# 计算题：

## 24.1-1

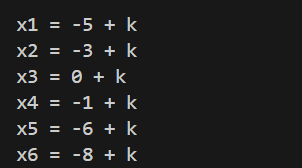
初始化z=0 s t x y=

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号/结点及前驱 | s | t | x | y |
| 1 | 2/z |  | 7/z |  |
| 2 | 2/z | 5/x | 7/z | 9/s |
| 3 | 2/z | 5/x | 6/y | 9/s |
| 4 | 2/z | 4/x | 6/y | 9/s |

将z-x改成4后，s出发，初始化s为0,其余为

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号/结点及前驱 | t | x | y | z |
| 1 | 6/s |  | 7/s |  |
| 2 | 6/s | 4/y | 7/s | 2/t |
| 3 | 2/x | 4/y | 7/s | 2/t |
| 4 | 2/x | 4/y | 7/s | -2/t |

## **24.4-1**

程序运行结果为  


#include <iostream>

#include <vector>

#include <limits>

using namespace std;

const int INF = numeric\_limits<int>::max(); // Define a large constant for infinity

// Structure to represent an edge in the graph

struct Edge

{

int from, to, weight; // 'from' node, 'to' node, and edge weight

};

// Bellman-Ford algorithm to find shortest paths from a source node

bool bellman\_ford(int n, vector<Edge>& edges, vector<int>& dist)

{

dist.assign(n + 1, INF); // Initialize distances to all nodes as infinity

dist[0] = 0; // Distance to the source node (node 0) is 0

// Relax all edges 'n' times to compute shortest paths

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

{

for (auto& e : edges)

{

// If the distance to the 'from' node is finite, attempt to relax the edge

if (dist[e.from] != INF && dist[e.to] > dist[e.from] + e.weight)

{

dist[e.to] = dist[e.from] + e.weight; // Update the distance to the 'to' node

}

}

}

// Check for negative-weight cycles by trying to relax once more

for (auto& e : edges)

{

if (dist[e.from] != INF && dist[e.to] > dist[e.from] + e.weight)

{

return false; // If a relaxation still occurs, a negative cycle exists

}

}

return true; // No negative-weight cycles detected

}

int main()

{

int n = 6; // Number of variables x1 to x6

vector<Edge> edges; // Vector to store edges representing constraints

// Correctly add constraints as directed edges in the graph

edges.push\_back({ 2, 1, 1 }); // x1 - x2 <= 1 -> x2 -> x1 with weight 1

edges.push\_back({ 4, 1, -4 }); // x1 - x4 <= -4 -> x4 -> x1 with weight -4

edges.push\_back({ 3, 2, 2 }); // x2 - x3 <= 2 -> x3 -> x2 with weight 2

edges.push\_back({ 5, 2, 7 }); // x2 - x5 <= 7 -> x5 -> x2 with weight 7

edges.push\_back({ 6, 2, 5 }); // x2 - x6 <= 5 -> x6 -> x2 with weight 5

edges.push\_back({ 6, 3, 10 }); // x3 - x6 <= 10 -> x6 -> x3 with weight 10

edges.push\_back({ 2, 4, 2 }); // x4 - x2 <= 2 -> x2 -> x4 with weight 2

edges.push\_back({ 1, 5, -1 }); // x5 - x1 <= -1 -> x1 -> x5 with weight -1

edges.push\_back({ 4, 5, 3 }); // x5 - x4 <= 3 -> x4 -> x5 with weight 3

edges.push\_back({ 3, 6, -8 }); // x6 - x3 <= -8 -> x3 -> x6 with weight -8

// Add an extra source node (node 0) with edges of weight 0 to all other nodes

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

{

edges.push\_back({ 0, i, 0 }); // Edge from source (node 0) to node i

}

vector<int> dist; // Vector to store shortest distances from the source node

// Run the Bellman-Ford algorithm

if (bellman\_ford(n, edges, dist))

{

// If successful, output the feasible solution

cout << "Feasible solution found (up to an arbitrary constant k):\n";

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

{

cout << "x" << i << " = " << dist[i] << " + k\n"; // Display solution with constant offset k

}

cout << "Where k is any constant.\n";

}

else

{

// If a negative cycle is detected, no feasible solution exists

cout << "No feasible solution exists (negative cycle detected).\n";

}

return 0; // End of program

}

## **25.2-1**

### K=1

0 INF INF INF -1 INF

1 0 2 0 INF INF

INF 2 0 INF INF -8

-4 INF INF 0 -5 INF

INF 7 0 INF 0 0

INF 5 10 INF 0 0

K=2  
0 INF INF INF -1 INF

1 0 2 0 2 INF

3 2 0 4 2 -8

-4 INF INF 0 -5 INF

8 7 9 0 0 0

6 5 10 7 5 0

### K=3

0 INF INF INF -1 INF

1 0 2 0 2 INF

3 2 0 4 2 -8

-4 INF INF 0 -5 INF

8 7 9 0 0 0

6 5 10 7 5 0

### K=4

0 INF INF -1 INF INF

-2 0 2 0 -3 INF

0 2 0 4 -1 -8

-4 INF INF 0 -5 INF

5 7 9 0 0 0

3 5 10 7 2 0

### K=5

0 6 INF INF -1 INF

-2 0 2 0 -3 INF

0 2 0 4 -1 -8

-4 2 INF 0 -5 INF

5 7 9 0 0 0

3 5 10 7 2 0

### K=6

0 6 INF 8 -1 INF

-2 0 2 0 -3 INF

-5 -3 0 -1 -6 -8

-4 2 INF 0 -5 INF

5 7 9 0 0 0

3 5 10 7 2 0

# 设计、证明题：

## **22.2-7**

构建二分图，通过BFS搜索判断能否分为两组，简要代码如下  
  
#include <iostream>

#include <vector>

#include <queue>

using namespace std;

const int MAX\_VERTICES = 100; // 假设最大顶点数

vector<int> adj[MAX\_VERTICES]; // 邻接表

int color[MAX\_VERTICES]; // 顶点颜色，-1 表示未着色

void addEdge(int u, int v) {

adj[u].push\_back(v);

adj[v].push\_back(u); // 无向图

}

bool bfs(int start) {

queue<int> q;

q.push(start);

color[start] = 0; // 从0开始交替着色

while (!q.empty()) {

int u = q.front();

q.pop();

for (int v : adj[u]) {

if (color[v] == -1) { // 如果v未着色

color[v] = 1 - color[u]; // 交替着色

q.push(v);

}

else if (color[v] == color[u]) { // 如果相邻顶点颜色相同

return false; // 不是二分图

}

}

}

return true;

}

bool isBipartite() {

for (int i = 0; i < MAX\_VERTICES; ++i) {

color[i] = -1; // 初始化颜色

}

for (int i = 0; i < MAX\_VERTICES; ++i) {

if (color[i] == -1) { // 如果顶点未访问

if (!bfs(i)) {

return false; // 如果发现不是二分图

}

}

}

return true;

}

int main() {

// 假设输入是顶点数和边数

int n, m;

cin >> n >> m;

// 读取边并构建图

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

int u, v;

cin >> u >> v;

addEdge(u, v);

}

// 检查是否为二分图并输出结果

if (isBipartite()) {

cout << "The wrestlers can be divided into babyfaces and heels." << endl;

// 输出每个顶点的颜色，0 表示 babyfaces，1 表示 heels

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

cout << "Wrestler " << i << " is a " << (color[i] == 0 ? "babyface" : "heel") << endl;

}

}

else {

cout << "It is not possible to divide the wrestlers into babyfaces and heels." << endl;

}

return 0;

}

## **24.1-3**

在Bellman-Ford算法中，每次迭代尝试松弛所有边。如果某个节点的最短路径值 d[v]在一次迭代中没有变化，那么之后的迭代中 d[v]d[v]d[v] 也不会再变化。那么，当一次完整的迭代中没有任何距离 d 被更新时，说明所有最短路径都已经收敛，算法可以提前终止。通过引入 “isUpdate” 变量记录当前迭代是否有更新，我们可以让 Bellman-Ford 算法在最短路径收敛时提前终止，从而保证在 m+1次迭代内结束，即使事先不知道 m。

bool bellman\_ford(int n, int source, const vector<Edge>& edges, vector<int>& dist) {

dist.assign(n, INF); // 初始化距离数组

dist[source] = 0; // 源点到自身的距离为0

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

bool updated = false; // 用于检查本轮是否有距离更新

for (const Edge& e : edges) {

if (dist[e.u] != INF && dist[e.v] > dist[e.u] + e.weight) {

dist[e.v] = dist[e.u] + e.weight;

updated = true; // 发生了更新

}

}

if (!updated) {

// 如果本轮没有任何更新，提前结束

cout << "提前终止于第 " << i + 1 << " 次迭代\n";

break;

}

}

return true; // 无负权环

}

## **24-3**

通过将汇率转换为负对数边权重，问题转化为负权环检测问题。使用 Bellman-Ford 算法检测负权环，并输出对应的套利序列。时间复杂度为O(VE)

伪代码: Bellman - Ford 算法检测负权环

输入 :

n - 顶点数量

start - 源点

edges - 边的列表，每个边包含起点(u)，终点(v)和权重(weight)

parent - 用于记录每个顶点的前驱节点

dist - 用于记录从源点到每个顶点的最短距离

初始化 :

对于所有顶点 v：

dist[v] = 无穷大

parent[v] = -1

dist[start] = 0

对于 i 从 1 到 n - 1:

x = -1

对于每条边 e ∈ edges :

如果 dist[e.u] + e.weight < dist[e.v] :

dist[e.v] = dist[e.u] + e.weight

parent[e.v] = e.u

x = e.v

如果 x = -1 :

返回 false (没有负权环)

寻找负权环 :

设置 x 为负权环中的某个顶点

对于 i 从 1 到 n :

x = parent[x]

初始化空列表 cycle

对于 v = x; true; v = parent[v]:

将 v 添加到 cycle 列表

如果 v = x 且 cycle 的长度 > 1:

停止循环

反转 cycle 列表

输出 "套利序列发现: " + cycle 列表

返回 true (发现负权环)

## **25-1**

### (a)

遍历所有顶点对，检测能否通过新的边构成一条新路径

void updateTransitiveClosure(vector<vector<bool>>& closure, int x1, int x2, int V) {

for (int u = 0; u < V; ++u) {

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

// 如果通过新边 (x1, x2)，可以连通 (u, v)，则更新闭包

if (closure[u][x1] && closure[x2][v]) {

closure[u][v] = true;

}

}

}

}

### (b)

构造一个完全二分图，两个集合的大小均为V/2，在这种情况下，任何新边的加入都会引起整个传递闭包的计算，需要检查所有组合

### (c)

每次加入一条新边u-v,需要将u的所有前驱x与v的所有后继y相连，即x-y，单词操作的时间复杂度上限为V^2,由于当每个元素只更新一次，求和后的时间复杂度为O(V^3)

#include <iostream>

#include <unordered\_map>

#include <unordered\_set>

#include <vector>

using namespace std;

// 定义全局树结构：用于维护祖先树和继承者树

unordered\_map<int, unordered\_set<int>> ancestorsTree; // ancestorsTree[u]: u 的所有祖先

unordered\_map<int, unordered\_set<int>> successorsTree; // successorsTree[v]: v 的所有后继

// 辅助函数：将 v 添加到 u 的祖先的后继集合中，递归更新

void AddToAncestorsSuccessors(int u, int v) {

// 如果 u 没有祖先，则返回（递归终止条件）

if (ancestorsTree[u].empty()) return;

// 遍历 u 的所有祖先

for (int ancestor : ancestorsTree[u]) {

// 如果 (ancestor, v) 还未记录在 successorsTree 中，进行更新

if (successorsTree[ancestor].find(v) == successorsTree[ancestor].end()) {

successorsTree[ancestor].insert(v); // 添加 v 为 ancestor 的后继

AddToAncestorsSuccessors(ancestor, v); // 递归更新祖先的后继关系

}

}

}

// 辅助函数：将 u 添加到 v 的后继的祖先集合中，递归更新

void AddToSuccessorsAncestors(int u, int v) {

// 如果 v 没有后继，则返回（递归终止条件）

if (successorsTree[v].empty()) return;

// 遍历 v 的所有后继

for (int successor : successorsTree[v]) {

// 如果 (u, successor) 还未记录在 ancestorsTree 中，进行更新

if (ancestorsTree[u].find(successor) == ancestorsTree[u].end()) {

ancestorsTree[u].insert(successor); // 添加 successor 为 u 的后继

AddToSuccessorsAncestors(u, successor); // 递归更新后继的祖先关系

}

}

}

// 更新传递闭包函数：当插入边 (u, v) 时，更新祖先树和继承者树

void UpdateTransitiveClosure(int u, int v) {

// 调用辅助函数，递归更新祖先和后继关系

AddToAncestorsSuccessors(u, v); // 更新祖先的后继集合

AddToSuccessorsAncestors(u, v); // 更新后继的祖先集合

// 将新边 (u, v) 添加到 successorsTree 和 ancestorsTree 中

if (successorsTree[u].find(v) == successorsTree[u].end()) {

successorsTree[u].insert(v);

}

if (ancestorsTree[v].find(u) == ancestorsTree[v].end()) {

ancestorsTree[v].insert(u);

}

}

// 主函数：测试代码

int main() {

int n, m; // n 表示节点数，m 表示边数

cout << "请输入顶点数和边数: ";

cin >> n >> m;

// 初始化树结构

ancestorsTree.clear();

successorsTree.clear();

cout << "请输入每条边 (u, v):" << endl;

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

int u, v;

cin >> u >> v; // 读取边 (u, v)

UpdateTransitiveClosure(u, v); // 更新传递闭包

}

// 输出继承者树 (Successors)

cout << "继承者树 (Successors):" << endl;

for (const auto& pair : successorsTree) {

cout << "节点 " << pair.first << " 的后继: ";

for (int node : pair.second) {

cout << node << " ";

}

cout << endl;

}

// 输出祖先树 (Ancestors)

cout << "祖先树 (Ancestors):" << endl;

for (const auto& pair : ancestorsTree) {

cout << "节点 " << pair.first << " 的祖先: ";

for (int node : pair.second) {

cout << node << " ";

}

cout << endl;

}

return 0;

}

思考题：

## 25.2-6

负权环会导致顶点到自身的最短路径权重持续减少，最终变为负数,所以对角线上元素若存在小于0的，则存在