**算法分析与设计实验报告**

**第 4 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 | 分支限界法求解 TSP 问题 | | | | | |
| 实验目的 | 通过本实验，深入理解和掌握分支限界法的设计思想，特别是如何通过分支限界法解决旅行商问题（TSP）。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 | 旅行商问题（TSP）是一个经典的NP-hard问题，目标是寻找一条最短的路径，该路径恰好访问每个城市一次并返回出发城市。分支限界法是一种有效的解决方法，通过队列从而广度地探索所有可能的解决方案空间，并剪枝不可能的解。 | | | | | |
| 实验步骤 | **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，  输出为遍历所有节点的最小距离dis，以及路径，  约束条件为最短距离  **②初始化并计算minout**  首先，初始化城市数量n和邻接矩阵e，该矩阵存储城市间的距离。同时，初始化一个数组minarr来存储每个城市的最小出边距离，并将其所有元素设置为INT\_MAX，通过循环读取用户输入的城市间距离，填充邻接矩阵e，对于每个城市，找到其最小的出边距离，并更新minarr数组，如果发现有城市没有出边（即minarr[i]仍为INT\_MAX），则输出NoEdge并结束程序。计算所有城市最小出边距离的总和minout，这将用作初始下界  **③分支限界搜索**  当优先队列不为空时，执行以下操作：  取出队列顶部节点。  如果当前节点的s值大于等于n - 1，表示已找到一个完整的路径，更新最佳费用bestc并记录路径bc。  如果当前节点的s值为n - 2，检查是否可以通过添加最后两个城市和返回起始城市的费用来更新最佳费用。  如果当前节点的s值小于n - 2，则探索所有可能的下一个城市，更新下界lcost和当前费用cw，并生成新的节点加入优先队列。  **④重复**  若优先队列不为空且未取出到叶子节点，重复③  **⑤输出**  搜索完成后，输出找到的最佳费用bestc和最佳路径bc | | | | | |
| 关键代码 | 1.使用邻接矩阵   1. **for** (**int** i = 0; i < n; i++) { // 计算minarr数组和minout值 2. **if** (minarr[i] == INT\_MAX) { 3. cout << "NoEdge" << endl; 4. **return** 0; 5. } 6. minout += minarr[i]; 7. } 8. **int** t[21]; 9. **for** (**int** i = 0; i < n; i++) t[i] = i; 10. q.push(node(minout, 0, 0, t)); 11. **while** (!q.empty()) { 12. auto no = q.top(); 13. q.pop(); 14. **int** lcost = no.lcost; // 下界 15. **int** cw = no.cw; // 当前费用 16. **int** s = no.s; // 根节点到当前路径，用于标记当前节点的位置 17. **int** x[21]; 18. **for** (**int** i = 0; i < 21; ++i) { 19. x[i] = no.x[i]; 20. } 21. **if** (s >= n - 1) { 22. bestc = cw; 23. **for** (**int** i = 0; i < n; ++i) { 24. bc[i] = x[i]; 25. } 26. **break**; 27. } 28. **if** (s == n - 2) { 29. **if** (e[x[n - 2]][x[n - 1]] != INT\_MAX && e[x[n - 1]][0] != INT\_MAX && 30. cw + e[x[n - 2]][x[n - 1]] + e[x[n - 1]][0] < bestc) { 31. ll nlcost = cw + e[x[n - 2]][x[n - 1]] + e[x[n - 1]][0]; 32. q.push(node(nlcost, nlcost, s + 1, x)); 33. } **else** **continue**; 34. } **else** { 35. **for** (**int** i = s + 1; i < n; i++) { 36. **if** (e[x[s]][x[i]] != INT\_MAX) { 37. **int** ncw = cw + e[x[s]][x[i]]; 38. **int** nlcost = lcost - minarr[x[s]] + e[x[s]][x[i]]; 39. **if** (nlcost < bestc) { 40. **int** tmp[21]; 41. **for** (**int** j = 0; j < 21; ++j) { 42. tmp[j] = x[j]; 43. } 44. swap(tmp[s + 1], tmp[i]); 45. q.push(node(nlcost, ncw, s + 1, tmp)); 46. } 47. } 48. } 49. } 50. } 51. cout << bestc << endl;   2.使用邻接表   1. **for**(**int** i=1;i<=n;i++){//计算minarr数组和minout值 2. **if**(minarr[i]==INT\_MAX){ 3. cout<<"NoEdge"<<endl; 4. **return** 0; 5. } 6. minout+=minarr[i]; 7. } 8. vector<**int**> t(n); 9. **for**(**int** i=0;i<n;i++)t[i]=i+1; 10. q.push(node(minout,0,0,t)); 11. **while**(!q.empty()){ 12. auto no=q.top(); 13. q.pop(); 14. ll lcost=no.lcost;//下界 15. ll cw=no.cw;//当前费用 16. ll s=no.s;//根节点到当前路径，用于标记当前节点的位置 17. auto x=no.x;//记录路径 18. **if**(s==n-1){ 19. bestc=cw+e[x[n-2]][x[n-1]]+e[x[n-1]][1]; 20. bc=x; 21. **break**; 22. } 23. **if**(s==n-2){ 24. **if**(e[x[n-2]][x[n-1]]!=INT\_MAX&&e[x[n-1]][1]!=INT\_MAX&& 25. cw+e[x[n-2]][x[n-1]]+e[x[n-1]][1]<bestc){ 26. ll ncw=cw+e[x[n-2]][x[n-1]]+e[x[n-1]][1]; 27. q.push(node(ncw,ncw,s+1,x)); 28. }**else** **continue**; 29. }**else**{ 30. **for**(**int** i=s+1;i<=n;i++){ 31. **if**(e[x[s]][x[i]]!=INT\_MAX){ 32. ll ncw=cw+e[x[s]][x[i]]; 33. ll nlcost=lcost-minarr[x[s]]+e[x[s]][x[i]]; 34. **if**(nlcost<bestc){ 35. auto tmp=x; 36. swap(tmp[s+1],tmp[i]); 37. q.push(node(nlcost,ncw,s+1,tmp)); 38. } 39. } 40. } 41. } 42. } | | | | | |
| 测试结果 | **(1)正确性：**    在洛谷上进行测试，满足正确性，但是该方法对数据规模有限制。  **(2)复杂度：**  时间复杂度：  **1.初始化和读取**：代码首先初始化邻接矩阵e和最小边数组minarr，这部分的时间复杂度为。  **2.优先队列操作**：在优先队列中，在最坏情况下，每个节点可能生成 n-1 个子节点（因为除了当前节点外，其他所有节点都可能是下一个节点）。因此，优先队列的大小可能达到，但实际上由于分支限界法的使用，很多节点会被剪枝，实际大小远小于。  **3.优先级下界计算**：计算下界lcost的时间复杂度为，因为需要从当前节点到下一个节点的最小费用。  总而言之，总的时间复杂度在最坏情况下接近 O(n!)，但是由于分支限界法的有效剪枝，实际的时间复杂度通常会小得多。  空间复杂度：  **1.邻接矩阵**：邻接矩阵e需要的空间来存储 n×n 的矩阵。  **2.优先队列**：优先队列中最多可能包含个节点，因此其空间复杂度为 。  **3.路径数组**：每个节点都需要一个大小为n的数组来存储路径，因此空间复杂度为 O(n)。  总而言之，总的空间复杂度主要由邻接矩阵和优先队列决定，为 O(n!)。在实际应用中，由于优先队列的大小通常远小于 O(n!)，空间复杂度通常小于理论最大值。  (3)构造了不同规模数据集并进行图像化演示：  构造不同规模和大小的数据集        可以看到，采用该方法对空间上要求大，最坏情况下空间复杂度为O(n!)，时间复杂度上符合最坏情况下是O(n!)，但是性能一般会好得多。 | | | | | |
| 实验心得 | 1.通过这次实验，深入学习了TSP问题的理论基础，理解了其作为NP-hard问题的核心难点，以及优先队列式分支限界法与启发式算法具有类似之处。  2.此外，还区分了邻接矩阵和邻接表，邻接矩阵解决问题更加直观，但是处理不了大规模的稀疏矩阵，会占用大量空间，类似于洛谷上的做法，而邻接表可以解决这个问题，因此最后补充了邻接表的实现，但是只是针对边来遍历，不能通过两个顶点来定位。  3.通过对算法性能的分析，我更加直观地理解了优先队列式分支限界法的时间复杂度和空间复杂度。我学会了如何通过理论分析和实际测试来评估算法的性能，并理解了算法参数对性能的影响。 | | | | | |
| 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<bits/stdc++.h>  using namespace std;  #define ll long long  const int N=1e4+10;  vector<vector<int>> e(N,vector<int>(N,INT\_MAX));  struct node{  ll lcost;//下界  ll cw;//当前费用  ll s;//根节点到当前路径，用于标记当前节点的位置  vector<int> x;//记录路径  node(ll lcost,ll cw,ll s,vector<int> x){  this->lcost=lcost;  this->cw=cw;  this->s=s;  this->x=x;  }  };  struct nodecmp{  bool operator()(const node& a,const node& b){  return a.lcost > b.lcost;  }  };  ll n,m;//顶点数和边数  ll u,v,w;  ll minout;  ll bestc=INT\_MAX;//最小费用  vector<int> bc;  priority\_queue<node,vector<node>,nodecmp> q;  int main(){  cin>>n>>m;  vector<ll> minarr(n+1,INT\_MAX);  for(int i=0;i<m;i++){  cin>>u>>v>>w;  e[u][v]=w;  e[v][u]=w;  if(w<minarr[u])minarr[u]=w;  if(w<minarr[v])minarr[v]=w;  }  for(int i=1;i<=n;i++){//计算minarr数组和minout值  if(minarr[i]==INT\_MAX){  cout<<"NoEdge"<<endl;  return 0;  }  minout+=minarr[i];  }  vector<int> t(n);  for(int i=0;i<n;i++)t[i]=i+1;  q.push(node(minout,0,0,t));  while(!q.empty()){  auto no=q.top();  q.pop();  ll lcost=no.lcost;//下界  ll cw=no.cw;//当前费用  ll s=no.s;//根节点到当前路径，用于标记当前节点的位置  auto x=no.x;//记录路径  if(s==n-1){  bestc=cw+e[x[n-2]][x[n-1]]+e[x[n-1]][1];  bc=x;  break;  }  if(s==n-2){  if(e[x[n-2]][x[n-1]]!=INT\_MAX&&e[x[n-1]][1]!=INT\_MAX&&  cw+e[x[n-2]][x[n-1]]+e[x[n-1]][1]<bestc){  ll ncw=cw+e[x[n-2]][x[n-1]]+e[x[n-1]][1];  q.push(node(ncw,ncw,s+1,x));  }else continue;  }else{  for(int i=s+1;i<=n;i++){  if(e[x[s]][x[i]]!=INT\_MAX){  ll ncw=cw+e[x[s]][x[i]];  ll nlcost=lcost-minarr[x[s]]+e[x[s]][x[i]];  if(nlcost<bestc){  auto tmp=x;  swap(tmp[s+1],tmp[i]);  q.push(node(nlcost,ncw,s+1,tmp));  }  }  }  }  }  cout<<bestc<<endl;  for(int i=1;i<=n;i++){  cout<<bc[i]<<" ";  }  } |

2.使用邻接表存储图

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  #define ll long long  const int N = 1e4 + 10;  struct node {  ll lcost; // 下界  ll cw; // 当前费用  ll s; // 根节点到当前路径，用于标记当前节点的位置  vector<int> x; // 记录路径  node(ll lcost, ll cw, ll s, vector<int> x) : lcost(lcost), cw(cw), s(s), x(x) {}  };  struct nodecmp {  bool operator()(const node& a, const node& b) {  return a.lcost > b.lcost; // 最小堆  }  };  ll n, m; // 顶点数和边数  ll u, v, w;  ll minout = 0; // 所有点的最小出边权重和  ll bestc = INT\_MAX; // 最小费用  vector<int> bc; // 最优路径  priority\_queue<node, vector<node>, nodecmp> q;  int main() {  cin >> n >> m;  vector<vector<pair<int, int>>> adj(n + 1); // 邻接表  vector<ll> minarr(n + 1, INT\_MAX); // 每个节点的最小出边权重  // 输入边  for (int i = 0; i < m; i++) {  cin >> u >> v >> w;  adj[u].emplace\_back(v, w);  adj[v].emplace\_back(u, w);  minarr[u] = min(minarr[u], (ll)w);  minarr[v] = min(minarr[v], (ll)w);  }  // 检查是否存在孤立点  for (int i = 1; i <= n; i++) {  if (minarr[i] == INT\_MAX) {  cout << "NoEdge" << endl;  return 0;  }  minout += minarr[i];  }  vector<int> t(n);  iota(t.begin(), t.end(), 1); // 初始化路径 1, 2, ..., n  q.push(node(minout, 0, 0, t));  while (!q.empty()) {  auto no = q.top();  q.pop();  ll lcost = no.lcost; // 下界  ll cw = no.cw; // 当前费用  ll s = no.s; // 当前路径位置  auto x = no.x; // 路径  if (s == n - 1) {  // 完成路径，计算总费用  for (auto& p : adj[x[n - 1]]) {  if (p.first == x[0]) {  bestc = min(bestc, cw + p.second);  bc = x;  break;  }  }  break;  }  if (s == n - 2) {  // 剩余最后两个点，直接连接起点  for (auto& p1 : adj[x[n - 2]]) {  if (p1.first == x[n - 1]) {  for (auto& p2 : adj[x[n - 1]]) {  if (p2.first == x[0] && cw + p1.second + p2.second < bestc) {  ll ncw = cw + p1.second + p2.second;  q.push(node(ncw, ncw, s + 1, x));  }  }  }  }  } else {  // 扩展路径  for (int i = s + 1; i < n; i++) {  for (auto& p : adj[x[s]]) {  if (p.first == x[i]) {  ll ncw = cw + p.second; // 新费用  ll nlcost = lcost - minarr[x[s]] + p.second; // 新下界  if (nlcost < bestc) {  auto tmp = x;  swap(tmp[s + 1], tmp[i]);  q.push(node(nlcost, ncw, s + 1, tmp));  }  }  }  }  }  }  // 输出结果  cout << bestc << endl;  for (int i = 0; i < n; i++) {  cout << bc[i] << " ";  }  cout << endl;  return 0;  } |