CSP-J 组专属的图论核心知识点 + 模板代码清单

一、核心知识点（仅覆盖CSP-J考点，无超纲内容）

1. 图的基础定义（必考基础）

- 核心概念：无向图、有向图、顶点（结点）、边、度（入度/出度）、路径、环、连通图、连通块；
- CSP-J考情：不直接考定义默写，侧重结合遍历、连通性问题考查理解（如判断图是否有环、统计连通块数量）。

2. 图的存储方式（核心技能，所有算法的基础）

- 邻接表：竞赛首选，适配顶点数多、边数少的稀疏图（CSP-J主流题型），用vector嵌套实现；
- 邻接矩阵：适配顶点数少（≤1000）的稠密图，用二维数组实现，查询边是否存在效率高；
- 备注：边集数组极少考查，无需重点掌握。

3. 图的遍历（必考模块）

- 深度优先搜索（DFS）：递归或栈实现，侧重“纵向探索”，适配连通块计数、路径查找、环检测；
- 广度优先搜索（BFS）：队列实现，侧重“横向探索”，适配最短路径（无权图）、层次遍历问题；
- 考情：单独考遍历较少，多结合其他考点（如拓扑排序、连通性）。

4. 最短路（必考，模板题为主）

- Dijkstra算法（朴素版）：适配无负权边的单源最短路，时间复杂度$O(n^2)$，适合$n \leqslant 1000$的场景（CSP-J重点）；
- Floyd-Warshall算法：多源最短路，三重循环实现，适配$n \leqslant 500$的场景，代码简洁，适合解决多起点最短路径问题；
- 备注：Bellman-Ford、SPFA为S组考点，J组不考，无需掌握。

5. 拓扑排序（高频考点）

- 核心：针对有向无环图（DAG），输出顶点的线性顺序，适配“任务规划”“课程表”类问题；
- 实现方式：BFS版（统计入度，依次删除入度为0的顶点），代码固定，易得分。

6. 并查集（图的连通性，必考）

- 核心功能：快速判断两个顶点是否连通、合并两个连通块，适配连通性统计、 Kruskal算法（J组偶考）；
- 实现要点：路径压缩（优化查询效率）、按秩合并（可选，优化合并效率，CSP-J不考但建议掌握，降低代码耗时）；
- 考情：可单独考（如连通块计数），也可结合其他考点（如最小生成树入门题）。

7. 补充说明（避坑重点）

- CSP-J不考内容：强连通分量、双连通分量、二分图、网络流、负权边/负环相关算法；
- 易错点：邻接表存储有向图时，边的添加方向需注意；DFS递归易栈溢出，可改用栈实现。

二、CSP-J 专属模板代码（可直接复制使用，适配考试题型）

模板1：图的存储（邻接表 + 邻接矩阵）

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

// 1. 邻接表（稀疏图首选，CSP-J主流）
vector<int> adj[10005]; // 顶点编号建议从1开始，避免0下标问题
// 添加无向边（u和v双向连通）
void addEdge_undir(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}
// 添加有向边（u指向v）
void addEdge_dir(int u, int v) {
    adj[u].push_back(v);
}

// 2. 邻接矩阵（稠密图，顶点数n≤1000）
const int MAXN = 1005;
int g[MAXN][MAXN]; // g[u][v] = 1表示有边，0表示无边，可存权值（最短路用）
// 初始化邻接矩阵
void initGraph(int n) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            g[i][j] = 0; // 无边时初始化为0，若存权值可初始化为无穷大
        }
    }
}
// 添加边（u到v，权值为w，无向边需添加两次）
void addEdge_mat(int u, int v, int w = 1) {
```

```
    g[u][v] = w;
    // 无向图需解开注释
    // g[v][u] = w;
}

```

模板2：图的遍历（DFS + BFS）

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

vector<int> adj[10005];
bool vis[10005]; // 标记顶点是否被访问过，避免重复遍历

// 1. DFS（递归版，简洁，适合顶点数少的场景）
void dfs(int u) {
    vis[u] = true; // 标记已访问
    cout << u << " "; // 可选，输出遍历顺序
    // 遍历u的所有邻接顶点
    for (int v : adj[u]) {
        if (!vis[v]) { // 未访问过则递归
            dfs(v);
        }
    }
}

// 2. BFS（队列版，无栈溢出问题，适配所有场景）
void bfs(int start) {
    queue<int> q;
    q.push(start);
    vis[start] = true;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        cout << u << " "; // 可选，输出遍历顺序
        // 遍历u的所有邻接顶点
        for (int v : adj[u]) {
            if (!vis[v]) {
                vis[v] = true;
                q.push(v);
            }
        }
    }
```

```
}

// 补充：统计连通块数量（DFS/BFS均可实现）
int countConnected(int n) {
    int cnt = 0;
    fill(vis, vis + n + 1, false); // 重置访问标记
    for (int i = 1; i <= n; i++) {
        if (!vis[i]) {
            cnt++;
            dfs(i); // 或bfs(i)
        }
    }
    return cnt;
}

int main() {
    int n, m; // n为顶点数，m为边数
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        addEdge_undir(u, v); // 无向边，根据题目调整
    }
    // 示例：遍历+统计连通块
    dfs(1);
    cout << endl;
    fill(vis, vis + n + 1, false); // 重置标记
    bfs(1);
    cout << endl;
    cout << "连通块数量：" << countConnected(n) << endl;
    return 0;
}

```

模板3：最短路（Dijkstra朴素版 + Floyd-Warshall）

```
#include <iostream>
#include <vector>
#include <climits> // 用于INT_MAX
using namespace std;

const int MAXN = 1005;
int g[MAXN][MAXN]; // 邻接矩阵存图（适配Dijkstra朴素版、Floyd）
int dist[MAXN]; // 存储起点到各顶点的最短距离
bool vis[MAXN]; // 标记顶点是否已确定最短距离
```

```
10
11  // 1. Dijkstra朴素版（单源最短路，无负权边）
12  void dijkstra(int n, int start) {
13      // 初始化距离数组：起点为0，其余为无穷大
14      for (int i = 1; i <= n; i++) {
15          dist[i] = INT_MAX;
16      }
17      dist[start] = 0;
18      fill(vis, vis + n + 1, false);
19
20      // 循环n次，每次确定一个顶点的最短距离
21      for (int i = 1; i <= n; i++) {
22          // 找到当前未确定最短距离的顶点中，距离最小的顶点u
23          int u = -1;
24          for (int j = 1; j <= n; j++) {
25              if (!vis[j] && (u == -1 || dist[j] < dist[u])) {
26                  u = j;
27              }
28          }
29          if (u == -1) break; // 无连通顶点，退出
30          vis[u] = true; // 标记u已确定最短距离
31
32          // 松弛操作：更新u的邻接顶点的距离
33          for (int v = 1; v <= n; v++) {
34              if (!vis[v] && g[u][v] != 0 && dist[u] != INT_MAX && dist[u] +
    g[u][v] < dist[v]) {
35                  dist[v] = dist[u] + g[u][v];
36              }
37          }
38      }
39  }
40
41  // 2. Floyd-Warshall（多源最短路，任意两点间最短距离）
42  const int INF = INT_MAX / 2; // 避免加法溢出
43  int floyd_g[MAXN][MAXN];
44  void floyd(int n) {
45      // 初始化距离矩阵
46      for (int i = 1; i <= n; i++) {
47          for (int j = 1; j <= n; j++) {
48              if (i == j) floyd_g[i][j] = 0;
49              else if (floyd_g[i][j] == 0) floyd_g[i][j] = INF;
50          }
51      }
52
53      // 三重循环：枚举中间点、起点、终点
54      for (int k = 1; k <= n; k++) {
55          for (int i = 1; i <= n; i++) {
```

```
            for (int j = 1; j <= n; j++) {
                if (floyd_g[i][k] + floyd_g[k][j] < floyd_g[i][j]) {
                    floyd_g[i][j] = floyd_g[i][k] + floyd_g[k][j];
                }
            }
        }
    }
}

int main() {
    int n, m;
    cin >> n >> m;
    // 初始化邻接矩阵（Dijkstra用）
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            g[i][j] = 0;
        }
    }
    // 初始化Floyd距离矩阵
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            floyd_g[i][j] = 0;
        }
    }

    // 读入边（权值为w，无向边需添加两次）
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        g[u][v] = w;
        g[v][u] = w; // 无向边，有向边注释此行
        floyd_g[u][v] = w;
        floyd_g[v][u] = w; // 无向边，有向边注释此行
    }

    // Dijkstra示例：起点为1，输出到各顶点的最短距离
    dijkstra(n, 1);
    cout << "Dijkstra（起点1）：" << endl;
    for (int i = 1; i <= n; i++) {
        if (dist[i] == INT_MAX) cout << "INF ";
        else cout << dist[i] << " ";
    }
    cout << endl;

    // Floyd示例：输出任意两点间最短距离
    floyd(n);
    cout << "Floyd（任意两点）：" << endl;
```

```
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (floyd_g[i][j] == INF) cout << "INF ";
            else cout << floyd_g[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}

```

模板4：拓扑排序（BFS版，有向无环图）

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

const int MAXN = 10005;
vector<int> adj[MAXN]; // 邻接表存有向图
int inDegree[MAXN]; // 存储每个顶点的入度
vector<int> topo; // 存储拓扑排序结果

// 拓扑排序（返回是否存在合法拓扑序，存在则存入topo数组）
bool topologicalSort(int n) {
    queue<int> q;
    // 入度为0的顶点入队
    for (int i = 1; i <= n; i++) {
        if (inDegree[i] == 0) {
            q.push(i);
        }
    }

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        topo.push_back(u); // 加入拓扑序列

        // 遍历u的邻接顶点，入度减1
        for (int v : adj[u]) {
            inDegree[v]--;
            if (inDegree[v] == 0) { // 入度为0则入队
                q.push(v);
            }
        }
```

```
    }

    // 若拓扑序列长度等于顶点数，说明无环，存在合法拓扑序；否则有环
    return topo.size() == n;
}

int main() {
    int n, m;
    cin >> n >> m;
    // 初始化入度数组
    fill(inDegree, inDegree + n + 1, 0);
    // 读入有向边（u指向v）
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        inDegree[v]++; // v的入度加1
    }

    if (topologicalSort(n)) {
        cout << "合法拓扑序: ";
        for (int x : topo) {
            cout << x << " ";
        }
    } else {
        cout << "图中存在环，无合法拓扑序";
    }
    return 0;
}
```

模板5：并查集（路径压缩 + 按秩合并）

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

const int MAXN = 10005;
int parent[MAXN]; // 存储每个顶点的父节点
int rank_[MAXN]; // 存储树的高度（按秩合并用）

// 初始化并查集
void initUnionFind(int n) {
    for (int i = 1; i <= n; i++) {
        parent[i] = i; // 每个顶点初始父节点是自身
        rank_[i] = 1; // 初始树高为1
```

```
    }
}

// 查找（路径压缩，优化查询效率）
int find(int x) {
    if (parent[x] != x) {
        parent[x] = find(parent[x]); // 递归压缩路径，父节点直接指向根节点
    }
    return parent[x];
}

// 合并（按秩合并，优化合并效率）
void unite(int x, int y) {
    x = find(x);
    y = find(y);
    if (x == y) return; // 已在同一集合，无需合并

    // 矮树合并到高树，避免树过高
    if (rank_[x] < rank_[y]) {
        parent[x] = y;
    } else {
        parent[y] = x;
        if (rank_[x] == rank_[y]) {
            rank_[x]++; // 高度相等时，合并后根节点高度加1
        }
    }
}

// 判断两个顶点是否连通
bool isConnected(int x, int y) {
    return find(x) == find(y);
}

int main() {
    int n, m;
    cin >> n >> m;
    initUnionFind(n);
    for (int i = 0; i < m; i++) {
        int op, x, y;
        cin >> op >> x >> y;
        if (op == 1) { // 操作1：合并x和y
            unite(x, y);
        } else if (op == 2) { // 操作2：判断x和y是否连通
            if (isConnected(x, y)) {
                cout << "是" << endl;
            } else {
                cout << "否" << endl;
```

```
            }
        }
    }
    return 0;
}

```

三、CSP-J 图论备考小贴士

- 模板优先级：邻接表 > DFS/BFS > 并查集 > Dijkstra > 拓扑排序 > Floyd，优先掌握前4个，必考且易得分；
- 刷题建议：每掌握一个模板，刷3-5道J组真题/模拟题（侧重连通块、无权最短路、拓扑排序、并查集基础题）；
- 易错点规避：初始化距离数组时避免溢出（Floyd用INF=INT_MAX/2）、DFS递归层数过多时改用栈实现、邻接表存储有向边时不遗漏方向。

（注：文档部分内容可能由 AI 生成）