作业八

T1

> 假设以邻接表表示法作为图的存储结构，设计图的深度优先遍历递归算法。

以无向图为例。为了防止重边和自环，在遍历时需要额外申请空间用来记录顶点的访问状态。时间复杂度 $O(n+e)$，其中 n 为顶点数，e 为边数。

```cpp
template<class T>
void ALGraph<T>::dfs() {
    vector<bool> vis(vex_cnt, false);

    // lambda dfs
    function<void(int)> dfs = [&](int now) {
        cout << now << " ";
        vis[now] = true;
        for (EdgeNode<T>* p = head[now].next; p; p = p->next) {
            if (!vis[p->toid]) {
                dfs(p->toid);
            }
        }
    };

    for (int i = 0; i < vex_cnt; i++) {
        if (!vis[i]) {
            cout << "Connected Component: ";
            dfs(i);
            cout << endl;
        }
    }
}
```

T2

> 试基于图的广度优先搜索策略编写一算法，判别以邻接表方式存储的有向图中是否存在由顶点 v_i 到顶点 v_j 的路径 $(i \neq j)$。

从 v_i 开始 bfs 遍历直到遇到 v_j 为止。时间复杂度 $O(n+e)$

```cpp
template<class T>
bool ALGraph<T>::findPathFromA2B(int a, int b) {
    if (a < 0 || a >= vex_cnt || b < 0 || b >= vex_cnt) {
        cerr << "Wrong Input! Out of Range!\n";
        exit(1);
    }

    vector<bool> vis(vex_cnt, false);
    auto bfs = [&](int start) -> bool {
        queue<int> q;
        vis[start] = true;
        q.push(start);
        while (q.size()) {
            auto now = q.front();
            q.pop();

            // find path
            if (now == b) {
                return true;
            }

            for (EdgeNode<T>* p = head[now].next; p; p = p->next) {
                if (!vis[p->toid]) {
                    vis[p->toid] = true;
                    q.push(p->toid);
                }
            }
        }
        return false;
    };

    return bfs(a);
}
```

T3

> 采用邻接表作存储结构，编写一个判别无向图中任意两个给定的两个顶点之间是否存在一条长度为 k 的简单路径的算法。

从起点开始 dfs 并记录当前搜索路径中的结点数，当遇到终点并且路径长度为 k 表示找到了长度为 k 的简单路径。为了便于判定，我们记录路径上的点。时间复杂度 $O(n+e)$

```cpp
template<class T>
vector<vector<int>> ALGraph<T>::findPathFromA2BLengthK(int a, int b, int k) {
    if (a < 0 || a >= vex_cnt || b < 0 || b >= vex_cnt) {
        cerr << "Wrong Input! Out of Range!\n";
        exit(1);
    }

    vector<bool> vis(vex_cnt, false);
    vector<vector<int>> paths;
    vector<int> path;

    // lambda function
    function<void(int)> dfs = [&](int now) {
        vis[now] = true;
        path.push_back(now);
        if (now == b && path.size() - 1 == k) {
            // store path length k to paths
            paths.push_back(path);
            vis[now] = false;
            path.pop_back();
            return;
        }
        for (EdgeNode<T>* p = head[now].next; p; p = p->next) {
            if (!vis[p->toid]) {
                dfs(p->toid);
            }
        }
        vis[now] = false;
        path.pop_back();
    };

    dfs(a);

    return paths;
}
```

T4

> 假设以邻接矩阵作为图的存储结构，编写算法判断在给定的有向图中是否存在一个简单有向回路。若存在，则以顶点序列的方式输出该回路（找到一条即可）。（选做题）

思路：

- 我们将此题分解为两个部分，判环与存环
- 判环：由于是有向图，判环更加简单，只需要直接判断当前遍历到的结点是否已经被标记过即可。如果被标记过了，就寻找到了一个环
- 存环：在上述判环时，如果判定到了一个环，就从那个点出发，将路径上已经被标记过的点进行存储即可

时间复杂度 $O(n^2)$

- 对于邻接矩阵，每一个结点在搜索邻接点时都是 $O(n)$，最坏的情况就是每一个点都遍历到，于是就是 $O(n^2)$

```cpp
template<class T>
vector<int> MGraph<T>::findDigraphLoop() {
    vector<int> loop;
    bool find_one_loop = false;
    vector<bool> vis(vex_cnt, false);
    vector<bool> vis2(vex_cnt, false);

    // store loop
    function<void(int)> getLoop = [&](int now) {
        vis2[now] = true;
        loop.push_back(now);
        for (int i = 0; i < vex_cnt; i++) {
            if (edges[now][i] != INF && vis[i] && !vis2[i]) {
                getLoop(i);
            }
        }
    };

    // check loop
    function<void(int)> dfs = [&](int now) {
        if (find_one_loop) {
            return;
        }
        if (vis[now]) {
```

```
            find_one_loop = true;
            getLoop(now);
            return;
        }
        vis[now] = true;
        for (int i = 0; i < vex_cnt; i++) {
            if (edges[now][i] != INF) {
                dfs(i);
            }
        }
        vis[now] = false;
    };

    // check all connected components
    for (int i = 0; i < vex_cnt; i++) {
        if (!find_one_loop && !vis[i]) {
            dfs(i);
        }
    }

    return loop;
}
```

T5

> 假设以邻接矩阵作为图的存储结构，设计一个算法判断一个给定无向图中是否存在回路？若存在，则以顶点序列的方式输出该回路。

思路：

- 同样按照 T4 的思路，分为判环与存环
- 判环：判环相比于有向图增加了一个 trick，即在探索当前点 `now` 的可达点时，要排除当前点的上一级点 `before`。排除方法就是在 dfs 搜索时，多传递一个参数即可
- 存环：存环相比于有向图需要改变一下策略，因为对于有向图，一旦遇到已经遇到的点 `v` 之后，直接从 `v` 点开始探索已标记的点即可；但是对于无向图这种策略是错误的，因为会有环之外的点被记录进去，解决方法就是：在 dfs 搜索记录路径时，采用 「**双端队列**」的形式，在探索到已探索的点时，将路径队列进行处理即可。处理方法就是对比队列的队头与队尾，不断弹出队头直到队头与队尾相等即可。

时间复杂度 $O(n^2)$

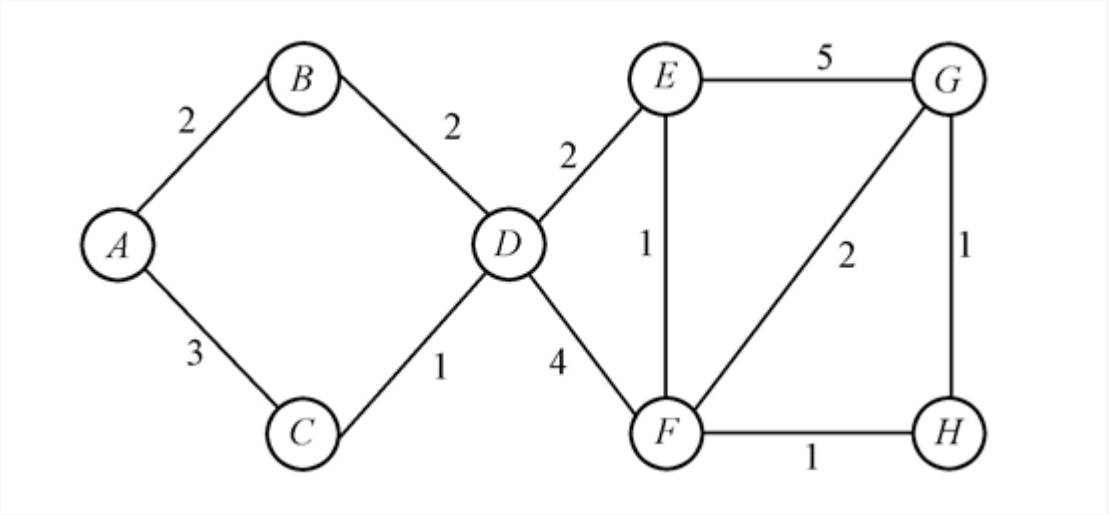
```cpp
template<class T>
deque<int> MGraph<T>::findUndigraphLoop() {
    deque<T> loop, path;
    vector<bool> vis(vex_cnt, false);
    bool find_one_loop = false;

    // find loop
    function<void(int, int)> dfs = [&](int parent, int now) {
        if (find_one_loop) {
            return;
        }
        if (vis[now]) {
            find_one_loop = true;
            while (path.front() != now) {
                path.pop_front();
            }
            loop = path;
            return;
        }
        vis[now] = true;
        path.push_back(now);
        for (int i = 0; i < vex_cnt; i++) {
            if (edges[now][i] != INF && i != parent) {
                dfs(now, i);
            }
        }
        vis[now] = false;
        path.pop_back();
    };

    // check all connected components
    for (int i = 0; i < vex_cnt; i++) {
        if (!find_one_loop && !vis[i]) {
            dfs(-1, i);
        }
    }

    return loop;
}
```

T6

> 对下图所示的连通网，请分别用 $prim$ 算法和 $kruskal$ 算法构造该网的最小生成树。
>
> 
>

- 均使用邻接矩阵进行存储，初始化图时，所有边权均初始化为无穷大 `INF`
- $prim$ 算法时间复杂度 $O(n^2)$
- $kruskal$ 算法时间复杂度 $O(e\log e)$

$prim$ 算法

```cpp
template<class T>
T MGraph<T>::prim(int v) {
    T length = 0;
    vector<T> d(vex_cnt, INF); // d[i] means min edge from i to MST
    vector<bool> MST(vex_cnt, false);
    auto min = [&](T a, T b) {
        return a < b ? a : b;
    };

    /* 1. choose v as the first point */
    MST[v] = true;
    for (int j = 0; j < vex_cnt; j++) {
        if (!MST[j]) {
            d[j] = min(d[j], edges[j][v]);
        }
    }

    /* 2. choose n-1 edges */
    for (int i = 0; i < vex_cnt - 1; i++) {
        // find the shortest edge min_e and its corresponding point vex
```

```
        int vex = -1;
        for (int j = 0; j < vex_cnt; j++) {
            if (!MST[j] && (vex == -1 || d[j] < d[vex])) {
                vex = j;
            }
        }

        // add vex to MST
        MST[vex] = true;

        // add min_e to length
        if (d[vex] == INF) {
            cerr << "unable to generate MST!\n";
            exit(1);
        } else {
            length += d[vex];
        }

        // dp method to update d
        for (int j = 0; j < vex_cnt; j++) {
            if (!MST[j]) {
                d[j] = min(d[j], edges[j][vex]);
            }
        }
    }

    return length;
}
```

kruskal 算法

```
template<class T>
vector<tuple<int, int, T>> MGraph<T>::kruskal() {
    vector<tuple<int, int, T>> res;
    vector<tuple<T, int, int>> edges_set;

    // store all edges
    for (int i = 0; i < vex_cnt; i++) {
        for (int j = 0; j < vex_cnt; j++) {
            if (edges[i][j] != INF) {
                edges_set.push_back({edges[i][j], i, j});
            }
```

```
        }
    }

    /* 1. sort with edge weight */
    sort(edges_set.begin(), edges_set.end());

    class DSU {
    public:
        vector<int> p;

        DSU(int n) {
            p.resize(n);
            for (int i = 0; i < n; i++) {
                p[i] = i;
            }
        }

        int findParent(int now) {
            if (p[now] != now) {
                p[now] = findParent(p[now]);
            }
            return p[now];
        }

        void unionSet(int u, int v) {
            p[findParent(u)] = findParent(v);
        }

        bool sameSet(int u, int v) {
            return findParent(u) == findParent(v);
        }
    };

    /* 2. choose n-1 edges */
    int cnt = 0; // count of edges
    DSU dsu(vex_cnt);
    for (auto [w, u, v]: edges_set) {
        if (!dsu.sameSet(u, v)) {
            res.push_back({u, v, w});
            dsu.unionSet(u, v);
            cnt++;
        }
```

```
54          if (cnt == vex_cnt - 1) {
55              break;
56          }
57      }
58      if (cnt < vex_cnt - 1) {
59          cerr << "unable to generate MST!\n";
60          exit(1);
61      }
62
63      return res;
64  }
```

T7

> 如何寻找一个图的最大生成树，也即总权重最大的生成树。

法一：

- 直接将所有的边权取相反数，然后执行两种最小生成树算法即可

法二：

- 修改 $prim$ 算法，每次选取交叉边从**最小权值交叉边**，改为每次选取**最大权值交叉边**
- 修改 $kruskal$ 算法，将边权按照**升序排序**进行选择，改为将边权按照**降序排序**进行选择

实验八

实验代码：https://github.com/Explorer-Dong/DataStructure/blob/main/Code/chapter8_Graph/Experiment_8.cpp

T1

实验题 8.1 编写程序，建立图的存储结构，并对图进行遍历操作。

基本要求：

(1) 建立一个无向图的邻接矩阵。

(2) 由第 (1) 步建立的邻接矩阵产生相应的邻接表结构。

(3) 分别基于邻接矩阵和邻接表结构对图进行深度优先遍历与广度优先遍历。

T2

实验题 8.2 编写程序，建立图的存储结构，并求出该图的最小生成树。

基本要求：

（1）建立一个无向网的邻接矩阵。

（2）分别用 Prim 算法和 Kruskal 算法构造最小生成树。

（3）以文本形式输出生成树中各条边以及它们的权值。

T3

实验题 8.3 编写程序，建立图的存储结构，求出该图的最短路径。

基本要求：

（1）建立一个有向网的邻接矩阵。

（2）用 Dijkstra 算法计算从某个结点出发的最短路径并输出。

（3）用 Floyd 算法计算该图每对顶点之间的最短路径并输出。

dijkstra

```cpp
template<class T>
vector<int> MGraph<T>::dijkstra(int a, int b) {
    vector<int> res;                        // a->b shortest path
    vector<int> pre(vex_cnt, 0);            // pre[i] means i's previous point
    vector<int> d(vex_cnt, INF);            // d[i] means a to i shortest path length
    vector<bool> SPT(vex_cnt, false);       // SPT[i] means i is in SPT

    // join a to SPT
    d[a] = 0;
    SPT[a] = true;
    pre[a] = -1;
    for (int j = 0; j < vex_cnt; j++) {
        if (!SPT[j] && d[j] > d[a] + edges[a][j]) {
            pre[j] = a;
            d[j] = d[a] + edges[a][j];
        }
    }

    // choose n-1 points
    for (int i = 1; i <= vex_cnt - 1; i++) {
        // 1. choose the shortest edge
        int vex = -1;
```

```
        for (int j = 0; j < vex_cnt; j++) {
            if (!SPT[j] && (vex == -1 || d[j] < d[vex])) {
                vex = j;
            }
        }

        // 2. join vex to SPT
        SPT[vex] = true;

        // 3. update the shortest path from vex to other points
        // record the previous point of the updated point as vex
        for (int j = 0; j < vex_cnt; j++) {
            if (!SPT[j] && d[j] > d[vex] + edges[vex][j]) {
                pre[j] = vex;
                d[j] = d[vex] + edges[vex][j];
            }
        }
    }

    // get path
    while (b != -1) {
        res.push_back(b);
        b = pre[b];
    }
    reverse(res.begin(), res.end());

    return res;
}
```

floyd

```
template<class T>
vector<tuple<int, int, vector<int>>> MGraph<T>::floyd() {
    vector<tuple<int, int, vector<int>>> res;
    // d[i][j] means i to j shortest path length
    vector<vector<int>> d(vex_cnt, vector<int>(vex_cnt, INF));
    // aft[i][j] means i to j first pass vex
    vector<vector<int>> aft(vex_cnt, vector<int>(vex_cnt, -1));

    // init
    for (int i = 0; i < vex_cnt; i++) {
        for (int j = 0; j < vex_cnt; j++) {
```

```
            if (i == j) {
                d[i][j] = 0;
            } else if (edges[i][j] != INF) {
                d[i][j] = edges[i][j];
                aft[i][j] = j;
            }
        }
    }

    // dp
    for (int k = 0; k < vex_cnt; k++) {
        for (int i = 0; i < vex_cnt; i++) {
            for (int j = 0; j < vex_cnt; j++) {
                if (d[i][k] != INF && d[k][j] != INF && d[i][k] + d[k][j] < d[i][j]) {
                    d[i][j] = d[i][k] + d[k][j];
                    aft[i][j] = k; // aft[i][j] = aft[i][k] = k
                }
            }
        }
    }

    // result
    for (int i = 0; i < vex_cnt; i++) {
        for (int j = 0; j < vex_cnt; j++) {
            if (d[i][j] != INF && d[i][j]) {
                vector<int> path;
                path.push_back(i);
                int next = aft[i][j];
                while (next != j) {
                    path.push_back(next);
                    next = aft[next][j];
                }
                path.push_back(j);
                res.push_back({i, j, path});
            }
        }
    }

    return res;
}
```

T4

实验题 8.4 编写程序，建立图的存储结构，对该图进行拓扑排序。

基本要求：

（1）建立一个有向图的邻接表结构。

（2）对该图进行拓扑排序并输出排序结果。