算法作业4

丁元杰 17231164

2019 年 12 月 21 日

1 判断题

1.1

正确。

1.2

错误，现有理论没有证明也没能证否这一点。

1.3

正确。

1.4

错误，这是必要条件而非充分条件。

2 最短路问题

如图所示的图中，Dijkstra 算法失效，原因如下。最开始出队的两个结点依次为 2 和 1，在这两个点出队以后，其最短路已经确定。但是使用 3 号结点继续更新其邻接的点时，会发现 2 号结点的距离可以进一步地缩短，从而导致 2 号结点需要再次入队。这样就破坏了 Dijkstra 所依赖的假设，从而使 Dijkstra 失效。

3 二分图判定问题

3.1

证 设二分图 $G=(V,E)$ 中存在一条奇环 $P=(v_0,v_1,\dots,v_{2K+1})$，其中 $v_0=v_{2K+1}$。

设 v_0 所属的二分图集合为 V^*，其中 $V^*=V_1$ 或 $V^*=V_2$，记 $V_c^*=V-V^*$，为 v_0 不在的那个二分图集合。那么由于二分图的限制，显然有

1. $v_{2k}\in V^*$，其中 $k=0,1,\dots,K$
2. $v_{2k+1}\in V_c^*$，其中 $k=0,1,\dots,K$

那么得出 $v_0 \in V^*$，而 $v_{2K+1} \in V_c^*$，又由于 $v_0 = v_{2K+1}$，所以 $v_0 \in V_c^*$。得出 $v_0 \in V_1 \cup V_2 \neq \emptyset$。由此得出原图并非二分图。

所以二分图中不存在奇环。

3.2 二分图判定算法

3.2.1 基本思想

二分图的色数为 2，因此可以使用 0-1 染色的方式判断某一个图是否为一个二分图。这里定义颜色 $c : V \to -1, 0, 1$，其中 -1 表示暂时无法确定的颜色，$0, 1$ 则表示二分图中应当填入的颜色。再记录一个集合 U，表示已经入队的结点集合，并在入队结束后不进行删除。设某一个结点 v，它的颜色为 $c(v)$，与一系列的点相邻 $v_1, v_2, \ldots, v_K$，则尝试依次将 v_k 入队，并分情况讨论：

1. $c(v_k) = -1$，则设置它的颜色 $c(v_k) = 1 - c(v)$ 后入队；
2. $c(v_k)/neq - 1$ 并且 $c(v_k) = 1 - c(v)$，忽略该点；
3. $c(v_k)/neq - 1$ 并且 $c(v_k) = c(v)$，说明发现冲突，该图不是二分图，算法结束。

由于该图未必是连通图，在从任意一个结点开始后，需要继续向后扫描未到达的点，以其为入口继续计算。

如果算法结束还没有发现任何一处冲突，则说明该图为一个二分图。

3.2.2 真代码

二分图判定方法的真代码如下

```
#include <cstdio>
#include <queue>

using namespace std;

const int MAX_N = 100, MAX_M = 200;
int N, M;     // N = #V, M = #E
int cne;
int color[MAX_N];

struct edge {
    int to;
    edge *nxt;
} E[MAX_M * 2], *node[MAX_N];

void add_edge(int f, int t) {
    E[cne].to = t;
    E[cne].nxt = node[f];
```

```
    node[f] = E+cne++;
}

void readin() {
    int a, b;
    scanf("%d%d", &N, &M);
    for(int i = 1; i <= M; ++i) {
        scanf("%d%d", &a, &b);
        add_edge(a, b);
        add_edge(b, a);
    }
    fill_n(color, N, -1);
}

bool bfs(int x) {
    queue<int> que;
    que.push(x);
    color[x] = 0;
    while(!que.empty()) {
        int now = que.front();
        que.pop();
        for(edge *i = node[now]; i; i = i->nxt) {
            int to = i->to;
            if (color[to] == -1) {
                color[to] = color[now] ^ 1;
                que.push(to);
            } else if (color[to] == color[now]) {
                continue;
            } else {
                return false;
            }
        }
    }
    return true;
}

bool check() {
    for (int i = 1; i <= N; ++i) {
        if (color[i] == -1) {
            if (!bfs(i)) {
                return false;
```

```
                }
            }
        }
        return true;
}

void process() {
        if (check()) {
                printf("YES!\n");
        } else {
                printf("NO!\n");
        }
}

int main() {

        readin();
        process();

        return 0;
}
```

3.2.3 正确性及复杂度分析

如果原图为二分图，BFS 的算法保证了每一个结点至少都会入队一次，从而会出队一次，在每个结点出队之时，都会检查其向外连出的边，对没有分配颜色的点分配颜色，对已经分配颜色的点检查颜色。因此，这种检查之下不会漏掉任何一种二分图中的染色限制，因而如果算法能够得到一个染色，必然是一个合法的染色。由于构造性地给出了一种染色方案，这就说明了原图为二分图。

此外，如果原图并非二分图，则说明不存在任何一种合法的染色方案，但是显然，对于二分图任何一个连通块，其染色方案只有两种，$c_1, c_2 : V \to 0, 1$，并且 $c_1 = 1 - c_2$。所以任选一个点做起点，任意向其指派一种颜色，便可以推得该连通块上的所有点的颜色，实际上方案只与初始点的颜色相关。而初始点分别为两种颜色的方案具有明显的对称性，同时存在和不存在，因而只需要任取一种颜色作为初始点的颜色，进行如上的检查即可。

最后关于算法的复杂度，每一个结点都只会被染色一次，也即入队和出队一次。在出队之时，会遍历与之相邻的所有点，因此会遍历图中的每一条边。综上所述，算法的时间复杂度为：

$$T(V, E) = O(V + E)$$

4 配对问题

4.1 基本思想

首先对所有的对有序化并去重，显然这一步不会影响最后的答案。设操作之后的对偶数量为 m'，集合为 S。现在引入一些记号，$d(x)$ 表示包含 x 这个整数的整数对的数量。如果两个整数 $x \neq y$ 是最终的答案，那么必然有 $d(x)+d(y)=m'$，此时 $(x,y) \notin S$；或者有 $d(x)+d(y)=m'+2$，此时 $(x,y) \in S$。因此，可以考虑预处理 d 的值，并建立按照从 d 到 x 的索引表 d^{-1}，然后根据

5 瓶颈值问题

5.1 算法思路

可以借鉴 Prim 的思想，从起点出发构造子图，逐步放宽瓶颈值的条件，直到终点被包含在了子图之中。具体而言，在过程中维护一个大根堆 H，其中保存了与当前子图相连的边的编号，按照其边的长短进行排序。记录了一个值 h，表示子图中最小的边权为 h，并且与子图相邻的所有边都小于 h。以此为基础进行拓展。设 H 的堆顶元素（最大元素）为 x，其对应的边为 e_x，那么：

1. 将 e_x 加入子图中，h 与 e_x 的长度 $d(e_x)$ 取最小值。
2. 如果与 e_x 相连的另一个节点 y 尚未被加入到子图中，将 y 加入子图，并且将与 y 相邻的所有边入堆。如果此时 $y=T$，是全局的终点，那么算法结束，h 就是最终的瓶颈值。

5.2 真代码

该算法的真代码如下

```
#include <cstdio>
#include <queue>
#include <utility>
#include <climits>

using namespace std;

const int MAX_N = 100, MAX_M = 200;
int N, M;     // N = #V, M = #E
int S, T;
int cne;
bool in[MAX_N];
typedef pair<int, int> pii;

struct edge {
    int to, v;
    edge *nxt;
```

```
} E[MAX_M * 2], *node[MAX_N];

void add_edge(int f, int t, int v) {
    E[cne].to = t;
    E[cne].v = v;
    E[cne].nxt = node[f];
    node[f] = E+cne++;
}

void readin() {
    int a, b, v;
    scanf("%d%d", &N, &M);
    scanf("%d%d", &S, &T);
    for(int i = 1; i <= M; ++i) {
        scanf("%d%d%d", &a, &b, &v);
        add_edge(a, b, v);
        add_edge(b, a, v);
    }
}


int prim_like(int x) {
    priority_queue<pii> que;
    in[x] = true;
    int H = INT_MAX;
    for (edge *i = node[x]; i; i = i->nxt) {
        que.push({i->v, i - E});
    }
    while (!que.empty()) {
        auto p = que.top();
        que.pop();
        H = min(H, p.first);
        edge *e = E + p.second;
        if (!in[e->to]) {
            in[e->to] = true;
            if (e->to == T) {
                return H;
            }
            for (edge *i = node[e->to]; i; i = i->nxt) {
                que.push({i->v, i - E});
            }
```

```
            }
        }
        return -1;
}

void process() {
        int ans = prim_like(S);
        if (ans == -1) {
                printf("No Solution!\n");
        } else {
                printf("%d\n", ans);
        }
}

int main() {

        readin();
        process();

        return 0;
}
```

5.3 复杂度分析

很明显，由于每个结点只会被到达一次，每一个边都会入队一次，因此时间复杂度为：

$$T(V,E) = O(V + E \log E)$$