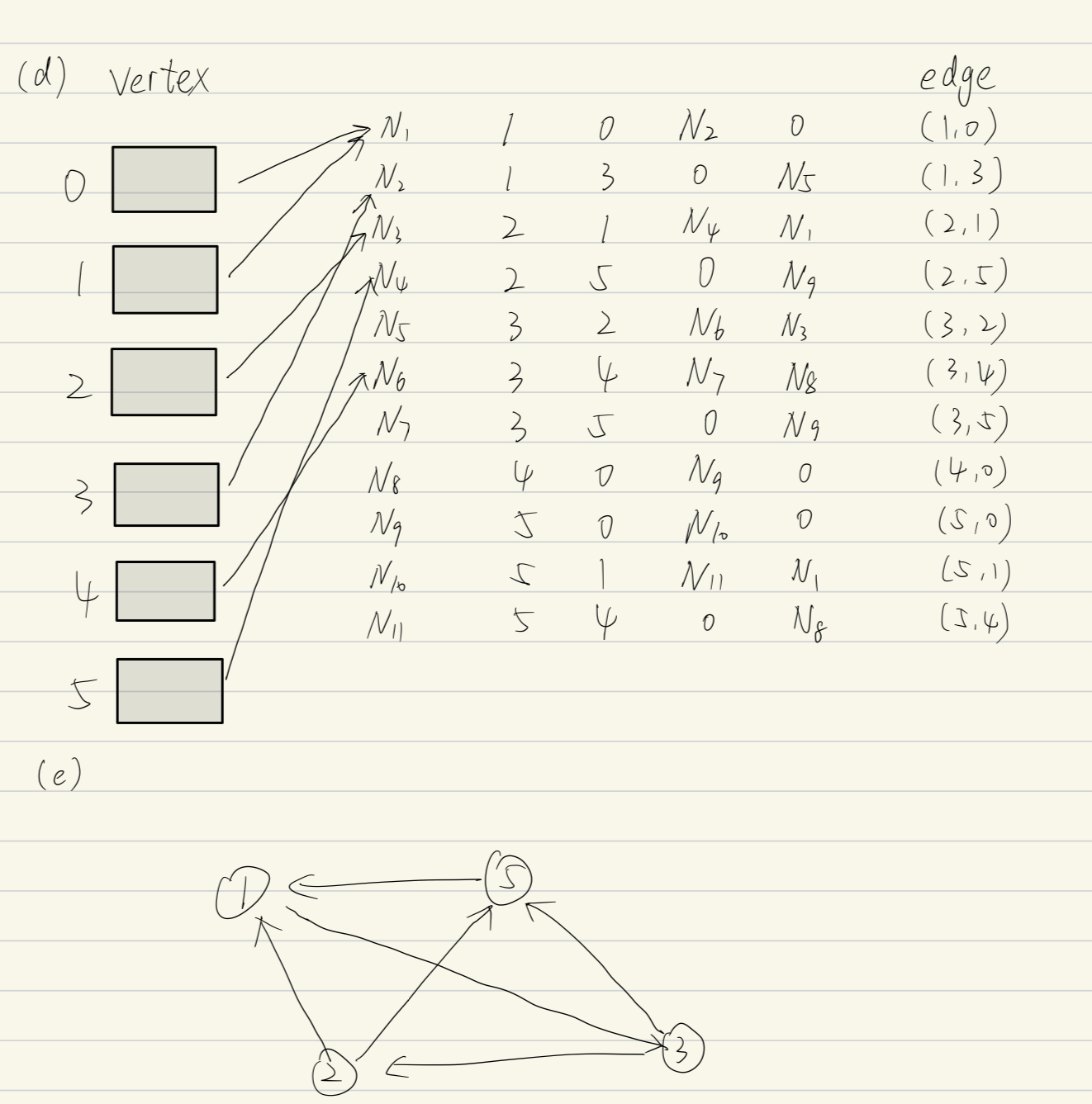
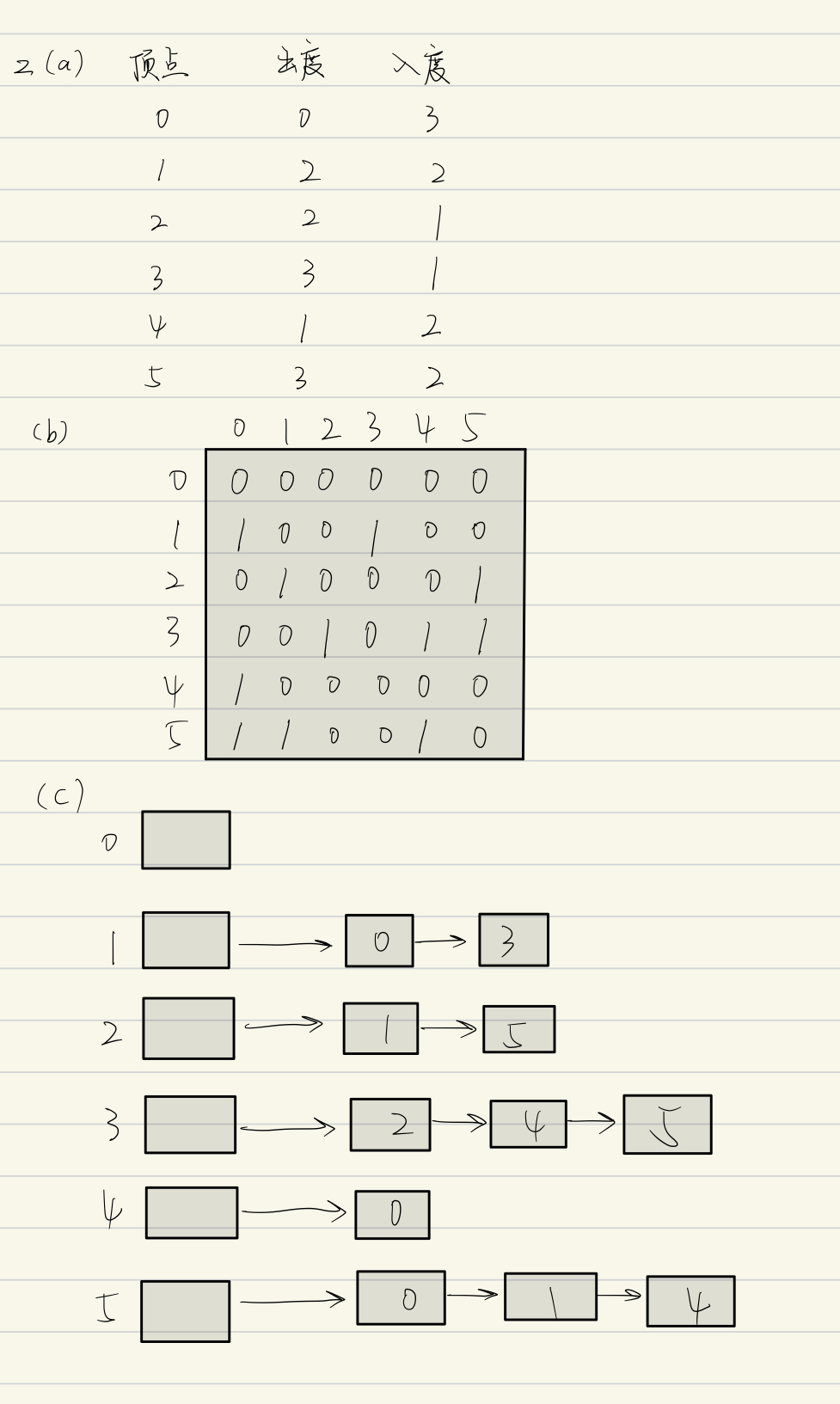
P339 2



P340 8

证明 (a) => (b):

如果G是一棵树，那么它是连通的，因为树的定义中就包括连通性。树中任意两个顶点之间有且只有一条路径，所以如果移除任何一条边，那么至少会有一对顶点之间不再连通，因此移除任意一条边后图不再连通。

证明 (b) => (c):

如果 G是连通的，但移除任意一条边就不再连通，这意味着在G中任意两个顶点之间至少有一条路径（连通性的定义）。由于移除任意一条边都会导致G不连通，这说明在两个顶点之间没有其他路径，否则图仍然会连通。因此，这两个顶点之间的路径是唯一的。

证明 (c) => (d):

如果G中任意两个顶点之间有唯一的简单路径，那么图中不能有环，因为如果有环，那么会有至少两条不同的路径连接环上的任意两点。所以G中没有环。而由于图是连通的，并且任意两点间只有一条路径，这就意味着每增加一个顶点就必须增加一条边来保持图的连通性，因此G有n-1条边。

证明 (d) => (a):

如果G中没有环并且有G条边，我们可以通过归纳法来证明G是连通的。首先，一个顶点的图自然是连通的。假设在k<n个顶点时，图是连通的。当我们添加第k+1个顶点和一条边时（由于没有环，这条新边必须连接到之前的某个顶点），新图仍然是连通的。因此，G是连通的，且由于没有环，所以 G是一棵树。

因此，这四个命题是等价的。

P372 2

#include <iostream>

#include <vector>

#include <queue>

// 定义树节点

struct TreeNode {

int val;

std::vector<TreeNode\*> children; // 假设是多叉树

TreeNode(int x) : val(x) {}

};

// BFS 函数来计算最短路径

std::vector<int> findShortestPaths(TreeNode\* root) {

std::vector<int> shortestPaths;

if (root == nullptr) return shortestPaths;

std::queue<TreeNode\*> q;

q.push(root);

// 根节点到自己的距离是0

shortestPaths.push\_back(0);

while (!q.empty()) {

TreeNode\* currentNode = q.front();

q.pop();

int currentPath = shortestPaths[currentNode->val];

for (TreeNode\* child : currentNode->children) {

q.push(child);

// 子节点的距离是当前节点距离加上边的长度，这里假设边的长度为1

shortestPaths.push\_back(currentPath + 1);

}

}

return shortestPaths;

}

int main() {

// 创建树的例子

TreeNode\* root = new TreeNode(0); // 根节点

root->children.push\_back(new TreeNode(1)); // 第一层子节点

root->children.push\_back(new TreeNode(2));

root->children[0]->children.push\_back(new TreeNode(3)); // 第二层子节点

root->children[0]->children.push\_back(new TreeNode(4));

root->children[1]->children.push\_back(new TreeNode(5));

std::vector<int> shortestPaths = findShortestPaths(root);

// 打印结果

for (int i = 0; i < shortestPaths.size(); ++i) {

std::cout << "Shortest path to node " << i << " is " << shortestPaths[i] << std::endl;

}

return 0;

}

P373 5

#include <iostream>

#include <vector>

#include <climits>

// 假定 MAXN 是图中最大的顶点数

const int MAXN = 100;

// 用 INT\_MAX 表示两个顶点之间没有边

const int INF = INT\_MAX;

// 邻接矩阵，存储图

int graph[MAXN][MAXN];

// Prim 算法实现最小生成树

// n 是顶点数，graph 是图的邻接矩阵

void prim(int n) {

// 用于存储最小生成树的边权值

int minWeight[MAXN];

// 用于标记顶点是否被包含在最小生成树中

bool inMST[MAXN];

// 用于存储最小生成树的父节点信息

int parent[MAXN];

// 初始化所有键值为无穷大，而且没有顶点被包含在 MST 中

for (int i = 0; i < n; ++i) {

minWeight[i] = INF;

inMST[i] = false;

}

// 从第一个顶点开始构造 MST

minWeight[0] = 0;

parent[0] = -1; // 第一个顶点是 MST 的根，设置父节点为 -1

// MST 会有 n 个顶点

for (int count = 0; count < n - 1; ++count) {

// 选择一个不在 MST 中，并且键值最小的顶点

int u = -1;

int min = INF;

for (int v = 0; v < n; ++v) {

if (!inMST[v] && minWeight[v] < min) {

min = minWeight[v];

u = v;

}

}

// 将这个顶点添加到 MST 中

inMST[u] = true;

// 更新相邻顶点和边的权值

for (int v = 0; v < n; ++v) {

// graph[u][v] 是非零的只有在顶点 u 和 v 之间有边，

// 而且当顶点 v 不在 MST 中，更新 v 的键值

if (graph[u][v] && !inMST[v] && graph[u][v] < minWeight[v]) {

parent[v] = u;

minWeight[v] = graph[u][v];

}

}

}

// 打印构建的 MST

std::cout << "Edge \tWeight\n";

for (int i = 1; i < n; ++i)

std::cout << parent[i] << " - " << i << " \t" << graph[i][parent[i]] << " \n";

}

int main() {

int n; // 图中的顶点数

// 假设 graph 是通过某种方式填充好的

// 调用 prim 函数

prim(n);

return 0;

}

P389 2

(a)

|  |  |  |
| --- | --- | --- |
| 活动 | 最早开始时间 | 最晚开始时间 |
| 0-1 | 0 | 4 |
| 0-2 | 0 | 0 |
| 1-3 | 5 | 9 |
| 2-3 | 6 | 6 |
| 2-4 | 6 | 12 |
| 3-4 | 12 | 12 |
| 3-5 | 12 | 15 |
| 3-6 | 12 | 12 |
| 4-6 | 15 | 15 |
| 4-7 | 15 | 15 |
| 5-9 | 16 | 19 |
| 6-8 | 16 | 16 |
| 7-8 | 19 | 19 |
| 8-9 | 21 | 21 |

(b)

最早可以在23天完成‘

(c)

关键活动为0-2 2-3 3-4 3-6 4-6 4-7 6-8 7-8 8-9

(d)

加速0-2 2-3 8-9 可以缩短整个工期