0x70 图论

0x71 邻接表

邻接表

```
int h[N], e[N], ne[N], idx = 0;
memset(h, -1, sizeof h);

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}
```

带边权的邻接表

```
int h[N], w[N], e[N], ne[N], idx = 0;
memset(h, -1, sizeof h);

void add(int a, int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++ ;
}
```

0x72 最短路

单源最短路径

堆优化 $Dijkstra$

- 邻接表，时间复杂度 $O(MlogN)$。

```
vector<PII> g[N];   // 边权邻接表(x, y, c)
int dijkstra(int sta, int ed)
{
    bool st[n + 1] = {0};
    vector<int> dis(n + 1, inf);
    priority_queue<PII, vector<PII>, greater<PII>> heap;

    dis[sta] = 0;
    heap.push({dis[sta], sta});

    while (!heap.empty())
    {
        auto [d, x] = heap.top(); heap.pop();
```

```
        if (st[x]) continue;
        st[x] = true;

        for (auto [y, w] : g[x])
            if (dis[y] > dis[x] + w)
            {
                dis[y] = dis[x] + w;
                heap.push({dis[y], y});
            }
    }

    return dis[ed];
}
```

$spfa$ 算法

- 时间复杂度 $O(M) \sim O(NM)$。

```
vector<PII> g[N];   // 边权邻接表
int spfa(int sta, int ed)
{
    bool st[n + 1] = {0};
    vector<int> dis(n + 1, inf);
    queue<int> q;

    st[sta] = true, dis[sta] = 0;
    q.push(sta);

    while (!q.empty())
    {
        auto x = q.front(); q.pop();
        st[x] = false;

        for (auto [y, w] : g[x])
            if (dis[y] > dis[x] + w)
            {
                dis[y] = dis[x] + w;
                if (!st[y])
                {
                    q.push(y);
                    st[y] = true;
                }
            }
    }

    return dis[ed];
}
```

判负环

```
bool spfa()
{
    bool st[n + 1] = {0};
    vector<int> dis(n + 1), cnt(n + 1); // 判断负环不需要对dis数组进行初始化
    queue<int> q;

    // 不仅仅从1开始，以为1的路径中可能不存在负环，因此要将所有点加入队列
    for (int i = 1; i <= n; i ++ ) q.push(i), st[i] = true;

    while (!q.empty())
    {
        auto x = q.front(); q.pop();
        st[x] = false;

        for (auto [y, w] : g[x])
        {
            if (dis[y] > dis[x] + w)  // 若存在负权，dis数组就改变
            {
                dis[y] = dis[x] + w;
                cnt[y] = cnt[x] + 1;

                // 如果从某一点到j点的路径中存在大于等于n条边，说明路径中存在负权回路
                // 或者说如果j点被改变了n多次，就说明路径中存在负权回路
                if (cnt[y] >= n) return true;
                if (!st[y])
                {
                    q.push(y);
                    st[y] = true;
                }
            }
        }
    }

    return false;
}
```

多源最短路径

> 在任意两点间最短路问题中，图一般比较稠密，所以用邻接矩阵来实现最合适不过。

Floyd

- 初始化

```
int dis[N][N];
for (int i = 1; i <= n; i ++ )
    for (int j = 1; j <= n; j ++ )
    {
        if (i == j) dis[i][j] = 0;
```

```
            else dis[i][j] = inf;
        }
```

- 时间复杂度 $O(N^3)$

```
void floyd()
{
    for (int k = 1; k <= n; k ++ )
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= n; j ++ )
                dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
}
```

0x73 最小生成树

> 给定一张边带权的无向图 $G=(V,E)$，$n=|V|$，$m=|E|$。由 V 中全部 n 个顶点和 E 中 $n-1$ 条边构成的无向连通子图被称为 G 的一个生成树。边的权值之和最小的生成树被称为无向图 G 的最小生成树。

- **定理**：任意一颗最小生成树一定包含无向图中权值最小的边。
- **推论**：给定一张无向图 $G=(V,E)$，$n=|V|$，$m=|E|$。从 E 中选出 $k<n-1$ 条边构成 G 的一个生成森林。

$Kruskal$ 算法

- 时间复杂度 $O(MlogM)$

```
struct Edge
{
    int x, y, w;
    bool operator< (const Edge& W) const
    {
        return w < W.w;
    }
} edges[M]; // 注意是边数

int kruskal()
{
    sort(edges, edges + m);
    DSU dsu(n);

    int cnt = 1, res = 0;
    for (int i = 0; i < m; i ++ )
    {
        auto [x, y, w] = edges[i];
        if (dsu.same(x, y)) continue;

        dsu.merge(x, y);
        cnt ++ , res += w;
```

```
    }

    if (cnt != n) return inf;
    else return res;
}
```

$Prim$ 算法

- $Prim$ 主要用于稠密图，尤其是完全图的最小生成树的求解。
- 初始化

  ```
  int g[N][N];
  memset(g, 0x3f, sizeof g);
  ```

- 时间复杂度 $O(N^2)$。

```
int prim()
{
    vector<int> dis(n + 1, inf);  // dis[i]表示点i到最小生成树的距离
    bool st[n + 1] = {0};

    int res = 0;  // res表示最小生成树的权重和
    for (int i = 0; i < n; i ++ )
    {
      // 找到距离最小生成树最近的点
        int t = -1;
        for (int j = 1; j <= n; j ++ )
        {
            if (!st[j] && (t == -1 || dis[t] > dis[j])) t = j;
        }

        // 如果加入最小生成树的点不是第一个点并且点t距离最小生成树的距离为INF，则不存在最小生成树
        if (i && dis[t] == inf) return inf;

        // 如果加入最小生成树的点不是第一个点，更新最小生成树的权重和
        if (i) res += dis[t];
        st[t] = true;

        // 用点t来更新最小生成树外的点到最小生成树的距离
        for (int j = 1; j <= n; j ++ ) dis[j] = min(dis[j], g[t][j]);
    }

    return res;
}
```

0x7A 欧拉图

定义

- **欧拉路径**：通过图中所有边恰好一次的通路。
- **欧拉回路**：通过图中所有边恰好一次的回路。
- **半欧拉图**：具有欧拉通路但不具有欧拉回路的无向图或有向图。

性质

- 欧拉图中所有**顶点的度数之和为偶数**。
- 对于无向连通图来说，
  - 存在欧拉路径的充分必要条件：度数为奇数的点只能有 0 或 2 个。
  - 存在欧拉回路的充分必要条件：度数为奇数的点只能有 0 个。
- 对于有向连通图来说，
  - 存在欧拉路径的充分必要条件：所有点的入度均等于出度；或存在两个点，其中点（起点）入度 = 出度 - 1，另一个点（终点）的出度 = 入度 - 1，剩余的点出度均等于入度。
  - 存在欧拉回路的充分必要条件：所有点的入度均等于出度。

0x7F 二分图匹配

> 如果一张无向图的 N 个节点 $(N \geq 2)$ 可以分成 A, B 两个非空集合，其中 $A \cap B = \emptyset$，并且在同一集合内的点之间都没有边相连，那么称这张无向图为一张**二分图**，A, B 分别成为二分图的左部和右部。

二分图判定

- **定理**：一张无向图是二分图，当且仅当图中不存在奇环（长度为奇数的环）。
- 根据该定理，我们可以用**染色法**进行二分图的判定。