**算法分析与设计实验报告**

**第 3 次实验**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 姓名 | 邹林壮 | 学号 | 202208040412 | | 班级 | 计算机科学与技术（拔尖班）2201 |
| 时间 | 2024.11.23 | 地点 | 院楼432 | | | |
| 实验名称 | 用 Dijkstra 贪心算法求解单源最短路径问题 | | | | | |
| 实验目的 | 通过本实验，深入理解和掌握 Dijkstra 贪心算法的问题描述、算法设计思想、程序设计，特别是如何通过 Dijkstra 算法解决单源最短路径问题。同时，通过实际编程练习，提高对 C++ 语言的运用能力，以及对算法性能的分析能力。 | | | | | |
| 实验原理 | Dijkstra 算法是一种用于寻找图中单源最短路径的算法。它的核心思想是维护一个顶点集合，该集合包含从源点出发已知最短路径的顶点，以及一个优先队列，用于存储待访问的顶点和它们当前的最短路径估计值。算法重复选择距离源点最近的未访问顶点，并更新其相邻顶点的最短路径估计值。 | | | | | |
| 实验步骤 | **1.使用邻接矩阵来储存图**  vector<vector<int>> e(N,vector<int>(N,INT\_MAX));  **①确定输入输出和约束条件:**  输入为顶点数n，边数m，各边的起点u，终点v和权值w，  输出为选定节点st到其他各顶点的最短距离dis，以及路径，  约束条件为最短距离  **②初始化**  初始化图，dis数组，是否访问过bool数组，前序节点pre数组  **③更新dis**  依次选择未访问过的最小距离的节点mini，再遍历所有节点，确定选定节点st经最小距离节点mini到该节点的距离是否缩短，刷新距离。  **④重复**  若仍存在未访问的节点，重复③  **2.使用邻接表来储存图**  struct Edge{  ll v;  ll w;  Edge(int v,int w){  this->v=v;  this->w=w;  }  };  vector<Edge> e[N];  **①确定输入输出和约束条件:**  输入为顶点数n，边数m，各边的起点u，终点v和权值w，  输出为选定节点st到其他各顶点的最短距离dis，以及路径，  约束条件为最短距离  **②初始化**  初始化图，dis数组，是否访问过bool数组，前序节点pre数组，node节点最小堆(用于选择最小距离的节点)  **③更新dis**  依次从最小堆中取出堆顶元素t，再遍历所有节点，确定选定节点st经最小距离节点t到该节点的距离是否缩短，刷新距离。  **④重复**  若最小堆中仍存在未访问的节点，重复③ | | | | | |
| 关键代码 | **1.使用邻接矩阵来储存图**   1. **for**(**int** i=1;i<=n;i++)dis[i]=INT\_MAX; 2. dis[st]=0; 3. book[st]=1; 4. **for**(**int** j=1;j<=n;j++)**if**(e[st][j]>=0)dis[j]=e[st][j]; 5. ll min1; 6. **int** mini; 7. **for**(**int** i=1;i<=n-1;i++){ 8. min1=INT\_MAX; 9. **for**(**int** j=1;j<=n;j++){ 10. **if**(dis[j]<min1&&book[j]==0){ 11. min1=dis[j]; 12. mini=j; 13. } 14. } 15. book[mini]=1; 16. **for**(**int** j=1;j<=n;j++){ 17. **if**(dis[mini]+e[mini][j]<dis[j]&&e[mini][j]<INT\_MAX){ 18. dis[j]=dis[mini]+e[mini][j]; 19. pre[j]=mini; 20. } 21. } 22. }   **2.使用邻接表来储存图**   1. **struct** cmp { 2. **bool** operator()(**const** node& a, **const** node& b) { 3. **return** a.dis > b.dis; 4. } 5. }; 6. priority\_queue<node, vector<node>, cmp> q; 7. **bool** book[N]; 8. ll n,m,st;//顶点数，边数，起始节点 9. ll dis[N];//最短距离 10. ll pre[N];//该顶点的前序顶点 11. **inline** **void** dijkstra(){ 12. fill(dis, dis + N, INT\_MAX); 13. dis[st]=0; 14. q.push(node(0,st)); 15. **while**(!q.empty()){ 16. node t=q.top(); 17. q.pop(); 18. ll d=t.dis; 19. **int** p=t.pos; 20. **if**(book[p])**continue**; 21. book[p]=1; 22. **int** v; 23. ll w; 24. **for**(auto edge:e[p]){ 25. v=edge.v; 26. w=edge.w; 27. **if**(dis[p]+w<dis[v]){ 28. dis[v]=dis[p]+w; 29. q.push(node(dis[v],v)); 30. pre[v]=p; 31. } 32. } 33. } 34. } | | | | | |
| 测试结果 | **(1)正确性：**    在洛谷上进行测试，在时间和空间上均满足题目要求，且数据量达到要求，保证了正确性。  **(2)复杂度：**  时间复杂度：  Dijkstra 算法的时间复杂度主要取决于两个因素：优先队列的操作次数和图中边的数量。  1.**优先队列操作**：在 Dijkstra 算法中，每个顶点都会被加入到优先队列中一次，并且每次从优先队列中取出一个顶点时，都需要进行对数级别的操作,对于V个顶点，优先队列的操作次数为 V，每次操作的时间复杂度为。  2.**边的处理**：对于每条边，进行两次操作：一次是从优先队列中取出一个顶点时，检查与该顶点相邻的边；另一次是更新邻接顶点的距离时，如果距离更短，则将邻接顶点加入到优先队列中。因此，对于E条边，边的处理次数为E。  总结来说，Dijkstra算法的时间复杂度为  空间复杂度：  Dijkstra 算法的空间复杂度主要取决于图中顶点和边的数量。  1.**顶点数组**：需要一个数组dis来存储每个顶点到源点的最短距离，这个数组的大小与图中顶点的数量成正比，即。  **2.边的存储**：使用邻接表来存储图中的边，每个顶点都有一个与之对应的边列表,因此，边的存储空间与图中边的数量成正比，即,如果使用邻接矩阵会造成稀疏矩阵，空间复杂度为。  3.**优先队列**：优先队列中最多会包含V个顶点，因此其空间复杂度为O(V)。  综合以上三点，Dijkstra 算法的空间复杂度为。  **(3)构造了不同规模数据集并进行图像化演示：**  生成了不同规模和大小的数据        **分析与验证：**  **1. 时间复杂度验证**  Dijkstra 算法的理论时间复杂度为 ，实际运行时间主要受边数 E 和节点数 V 影响。从运行时间增长规律来看，以下现象支持时间复杂度的增长模式：  **小规模：**节点数 N=10，边数 M=20，运行时间非常短，接近 0 秒，符合理论复杂度中的低消耗。  **中规模：**节点数 N=1000，边数 M=5000,时间增长到约 0.001 秒。时间增长与的变化一致。  **大规模：**节点数 N=1,000,000，边数 M=5,000,000。运行时间为 0.165 秒，时间增加符合预期。对比中小规模，运行时间呈现次线性增长。  **2. 时间复杂度分析**  我们进一步分析理论复杂度与实际运行时间的关系：  运行时间比例大致为 1 : 300 : 300000，对应的实际运行时间从微秒级增长到数百毫秒级，与理论复杂度一致。  **3. 空间复杂度分析**  使用邻接表存储边：空间复杂度为。对于大规模图，节点数和边数大，内存消耗主要由边信息占据。  对大规模数据（1000000节点，5000000边）:理论内存消耗约为 8×(N+M)≈48MB，符合内存负载能力，未出现内存瓶颈。 | | | | | |
| 实验心得 | 1.通过这次实验，回顾了dijkstra求单源最短路径的方法，感受到了贪心算法性能的高效。  2.掌握了可以通过使用优先队列来提高查找最小节点，从而优化算法的性能。3.同时也意识到了dijkstra算法设计中贪心策略的局限性，比如它不适用于包含负权边的图。 | | | | | |
| 实验得分 |  | 助教签名 | |  | | |

**附录：完整代码**

1. **使用邻接矩阵来储存图**

|  |
| --- |
| #include<iostream>  #include<vector>  #include<climits>  using namespace std;  #define ll long long  const int N=2e3+1;  bool book[N];  ll dis[N];  ll pre[N];  vector<vector<int>> e(N,vector<int>(N,INT\_MAX));  ll n,m;//顶点个数,边条数  ll st; //起始点  int main(){  cin>>n>>m;  cin>>st;  int u,v,w;  for(int i=1;i<=n;i++)  for(int j=1;j<=n;j++)if(i==j)e[i][j]=0;  for(int i=1;i<=m;i++){  cin>>u>>v>>w;  e[u][v]=w;  }  for(int i=1;i<=n;i++)dis[i]=INT\_MAX;  dis[st]=0;  book[st]=1;  for(int j=1;j<=n;j++)if(e[st][j]>=0)dis[j]=e[st][j];  ll min1;  int mini;  for(int i=1;i<=n-1;i++){  min1=INT\_MAX;  for(int j=1;j<=n;j++){  if(dis[j]<min1&&book[j]==0){  min1=dis[j];  mini=j;  }  }  book[mini]=1;  for(int j=1;j<=n;j++){  if(dis[mini]+e[mini][j]<dis[j]&&e[mini][j]<INT\_MAX){  dis[j]=dis[mini]+e[mini][j];  pre[j]=mini;  }  }  }  for(int i=1;i<=n;i++)cout<<dis[i]<<" ";  cout<<"输入查询的节点路径：";  int x=0;  cin>>x;  for(;x!=0;){  if(x==st)cout<<x;  cout<<x<<"<-";  x=pre[x];  }  } |

**2.使用邻接表来储存图**

|  |
| --- |
| #include<iostream>  #include<climits>  #include<vector>  #include<queue>  using namespace std;  #define ll long long  const int N=1e6+10;  const int M=5e6+10;  struct Edge{  ll v;  ll w;  Edge(int v,int w){  this->v=v;  this->w=w;  }  };  vector<Edge> e[N];  struct node{  int dis;  int pos;  node(int dis,int pos){  this->dis=dis;  this->pos=pos;  }  bool operator()(const node &a,const node &b)const{  return a.dis>b.dis;  }  };  struct cmp {  bool operator()(const node& a, const node& b) {  return a.dis > b.dis;  }  };  priority\_queue<node, vector<node>, cmp> q;  bool book[N];  ll n,m,st;//顶点数，边数，起始节点  ll dis[N];//最短距离  ll pre[N];//该顶点的前序顶点  inline void dijkstra(){  fill(dis, dis + N, INT\_MAX);  dis[st]=0;  q.push(node(0,st));  while(!q.empty()){  node t=q.top();  q.pop();  ll d=t.dis;  int p=t.pos;  if(book[p])continue;  book[p]=1;  int v;  ll w;  for(auto edge:e[p]){  v=edge.v;  w=edge.w;  if(dis[p]+w<dis[v]){  dis[v]=dis[p]+w;  q.push(node(dis[v],v));  pre[v]=p;  }  }  }  }  int main(){  cin>>n>>m>>st;  ll u,v,w;  for(int i=1;i<=m;i++){  cin>>u>>v>>w;  e[u].push\_back(Edge(v,w));  }  dijkstra();  for(int i=1;i<=n;i++){  if(dis[i]==INT\_MAX)cout<<INT\_MAX<<" ";  else cout<<dis[i]<<" ";  }  cout<<"输入查询的节点路径：";  int x=0;  cin>>x;  for(;x!=0;){  if(x==st){cout<<x;break;  }  cout<<x<<"<-";  x=pre[x];  }    } |

3.数据测试

|  |
| --- |
| #include <iostream>  #include <fstream>  #include <vector>  #include <queue>  #include <climits>  #include <chrono>  using namespace std;  #define ll long long  const int N = 1e6 + 10;  struct Edge {  ll v, w;  Edge(ll v, ll w) : v(v), w(w) {}  };  vector<Edge> e[N];  struct Node {  ll dis, pos;  Node(ll dis, ll pos) : dis(dis), pos(pos) {}  bool operator>(const Node &a) const { return dis > a.dis; }  };  priority\_queue<Node, vector<Node>, greater<Node>> q;  bool book[N];  ll n, m, st; // 顶点数，边数，起始节点  ll dis[N]; // 最短距离  ll pre[N]; // 前序顶点  // 生成测试数据  void generate\_test\_data(const string &filename, ll num\_nodes, ll num\_edges, ll max\_weight) {  ofstream out(filename);  if (!out) {  cerr << "Error: Unable to open file " << filename << endl;  return;  }  out << num\_nodes << " " << num\_edges << " " << 1 << "\n"; // 顶点数，边数，起始节点固定为 1  for (ll i = 0; i < num\_edges; i++) {  ll u = rand() % num\_nodes + 1; // 随机起点  ll v = rand() % num\_nodes + 1; // 随机终点  while (u == v) v = rand() % num\_nodes + 1; // 避免自环  ll w = rand() % max\_weight + 1; // 随机权重  out << u << " " << v << " " << w << "\n";  }  out.close();  cout << "Generated data for " << filename << " with " << num\_nodes << " nodes and " << num\_edges << " edges.\n";  }  // Dijkstra 算法  void dijkstra() {  fill(dis, dis + n + 1, LLONG\_MAX); // 初始化为无穷大  fill(book, book + n + 1, false); // 重置访问标记  dis[st] = 0;  q.push(Node(0, st));  while (!q.empty()) {  Node t = q.top();  q.pop();  ll d = t.dis, p = t.pos;  if (book[p]) continue;  book[p] = true;  for (auto edge : e[p]) {  ll v = edge.v, w = edge.w;  if (dis[p] + w < dis[v]) {  dis[v] = dis[p] + w;  pre[v] = p;  q.push(Node(dis[v], v));  }  }  }  }  // 测试数据  void test\_data(const string &filename) {  ifstream in(filename);  if (!in) {  cerr << "Error: Unable to open file " << filename << endl;  return;  }  // 加载数据  in >> n >> m >> st;  for (ll i = 1; i <= n; i++) e[i].clear(); // 清空邻接表  for (ll i = 0; i < m; i++) {  ll u, v, w;  in >> u >> v >> w;  e[u].push\_back(Edge(v, w));  }  in.close();  // 测试算法并计时  auto start\_time = chrono::high\_resolution\_clock::now();  dijkstra();  auto end\_time = chrono::high\_resolution\_clock::now();  // 输出结果  cout << "Test for " << filename << ":\n";  cout << "Actual edges: " << m << "\n";  cout << "Time taken: "  << chrono::duration<double>(end\_time - start\_time).count()  << " seconds\n\n";  }  int main() {  srand(time(0)); // 设置随机种子  // 生成测试数据  generate\_test\_data("dijkstra\_small.txt", 10, 20, 100); // 小规模  generate\_test\_data("dijkstra\_medium.txt", 1000, 5000, 1000); // 中规模  generate\_test\_data("dijkstra\_large.txt", 1000000, 5000000, 10000); // 大规模  cout << "\nData generation complete.\n\n";  // 测试小规模数据  test\_data("dijkstra\_small.txt");  // 测试中规模数据  test\_data("dijkstra\_medium.txt");  // 测试大规模数据  test\_data("dijkstra\_large.txt");  return 0;  } |