山东大学 计算机科学与技术 学院

数据结构与算法课程设计 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202300130207 | 姓名：李臣思 | | 班级：23.5 |
| 实验题目：网络放大器设置问题 | | | |
| 实验学时：6 | | 实验日期：2025.4.30 | |
| 实验目的：针对网络设计问题考虑使用两种方法解决，并比较两种方法的时间性能，用图表显示比较结果. | | | |
| 软件开发工具：Visual Studio Code | | | |
| 1. 实验内容   **问题描述：**  一个汽油传送网络可由**加权有向无环图G**表示。图中有一个称为源点的**顶点S**。从S出发，汽油被输送到图中的其他顶点。**S的入度为0**，每一条边上的权给出了它所连接的**两点间的距离**。通过网络输送汽油时，**压力的损失是所走距离的函数**。为了保证网络的正常运转，在网络传输中必须保证**最小压力Pmin**。为了维持这个最小压力，可将**压力放大器**放在网络中的一些或全部顶点。压力放大器可将**压力恢复至最大**可允许的量级**Pmax**。令**d**为汽油在压力由**Pmax降为Pmin时所走的距离**。在设置信号放大器问题中，需要放置**最少数量的放大器**，以便在遇到一个放大器之前汽油所走的距离不超过d。编写一个程序来求解该问题。  **基本要求：**  针对网络设计问题考虑使用**两种方法解决**，并比较两种方法的时间性能；**用图表显示比较结果**。   1. 数据结构与算法描述（整体思路描述，所需要的数据结构与算法）   分为动态规划以及优先队列BFS两种。   1. 对于**动态规划法**：   **数据结构**：   * 逆邻接表 reverse\_adj[]：存储每个节点的前驱边，用于递推。 * 拓扑排序数组 topo[]：确定节点处理顺序 * 每个节点u维护三个状态结构体State： * setcount：在u放置放大器时的放大器放置的最小数量。 * nosetcount：不在u放置放大器时的放大器的最小数量。 * remainvalue：不放置放大器时的剩余可用压强。 * 同时在动态规划中，构建逆邻接表图，每个节点ReverseEdge包含： * from：前驱节点索引。 * weight：边权。 * next：指向下一个节点   **整体思路：**   * 基于拓扑排序处理节点，从源节点出发，依据拓扑排序递推每个节点的   状态。   * 对于每个节点分别维护各自的setcount、nosetcount以及remainvalue。 * dynamic(int n, int s, int d, ReverseEdge\* adj[], int topo[])：   其中n为节点个数，s为源节点索引，d为最大的压力距离，adj为前期准备好的逆邻接表图，topo为拓扑排序后得到的顺序。  初始化每个State节点，将源节点setcount、nosetcount、remainvalue分别为1、0、d。然后根据拓扑排序依次处理每个节点，针对每个节点首先计算若在u放置，更新setcount，取前驱节点的最优解加1；若不在u放置，则须保证前驱节点到u的路径不超过剩余距离，分为前驱节点放置以及不放置两种情况，分别计算得到最优解，更新此时的nosetcount以及remainvalue。  最后得到的每个节点中setcount以及nosetcount的较小的一方中的最大者即为动态规划后得到的放大器个数。   1. 对于优先队列**bfs**：   **数据结构：**   * 邻接表adj[]:存储正向邻接链表。其中每个节点包含Edge： * to：后继节点的索引。 * weight：边权。 * next：指向下一个节点。 * 优先队列中每个节点状态PQState： * u:当前节点索引。 * remian:剩余距离 * count：放大器数量。 * 重载符号<:按放大器数量升序，剩余距离降序排序。   **整体思路：**   * 使用优先队列优化搜索，优先许选择放大器数量少的且剩余距离大的路径。 * 每个节点状态包含：放大器数量count以及剩余距离remaining。 * 剪枝策略：当当前状态比已记录的状态更差时，则直接跳过。 * 优先队列中按放大器数量升序，剩余距离降序排序。 * bfs(int n, int s, int d, Edge\* adj[])：   从源节点s出发，初始状态为放大器数量0，剩余距离d。对于每个节点分为两种情况处理，不放置放大器时，若剩余距离足够，则继承当前状态；放置放放大器时，放大器数量加一，重置剩余距离。最终结果为所有节点中的放大器数量的最大值。   1. 测试结果（测试输入，测试输出）   输入输出结果：     1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   对于测试用例分析可以得到，结果应为放置一个放大器。  **动态规划**分析：   * 源点（节点0）：   放置放大器：setcount=1  不放置：nosetcount=0，剩余距离 d=10。   * 节点1：   放置：setcount=1（前驱最优解 0 + 1）。  不放置：继承源点剩余距离 10 - 3 = 7，nosetcount=0。   * 节点3：   放置：setcount=1（前驱节点1的最优解 0 + 1）。  不放置：前驱节点1不放置时剩余 7 - 5 = 2，nosetcount=0。   * 节点4：   放置：setcount=1（前驱节点3的最优解 0 + 1）。  不放置：前驱节点3不放置时剩余 2 - 6 无效，必须放置，nosetcount=1。  所以应该为1个放大器。  **bfs**分析：   * 从源点出发（放大器数0，剩余距离10）：   到节点1：剩余7，不放置。  到节点2：剩余8，不放置。   * 从节点1出发：   到节点3：剩余2，不放置。  到节点4：必须放置放大器（剩余7→10-8=2），放大器数1。   * 从节点3出发：   到节点4：必须放置放大器（剩余2→10-6=4），放大器数1。  所以放大器数量为1.  **时间性能**差异：  动态规划：时间复杂度为 O(n+m)，直接按拓扑顺序递推，效率高。  BFS：使用优先队列维护状态，复杂度约为 O(nlogn)，但因剪枝策略优化，实际耗时可能更低（如测试中BFS时间11μs）。   1. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）   #include <iostream>  #include <climits>  #include <queue>  #include <stack>  #include <algorithm>  #include <chrono>  using namespace std;  using namespace std::chrono;  const int MAX\_N = 10000;  // 动态规划法结构定义  struct ReverseEdge {  int from;//前驱节点  int weight;//边权  ReverseEdge\* next;//下一个节点指针  };  struct State {//动态规划状态  int setcount;//放置放大器时的最小数量  int nosetcount;//不放置时的最小数量  int remainvalue;//不放置时的剩余可用距离  };  // 优先队列BFS结构定义  struct Edge {  int to;//后继节点  int weight;  Edge\* next;  };  struct NodeState {  int count;//放大器数量  int remaining;//剩余可用距离  };  struct PQState {//优先队列中的状态  int u;//当前节点  int remain;  int count;//放大器数量  bool operator<(const PQState& other) const {//优先队列排序  if(count != other.count) return count > other.count;//数量少的优先  return remain < other.remain;//剩余距离大的优先  }  };  //动态规划  int dynamic(int n, int s, int d, ReverseEdge\* adj[], int topo[]) {  State states[MAX\_N];  for(int i=0; i<n; ++i) {//初始化  states[i].setcount = INT\_MAX;  states[i].nosetcount = INT\_MAX;  states[i].remainvalue = 0;  }  //源点初始化  states[s].setcount = 1;//假设源点必须放置  states[s].nosetcount = 0;//特殊处理源点不放置的情况  states[s].remainvalue = d;  for(int i=0; i<n; ++i) {//拓扑排序处理每个节点  int u = topo[i];  if(u == s) continue;//源点已经处理了  //计算放置放大器时的最优值  int min1 = INT\_MAX;  for(ReverseEdge\* e = adj[u]; e != nullptr; e = e->next) {  int v = e->from;//前驱节点  int current = min(states[v].setcount, states[v].nosetcount);//前驱节点v的最优选择  min1 = min(min1, current);  }  if(min1 != INT\_MAX) states[u].setcount = min1 + 1;  //计算不放置放大器时的最优值  int best\_count = INT\_MAX, best\_remain = 0;  for(ReverseEdge\* e = adj[u]; e != nullptr; e = e->next) {  int v = e->from;  int w = e->weight;  //前驱放置放大器的情况  if(states[v].setcount != INT\_MAX) {  int rem = d - w;//从v放置后，到u的剩余距离  if(rem >= 0 && states[v].setcount < best\_count) {  best\_count = states[v].setcount;  best\_remain = rem;  }  }  //前驱不放置放大器的情况  if(states[v].nosetcount != INT\_MAX && states[v].remainvalue >= w) {  int rem = states[v].remainvalue - w;  if(states[v].nosetcount < best\_count) {  best\_count = states[v].nosetcount;  best\_remain = rem;  }  }  }  if(best\_count != INT\_MAX) {//更新不放置的状态  states[u].nosetcount = best\_count;  states[u].remainvalue = best\_remain;  }  if(states[u].setcount == INT\_MAX && states[u].nosetcount == INT\_MAX)  return -1;  }  // 最终结果取所有节点的最大值  int result = 0;  for(int i=0; i<n; ++i) {  int current = min(states[i].setcount, states[i].nosetcount);  if(current == INT\_MAX) return -1;  result = max(result, current);  }  return result;  }  //优先队列BFS  int bfs(int n, int s, int d, Edge\* adj[]) {  NodeState node\_states[MAX\_N];  for(int i=0; i<n; ++i) {  node\_states[i].count = INT\_MAX;  node\_states[i].remaining = 0;  }  priority\_queue<PQState> pq;  pq.push({s, d, 0});  while(!pq.empty()) {  auto [u, r, c] = pq.top();  pq.pop();  //剪枝：已有更优状态  if(c > node\_states[u].count || (c == node\_states[u].count && r <= node\_states[u].remaining))  continue;  node\_states[u].count = c;  node\_states[u].remaining = r;  for(Edge\* e = adj[u]; e != nullptr; e = e->next) {  int v = e->to;  int w = e->weight;  // 不放置放大器的情况  if(r >= w) {  int new\_r = r - w;  if(c < node\_states[v].count || (c == node\_states[v].count && new\_r > node\_states[v].remaining)) {  pq.push({v, new\_r, c});  }  }  // 放置放大器的情况  int new\_r = d - w;  int new\_c = c + 1;  if(new\_c < node\_states[v].count || (new\_c == node\_states[v].count && new\_r > node\_states[v].remaining)) {  pq.push({v, new\_r, new\_c});  }  }  }  // 取所有节点的最大值  int result = 0;  for(int i=0; i<n; ++i) {  if(node\_states[i].count == INT\_MAX) return -1;  result = max(result, node\_states[i].count);  }  return result;  }  //拓扑排序  void toposort(int n, Edge\* adj[], int in\_degree[], int topo[]) {  queue<int> q;  for(int i=0; i<n; ++i)  if(in\_degree[i] == 0)//入度为0的节点入队  q.push(i);  int idx = 0;  while(!q.empty()) {  int u = q.front();  q.pop();  topo[idx++] = u;//记录拓扑排序  //遍历所有出边，更新后继节点的入度  for(Edge\* e = adj[u]; e != nullptr; e = e->next) {  int v = e->to;  if(--in\_degree[v] == 0)//入度减一  q.push(v);  }  }  }  int main() {  int n, m, d, s;  cin >> n >> m >> d >> s;//n个节点，m条边，pmax-pmin=10，源节点s  s--; // 转换为0-based索引  Edge\* adj[MAX\_N] = {nullptr};//邻接表  ReverseEdge\* reverse\_adj[MAX\_N] = {nullptr};//逆邻接表  int in\_degree[MAX\_N] = {0};//入度数组  //读取输入并建图  bool possible = true;  for(int i=0; i<m; ++i) {  int u, v, w;  cin >> u >> v >> w;//u->v,边权w  u--;  v--;//转换为0-based索引    if(w > d) possible = false;//存在不可达边  //正向图  Edge\* e = new Edge{v, w, adj[u]};  adj[u] = e;  in\_degree[v]++;//更新入度  //反向图  ReverseEdge\* re = new ReverseEdge{u, w, reverse\_adj[v]};  reverse\_adj[v] = re;  }  if(!possible) {  cout << "-1\n";  return 0;  }  //得到拓扑排序  int topo[MAX\_N];  toposort(n, adj, in\_degree, topo);  //运行动态规划  auto start = high\_resolution\_clock::now();//计时  int dp\_result = dynamic(n, s, d, reverse\_adj, topo);  auto stop = high\_resolution\_clock::now();  auto dp\_duration = duration\_cast<microseconds>(stop - start);  //运行BFS  start = high\_resolution\_clock::now();  int bfs\_result = bfs(n, s, d, adj);  stop = high\_resolution\_clock::now();  auto bfs\_duration = duration\_cast<microseconds>(stop - start);  //输出结果  cout << "动态规划结果: " << dp\_result << "\n";  cout << "BFS结果: " << bfs\_result << "\n";  cout << "动态规划的时间: " << dp\_duration.count() << "μs\n";  cout << "BFS的时间: " << bfs\_duration.count() << "μs\n";  //释放内存  for(int i=0; i<n; ++i) {  while(adj[i]) {  Edge\* temp = adj[i];  adj[i] = adj[i]->next;  delete temp;  }  while(reverse\_adj[i]) {  ReverseEdge\* temp = reverse\_adj[i];  reverse\_adj[i] = reverse\_adj[i]->next;  delete temp;  }  }  return 0;  } | | | |