
      low[u] = min(low[u], low[v]);
    } else if (in_stack[v]) {
      low[u] = min(low[u], dfn[v]);
    }
  }
  if (dfn[u] == low[u]) {
    ++sc;
    while (s[tp] != u) {
      scc[s[tp]] = sc;
      sz[sc]++;
      in_stack[s[tp]] = 0;
      --tp;
    }
    scc[s[tp]] = sc;
    sz[sc]++;
    in_stack[s[tp]] = 0;
    --tp;
  }
}
```

圆方树

```cpp
#include <algorithm>
#include <cstdio>
#include <vector>

const int MN = 100005;

int N, M, cnt;
std::vector<int> G[MN], T[MN * 2];

int dfn[MN], low[MN], dfc;
int stk[MN], tp;

void Tarjan(int u) {
  printf("  Enter : #%d\n", u);
  low[u] = dfn[u] = ++dfc;                // low 初始化为当前节点 dfn
  stk[++tp] = u;                          // 加入栈中
  for (int v : G[u]) {                    // 遍历 u 的相邻节点
    if (!dfn[v]) {                        // 如果未访问过
      Tarjan(v);                          // 递归
      low[u] = std::min(low[u], low[v]);  // 未访问的和 low 取 min
      if (low[v] == dfn[u]) {  // 标志着找到一个以 u 为根的点双连通分量
        ++cnt;                 // 增加方点个数
        printf("  Found a New BCC #%d.\n", cnt - N);
        // 将点双中除了 u 的点退栈，并在圆方树中连边
        for (int x = 0; x != v; --tp) {
          x = stk[tp];
          T[cnt].push_back(x);
          T[x].push_back(cnt);
          printf("    BCC #%d has vertex #%d\n", cnt - N, x);
        }
        // 注意 u 自身也要连边（但不退栈）
        T[cnt].push_back(u);
        T[u].push_back(cnt);
        printf("    BCC #%d has vertex #%d\n", cnt - N, u);
      }
    } else
      low[u] = std::min(low[u], dfn[v]);  // 已访问的和 dfn 取 min
  }
  printf("  Exit : #%d : low = %d\n", u, low[u]);
  printf("  Stack:\n    ");
  for (int i = 1; i <= tp; ++i) printf("%d, ", stk[i]);
  puts("");
}

int main() {
  scanf("%d%d", &N, &M);
  cnt = N;  // 点双 / 方点标号从 N 开始
  for (int i = 1; i <= M; ++i) {
```

```
    int u, v;
    scanf("%d%d", &u, &v);
    G[u].push_back(v);  // 加双向边
    G[v].push_back(u);
  }
  // 处理非连通图
  for (int u = 1; u <= N; ++u)
    if (!dfn[u]) Tarjan(u), --tp;
  // 注意到退出 Tarjan 时栈中还有一个元素即根，将其退栈
  return 0;
}
```

欧拉回路

```cpp
#include <algorithm>
#include <cstdio>
#include <stack>
#include <vector>
using namespace std;

struct edge {
    int to;
    bool exists;
    int revref;

    bool operator<(const edge& b) const { return to < b.to; }
};

vector<edge> beg[505];
int cnt[505];

const int dn = 500;
stack<int> ans;

void Hierholzer(int x) {  // 关键函数
    for (int& i = cnt[x]; i < (int)beg[x].size();) {
        if (beg[x][i].exists) {
            edge e = beg[x][i];
            beg[x][i].exists = 0;
            beg[e.to][e.revref].exists = 0;
            ++i;
            Hierholzer(e.to);
        } else {
            ++i;
        }
    }
    ans.push(x);
}

int deg[505];
int reftop[505];

int main() {
    for (int i = 1; i <= dn; ++i) {
        beg[i].reserve(1050);  // vector 用 reserve 避免动态分配空间，加快速度
    }

    int m;
    scanf("%d", &m);
    for (int i = 1; i <= m; ++i) {
        int a, b;
```

```
        scanf("%d%d", &a, &b);
        beg[a].push_back((edge){b, 1, 0});
        beg[b].push_back((edge){a, 1, 0});
        ++deg[a];
        ++deg[b];
    }

    for (int i = 1; i <= dn; ++i) {
        if (!beg[i].empty()) {
            sort(beg[i].begin(), beg[i].end());  // 为了要按字典序贪心，必须排序
        }
    }

    for (int i = 1; i <= dn; ++i) {
        for (int j = 0; j < (int)beg[i].size(); ++j) {
            beg[i][j].revref = reftop[beg[i][j].to]++;
        }
    }

    int bv = 0;
    for (int i = 1; i <= dn; ++i) {
        if (!deg[bv] && deg[i]) {
            bv = i;
        } else if (!(deg[bv] & 1) && (deg[i] & 1)) {
            bv = i;
        }
    }

    Hierholzer(bv);

    while (!ans.empty()) {
        printf("%d\n", ans.top());
        ans.pop();
    }
}
```

网络流

Dinic

```
const int N = 300005, INF = 0x7fffffff;
int n, s, t, head[N], tot = 1, q[N], h, j, d[N], g[N], ans;
struct Edge { int v, next, c; } e[1000005];
void add(int x, int y, int c, int C = 0) {
    e[++tot] = (Edge) { y, head[x], c }, head[x] = tot;
    e[++tot] = (Edge) { x, head[y], C }, head[y] = tot;
}
bool bfs() {
        memset(d, 0, sizeof d);
        d[s] = 1, h = 1, j = 1, q[h] = s, g[s] = head[s];
        while(h <= j) {
                int u = q[h++];
                for (int x = head[u]; x; x = e[x].next)
                if (e[x].c && !d[e[x].v]) {
                        d[e[x].v] = d[u] + 1, q[++j] = e[x].v, g[e[x].v] = head[e[x].v]
                        if (e[x].v == t) return 1;
                }
        }
        return 0;
}
int dinic(int u, int flow) {
        if (u == t) return flow;
        int rest = flow, k;
        for (int x = g[u]; x && rest; g[u] = x, x = e[x].next)
                if (e[x].c && d[e[x].v] == d[u] + 1) {
                        k = dinic(e[x].v, std::min(rest, e[x].c));
                        if (!k) d[e[x].v] = 0;
                        e[x].c -= k, e[x ^ 1].c += k, rest -= k;
                }
        return flow - rest;
}
int main() {
    // ...建图...
    while(bfs()) ans += dinic(s, INF);
    printf("%d\n", ans);
}
```

EK

```
#define maxn 250
#define INF 0x3f3f3f3f

struct Edge {
  int from, to, cap, flow;

  Edge(int u, int v, int c, int f) : from(u), to(v), cap(c), flow(f) {}
};

struct EK {
  int n, m;              // n: 点数, m: 边数
  vector<Edge> edges;    // edges: 所有边的集合
  vector<int> G[maxn];   // G: 点 x -> x 的所有边在 edges 中的下标
  int a[maxn], p[maxn];  // a: 点 x -> BFS 过程中最近接近点 x 的边给它的最大流
                         // p: 点 x -> BFS 过程中最近接近点 x 的边

  void init(int n) {
    for (int i = 0; i < n; i++) G[i].clear();
    edges.clear();
  }

  void AddEdge(int from, int to, int cap) {
    edges.push_back(Edge(from, to, cap, 0));
    edges.push_back(Edge(to, from, 0, 0));
    m = edges.size();
    G[from].push_back(m - 2);
    G[to].push_back(m - 1);
  }

  int Maxflow(int s, int t) {
    int flow = 0;
    for (;;) {
      memset(a, 0, sizeof(a));
      queue<int> Q;
      Q.push(s);
      a[s] = INF;
      while (!Q.empty()) {
        int x = Q.front();
        Q.pop();
        for (int i = 0; i < G[x].size(); i++) {  // 遍历以 x 作为起点的边
          Edge& e = edges[G[x][i]];
          if (!a[e.to] && e.cap > e.flow) {
            p[e.to] = G[x][i];  // G[x][i] 是最近接近点 e.to 的边
            a[e.to] =
                min(a[x], e.cap - e.flow);  // 最近接近点 e.to 的边赋给它的流
            Q.push(e.to);
          }
        }
```

```
        if (a[t]) break;  // 如果汇点接受到了流，就退出 BFS
      }
      if (!a[t])
        break;  // 如果汇点没有接受到流，说明源点和汇点不在同一个连通分量上
      for (int u = t; u != s;
           u = edges[p[u]].from) {  // 通过 u 追寻 BFS 过程中 s -> t 的路径
        edges[p[u]].flow += a[t];       // 增加路径上边的 flow 值
        edges[p[u] ^ 1].flow -= a[t];  // 减小反向路径的 flow 值
      }
      flow += a[t];
    }
    return flow;
  }
};
```

ISAP

```
struct Edge {
  int from, to, cap, flow;

  Edge(int u, int v, int c, int f) : from(u), to(v), cap(c), flow(f) {}
};

bool operator<(const Edge& a, const Edge& b) {
  return a.from < b.from || (a.from == b.from && a.to < b.to);
}

struct ISAP {
  int n, m, s, t;
  vector<Edge> edges;
  vector<int> G[maxn];
  bool vis[maxn];
  int d[maxn];
  int cur[maxn];
  int p[maxn];
  int num[maxn];

  void AddEdge(int from, int to, int cap) {
    edges.push_back(Edge(from, to, cap, 0));
    edges.push_back(Edge(to, from, 0, 0));
    m = edges.size();
    G[from].push_back(m - 2);
    G[to].push_back(m - 1);
  }

  bool BFS() {
    memset(vis, 0, sizeof(vis));
    queue<int> Q;
    Q.push(t);
    vis[t] = 1;
    d[t] = 0;
    while (!Q.empty()) {
      int x = Q.front();
      Q.pop();
      for (int i = 0; i < G[x].size(); i++) {
        Edge& e = edges[G[x][i] ^ 1];
        if (!vis[e.from] && e.cap > e.flow) {
          vis[e.from] = 1;
          d[e.from] = d[x] + 1;
          Q.push(e.from);
        }
      }
    }
    return vis[s];
  }
```

```
void init(int n) {
  this->n = n;
  for (int i = 0; i < n; i++) G[i].clear();
  edges.clear();
}

int Augment() {
  int x = t, a = INF;
  while (x != s) {
    Edge& e = edges[p[x]];
    a = min(a, e.cap - e.flow);
    x = edges[p[x]].from;
  }
  x = t;
  while (x != s) {
    edges[p[x]].flow += a;
    edges[p[x] ^ 1].flow -= a;
    x = edges[p[x]].from;
  }
  return a;
}

int Maxflow(int s, int t) {
  this->s = s;
  this->t = t;
  int flow = 0;
  BFS();
  memset(num, 0, sizeof(num));
  for (int i = 0; i < n; i++) num[d[i]]++;
  int x = s;
  memset(cur, 0, sizeof(cur));
  while (d[s] < n) {
    if (x == t) {
      flow += Augment();
      x = s;
    }
    int ok = 0;
    for (int i = cur[x]; i < G[x].size(); i++) {
      Edge& e = edges[G[x][i]];
      if (e.cap > e.flow && d[x] == d[e.to] + 1) {
        ok = 1;
        p[e.to] = G[x][i];
        cur[x] = i;
        x = e.to;
        break;
      }
    }
    if (!ok) {
      int m = n - 1;
      for (int i = 0; i < G[x].size(); i++) {
```

```
          Edge& e = edges[G[x][i]];
          if (e.cap > e.flow) m = min(m, d[e.to]);
        }
        if (--num[d[x]] == 0) break;
        num[d[x] = m + 1]++;
        cur[x] = 0;
        if (x != s) x = edges[p[x]].from;
      }
    }
    return flow;
  }
};
```

最高标号预流推进

```
#include <cstdio>
#include <cstring>
#include <queue>
#include <stack>
using namespace std;
const int N = 1200, M = 120000, INF = 0x3f3f3f3f;
int n, m, s, t;

struct qxx {
  int nex, t, v;
};

qxx e[M * 2 + 1];
int h[N + 1], cnt = 1;

void add_path(int f, int t, int v) { e[++cnt] = (qxx){h[f], t, v}, h[f] = cnt; }

void add_flow(int f, int t, int v) {
  add_path(f, t, v);
  add_path(t, f, 0);
}

int ht[N + 1], ex[N + 1],
    gap[N];  // 高度; 超额流; gap 优化 gap[i] 为高度为 i 的节点的数量
stack<int> B[N];  // 桶 B[i] 中记录所有 ht[v]==i 的v
int level = 0;    // 溢出节点的最高高度

int push(int u) {      // 尽可能通过能够推送的边推送超额流
  bool init = u == s;  // 是否在初始化
  for (int i = h[u]; i; i = e[i].nex) {
    const int &v = e[i].t, &w = e[i].v;
    if (!w || init == false && ht[u] != ht[v] + 1)  // 初始化时不考虑高度差为1
      continue;
    int k = init ? w : min(w, ex[u]);
    // 取到剩余容量和超额流的最小值，初始化时可以使源的溢出量为负数。
    if (v != s && v != t && !ex[v]) B[ht[v]].push(v), level = max(level, ht[v]);
    ex[u] -= k, ex[v] += k, e[i].v -= k, e[i ^ 1].v += k;  // push
    if (!ex[u]) return 0;  // 如果已经推送完就返回
  }
  return 1;
}

void relabel(int u) {  // 重贴标签（高度）
  ht[u] = INF;
  for (int i = h[u]; i; i = e[i].nex)
    if (e[i].v) ht[u] = min(ht[u], ht[e[i].t]);
  if (++ht[u] < n) {  // 只处理高度小于 n 的节点
    B[ht[u]].push(u);
```

```
    level = max(level, ht[u]);
    ++gap[ht[u]];  // 新的高度，更新 gap
  }
}

bool bfs_init() {
  memset(ht, 0x3f, sizeof(ht));
  queue<int> q;
  q.push(t), ht[t] = 0;
  while (q.size()) {  // 反向 BFS，遇到没有访问过的结点就入队
    int u = q.front();
    q.pop();
    for (int i = h[u]; i; i = e[i].nex) {
      const int &v = e[i].t;
      if (e[i ^ 1].v && ht[v] > ht[u] + 1) ht[v] = ht[u] + 1, q.push(v);
    }
  }
  return ht[s] != INF;  // 如果图不连通，返回 0
}

// 选出当前高度最大的节点之一，如果已经没有溢出节点返回 0
int select() {
  while (B[level].size() == 0 && level > -1) level--;
  return level == -1 ? 0 : B[level].top();
}

int hlpp() {                    // 返回最大流
  if (!bfs_init()) return 0;  // 图不连通
  memset(gap, 0, sizeof(gap));
  for (int i = 1; i <= n; i++)
    if (ht[i] != INF) gap[ht[i]]++;  // 初始化 gap
  ht[s] = n;
  push(s);  // 初始化预流
  int u;
  while ((u = select())) {
    B[level].pop();
    if (push(u)) {  // 仍然溢出
      if (!--gap[ht[u]])
        for (int i = 1; i <= n; i++)
          if (i != s && i != t && ht[i] > ht[u] && ht[i] < n + 1)
            ht[i] = n + 1;  // 这里重贴成 n+1 的节点都不是溢出节点
      relabel(u);
    }
  }
  return ex[t];
}

int main() {
  scanf("%d%d%d%d", &n, &m, &s, &t);
  for (int i = 1, u, v, w; i <= m; i++) {
    scanf("%d%d%d", &u, &v, &w);
```

```
    add_flow(u, v, w);
  }
  printf("%d", hlpp());
  return 0;
}
```